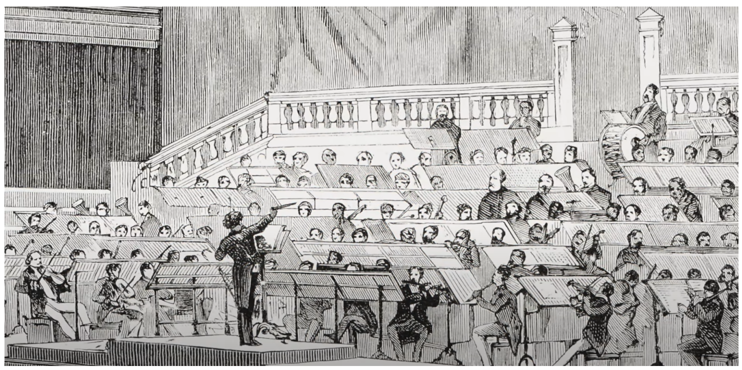
Gravity Demo

Martin N. Håvardsen

October 11, 2025



1 Generell protokoll

Prosesson for simuleringen per objekt skal gå som følger:

- tegn
- kalkuler akselerasjon
- kalkuler fart
- kalkuler posisjon
- oppdater alle egenskaper — poisjon, fart og akselerasjon

2 Objekt

2.1 Egenskaper

alle objekt skal ha følgende egenskaper:

- posisjon
- fart
- akselerasjon
- masse

2.2 Teori for å kalkulere akselerasjon

For et objekt A med masse m_A og distansen r til et objekt B, så vil kraften G_B som B påfører A vil være gitt ved følgende:

$$G_B = \gamma \frac{m_A m_B}{r_B^2} \tag{2.1}$$

Gravitasjonskonstanten, γ, vil være en svært liten konstant.

Fra Newtons andre lov får vi at akselerasjonen til objekt A, $\vec{a} = \frac{\overrightarrow{\Sigma F}}{m_A}$. Siden $\overrightarrow{\Sigma F} = \vec{G}_B + \vec{G}_C + \ldots$, for gravitasjonskrefter fra andre objekt B, C osv. $\overrightarrow{\Sigma F}$ kommer bare til å bestå av gravitasjonskrefter i denne simuleringen.

$$\vec{a} = \frac{\hat{G}_B \cdot \gamma \frac{m_A m_B}{r_B^2} + \hat{G}_C \cdot \gamma \frac{m_A m_C}{r_C^2} + \ldots}{m_A} \tag{2.2}$$

$$\vec{a} = \gamma \left(\frac{\hat{G}_B \cdot m_B}{r_B^2} + \frac{\hat{G}_C \cdot m_C}{r_C^2} \right) \tag{2.3}$$

$\hat{G}_B$ vil alltid være rettet mot B, fra A.

$$\hat{G}_B = \frac{\vec{s}_B - \vec{s}_A}{r} \tag{2.4}$$

2.3 Metode for å kalkulere akselerasjon

1. Finn alle retningsvektorer for gravitasjonskreftene.
2. Regn ut $\vec{a}$

Her er hvordan dette ble implementert:

```python
    def update_acceleration(self, all_objects):
        """
        Utfører metode for å kalkulere
        akselerasjon fra seksjon 2.3
        """
        gm = pygame.Vector2(0.0,0.0)
        for obj in all_objects:
            vec=obj.s-self.s
            length=vec.length()
            if length!=0:
                g_hat = vec/length
                gm=gm + g_hat*obj.m/(length**2)
        self.a = gm*GAMMA
```

`obj` her er et `PhysObj` objekt som inneholder sin masse, posisjon, fart og akselerasjon. Denne funksjonen utføres en gang per *frame* for hvert objekt.

3 Perspektiv og koordinatsystem

3.1 Mål

Simuleringen skal ha kamerafunksjonalitet. Man skal kunne bevege koordinatsystemet/rutenettet med musen — bevege *kameraet*. I tillegg skal det være mulig å forstørre rutenettet — *zoome*.

3.2 Teori

Planet som objektene tilhører, er kartesisk og har to dimensjonene. Disse dimensjonene kan fremstilles med to retningsvektorer, $\hat{\imath}, \hat{\jmath}$, som — skalert, kan kombineres til å få hva som helst punkt på planet.
Koordinaten i vinduet som objektet blir tegnet i er ikke nødvendigvis lik koordinaten på planet. Hvordan transformeringen av romkoordinaten til vinduskoordinat skjer kan variere. Relevante faktorer her er *kameraet* sin posisjon, eventuell *zoom*, og *rotasjon*. Her ignorerer vi rotasjon.
Vi har da at transformeringen $T : \mathbb{R}^2 \to \mathbb{R}^2$,

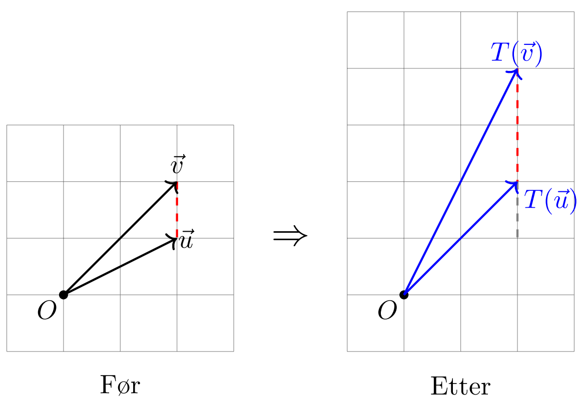


Figure 1: Transformeringen av $\vec{v}$ og $\vec{u}$ under T

$$T(\vec{v}) = k\vec{v} + \vec{c} \tag{3.1}$$

Musehjulet skal da forandre k fra (3.1), dette vil forstørre/forminske rutenettet.
Merk at $k \in \langle 0, \infty \rangle$
$\vec{c}$ fra (3.1) er koordinaten på planet som tilsvarer `(0,0)` i vinduet. Vi finner også den inverse transformeringen ved:

$$T^{-1}(\vec{v}) = \frac{\vec{v} - \vec{c}}{k} = \frac{1}{k}\vec{v} - \frac{1}{k}\vec{c} \tag{3.2}$$

$T^{-1}(\vec{v})$ vil da være koordinaten i vinduet som tilsvarer koordinaten på planet.

4 Kollisjon

4.1 Objekt vs barriere

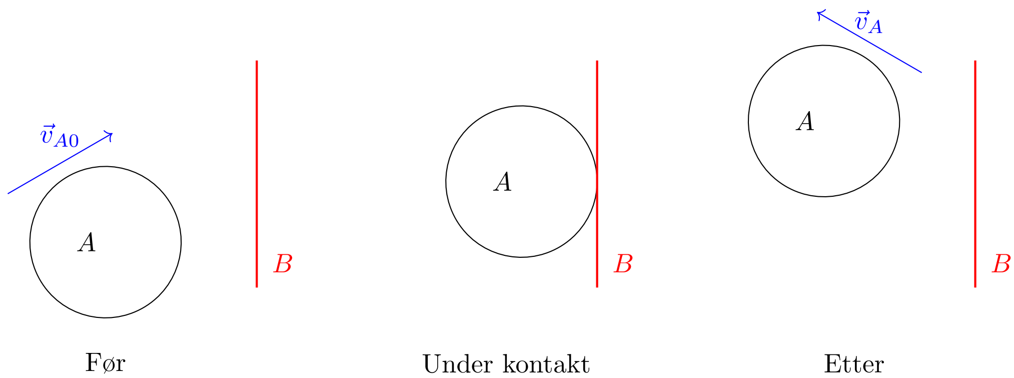


Figure 2: Kollisjon mellom objekt A og barriere B

$$\vec{\rho}_{A0} + \vec{\rho}_{B0} = \vec{\rho}_A + \vec{\rho}_B \tag{4.1}$$

$$\|\vec{\rho}_{B0}\| = \rho_{B0} = 0 \implies \vec{\rho}_{A0} = \vec{\rho}_A + \vec{\rho}_B \tag{4.2}$$

La kinetisk energi, E_k, bevares, $E_{k0} = E_k$

$$E_{kA0} + E_{kB0} = E_{kA} + E_{kB} \tag{4.3}$$

$$\frac{1}{2}m_A v_{A0} + \frac{1}{2}m_B v_{B0} = \frac{1}{2}m_A v_A + \frac{1}{2}m_B v_B \tag{4.4}$$

$$m_A v_{A0} - m_A v_A = m_B v_B - m_B v_{B0} \tag{4.5}$$

$$m_A(v_{A0} - v_A) = m_B(v_B - v_{B0}) \tag{4.6}$$

$$-\Delta\rho_A = \Delta\rho_B \tag{4.7}$$

Vi vet at $v_B = 0$. Derfor må $\Delta\rho_A = 0$. Siden barrieren kun kan utøve en kraft som ligger normalt til barriereplanet, så vil $\vec{v}_A = \vec{v}_{A0} + \vec{N}_{normal}$. Dette vil si at siden bevegelsesmengden er bevart for A, så vil innfalsvinkelen og utfallsvinkelen være like.

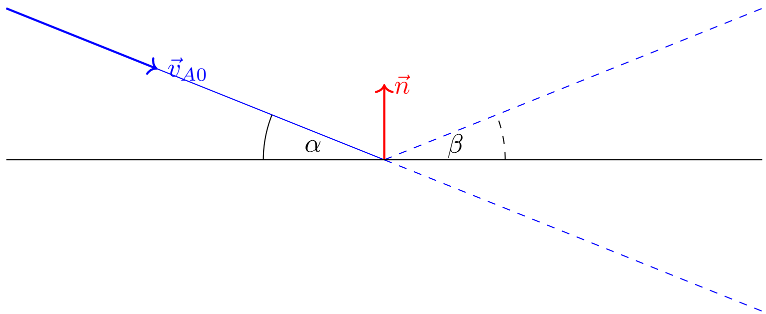


Figure 3: Speiling

4.2 Objekt vs Objekt

5 Program struktur

5.1 Globale variabler

- `SCREEN_WIDTH = 1280`
- `SCREEN_HEIGHT = 720`
- `FPS_TARGET = 180`
- `screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))`
- `pygame.display.set_caption("Gravity Demo")`
- `font = pygame.font.Font(None, 36)`
- `clock = pygame.time.Clock()`
- `meters_per_screenwidth = 150E9*2`
- `k_meters_per_pixel = meters_per_screenwidth/SCREEN_WIDTH`
- `k_increment = k_meters_per_pixel*0.05`
- `c_coordinate_at_origin = pygame.Vector2(0,0)`
- `mouse_drag_start = pygame.Vector2(-1,-1)`
- `global_mb2_down = False`
- `object_size_pixels = 100`
- `RUNNING = True`
- `GAMMA = 7E-11`
- `R_RELATIVE_TO_M=False`

- `objA = PhysObj(...)`
- `objB = PhysObj(...)`
- `objC = PhysObj(...)`
- `objD = PhysObj(...)`
- `ALL_OBJECTS = [objA, objB, objC, objD]`

5.2 Funksjoner

Funksjon	**input**	**output**
`update_objects`		
`add_object_with_vel`	`pos` : *vec2* `color` : *vec3* `mass` : *float* `vel` : *vec2*	
`add_object`	`pos` : *vec2* `color` : *vec3* `mass` : *float*	
`transform_window_to_plane`	`v` : *vec2*	transformed v
`transform_plane_to_window`	`v` : *vec2*	transformed v
`transform_plane_to_window_tuplet`	`v` : *vec2*	transformed v
`draw_circle_in_plane`	`window` : *pygame.surface* `color` : *vec3* `pos` : *vec2* `rad` : *uint* `outline_thickness` : *uint*	
`handle_mb2`		

Table 1: Matrise med funksjoner

- **Global Scope**
  - Variables: `RUNNING, MB1, MB2, MB3, MX, MY, global_mb2_down, object_size_pixels, k_meters_per_pixel, k_increment, clock, FPS_TARGET, font, screen`
  - Functions: `add_object, add_object_with_vel, transform_window_to_plane, transform_plane_to_window, handle_mb2, update_objects`
- **Main While Loop:** `while RUNNING:`
  - **Event Loop:** `for event in pygame.event.get():`
    - `if event.type == pygame.QUIT:` set `RUNNING = False`
    - `if event.type == pygame.KEYDOWN:`

        - `if key == K_s`: update `object_size_pixels` from input
    - `if event.type == pygame.MOUSEBUTTONDOWN`:
        - Read mouse buttons: `MB1, MB2, MB3` and position `MX, MY`
        - `if MB1`: input mass, compute color, call `add_object()`
        - `if MB3`: input mass/velocity, compute color, call `add_object_with_vel()`
        - `if MB2`: set `global_mb2_down = True`
    - `if event.type == pygame.MOUSEWHEEL`: adjust `k_meters_per_pixel` based on `event.y`
    - `if event.type == pygame.MOUSEBUTTONUP`:
        - `if MB2`: reset `global_mb2_down` and `mouse_drag_start`
- **Post-Event Loop Updates:**
    - `screen.fill(pygame.Color(0,0,0))`
    - `if global_mb2_down: handle_mb2()`
    - `update_objects()`
    - Render text and blit: show `k_meters_per_pixel`
    - `pygame.display.flip()`
    - `clock.tick(FPS_TARGET)`