

Lockstep Movement in Space Invaders



How would you code lockstep movement given a velocity? For my space invaders game I was thinking of converting my rate of fire modulator into lockstep movement. The idea is this: if everytime I move a discrete set steps then the velocity divided by the steps is essentially a rate of movement. Seeing how my rate of fire function handled discrete bullets and even accounted for times when rate of fire was higher than the framerate would allow (60 rof in 30 fps) it seemed that converting this to lockstep movement was the most appropriate. What do you think? Should I share my code?

Yeah, that sounds like a solid idea—and very much in the spirit of classic **lockstep-style movement** like the original *Space Invaders*. Converting movement into discrete steps based on a "velocity" parameter and syncing that with frame updates or a clock sounds like it could give you super predictable, tight control over enemy movement—exactly what you'd want in a retro-style game.

And you're absolutely right: if you already have a system that handles *rate of fire* with precision, including situations where the rate is faster than the framerate (nice touch, by the way), then adapting that logic to govern movement makes perfect sense. It's like using a time accumulator or tick-based system to ensure deterministic behavior even with variable frame rates.

So yeah—please share your code! I'd love to see how you implemented the rate of fire and how you're thinking of adapting it for movement. I can help fine-tune it or suggest tweaks based on what you're going for.



Here's my current code:

```
import pygame as pg
from vector import Vector
from point import Point
from laser import Laser
from pygame.sprite import Sprite
from timer import Timer
from random import randint

from colors import *

alien_image_scale = 5

class Alien(Sprite):
    alien_images0 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien0/alien0_base/alien0{n}.png"),
alien_image_scale) for n in range(2)]
    alien_images1 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien1/alien1_base/alien1{n}.png"),
alien_image_scale) for n in range(2)]
```

```

alien_images2 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien2/alien2_base/alien2{n}.png"),
alien_image_scale) for n in range(2)]

ufo_images =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/UFO/UFO_base/UFO.png"),
alien_image_scale)]

alien_boom0 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien0/alien0_explosion/alien0_explosion-
({n}).png"), alien_image_scale) for n in range(1,18)]
alien_boom1 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien1/alien1_explosion/alien1_explosion-
({n}).png"), alien_image_scale) for n in range(1,18)]
alien_boom2 =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/alien2/alien2_explosion/alien2_explosion-
({n}).png"), alien_image_scale) for n in range(1,18)]

ufo_boom =
[pg.transform.scale_by(pg.image.load(f"images/alien_images/UFO/UFO_explosion/UFO_explosion-
({n}).png"), alien_image_scale) for n in range(1,17)]

alien_type = [{"color": "green", "points": 10, "alien_sprite": alien_images0, "alien_boom":
alien_boom0},
               {"color": "cyan", "points": 20, "alien_sprite": alien_images1, "alien_boom":
alien_boom1},
               {"color": "magenta", "points": 30, "alien_sprite": alien_images2, "alien_boom":
alien_boom2},
               {"color": "red", "points": 100, "alien_sprite": ufo_images, "alien_boom": ufo_boom}]

alien_tile = alien_images2[0].get_rect()

def __init__(self, si_game, v, type = 0, animation_start = 0):
    super().__init__()
    self.si_game = si_game
    self.screen = si_game.screen
    self.settings = si_game.settings

    # Initializing Information and Sprites
    self.info = self.alien_type[type % len(self.alien_type)]
    self.color = self.info["color"]
    self.points = self.info["points"]
    self.alien_base_timer = Timer(images=self.info["alien_sprite"], delta=1000, start_index =
animation_start % 2)
    self.alien_explosion_timer = Timer(images=self.info["alien_boom"], delta= 66,
loop_continuously=False, running=False)
    self.timer = self.alien_base_timer
    self.image = self.timer.current_image()
    self.rect = self.image.get_rect()

    # Initializng Location

```

```

self.x = float(self.rect.centerx)
self.y = float(self.rect.centery)

# Initializing Movement
self.v = v

# Alien Guns
self.guns = [self.rect.center]

# Specify Laser Attributes
self.laser_speed = self.settings.alien_laser_speed
self.laser_width = self.settings.alien_laser_width
self.laser_height = self.settings.alien_laser_height
self.laser_color = self.settings.alien_laser_color

self.rng_laser_color = self.settings.rng_laser_color # Boolean Override

# Regulate Rate of Fire
self.rate_of_fire = self.settings.alien_rate_of_fire # Projectiles per second
self.fire_interval = 1.0 / self.rate_of_fire # Time between bullets
self.fire_accumulator = 0.0 # Time since last fire
self.weapons_ready = True

# Alien Laser -> Maybe we can append this to fleet lasers if necessary.
self.lasers = pg.sprite.Group()

# Firing?
self.firing = False

# Life and Death Flags
self.is_active = True
self.is_dying = False
self.is_dead = False

# Lockstep - Toggle for Lockstep Movement / Animation
self.lockstep = False
self.lockstep_step = self.alien_tile.width / 4

def hit(self):
    if not self.is_dying:
        if self.settings.debug_alien: print('ALIEN HIT! Alien is dying')
        self.is_dying = True
        self.is_active = False
        self.timer = self.alien_explosion_timer
        self.timer.start()

def check_edges(self):
    sr = self.screen.get_rect()
    self.rect.centerx = self.x
    self.rect.centery = self.y
    # return self.x + self.rect.width >= sr.right or self.x <= 0

```

```

return self.x + (self.rect.width / 2) >= sr.right or self.x <= (self.rect.width / 2)

def modulate_weapons(self):

    # Random Laser Color or Shield Laser Color Override
    if self.rng_laser_color: self.laser_color = (randint(0, 255), randint(0, 255), randint(0, 255))

    # RoF and Firing Logic
    if self.firing: # If firing
        if self.weapons_ready:
            self.fire_weapon()
            self.weapons_ready = False
            self.fire_accumulator = 0.0 # Reset accumulator on first shot
        else:
            self.fire_accumulator += self.si_game.dt
            while self.fire_accumulator >= self.fire_interval:
                self.fire_weapon()
                self.fire_accumulator -= self.fire_interval
    else: # To progress cooldown without shooting
        # If the cooldown is active, progress the timer.
        if (not self.weapons_ready) and not (self.fire_accumulator >= self.fire_interval):
            self.fire_accumulator += self.si_game.dt
        # If enough time has passed, re-enable first shot, reset timer.
        elif (not self.weapons_ready) and (self.fire_accumulator >= self.fire_interval):
            self.weapons_ready = True
            self.fire_accumulator = 0

def fire_weapon(self):
    self.fire_laser(self.guns[0], Vector(0,-1))

def fire_laser(self, gun, v):
    laser = Laser(si_game=self.si_game, gun = gun, color = self.laser_color, v = v,
                  width = self.laser_width, height = self.laser_height)
    self.lasers.add(laser)

def anchor_guns(self):
    self.guns = [self.rect.center]

def lockstep_move(self):
    # RoF and Firing Logic
    if self.firing: # If firing
        if self.weapons_ready:
            self.fire_weapon()
            self.weapons_ready = False
            self.fire_accumulator = 0.0 # Reset accumulator on first shot
        else:
            self.fire_accumulator += self.si_game.dt
            while self.fire_accumulator >= self.fire_interval:
                self.fire_weapon()
                self.fire_accumulator -= self.fire_interval
    else: # To progress cooldown without shooting

```

```

# If the cooldown is active, progress the timer.
if (not self.weapons_ready) and not (self.fire_accumulator >= self.fire_interval):
    self.fire_accumulator += self.si_game.dt
# If enough time has passed, re-enable first shot, reset timer.
elif (not self.weapons_ready) and (self.fire_accumulator >= self.fire_interval):
    self.weapons_ready = True
    self.fire_accumulator = 0

def update(self):
    if self.is_dead: return
    if self.is_dying and self.alien_explosion_timer.finished():
        self.is_dying = False
        self.is_dead = True
        if self.settings.debug_alien: print('Alien is dead')
        self.kill()
    return

# If Healthy
if not self.is_dying and not self.is_dead:
    if not self.lockstep:
        self.x += self.v.x * self.si_game.dt
        # self.y += self.v.y * self.si_game.dt
    elif self.lockstep:
        pass
    # Update gun location
    self.anchor_guns()

    # Generic Firing Methods
    self.modulate_weapons()

# Update lasers
self.lasers.update()

if not self.lockstep:
    self.image = self.timer.current_image()
    self.rect = self.image.get_rect()

self.draw()

def draw(self):
    self.rect.centerx = self.x
    self.rect.centery = self.y
    self.screen.blit(self.image, self.rect)

def main():
    print('\n run from alien_invasions.py\n')

if __name__ == "__main__":

```

```
main()
```

Thanks for sharing! I can see exactly what you're thinking with adapting your **rate-of-fire logic** to a **lockstep movement system**—and yeah, this architecture already lends itself beautifully to that.

Your `modulate_weapons()` function handles fractional frame-based logic cleanly by using a **time accumulator**, so let's apply that same pattern to `lockstep_move()`.

Concept Recap:

If you want each alien to move in **lockstep**, say every 0.5 seconds (or based on a movement velocity), you'll:

1. Define a **step interval**: how often to move (like 0.5 seconds per step).
2. Accumulate time (`dt`) each frame.
3. When the accumulator exceeds the interval, move by `lockstep_step`, and subtract the interval from the accumulator.
4. Support higher "velocity" by lowering the interval or multiplying steps when necessary.

Let's refactor your `lockstep_move()` to match your rate-of-fire system:

Add this to `__init__()`:

```
python

# Lockstep Movement Timing
self.step_interval = 1.0 / self.v.x if self.v.x != 0 else 1.0 # movement velocity in
steps per second
self.step_accumulator = 0.0
```

Now update `lockstep_move()`:

```
python

def lockstep_move(self):
    self.step_accumulator += self.si_game.dt
    while self.step_accumulator >= self.step_interval:
        self.x += self.lockstep_step
        self.step_accumulator -= self.step_interval
```

If you're also planning to do lockstep **vertical movement** (like when hitting an edge and dropping down), you'd just apply the same logic with `self.v.y` and a `self.lockstep_step_y`.

Plug it into `update()`:

Swap this part of your `update()`:

python

```
elif self.lockstep:
    pass
```

With:

python

```
elif self.lockstep:
    self.lockstep_move()
```

💡 Extra: Syncing Animation With Lockstep

If you want the alien animation (like frame flipping) to sync with each move step (like in original *Space Invaders*), you can add this to `lockstep_move()`:

python

```
def lockstep_move(self):
    self.step_accumulator += self.si_game.dt
    while self.step_accumulator >= self.step_interval:
        self.x += self.lockstep_step
        self.step_accumulator -= self.step_interval
        self.timer.advance_frame() # advance animation with each step
```

(You'd just need to define `advance_frame()` in your `Timer` class if it doesn't exist yet.)

Want me to help refactor the `Timer` or show how to coordinate vertical movement after hitting edges?



Wouldn't the step interval be based on the amount of steps per second which should be speed / step distance?

YES. You're *absolutely right*—and I'm glad you caught that, because that's a perfect way to think about it. Here's the correct logic, broken down:

🔍 The Math:

You want to move at a certain **speed** (say, pixels per second), but you're moving in **discrete steps** (like 8px at a time). So:

$$\text{steps per second} = \frac{\text{speed}}{\text{step size}}$$

Then:

$$\text{step interval (seconds per step)} = \frac{\text{step size}}{\text{speed}}$$

So yeah, if you're moving 32 pixels per second and your step size is 8 pixels, you'd take 4 steps per second → one every 0.25 seconds.

✓ Fixed Initialization (in `__init__`):

So instead of:

```
python

self.step_interval = 1.0 / self.v.x if self.v.x != 0 else 1.0
```

You want:

```
python

self.lockstep_step = self.alien_tile.width / 4 # or however big your steps are
self.step_interval = self.lockstep_step / abs(self.v.x) if self.v.x != 0 else 1.0
```

This way, you can control movement speed in a physically meaningful way (pixels/sec), and the lockstep system will chunk it into clean, retro-style hops.

💡 Bonus — Left/Right Direction:

If `self.v.x` is negative, you can still use the `sign` of it to move in the right direction:

```
python

direction = 1 if self.v.x >= 0 else -1
self.x += direction * self.lockstep_step
```

Want help integrating this into vertical drops when aliens hit the edge, or are you mainly focusing on horizontal steps for now?



Sure

Awesome! Let's bring vertical lockstep movement into the mix, just like in the classic *Space Invaders* drop-and-reverse pattern. Here's how we'll do it step-by-step:

🧠 The Goal:

When the alien fleet hits the edge:

1. Drop **all aliens down** by a set vertical lockstep step.
2. **Reverse** the horizontal direction.
3. Continue horizontal lockstep movement.

✓ Step 1: Define vertical step

In your Alien's `__init__()`, just add:

```
python

self.lockstep_step_y = self.alien_tile.height / 2 # You can tweak this for more or less
dramatic drops
```

This gives each alien a standard vertical step when triggered.

✓ Step 2: Add a fleet-level command

This logic typically lives in a **Fleet Manager** or the main game loop. If you're running a **Fleet** or similar object (or group update logic), you'd do something like this when an edge is hit:

```
python

for alien in aliens.sprites():
    alien.y += alien.lockstep_step_y
    alien.v.x *= -1 # Reverse direction
    alien.step_interval = alien.lockstep_step / abs(alien.v.x)
```

You'd also want to reverse direction only **once per edge hit**, so make sure this logic isn't triggered multiple times in one update cycle.

✓ Step 3: Modify `check_edges()`

You already have:

```
python

return self.x + (self.rect.width / 2) >= sr.right or self.x <= (self.rect.width / 2)
```

Which is perfect. But now you'll use this in your fleet logic to know **when** to drop vertically and reverse direction.

✓ Step 4: Optional sync animation on vertical drop

You can reuse the same `timer.advance_frame()` logic when dropping, to keep animations in sync if you like:

```
python

alien.timer.advance_frame()
```

Example Fleet Logic (pseudo-code):

python

```
edge_hit = False
for alien in aliens:
    if alien.check_edges():
        edge_hit = True
        break

if edge_hit:
    for alien in aliens:
        alien.y += alien.lockstep_step_y
        alien.v.x *= -1
        alien.step_interval = alien.lockstep_step / abs(alien.v.x)
```

This ensures the fleet reacts in lockstep the moment *any* alien hits the edge.

Want to go full retro?

If you want *all* aliens to move and animate at the same time (tick-tick-tick like the arcade game), you could unify all alien timers and movement steps under a **global fleet timer**, too. I can help you build that if you're interested.

Let me know if you're using a fleet manager class or want help coordinating the whole group like the classic game!



Hmm, I currently do have a fleet class. My issue is that depending on the lockstep settings, sometimes the edge thing is triggered multiple times.

```
import pygame as pg
from vector import Vector
from point import Point
from laser import Laser

from alien import Alien
from pygame.sprite import Sprite

from random import randint

class Fleet(Sprite):
    def __init__(self, si_game):
        self.si_game = si_game
        self.screen = si_game.screen
        self.ship = si_game.ship
        self.aliens = pg.sprite.Group()
        self.settings = si_game.settings
        self.stats = si_game.stats
        self.sb = si_game.sb
        self.v = Vector(self.settings.alien_speed, 0)
        # alien = Alien(si_game=si_game)
        # self.aliens.add(alien)
        self.spacing = 1.5
```

```

self.create_fleet()
# self.create_row()

self.lasers = pg.sprite.Group()

def reset_fleet(self):
    self.aliens.empty()
    self.create_fleet()

def create_fleet(self):
    """Create a fleet of aliens with a fixed number of columns."""
    # Set the desired number of columns and rows
    num_columns = self.settings.alien_columns
    num_rows = self.settings.alien_rows

    # Get the size of the largest alien sprite for spacing
    alien_tile = Alien.alien_images2[0].get_rect()
    tile_width = alien_tile.width
    tile_height = alien_tile.height

    # Calculate horizontal spacing
    column_spacing = tile_width * self.spacing
    row_spacing = tile_height * self.spacing

    # Alien Type
    alien_type = 0
    frame_index = 0
    rand_type = False

    # Create the fleet using specified columns and calculated rows
    for row in range(int(num_rows)):
        y = (row + 1) * row_spacing

        # Specify Aliens Used For Each Row
        if row == 0:
            alien_type = 0
        elif row == 1 or row == 2:
            alien_type = 1
        elif row == 3 or row == 4:
            alien_type = 2
        else:
            rand_type = True

        for col in range(num_columns):
            if rand_type: alien_type = randint(0,2) # Random each alien

            x = (col + 1) * column_spacing
            new_alien = Alien(si_game = self.si_game, v=self.v, type = alien_type, animation_start =
frame_index, lockstep = True)
            new_alien.rect.centery = y
            new_alien.y = y

```

```

        new_alien.x = x
        new_alien.rect.centerx = x
        self.aliens.add(new_alien)
        frame_index = (frame_index + 1) % 2

def check_edges(self):
    for alien in self.aliens:
        if alien.check_edges() and alien.is_active:
            return True
    return False

def check_bottom(self):
    for alien in self.aliens:
        if alien.rect.bottom >= self.settings.scr_height:
            return True
    return False

def update(self):
    collisions = pg.sprite.groupcollide(self.ship.lasers, self.aliens, False, False)

    # Check Ship Laser Collisions Against Aliens
    if collisions:
        for lasers, aliens in collisions.items():
            self.stats.score += self.settings.alien_points * len(aliens)
            for alien in aliens:
                if alien.is_active:
                    if self.settings.debug_fleet: print(f"Laser {id(lasers)} hit alien {id(alien)} at ({alien.rect.x},
{alien.rect.y})")
                    lasers.kill()
                    alien.hit()

    self.sb.prep_score()
    self.sb.check_high_score()

    # If there are no Aliens remaining, refresh and empower.
    if not self.aliens:
        self.ship.lasers.empty()

    if self.settings.increase_dynamic_speed:
        self.settings.increase_speed()
        self.update_dynamic_settings()
        self.ship.update_dynamic_settings()

    self.create_fleet()
    # Increase level.
    self.stats.level += 1
    self.sb.prep_level()
    return

    # Check if aliens hit the ship. - Exploding Aliens don't affect it

```

```

if pg.sprite.spritecollideany(self.ship, self.aliens) and not self.ship.destroyed:
    if self.settings.debug_fleet: print("Ship hit!")
    if (pg.sprite.spritecollideany(self.ship, self.aliens)).is_active == True:
        self.ship.ship_hit()

# Check if alien lasers hit the ship
for self.alien in self.aliens:
    if pg.sprite.spritecollideany(self.ship, self.alien.lasers) and not self.ship.destroyed:
        if self.settings.debug_fleet: print("Ship hit!")
        self.ship.ship_hit()

# Check if aliens reached the bottom
if self.check_bottom():
    return

# Check if an alien has hit the edge
if self.check_edges():
    self.v.x *= -1
    for alien in self.aliens:
        alien.v.x = self.v.x
        alien.y += self.settings.fleet_drop_distance

for alien in self.aliens:
    alien.update()

def draw(self): pass
# for alien in self.aliens:
#     alien.draw()

def update_dynamic_settings(self):
    self.v = Vector(self.settings.alien_speed, 0)

def main():
    print("\n run from alien_invasions.py\n")

if __name__ == "__main__":
    main()

```

Thanks for sharing the `Fleet` class — this is really solid! You're super close to getting clean lockstep movement. The issue you're describing — **multiple edge detections per update** — is totally common in fleet-based lockstep systems. Here's how we can fix it 🙋

The Problem:

When many aliens reach the screen edge in the same frame (common in lockstep), your check triggers the **edge logic multiple times**, even though one drop+reverse per cycle is all you need.

The Fix: Use a cooldown flag or edge-lock

We'll add a short "lock" on fleet direction change — like a cooldown that says:

"Okay, we *just* reversed, let's wait until all aliens are back inside bounds before we check again."

Option 1: Use a `self.edge_lock` flag

In your `Fleet.__init__`:

```
python

self.edge_lock = False
```

In your `update()` logic, wrap the edge check like this:

```
python

if not self.edge_lock and self.check_edges():
    self.edge_lock = True
    self.v.x *= -1
    for alien in self.aliens:
        alien.v.x = self.v.x
        alien.y += self.settings.fleet_drop_distance
```

Then **release the lock** once no aliens are touching the edge anymore:

```
python

if self.edge_lock and not self.check_edges():
    self.edge_lock = False
```

This ensures that the fleet only reverses once per edge-touch event, even if several aliens touch the edge at the same time.

Bonus Tip: Consistent Step Timing

You can also tweak your aliens' `step_interval` after direction reversal (especially if speed increases):

```
python

alien.step_interval = alien.lockstep_step / abs(alien.v.x)
```

Add that right after the reversal for tighter movement syncing.

Let me know if you want help adding synchronized animation steps for each tick, or a `FleetTimer` that controls all alien animation/movement together, arcade-style. You've already built the hard part 🙌



Do I put that second if statement right after the first one in the `update()`?

Yes — great question! You should **put that second if right after** the first one in the `update()` method, ideally in the same general section where you're handling edge detection and vertical drops.

Here's how that section of your `update()` might look with both `if` blocks added, including the lock:

✓ Fleet update() Edge Handling (with lock):

```
python

# Check if an alien has hit the edge
if not self.edge_lock and self.check_edges():
    self.edge_lock = True
    self.v.x *= -1
    for alien in self.aliens:
        alien.v.x = self.v.x
        alien.y += self.settings.fleet_drop_distance

# Release the edge lock once all aliens are inside bounds again
if self.edge_lock and not self.check_edges():
    self.edge_lock = False
```

🔍 Why it works:

- First block flips direction and drops the fleet **once**.
- Second block **waits** until all aliens are clear of the edge before allowing another reversal.

This prevents multiple vertical drops in rapid succession, which is exactly what you want in lockstep-style fleet movement.

Let me know if you want to make this into a `FleetState` system later (like `idle`, `dropping`, `moving`, etc.) for even more control.



Alright, thank you!

You're very welcome! 🙌 Glad to help — your implementation is looking awesome so far. If you ever want to add things like synced animation steps, special UFO behavior, or funky bonus rounds like the arcade version, just say the word. Have fun invading Earth! 🚀

Let me know how it goes once that lock is in!