**ДОДАТОК Б ЛІСТИНГ**

**arrows\_game.py**

import os  
  
import pygame  
  
from assets.board import Board  
from assets.buttons.add\_arrow\_button import AddArrowButton  
from assets.buttons.delete\_arrow\_button import DeleteArrowButton  
from assets.buttons.end\_session\_button import EndSessionButton  
from assets.buttons.gen\_new\_board\_button import GenNewBoardButton  
from assets.messages.correct\_message import CorrectMessage  
from assets.messages.start\_message import StartMessage  
from assets.messages.wrong\_message import WrongMessage  
from control import colors  
from control.screen import Screen  
from control.settings import Settings  
from control.states import States  
from control.time\_control import clock  
from utils.core import Core  
  
  
class ArrowsGame:  
 *"""Main app class"""* def \_\_init\_\_(self):  
 *"""  
 Init game objects  
 """* pygame.init()  
  
 # initialize game objects  
 self.board: Board = Board()  
 self.gen\_new\_board\_button: GenNewBoardButton = GenNewBoardButton()  
 self.add\_arrows\_buttons: list = []  
 self.delete\_arrow\_button: DeleteArrowButton | None = None  
 self.end\_session\_button: EndSessionButton | None = None  
 self.message: StartMessage | WrongMessage | CorrectMessage | None = StartMessage()  
  
 # get screen surface to create window  
 Screen.set\_caption('Arrows')  
  
 def \_handle\_events(self) -> None:  
 *"""  
 Handle pygame events queue  
 """* for event in pygame.event.get():  
 # handle quit event  
 if event.type == pygame.QUIT:  
 exit(0)  
  
 # handle mouse events  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 mouse\_pos = pygame.mouse.get\_pos()  
  
 # active game events  
 if States.current\_state == States.GAME\_ACTIVE:  
 # button click events  
 self.\_handle\_gen\_new\_board\_event(mouse\_pos)  
 self.\_handle\_add\_arrow\_event(mouse\_pos)  
 self.\_handle\_delete\_arrow\_event(mouse\_pos)  
 self.\_handle\_end\_session\_event(mouse\_pos)  
 # arrow and number selection event  
 self.\_handle\_arrow\_selection\_event(mouse\_pos)  
 self.\_handle\_number\_selection\_event(mouse\_pos)  
  
 # non-active game events  
 elif States.current\_state == States.GAME\_START:  
 self.\_handle\_start\_message\_events(mouse\_pos)  
 elif States.current\_state == States.GAME\_END\_WRONG:  
 self.\_handle\_end\_message\_wrong\_events(mouse\_pos)  
 elif States.current\_state == States.GAME\_END\_CORRECT:  
 self.\_handle\_end\_message\_correct\_events(mouse\_pos)  
  
 def \_handle\_gen\_new\_board\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Generate new board if gen\_new\_board\_button is clicked, clear add and delete arrow buttons  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.gen\_new\_board\_button.is\_clicked(mouse\_pos):  
 self.board = Board()  
 self.add\_arrows\_buttons.clear()  
 self.delete\_arrow\_button = None  
  
 def \_handle\_add\_arrow\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Set arrow image and direction of select arrow grid square if add\_arrow\_button is clicked. Updates selection for  
 correct highlighting of numbers that the arrow points to. Creates delete arrow button. Creates and end session  
 button if every arrow grid square is filled and gets rid of error numbers highlighting.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* for add\_arrow\_button in self.add\_arrows\_buttons:  
 if add\_arrow\_button.is\_clicked(mouse\_pos):  
 self.board.set\_arrow\_image(\*add\_arrow\_button.handle\_click())  
 self.board.update\_selection()  
 self.delete\_arrow\_button = DeleteArrowButton(len(self.add\_arrows\_buttons))  
  
 filled\_arrow\_squares = [arrow for arrow in self.board.arrows if arrow.direction]  
 if len(filled\_arrow\_squares) == 2 \* (Settings.grid\_count.x + Settings.grid\_count.y):  
 self.end\_session\_button = EndSessionButton()  
 if self.board.wrong\_numbers:  
 self.board.dehighlight\_errors()  
 self.board.wrong\_numbers.clear()  
  
 def \_handle\_delete\_arrow\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Sets selected image and direction to None. Updates selection for correct highlighting of numbers that the arrow  
 points to. Deletes delete arrow button and end session button, gets rid of error numbers highlighting.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if not self.delete\_arrow\_button:  
 return  
 if self.delete\_arrow\_button.is\_clicked(mouse\_pos):  
 self.board.set\_arrow\_image(None**,** None**,** colors.highlighted\_blue)  
 self.delete\_arrow\_button = None  
 self.board.update\_selection()  
 self.end\_session\_button = None  
 if self.board.wrong\_numbers:  
 self.board.dehighlight\_errors()  
 self.board.wrong\_numbers = []  
  
 def \_handle\_end\_session\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Evaluates correctness of arrows and sets appropriate game state  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if not self.end\_session\_button:  
 return  
 if self.end\_session\_button.is\_clicked(mouse\_pos):  
 self.board.check\_correctness()  
 if self.board.wrong\_numbers:  
 States.current\_state = States.GAME\_END\_WRONG  
 self.message = WrongMessage()  
 else:  
 States.current\_state = States.GAME\_END\_CORRECT  
 self.message = CorrectMessage()  
  
 def \_handle\_arrow\_selection\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Highlights selected arrow and numbers that it points to. Adds button for adding arrows to selected square.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.board.check\_arrow\_selection(mouse\_pos):  
 self.board.handle\_arrow\_selection(mouse\_pos)  
 arrow = self.board.get\_arrow(mouse\_pos)  
 self.add\_arrows\_buttons.clear()  
 self.delete\_arrow\_button = None  
 if arrow.selected:  
 possible\_directions = Core.get\_possible\_directions(arrow.arrow\_set**,** arrow.arrow\_num)  
 for i**,** direction in enumerate(possible\_directions):  
 self.add\_arrows\_buttons.append(AddArrowButton(direction**,** i))  
 if arrow.direction:  
 self.delete\_arrow\_button = DeleteArrowButton(len(self.add\_arrows\_buttons))  
  
 def \_handle\_number\_selection\_event(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Highlights selected number and arrows that point to it. Clears all arrow manipulation buttons  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.board.check\_number\_selection(mouse\_pos):  
 self.board.handle\_number\_selection(mouse\_pos)  
 self.add\_arrows\_buttons.clear()  
 self.delete\_arrow\_button = None  
  
 def \_handle\_start\_message\_events(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Sets game state to active and deletes message.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.message.collide\_rect.collidepoint(mouse\_pos):  
 States.current\_state = States.GAME\_ACTIVE  
 self.message = None  
  
 def \_handle\_end\_message\_correct\_events(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Handles end message button click when game is over and all numbers on grid match with number of arrows that  
 point to it.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.message.collide\_rect\_again.collidepoint(mouse\_pos):  
 States.current\_state = States.GAME\_ACTIVE  
 self.board = Board()  
 self.message = None  
 self.add\_arrows\_buttons.clear()  
 self.delete\_arrow\_button = None  
 self.end\_session\_button = None  
 elif self.message.collide\_rect\_quit.collidepoint(mouse\_pos):  
 exit(0)  
  
 def \_handle\_end\_message\_wrong\_events(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Handles end message button click when game is over and at least one number on grid don't with number of arrows  
 that point to it.  
  
 :return: None  
 :param mouse\_pos: Mouse position  
 """* if self.message.collide\_rect\_continue.collidepoint(mouse\_pos):  
 self.board.highlight\_errors()  
 States.current\_state = States.GAME\_ACTIVE  
 self.message = None  
 self.board.deselect\_all()  
 self.add\_arrows\_buttons.clear()  
 self.delete\_arrow\_button = None  
 self.end\_session\_button = None  
 elif self.message.collide\_rect\_quit.collidepoint(mouse\_pos):  
 exit(0)  
  
 def \_update\_screen(self):  
 *"""  
 Render updated objects on screen and update screen  
 """* # fill background  
 Screen.surface.fill(Screen.bg\_color)  
  
 # redraw objects based on current state  
 if States.current\_state == States.GAME\_ACTIVE:  
 self.board.draw()  
 self.gen\_new\_board\_button.draw()  
 for add\_replace\_button in self.add\_arrows\_buttons:  
 add\_replace\_button.draw()  
 if self.delete\_arrow\_button:  
 self.delete\_arrow\_button.draw()  
 if self.end\_session\_button:  
 self.end\_session\_button.draw()  
 if States.current\_state in [States.GAME\_START**,** States.GAME\_END\_CORRECT**,** States.GAME\_END\_WRONG]:  
 self.message.draw()  
  
 # update screen to expose newly drawn objects  
 pygame.display.update()  
  
 def run(self):  
 *"""  
 Run main game loop  
 """* while True:  
 self.\_handle\_events()  
 self.\_update\_screen()  
 clock.tick(Settings.fps)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # set absolute path when launching from shortcuts  
 os.chdir(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
  
 # create and run game  
 arrows\_game = ArrowsGame()  
 arrows\_game.run()

**collide\_rect\_evaluetor.py**

import pygame  
  
  
class CollideRectEvaluator:  
 *"""Evaluates collision rects for message buttons"""* @classmethod  
 def evaluate\_from\_ratios(cls**,** ratios: list[tuple[float**,** float]]**,** base\_rect: pygame.Rect) -> pygame.Rect:  
 *"""  
 Get collision rect for message buttons  
  
 :return: Collision rect for message buttons  
 :param ratios: Coordinates of collide rect box divided by image size  
 :param base\_rect: Rect of image that holds collide rect  
 """* collide\_rect\_width = (ratios[1][0] - ratios[0][0]) \* base\_rect.width  
 collide\_rect\_top = base\_rect.top + ratios[0][1] \* base\_rect.height  
 collide\_rect\_left = base\_rect.left + ratios[0][0] \* base\_rect.width  
 collide\_rect\_height = (ratios[1][1] - ratios[0][1]) \* base\_rect.height  
  
 collide\_rect = pygame.Rect(  
 (collide\_rect\_left**,** collide\_rect\_top)**,** (collide\_rect\_width**,** collide\_rect\_height)  
 )  
  
 return collide\_rect

**core.py**

import random  
  
import pygame.sprite  
from pygame.math import Vector2  
  
from control.settings import Settings  
  
  
class Core:  
 *"""Class for various functions and variables for core game logic"""* arrow\_directions: list[tuple[int**,** int]] = [  
 (1**,** -1)**,** (1**,** 0)**,** (1**,** 1)**,** (0**,** 1)**,** (-1**,** 1)**,** (-1**,** 0)**,** (-1**,** -1)**,** (0**,** -1)**,** ]  
  
 arrow\_sets: dict[tuple[int**,** int]**,** list[tuple[int**,** int]]] = {  
 (0**,** -1): ((0**,** 1)**,** (1**,** 1)**,** (-1**,** 1))**,** (1**,** 0): ((-1**,** 0)**,** (-1**,** 1)**,** (-1**,** -1))**,** (-1**,** 0): ((1**,** 0)**,** (1**,** 1)**,** (1**,** -1))**,** (0**,** 1): ((0**,** -1)**,** (1**,** -1)**,** (-1**,** -1))  
 }  
  
 forbidden\_directions: dict[tuple[int**,** int]**,** list[tuple[int**,** int]]] = {  
 (0**,** -1): [(-1**,** 1)**,** (1**,** 1)]**,** (1**,** 0): [(-1**,** -1)**,** (-1**,** 1)]**,** (-1**,** 0): [(1**,** -1)**,** (1**,** 1)]**,** (0**,** 1): [(-1**,** -1)**,** (1**,** -1)]  
 }  
  
 arrows: dict[tuple[int**,** int]**,** list[tuple[int**,** int]]] = {}  
 numbers: list[list[int]] = []  
  
 @classmethod  
 def gen\_arrows(cls) -> None:  
 *"""  
 Generate arrows dict with keys as direction on game board and list of arrows as values  
 """* arrows = {}  
 for arrow\_set in cls.arrow\_sets:  
 arrows[arrow\_set] = []  
 grid\_count = Settings.grid\_count.x if arrow\_set in [(0**,** -1)**,** (0**,** 1)] else Settings.grid\_count.y  
 for i in range(int(grid\_count)):  
 possible\_directions = cls.get\_possible\_directions(arrow\_set**,** i)  
 choice = random.choice(possible\_directions)  
 arrows[arrow\_set].append(choice)  
  
 cls.arrows = arrows  
  
 @classmethod  
 def get\_possible\_directions(cls**,** arrow\_set: tuple[int**,** int]**,** arrow\_num: int) -> list[tuple[int**,** int]]:  
 *"""  
 Get possible arrow directions for given arrow location  
  
 :return: List of possible directions that arrow can point to  
 :param arrow\_set: Direction in which the arrow is located  
 :param arrow\_num: Sequence number of arrow on arrow set counting from up or left  
 """* grid\_count = Settings.grid\_count.x if arrow\_set in [(0**,** -1)**,** (0**,** 1)] else Settings.grid\_count.y  
 possible\_directions = list(cls.arrow\_sets[arrow\_set])  
 if arrow\_num == 0:  
 possible\_directions.remove(cls.forbidden\_directions[arrow\_set][0])  
 if arrow\_num == grid\_count - 1:  
 possible\_directions.remove(cls.forbidden\_directions[arrow\_set][1])  
 return possible\_directions  
  
 @classmethod  
 def get\_position(cls**,** arrows\_set\_direction: tuple[int**,** int]**,** arrow\_num: int) -> Vector2:  
 *"""  
 Get position relative to board of given arrow  
  
 :return: Position vector of arrow relative to game board  
 :param arrows\_set\_direction: Direction in which the arrow is located  
 :param arrow\_num: Sequence number of arrow on arrow set counting from up or left  
 """* if arrows\_set\_direction == (0**,** -1):  
 position = arrow\_num**,** -1  
 elif arrows\_set\_direction == (1**,** 0):  
 position = Settings.grid\_count.x**,** arrow\_num  
 elif arrows\_set\_direction == (-1**,** 0):  
 position = -1**,** arrow\_num  
 elif arrows\_set\_direction == (0**,** 1):  
 position = arrow\_num**,** Settings.grid\_count.y  
 else:  
 raise ValueError('arrow set direction not valid')  
  
 return Vector2(position)  
  
 @classmethod  
 def get\_span(cls**,** position: Vector2**,** arrow: tuple[int**,** int] | None = None) -> list[Vector2**,** ...]:  
 *"""  
 Get all grid squares that given arrow points to  
  
 :return: List of grid square positions that given arrow points to  
 :param position: Position vector of arrow relative to game board  
 :param arrow: Direction in which arrow points to  
 """* if not arrow:  
 return []  
  
 grid\_squares = []  
 grid\_square = position.copy()  
 while True:  
 grid\_square += Vector2(arrow)  
  
 rect = pygame.Rect((0**,** 0)**,** tuple(Settings.grid\_count))  
 if not rect.collidepoint(tuple(grid\_square)):  
 break  
  
 grid\_squares.append(grid\_square.copy())  
  
 return grid\_squares  
  
 @classmethod  
 def get\_pointings(cls**,** grid\_square: Vector2) -> list[tuple[tuple[int**,** int]**,** int]]:  
 *"""  
 Get arrows that point to specified location on board  
  
 :return: List of arrow sets and arrow numbers that point to given grid square  
 :param grid\_square: Position of grid square relative to board  
 """* result = []  
 for arrows\_set\_direction**,** arrows\_set in cls.arrows.items():  
 for arrow\_num**,** arrow in enumerate(arrows\_set):  
 position = cls.get\_position(arrows\_set\_direction**,** arrow\_num)  
 if arrow:  
 if grid\_square in cls.get\_span(position**,** arrow):  
 result.append((arrows\_set\_direction**,** arrow\_num))  
 return result  
  
 @classmethod  
 def count\_pointings(cls**,** grid\_square: Vector2) -> int:  
 *"""  
 Count number of arrows that point to specified location on board  
  
 :return: Number of arrows that point to given grid square  
 :param grid\_square: Position of grid square relative to board  
 """* return len(cls.get\_pointings(grid\_square))  
  
 @classmethod  
 def evaluate\_correctness(cls) -> list[tuple[int**,** int]]:  
 *"""  
 Get all numbers that don't match with number of arrows that point to them  
  
 :return: List of numbers, values of which don't match with number of arrows that point to them  
 """* wrong\_numbers = []  
 for col**,** numbers\_col in enumerate(cls.numbers):  
 for row**,** number in enumerate(numbers\_col):  
 if number != cls.count\_pointings(Vector2(col**,** row)):  
 wrong\_numbers.append((col**,** row))  
 return wrong\_numbers  
  
 @classmethod  
 def gen\_numbers(cls) -> None:  
 *"""  
 Generate numbers matrix based on previously generated arrows  
 """* cls.gen\_arrows()  
  
 cls.numbers = [  
 [  
 cls.count\_pointings(Vector2(col**,** row))  
 for row in range(int(Settings.grid\_count.y))  
 ]  
 for col in range(int(Settings.grid\_count.x))  
 ]

**colors.py**

*"""  
Colors of all game objects  
"""*background = (29**,** 29**,** 38)  
board\_frame = (199**,** 166**,** 93)  
grid\_square\_frames = (224**,** 149**,** 123)  
numbers = (148**,** 85**,** 141)  
arrows = (135**,** 135**,** 197)  
  
highlighted\_blue = (30**,** 30**,** 50)  
highlighted\_yellow = (45**,** 35**,** 10)  
highlight\_red = (30**,** 0**,** 0)  
  
delete\_arrow\_button = (240**,** 62**,** 92)  
end\_session\_button = (108**,** 192**,** 108)  
arrows\_button = (200**,** 200**,** 220)  
new\_board\_button = (68**,** 174**,** 221)

**grid\_position.py**

from control.settings import Settings  
from pygame.math import Vector2  
  
  
class GridPosition:  
 *"""Class to manage coordinates of game objects on game board"""* def \_\_init\_\_(self**,** grid\_square\_pos: Vector2 | tuple):  
 *"""  
 :return: None  
 :param grid\_square\_pos: Column and row of grid square relative to board  
 """* self.grid\_square\_pos = Vector2(grid\_square\_pos)  
  
 def get\_coords(self) -> Vector2:  
 *"""  
 Get exact pixel position on upper right corner of the object  
  
 :return: Vector of pixel coordinates of topleft corner of grid square  
 """* return (self.grid\_square\_pos + Settings.window\_margin) \* Settings.grid\_size  
  
 def get\_coords\_center(self) -> Vector2:  
 *"""  
 Get center of specified grid square in pixels  
  
 :return: Vector of pixel coordinates of the center of grid square  
 """* return self.get\_coords() + Vector2(Settings.grid\_size**,** Settings.grid\_size) / 2

**screen.py**

import pygame  
from dataclasses import dataclass  
  
from control import colors  
from control.settings import Settings  
  
  
@dataclass  
class Screen:  
 *"""Class to store screen rect and surface"""* surface = pygame.display.set\_mode(Settings.get\_resolution())  
 rect = surface.get\_rect()  
 bg\_color = colors.background  
  
 @classmethod  
 def set\_caption(cls**,** name: str) -> None:  
 *"""  
 Set caption for game window  
  
 :return: None  
 :param name: Name for window caption  
 """* pygame.display.set\_caption(name)

**settings.py**

from dataclasses import dataclass  
  
from pygame.math import Vector2  
  
  
@dataclass(frozen=True)  
class Settings:  
 *"""Class to store settings for the app"""* # basic settings  
 grid\_count: Vector2 = Vector2(8**,** 8)  
 window\_margin: Vector2 = Vector2(2**,** 2)  
 grid\_size: int = 75  
 fps: int = 30  
  
 # arrows settings  
 arrow\_size: int = 35  
  
 # numbers size  
 number\_size: int = 48  
  
 # button settings  
 button\_icon\_size: int = 40  
  
 # message settings  
 message\_size: int = 500  
  
 @classmethod  
 def get\_resolution(cls) -> Vector2:  
 *"""  
 Get actual pixel resolution of entire screen  
  
 :return: Actual pixel resolution of entire screen  
 """* return (cls.grid\_count + 2 \* cls.window\_margin + Vector2(0**,** 1)) \* cls.grid\_size

**states.py**

from dataclasses import dataclass  
  
  
@dataclass  
class States:  
 *"""Class to store all and current game states"""* # all game states  
 GAME\_ACTIVE: int = 0  
 GAME\_START: int = 1  
 GAME\_END\_CORRECT: int = 2  
 GAME\_END\_WRONG: int = 3  
  
 # current game state  
 current\_state = GAME\_START

**time\_control.py**

*"""  
Clock to control fps of the game and other time dependent objects  
"""*import pygame  
  
clock = pygame.time.Clock()

**add\_arrows\_button.py**

from assets.arrow import Arrow  
from assets.buttons.button import Button  
from control import colors  
from control.grid\_position import GridPosition  
from control.settings import Settings  
  
  
class AddArrowButton(Button):  
 *"""Buttons for adding and replacing"""* def \_\_init\_\_(self**,** direction: tuple[int**,** int]**,** position: int):  
 *"""  
 :return: None  
 :param direction: Direction of arrow that it adds when button is clicked  
 """* super().\_\_init\_\_(  
 f'../assets/buttons/arrow{direction}.png'**,** GridPosition((position - 2**,** Settings.grid\_count.y + 2))**,** colors.arrows\_button**,** False**,** Settings.button\_icon\_size / 500  
 )  
 self.direction = direction  
  
 def handle\_click(self) -> tuple[Arrow**,** tuple[int**,** int]**,** tuple[int]]:  
 *"""  
 Return new arrow image of given direction to set as image attribute of selected square  
  
 :return: Needed attributes for changing the image of arrow grid square: new arrow image, its direction and color  
 that it was highlighted with  
 """* return Arrow(self.direction).image**,** self.direction**,** colors.highlighted\_blue

**button.py**

import pygame  
from pygame.constants import BLEND\_RGB\_MULT  
from pygame.math import Vector2  
  
from abc import abstractmethod  
  
from control import colors  
from control.grid\_position import GridPosition  
from control.screen import Screen  
from control.settings import Settings  
  
  
class Button:  
 *"""Abstract button class for user interface buttons"""* def \_\_init\_\_(self**,** image\_path: str**,** position: GridPosition**,** color: tuple[int**,** ...]**,** conform\_size: bool = True**,** scaling\_nonconformal: float = None):  
 *"""  
 :return: None  
 :param image\_path: Filepath to button image  
 :param position: Grid square position relative to board  
 :param color: Color to fill the button image with  
 :param conform\_size: Scale image according to button icon size value in settings. If False -  
 scaling\_nonconformal parameter is required  
 :param scaling\_nonconformal: Value to scale image with  
 """* # basic attributes  
 self.image = pygame.image.load(image\_path).convert\_alpha()  
 self.image\_size = self.image.get\_size()  
 self.position = position  
  
 # scale image if necessary  
 if conform\_size:  
 image\_size = Vector2(self.image.get\_size())  
 scaling\_factor = Settings.button\_icon\_size / image\_size.x  
 self.image\_size = tuple(scaling\_factor \* image\_size)  
 self.image = pygame.transform.smoothscale(self.image**,** self.image\_size)  
 if scaling\_nonconformal:  
 self.image\_size = tuple(scaling\_nonconformal \* Vector2(self.image\_size))  
 self.image = pygame.transform.smoothscale(self.image**,** self.image\_size)  
  
 # color button image  
 self.image.fill(color**,** special\_flags=BLEND\_RGB\_MULT)  
 background = pygame.Surface(self.image\_size)  
 background.fill(colors.background)  
 background.blit(self.image**,** (0**,** 0))  
 self.image = background  
  
 # set rect  
 self.rect = self.image.get\_rect()  
 self.rect.center = position.get\_coords\_center()  
  
 def is\_clicked(self**,** mouse\_pos: tuple[int**,** int]) -> bool:  
 *"""Check if button is clicked"""* if self.rect.collidepoint(mouse\_pos):  
 return True  
 return False  
  
 def draw(self) -> None:  
 *"""Draw object to given surface"""* Screen.surface.blit(self.image**,** self.rect)  
  
 @abstractmethod  
 def handle\_click(self**,** \*args**,** \*\*kwargs):  
 *"""Abstract method for action that button makes"""* return

**delete\_arrow\_button.py**

from assets.buttons.button import Button  
from control import colors  
from control.grid\_position import GridPosition  
from control.settings import Settings  
  
  
class DeleteArrowButton(Button):  
 *"""Buttons for deleting arrows"""* def \_\_init\_\_(self**,** position: int):  
 super().\_\_init\_\_(  
 '../assets/buttons/delete\_arrow\_button.png'**,** GridPosition((position - 2**,** Settings.grid\_count.y + 2))**,** colors.delete\_arrow\_button  
 )  
  
 def handle\_click(self):  
 *"""Handles in main game class"""* return

**end\_session\_button.py**

from pygame import Vector2  
  
from assets.buttons.button import Button  
from control import colors  
from control.grid\_position import GridPosition  
from control.settings import Settings  
  
  
class EndSessionButton(Button):  
 *"""Button class for ending session and checking if player won"""* def \_\_init\_\_(self):  
 super().\_\_init\_\_(  
 '../assets/buttons/end\_session\_button.png'**,** GridPosition(Settings.grid\_count + Vector2(0**,** 2))**,** colors.end\_session\_button  
 )  
  
 def handle\_click(self):  
 *"""Handles in main game class"""* return

**gen\_new\_board\_button.py**

from pygame.math import Vector2  
  
from assets.buttons.button import Button  
from control import colors  
from control.grid\_position import GridPosition  
from control.settings import Settings  
  
  
class GenNewBoardButton(Button):  
 *"""Button class for genereting new board"""* def \_\_init\_\_(self):  
 super().\_\_init\_\_(  
 '../assets/buttons/gen\_new\_board\_button.png'**,** GridPosition(Settings.grid\_count + Vector2(1**,** 2))**,** colors.new\_board\_button  
 )  
  
 def handle\_click(self):  
 *"""Handles in main game class"""* return

**arrow\_grid\_square.py**

import pygame  
  
from assets.grid\_squares.grid\_square import GridSquare  
from control.grid\_position import GridPosition  
from utils.core import Core  
  
  
class ArrowGridSquare(GridSquare):  
 *"""Subclass of grid square to hold arrows and related attributes"""* def \_\_init\_\_(self**,** content: pygame.Surface | None**,** arrow\_set: tuple[int**,** int]**,** arrow\_num: int**,** direction: tuple[int**,** int] | None = None):  
 *"""  
 :return: None  
 :param arrow\_set: Direction in which the arrow is located  
 :param arrow\_num: Sequence number of arrow on arrow set counting from up or left  
 """* super().\_\_init\_\_(content)  
 self.arrow\_set = arrow\_set  
 self.arrow\_num = arrow\_num  
 self.position = GridPosition(Core.get\_position(arrow\_set**,** arrow\_num))  
 self.rect.topleft = self.position.get\_coords()  
 self.direction = direction  
  
 def set\_image(self**,** image: pygame.Surface**,** direction: tuple[int**,** int]) -> None:  
 *"""  
 Set arrow image for arrow grid square  
  
 :return: None  
 :param image: Arrow image to fill the arrow grid square  
 :param direction: Direction that arrow points to  
 """* self.\_\_init\_\_(image**,** self.arrow\_set**,** self.arrow\_num**,** direction)

**grid\_square.py**

import pygame  
from pygame.constants import BLEND\_RGB\_ADD**,** BLEND\_RGB\_SUB  
from pygame.math import Vector2  
  
from control import colors  
from control.screen import Screen  
from control.settings import Settings  
  
  
class GridSquare:  
 *"""Class for grid square objects that hold arrows or numbers"""* def \_\_init\_\_(self**,** content: pygame.Surface | None):  
 *"""  
 :return: None  
 :param content: Image to fill the square with  
 """* super().\_\_init\_\_()  
 # frame that surrounds the square  
 self.frame\_size = (Settings.grid\_size**,** Settings.grid\_size)  
 self.frame\_rect = pygame.Rect((0**,** 0)**,** self.frame\_size)  
 self.frame\_width = 1  
 self.frame\_color = colors.grid\_square\_frames  
  
 # get content rect  
 if content:  
 self.content\_image = content  
 self.content\_rect = content.get\_rect()  
 self.content\_rect.center = Vector2(Settings.grid\_size**,** Settings.grid\_size) / 2  
  
 # Indicator for toggling selection and related attributes  
 self.selected = False  
  
 # get image and rect for sprite.draw() method  
 self.image = pygame.Surface(self.frame\_size)  
 self.image.fill(colors.background)  
 pygame.draw.rect(self.image**,** self.frame\_color**,** self.frame\_rect**,** self.frame\_width)  
 if content:  
 self.image.blit(self.content\_image**,** self.content\_rect)  
  
 self.rect = self.image.get\_rect()  
  
 def draw(self) -> None:  
 Screen.surface.blit(self.image**,** self.rect)  
  
 def select(self**,** highlight\_color: tuple[int]) -> None:  
 *"""  
 Add given color to image  
  
 :return: None  
 :param highlight\_color: Color to restore highlighting with  
 """* self.selected = True  
 self.image.fill(highlight\_color**,** special\_flags=BLEND\_RGB\_ADD)  
  
 def deselect(self**,** highlight\_color: tuple[int]) -> None:  
 *"""  
 Subtract given color from image  
  
 :return: None  
 :param highlight\_color: Color to restore highlighting with  
 """* self.selected = False  
 self.image.fill(highlight\_color**,** special\_flags=BLEND\_RGB\_SUB)

**number\_grid\_square.py**

import pygame  
from pygame.constants import BLEND\_RGB\_ADD**,** BLEND\_RGB\_SUB  
  
from assets.grid\_squares.grid\_square import GridSquare  
from control import colors  
from control.grid\_position import GridPosition  
  
  
class NumberGridSquare(GridSquare):  
 *"""Subclass of grid square to hold arrows and related attributes"""* def \_\_init\_\_(self**,** content: pygame.Surface | None**,** col: int**,** row: int**,** value: int):  
 *"""  
 :return: None  
 :param content: Image to fill the square with  
 :param col: Column of number position relative to board  
 :param row: Row of number position relative to board  
 :param value: Numeric value of number object  
 """* super().\_\_init\_\_(content)  
 self.col = col  
 self.row = row  
 self.value = value  
 self.position = GridPosition((col**,** row))  
 self.rect.topleft = self.position.get\_coords()  
  
 def highlight\_error(self) -> None:  
 *"""  
 Add red highlight color to image  
 """* self.image.fill(colors.highlight\_red**,** special\_flags=BLEND\_RGB\_ADD)  
  
 def dehighlight\_error(self) -> None:  
 *"""  
 Subtract red highlight color from image  
 """* self.image.fill(colors.highlight\_red**,** special\_flags=BLEND\_RGB\_SUB)

**correct\_message.py**

from assets.messages.message import Message  
from utils.collide\_rect\_evaluetor import CollideRectEvaluator  
  
  
class CorrectMessage(Message):  
 *"""Message that appears before the game starts"""* def \_\_init\_\_(self):  
 super().\_\_init\_\_('../assets/messages/message\_correct.png')  
 collide\_rect\_ratios\_again = [(144 / 2000**,** 660 / 1000)**,** (838 / 2000**,** 887 / 1000)]  
 collide\_rect\_ratios\_quit = [(1162 / 2000**,** 660 / 1000)**,** (1856 / 2000**,** 887 / 1000)]  
 self.collide\_rect\_again = CollideRectEvaluator.evaluate\_from\_ratios(collide\_rect\_ratios\_again**,** self.rect)  
 self.collide\_rect\_quit = CollideRectEvaluator.evaluate\_from\_ratios(collide\_rect\_ratios\_quit**,** self.rect)

**message.py**

import pygame  
from pygame.math import Vector2  
  
from control.screen import Screen  
from control.settings import Settings  
  
  
class Message:  
 *"""Base class for messages"""* def \_\_init\_\_(self**,** image\_path: str):  
 *"""  
 :return: None  
 :param image\_path: Path to message image path  
 """* # get and scale image  
 self.image = pygame.image.load(image\_path).convert\_alpha()  
 size = self.image.get\_size()  
 scaling\_factor = Settings.message\_size / size[0]  
 new\_size = tuple(scaling\_factor \* Vector2(size))  
 self.image = pygame.transform.smoothscale(self.image**,** new\_size)  
 self.image\_size = self.image.get\_size()  
  
 # get and center rect  
 self.rect = self.image.get\_rect()  
 self.rect.center = Settings.get\_resolution() // 2  
  
 def draw(self) -> None:  
 Screen.surface.blit(self.image**,** self.rect)

**start\_message.py**

from assets.messages.message import Message  
from utils.collide\_rect\_evaluetor import CollideRectEvaluator  
  
  
class StartMessage(Message):  
 *"""Message that appears before the game starts"""* def \_\_init\_\_(self):  
 super().\_\_init\_\_('../assets/messages/message\_start.png')  
 collide\_rect\_ratios = [(557 / 2000**,** 544 / 1000)**,** (1459 / 2000**,** 837 / 1000)]  
 self.collide\_rect = CollideRectEvaluator.evaluate\_from\_ratios(collide\_rect\_ratios**,** self.rect)

**wrong\_message.py**

from assets.messages.message import Message  
from utils.collide\_rect\_evaluetor import CollideRectEvaluator  
  
  
class WrongMessage(Message):  
 *"""Message that appears before the game starts"""* def \_\_init\_\_(self):  
 super().\_\_init\_\_('../assets/messages/message\_wrong.png')  
 collide\_rect\_ratios\_continue = [(131 / 2000**,** 604 / 1000)**,** (826 / 2000**,** 832 / 1000)]  
 collide\_rect\_ratios\_quit = [(1149 / 2000**,** 604 / 1000)**,** (1843 / 2000**,** 832 / 1000)]  
 self.collide\_rect\_continue = CollideRectEvaluator.evaluate\_from\_ratios(collide\_rect\_ratios\_continue**,** self.rect)  
 self.collide\_rect\_quit = CollideRectEvaluator.evaluate\_from\_ratios(collide\_rect\_ratios\_quit**,** self.rect)

**arrow.py**

import pygame  
from pygame.constants import BLEND\_RGB\_MULT  
from pygame.math import Vector2  
  
from control import colors  
from control.settings import Settings  
  
  
class Arrow:  
 *"""Class for rendering arrows"""* def \_\_init\_\_(self**,** direction: tuple[int**,** int]):  
 *"""  
 :return: None  
 :param direction: Direction of arrow to render appropriate image  
 """* # load image and scale  
 arrow\_image = pygame.image.load(f'../assets/arrows/{direction}.png').convert\_alpha()  
  
 image\_size\_vec = Vector2(arrow\_image.get\_size())  
 scaling\_factor = Settings.arrow\_size / image\_size\_vec.length()  
 self.arrow\_image\_size = tuple(scaling\_factor \* image\_size\_vec)  
  
 arrow\_image = pygame.transform.smoothscale(arrow\_image**,** self.arrow\_image\_size)  
  
 # fill arrow with specific color  
 arrow\_image.fill(colors.arrows**,** special\_flags=BLEND\_RGB\_MULT)  
  
 # background surface to draw arrow on  
 background = pygame.Surface(self.arrow\_image\_size)  
 background.fill(colors.background)  
  
 # blit arrow to background  
 background.blit(arrow\_image**,** (0**,** 0))  
  
 self.image = background

**board.py**

import pygame  
from pygame.math import Vector2  
  
from assets.grid\_squares.arrow\_grid\_square import ArrowGridSquare  
from assets.grid\_squares.number\_grid\_square import NumberGridSquare  
from assets.number import Number  
from control import colors  
from control.grid\_position import GridPosition  
from control.screen import Screen  
from control.settings import Settings  
from utils.core import Core  
  
  
class Board:  
 *"""Class for game board that holds arrows and numbers"""* def \_\_init\_\_(self):  
 # frame that surrounds grid squares that hold numbers  
 self.frame\_size = tuple(Settings.grid\_count \* Settings.grid\_size)  
 self.frame\_rect = pygame.Rect(  
 tuple(GridPosition((0**,** 0)).get\_coords())**,** self.frame\_size  
 )  
 self.frame\_color = colors.board\_frame  
 self.frame\_width = 5  
  
 # generate random values matrix and arrows dictionary  
 Core.gen\_numbers()  
 arrows = Core.arrows  
 numbers = Core.numbers  
  
 # wrong numbers list for highlighting at the end of the game  
 self.wrong\_numbers = []  
  
 # create grid squares group that hold numbers  
 self.numbers = []  
 for col**,** numbers\_col in enumerate(numbers):  
 for row**,** number in enumerate(numbers\_col):  
 self.numbers.append(NumberGridSquare(  
 Number(number).image**,** col**,** row**,** number  
 ))  
  
 # create empty grid squares that hold arrows  
 self.arrows = []  
 for arrows\_set\_direction**,** arrows\_set in arrows.items():  
 for arrow\_num**,** arrow in enumerate(arrows\_set):  
 self.arrows.append(ArrowGridSquare(  
 None**,** arrows\_set\_direction**,** arrow\_num  
 ))  
  
 # current selection indicator for managing selections  
 self.currently\_selected: ArrowGridSquare | NumberGridSquare | None = None  
  
 def update\_selection(self) -> None:  
 *"""  
 Deselect and select object for correct arrow adding and deletion  
 """* selected\_arrow = self.get\_selected\_arrow()  
 if selected\_arrow:  
 self.handle\_arrow\_selection(selected\_arrow.position.get\_coords\_center())  
 self.handle\_arrow\_selection(selected\_arrow.position.get\_coords\_center())  
  
 def deselect\_all(self) -> None:  
 *"""  
 Deselect any selection  
 """* if isinstance(self.currently\_selected**,** ArrowGridSquare):  
 self.handle\_arrow\_selection(self.currently\_selected.position.get\_coords\_center())  
 if isinstance(self.currently\_selected**,** NumberGridSquare):  
 self.handle\_number\_selection(self.currently\_selected.position.get\_coords\_center())  
  
 def get\_selected\_arrow(self) -> ArrowGridSquare:  
 *"""  
 Return selected arrow object if any were selected  
  
 :return: Arrow object that is selected if any  
 """* for arrow in self.arrows:  
 if arrow.selected:  
 return arrow  
  
 def dehighlight\_errors(self) -> None:  
 *"""  
 Get rid of highlighting on numbers that don't match  
 """* for number in self.numbers:  
 if tuple(number.position.grid\_square\_pos) in self.wrong\_numbers:  
 number.dehighlight\_error()  
  
 def highlight\_errors(self) -> None:  
 *"""  
 Highlight numbers that don't match  
 """* for number in self.numbers:  
 if tuple(number.position.grid\_square\_pos) in self.wrong\_numbers:  
 number.highlight\_error()  
  
 def check\_correctness(self) -> None:  
 *"""  
 Load current arrows and numbers to core class and evaluate correctness  
 """* for arrow in self.arrows:  
 Core.arrows[arrow.arrow\_set][arrow.arrow\_num] = arrow.direction  
 for number in self.numbers:  
 Core.numbers[number.col][number.row] = number.value  
 self.wrong\_numbers = Core.evaluate\_correctness()  
  
 def get\_arrow(self**,** pos: tuple[int**,** int]) -> ArrowGridSquare:  
 *"""  
 Get arrow by pixel position  
  
 :return: arrow that is under given position  
 :param pos: position to get arrow by  
 """* for arrow in self.arrows:  
 if arrow.rect.collidepoint(pos):  
 return arrow  
  
 def set\_arrow\_image(self**,** image: pygame.Surface | None**,** direction: tuple[int**,** int] | None**,** highlight\_color: tuple[int]) -> None:  
 *"""  
 Set arrow image for selected arrow square if any  
  
 :return: None  
 :param image: New image to set  
 :param direction: direcion of arrow on image to set direction attribute  
 :param highlight\_color: color that object was highlighted to restore it  
 """* for arrow in self.arrows:  
 if arrow.selected:  
 arrow.set\_image(image**,** direction)  
 arrow.select(highlight\_color)  
  
 def check\_arrow\_selection(self**,** mouse\_pos: tuple[int**,** int]) -> bool:  
 *"""  
 Check if arrow was selected  
  
 :return: True if arrow under current mouse position is selected  
 :param mouse\_pos: Mouse position in pixel coordinates  
 """* for arrow in self.arrows:  
 if arrow.rect.collidepoint(mouse\_pos):  
 return True  
 return False  
  
 def check\_number\_selection(self**,** mouse\_pos: tuple[int**,** int]) -> bool:  
 *"""  
 Check if number was selected  
  
 :return: True if number under current mouse position is selected  
 :param mouse\_pos: Mouse position in pixel coordinates  
 """* for number in self.numbers:  
 if number.rect.collidepoint(mouse\_pos):  
 return True  
 return False  
  
 def handle\_arrow\_selection(self**,** mouse\_pos: tuple[int**,** int]) -> None:  
 *"""  
 Select arrow and numbers it points to  
  
 :return: None  
 :param mouse\_pos: Current mouse position  
 """* # deselect previously selected numbers and its arrows  
 if isinstance(self.currently\_selected**,** NumberGridSquare):  
 pointing\_arrows = Core.get\_pointings(Vector2(self.currently\_selected.col**,** self.currently\_selected.row))  
 for arrow in self.arrows:  
 if (arrow.arrow\_set**,** arrow.arrow\_num) in pointing\_arrows:  
 arrow.deselect(colors.highlighted\_yellow)  
 self.currently\_selected.deselect(colors.highlighted\_blue)  
 self.currently\_selected = None  
  
 # deselect previous arrow and its spanned numbers  
 if isinstance(self.currently\_selected**,** ArrowGridSquare):  
 for number in self.numbers:  
 if number.selected:  
 number.deselect(colors.highlighted\_yellow)  
 self.currently\_selected.deselect(colors.highlighted\_blue)  
  
 # get currently selected arrows  
 curr\_arrow = None  
 for arrow in self.arrows:  
 if arrow.rect.collidepoint(mouse\_pos):  
 curr\_arrow = arrow  
 break  
  
 # select current arrow and spanned numbers  
 if curr\_arrow != self.currently\_selected:  
 pointed\_numbers = Core.get\_span(curr\_arrow.position.grid\_square\_pos**,** curr\_arrow.direction)  
 for number in self.numbers:  
 if (number.col**,** number.row) in pointed\_numbers:  
 number.select(colors.highlighted\_yellow)  
 curr\_arrow.select(colors.highlighted\_blue)  
 self.currently\_selected = curr\_arrow  
 else:  
 self.currently\_selected = None  
  
 def handle\_number\_selection(self**,** mouse\_pos: tuple[int**,** int]):  
 *"""  
 Select number and arrows that point to it  
  
 :return: None  
 :param mouse\_pos: Current mouse position  
 """* # deselect previously selected arrow and numbers that it points to  
 if isinstance(self.currently\_selected**,** ArrowGridSquare):  
 pointing\_numbers = Core.get\_span(self.currently\_selected.position.grid\_square\_pos**,** self.currently\_selected.direction)  
 for number in self.numbers:  
 if (number.col**,** number.row) in pointing\_numbers:  
 number.deselect(colors.highlighted\_yellow)  
 self.currently\_selected.deselect(colors.highlighted\_blue)  
 self.currently\_selected = None  
  
 # deselect previous number and all selected arrows if there is  
 if isinstance(self.currently\_selected**,** NumberGridSquare):  
 for arrow in self.arrows:  
 if arrow.selected:  
 arrow.deselect(colors.highlighted\_yellow)  
 self.currently\_selected.deselect(colors.highlighted\_blue)  
  
 # get previously selected and currently selected numbers  
 curr\_num = None  
 for number in self.numbers:  
 if number.rect.collidepoint(mouse\_pos):  
 curr\_num = number  
 break  
  
 # select current number and arrows if there is  
 for arrow in self.arrows:  
 Core.arrows[arrow.arrow\_set][arrow.arrow\_num] = arrow.direction  
 if curr\_num != self.currently\_selected:  
 pointed\_arrows = Core.get\_pointings(Vector2(curr\_num.col**,** curr\_num.row))  
 for arrow in self.arrows:  
 if (arrow.arrow\_set**,** arrow.arrow\_num) in pointed\_arrows:  
 arrow.select(colors.highlighted\_yellow)  
 curr\_num.select(colors.highlighted\_blue)  
 self.currently\_selected = curr\_num  
 else:  
 self.currently\_selected = None  
  
 def draw(self) -> None:  
 *"""  
 Draw object to given surface  
 """* # draw numbers grid squares  
 for number in self.numbers:  
 number.draw()  
  
 # draw arrows grid squares  
 for arrow in self.arrows:  
 arrow.draw()  
  
 # draw frame  
 pygame.draw.rect(Screen.surface**,** self.frame\_color**,** self.frame\_rect**,** self.frame\_width)

**number.py**

import pygame  
  
from control import colors  
from control.settings import Settings  
  
  
class Number:  
 *"""Class for number image to render on game board"""* def \_\_init\_\_(self**,** value: int):  
 *"""  
 :return: None  
 :param value: Numeric value of number object  
 """* # get text font and related attributes  
 self.color = colors.numbers  
 self.font = pygame.font.SysFont('BAUHS93'**,** Settings.number\_size)  
  
 # render image  
 self.image = self.font.render(str(value)**,** True**,** self.color**,** colors.background)