

ЛІСТИНГ КОДУ

Код наведений в додатку і в роботі можуть мати розбіжності, так як робота коментувалась в процесі її розробки, а після вона могла видозмінюватись, і не всі зміни в майбутньому коментувались.

Посилання на github із кодом застосунку, та з допоміжним кодом для ілюстрацій роботи інших алгоритмів(папка 'midpoint') можна знайти на репозиторії за посиланням:

<https://github.com/Artem1k/Landscapes>

Головний файл 'tk_gui.py':

```
from terrain import *
from visualize_3d_terrain import *
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk

MAX_VAL = 8

def quit_me():
    export_plot(out_terrain.updated_terrain)
    window.quit()
    window.destroy()

def new_terrain():
    global out_terrain
    out_terrain = Terrain(MAX_VAL)
    out_terrain.change_size(size)
    visualize_first_terrain_3d(out_terrain, var=button_var.get(), canvas=canvas, ani_var=button_animation_var.get())

def toggle_button(var, but, txt):
    if var.get():
        var.set(False)
        but.config(text=f"{txt}_OFF")
        update_terrain(False)
    else:
        var.set(True)
        but.config(text=f"{txt}_ON")
        update_terrain(True)

def update_terrain_smoothness(var):
    # Generate the terrain with the updated parameters
    out_terrain.set_smoothness(smoothness_slider.get())
    update_terrain(True)

def update_terrain(var):
    var = button_var.get()
    ani_var = button_animation_var.get()
    new_size = int(size_slider.get())
    out_terrain.change_size(size=new_size)
    visualize_terrain_3d(out_terrain.updated_terrain, var=var, canvas=canvas, ani_var=ani_var)

def update_size_label(value):
```

```

size = 2 ** int(float(value))
size_label_value.config(text=f'{str(int(float(value)))} pow of 2 = ({size}+1)*({size}+1) = {(size + 1) ** 2}')

def update_smoothness_label(value):
    smoothness_label_value.config(text=str(round(float(value), 2)))

# Define the parameters and their initial values
size = 5
smoothness = 0.5
out_terrain = Terrain(MAX_VAL)
out_terrain.change_size(size)

# Create the tkinter window
window = tk.Tk()
window.title("Terrain Generator")
window.protocol("WM_DELETE_WINDOW", quit_me)
window.geometry("1000x700")

# Set grid columns weights
for i in range(4):
    window.grid_columnconfigure(i, weight=1)
window.grid_rowconfigure(0, weight=1)
window.grid_rowconfigure(1, weight=1)
window.grid_rowconfigure(2, weight=10)

# Create a slider for size
size_label = ttk.Label(window, text="Size")
size_label.grid(row=0, column=0)
size_slider = ttk.Scale(window, from_=0, to=MAX_VAL, value=size, orient=tk.HORIZONTAL,
                        command=lambda value: [update_terrain(value), update_size_label(value)])
size_slider.grid(row=0, column=1, sticky="ew")
# Create a label to display the current value of the size slider
size_label_value = ttk.Label(window)
size_label_value.grid(row=0, column=2)
# Initial update of the size label value
update_size_label(size_slider.get())

# Create a slider for smoothness
smoothness_label = ttk.Label(window, text="Smoothness")
smoothness_label.grid(row=1, column=0)
smoothness_slider = ttk.Scale(window, from_=0, to=2, value=smoothness, orient=tk.HORIZONTAL,
                              command=lambda value: [update_terrain_smoothness(value), update_smoothness_label(value)])
smoothness_slider.grid(row=1, column=1, sticky="ew")
# Create a label to display the current value of the smoothness slider
smoothness_label_value = ttk.Label(window)
smoothness_label_value.grid(row=1, column=2)
# Initial update of the smoothness label value
update_smoothness_label(smoothness_slider.get())

# Create a button to generate a new terrain
new_button = tk.Button(window, text="New Terrain", command=new_terrain)
new_button.grid(row=0, column=3, sticky="nsew")

# Button for setting axis ON/OFF
button_var = tk.BooleanVar()
button = tk.Button(window, text="AXIS_OFF", command=lambda: toggle_button(button_var, button, 'AXIS'))
button.grid(row=1, column=3, sticky="nsew")

# Create a blank figure and canvas for Matplotlib plot
canvas = FigureCanvasTkAgg(figure, master=window)
canvas.get_tk_widget().grid(row=2, columnspan=4, sticky="nsew")

toolbarFrame = tk.Frame(master=window)

```

```

toolbarFrame.grid(row=3, columnspan=3, sticky="nsew")
toolbar = NavigationToolbar2Tk(canvas, toolbarFrame)
toolbar.update()
toolbar.pack(side=tk.LEFT) # (anchor='w') as an alternative

# Button for setting axis ON/OFF
button_animation_var = tk.BooleanVar(value=True)
button_animation = tk.Button(window, text="ANI_ON",
                             command=lambda: toggle_button(button_animation_var, button_animation, 'ANI'))
button_animation.grid(row=3, column=3, sticky="nsew")

# Visualize the initial terrain
visualize_first_terrain_3d(terrain=out_terrain, var=False, canvas=canvas, ani_var=True)

# Start the tkinter event loop
window.mainloop()

```

Файл для генерації ландшафту, зберігання та маніпуляції ‘**terrain.py**’:

```

from numpy import random
import copy

class GenerateTerrain:
    def __init__(self, sizer: int):
        self.sizer = sizer
        self.size = 2 ** sizer
        self.length = self.size + 1
        self.mat = [[0] * self.length for _ in range(self.length)]
        self.random_values = [[None] * self.length for _ in range(self.length)]
        self.smoothness = 0.5
        self.get_h = self.get_h
        self.frames = None

    def iterate(self):
        """Performs the iteration of the Diamond-Square algorithm to generate the terrain."""
        self.frames = [[0] * self.length for _ in range(self.length)]
        for counter in range(self.sizer):
            num_segs = 1 << counter
            span = self.size // num_segs
            half = span // 2
            self.diamond(counter + 1, span, half)
            self.square(counter + 1, span, half)
            self.frames.append(copy.deepcopy(self.mat))

    def diamond(self, depth, span, half):
        """Performs the diamond step of the Diamond-Square algorithm for a given depth and span."""
        for y in range(0, self.size, span):
            for x in range(0, self.size, span):
                # e = avg(a, b, c, d)
                # (x,y) Our sub-square
                # \
                # a--ab--b
                # | \| / |
                # ad---e---bc
                # | / | \ |
                # d---cd---c
                # \_____/
                #     v
                #     span
                ne = [x + half, y + half] # center of current square
                na = [x, y]
                nb = [x + span, y]
                nc = [x + span, y + span]
                nd = [x, y + span]

```

```

heights = [self.mat[n[1]][n[0]] for n in [na, nb, nc, nd]]
avg = self.average(heights)
offset = self.get_h(depth, ne)

```

```

self.mat[ne[1]][ne[0]] = avg + offset

```

```

def square(self, depth, span, half):

```

```

    """Performs the square step of the Diamond-Square algorithm for a given depth and span."""

```

```

    for y in range(0, self.size, span):

```

```

        for x in range(0, self.size, span):

```

```

            # If second iteration, then

```

```

            # bc = avg(b, g, c, e)

```

```

            # ab = avg(a, e, d, g)

```

```

            # h = ad

```

```

            # (x,y)

```

```

            # \

```

```

            # a---ab---b---e---f

```

```

            # | \| / \| / \| /

```

```

            # ad---e---bc---g---h

```

```

            # | / \| / \| / \| /

```

```

            # d---cd---c---i---j

```

```

            # | \| / \| / \| /

```

```

            # k---l---m---n---o

```

```

            # | / \| / \| / \| /

```

```

            # p---q---r---s---t

```

```

            # \_____ /

```

```

            #     v

```

```

            #     span

```

```

            ne = [x + half, y + half] # center of current square

```

```

            na = [x, y]

```

```

            nb = [x + span, y]

```

```

            nc = [x + span, y + span]

```

```

            nd = [x, y + span]

```

```

            nab = [x + half, y]

```

```

            nbc = [x + span, y + half]

```

```

            ncd = [x + half, y + span]

```

```

            nad = [x, y + half]

```

```

            nu = [x + half, y - half]

```

```

            if nu[1] < 0:

```

```

                nu[1] = self.size - half

```

```

            nl = [x - half, y + half]

```

```

            if nl[0] < 0:

```

```

                nl[0] = self.size - half

```

```

            nr = [x + half * 3, y + half]

```

```

            if nr[0] > self.size:

```

```

                nr[0] = half

```

```

            ndo = [x + half, y + half * 3]

```

```

            if ndo[1] > self.size:

```

```

                ndo[1] = half

```

```

            self.square_helper(depth, na, nu, nb, ne, nab)

```

```

            self.square_helper(depth, nb, nr, nc, ne, nbc)

```

```

            self.square_helper(depth, nc, ndo, nd, ne, ncd)

```

```

            self.square_helper(depth, na, ne, nd, nl, nad)

```

```

    for y in range(0, self.size, span):

```

```

        self.mat[y][self.size] = self.mat[y][0]

```

```

    for x in range(0, self.size, span):

```

```

        self.mat[self.size][x] = self.mat[0][x]

```

```

def square_helper(self, depth, *args):
    """Helper function for the square step that calculates the average height and offset
    and applies this for last given node."""
    heights = [self.mat[n[1]][n[0]] for n in args[:-1]]
    avg = self.average(heights)
    offset = self.get_h(depth, args[-1])
    self.mat[args[-1][1]][args[-1][0]] = avg + offset

def get_h(self, depth, el):
    """Calculates the random height offset for a given element based on the current depth and smoothness."""
    h = self.h(depth, self.smoothness)
    rand = random.random()
    self.random_values[el[1]][el[0]] = rand # Ignore emphasis
    return (1 - 2 * rand) * h

def new_h(self, depth, el):
    """An alternative implementation of get_h that takes pre-computed random values. It is used to update"""
    h = self.h(depth, self.smoothness)
    rand = self.random_values[el[1]][el[0]]
    return (1 - 2 * rand) * h # Ignore emphasis

@staticmethod
def h(d, s):
    """Sets limit for selecting a random offset"""
    return pow(2, -2 * d * s) * 15

@staticmethod
def average(numbers):
    return sum(numbers) / len(numbers)

class Terrain:
    def __init__(self, size: int):
        """Initializes the Terrain object with a given size parameter.
        It creates an instance of GenerateTerrain and generates the initial terrain."""
        self.square_terrain = GenerateTerrain(size)
        self.square_terrain.iterate()
        self.square_terrain.get_h = self.square_terrain.new_h
        self.updated_terrain = None

    def change_size(self, sizer: int):
        """It is used to update"""
        terrain = copy.deepcopy(self.square_terrain.mat)
        counter = self.square_terrain.sizer - sizer
        span = pow(2, counter)
        half = span // 2
        for _ in range(counter):
            for i in range(0, self.square_terrain.size, span):
                for col in terrain:
                    col[i + half] = None # Ignore emphasis
                    terrain[i + half] = [None] * self.square_terrain.length
                span = half
                half //= 2
        self.updated_terrain = [list(filter(lambda el: el is not None, row)) for row in terrain if
                                any(el is not None for el in row)]

    def set_smoothness(self, sm):
        """It is used to update"""
        self.square_terrain.smoothness = sm
        self.square_terrain.iterate()

```

Файл для візуалізації ландшафту, його анімації та збереження **'visualize_3d_terrain'**:

```

import numpy as np
import matplotlib.pyplot as plt

```

```

from matplotlib.animation import FuncAnimation
from stl import mesh
from mpl_toolkits.mplot3d import Axes3D

def animation_rotate(ax, canvas):
    def animate(i):
        ax.view_init(elev=30, azim=i)
        return figure

    ani = FuncAnimation(figure, animate, frames=360, interval=10, blit=False)
    canvas.draw()

def visualize_terrain_3d(terrain: list, var, canvas, ani_var):
    length = len(terrain) - 1
    plt.clf()
    # Convert the terrain data to a NumPy array
    terrain_array = np.array(terrain)

    # Create a grid of coordinates
    x = np.arange(terrain_array.shape[1])
    y = np.arange(terrain_array.shape[0])
    x_grid, y_grid = np.meshgrid(x, y)

    # Create a 3D plot
    ax = figure.add_subplot(111, projection='3d')
    ax.set_position([0, 0, 1, 1]) # Set the position and size of the subplot within the figure
    ax.set_box_aspect([2, 2, 1])

    # Plot the terrain surface using matplotlib's plot_surface function
    ax.plot_surface(x_grid, y_grid, terrain_array, cmap='terrain')

    # Set labels and title, axis and limits
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Elevation')
    ax.set_axis_off()
    if var:
        ax.set_axis_on()
    # ax.set_zlim(-0.5, 0.5)
    ax.set_zlim(-7.5, 7.5)
    ax.set_xlim(0, length)
    ax.set_ylim(0, length)

    # ax.set_title('Fractal Terrain')
    if ani_var:
        animation_rotate(ax, canvas)

    canvas.draw()

def visualize_first_terrain_3d(terrain, var, canvas, ani_var):
    terrain_frames = terrain.square_terrain.frames

    def update_figure(i):
        if i == len(terrain_frames):
            if ani_var:
                visualize_terrain_3d(terrain_frames[-1], var, canvas, ani_var)
            else:
                terrain_list = terrain_frames[i]
                visualize_terrain_3d(terrain_list, var, canvas, False)

    animation = FuncAnimation(figure, update_figure, frames=len(terrain_frames) + 1, interval=200, repeat=False)

    canvas.draw()

```

```

def export_plot(terrain):
    terrain_array = np.array(terrain)

    # Create a grid of coordinates
    x = np.arange(terrain_array.shape[1])
    y = np.arange(terrain_array.shape[0])
    x_grid, y_grid = np.meshgrid(x, y)

    # Convert the data to mesh representation
    vertices = np.column_stack([x_grid.flatten(), y_grid.flatten(), terrain_array.flatten()])

    # Generate the triangular faces
    rows, cols = terrain_array.shape
    triangles = []
    for i in range(rows - 1):
        for j in range(cols - 1):
            v1 = i * cols + j
            v2 = i * cols + j + 1
            v3 = (i + 1) * cols + j
            v4 = (i + 1) * cols + j + 1
            triangles.append([v1, v2, v3])
            triangles.append([v2, v4, v3])
    triangles = np.array(triangles)

    # Create the mesh object
    mesh_data = mesh.Mesh(np.zeros(triangles.shape[0], dtype=mesh.Mesh.dtype))
    mesh_data.vectors = vertices[triangles]
    mesh_data.update_normals()

    # Save the mesh to an STL file
    mesh_data.save('terrain.stl')

figure = plt.figure()

```