import asyncio

import random

import sys

from .fun import all\_art

from .brush import BrushWipe

from .path import PathZigZag, PathRectEdge

from .renderer import AnimationRender

def main(

        frame\_interval\_s=0.003,

        clean\_after=None,

        min\_frame\_delay=0

):

    if clean\_after is None:

        clean\_after = 1

    r = AnimationRender()

    # You might customize the wipe brush

    bw = BrushWipe()

    common\_brush\_cfg = dict(

        size=bw.width,

        brush\_deformation\_factor=bw.deformation\_factor,

        max\_x=r.screen\_size.width,

        max\_y=r.screen\_size.height,

    )

    # You might create your own wipe path

    path\_points = PathZigZag(\*\*common\_brush\_cfg).get\_points\_list()

    path\_points += PathRectEdge(path\_points[-1], \*\*common\_brush\_cfg).get\_points\_list()

    for idx, pp in enumerate(path\_points):

        for bwp in bw.get\_points(\*pp.coord, pp.angle):

            r.schedule\_draw(

                frame=idx,

                p=bwp.coord,

                s=bwp.char,

                clean\_after=clean\_after

            )

    try:

        asyncio.new\_event\_loop().run\_until\_complete(

            r.render\_frames(frame\_interval\_s, min\_frame\_delay)

        )

    except KeyboardInterrupt:

        pass

    r.clear()

    r.move\_cursor\_home()

def cli(\*args):

    import argparse

    from argparse import RawTextHelpFormatter

    text = """

    wipe-clean - Clean your terminal in a ritual way

    """

    default = {

        'f': 0.003,

        'c': 1,

        'm': 0

    }

    parser = argparse.ArgumentParser(description=random.choice(all\_art) + text, formatter\_class=RawTextHelpFormatter)

    parser.add\_argument('-f', '--frame-interval', default=default['f'],

                        type=float, help='Frame interval (in second)')

    parser.add\_argument('-c', '--clean-after', default=default['c'],

                        type=float, help='Clean drawn after number of frame(s)')

    parser.add\_argument('-m', '--min-frame-delay', default=default['m'],

                        type=float, help='Minimum frame delay (in second). A delay will only be will be'

                                         ' scheduled when frame interval is larger than this value.'

                                         ' This may help solve the inaccurate sleep on Windows.')

    parsed\_args = parser.parse\_args(args)

    main(

        frame\_interval\_s=parsed\_args.frame\_interval,

        clean\_after=parsed\_args.clean\_after,

        min\_frame\_delay=parsed\_args.min\_frame\_delay

    )

def \_outer\_cli():

    sys.exit(

        cli(\*sys.argv[1:])

    )

if \_\_name\_\_ == '\_\_main\_\_':

    \_outer\_cli()

import math

from abc import ABC, abstractmethod

from typing import NamedTuple, Iterable, List, Tuple

from .screen import ScreenPoint

class PathPoint(NamedTuple):

    coord: ScreenPoint

    angle: float

class Path(ABC):

    @abstractmethod

    def get\_points(self) -> Iterable[PathPoint]:

        pass

    def get\_points\_list(self) -> List[PathPoint]:

        """Return a list of points"""

        return list(self.get\_points())

class PathCircle(Path):

    angle\_step = math.radians(5)

    def \_\_init\_\_(self, radius, start, end, deformation\_factor=3):

        self.radius = radius

        self.start = start

        self.end = end

        self.deformation\_factor = deformation\_factor

    def get\_points(self) -> Iterable[PathPoint]:

        points = []

        angle = self.start

        while angle < self.end:

            points.append(PathPoint(

                ScreenPoint(

                    x=math.cos(angle) \* self.radius \* self.deformation\_factor,

                    y=math.sin(angle) \* self.radius,

                ),

                angle

            ))

            angle += self.angle\_step

        return points

class PathLine(Path):

    def \_\_init\_\_(self, start: ScreenPoint, end: ScreenPoint):

        self.start = start

        self.end = end

    def get\_points(self) -> Iterable[PathPoint]:

        points = []

        y\_step = (self.end.y - self.start.y) / abs(self.end.x - self.start.x)

        x\_dir = 1 if self.end.x > self.start.x else -1

        angle = -math.atan((self.end.y - self.start.x) / self.start.y - self.end.y)

        for step in range(int(abs(self.end.x - self.start.x))):

            points.append(PathPoint(

                ScreenPoint(

                    x=step \* x\_dir + self.start.x,

                    y=step \* y\_step + self.start.y

                ),

                angle

            ))

        return points

class PathZigZag(Path):

    def \_\_init\_\_(self, size, brush\_deformation\_factor, max\_x, max\_y):

        self.size = size

        self.brush\_deformation\_factor = brush\_deformation\_factor

        self.max\_x = max\_x

        self.max\_y = max\_y

    def \_get\_key\_points(self) -> List[Tuple[ScreenPoint, ScreenPoint]]:

        half\_d = (self.size \* self.brush\_deformation\_factor) / 2

        points = []

        step = 0

        while (self.size / 2) \* 3 + (step - 1) \* self.size < self.max\_y:

            points.append((

                ScreenPoint(

                    x=half\_d \* 2,

                    y=(self.size / 2) \* 2 + step \* self.size

                ),

                ScreenPoint(

                    x=self.max\_x - half\_d \* 2,

                    y=(self.size / 2) \* 1 + step \* self.size

                )

            ))

            points.append((

                ScreenPoint(

                    x=self.max\_x - half\_d \* 2,

                    y=(self.size / 2) \* 3 + step \* self.size

                ),

                ScreenPoint(

                    x=half\_d \* 2,

                    y=(self.size / 2) \* 2 + step \* self.size

                ),

            ))

            step += 1

        return points

    def get\_points(self) -> Iterable[PathPoint]:

        # Get half circle path

        circle\_points\_left = PathCircle(

            self.size // 2,

            math.pi / 2,

            (math.pi \* 3) / 2,

        ).get\_points\_list()

        circle\_points\_left.reverse()

        circle\_points\_right = PathCircle(

            self.size // 2,

            (math.pi \* 3) / 2,

            (math.pi \* 5) / 2,

        ).get\_points\_list()

        key\_points = self.\_get\_key\_points()

        points = []

        for step in range(len(key\_points)):

            p0, p1 = key\_points[step]

            line = PathLine(p0, p1).get\_points\_list()

            turn = circle\_points\_right if step % 2 == 0 else circle\_points\_left

            turn = [PathPoint(ScreenPoint(

                p.coord.x + p1.x,

                p.coord.y + p1.y + self.size / 2,

            ), p.angle) for p in turn]

            points.extend(line)

            points.extend(turn)

        return points

class PathRectEdge(Path):

    def \_\_init\_\_(self, start: PathPoint, size, brush\_deformation\_factor, max\_x, max\_y):

        self.start = start

        self.size = size

        self.brush\_deformation\_factor = brush\_deformation\_factor

        self.max\_x = max\_x

        self.max\_y = max\_y

    def get\_points(self) -> Iterable[PathPoint]:

        margin\_v = self.size / 2

        margin\_h = self.size \* self.brush\_deformation\_factor / 2

        start = ScreenPoint(self.start.coord.x, self.max\_y - margin\_v)

        points = []

        for x in range(int(start.x), -2, -1):

            points.append(PathPoint(ScreenPoint(

                x,

                self.max\_y - margin\_v + 1

            ), math.pi / 2))

        angle\_points = PathCircle(self.size / 2, 0, math.pi / 2).get\_points\_list()

        angle\_points = [PathPoint(ScreenPoint(

            p.coord.x,

            p.coord.y + self.max\_y - margin\_v \* 2

        ), p.angle) for p in angle\_points]

        points.extend(angle\_points[::-1])

        for y in range(int(self.max\_y - margin\_v - 3), -1, -1):

            points.append(PathPoint(ScreenPoint(

                margin\_h - 1,

                y

            ), math.pi))

        angle\_points = PathCircle(self.size / 2, math.pi, math.pi \* 3 / 2).get\_points\_list()

        angle\_points = [PathPoint(ScreenPoint(

            p.coord.x + margin\_h \* 2,

            p.coord.y

        ), p.angle) for p in angle\_points]

        points.extend(angle\_points[::-1])

        for x in range(int(margin\_h + 3), self.max\_x, 1):

            points.append(PathPoint(ScreenPoint(

                x,

                margin\_v - 1

            ), math.pi / 2))

        angle\_points = PathCircle(self.size / 2, math.pi, math.pi \* 3 / 2).get\_points\_list()

        angle\_points = [PathPoint(ScreenPoint(

            p.coord.x + self.max\_x,

            p.coord.y + margin\_v \* 2

        ), p.angle) for p in angle\_points]

        points.extend(angle\_points[::-1])

        for y in range(int(margin\_v + 3), self.max\_y, 1):

            points.append(PathPoint(ScreenPoint(

                self.max\_x - margin\_h,

                y

            ), math.pi))

        angle\_points = PathCircle(self.size / 2, math.pi \* 3 / 2, math.pi \* 4 / 2).get\_points\_list()

        angle\_points = [PathPoint(ScreenPoint(

            p.coord.x + self.max\_x - margin\_h \* 2,

            p.coord.y + self.max\_y

        ), p.angle) for p in angle\_points]

        points.extend(angle\_points[::-1])

        for x in range(int(self.max\_x - margin\_h - 3), self.max\_x, -1):

            points.append(PathPoint(ScreenPoint(

                x,

                self.max\_y - margin\_v

            ), math.pi / 2))

        return points

import asyncio

import heapq

import platform

import time

from typing import NamedTuple, List, Optional, Dict, Union, Tuple

from .\_rich.control import ControlType, CONTROL\_CODES\_FORMAT

from .\_rich.simple\_console import SimpleConsole

from .utils import clamp, cache

from .screen import ScreenPoint

WINDOWS = platform.system() == 'Windows'

async def sleep(second: float):

    if second <= 0:

        return

    if second > 0.1:

        return await asyncio.sleep(second)

    start = time.perf\_counter()

    while time.perf\_counter() - start < second:

        continue

class Render(SimpleConsole):

    @property

    @cache

    def screen\_size(self):

        return self.size

    def move\_cursor\_home(self):

        char\_control = CONTROL\_CODES\_FORMAT[ControlType.HOME]()

        self.write(char\_control)

        self.flush()

    def clear(self):

        char\_control = CONTROL\_CODES\_FORMAT[ControlType.CLEAR]()

        self.write(char\_control)

        self.flush()

    def string\_at(self, p: Union[Tuple, ScreenPoint], s: str) -> str:

        """Use the terminal control character to position a string"""

        if not isinstance(p, ScreenPoint):

            p = ScreenPoint(\*p)

        # if p.x > self.screen\_size.width - 1:

        #     return

        # if p.y > self.screen\_size.height - 2:

        #     return

        # if p.x < 0:

        #     return

        # if p.y < 0:

        #     return

        # x = int(p.x)

        # y = int(p.y)

        x = int(clamp(0, p.x, self.screen\_size.width - 1))

        y = int(clamp(0, p.y, self.screen\_size.height - 2))

        char\_control = CONTROL\_CODES\_FORMAT[ControlType.CURSOR\_MOVE\_TO](x, y)

        return char\_control + s

    def draw\_string\_at(self, p: Union[Tuple, ScreenPoint], s: str, flush=True):

        self.write(self.string\_at(p, s))

        if flush:

            self.flush()

class AnimationRender(Render):

    CLEAR\_CHAR = ' '

    class TimedDrawStruct(NamedTuple):

        frame\_idx: int

        point: ScreenPoint

        char: str

    class TimedDrawFullFrame(NamedTuple):

        frame\_idx: int

        buffer: str

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super().\_\_init\_\_(\*args, \*\*kwargs)

        # This will be maintained sorted (priority queue)

        self.\_scheduled: List[AnimationRender.TimedDrawStruct] = []

        self.\_evt\_stop = asyncio.Event()

    def schedule\_draw(

            self,

            frame: int, p: ScreenPoint, s: str,

            clean\_after: Optional[int] = None

    ):

        heapq.heappush(self.\_scheduled, AnimationRender.TimedDrawStruct(frame, p, s))

        if clean\_after is not None:

            self.schedule\_draw(frame + clean\_after, p, self.CLEAR\_CHAR, clean\_after=None)

    def \_process\_frames(self, frames: List[TimedDrawStruct]) -> List[TimedDrawFullFrame]:

        """

        Return a time-indexed full frame. The potential issue of this implementation is that,

        The full frame string will be fixed during such `process`-time, so if terminal got resized this,

        the drawing will be messed up.

        Assuming `frames` are already sorted (by time).

        """

        if len(frames) == 0:

            return []

        frame\_list: List[AnimationRender.TimedDrawFullFrame] = []

        def \_insert(chuck: List[AnimationRender.TimedDrawStruct]):

            if not chuck:

                return

            frame\_idx = chuck[0].frame\_idx

            frame: Dict[ScreenPoint, str] = {}

            for tds in chuck:

                if tds.frame\_idx != frame\_idx:

                    raise RuntimeError('All the `TimedDrawStruct`s must have the same frame index')

                same\_pos = frame.get(tds.point)

                if tds.char == self.CLEAR\_CHAR and same\_pos and same\_pos != self.CLEAR\_CHAR:

                    # If the (x, y) is already scheduled with a non-clear-char, don't put the clear-char here.

                    continue

                frame[tds.point] = tds.char

            frame\_str = ''.join([self.string\_at(p, frame[p]) for p in frame])

            frame\_list.append(AnimationRender.TimedDrawFullFrame(frame\_idx, frame\_str))

        # Chunk all the frames in O(n) time

        slow, fast = 0, 1

        while True:

            if fast == len(frames):

                \_insert(frames[slow:fast])

                break

            if frames[fast].frame\_idx != frames[slow].frame\_idx:

                \_insert(frames[slow:fast])

                slow = fast

            fast += 1

        return frame\_list

    async def render\_frames(self, frame\_interval\_s=0.005, min\_sleep\_delay=0):

        # NOTED: We assume all frames are already scheduled i.e. `self.\_scheduled` is fixed

        if len(self.\_scheduled) == 0:

            return

        full\_frame\_list = self.\_process\_frames(

            # Pop all the items (sorted)

            heapq.nsmallest(len(self.\_scheduled), self.\_scheduled)

        )

        current\_frame = 0

        for frame in full\_frame\_list:

            empty\_frames = frame.frame\_idx - current\_frame

            if empty\_frames > 0:

                sleep\_time = frame\_interval\_s \* empty\_frames

                if sleep\_time > min\_sleep\_delay:

                    await sleep(sleep\_time)

            self.write(frame.buffer)

            self.flush()

            current\_frame = frame.frame\_idx

import math

from abc import ABC, abstractmethod

from typing import List

from .screen import ScreenPoint, ScreenPointDrawing

class Brush(ABC):

    """

    Brush base class

    You may implement your own brush style.

    """

    @abstractmethod

    def get\_points(self, x, y, angle) -> List[ScreenPointDrawing]:

        """Return all the brush points"""

        pass

class BrushWipe(Brush):

    width = 6

    deformation\_factor = 2

    def get\_points(self, x, y, angle) -> List[ScreenPointDrawing]:

        """Return all the brush points"""

        half\_width = self.width / 2

        opposite\_angle = angle + math.pi

        ret = []

        for step in range(0, int(half\_width \* self.deformation\_factor)):

            fac = half\_width / (half\_width \* self.deformation\_factor) \* step

            p = ScreenPoint(

                x + math.cos(angle)          \* fac \* self.deformation\_factor,  # noqa: E221

                y + math.sin(angle)          \* fac                             # noqa: E221

            )

            ret.append(ScreenPointDrawing(p, '#'))

            p = ScreenPoint(

                x + math.cos(opposite\_angle) \* fac \* self.deformation\_factor,

                y + math.sin(opposite\_angle) \* fac

            )

            ret.append(ScreenPointDrawing(p, '#'))

        return ret