Key Code Descriptions

This section provides an expanded and detailed explanation of the critical segments of the Space Invaders code. Each feature is discussed in-depth, focusing on the reasoning behind the code choices and the functionality of its components. Unique mechanisms, such as the homing bullets, are highlighted to illustrate the advanced elements of the game.

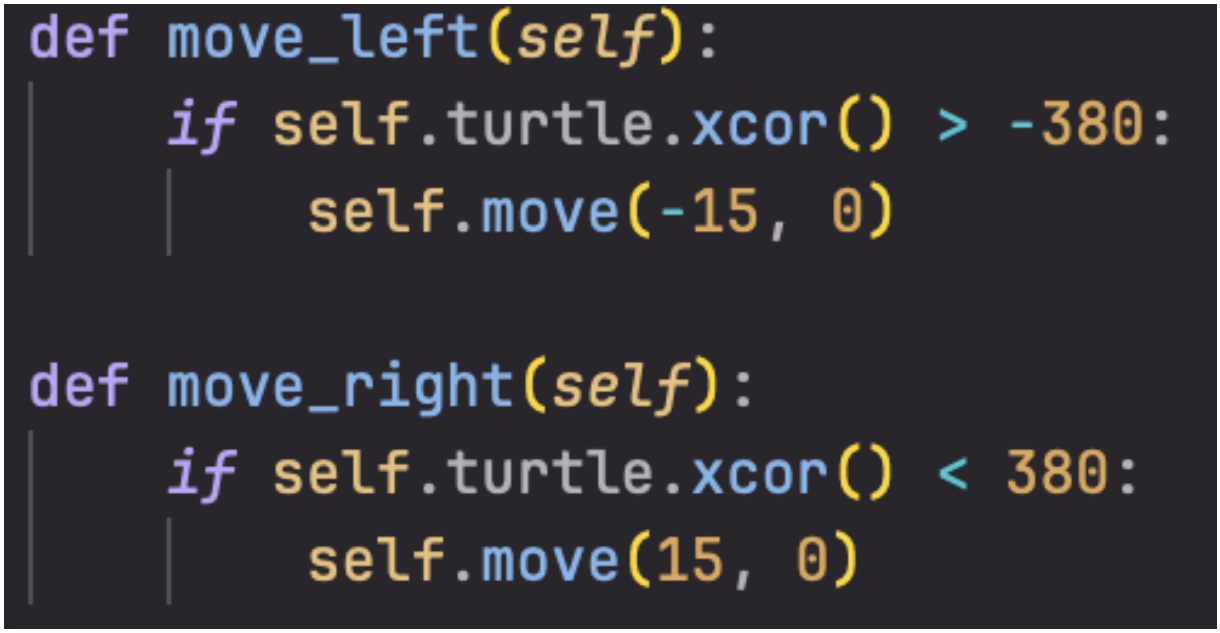
Player Spaceship: Movement and Shooting

Choice of Code

The player spaceship's movement and shooting capabilities are fundamental to the gameplay. These features allow players to interact with enemies, dodge incoming fire, and collect power-ups. Smooth movement and precise shooting mechanics are essential for creating an engaging and balanced gaming experience.

Code Breakdown

- **Player Movement**:
    - The `move_left` and `move_right` functions restrict the spaceship's movement to the horizontal plane within the screen's boundaries.
    - The boundary checks (`self.turtle.xcor() > -380` and `self.turtle.xcor() < 380`) ensure the spaceship remains visible and accessible to the player.



- **Shooting Mechanics**:

- The `fire_bullet` function creates a new `Bullet` object and adds it to the `player_bullets` list.
- The cooldown mechanism (`current_time - last_bullet_time >= bullet_cooldown`) ensures that the player cannot shoot excessively, maintaining game balance.

```
def fire_bullet():
    global last_bullet_time
    current_time = time.time()
    if current_time - last_bullet_time >= bullet_cooldown:
        if double_bullets:
            bullet1 = Bullet(player.turtle.xcor() - 10, player.turtle.ycor() + 20, 'red', 90)
            bullet2 = Bullet(player.turtle.xcor() + 10, player.turtle.ycor() + 20, 'red', 90)
            player_bullets.extend([bullet1, bullet2])
        else:
            bullet = Bullet(player.turtle.xcor(), player.turtle.ycor() + 20, 'red', 90)
            player_bullets.append(bullet)
        last_bullet_time = current_time
```

Alien Spaceships: Movement and Attacks

Choice of Code

The aliens' coordinated movement and projectile attacks emulate the behavior of classic Space Invaders. This design introduces a structured challenge that becomes progressively harder as waves advance.

Code Breakdown

- **Organized Alien Movement**:
  - The `move` function updates the position of each alien by adding a displacement (`dx`, `dy`) to its current coordinates.
  - This behavior creates a predictable, yet increasingly challenging, pattern.

```
def move(self, dx, dy):
    if self.alive:
        self.turtle.setx(self.turtle.xcor() + dx)
        self.turtle.sety(self.turtle.ycor() + dy)
```

- **Alien Projectile Firing**:

- The `fire_alien_bullets` function uses a random chance (`random.random() < fire_probability`) to determine whether an alien fires a projectile.
- As waves progress, the `fire_probability` increases, escalating the challenge.

```
# Alien/Boss firing logic
    if current_time - last_alien_fire_time >= alien_fire_cooldown:
        if boss_alien and boss_alien.alive:
            fire_probability = 0.8
            if random.random() < fire_probability:
                alien_bullet = Bullet(
                    boss_alien.turtle.xcor(),
                    boss_alien.turtle.ycor() - 20,
                    'red',
                    270,
                    curve_probability=1.0
                )
                alien_bullets.append(alien_bullet)
            last_alien_fire_time = current_time - (alien_fire_cooldown / 2)
        else:
            fire_probability = 0.1 + (wave_number * 0.01)
            fire_probability = min(fire_probability, 0.5)
            for alien in aliens:
                if random.random() < fire_probability:
                    alien_bullet = Bullet(alien.turtle.xcor(), alien.turtle.ycor() - 20, 'yellow', 270)
                    alien_bullets.append(alien_bullet)
            last_alien_fire_time = current_time
```

Homing Bullets and Advanced Behavior

Choice of Code

Homing bullets add complexity by requiring players to actively dodge attacks. This feature enhances gameplay dynamics by making enemy attacks less predictable and more engaging.

Homing Bullet Trajectory

The `move_forward` function is responsible for controlling the movement of homing bullets, a feature that adds strategic complexity to the game by allowing certain bullets to track the player's position. This functionality is primarily used by the boss alien but can also be applied to other advanced enemies.

Code Breakdown:

- **Curving Activation**: The function determines whether the bullet will curve based on a probability value (`curve_probability`). Once activated, the bullet's color changes to red, signaling its new behavior.

- **Target Angle Calculation**: The function computes the angle between the bullet and the player's current position using trigonometric functions like `math.atan2`. This ensures accurate tracking.
- **Curving Control**: To prevent erratic movements, the angle difference between the current heading and the target angle is constrained within a predefined range (`self.max_total_turn`). The bullet adjusts its trajectory incrementally using a `max_turn_angle` to maintain smoothness.
- **Forward Movement**: After updating its heading, the bullet continues moving forward at its set speed, creating a dynamic and persistent threat.
- **Max Total Turn:** Limiting the total turning angle for the bullet to 45° to prevent the bullet from being undodgeable

```
def move_forward(self, target_x=None, target_y=None):
    # Decide whether to curve only once per bullet
    if not self.curve_initiated and self.curve_probability > 0:
        if random.random() < self.curve_probability:
            self.curving = True
            self.turtle.color('red')  # Change to red when it starts curving
        self.curve_initiated = True


    if self.curving and target_x is not None and target_y is not None:
        dx = target_x - self.turtle.xcor()
        dy = target_y - self.turtle.ycor()
        angle_to_target = math.degrees(math.atan2(dy, dx))


        angle_difference = (angle_to_target - self.initial_heading + 360) % 360
        if angle_difference > 180:
            angle_difference -= 360


        angle_difference = max(-self.max_total_turn, min(self.max_total_turn, angle_difference))
        desired_heading = self.initial_heading + angle_difference


        current_heading = self.turtle.heading()
        angle_diff_current = (desired_heading - current_heading + 360) % 360
         if angle_diff_current > 180:
            angle_diff_current -= 360


        max_turn_angle = 2
        turn_angle = max(-max_turn_angle, min(max_turn_angle, angle_diff_current))
        new_heading = current_heading + turn_angle
        self.turtle.setheading(new_heading)
    self.turtle.forward(self.speed)
```

```
self.max_total_turn = 45  # Max total angle bullet can turn
```

Boss Mechanics:

- The boss alien's health bar, updated by `update_boss_health_bar`, provides players with clear feedback on their progress during battles.

```python
def update_boss_health_bar():
    boss_health_bar.clear()
    if boss_alien and boss_alien.alive:
        boss_health_bar.showturtle()
        boss_health_bar.penup()


        # Position the health bar relative to the boss's current position
        boss_x = boss_alien.turtle.xcor()
        boss_y = boss_alien.turtle.ycor()


        # Adjust the offset as needed (e.g., 40 pixels below the boss)
        x = boss_x - 100  # Starting 100 pixels to the left of the boss
        y = boss_y - 40  # 40 pixels below the boss


        boss_health_bar.goto(x, y)
        boss_health_bar.color("red")
        boss_health_bar.width(3)
        boss_health_bar.setheading(0)
        boss_health_bar.pendown()


        max_health = 10
        health_ratio = boss_alien.health / max_health
        # 200 pixels wide at full health
        boss_health_bar.forward(200 * health_ratio)
```

Collision Detection

Choice of Code

Collision detection is vital for ensuring realistic interactions between objects, such as bullets and spaceships. Accurate detection provides immediate feedback and consequences for player and enemy actions.

Code Breakdown

- **Collision Calculation**:
    - The `is_collision` function calculates the distance between two objects using the Euclidean formula.
    - If the distance is less than the threshold (`< 20`), a collision is registered, triggering appropriate game events.

```python
def is_collision(obj1, obj2):
    if obj1.alive and obj2.alive:
        distance = math.hypot(
            obj1.turtle.xcor() - obj2.turtle.xcor(),
            obj1.turtle.ycor() - obj2.turtle.ycor()
        )
        return distance < 20  # Collision threshold
    return False
```

Power-Ups: Enhancing Gameplay

Choice of Code

Power-ups provide variety and strategic opportunities for players to enhance their performance temporarily. Randomized spawning ensures unpredictability and rewards exploration.

Code Breakdown

- **Power-Up Application**:
    - The `apply_powerup` function modifies global variables based on the power-up type (e.g., `bullet_cooldown` for faster fire, `lives` for extra life).
    - The effects are time-limited, reverting to default settings after expiration.

```python
def apply_powerup(powerup_type):
    global bullet_cooldown, double_bullets, lives, powerup_end_times, score_multiplier
    current_time = time.time()
    if powerup_type == 'faster_fire':
        bullet_cooldown = 0.1
        powerup_end_times['faster_fire'] = current_time + powerup_duration
    elif powerup_type == 'double_bullets':
        double_bullets = True
        powerup_end_times['double_bullets'] = current_time + powerup_duration
    elif powerup_type == 'extra_life':
        lives += 1
        update_score_display()
```