Simple 3D engine med pygame

Christian Linde Ahrnkiel
Synopsis Prog B eksamensprojekt
Odense Tekniske Gymnasium

9. Maj 2025

Indholdsfortegnelse

1 Indledning

1.1 Formål

Formålet med dette projekt er at lave en simple 3D engine. En 3D engine er et program der kan vise 3D objekter på en 2D skærm. Det er en vigtig del af mange computerspil og grafiske applikationer. Et eksepmle på et program som har en 3D engine er blender. Det kan ses på figur 1.

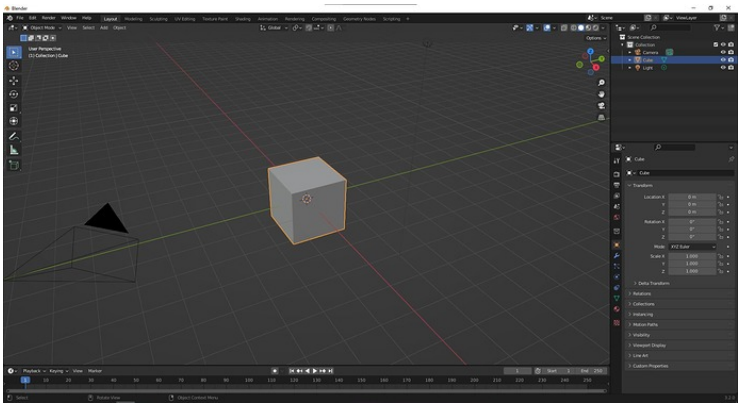

Figure 1: Billede af 3D modellerings programmer blender.

I dette projekt har jeg lavet en simpel 3D engine i python med pygame biblioteket. Man kan se det endlige program `sigma()` på figur 2.

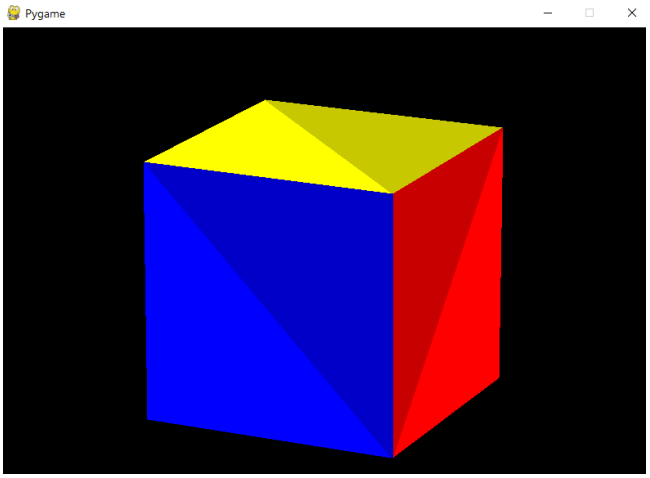

Figure 2: Billede af mit program.

2 3D projektion

3D projektion er en metode til at vise 3D objekter på en 2D skærm. Der findes mange forskellige metoder til at lave 3D projektion. Se figur 3 for en Illustration af de forskellige 3D projektion.

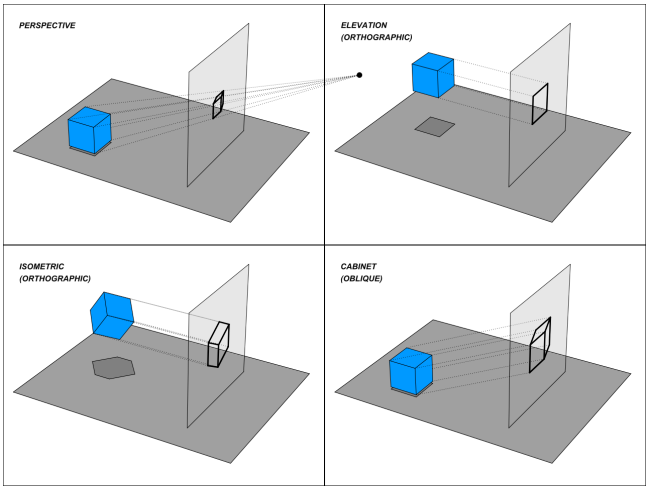

Figure 3: Illustration af de forskellige 3D projektion fra wikipedia.org.

Den som jeg har valgt at bruge er den som hedder perspective projection. Perspective projection fungere ved at man har et punkt i 3D rummet, som kan forstårs som et kamera. Der er en plan, som kan forstårs som en skærm. Og så er der et punkt i 3D rummet, som er en del at et objekt. Man kan så tegne en linje fra kameraet til punktet på objektet, og så finde ud af hvor den skærer skærmen. Der vil man så tegne punktet på skærmen.

På figur 4 kan man se perspective projection oppe fra som 2D. Der er et punkt, som hedder $Kamera$ som er kameraet. Der er en plan, som er den sorte linje, det er skærmen. Der er et punkt, som hedder p_c som er et punkt på objektet, og i dette tilfælle er en firekant. Man kan se at der er en linje fra kameraet til punktet p_c. Der hvor linjen skærer skærmen, der er punktet p_s. Det er sådan metoden fungere.

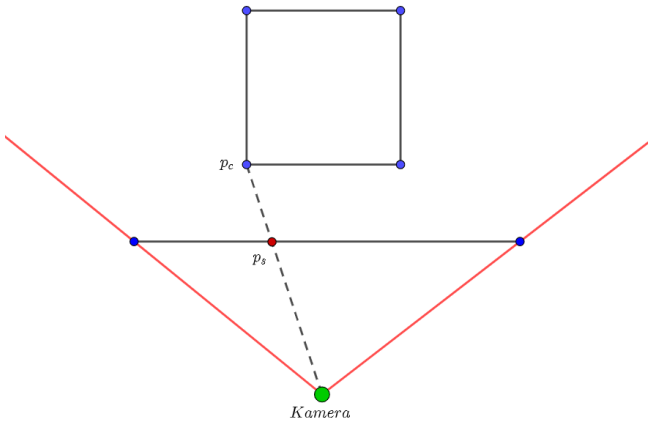

Figure 4: Illustration af 3D perspective projection set oppe fra som 2D.

3 Pygame

Pygame er et bibliotek til python, som gør det muligt at lave grafiske applikationer såsom spil. Pygame laver et popup vindue, og man tenge figurer og billeder i vinduet. Pygame har noget som hedder et game loop. Et game loop er et loop som opdaterer progammet hele tiden. Den start med at først håndterer input fx. tastatur eller mus. Så opdaterer spillets tilstand og logikken fx. bevægelse af objekter. Og til sidst tegner det objekterne på skærmen. Et meget simpelt game loop kan ses i koden på figur 5.

```python
import pygame
import sys

pygame.init()
screenWidth = 800
screenHeight = 600
black = (0, 0, 0)

screen = pygame.display.set_mode((screenWidth, screenHeight))
pygame.display.set_caption("Pygame")
clock = pygame.time.Clock()
FPS = 60
# game loop
running = True
while running:
    clock.tick(FPS)
    # inputs
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
    # updates

    # draw
    screen.fill(black)
    pygame.display.flip()
```

Figure 5: Kode for meget simpelt game loop.

4 3D punkt til 2D punkt

For at kunne være i stand til at tegne 3D objekter, på en 2D skærm, startet jeg med at lave en funktion som hedder `getPointOnPlan`. `getPointOnPlan` tager 3 argumenter. Det første argument `fig` er en list af 3D punkt. (Et 3D punkt er en list med 3 floats) Det andet argument `plan` er en list som beskriver ligningen for en plan på denne både måde: `[a, b, c, k]`, hvor plansligning ser sådan ud $ax + by + cz + k = 0$. Det sidste argument `pov` er et 3D punkt som beskriver kameraet. Funktionen `getPointOnPlan` returnere en liste punkter på planen som 2D punkter. (Et 2D punkt er en list med 2 floats) Koden kan ses i figur 6.

```
1  def getPointOnPlan(fig,plan,pov):
2      final = []
3      for i in range(len(fig)):
4          r = []
5          for j in range(len(fig[i])):
6              r.append(fig[i][j]-pov[j])
7          x = [pov[0],r[0]]
8          y = [pov[1],r[1]]
9          z = [pov[2],r[2]]
10         ligning_num = plan[0]*x[0] + plan[1]*y[0] + plan[2]*z[0] + plan[3]
11         ligning_t = plan[0]*x[1] + plan[1]*y[1] + plan[2]*z[1]
12         t = (-(ligning_num))/ligning_t
13         final.append([y[0]+t*y[1], z[0]+t*z[1]])
14     return final
```

Figure 6: Kode for `getPointOnPlan`.

For at ikke at gå for meget i dybden med matematikken, så vil jeg gå lidt overfladisk om det. På linje 3 laver jeg et for loop som går igennem alle punkter i listen `fig`, så et punkt er `fig[i]`. Derefter på linje 4, 5 og 6 laver jeg en retningsvektor `r` fra `pov` til `fig[i]` for at lave en linje mellem dem. Så på linje 7, 8 og 9 opsætter jeg en liste `x`, `y` og `z` som er `pov[0]` og `r[0]` for at løse en ligning mellem planen og og linjen. Bageefter på linje 10, 11 og 12 løser jeg en ligning for t. Det er hvor langt henne linjen skærer planen. Til sidst på linje 13 tilføjer jeg y og z koordinaterne til listen `final` som er en liste af 2D punkter.

Hvis man så giver `getPointOnPlan` en liste af 3D punkter som laver en kasse, og tegner cirekler ved de 2D punkter som den returnere. Så får man 8 punkter som er kanterne af en kasse. Det kan ses på figur 7.

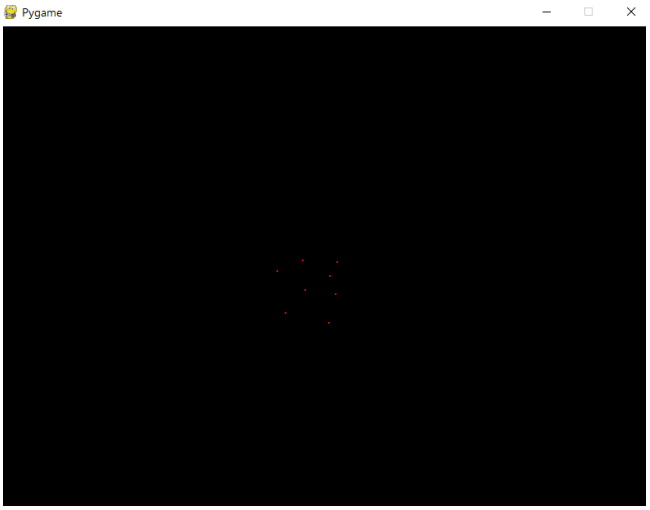

Figure 7: Illustration af 3D perspective projection set oppe fra som 2D.

5 Polygoner

6 Painters algorithm

7 Egen algorithmme

8 Kasser

9 Konklusion