**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),

            )