**Code**

main.py

from os import system

from nea\_game.config import NeaGameConfig

if \_\_name\_\_ == "\_\_main\_\_":

    config = NeaGameConfig()

    if config.debug:

        system(f"python app.py --fname {config.debug\_file}")

    else:

        system("python app.py")

app.py

from nea\_game.nea\_game import NeaGame

from nea\_game.config import NeaGameConfig

def app():

    nea\_game = NeaGame(config)

    running = True

    while running:

        nea\_game.update()

if \_\_name\_\_ == "\_\_main\_\_":

    config = NeaGameConfig()

    app()

base\_entity.py

class BaseEntity:

    x: float

    y: float

    def \_\_init\_\_(self, position: tuple[int, int]):

        self.x, self.y = position

    def update(self, delta\_time: float):

        """Called each frame in order to update the entity"""

level\_finish.py

from pygame import Rect

from nea\_game.entity.base\_entity import BaseEntity

class LevelFinish(BaseEntity):

    """Creates an entity to assist with the transition of levels"""

    rect: Rect

    new\_world: bool

    next\_level\_identifier: str

    def \_\_init\_\_(

        self,

        position: tuple[int, int],

        height: int,

        width: int,

        new\_world: bool,

        next\_level\_identifier: str,

    ):

        super().\_\_init\_\_(position)

        self.rect = Rect(self.x, self.y, width, height)

        self.new\_world = new\_world

        self.next\_level\_identifier = next\_level\_identifier

player.py

import itertools

from os.path import isdir

from os import listdir

from pathlib import Path

from pygame.event import Event

from pygame.image import load

from pygame import Rect, Surface

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.input import Input

from nea\_game.components.renderer import AnimatedRenderer

from nea\_game.components.rigidbody2d import RigidBody2D

from nea\_game.entity.base\_entity import BaseEntity

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

from nea\_game.player.sub\_states.player\_dash\_state import PlayerDashState

from nea\_game.player.sub\_states.player\_idle\_state import PlayerIdleState

from nea\_game.player.sub\_states.player\_in\_air\_state import PlayerInAirState

from nea\_game.player.sub\_states.player\_jump\_state import PlayerJumpState

from nea\_game.player.sub\_states.player\_land\_state import PlayerLandState

from nea\_game.player.sub\_states.player\_run\_state import PlayerRunState

from nea\_game.player.sub\_states.player\_slide\_state import PlayerSlideState

from nea\_game.player.sub\_states.player\_wall\_jump\_state import PlayerWallJumpState

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.states.player\_state\_machine import StateMachine

class Player(BaseEntity):

    chunks: dict[tuple[int, int], list[LevelTile]]

    idle\_state: PlayerIdleState

    run\_state: PlayerRunState

    land\_state: PlayerLandState

    dash\_state: PlayerDashState

    jump\_state: PlayerJumpState

    wall\_jump\_state: PlayerWallJumpState

    in\_air\_state: PlayerInAirState

    slide\_state: PlayerSlideState

    renderer: AnimatedRenderer

    input\_: Input

    rigid\_body: RigidBody2D

    state\_machine: StateMachine

    direction: int

    rect: Rect

    old\_rect: Rect

    x\_run\_speed: float

    acceleration\_rate: float

    jump\_hang\_acceleration\_mult: float

    jump\_hang\_max\_speed\_mult: float

    deceleration\_rate: float

    air\_acceleration\_multiplier: float

    velocity\_power: float

    jump\_force: float

    jump\_hang\_time\_threshold: float

    jump\_hang\_gravity\_mult: float

    jump\_fast\_fall\_mult: float

    coyote\_time: float

    jump\_buffer\_time: float

    max\_fall: float

    land\_animation\_time: float

    wall\_jump\_force: Vector2D

    wall\_jump\_time: float

    wall\_jump\_lerp: float

    wall\_slide\_velocity: float

    can\_dash: bool

    dash\_time: float

    dash\_speed: float

    friction: float

    def \_\_init\_\_(

        self,

        player\_folder: Path,

        chunks: dict[tuple[int, int], list[LevelTile]],

        action\_bindings: list[int],

        internal\_fps: int,

        position: tuple[int, int],

    ):

        """Defines player movement constants and starts the player in the idle state

        Args:

            player\_folder (Path): The path to the player folder

            chunks (dict[tuple[int, int], list[LevelTile]]) A dictionary that stores every tile in the the level

            action\_bindings (list[int]): The list of the key bindings for the player actions

            level\_data (list[LevelTile]): The data of the level

            x (int): The x position of the player

            y (int): The y position of the player

        """

        super().\_\_init\_\_(position)

        self.chunks = chunks

        self.idle\_state = PlayerIdleState(self, "Idle")

        self.run\_state = PlayerRunState(self, "Run")

        self.land\_state = PlayerLandState(self, "Land")

        self.dash\_state = PlayerDashState(self, "Dash")

        self.jump\_state = PlayerJumpState(self, "Jump")

        self.wall\_jump\_state = PlayerWallJumpState(self, "WallJump")

        self.in\_air\_state = PlayerInAirState(self, "InAir")

        self.slide\_state = PlayerSlideState(self, "Slide")

        frames: dict[str, list[Surface]] = {}

        for folder in listdir(player\_folder):

            if isdir(player\_folder / folder):

                frames[folder] = [

                    load(player\_folder / (f"{folder}/{image\_name}"))

                    for image\_name in listdir(player\_folder / folder)

                ]

        self.renderer = AnimatedRenderer(frames)

        self.input\_ = Input(PlayerActionSpace, action\_bindings)

        self.rigid\_body = RigidBody2D(4, 0.4, internal\_fps)

        self.state\_machine = StateMachine(self.idle\_state)

        self.direction = 1

        self.rect = Rect(

            (self.x, self.y),

            self.renderer.frames[self.state\_machine.current\_state.state\_name][

                0

            ].get\_size(),

        )

        self.old\_rect = self.rect

        self.x\_run\_speed = 1.6

        self.acceleration\_rate = 3

        self.jump\_hang\_acceleration\_mult = 2

        self.jump\_hang\_max\_speed\_mult = 1.2

        self.deceleration\_rate = 10

        self.air\_acceleration\_multiplier = 0.6

        self.velocity\_power = 0.6

        self.jump\_force = 10

        self.jump\_hang\_time\_threshold = 0.5

        self.jump\_hang\_gravity\_mult = 0.6

        self.jump\_fast\_fall\_mult = 2

        self.coyote\_time = 0.1

        self.jump\_buffer\_time = 0.1

        self.max\_fall = 3

        self.land\_animation\_time = 0.15

        self.wall\_jump\_force = Vector2D(8, 12)

        self.wall\_jump\_time = 3

        self.wall\_jump\_lerp = 0.08

        self.wall\_slide\_velocity = 0.74

        self.can\_dash = False

        self.dash\_time = 0.1

        self.dash\_speed = 13

        self.friction = 10

    def get\_collisions(self) -> list[LevelTile]:

        """Returns a list of each LevelTile object that the player has collided with on the current frame

        Returns:

            list[LevelTile]: The list of LevelTile objects that the player has collided with on the current frame

        """

        return [

            tile

            for tile in itertools.chain(\*self.chunks.values())

            if self.rect.colliderect(tile.rect)

        ]

    def is\_alive(self, level\_height: int) -> bool:

        """A method to determine if the player is alive

        Args:

            level\_height (int): The height of the level in pixels

        Returns:

            bool: Whether the player is alive or not

        """

        if self.rect.y < 0 or self.rect.y > level\_height:

            return False

        for collision in self.get\_collisions():

            if collision.collision\_type == CollisionType.SPIKE:

                return False

        return True

    @property

    def is\_grounded(self) -> bool:

        """A method to determine if the player is grounded

        Returns:

            bool: Whether the player is alive or not

        """

        for collision in [

            tile

            for tile in itertools.chain(\*self.chunks.values())

            if self.rect.move(0, 1).colliderect(tile.rect)

        ]:

            if collision.collision\_type in (CollisionType.WALL, CollisionType.PLATFORM):

                if (

                    self.rect.bottom >= collision.rect.top

                    and self.old\_rect.bottom <= collision.rect.top

                ):

                    return True

        return False

    @property

    def is\_touching\_wall(self) -> int:

        """A method to determine if the player is touching a wall

        Returns:

            int: The direction of the wall relative to the player

        """

        for collision in [

            tile

            for tile in itertools.chain(\*self.chunks.values())

            if self.rect.move(1, 0).colliderect(tile.rect)

        ]:

            if collision.collision\_type == CollisionType.WALL:

                if (

                    self.rect.right >= collision.rect.left

                    and self.old\_rect.right <= collision.rect.left

                ):

                    return 1

        for collision in [

            tile

            for tile in itertools.chain(\*self.chunks.values())

            if self.rect.move(-1, 0).colliderect(tile.rect)

        ]:

            if collision.collision\_type == CollisionType.WALL:

                if (

                    self.rect.left <= collision.rect.right

                    and self.old\_rect.left >= collision.rect.right

                ):

                    return -1

        return 0

    def handle\_x\_collisions(self):

        """Changes the x position of the player according to any collisions on the x axis

        """

        for collision in self.get\_collisions():

            match collision.collision\_type:

                case CollisionType.WALL:

                    if (

                        self.rect.right >= collision.rect.left

                        and self.old\_rect.right <= collision.rect.left

                    ):

                        self.rect.right = collision.rect.left

                        self.x = self.rect.x

                        self.rigid\_body.velocity = Vector2D(

                            0, self.rigid\_body.velocity.y

                        )

                    if (

                        self.rect.left <= collision.rect.right

                        and self.old\_rect.left >= collision.rect.right

                    ):

                        self.rect.left = collision.rect.right

                        self.x = self.rect.x

                        self.rigid\_body.velocity = Vector2D(

                            0, self.rigid\_body.velocity.y

                        )

                case \_:

                    pass

    def handle\_y\_collisions(self):

        """Changes the y position of the player according to any collisions on the y axis

        """

        for collision in self.get\_collisions():

            match collision.collision\_type:

                case CollisionType.WALL:

                    if (

                        self.rect.bottom >= collision.rect.top

                        and self.old\_rect.bottom <= collision.rect.top

                    ):

                        self.rect.bottom = collision.rect.top

                        self.y = self.rect.y

                        self.rigid\_body.velocity = Vector2D(

                            self.rigid\_body.velocity.x, 0

                        )

                    if (

                        self.rect.top <= collision.rect.bottom

                        and self.old\_rect.top >= collision.rect.bottom

                    ):

                        self.rect.top = collision.rect.bottom

                        self.y = self.rect.y

                        self.rigid\_body.velocity = Vector2D(

                            self.rigid\_body.velocity.x, 0

                        )

                case CollisionType.PLATFORM:

                    if (

                        self.rect.bottom >= collision.rect.top

                        and self.old\_rect.bottom <= collision.rect.top

                    ):

                        self.rect.bottom = collision.rect.top

                        self.y = self.rect.y

                        self.rigid\_body.velocity = Vector2D(

                            self.rigid\_body.velocity.x, 0

                        )

                case \_:

                    pass

    def event\_handler(self, events: list[Event]):

        """Handles the events for the player

        Args:

            events (list[Event]): The list of events that have occurred on the current frame

        """

        self.input\_.update\_actions\_performed\_on\_current\_frame(events)

    def input\_handler(self):

        self.state\_machine.current\_state.input\_handler()

    def update(self, delta\_time: float):

        if self.input\_.get\_axis\_raw().x:

            self.direction = int(self.input\_.get\_axis\_raw().x)

        self.old\_rect = self.rect.copy()

        self.input\_handler()

        self.state\_machine.current\_state.update(delta\_time)

        self.x += self.rigid\_body.velocity.x

        self.rect.x = int(self.x)

        self.handle\_x\_collisions()

        self.y += self.rigid\_body.velocity.y

        self.rect.y = int(self.y)

        self.handle\_y\_collisions()

player\_state.py

from \_\_future\_\_ import annotations

import typing

from time import perf\_counter

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerState:

    """

    ###

    A: Complex OOP Model to model player states

    ###

    """

    player: Player

    state\_name: str

    animation\_index: int

    start\_time: float

    is\_exiting\_state: bool

    def \_\_init\_\_(self, player: Player, state\_name: str):

        """A template PlayerState class

        Args:

            player (Player): The entity that the PlayerState belongs to

            state\_name (str): The string representation of the PlayerState

        """

        self.player = player

        self.state\_name = state\_name

        self.animation\_index = 0

        self.is\_exiting\_state = False

    def enter(self):

        """Called when entering the PlayerState"""

        self.start\_time = perf\_counter()

        self.is\_exiting\_state = False

    def exit(self):

        """Called when leaving the PlayerState"""

        self.is\_exiting\_state = True

        self.animation\_index = 0

    def input\_handler(self):

        """Handles the inputs"""

    def update(self, delta\_time: float):

        """Called each frame"""

        if self.player.is\_grounded:

            self.player.can\_dash = True

player\_grounded\_state.py

from \_\_future\_\_ import annotations

from nea\_game.calc.vector2d import Vector2D

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.states.player\_state import PlayerState

class PlayerGroundedState(PlayerState):

    move\_input: Vector2D

    def input\_handler(self):

        ###A: Overriding the parent implementation of input\_handler###

        self.move\_input = self.player.input\_.get\_axis\_raw()

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        super().update(delta\_time)

        if (

            self.player.input\_.get\_action\_down(PlayerActionSpace.DASH)

            and self.player.can\_dash

        ):

            self.player.state\_machine.change\_state(self.player.dash\_state)

        if self.player.input\_.get\_action\_down(PlayerActionSpace.UP):

            self.player.state\_machine.change\_state(self.player.jump\_state)

        if not self.player.is\_grounded:

            self.player.state\_machine.change\_state(self.player.in\_air\_state)

player\_idle\_state.py

from nea\_game.calc.vector2d import Vector2D

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

class PlayerIdleState(PlayerGroundedState):

    def enter(self):

        ###A: Overriding the parent implementation of enter###

        super().enter()

        self.player.rigid\_body.velocity = Vector2D(0, self.player.rigid\_body.velocity.y)

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        super().update(delta\_time)

        if not self.is\_exiting\_state:

            if self.move\_input.x:

                self.player.state\_machine.change\_state(self.player.run\_state)

player\_run\_state.py

from \_\_future\_\_ import annotations

from time import perf\_counter

import typing

from nea\_game.calc.near\_zero import near\_zero

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerRunState(PlayerGroundedState):

    animation\_frame\_time: float

    last\_animation\_index\_change: float

    def \_\_init\_\_(self, player: Player, state\_name: str):

        super().\_\_init\_\_(player, state\_name)

        self.animation\_frame\_time = 0.05

        self.last\_animation\_index\_change = perf\_counter()

    def update(self, delta\_time: float):

        ###A: Overriding the implementation of update###

        super().update(delta\_time)

        if not self.is\_exiting\_state:

            if near\_zero(self.player.rigid\_body.velocity.x) and self.move\_input.x == 0:

                self.player.state\_machine.change\_state(self.player.idle\_state)

            else:

                if (

                    perf\_counter() - self.last\_animation\_index\_change

                    > self.animation\_frame\_time

                ):

                    self.animation\_index = (self.animation\_index + 1) % len(

                        self.player.renderer.frames[self.state\_name]

                    )

                    self.last\_animation\_index\_change = perf\_counter()

                ###A: User Defined complex alogithm###

                target\_speed = self.move\_input.x \* self.player.x\_run\_speed

                speed\_difference = target\_speed - self.player.rigid\_body.velocity.x

                acceleration\_rate = (

                    self.player.acceleration\_rate

                    if abs(target\_speed) > 0

                    else self.player.deceleration\_rate

                )

                movement = pow(

                    abs(speed\_difference) \* acceleration\_rate,

                    self.player.velocity\_power,

                ) \* sign(speed\_difference)

                self.player.rigid\_body.add\_force(

                    Vector2D(1, 0).scale(movement), delta\_time

                )

                if abs(target\_speed) == 0:

                    friction = sign(self.player.rigid\_body.velocity.x) \* min(

                        abs(self.player.rigid\_body.velocity.x), self.player.friction

                    )

                    self.player.rigid\_body.add\_force(

                        Vector2D(self.move\_input.x, 0).scale(friction),

                        force\_mode=ForceMode.IMPULSE,

                    )

player\_land\_state.py

from time import perf\_counter

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

class PlayerLandState(PlayerGroundedState):

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        super().update(delta\_time)

        if not self.is\_exiting\_state:

            if self.move\_input.x:

                self.player.state\_machine.change\_state(self.player.run\_state)

            elif perf\_counter() - self.start\_time > self.player.land\_animation\_time:

                self.player.state\_machine.change\_state(self.player.idle\_state)

player\_in\_air\_state.py

from \_\_future\_\_ import annotations

from time import perf\_counter

import typing

from nea\_game.calc.lerp import lerp

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

from nea\_game.states.player\_state import PlayerState

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerInAirState(PlayerState):

    move\_input: Vector2D

    jump\_input\_time: float

    def \_\_init\_\_(self, player: Player, state\_name: str):

        super().\_\_init\_\_(player, state\_name)

        self.jump\_input\_time = 0

    def input\_handler(self):

        ###A: Overriding the parent implementation of input\_handler###

        self.move\_input = self.player.input\_.get\_axis\_raw()

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        if (

            self.player.input\_.get\_action\_down(PlayerActionSpace.DASH)

            and self.player.can\_dash

        ):

            self.player.state\_machine.change\_state(self.player.dash\_state)

        elif self.player.input\_.get\_axis\_raw().x:

            if (

                self.player.is\_touching\_wall == self.player.input\_.get\_axis\_raw().x

                and self.player.rigid\_body.velocity.y > 0

            ):

                self.player.state\_machine.change\_state(self.player.slide\_state)

        # Coyote Time

        if (

            self.player.input\_.get\_action\_down(PlayerActionSpace.UP)

            and perf\_counter() - self.start\_time < self.player.coyote\_time

            and issubclass(

                type(self.player.state\_machine.previous\_state), PlayerGroundedState

            )

        ):

            self.player.state\_machine.change\_state(self.player.jump\_state)

        # Jump Buffering

        if self.player.input\_.get\_action\_down(PlayerActionSpace.UP):

            self.jump\_input\_time = perf\_counter()

        if self.player.is\_grounded:

            if perf\_counter() - self.jump\_input\_time < self.player.jump\_buffer\_time:

                self.player.state\_machine.change\_state(self.player.jump\_state)

            elif self.player.rigid\_body.velocity.y >= 0:

                self.player.state\_machine.change\_state(self.player.land\_state)

        else:

            target\_speed = self.move\_input.x \* self.player.x\_run\_speed

            if self.player.state\_machine.previous\_state == self.player.wall\_jump\_state:

                if (

                    perf\_counter() - self.player.state\_machine.previous\_state.start\_time

                    < self.player.wall\_jump\_time

                ):

                    target\_speed = lerp(

                        self.player.rigid\_body.velocity.x,

                        target\_speed,

                        self.player.wall\_jump\_lerp,

                    )

            speed\_difference = target\_speed - self.player.rigid\_body.velocity.x

            acceleration\_rate = (

                self.player.acceleration\_rate

                if abs(target\_speed) > 0

                else self.player.deceleration\_rate

            )

            if (

                self.player.state\_machine.previous\_state

                in (self.player.jump\_state, self.player.wall\_jump\_state)

                and abs(self.player.rigid\_body.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                acceleration\_rate \*= self.player.jump\_hang\_acceleration\_mult

                target\_speed \*= self.player.jump\_hang\_max\_speed\_mult

            movement = pow(

                abs(speed\_difference) \* acceleration\_rate, self.player.velocity\_power

            ) \* sign(speed\_difference)

            self.player.rigid\_body.add\_force(Vector2D(1, 0).scale(movement), delta\_time)

            if abs(target\_speed) == 0:

                friction = sign(self.player.rigid\_body.velocity.x) \* min(

                    abs(self.player.rigid\_body.velocity.x), self.player.friction

                )

                self.player.rigid\_body.add\_force(

                    Vector2D(1, 0).scale(friction), force\_mode=ForceMode.IMPULSE

                )

            if (

                self.player.state\_machine.previous\_state

                in (self.player.jump\_state, self.player.wall\_jump\_state)

                and abs(self.player.rigid\_body.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                gravity\_scale = (

                    self.player.rigid\_body.gravity\_scale

                    \* self.player.jump\_hang\_gravity\_mult

                )

            elif self.player.rigid\_body.velocity.y > 0:

                gravity\_scale = (

                    self.player.rigid\_body.gravity\_scale

                    \* self.player.jump\_fast\_fall\_mult

                )

            else:

                gravity\_scale = self.player.rigid\_body.gravity\_scale

            self.player.rigid\_body.add\_force(

                Vector2D(0, 1).scale(gravity\_scale),

                delta\_time,

                ForceMode.ACCELERATION,

            )

            self.player.rigid\_body.velocity = Vector2D(

                self.player.rigid\_body.velocity.x,

                min(self.player.rigid\_body.velocity.y, self.player.max\_fall),

            )

player\_slide\_state.py

from \_\_future\_\_ import annotations

from time import perf\_counter

import typing

from nea\_game.calc.lerp import lerp

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.player.super\_states.player\_grounded\_state import PlayerGroundedState

from nea\_game.states.player\_state import PlayerState

if typing.TYPE\_CHECKING:

    from nea\_game.player.player import Player

class PlayerInAirState(PlayerState):

    move\_input: Vector2D

    jump\_input\_time: float

    def \_\_init\_\_(self, player: Player, state\_name: str):

        super().\_\_init\_\_(player, state\_name)

        self.jump\_input\_time = 0

    def input\_handler(self):

        ###A: Overriding the parent implementation of input\_handler###

        self.move\_input = self.player.input\_.get\_axis\_raw()

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        if (

            self.player.input\_.get\_action\_down(PlayerActionSpace.DASH)

            and self.player.can\_dash

        ):

            self.player.state\_machine.change\_state(self.player.dash\_state)

        elif self.player.input\_.get\_axis\_raw().x:

            if (

                self.player.is\_touching\_wall == self.player.input\_.get\_axis\_raw().x

                and self.player.rigid\_body.velocity.y > 0

            ):

                self.player.state\_machine.change\_state(self.player.slide\_state)

        # Coyote Time

        if (

            self.player.input\_.get\_action\_down(PlayerActionSpace.UP)

            and perf\_counter() - self.start\_time < self.player.coyote\_time

            and issubclass(

                type(self.player.state\_machine.previous\_state), PlayerGroundedState

            )

        ):

            self.player.state\_machine.change\_state(self.player.jump\_state)

        # Jump Buffering

        if self.player.input\_.get\_action\_down(PlayerActionSpace.UP):

            self.jump\_input\_time = perf\_counter()

        if self.player.is\_grounded:

            if perf\_counter() - self.jump\_input\_time < self.player.jump\_buffer\_time:

                self.player.state\_machine.change\_state(self.player.jump\_state)

            elif self.player.rigid\_body.velocity.y >= 0:

                self.player.state\_machine.change\_state(self.player.land\_state)

        else:

            target\_speed = self.move\_input.x \* self.player.x\_run\_speed

            if self.player.state\_machine.previous\_state == self.player.wall\_jump\_state:

                if (

                    perf\_counter() - self.player.state\_machine.previous\_state.start\_time

                    < self.player.wall\_jump\_time

                ):

                    target\_speed = lerp(

                        self.player.rigid\_body.velocity.x,

                        target\_speed,

                        self.player.wall\_jump\_lerp,

                    )

            speed\_difference = target\_speed - self.player.rigid\_body.velocity.x

            acceleration\_rate = (

                self.player.acceleration\_rate

                if abs(target\_speed) > 0

                else self.player.deceleration\_rate

            )

            if (

                self.player.state\_machine.previous\_state

                in (self.player.jump\_state, self.player.wall\_jump\_state)

                and abs(self.player.rigid\_body.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                acceleration\_rate \*= self.player.jump\_hang\_acceleration\_mult

                target\_speed \*= self.player.jump\_hang\_max\_speed\_mult

            movement = pow(

                abs(speed\_difference) \* acceleration\_rate, self.player.velocity\_power

            ) \* sign(speed\_difference)

            self.player.rigid\_body.add\_force(Vector2D(1, 0).scale(movement), delta\_time)

            if abs(target\_speed) == 0:

                friction = sign(self.player.rigid\_body.velocity.x) \* min(

                    abs(self.player.rigid\_body.velocity.x), self.player.friction

                )

                self.player.rigid\_body.add\_force(

                    Vector2D(1, 0).scale(friction), force\_mode=ForceMode.IMPULSE

                )

            if (

                self.player.state\_machine.previous\_state

                in (self.player.jump\_state, self.player.wall\_jump\_state)

                and abs(self.player.rigid\_body.velocity.y)

                < self.player.jump\_hang\_time\_threshold

            ):

                gravity\_scale = (

                    self.player.rigid\_body.gravity\_scale

                    \* self.player.jump\_hang\_gravity\_mult

                )

            elif self.player.rigid\_body.velocity.y > 0:

                gravity\_scale = (

                    self.player.rigid\_body.gravity\_scale

                    \* self.player.jump\_fast\_fall\_mult

                )

            else:

                gravity\_scale = self.player.rigid\_body.gravity\_scale

            self.player.rigid\_body.add\_force(

                Vector2D(0, 1).scale(gravity\_scale),

                delta\_time,

                ForceMode.ACCELERATION,

            )

            self.player.rigid\_body.velocity = Vector2D(

                self.player.rigid\_body.velocity.x,

                min(self.player.rigid\_body.velocity.y, self.player.max\_fall),

            )

player\_ability\_state.py

from time import perf\_counter

from nea\_game.calc.vector2d import Vector2D

from nea\_game.states.player\_state import PlayerState

class PlayerAbilityState(PlayerState):

    move\_input: Vector2D

    is\_ability\_done: bool

    def enter(self):

        ###A: Overriding the parent implementation of enter###

        self.start\_time = perf\_counter()

        self.is\_ability\_done = False

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        super().update(delta\_time)

        if self.is\_ability\_done:

            if self.player.is\_grounded and self.player.rigid\_body.velocity.x == 0:

                self.player.state\_machine.change\_state(self.player.idle\_state)

            else:

                self.player.state\_machine.change\_state(self.player.in\_air\_state)

player\_jump\_state.py

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerJumpState(PlayerAbilityState):

    def enter(self):

        ###A: Overriding the parent implementation of enter###

        super().enter()

        self.player.rigid\_body.add\_force(

            Vector2D(0, -1).scale(self.player.jump\_force), force\_mode=ForceMode.IMPULSE

        )

        self.is\_ability\_done = True

player\_wall\_jump\_state.py

from nea\_game.calc.sign import sign

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerWallJumpState(PlayerAbilityState):

    def enter(self):

        ###A: Overriding the parent implementation of update###

        super().enter()

        force = Vector2D(

            self.player.wall\_jump\_force.x \* self.player.direction \* -1,

            self.player.wall\_jump\_force.y \* -1,

        )

        if sign(self.player.rigid\_body.velocity.x) != sign(force.x):

            force = Vector2D(force.x - self.player.rigid\_body.velocity.x, force.y)

        if self.player.rigid\_body.velocity.y > 0:

            force = Vector2D(force.x, force.y - self.player.rigid\_body.velocity.y)

        self.player.rigid\_body.add\_force(force, force\_mode=ForceMode.IMPULSE)

        self.is\_ability\_done = True

player\_dash\_state.py

from time import perf\_counter

from nea\_game.calc.vector2d import Vector2D

from nea\_game.components.rigidbody2d import ForceMode

from nea\_game.player.super\_states.player\_ability\_state import PlayerAbilityState

class PlayerDashState(PlayerAbilityState):

    def enter(self):

        ###A: Overriding the parent implementation of enter###

        super().enter()

        move\_input = self.player.input\_.get\_axis\_raw()

        if move\_input == Vector2D(0, 0):

            self.is\_ability\_done = True

        else:

            self.player.rigid\_body.velocity = Vector2D(0, 0)

            force = Vector2D(move\_input.x, move\_input.y).scale(self.player.dash\_speed)

            self.player.rigid\_body.add\_force(force, force\_mode=ForceMode.IMPULSE)

            self.player.can\_dash = False

    def exit(self):

        ###A: Overriding the parent implementation of exit###

        super().exit()

        self.player.rigid\_body.velocity = Vector2D(

            self.player.rigid\_body.velocity.x, self.player.rigid\_body.velocity.y \* 0.2

        )

    def update(self, delta\_time: float):

        ###A: Overriding the parent implementation of update###

        if self.player.is\_grounded:

            self.is\_ability\_done = True

        if perf\_counter() - self.start\_time > self.player.dash\_time:

            self.is\_ability\_done = True

        super().update(delta\_time)

window.py

from pygame.event import Event

from pygame import Surface

class Window:

    screen: Surface

    display\_surface: Surface

    scale\_factor: int

    def \_\_init\_\_(self, screen: Surface, display\_surface: Surface):

        self.screen = screen

        self.display\_surface = display\_surface

        self.scale\_factor = self.screen.get\_width() // self.display\_surface.get\_width()

        if (

            self.scale\_factor

            != self.screen.get\_height() // self.display\_surface.get\_height()

        ):

            raise ValueError("Display surface and screen must be proportional")

    def reload(self):

        """Called when the active window is changed"""

    def event\_handler(self, events: list[Event]):

        """Passes down the event to the event\_handler for any relvant operations

        Args:

            events (list[Event]): A list of pygame events

        """

    def update(self, delta\_time: float):

        """Updates window"""

    def draw(self):

        """Draws window"""

game.py

from \_\_future\_\_ import annotations

import itertools

from json import dump, load as load\_json

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.event import Event

from pygame import Surface

from nea\_game.game.camera import Camera

from nea\_game.game.render\_object import RenderObject

from nea\_game.gui.window import Window

from nea\_game.menu.background\_layer import BackgroundLayer

from nea\_game.menu.pause\_menu import Pause

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

from nea\_game.ldtk\_world\_loader.world import World

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.player.player import Player

from nea\_game.config import NeaGameConfig

from nea\_game.circular\_queue import CircularQueue

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class Game(Window):

    """Creates a window that allows a player to play the specified level, when the level is beaten the next one is unlocked by writing to the config.json file"""

    parent: NeaGame

    config: NeaGameConfig

    world\_identifier: str

    level\_identifier: str

    world: World

    render\_queue: CircularQueue[RenderObject]

    background\_layers: list[BackgroundLayer]

    player: Player

    camera: Camera

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        world\_identifier: str,

        level\_identifier: str,

        background\_image\_layers\_path: Path,

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.config = self.parent.config

        self.world\_identifier = world\_identifier

        self.level\_identifier = level\_identifier

        self.world = World(

            self.world\_identifier,

            self.config.directories["worlds"],

            self.config.chunk\_size,

        )

        self.render\_queue = CircularQueue(

            400, RenderObject

        )  # A: Circular queue to model render queue

        self.background\_layers = [

            BackgroundLayer(pygame.image.load(background\_image\_layers\_path / filename))

            for filename in sorted(listdir(background\_image\_layers\_path))

            if filename.endswith(".png") and filename not in ["-1.png", "2.png"]

        ]

        self.player = Player(

            self.config.directories["player"],

            self.world.levels[self.level\_identifier].level\_data["chunks"],

            self.config.key\_bindings,

            self.config.internal\_fps,

            self.world.levels[self.level\_identifier].level\_data["player\_position"],

        )

        self.camera = Camera(

            self.world.levels[self.level\_identifier].height,

            self.world.levels[self.level\_identifier].width,

            self.display\_surface.get\_height(),

            self.display\_surface.get\_width(),

        )

    def event\_handler(self, events: list[Event]):

        for event in events:

            if event.type == pygame.KEYDOWN and event.key == pygame.K\_ESCAPE:

                window = Pause(

                    self.parent,

                    self.screen,

                    self.display\_surface,

                    self.parent.config.directories["gui"] / "pause",

                )

                self.parent.windows["pause"] = window

                self.parent.show\_window("pause")

        self.player.event\_handler(events)

    def update(self, delta\_time: float):

        if self.has\_level\_finished():

            if self.parent.is\_transition\_done:

                self.end\_level()

                self.parent.show\_window("level\_selection")

            elif not self.parent.is\_transitioning:

                self.parent.set\_transitioning(

                    self.parent.transition\_circle\_out,

                    2,

                    tuple(

                        map(

                            lambda x: x \* self.scale\_factor,

                            (

                                self.player.rect.centerx - self.camera.scroll\_x,

                                self.player.rect.centery - self.camera.scroll\_y,

                            ),

                        )

                    ),

                )

            return

        if not self.player.is\_alive(self.camera.height):

            if self.player.input\_.get\_action\_down(PlayerActionSpace.DASH):

                self.parent.is\_transition\_done = True

                window = Game(

                    self.parent,

                    self.screen,

                    self.display\_surface,

                    self.world\_identifier,

                    self.level\_identifier,

                    self.parent.config.directories["background"] / "sky\_mountain",

                )

                self.parent.windows["game"] = window

                self.parent.show\_window("game")

            if self.parent.is\_transition\_done:

                self.parent.show\_window("level\_selection")

                self.parent.sound\_manager.play\_sound("click")

            elif not self.parent.is\_transitioning:

                self.parent.set\_transitioning(

                    self.parent.transition\_circle\_out,

                    1,

                    tuple(

                        map(

                            lambda x: x \* self.scale\_factor,

                            (

                                self.player.rect.centerx - self.camera.scroll\_x,

                                self.player.rect.centery - self.camera.scroll\_y,

                            ),

                        )

                    ),

                )

        if not self.parent.is\_transitioning:

            self.parent.is\_transition\_done = False

            self.player.update(delta\_time)

        if self.player.state\_machine.current\_state == self.player.jump\_state:

            self.parent.sound\_manager.play\_sound("jump")

        self.camera.update(self.player.rect)

    def has\_level\_finished(self) -> bool:

        """Checks if the level has finished

        Returns:

            bool: Whether the level has finished or not

        """

        if self.player.rect.colliderect(

            self.world.levels[self.level\_identifier].level\_data["level\_finish"].rect

        ):

            return True

        return False

    def end\_level(self):

        """Ends the level by unlocking the next level and reloading the config.json file

        """

        level\_finish = self.world.levels[self.level\_identifier].level\_data[

            "level\_finish"

        ]

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            new\_settings\_json = load\_json(settings\_json)

            if level\_finish.new\_world:

                new\_world = str(int(self.world\_identifier) + 1)

            else:

                new\_world = self.world\_identifier

            new\_level = level\_finish.next\_level\_identifier[1:].replace("\_", "-")

            new\_settings\_json["unlocked\_levels"][new\_world + new\_level] = True

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="w"

        ) as settings\_json:

            dump(new\_settings\_json, settings\_json, indent=4)

        self.parent.config.reload()

    def get\_visible\_chunks(self) -> list[list[LevelTile]]:

        """Returns the list of LevelTiles that are currently on the screen

        Returns:

            list[list[LevelTile]]: The LevelTile list for each chunk that is visible on the screen

        """

        ###B: Two Dimensional List###

        visible\_chunks: list[list[LevelTile]] = []

        for chunk\_x, chunk\_y in self.world.levels[self.level\_identifier].level\_data[

            "chunks"

        ]:

            if pygame.Rect(

                chunk\_x \* self.config.chunk\_size,

                chunk\_y \* self.config.chunk\_size,

                self.config.chunk\_size,

                self.config.chunk\_size,

            ).colliderect(

                pygame.Rect(

                    self.camera.scroll\_x,

                    self.camera.scroll\_y,

                    self.config.internal\_resoloution[0],

                    self.config.internal\_resoloution[1],

                )

            ):

                visible\_chunks.append(

                    self.world.levels[self.level\_identifier].level\_data["chunks"][

                        (chunk\_x, chunk\_y)

                    ]

                )

        return visible\_chunks

    def update\_render\_queue(self):

        """Updates the render queue to only render the tiles in chunks that are currently visible

        """

        for chunk in self.get\_visible\_chunks():

            for tile in chunk:

                self.render\_queue.enqueue(

                    RenderObject(tile.rect.x, tile.rect.y, tile.image)

                )

    def draw(self):

        self.display\_surface.fill((0, 0, 0))

        self.update\_render\_queue()

        self.display\_surface.blit(self.background\_layers[0].image, (0, 0))

        for background\_layer in self.background\_layers[1:2]:

            self.display\_surface.blit(background\_layer.get\_new\_sub\_image(), (0, 0))

        while not self.render\_queue.is\_empty():

            render\_object = self.render\_queue.dequeue()

            self.display\_surface.blit(

                render\_object.image,

                (

                    render\_object.x - self.camera.scroll\_x,

                    render\_object.y - self.camera.scroll\_y,

                ),

            )

        self.player.renderer.render\_entity(

            self.player.state\_machine.current\_state,

            self.player.direction == -1,

            self.display\_surface,

            self.player.rect.x - self.camera.scroll\_x,

            self.player.rect.y - self.camera.scroll\_y,

        )

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        if self.parent.is\_transitioning:

            self.parent.run\_transition()

        pygame.display.flip()

level\_selection.py

from \_\_future\_\_ import annotations

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.game.game import Game

from nea\_game.gui.button import Button

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class LevelSelection(Window):

    parent: NeaGame

    button\_folder\_path: Path

    padding: tuple[int, int]

    spacing: tuple[int, int]

    buttons: dict[str, Button]

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        path: Path,

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.button\_folder\_path = path / "buttons"

        self.padding = (30, 50)

        self.spacing = (59, 38)

        self.buttons = {}

        for button in listdir(self.button\_folder\_path):

            self.buttons[button] = Button(self.button\_folder\_path / button)

        for button\_name, button in zip(self.buttons, self.buttons.values()):

            if button\_name == "back":

                button.rect.bottomright = (383, 215)

            else:

                world\_num = int(button\_name[0])

                level\_num = int(button\_name[2])

                button.rect.topleft = (

                    self.padding[0] + (level\_num - 1) \* self.spacing[0],

                    self.padding[1] + (world\_num - 1) \* self.spacing[1],

                )

        for name, unlocked in self.parent.config.unlocked\_levels.items():

            if not unlocked:

                self.buttons[name].active\_image = self.buttons[name].passive\_image

                self.buttons[name].on\_click\_image = self.buttons[name].passive\_image

                self.buttons[name].can\_be\_clicked = False

    def reload(self):

        self.buttons = {}

        for button in listdir(self.button\_folder\_path):

            self.buttons[button] = Button(self.button\_folder\_path / button)

        for button\_name, button in zip(self.buttons, self.buttons.values()):

            if button\_name == "back":

                button.rect.bottomright = (383, 215)

            else:

                world\_num = int(button\_name[0])

                level\_num = int(button\_name[2])

                button.rect.topleft = (

                    self.padding[0] + (level\_num - 1) \* self.spacing[0],

                    self.padding[1] + (world\_num - 1) \* self.spacing[1],

                )

        for name, unlocked in self.parent.config.unlocked\_levels.items():

            if not unlocked:

                self.buttons[name].active\_image = self.buttons[name].passive\_image

                self.buttons[name].on\_click\_image = self.buttons[name].passive\_image

                self.buttons[name].can\_be\_clicked = False

    def update(self, delta\_time: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, delta\_time)

        if self.buttons["back"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            self.parent.show\_window("main\_menu")

        for button\_name, button in self.buttons.items():

            if button\_name == "back":

                continue

            if button.clicked:

                self.parent.sound\_manager.play\_sound("click")

                world\_and\_level = button\_name.split("-")

                window = Game(

                    self.parent,

                    self.screen,

                    self.display\_surface,

                    world\_and\_level[0],

                    f"\_{world\_and\_level[1]}",

                    self.parent.config.directories["background"] / "sky\_mountain",

                )

                self.parent.windows["game"] = window

                self.parent.show\_window("game")

                self.parent.set\_transitioning(

                    self.parent.transition\_circle\_in,

                    1,

                    tuple(

                        map(

                            lambda x: x \* self.scale\_factor,

                            self.parent.windows["game"].player.rect.center,

                        )

                    ),

                )

    def draw(self):

        self.display\_surface.fill((8, 169, 252))

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

main\_menu.py

from \_\_future\_\_ import annotations

from os import listdir

from pathlib import Path

from sys import exit as sys\_exit

import typing

import pygame

from pygame.image import load

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.menu.level\_selection import LevelSelection

from nea\_game.menu.settings\_menu import Settings

from nea\_game.menu.background\_layer import BackgroundLayer

from nea\_game.gui.button import Button

from nea\_game.gui.title import Title

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class MainMenu(Window):

    parent: NeaGame

    buttons: dict[str, Button]

    background\_layers: list[BackgroundLayer]

    splash\_screen\_opacity: int

    title: Title

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        path: Path,

        background\_image\_layers\_path: Path,

        background\_transparency\_percentage: float = 1,

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.buttons = {}

        self.background\_layers = [

            BackgroundLayer(load(background\_image\_layers\_path / filename))

            for filename in sorted(listdir(background\_image\_layers\_path))

            if filename.endswith(".png") and filename != "-1.png"

        ]

        self.set\_background\_transparency\_percentage(background\_transparency\_percentage)

        self.title = Title(path / "title.png")

        self.title.rect.y = 0

        self.title.center\_on\_x\_axis(self.display\_surface.get\_width())

        for button in listdir(path / "buttons"):

            self.buttons[button] = Button(path / "buttons" / button)

        self.buttons["play\_game"].rect.y = 115

        self.buttons["play\_game"].center\_on\_x\_axis(self.display\_surface.get\_width())

        self.buttons["settings"].rect.y = 150

        self.buttons["settings"].center\_on\_x\_axis(self.display\_surface.get\_width())

        self.buttons["exit"].rect.y = 185

        self.buttons["exit"].center\_on\_x\_axis(self.display\_surface.get\_width())

    def reload(self):

        super().reload()

        self.parent.sound\_manager.set\_bgm\_volume(self.parent.config.music\_volume)

        self.parent.sound\_manager.set\_sfx\_volume(self.parent.config.sfx\_volume)

    def update(self, delta\_time: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, delta\_time)

        if self.buttons["play\_game"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            window = LevelSelection(

                self.parent,

                self.screen,

                self.display\_surface,

                self.parent.config.directories["gui"] / "level\_selection",

            )

            self.parent.windows["level\_selection"] = window

            self.parent.show\_window("level\_selection")

        if self.buttons["settings"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            window = Settings(

                self.parent,

                self.screen,

                self.display\_surface,

                self.parent.config.directories["gui"] / "settings",

                self.parent.config.key\_bindings,

            )

            self.parent.windows["settings"] = window

            self.parent.show\_window("settings")

        if self.buttons["exit"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            pygame.quit()

            sys\_exit()

    def draw(self):

        self.display\_surface.fill((0, 0, 0))

        self.display\_surface.blit(self.background\_layers[0].image, (0, 0))

        self.display\_surface.blit(self.background\_layers[1].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[2].image, (0, 0))

        self.display\_surface.blit(self.background\_layers[3].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[4].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.background\_layers[5].get\_new\_sub\_image(), (0, 0))

        self.display\_surface.blit(self.title.image, self.title.rect.topleft)

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

    def set\_background\_transparency\_percentage(self, transparency\_percentage: float):

        """Sets the transparency for the background image as a number between 0 and 255 from a percenatge

        Args:

            transparency\_percentage (float): The percentage transparency for the back ground image (0= fully transparent, 1=fully opaque)

        Raises:

            ValueError: Raised when the transparency percentage is not between 0 and 1 inclusive

        """

        if transparency\_percentage < 0 or transparency\_percentage > 1:

            raise ValueError("Percentage must be between 0 and 1 inlcusive")

        for layer in self.background\_layers:

            layer.image.set\_alpha(int(255 \* transparency\_percentage))

settings\_menu.py

from \_\_future\_\_ import annotations

from json import dump, load as load\_json

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.image import load

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.menu.action\_button import ActionButton

from nea\_game.gui.button import Button

from nea\_game.gui.slider import Slider

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class Settings(Window):

    parent: NeaGame

    title\_image: Surface

    text: Surface

    music\_icon: Surface

    sound\_icon: Surface

    buttons: dict[str, Button]

    action\_buttons: dict[str, ActionButton]

    sliders: dict[str, Slider]

    def \_\_init\_\_(

        self,

        parent: NeaGame,

        screen: Surface,

        display\_surface: Surface,

        path: Path,

        key\_bindings: list[int],

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.title\_image = load(path / "title.png").convert\_alpha()

        self.text = load(path / "text.png").convert\_alpha()

        self.music\_icon = load(path / "music.png")

        self.sound\_icon = load(path / "sound.png")

        self.buttons = {}

        self.action\_buttons = {}

        self.sliders = {}

        self.sliders["music\_volume"] = Slider(

            path / "sliders/volume",

            0,

            100,

            int(self.parent.config.music\_volume \* 100),

        )

        self.sliders["sfx\_volume"] = Slider(

            path / "sliders/volume",

            0,

            100,

            int(self.parent.config.sfx\_volume \* 100),

        )

        for button in listdir(path / "buttons"):

            self.buttons[button] = Button((path / "buttons") / button)

        for action\_button, key\_binding in zip(

            listdir(path / "action\_buttons"), key\_bindings

        ):

            self.action\_buttons[action\_button] = ActionButton(

                path / "action\_buttons" / action\_button,

                path / "keys",

                key\_binding,

            )

        self.buttons["back"].rect.bottomright = (383, 215)

        for index, action\_button in enumerate(self.action\_buttons.values()):

            action\_button.rect.topleft = (20, 20 + 40 \* index)

        self.sliders["music\_volume"].set\_topleft(195, 175)

        self.sliders["sfx\_volume"].set\_topleft(195, 200)

    def update(self, delta\_time: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for slider in self.sliders.values():

            slider.update(scaled\_mouse\_pos, mouse\_clicked)

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, delta\_time)

        for action\_button in self.action\_buttons.values():

            if action\_button.click\_timer >= 0:

                other\_binds = [

                    other\_action\_button.key

                    for other\_action\_button in self.action\_buttons.values()

                    if other\_action\_button != action\_button

                ]

                for event in pygame.event.get():

                    action\_button.update(

                        scaled\_mouse\_pos, mouse\_clicked, 0, other\_binds, event

                    )

            action\_button.update(scaled\_mouse\_pos, mouse\_clicked, delta\_time)

        if self.buttons["back"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            self.save()

            self.parent.show\_window("main\_menu")

    def draw(self):

        self.display\_surface.fill((8, 169, 252))

        self.display\_surface.blit(self.title\_image, (195, 5))

        self.display\_surface.blit(self.text, (150, 44))

        self.display\_surface.blit(self.music\_icon, (180, 176))

        self.display\_surface.blit(self.sound\_icon, (180, 200))

        for slider in self.sliders.values():

            self.display\_surface.blit(slider.current\_image, slider.rect.topleft)

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        for action\_button in self.action\_buttons.values():

            self.display\_surface.blit(

                action\_button.current\_image, action\_button.rect.topleft

            )

            self.display\_surface.blit(

                action\_button.key\_image,

                (action\_button.rect.right + 10, action\_button.rect.top - 12),

            )

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

    def save(self):

        """Saves the new settings to the config.json file

        """

        ###B: Writing to a JSON file###

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            new\_settings\_json = load\_json(settings\_json)

            new\_settings\_json["music\_volume"] = (

                self.sliders["music\_volume"].value - 1

            ) / self.sliders["music\_volume"].max\_value

            new\_settings\_json["sfx\_volume"] = (

                self.sliders["sfx\_volume"].value - 1

            ) / self.sliders["sfx\_volume"].max\_value

            new\_settings\_json["key\_bindings"] = [

                action\_button.key for action\_button in self.action\_buttons.values()

            ]

        with (self.parent.config.directories["platformer"] / "config.json").open(

            mode="w"

        ) as settings\_json:

            dump(new\_settings\_json, settings\_json, indent=4)

        self.parent.config.reload()

pause\_menu.py

from \_\_future\_\_ import annotations

from os import listdir

from pathlib import Path

import typing

import pygame

from pygame.mouse import get\_pos as get\_mouse\_pos, get\_pressed as get\_mouse\_pressed

from pygame import Surface

from nea\_game.gui.button import Button

from nea\_game.gui.window import Window

if typing.TYPE\_CHECKING:

    from nea\_game.nea\_game import NeaGame

class Pause(Window):

    parent: NeaGame

    buttons: dict[str, Button]

    def \_\_init\_\_(

        self, parent: NeaGame, screen: Surface, display\_surface: Surface, path: Path

    ):

        super().\_\_init\_\_(screen, display\_surface)

        self.parent = parent

        self.buttons = {}

        for button in listdir(path / "buttons"):

            self.buttons[button] = Button(path / "buttons" / button)

        self.buttons["resume"].rect.y = 115

        self.buttons["resume"].center\_on\_x\_axis(self.display\_surface.get\_width())

        self.buttons["exit"].rect.y = 150

        self.buttons["exit"].center\_on\_x\_axis(self.display\_surface.get\_width())

    def update(self, delta\_time: float):

        mouse\_pos: tuple[int, int] = get\_mouse\_pos()

        scaled\_mouse\_pos: tuple[int, int] = (

            mouse\_pos[0] // self.scale\_factor,

            mouse\_pos[1] // self.scale\_factor,

        )

        mouse\_clicked = get\_mouse\_pressed()[0]

        for button in self.buttons.values():

            button.update(scaled\_mouse\_pos, mouse\_clicked, delta\_time)

        if self.buttons["resume"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            self.parent.show\_window("game")

        if self.buttons["exit"].clicked:

            self.parent.sound\_manager.play\_sound("click")

            self.parent.show\_window("main\_menu")

    def draw(self):

        self.display\_surface.fill((8, 169, 252))

        for button in self.buttons.values():

            self.display\_surface.blit(button.current\_image, button.rect.topleft)

        pygame.transform.scale(

            self.display\_surface, self.screen.get\_size(), dest\_surface=self.screen

        )

        pygame.display.flip()

root.py

import pygame

from pygame import Surface

from nea\_game.gui.window import Window

from nea\_game.game.engine import Engine

from nea\_game.sound\_manager import SoundManager

class Root:

    screen: Surface

    display\_surface: Surface

    active\_window: Window

    engine: Engine

    sound\_manager: SoundManager

    windows: dict[str, Window]

    def \_\_init\_\_(

        self,

        screen\_resolution: tuple[int, int],

        display\_surafce\_resolution: tuple[int, int],

        fps: int,

    ):

        pygame.init()

        self.screen = pygame.display.set\_mode(screen\_resolution)

        self.display\_surface = Surface(display\_surafce\_resolution)

        self.engine = Engine(fps)

        self.sound\_manager = SoundManager()

        ###B: Dictionary used to store windows###

        self.windows = {}

    def show\_window(self, window: str):

        """Shows the specified window

        Args:

            window (str): The key of the window to be shown

        """

        self.active\_window = self.windows[window]

        self.active\_window.reload()

    def update(self):

        """Updates the delta\_time variable and updates the active\_window as well as drawing the active\_window

        """

        self.engine.update()

        self.active\_window.update(self.engine.delta\_time)

        self.active\_window.draw()

nea\_game.py

from typing import Callable

from os.path import splitext

from os import listdir

from sys import exit as sys\_exit

from pygame.event import Event

import pygame

from nea\_game.config import NeaGameConfig

from nea\_game.gui.root import Root

from nea\_game.menu.main\_menu import MainMenu

class NeaGame(Root):

    config: NeaGameConfig

    transition: None | Callable[[], None]

    transition\_start\_radius: int

    transition\_circle\_centre: tuple[int, int]

    current\_transition\_frame: int

    is\_transitioning: bool

    is\_transition\_done: bool

    def \_\_init\_\_(self, config: NeaGameConfig):

        self.config = config

        pygame.display.set\_caption("NEA Game")

        # pygame.display.set\_icon()

        super().\_\_init\_\_(

            self.config.resoloution,

            self.config.internal\_resoloution,

            self.config.fps,

        )

        self.windows["main\_menu"] = MainMenu(

            self,

            self.screen,

            self.display\_surface,

            config.directories["gui"] / "main\_menu",

            config.directories["background"] / "sky\_mountain",

        )

        self.transition: None | Callable[[], None]

        self.transition\_start\_radius = 0

        self.transition\_time = 0

        self.transition\_circle\_centre = (0, 0)

        self.current\_transition\_frame = 0

        self.is\_transitioning = False

        self.is\_transition\_done = False

        self.sound\_manager.set\_bgm(self.config.directories["music"] / "bgm.wav")

        for sound\_effect in listdir(self.config.directories["sfx"]):

            self.sound\_manager.load\_sound(

                splitext(sound\_effect)[0], self.config.directories["sfx"] / sound\_effect

            )

        self.show\_window("main\_menu")

    def show\_window(self, window: str):

        super().show\_window(window)

        self.is\_transitioning = False

        self.is\_transition\_done = False

    def update(self):

        self.get\_events()

        super().update()

    def get\_events(self):

        events: list[Event] = []

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                sys\_exit()

            else:

                events.append(event)

        self.active\_window.event\_handler(events)

    def set\_transitioning(

        self,

        transition: Callable[[], None],

        transition\_time: float,

        circle\_centre: tuple[int, int],

    ):

        self.is\_transitioning = True

        self.is\_transition\_done = False

        self.transition = transition

        circle\_x, circle\_y = circle\_centre

        self.transition\_time = transition\_time

        self.transition\_circle\_centre = circle\_centre

        match transition:

            case self.transition\_circle\_out:

                self.transition\_start\_radius = max(circle\_x, self.screen.get\_width() - circle\_x, circle\_y, self.screen.get\_height() - circle\_y)

                self.current\_transition\_frame = 80

            case self.transition\_circle\_in:

                self.transition\_start\_radius = max(circle\_x, self.screen.get\_width() - circle\_x, circle\_y, self.screen.get\_height() - circle\_y)

                self.current\_transition\_frame = 0

            case \_:

                pass

    def run\_transition(self):

        if self.transition is not None:

            self.transition()

    def transition\_circle\_out(self):

        surface = pygame.Surface(self.screen.get\_size())

        radius = (

            self.transition\_start\_radius / (self.transition\_time \* self.engine.fps)

        ) \* self.current\_transition\_frame

        pygame.draw.circle(surface, "red", self.transition\_circle\_centre, radius)

        surface.set\_colorkey("red")

        self.screen.blit(surface, (0, 0))

        self.current\_transition\_frame -= 1

        if self.current\_transition\_frame <= 0:

            self.is\_transitioning = False

            self.is\_transition\_done = True

    def transition\_circle\_in(self):

        surface = pygame.Surface(self.screen.get\_size())

        radius = (

            self.transition\_start\_radius / (self.transition\_time \* self.engine.fps)

        ) \* self.current\_transition\_frame

        pygame.draw.circle(surface, "red", self.transition\_circle\_centre, radius)

        surface.set\_colorkey("red")

        self.screen.blit(surface, (0, 0))

        self.current\_transition\_frame += 1

        if self.current\_transition\_frame >= self.transition\_time \* self.engine.fps:

            self.is\_transitioning = False

            self.is\_transition\_done = True

tileset\_tile.py

from typing import Optional

from pygame import Rect, Surface

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

class TilesetTile:

    """A class to represent a tile in a tileset

    """

    identifier: int

    image: Surface

    collision\_type: CollisionType

    rect: Rect

    def \_\_init\_\_(

        self,

        identifier: int,

        image: Surface,

        collision\_type: CollisionType,

        rect: Optional[Rect] = None,

    ):

        self.identifier = identifier

        self.image = image

        self.collision\_type = collision\_type

        if rect:

            self.rect = rect

        else:

            self.rect = self.image.get\_bounding\_rect()

level\_tile.py

from typing import Self

from pygame import Surface

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

from nea\_game.ldtk\_world\_loader.tileset\_tile import TilesetTile

class LevelTile(TilesetTile):

    """A class to represent a tile within a level

    """

    def \_\_init\_\_(

        self,

        identifier: int,

        image: Surface,

        collision\_type: CollisionType,

        px: tuple[int, int],

    ):

        super().\_\_init\_\_(identifier, image, collision\_type)

        self.rect.topleft = px

    @classmethod

    def from\_tileset\_tile(cls, tileset\_tile: TilesetTile, px: tuple[int, int]) -> Self:

        """Returns a LevelTile object based off of a TilesetTile object

        Args:

            tileset\_tile (TilesetTile): The given TileseTile

            px (tuple[int, int]): The position of the tile in pixels in the level

        Returns:

            Self: A LevelTile from a TilesetTile object

        """

        return cls(

            tileset\_tile.identifier,

            tileset\_tile.image,

            tileset\_tile.collision\_type,

            px,

        )

renderer.py

from pygame import Surface

import pygame

from nea\_game.states.player\_state import PlayerState

class StaticRenderer:

    """Provides functionality for rendering a static entity"""

    static\_frame: Surface

    def \_\_init\_\_(self, static\_frame: Surface):

        """Creates a StaticRenderer component for a static object

        Args:

            static\_frame (Surface): The static frame to be rendered

        """

        self.static\_frame = static\_frame

    def render\_entity(self, surface: Surface, x: float, y: float):

        """Renders the static frame onto the surface at the given position

        Args:

            surface (Surface): The surface that the frame is rendered onto

            x (int): The x position that the static frame is rendered at

            y (int): The y position that the static frame is rendered at

        """

        surface.blit(self.static\_frame, (x, y))

class AnimatedRenderer:

    """Provides functionality for rendering an animated entity with multiple states"""

    frames: dict[str, list[Surface]]

    def \_\_init\_\_(self, frames: dict[str, list[Surface]]):

        """Creates an AnimatedRenderer component for an animated object

        Args:

            frames (dict[str, list[Surface]]): The list of each possible frame for the animated object

        """

        self.frames = frames

    def render\_entity(

        self,

        state: PlayerState,

        flip\_x: bool,

        surface: Surface,

        x: float,

        y: float,

    ):

        """Renders the current frame onto the surface at the given position

        Args:

            state\_name (str): The current state of the entity

            surface (Surface): The surface that the frame is rendered onto

            x (int): The x position that the frame is rendered onto

            y (int): The y position that the frame is rendered onto

        """

        surface.blit(

            pygame.transform.flip(

                self.frames[state.state\_name][state.animation\_index], flip\_x, False

            ),

            (x, y),

        )

input.py

from pygame.event import Event

import pygame

from nea\_game.player.player\_action\_space import PlayerActionSpace

from nea\_game.calc.vector2d import Vector2D

class Input:

    """Provides an interface for accessing the inputs of a player given actions"""

    action\_space: type[PlayerActionSpace]

    actions: dict[PlayerActionSpace, int]

    actions\_performed\_on\_current\_frame: dict[PlayerActionSpace, bool]

    def \_\_init\_\_(

        self, action\_space: type[PlayerActionSpace], action\_bindings: list[int]

    ):

        """Summary

        Args:

            action\_space (type[PlayerActionSpace]): An Enum for each possible action the player can make

            action\_bindings (list[int]): The binding for each corresponding action in action\_space

        """

        self.action\_space = action\_space

        self.actions = dict(zip(list(self.action\_space), action\_bindings))

        self.actions\_performed\_on\_current\_frame = {

            action: False for action in list(self.action\_space)

        }

    def get\_axis\_raw(self) -> Vector2D:

        """Calculates the vector of the directional inputs based on the bindings of the actions UP, DOWN, LEFT, RIGHT. The value for both axis will be either -1, 0 or 1.

        Returns:

            Vector2D: The vector of the directional inputs"""

        horizontal = (

            -1

            if pygame.key.get\_pressed()[self.actions[self.action\_space.LEFT]]

            else 1

            \* (

                pygame.key.get\_pressed()[self.actions[self.action\_space.LEFT]]

                ^ pygame.key.get\_pressed()[self.actions[self.action\_space.RIGHT]]

            )

        )

        vertical = (

            1

            if pygame.key.get\_pressed()[self.actions[self.action\_space.DOWN]]

            else -1

            \* (

                pygame.key.get\_pressed()[self.actions[self.action\_space.DOWN]]

                ^ pygame.key.get\_pressed()[self.actions[self.action\_space.UP]]

            )

        )

        return Vector2D(horizontal, vertical)

    def update\_actions\_performed\_on\_current\_frame(self, events: list[Event]):

        """Updates the dictionary corresponding to the actions performed on the current frame"""

        self.actions\_performed\_on\_current\_frame = {

            action: False for action in self.action\_space

        }

        for event in events:

            if event.type == pygame.KEYDOWN:

                for action, binding in self.actions.items():

                    if event.key == binding:

                        self.actions\_performed\_on\_current\_frame[action] = True

    def get\_action\_down(self, action: PlayerActionSpace) -> bool:

        """Returns true during the frame the user starts pressing down the key identified by the action action enum parameter.

        Args:

            action (Enum): The action that is being checked

        Returns:

            bool: Whether that action was performed that frame

        """

        return self.actions\_performed\_on\_current\_frame[action]

rigidbody\_2d.py

from enum import Enum, auto

from nea\_game.calc.vector2d import Vector2D

class ForceMode(Enum):

    """Provides a container for the valid ways for a force to be applied to a rigid body

    Attributes:

        FORCE (TYPE): Add a continuous force to the rigidbody, using its mass

        IMPULSE (TYPE): Add an instant force impulse to the rigidbody, using its mass.

    """

    FORCE = auto()

    IMPULSE = auto()

    ACCELERATION = auto()

class RigidBody2D:

    mass: float

    gravity\_scale: float

    internal\_fps: int

    velocity: Vector2D

    """Creates a 2D rigid body that forces can be applied to"""

    def \_\_init\_\_(self, mass: float, gravity\_scale: float, internal\_fps: int):

        """Creates a 2D rigid body

        Args:

            mass (float): The mass of the body in kg

            gravity\_scale (float): The degree to which the body is affected by gravity

            internal\_fps (int): the interanl fps of the simulation

        """

        self.mass = mass

        self.gravity\_scale = gravity\_scale

        self.internal\_fps = internal\_fps

        self.velocity = Vector2D(0, 0)

    def add\_force(

        self,

        force: Vector2D,

        delta\_time: float = 0,

        force\_mode: ForceMode = ForceMode.FORCE,

    ):

        """Adds a force to the rigid body

        Args:

            force (Vector2D): The unscaled force that will act on the body

            delta\_time (float): The time difference between when the previous frame that was drawn and the current frame was drawn

            force\_mode (ForceModes, optional): Dictates the way that the force is applied

        Raises:

            ValueError: A ValueError is raised when the given force\_mode is not a valid force\_type

        """

        ###A: Uses physics equation f=ma###

        delta\_time \*= self.internal\_fps

        match force\_mode:

            case ForceMode.FORCE:

                self.velocity += force.scale(delta\_time / self.mass)

            case ForceMode.IMPULSE:

                self.velocity += force.scale(1 / self.mass)

            case ForceMode.ACCELERATION:

                self.velocity += force.scale(delta\_time)

            case \_:

                raise ValueError(

                    f"The given force\_type: {force\_mode} is not in {[member.value for member in ForceMode]}"

                )

        self.velocity = self.velocity.near\_zero()

player\_state\_machine.py

from nea\_game.states.player\_state import PlayerState

class StateMachine:

    current\_state: PlayerState

    previous\_state: PlayerState

    def \_\_init\_\_(self, starting\_state: PlayerState):

        """A state machine for an entity

        Args:

            starting\_state (State): The starting state for the entity

        """

        self.current\_state = starting\_state

        self.previous\_state = starting\_state

    def change\_state(self, new\_state: PlayerState):

        """Provides functionlaity for changing state

        Args:

            new\_state (State): The new state that the entity will change to

        """

        self.current\_state.exit()

        self.previous\_state = self.current\_state

        self.current\_state = new\_state

        self.current\_state.enter()

vector2d.py

"""Provides a Vector class in the 2-Dimensional plane with the associated methods for vectors"""

from \_\_future\_\_ import annotations

from typing import NamedTuple

from math import sqrt

from nea\_game.calc.near\_zero import near\_zero

class Vector2D(NamedTuple):

    """Creates a 2 dimensional vector"""

    x: float

    y: float

    def \_\_repr\_\_(self) -> str:

        """Returns a string representation of the vector displaying the both the x and y components

        Returns:

            String: The resulting representation of the vector

        """

        return f"Vector2D: ({self.x, self.y})"

    def \_\_add\_\_(self, other: Vector2D) -> Vector2D:

        """Calculates the sum of 2 vectors

        Args:

            other (Vector2D): The other vector to add

        Returns:

            Vector2D: The sum of the 2 vectors

        """

        return Vector2D(self.x + other.x, self.y + other.y)

    def near\_zero(self) -> Vector2D:

        return Vector2D(near\_zero(self.x), near\_zero(self.y))

    def magnitude(self) -> float:

        """Returns the magnitude of the vector

        Returns:

            float: The magnitude of the vector

        """

        return sqrt(self.x \* self.x + self.y \* self.y)

    def normalise(self) -> Vector2D:

        """Returns a vector with the same direction but a magnitude of 1

        ###

        A: Normalises a 2D vector

        ###

        Returns:

            Vector2D: The vector with the same direction but a magnitude of 1

        """

        return Vector2D(self.x / self.magnitude(), self.y / self.magnitude())

    def scale(self, scale\_factor: float) -> Vector2D:

        """Scales the vector by a given scale factor

        Args:

            scale\_factor (float): The amount that the vector is scaled by

        Returns:

            Vector2D: The scaled vector

        """

        return Vector2D(self.x \* scale\_factor, self.y \* scale\_factor)

    def scale\_to\_length(self, length: float) -> Vector2D:

        """Scales the vector to a given length

        Args:

            length (float): The length that the vector is scaled to

        Raises:

            ValueError: Raises a ValueError when the magnitude of the vector is 0 as

            that can't be scaled to a given length

        Returns:

            Vector2D: The vector scaled to the given length

        """

        if self.magnitude() == 0:

            raise ValueError("The Magnitude of the vector must be greater than zero")

        return self.normalise().scale(length)

    def dot(self, other: Vector2D) -> float:

        """Calculates the dot product with the other vector

        Args:

            other (Vector2D): The other vector that is used to calculate the dot product

        Returns:

            Float: The resulting scalar from performing the dot product

        """

        return self.x \* other.x + self.y \* other.y

    def cross(self, other: Vector2D) -> float:

        """Calculates the cross product with the other vector

        ###

        A: Calculates the cross product with another vector

        ###

        Args:

            other (Vector2D): The other vector that is used to calculate the cross product

        Returns:

            Float: The resulting scalar from performing the cross product

        """

        return (self.x \* other.y) - (self.y \* other.x)

engine.py

from time import perf\_counter

from pygame.time import Clock

class Engine:

    """

    Provides an interface to access the time difference between the when the previous frame that was drawn and the current frame was drawn

    """

    clock: Clock

    previous\_time: float

    delta\_time: float

    fps: int

    def \_\_init\_\_(self, fps: int):

        self.clock = Clock()

        self.previous\_time = perf\_counter()

        self.delta\_time = 0

        self.fps = fps

    def update(self):

        """Limits the time between frames to the fps set at initialisation and updates delta\_time to the time between the previous call"""

        self.clock.tick(self.fps)

        self.delta\_time = perf\_counter() - self.previous\_time

        self.previous\_time = perf\_counter()

render\_object.py

from pygame import Surface

class RenderObject:

    """

    A standardised object to be rendered onto the screen

    """

    x: int

    y: int

    image: Surface

    def \_\_init\_\_(self, x: int, y: int, image: Surface):

        self.x = x

        self.y = y

        self.image = image

sound\_manager.py

from pathlib import Path

from pygame.mixer import Sound

import pygame

class SoundManager:

    sound\_effects: dict[str, Sound]

    bgm\_volume: float

    sfx\_volume: float

    current\_bgm: Path

    def \_\_init\_\_(self):

        pygame.mixer.init()

        self.sound\_effects = {}

        self.bgm\_volume = 1

        self.sfx\_volume = 1

        self.current\_bgm: Path

    def load\_sound(self, name: str, path: Path):

        """Loads a sound effect into the sound effects

        Args:

            name (str): The name of the sound effect which will be used as the key in the sound effects dictionary

            path (Path): The path to the sound effect

        """

        self.sound\_effects[name] = Sound(path)

    def play\_sound(self, name: str):

        """Plays the specified sound effect

        Args:

            name (str): The name of the sound effect to be played

        """

        sound = self.sound\_effects.get(name)

        if sound:

            sound.set\_volume(self.sfx\_volume)

            sound.play()

    def set\_bgm(self, file: Path):

        """Sets the backgroudn music

        Args:

            file (Path): The path to background music

        """

        pygame.mixer.music.load(file)

        pygame.mixer.music.set\_volume(self.bgm\_volume)

        pygame.mixer.music.play(-1)

        self.current\_bgm = file

    def stop\_bgm(self):

        """Stops the background music from playing"""

        pygame.mixer.music.stop()

    def set\_bgm\_volume(self, volume: float):

        """Sets the volume of the background music

        Args:

            volume (float): The new volume for the background music

        """

        self.bgm\_volume = volume

        pygame.mixer.music.set\_volume(volume)

        pygame.mixer.music.unload()

        self.set\_bgm(self.current\_bgm)

    def set\_sfx\_volume(self, volume: float):

        """Sets the volume for all sound effects

        Args:

            volume (float): The new volume for all sound effects

        """

        self.sfx\_volume = volume

title.py

from pathlib import Path

from pygame.image import load

from pygame import Mask

from pygame import Rect

from pygame import Surface

from pygame.mask import from\_surface

class Title:

    image: Surface

    rect: Rect

    mask: Mask

    def \_\_init\_\_(self, title\_image\_path: Path):

        self.image = load(title\_image\_path).convert\_alpha()

        self.rect = self.image.get\_rect()

        self.mask = from\_surface(self.image)

        self.rect.topleft = 0, 0

    def center\_on\_x\_axis(self, x\_axis\_width: int):

        """Centres the buttons on the x-axis

        Args:

            x\_axis\_width (int): The width of the x-axis in pixels

        """

        self.rect.x = (x\_axis\_width - self.rect.width) // 2

background\_layer.py

from math import sin

from time import perf\_counter

from random import uniform

from pygame import Rect, Surface

class BackgroundLayer:

    image: Surface

    sine\_scale\_factor: float

    sine\_stretch\_factor: float

    sine\_translation\_factor: float

    x\_scroll: float

    def \_\_init\_\_(self, image: Surface):

        self.image = image

        self.sine\_scale\_factor = uniform(0.001, 0.1)

        self.sine\_stretch\_factor = pow(10, 20)

        self.sine\_translation\_factor = uniform(0.1, 0.7)

        self.x\_scroll = 0

    def get\_new\_sub\_image(self, x\_scroll: float = 0, y\_scroll: float = 0) -> Surface:

        """Generates a new scroll value to adjust which subsection of the image is returned

        Returns:

            Surface: The sub image according to the newly generated scroll value

        """

        self.x\_scroll += (

            self.sine\_scale\_factor \* abs(sin(perf\_counter() \* self.sine\_stretch\_factor))

            + self.sine\_translation\_factor

        ) + x\_scroll

        x\_scroll = int(self.x\_scroll) % (self.image.get\_width() // 2)

        handle\_image = self.image.copy()

        clip\_rect = Rect(

            x\_scroll, y\_scroll, self.image.get\_width() // 2, self.image.get\_height()

        )

        handle\_image.set\_clip(clip\_rect)

        return self.image.subsurface(handle\_image.get\_clip())

button.py

from os.path import join

from pathlib import Path

from pygame.image import load

from pygame import Mask

from pygame import Rect

from pygame import Surface

from pygame.mask import from\_surface

class Button:

    """Creates a button class that has a passive image, an image if the mouse is hovered over it and an image when the button has been clicked

    """

    passive\_image: Surface

    active\_image: Surface

    on\_click\_image: Surface

    current\_image: Surface

    rect: Rect

    mask: Mask

    can\_be\_clicked: bool

    clicked: bool

    click\_delay: float

    click\_timer: float

    def \_\_init\_\_(self, path: Path):

        self.passive\_image = load(join(path, "0.png")).convert\_alpha()

        self.active\_image = load(join(path, "1.png")).convert\_alpha()

        self.on\_click\_image = load(join(path, "2.png")).convert\_alpha()

        self.current\_image = self.passive\_image

        self.rect = self.passive\_image.get\_rect()

        self.mask = from\_surface(self.passive\_image)

        self.rect.topleft = 0, 0

        self.can\_be\_clicked = True

        self.clicked = False

        self.click\_delay = 0.08

        self.click\_timer = -1

    def update(self, mouse\_pos: tuple[int, int], mouse\_clicked: bool, delta\_time: float):

        """Updates the current\_image attribute and the clicked flag

        Args:

            mouse\_pos (tuple[int, int]): The position of the mouse on the screen

            mouse\_clicked (bool): A boolean value stating whether the mouse is clicked or not on the current frame

        """

        mouse\_pos\_x, mouse\_pos\_y = mouse\_pos

        mouse\_pos\_in\_mask = mouse\_pos\_x - self.rect.x, mouse\_pos\_y - self.rect.y

        self.clicked = False

        if self.rect.collidepoint(mouse\_pos) and self.mask.get\_at(mouse\_pos\_in\_mask):

            if mouse\_clicked and self.can\_be\_clicked:

                self.current\_image = self.on\_click\_image

                if self.click\_timer == -1:

                    self.click\_timer = 0

            else:

                self.current\_image = self.active\_image

        else:

            self.current\_image = self.passive\_image

        if self.click\_timer > self.click\_delay and self.can\_be\_clicked:

            self.clicked = True

            self.click\_timer = -1

        if self.click\_timer >= 0:

            self.click\_timer += delta\_time

    def center\_on\_x\_axis(self, x\_axis\_width: int):

        """Centres the buttons on the x-axis

        Args:

            x\_axis\_width (int): The width of the x-axis in pixels

        """

        self.rect.x = (x\_axis\_width - self.rect.width) // 2

camera.py

from pygame import Rect

class Camera:

    """

    Provides a camera class that can adjust the scroll values on both axis in order to focus and move to smoothly a specific pygame.Rect

    """

    height: int

    width: int

    display\_surface\_height: int

    display\_surface\_width: int

    scroll\_x: int

    scroll\_y: int

    trauma: float

    def \_\_init\_\_(

        self,

        height: int,

        width: int,

        display\_surface\_height: int,

        display\_surface\_width: int,

    ):

        self.height = height

        self.width = width

        self.display\_surface\_height = display\_surface\_height

        self.display\_surface\_width = display\_surface\_width

        self.scroll\_x = 0

        self.scroll\_y = 0

        self.trauma = 0

    def update(self, target\_rect: Rect):

        """Updates the scroll values of the camera to focus on the target

        Args:

            target\_rect (Rect): The rect for a given target

        """

        self.scroll\_x += int(

            (

                target\_rect.x

                - self.scroll\_x

                - (self.display\_surface\_width + target\_rect.width) // 2

            )

            \* 0.2

        )

        self.scroll\_y += int(

            (

                target\_rect.y

                - self.scroll\_y

                - (self.display\_surface\_height + target\_rect.height) // 2

            )

            \* 0.2

        )

        self.scroll\_x = min(

            max(0, self.scroll\_x), self.width - self.display\_surface\_width

        )

        self.scroll\_y = min(

            max(0, self.scroll\_y), self.height - self.display\_surface\_height

        )

circular\_queue.py

from numpy import empty

from typing import TypeVar, Generic, Iterator

T = TypeVar("T")

class CircularQueue(Generic[T]):

    """

    Provides a circular queue that has a maximum size and provids the methods associated with a circular queue

    """

    def \_\_init\_\_(self, max\_size: int, dtype: type[T]):

        self.max\_size = max\_size

        self.queue = empty(self.max\_size, dtype=dtype)

        self.front, self.rear = -1, -1

    def \_\_iter\_\_(self) -> Iterator[T]:

        return iter(self.queue)

    def is\_full(self) -> bool:

        """A method that returns True if the queue is full

        Returns:

            bool: Whether the queue is full or not

        """

        return ((self.rear + 1) % self.max\_size) == self.front

    def is\_empty(self) -> bool:

        """A method that returns True if the queue is empty

        Returns:

            bool: Whether the queue is empty or not

        """

        return self.front == -1

    def enqueue(self, data: T):

        """Adds an item to the end of the queue

        Args:

            data (T): The item to be added to the end of the queue

        Raises:

            FullQueue: If the queue is full then a FullQueue exception is raised

        """

        ###A: Queue Operation###

        if self.is\_full():

            raise FullQueue

        if self.is\_empty():

            self.front = 0

            self.rear = 0

        else:

            self.rear = (self.rear + 1) % self.max\_size

        self.queue[self.rear] = data

    def dequeue(self) -> T:

        """Removes the frontmost item from the queue and returns it

        Raises:

            EmptyQueue: If the queue is empty then an EmptyQueue exception is raised

        Returns:

            (T): The frontmost item in the queue

        """

        ###A: Queue Operation###

        if self.is\_empty():

            raise EmptyQueue

        temp = self.queue[self.front]

        if self.front == self.rear:

            self.front = -1

            self.rear = -1

        else:

            self.front = (self.front + 1) % self.max\_size

        return temp

class EmptyQueue(Exception):

    """Raised when a queue is empty"""

class FullQueue(Exception):

    """Raised when a queue is full"""

config.py

"""Provides an interface for storing and loading game wide values"""

from json import load

from pathlib import Path

from time import strftime

from typing import Any

class NeaGameConfig:

    """Handles loading of the config.json file"""

    debug: bool

    debug\_file: str

    directories: dict[str, Path]

    resoloution: tuple[int, int]

    internal\_resoloution: tuple[int, int]

    fps: int

    internal\_fps: int

    chunk\_size: int

    key\_bindings: list[int]

    music\_volume: float

    sfx\_volume: float

    unlocked\_levels: dict[str, bool]

    def \_\_init\_\_(self):

        # Debug

        self.debug = True

        debug\_filename = "Test"

        self.debug\_file = (strftime("%m-%d-%Y")) + "-" + debug\_filename + ".prof"

        # Directories

        platformer\_folder = Path(\_\_file\_\_).absolute().parent

        game\_folder = platformer\_folder.parent

        assets\_folder = game\_folder / "assets"

        background\_folder = assets\_folder / "background"

        gui\_folder = assets\_folder / "gui"

        music\_folder = assets\_folder / "music"

        player\_folder = assets\_folder / "player"

        sfx\_folder = assets\_folder / "sfx"

        worlds\_folder = assets\_folder / "worlds"

        self.directories = {

            "game": game\_folder,

            "assets": assets\_folder,

            "background": background\_folder,

            "gui": gui\_folder,

            "player": player\_folder,

            "music": music\_folder,

            "sfx": sfx\_folder,

            "worlds": worlds\_folder,

            "platformer": platformer\_folder,

        }

        # Display

        width = self.get\_int\_setting("x\_resolution")

        height = self.get\_int\_setting("y\_resolution")

        self.resoloution = (width, height)

        ds\_width = 384

        ds\_height = 216

        self.internal\_resoloution = (ds\_width, ds\_height)

        self.fps = self.get\_int\_setting("fps")

        self.internal\_fps = 30

        self.chunk\_size = 8

        self.key\_bindings = self.get\_int\_list\_setting("key\_bindings")

        self.music\_volume = self.get\_float\_setting("music\_volume")

        self.sfx\_volume = self.get\_float\_setting("sfx\_volume")

        self.unlocked\_levels = self.get\_setting("unlocked\_levels")

    def get\_int\_setting(self, setting: str) -> int:

        """\_summary\_

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [int]

        Returns:

            int: The value of the specified setting

        """

        ###B: Parsing from a JSON file###

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if isinstance(settings[setting], int):

                return settings[setting]

            raise ValueError

    def get\_float\_setting(self, setting: str) -> float:

        """\_summary\_

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [float]  or [int]

        Returns:

            list[int]: The value of the specified setting

        """

        ###B: Parsing from a JSON file###

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if isinstance(settings[setting], (float, int)):

                return settings[setting]

            raise ValueError

    def get\_int\_list\_setting(self, setting: str) -> list[int]:

        """Gets the relevant setting from config.json that is a list of integers

        Args:

            setting (str): the identifier of the setting in [config.json]

        Raises:

            ValueError: If the value of the specified setting is not of type [float] or [int]

        Returns:

            list[int]: The value of the specified setting

        """

        ###B: Parsing from a JSON file###

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            if all(isinstance(element, int) for element in settings[setting]):

                return settings[setting]

            raise ValueError

    def get\_setting(self, setting: str) -> Any:

        ###B: Parsing from a JSON file###

        with (self.directories["platformer"] / "config.json").open(

            mode="r"

        ) as settings\_json:

            settings = load(settings\_json)

            return settings[setting]

    def reload(self):

        """Reloads the settings from config.json"""

        width = self.get\_int\_setting("x\_resolution")

        height = self.get\_int\_setting("y\_resolution")

        self.resoloution = (width, height)

        self.fps = self.get\_int\_setting("fps")

        self.music\_volume = self.get\_float\_setting("music\_volume")

        self.sfx\_volume = self.get\_float\_setting("sfx\_volume")

        self.key\_bindings = self.get\_int\_list\_setting("key\_bindings")

        self.unlocked\_levels = self.get\_setting("unlocked\_levels")

action\_button.py

from pathlib import Path

import pygame

from pygame.event import Event

from pygame.image import load

from pygame.key import name as get\_key\_name

from pygame.mask import Mask, from\_surface

from pygame import Rect, Surface

class ActionButton:

    passive\_image: Surface

    active\_image: Surface

    current\_image: Surface

    rect: Rect

    mask: Mask

    key: int

    key\_images\_path: Path

    key\_image: Surface

    clicked: bool

    click\_time: float

    click\_timer: float

    def \_\_init\_\_(self, path: Path, key\_images\_path: Path, key: int):

        self.passive\_image = load(path / "0.png")

        self.active\_image = load(path / "1.png")

        self.current\_image = self.passive\_image

        self.rect = self.passive\_image.get\_rect()

        self.mask = from\_surface(self.passive\_image)

        self.rect.topleft = 0, 0

        self.key\_images\_path = key\_images\_path

        self.initial\_update\_key\_image(key)

        self.clicked = False

        self.click\_time = 5

        self.click\_timer = -1

    def update(

        self,

        mouse\_pos: tuple[int, int],

        mouse\_clicked: bool,

        delta\_time: float,

        other\_binds: list[int] | None = None,

        event: Event | None = None,

    ):

        """Updates the current\_image attribute and the clicked flag

        Args:

            mouse\_pos (tuple[int, int]): The position of the mouse on the screen

            mouse\_clicked (bool): A boolean value stating whether the mouse is clicked or not on the current frame

        """

        mouse\_pos\_x, mouse\_pos\_y = mouse\_pos

        mouse\_pos\_in\_mask = mouse\_pos\_x - self.rect.x, mouse\_pos\_y - self.rect.y

        self.clicked = False

        if (

            self.rect.collidepoint(mouse\_pos)

            and self.mask.get\_at(mouse\_pos\_in\_mask)

            and mouse\_clicked

            or self.click\_timer >= 0

        ):

            self.current\_image = self.active\_image

            self.clicked = True

            self.click\_timer = max(0, self.click\_timer)

        else:

            self.current\_image = self.passive\_image

        if self.click\_timer >= 0:

            self.click\_timer += delta\_time

            if event:

                if event.type == pygame.KEYDOWN:

                    if self.update\_key\_image(event.key, other\_binds):

                        self.click\_timer = -1

        if self.click\_timer >= self.click\_time:

            self.click\_timer = -1

    def initial\_update\_key\_image(self, key: int) -> bool:

        """Updates the key image for the initial key bind

        Args:

            key (int): The int value for the key

        Returns:

            bool: Whether the key image was updated successfully

        """

        key\_name = get\_key\_name(key).replace(" ", "\_")

        try:

            self.key\_image = load(self.key\_images\_path / f"{key\_name}-key.png")

        except FileNotFoundError:

            return False

        handle\_key\_image = self.key\_image.copy()

        clip\_rect = Rect(

            self.key\_image.get\_width() // 2,

            0,

            self.key\_image.get\_width() // 2,

            self.key\_image.get\_height(),

        )

        handle\_key\_image.set\_clip(clip\_rect)

        self.key = key

        self.key\_image = self.key\_image.subsurface(

            handle\_key\_image.get\_clip()

        ).convert\_alpha()

        return True

    def update\_key\_image(self, key: int, other\_binds: list[int] | None = None) -> bool:

        """Updates the key image given a keybind and a list of the other key binds

        Args:

            key (int): The int value of the key

            other\_binds (list[int] | None, optional): The other key bindings. Defaults to None.

        Returns:

            bool: Whether the key image was updated successfully

        """

        if other\_binds is None:

            return False

        if key in other\_binds:

            return False

        key\_name = get\_key\_name(key).replace(" ", "\_")

        try:

            self.key\_image = load(self.key\_images\_path / f"{key\_name}-key.png")

        except FileNotFoundError:

            return False

        handle\_key\_image = self.key\_image.copy()

        clip\_rect = Rect(

            self.key\_image.get\_width() // 2,

            0,

            self.key\_image.get\_width() // 2,

            self.key\_image.get\_height(),

        )

        handle\_key\_image.set\_clip(clip\_rect)

        self.key = key

        self.key\_image = self.key\_image.subsurface(

            handle\_key\_image.get\_clip()

        ).convert\_alpha()

        return True

slider.py

from pathlib import Path

from pygame.image import load

from pygame.mask import from\_surface

from pygame import Mask, Rect, Surface

class Slider:

    """

    A class that allows a slider to be used for the user to specify a valeu between two endpoints

    """

    passive\_bar: Surface

    active\_bar: Surface

    handle\_image: Surface

    rect: Rect

    mask: Mask

    bar\_rect: Rect

    handle\_rect: Rect

    min\_value: int

    max\_value: int

    value: int

    dragging: bool

    def \_\_init\_\_(

        self,

        path: Path,

        min\_value: int,

        max\_value: int,

        value: int,

    ):

        self.passive\_bar = load(path / "0.png")

        self.active\_bar = load(path / "1.png")

        self.handle\_image = load(path / "2.png")

        self.rect = self.passive\_bar.get\_rect()

        self.mask = from\_surface(self.passive\_bar)

        self.bar\_rect = self.active\_bar.get\_rect()

        self.handle\_rect = self.handle\_image.get\_rect()

        self.min\_value = min\_value

        self.max\_value = max\_value

        self.value = value

        self.set\_value(value)

        self.dragging = False

    def set\_topleft(self, x: int, y: int):

        """Sets the topleft of both parts of the slider

        Args:

            x (int): The x position of the slider

            y (int): The y position of the slider

        """

        self.rect.topleft = x, y

        self.bar\_rect.topleft = x - 1, y - 1

    def update(self, mouse\_pos: tuple[int, int], mouse\_clicked: bool):

        """Updates the value of the slider based on whether the slider is being dragged

        Args:

            mouse\_pos (tuple[int, int]): The position of the mouse on the screen

            mouse\_clicked (bool): A boolean value stating whether the mouse is clicked or not on the current frame

        """

        mouse\_pos\_x, mouse\_pos\_y = mouse\_pos

        mouse\_pos\_in\_mask = mouse\_pos\_x - self.rect.x, mouse\_pos\_y - self.rect.y

        self.dragging = False

        self.dragging = (

            self.rect.collidepoint(mouse\_pos)

            and self.mask.get\_at(mouse\_pos\_in\_mask) == 1

            and mouse\_clicked

        )

        if self.dragging:

            self.set\_value(int(mouse\_pos\_x - self.bar\_rect.left))

    @property

    def current\_image(self) -> Surface:

        """Returns one surface showing the current value of the slider

        Returns:

            Surface: The current image of the slider

        """

        current\_image = self.passive\_bar.copy()

        current\_image.blit(self.passive\_bar, self.bar\_rect.topleft)

        current\_image.blit(

            self.active\_bar.subsurface(

                Rect(

                    0,

                    0,

                    (self.value / self.max\_value) \* self.bar\_rect.width,

                    self.bar\_rect.height,

                )

            ),

            (1, 1),

        )

        current\_image.blit(

            self.handle\_image,

            (

                int((self.value / self.max\_value) \* self.bar\_rect.width)

                - self.handle\_rect.width

                + 1,

                1,

            ),

        )

        return current\_image

    def set\_value(self, value: int):

        """Sets the value of the slider

        Args:

            value (int): The new value of the slider

        """

        self.value = max(self.min\_value, min(self.max\_value, value))

world.py

from json import load

from typing import Any

from pathlib import Path

from nea\_game.ldtk\_world\_loader.level import Level

from nea\_game.ldtk\_world\_loader.tileset import Tileset

class World:

    """A class that generates a world data based on world.json file from the LDTK editor

    """

    data: dict[str, Any]

    tileset: Tileset

    levels: dict[str, Level]

    def \_\_init\_\_(self, world\_identifier: str, world\_directory: Path, chunk\_size: int):

        with (world\_directory / f"{world\_identifier}.json").open() as world\_json:

            self.data = load(world\_json)

        self.tileset = Tileset(self.data["defs"]["tilesets"][0], world\_directory)

        levels = [

            Level(level\_data, self.tileset, chunk\_size)

            for level\_data in self.data["levels"]

        ]

        self.levels = {level.identifier: level for level in levels}

level.py

from typing import Any

from nea\_game.entity.level\_finish import LevelFinish

from nea\_game.ldtk\_world\_loader.level\_data import LevelData

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

from nea\_game.ldtk\_world\_loader.tileset import Tileset

class Level:

    data: dict[str, Any]

    identifier: str

    tileset: Tileset

    height: int

    width: int

    chunk\_size: int

    level\_data: LevelData

    def \_\_init\_\_(self, data: dict[str, Any], tileset: Tileset, chunk\_size: int):

        self.data = data

        self.identifier: str = data["identifier"]

        self.tileset = tileset

        self.height = self.data["pxHei"]

        self.width = self.data["pxWid"]

        self.chunk\_size = chunk\_size

        self.level\_data = self.generate\_level\_data()

    def generate\_level\_data(self) -> LevelData:

        """Generates a LevelData object based on the level definition in the world.json file

        Raises:

            ValueError: Raised if the player position has not been specified

            ValueError: Raised if there is no level finish object

        Returns:

            LevelData: The data for a given level

        """

        tiles: list[LevelTile] = []

        chunks: dict[tuple[int, int], list[LevelTile]] = {}

        player\_position: tuple[int, int] = (-1, -1)

        level\_finish = None

        for layer in self.data["layerInstances"]:

            match layer["\_\_identifier"]:

                case "AutoTiles":

                    for tile in layer["autoLayerTiles"]:

                        tiles.append(

                            LevelTile.from\_tileset\_tile(

                                self.tileset.tiles[tile["t"]], tile["px"]

                            )

                        )

                case "Tiles":

                    for tile in layer["gridTiles"]:

                        tiles.append(

                            LevelTile.from\_tileset\_tile(

                                self.tileset.tiles[tile["t"]], tile["px"]

                            )

                        )

                case "Entities":

                    for entity\_instance in layer["entityInstances"]:

                        match entity\_instance["\_\_identifier"]:

                            case "Player":

                                player\_position = entity\_instance["px"]

                            case "Finish":

                                new\_world = entity\_instance["fieldInstances"][0][

                                    "\_\_value"

                                ]

                                next\_level\_identifier = entity\_instance[

                                    "fieldInstances"

                                ][1]["\_\_value"]

                                level\_finish = LevelFinish(

                                    entity\_instance["px"],

                                    entity\_instance["height"],

                                    entity\_instance["width"],

                                    new\_world,

                                    next\_level\_identifier,

                                )

                            case \_:

                                pass

                case \_:

                    pass

        if player\_position == (-1, -1):

            raise ValueError("Player Position is not specified")

        if level\_finish is None:

            raise ValueError("Level Finish is not specified")

        for tile in tiles:

            chunk\_x = tile.rect.x // self.chunk\_size

            chunk\_y = tile.rect.y // self.chunk\_size

            if (chunk\_x, chunk\_y) not in chunks:

                chunks[(chunk\_x, chunk\_y)] = [tile]

            else:

                chunks[(chunk\_x, chunk\_y)].append(tile)

        return {

            "chunks": chunks,

            "player\_position": player\_position,

            "level\_finish": level\_finish,

        }

tileset.py

from pathlib import Path

from typing import Any

from pygame.image import load

from pygame import Rect, Surface

from nea\_game.ldtk\_world\_loader.collision\_type import CollisionType

from nea\_game.ldtk\_world\_loader.tileset\_tile import TilesetTile

class Tileset:

    """A class that stores the relevent data about a tileset

    """

    grid\_height: int

    grid\_width: int

    grid\_size: int

    spacing: int

    padding: int

    image: Surface

    collision\_types: list[dict[str, Any]]

    tiles: dict[int, TilesetTile]

    def \_\_init\_\_(self, data: dict[str, Any], world\_path: Path):

        self.grid\_height = data["\_\_cHei"]

        self.grid\_width = data["\_\_cWid"]

        self.grid\_size = data["tileGridSize"]

        self.spacing = data["spacing"]

        self.padding = data["padding"]

        self.image = load(world\_path / data["relPath"])

        self.collision\_types: list[dict[str, Any]] = data["enumTags"]

        tiles: dict[int, str] = {}

        for collision\_type in self.collision\_types:

            for tile\_id in collision\_type["tileIds"]:

                tiles[tile\_id] = collision\_type["enumValueId"]

        self.tiles = {}

        for tile\_id, collision\_type in tiles.items():

            grid\_x = tile\_id - (self.grid\_width \* (tile\_id // self.grid\_width))

            pixel\_x = self.padding + (grid\_x \* (self.grid\_size + self.spacing))

            grid\_y = tile\_id // self.grid\_width

            pixel\_y = self.padding + (grid\_y \* (self.grid\_size + self.spacing))

            handle\_image = self.image.copy()

            clip\_rect = Rect(pixel\_x, pixel\_y, self.grid\_size, self.grid\_size)

            handle\_image.set\_clip(clip\_rect)

            tile\_image = self.image.subsurface(handle\_image.get\_clip())

            if getattr(CollisionType, collision\_type.upper()) == CollisionType.PLATFORM:

                self.tiles[tile\_id] = TilesetTile(

                    tile\_id,

                    tile\_image,

                    getattr(CollisionType, collision\_type.upper()),

                    Rect((0, 0), (3, tile\_image.get\_height())),

                )

                self.tiles[tile\_id] = TilesetTile(

                    tile\_id,

                    tile\_image,

                    getattr(CollisionType, collision\_type.upper()),

                    Rect((0, 0), (3, tile\_image.get\_height())),

                )

            else:

                self.tiles[tile\_id] = TilesetTile(

                    tile\_id, tile\_image, getattr(CollisionType, collision\_type.upper())

                )

lerp.py

"""Defines a linear interpolation function

"""

def lerp(flt\_1: float, flt\_2: float, lerp\_amount: float) -> float:

    """Linearly interpolates between two points.

    Interpolates between the points a and b by the interpolant lerp\_amount.

    The parameter lerp\_amount is clamped to the range [0, 1].

    This is most commonly used to find a point some fraction of the way along a line between two endpoints

        Args:

            a (float): Endpoint 1

            b (float): Endpoint 2

            lerp\_amount (float): The interpolent

        Returns:

            float: The interpolated value

    """

    return (flt\_1 \* (1 - lerp\_amount)) + (flt\_2 \* lerp\_amount)

near\_zero.py

"""Defines a thresholod and a funtion to approximate floats close to zero

"""

from math import isclose

from functools import partial

THRESHOLD = 1e-1

is\_close\_to\_zero = partial(isclose, b=0, abs\_tol=THRESHOLD)

def near\_zero(flt: float) -> float:

    """If a float is close to zero then zero is returned othewise the flaot itself is returned

    Args:

        flt (float): The float to be approximated

    Returns:

        float: The approximated value of the float

    """

    if is\_close\_to\_zero(flt):

        return 0

    return flt

sign.py

"""Returns the sign of flt. Return value is 1 when flt is positive or zero, -1 when flt is negative.

"""

def sign(flt: float) -> float:

    """Returns the sign of flt.

       Return value is 1 when flt is positive or zero, -1 when flt is negative.

    Args:

        flt (float): The value to determine the sign of

    Returns:

        float: The sign of flt

    """

    return 1 if flt >= 0 else -1

player\_action\_space.py

from enum import Enum

class PlayerActionSpace(Enum):

    UP = 0

    DOWN = 1

    LEFT = 2

    RIGHT = 3

    DASH = 4

collision\_type.py

"""An enum for the different collision types a tile can have

"""

from enum import Enum, auto

class CollisionType(Enum):

    """An enum for the collision types a tile can have"""

    WALL = auto()

    SPIKE = auto()

    PLATFORM = auto()

level\_data.py

from typing import TypedDict

from nea\_game.entity.level\_finish import LevelFinish

from nea\_game.ldtk\_world\_loader.level\_tile import LevelTile

class LevelData(TypedDict):

    """

    A typed dictionary to store the data about a level

    """

    chunks: dict[tuple[int, int], list[LevelTile]]

    player\_position: tuple[int, int]

    level\_finish: LevelFinish