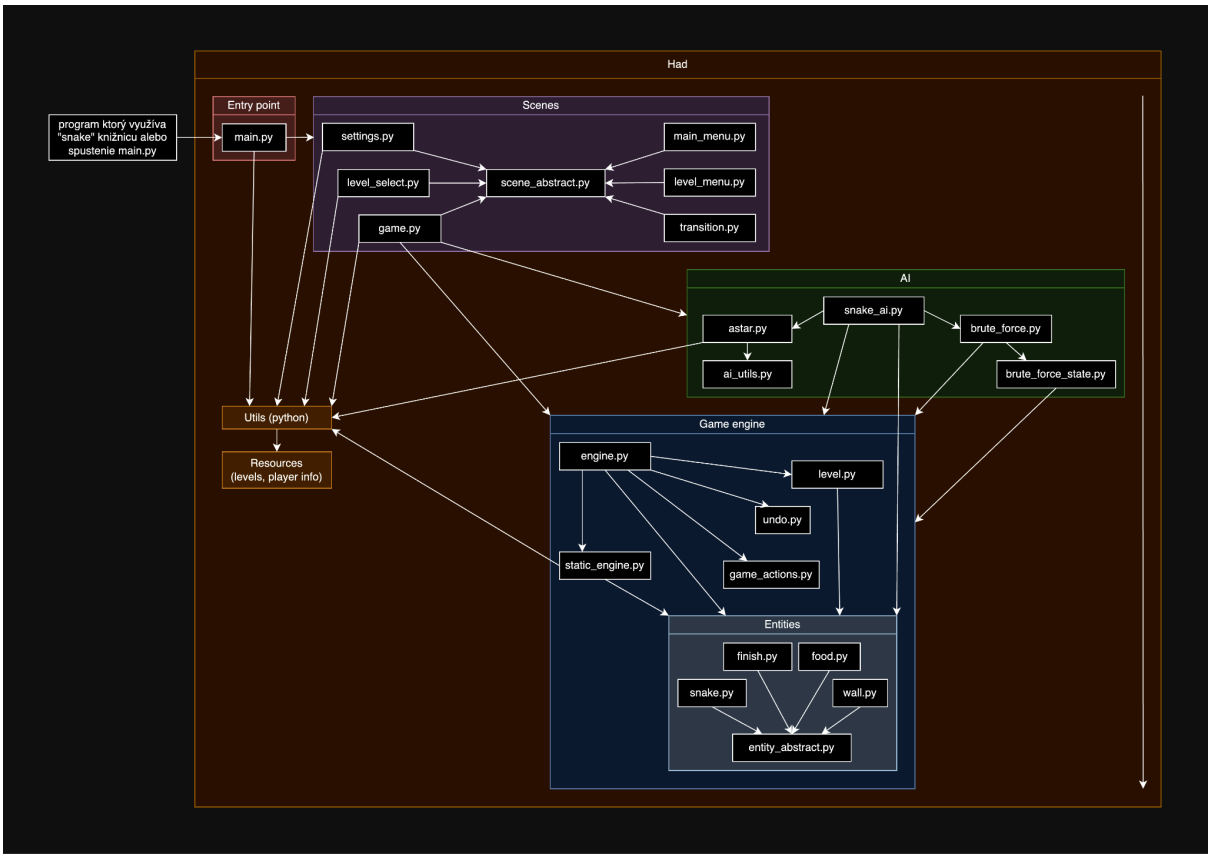
Technická dokumentácia

Had

Prehľad

Aplikácia had sa skladá z viacerých modulov kde každý zabezpečuje časť funkcionality celej aplikácie. Takto vyzerá jej štruktúra v grafe:



A → B na tomto grafe znamená že A používa/importuje B

Toto robia jednotlivé časti:

- *main.py* je hlavný riadiaci bod aplikácie, stará sa o to aby aplikácia mala kde ukazovať výstup, aby sa správne menila veľkosť okna aplikácie, aby sa správne spracoval vstup používateľa, aby sa na výstupe ukazovalo všetko potrebné a spravuje komunikáciu medzi jednotlivými scénami.
- *scény* sú jednotlivé režimy ktoré práve aplikácia vykonáva medzi ktorými sa dá ľahko prepínať ako napríklad *menu*, *hra*, *nastavenia.* Každá scéna riadi niečo iné a má svoje vlastné spracovanie a grafický výstup. Najdôležitejšia je scéna `scenes.Game` ktorá umožňuje používateľovi hrať hada.
- *umelá "inteligencia"* riadi *autoplay* režim aplikácie - teda snaží sa nájsť cestu pre hada do cieľa pomocou rôznych algoritmov bez zásahu uživateľa.
- *game engine* (to neviem ako by som preložil) riadi beh hry, najdôležitejší je tu objekt `game_engine.Engine` ktorý dostáva ako vstup pohyb hada (nerieši či od človeka alebo od algoritmu) a spracuje ho v rámci pravidiel hry. Tiež dôležitý je tu objekt

`game_engine.Engine` ktorý pri spustení levelu predspracuje stav hry čo umožňuje jednoduchý a rýchly výpočet napríklad kolízií
- *entity* sú najmenšie stavebné bloky interného riadenia hry - sú to jednotlivé prvky hry ako napríklad *had*, *stena*, *jedlo*, tu sa nič zaujímavé z technickej stránky nedeje.
- *utils* rieši doplnkovú funkcionalitu ktorá mi nikde inde nesedela. Napríklad načítavanie levelu zo súboru, načítavanie a ukladanie dát medzi spusteniami a rozdeľovanie objektov na komponenty podľa spojitosti
- *resources v sebe nemá žiadny kód ale obsahuje definície jednotlivých levelov a ukladajú sa tam informácie o používateľovi aby bola aplikácia v rovnakom stave ako ju používateľ zanechal*

dopodrobna budú jednotlivé časti vysvetlené v opačnom poradí - v takom ako program dané objekty definuje aby sa na nich dalo potom stavať.

Varovanie

V tomto súbore budem písať o veciach ako elektrika, ktorá má vlastné pravidlá v rámci hada alebo napríklad entity ktoré môžu hada zraniť a v tomto bode to do hada ešte vôbec nie je implementovaná takže to netreba moc riešiť.

Resources

Sú tu dva druhy súborov
1) .hadik - V tomto súbore sú uložené všetky dáta ktoré je potrebné na spustenie levelu takže jeho šírka, výška, camera offset, začiatočná pozícia hada a všetky entity
2) .save - V tomto súbore sú uložené zvolené nastavenia používateľa medzi spusteniami aplikácie čo sú fullscreen a autoplay možnosti. Tiež je tu uložené ktoré levely už sú odomknuté lebo na začiatku je odomknutý len prvý level.

Utils

player_data.py

Definuje triedu `PlayerData` ktorá má v sebe informácie o momentálnom stave fullscreenu, autoplay režimu a odomknutých leveloch, jej prvá metóda `load()` načíta dáta zo súboru player_data.save ktorý sa nachádza v snake/resources. Jej druhá metóda `save()` do tohoto súboru zase všetky aktuálne dáta uloží.

load_level.py

Implementuje jedinú funkciu:

```
load_level(level_number) -> tuple[Level, int, int]
```

Táto funkcia prečíta súbor `f"{level_number}.hadik"` ktorý sa nachádza v snake/resources a vráti inicializovaný level ktorý sa dá použiť na hranie hada. Tie zvyšné

dva inty čo vracia sú camera offset x, y pretože tie nesúvisia s hraním daného levelu, len s jeho ukazovaním na obrazovku. Súbory *.hadik majú svoju špecifickú štruktúru ktorá musí byť dodržaná a momentálne nie je žiadny rozumný spôsob ako si spraviť svoj level.

group.py

Implementuje dve funkcie pre rozdeľovanie mnoho vecí na menšie skupiny podľa toho ktoré sú spojené.

```
get_connected_blocks(blocks: list[tuple[int, int]]) ->
list[list[tuple[int, int]]]:
```

je klasický algoritmus na rozdeľovanie 2D grafu na komponenty podľa spojitosti. Nič zvláštne. Používa ho umelá “inteligencia” na hľadanie cesty pre hada, napríklad iný algoritmus vráti 20 možných pozícií na ktoré teoreticky vie had z momentálnej pozície prejsť ale keď sú rozdelené (povedzme stenou) tak had vie ísť len na pozície z tej skupiny kde sa sám nachádza.

```
get_connected_conductive_groups(entities: list[StaticEntity]) ->
list[list[StaticEntity]]:
```

Robí zhruba to isté ale využíva vlastnú logiku v hadovi na rozdelenie statických objektov v leveli na skupiny ktoré sú navzájom vodivé, to znamená že sú dosť blízko seba na to aby zdieľali elektrický náboj. Túto funkciu používa statický engine na predspracovanie levelu na elektrické grupy čo bude asi vysvetlené neskôr.

Entity

Entity existujú vo vlastnom súradnicovom systéme hry ktorý sa využíva na beh hry (každá kocka je jedna súradnica) a pri vykresľovaní sa ich súradnice menia na súradnice pixelov na plátne canvas. Prvá súradnica je vždy osa x a druhá je vždy osa y s tým že bod (0, 0) je v ľavom hornom rohu.

entity_abstract.py

Definuje materskú triedu `Entity` z ktorej dedia všetky ostatné entity, definuje všeobecné správanie entity. Všetky jej metódy sú abstraktné aby sa zabezpečilo že z nej zdedené entity budú implementovať všetko čo by entita mala vedieť. Má tieto metódy:

```
__init__(self, conductive: bool, charge: bool)
```

Parametre určujú či táto entita je vodivá a ak hej, či je hneď nabitá elektrikou.

```
draw(self, canvas: tkinter.Canvas, paddingx: int, paddingy: int,
block_size: int) -> None
```

Definuje ako bude entita vyzerať v hre, parametre *paddingx, paddingy* a *block_size* slúžia na konverziu z herných súradníc na súradnice plátna canvas.

```
get_collision_coords(self) -> list[tuple[int, int]]
```

Definuje či a ako je entita pevná. Nedá sa prejsť cez súradnice ktoré vráti táto funkcia.

```
get_hurt_coords(self) -> list[tuple[int, int]]
```
Definuje či a ako je entita nebezpečná. Prejdenie cez súradnice ktoré vráti táto funkcia odreže článok hada ktorý cez ne prešiel.
```
get_electricity_coords(self) -> list[tuple[int, int]]
```
Definuje či a ako entita šíri elektriku. Keď je entita nabitá elektrinou aj súradnice ktoré vráti táto funkcie sú nabité elektrinou.
```
get_interact_coords(self) -> list[tuple[int, int]]
```
Definuje či a ako sa dá interagovať s touto entitou. Keď je had na súradniciach ktoré vráti táto funkcia, niečo sa stane.
```
get_interact_type(self) -> "Entity.InteractType"
```
Definuje aká interakcia sa stane keď had interaguje s touto entitou. Možnosti sú enum typu `Entity.InteractType` ktorý má hodnoty `NONE`, `FINISH`, `FOOD`, `CHECKPOINT`.

Okrem toho tento súbor definuje aj dva typy entít ktoré dedia z `Entity`:

1) `StaticEntity` - entita ktorá má obdĺžnikový tvar a nemení pozíciu
2) `DynamicEntity` - entita ktorá môže mať ľubovoľný tvar a môže sa aj hýbať a môže na ne pôsobiť gravitácia

snake.py

Implementuje triedu `Snake` ktorá dedí z `scenes.DynamicEntity`. Je to entita ktorá má dynamický tvar, vedie elektrinu a pôsobí na ňu gravitácia. Je tvorená z fronty jednotlivých blokov aby bolo jednoduché pohybovať hadom tak ako sa had v hre pohybuje (chvost v predu sa každým pohybom popne a appendne sa nová hlava). Had je modrý keď ním neprechádza elektrika a zlatý keď ním prechádza.

wall.py

Implementuje triedu `Wall` ktorá dedí zo `scenes.StaticEntity`. Je to stena ktorá vedie elektrinu a had cez ňu neprejde.

food.py

Implementuje triedu `Food` ktorá dedí zo `scenes.StaticEntity`. Je to jedlo ktoré nevedie elektrinu a predĺži hada o 1 článok.

finish.py

Implementuje triedu `Finish` ktorá dedí zo `scenes.StaticEntity`. Je to cieľ, ktorého dosiahnutie úspešne ukončuje level.

Game engine

undo.py

Implementuje triedy `EatenFood`, `EntityPosition` a `Undo`. `Undo` je hlavná trieda ktorá na svoj beh využíva zvyšné dve triedy.

Trieda `EatenFood` si pamätá event ktorý sa stal (eventuálne by eventov malo byť viac ako len jedenie jedla) a kde sa stal, teda (x, y) súradnice kde jedlo bolo zjedené.

Trieda `EntityPosition` si pamätá pozíciu nejakej dynamickej entity. Dynamické entity okrem samotného hada môžu mať pestré tvary ale keďže sa ich tvar počas behu levelu nemení stačí si pamätať ich (x, y) súradnice. Tiež si ukladá referenciu na samotnú entitu aby mohla zmeniť jej súradnice v prípade že treba ísť naspäť v čase.

Trieda `Undo` si pamätá jeden frame vecí ktoré sa v hadovi stali, normálne je vracanie sa naspäť v čase možné len v debug móde ale umelá “inteligencia” ho využíva na brute force algoritmus takže je to potrebné. Táto trieda má 3 hodnoty:

```
snake: collections.deque[tuple[int, int]]
```

Keďže had vie meniť svoj tvar pamätá si undo pozíciu všetkých jeho článkov v danom frame.

```
dynamic_entities: list[EntityPosition]
```

Pamätá si pozíciu všetkých dynamických entít v danom frame.

```
events: list[EatenFood]
```

Pamätá si eventy ktoré sa v danom frame stali, zatiaľ len jedenie jedla.

game_actions.py

Implementuje jednoduchý enum `Action` všetkých inputov ktoré game engine pozná. Každý frame game engine dostane jeden z týchto inputov a na základe toho beží celá hra. Táto trieda predstavuje interface cez ktorý aplikácia hovorí game enginu čo má robiť.
Má tieto hodnoty:
`DO_NOTHING` - Had sa nehýbe, jedine že by padal kvôli gravitácii
`MOVE_LEFT` - Skús pohnúť hadom o jedno políčko doľava
`MOVE_RIGHT` - Skús pohnúť hadom o jedno políčko doľava
`MOVE_UP` - Skús pohnúť hadom o jedno políčko dohora
`MOVE_DOWN` - Skús pohnúť hadom o jedno políčko dolu
`STOP_MOVEMENT` - Zastav automatické procesy ako padanie. Povolené len v debug režime.
`UNDO_MOVEMENT` - Vráť sa o jeden frame naspäť. Povolené len pre umelú “inteligenciu” alebo v debug režime.

level.py

Implementuje jednoduchú triedu `Level` ktorá uchováva všetky informácie o momentálnom stave levelu. Má hodnoty pre výšku, šírku levelu, pozíciu a veľkosť hada (fronta jeho

článkov) a samostatné hodnoty pre statické a dynamické entity. Statické entity sa spracujú na začiatku levelu a ďalej sa len vykresľujú, dynamické entity sa spracúvajú každý frame.

static_engine.py

Implementuje statické spracovanie hry pre jednoduché implementovanie algoritmu ktorý hrá hada. Tvorí interface medzi vytvorenými entitami (ktoré majú napríklad x, y súradnice) a statickým stavom hry kde sú ku x, y súradniciam v hre priradené stavy ktoré sa na týchto súradniciach dejú ako napríklad či tam je stena alebo jedlo. Celé to riadi trieda `StaticEngine` ktorá interne priraďuje ku súradniciam hry indexy objektu `InteractionGroup`. Objekt `InteractionGroup` zase vie o akú interakciu sa jedná a prípadne aj ktoré všetky entity jej patria, napríklad existuje vždy `InteractionGroup` s indexom 1 ktorá zahŕňa všetky steny takže keď je na pozícií (2, 3) index 1, znamená to že na tejto pozícií je stena a nedá sa cez ňu prejsť. Iné `InteractionGroup` objekty sa tvoria vždy podľa levelu, napríklad všetky steny ktoré zdieľajú elektrický náboj patria do jednej grupy ktorá sa tým pádom dá celá zelektrizovať. Konštruktor triedy `StaticEngine` berie ako parameter `list[game_engine.entities.StaticEntity])` a inicializuje tieto priradenia (technicky sú to dictionary) a po vytvorení ju treba len málo updatovať, napríklad keď had zje jedlo. Pre získanie stavu nejakej súradnice stačí na takto inicializovaný static engine zavolať

```
get_interactions(self, x: int, y: int) -> set[Interaction]
```

možné typy interakcií sú `NOTHING`, `WALL`, `HAZARD`, `CHARGE`, `FINISH`, `FOOD`.

engine.py

Implementuje triedu `Engine` ktorá riadi beh celej hry podľa pravidiel “fyziky” v hre. Konštruktor triedy `Engine` berie ako parameter `game_engine.Level` a inicializuje všetko potrebné, ako napríklad aj svoj `game_engine.StaticEngine` ktorý by nemal byť prístupný mimo Engine. Potom pre postup hry treba volať jej metódu `process_frame(self, action: Action) -> None`
ktorá spracuje jeden frame hry a pokúsi sa splniť inštrukcie ktoré dostala na parametri `action`. `Engine` si necháva referenciu na level ktorý dostal pri inicializácií takže momentálny stav hry sa dá nájsť potom priamo v tej premennej level, okrem toho sa dajú po každom `process_frame(...)` prečítať stavové premenné `Engine` aby hra vedela čo sa v poslednom frame stalo:

```
level_finished: bool
```

True ak had prišiel úspešne do cieľa, nemá zmysel procesovať viac framov

```
movement_happened: bool
```

True ak sa had pohol tento frame, dôležité kvôli správnemu nastaveniu camera offsetu ktorý nerieši samotný `Engine` ale trieda ktorá displayuje samotný level pretože offset nie je založený len na inputoch, napríklad keď had padá bolo by fajn keby ho kamera nasledovala.

```
last_movement: Action
```

Ak je `Engine.movement_happened` True tak tu sa dá nájsť pohyb hada ktorý nastal predošlý frame.

Umelá “inteligencia”

V každom leveli je cieľ prejsť od začiatku do cieľa, umelá “inteligencia” sa snaží nájsť správne poradie inputov ktoré toto dosiahnu. Preto všade píšem inteligencia v úvodzovkách lebo je to viac pokus omyl ako nejaká inteligentná heuristika - to vôbec neprekáža.

Ako to funguje?

Algoritmus ktorý hľadá správne poradie inputov je rozdelený na fázy kde každá rozdelí cestu na menšie kúsky pre ktoré sa cesta hľadá jednoduchšie.

1) Hlavná logika je že pravdepodobne je treba nájsť všetko jedlo pretože je pravdepodobne potrebné pre dokončenie levelu, rozdelí teda level na cesty od začiatku ku vždy najbližšiemu jedlu a keď je všetko jedlo zjedené tak do cieľa.
2) Teraz už máme jasný cieľ (napríklad najbližšie jedlo alebo priamo cieľ). Tento krok staticky nájde “medzikroky” na ktoré sa musí had dostať, jeden medzikrok je tak veľký že had sa naňho vie dostať bez toho aby všetkými článkami opustil kocku na ktorej stojí.
3) Teraz je cesta rozdelená na fakt malé cesty ktoré nie sú ďaľej od seba ako momentálna dĺžka hada. V tomto kroku sa cesta nehľadá staticky ale brute force prehľadávaním stavového priestoru (vstupy sa zadávajú do `game_engine.Engine` a len sa sleduje či bol cieľ úspešne dosiahnutý), kým nie sú implementované zložitejšie objekty ako krabice a elektrika môže tento krok vyzerať zbytočný ale keď budú statická analýza by už nestačila na nájdenie správnej cesty.
4) Po úspešnom dojdení do cieľa je možné zopakovať celú správnu cestu od začiatku až po cieľ bez zbytočných zatáčok - vyzerá to pekne pre používateľa.

brute_force_state.py

Tento súbor implementuje triedu `State` ktorá určuje stav jedného políčka hry pri prehľadávaní stavového priestoru. Má tieto premenné:

```
coords: tuple[int, int]
```

x, y súradnice tohoto stavu

```
depth: int
```

Hĺbka rekurzie prehľadávania stavového priestoru v tomto stave,

```
moves: list[tuple[tuple[int, int], Action]]
```

List možných pohybov ktoré had ešte z tejto pozície neskúsil, zoradený podľa heuristiky, každý možný pohyb je dvojica súradníc na ktoré by sa had týmto pohybom dostal a `game_engine.Action` ktorá by ho tam dostala a treba ju zadať do game engine aby sa tam had dostal.

Tieto premenné sa inicializujú v konštruktore ktorý berie takéto parametre:

```
coords: tuple[int, int]
```

x, y súradnice tohoto stavu v hre

```
parent: tuple[int, int] | None
```
x, y súradnice materského stavu odkiaľ sa dá sem dostať
```
destination: tuple[int, int]
```
x, y súradnice políčka na ktoré sa chce had dostať kvôli heuristike
```
engine: game_engine.StaticEngine
```
Statický engine ktorý umožňuje skontrolovať súradnice na ktoré sa dá dostať
```
move_up: bool
```
Či je možné z tohoto stavu ísť vyššie (kvôli tomu aby had neskúšal ísť priamo hore čo sa kvôli gravitácií nedá
```
max_width: int
```
Šírka levelu
```
max_height: int
```
Výška levelu
```
depth: int
```
Hĺbka rekurzie prehľadávania stavového priestoru v tomto stave
```
max_depth: int
```
Maximálna povolená hĺbka rekurzie

brute_force.py

Implementuje bod 3) zo sekcie “Ako to funguje?” čiže prehľadávanie stavového priestoru hada brute force skúšaním vstupov do `game_engine.Engine`. Prehľadávanie je kvôli tomuto možné len do hĺbky. Keď používame brute force je už zaručené že blok na ktorý had potrebuje ísť je dosť blízko aby bola hĺbka rekurzie obmedzená dĺžkou hada.

Implementácia je pomocou triedy `FindPathForce` ktorú stačí inicializovať, potom si už internú logiku prehľadávania stavového priestoru rieši sama. Vždy dostaneme ďalší pohyb ktorý AI chce skúsiť jej metódou:
```
get_next_move(self) -> game_engine.Action
```
Buď vráti nejaký smer ktorým sa had má pohnúť alebo `game_engine.Action.UNDO_MOVEMENT` keď chce ísť o jednu hĺbku menej v prehľadávaní stavového priestoru. Konštruktor `FindPathForce` berie takéto parametre:
```
start: tuple[int, int]
```
x, y súradnice počiatočného bodu prehľadávania stavového priestoru
```
destination: tuple[int, int]
```
x, y súradnice cieľového bodu prehľadávania stavového priestoru
```
engine: game_engine.StaticEngine
```
Statický engine ktorý umožňuje skontrolovať súradnice na ktoré sa dá dostať

```
snake_length: int
```
Dĺžka hada ktorá určuje možnú hĺbku rekurzie
```
level_width: int
```
Šírka levelu
```
level_height: int
```
Výška levelu

ai_utils.py

Implementuje funkciu:
```
get_reach(current: tuple[int, int], engine: game_engine.StaticEngine , lenght: int, level_width: int, level_height: int) -> list[tuple[int, int]]
```
Ktorá vracia všetky možné pozície na ktoré sa had môže dostať zo svojej momentálnej polohy bez toho aby všetky jeho bloky odišli z kocky na ktorej práve stojí. Využíva sa pre bod 2) zo sekcie “Ako to funguje?”

astar.py

Implementuje bod 2) zo sekcie “Ako to funguje?” čiže hľadá cestu pre hada ku niečomu čo môže byť ľubovoľne vzdialené, napríklad od začiatku ku jedlu, a rozdeľuje ju na menšie kroky kde sa dá jeden krok spraviť bez toho aby všetky články hada opustili kocku na ktorej had stojí.

Implementácia je pomocou triedy `FindPathStatic` ktorá dedí z triedy `AStar` z knižnice `astar`, čiže používa A* algoritmus na nájdenie cesty. Táto trieda sa inicializuje v konštruktore ktorý berie takéto parametre:
```
engine: game_engine.StaticEngine
```
Statický engine ktorý umožňuje skontrolovať súradnice na ktoré sa dá dostať
```
level_width: int
```
Šírka levelu
```
level_height: int
```
Výška levelu

Po inicializácií triedu treba aktualizovať len ak had zmení veľkosť a to metódou:
```
update_lenght(self, snake_lenght: int) -> None
```
Keď je trieda inicializovaná, nájde jednotlivé kroky cesty ku zadanému cieľu pomocou metódy:
```
astar(self, start: tuple[int, int], goal: tuple[int, int], reversePath=False) -> collections.deque[tuple[int, int]]
```

snake_ai.py

Implementuje bod 1) a 4) zo sekcie “Ako to funguje?” čiže hlavnú logiku ktorá spája všetky algoritmy dokopy a hovorí im kde sa majú chcieť dostať a kde sa had práve nachádza. Po úspešnom nájdení input sa dá získať celá fronta správnych inputov.

Implementácia je pomocou triedy `SnakeAI` kde po inicializácií vždy dostaneme ďalší pohyb ktorý AI chce skúsiť jej metódou:

```
get_next_move(self) -> game_engine.Action
```

Po každom zavolaní `get_next_move()` a následnom spracovaní tejto akcie enginom sa dá skontrolovať stavová premenná `SnakeAI`:

```
level_finished: bool
```

True znamená že AI našlo správne poradie inputov ktoré vyhrá daný level, následne sa dá získať správna fronta inputov v premennej:

```
final_path: collections.deque[game_engine.Action]
```

Trieda sa inicializuje v konštruktore ktorý berie takéto parametre:

```
level: game_engine.Level
```

Level v ktorom chceme hľadať cestu

```
engine: game_engine.StaticEngine
```

Statický engine ktorý umožňuje skontrolovať súradnice hry na ktoré sa dá dostať

Scény

scene_abstract.py

Definuje materskú triedu `Scene` z ktorej dedia všetky ostatné scény, definuje všeobecné správanie scény. Niektoré jej metódy sú abstraktné aby sa zabezpečilo že z nej zdedené scény budú implementovať všetko čo by entita mala vedieť. Má tieto metódy:

```
__init__(self, canvas: tkinter.Canvas, transparent: bool)
```

Každá scéna potrebuje referenciu na plátno canvas aby mala kde ukazovať output. Keď je scéna transparentná znamená to že nezaberá celú obrazovku a je možné vykresľovať ju na vrch scény pod ňou.

```
process_frame(self, key_press: KeyboardInput | None) -> None
```

Definuje logiku spracovania uživateľského inputu danej scény. `KeyboardInput` je enum ktorý môže mať hodnoty `ESCAPE`, `ENTER`, `UP`, `DOWN`, `RIGHT`, `LEFT`, `UNDO`, `STOP_MOVEMENT`. Tiež sa definuje v tomto súbore.

```
display_frame(self, paddingx: int, paddingy: int, screen_size: int) ->
None
```

Definuje grafický output tejto scény každý frame. Padding x, y hovoria s odstupom koľko pixelov treba scénu kresliť, *screen_size* je šírka aj výška (vždy štvorcovej) scény.

```
normalize_to_frame(x, y, paddingx, paddingy, screen_size)
```

Táto metóda je statická, využíva sa v `display_frame(...)` na vykreslenie inputu správne na obrazovku. parametre x, y sú súradnice ktoré treba normalizovať, zvyšné parametre sú tie isté ako v `display_frame(...)`

game.py

Implementuje scénu `Game` ktorá riadi beh hry pomocou `game_engine.Engine` ktorý inicializuje načítaným levelom a tiež počíta camera offset levelu aby bol intuitívny a pekný. Ukazuje momentálny stav levelu a ak je zapnutý debug režim ukazuje aj navyše technické informácie o behu hry.

Má takéto režimy:

1) Level hrá uživateľ - zadáva enginu vstupy ktoré stláča uživateľ na klávesnici
2) Umelá “inteligencia” sa snaží nájst správnu cestu - zadáva enginu také vstupy ktoré mu umelá “inteligencia” povie
3) Umelá “inteligencia” už našla cestu - pomaly znovu ukazuje celú správnu cestu od začiatku až do cieľu

Jej konštruktor má okrem canvas aj parametre:

```
level_number: int
```

Index levelu ktorý sa má načítať a hrať

```
autoplay: bool
```

Či hada ovláda umelá “inteligencia” alebo uživateľ

```
debug: bool
```

Či je zapnutý debug režim

main_menu.py

Implementuje scénu `MainMenu` ktorá riadi chod hlavného menu. Dá sa tu zapnúť prvý level, otvoriť menu výberu levelov, otvoriť nastavenia alebo vypnúť hru.

level_menu.py

Implementuje scénu `LevelMenu` ktorá slúži ako menu keď je zapnutá hra. Dá sa v ňom opustiť level, reštartovať level alebo otvoriť nastavenia.

level_select.py

Implementuje scénu `LevelSelect` kde si môže užívateľ vybrať ktorý level chce hrať.

Jej konštruktor má okrem canvas aj parameter:

```
player_data: utils.PlayerData
```
Aby vedela ktoré levely sú už odomknuté

settings.py

Implementuje scénu `Settings` kde môže uživateľ zmeniť nastavenia, čiže fullscreen a režim autoplay.

Jej konštruktor má okrem canvas aj parameter:
```
player_data: utils.PlayerData
```
V ktorom sú momentálne nastavenia ktoré si necháva ako referenciu a mení ich počas behu.

transition.py

Implementuje scénu `Transition` ktorá nepríjma žiadny uživateľský vstup ale sama sa po nejakom čase vypne. Ukazuje špirálový prechod medzi dvomi inými scénami alebo prechod z hlavného menu do vypnutia aplikácie.

Jej konštruktor má okrem canvas aj parametre:
```
type: Transition.Type
```
Ktorý typ prechodu sa má ukázať, možnosti sú `GENERIC_FIRST_HALF`, `GENERIC_SECOND_HALF`, `START_LEVEL_FIRST_HALF`, `START_LEVEL_SECOND_HALF`, `END_APPLICATION`
```
level_number: int | None
```
Ak je typ aplikácie `START_LEVEL_FIRST_HALF` alebo `START_LEVEL_SECOND_HALF` tak počas prechodu ukazuje aj číslo levelu ktoré získava z tohoto parametru.

Aplikácia (main.py)

Implementuje triedu `SnakeApplication` ktorá je hlavný riadiaci bod celej aplikácie, ak je tento súbor spustený tak aplikáciu priamo aj spustí.

Trieda `SnakeApplication` je interface medzi vstupom, spracovaním vnútornej logiky a výstupom aplikácie. Tvorí tkinter výstup (Tk a Canvas) a riadi komunikáciu medzi naozajstnou veľkosťou aplikácie a interným kreslením na canvas. Má zásobník scén ktorým posiela vstup uživateľa a ukazuje ich výstup každý frame (60 FPS). Každý frame sleduje či scéna na vrchu zásobníku ešte stále beží, keď nie, zariadi aby na zásobníku boli také scény aké tam majú byť.

Aplikácia sa inicializuje konštruktorom ktorý berie takéto parametre:
```
window_size: int = 700
force_fullscreen: bool = False
force_autoplay: bool = False
```

```
debug: bool = False
```
Čo robia sa dá nájsť v uživateľskej dokumentácií.

Aplikácia sa po inicializácií spúšťa metódou:
```
run(self) -> None
```

Záver

Keby boli nejaké ďalšie otázky na fungovanie kódu, rád ich zodpoviem!!