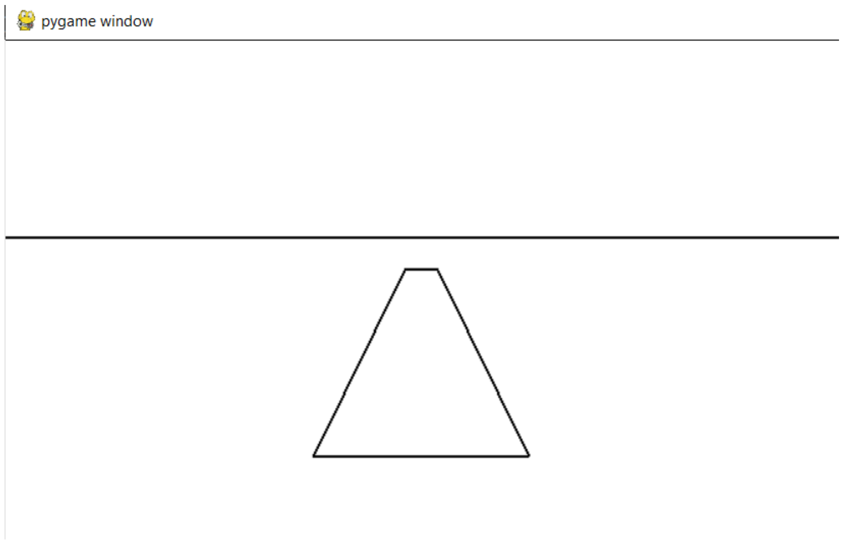
ENER
GY
CHAL
LEN
GE

This year, TU Delft celebrates its 180th anniversary. To mark the occasion, many different activities are planned to highlight our role in the energy transition, also in the educational programme. You've already seen the first assignment in this theme, this assignment is the second.

Assignment 5: Continuous Descent Approach simulator

In this assignment you will make a small (symmetrical) keyboard-controlled flight simulator game allowing you to fly continuous descent approaches. The goal is to try to fly as efficiently as possible while still landing safely. You will use the equations of motion from the *Introduction to Aerospace Engineering* course, which are given in this assignment. The result will look like this (inverted colours):



You will calculate the screen coordinates using the angles and distance calculated in your model. We will first build the view step by step, then add basic motion and then add the model to generate correct motion. For the Out of The Window View (OTWV) we'll make a simple pseudo-perspective view, based on two angles: Azimuth, which refers to the horizontal viewing angle (left negative) and Elevation, which refers to the vertical viewing angle (positive up). To draw the OTWV we'll first calculate these angles, and then convert them to screen pixel locations. To help you with the PyGame functions, please use information and examples from chapter 12 from the AE1205 reader as well as the PyGame documentation.

The formulae for the geometry of the horizon and the runway trapezium are given in this assignment, but you can also use your own derivations and method if that is easier for you. A stepwise approach ensures you will have working program which we can test at any moment. This makes it easier to debug it. Gradually it will become the final program.

Step 1: Horizon-only Out-of-the-window view

Create a file called `view.py`, and a file called `main.py`. Templates for both files are given below. In the `view.py` file you will define all functions that relate to PyGame. The `main.py` template below contains a basic testing set-up, with a very simple loop. Some of these statements are just for you to be able to test the code you make in `view.py`. You will adapt and replace these parts later in the assignment.

The `main.py` template:

main.py

```
import view       # file with your otwv drawing functions

# Size of window
xmax = 1000 #[pixels] window width (= x-coordinate runs of right side right+1)
ymax = 700  #[pixels] window height(= y-coordinate of lower side of window+1)

# Size of viewport (angles in degrees)
minelev = -14.0 #[deg] elevation angle lower side
maxelev = 14.0  #[deg] elevation angle top side
minazi = -20.0  #[deg] azimuth angle left side
maxazi = 20.0   #[deg] azimuth angle right side

# Set pitch angle
theta = float(input(”Enter pitch angle[deg]:”)) # pitch angle [deg]

# Set up window, scr is surface of screen
scr = view.openwindow(xmax, ymax)

running = True
while running:
    # Clear screen scr
    view.clr(scr)

    # Get user inputs by processing events
    dalpha, dthrottle, dflaps, gearpressed, brakepressed, userquit = view.processevents()

    # Draw horizon on scr, using pitch angle theta, and screen dimensions
    view.drawhor(scr, theta, xmax, ymax, minelev, maxelev)

    # Update screen
    view.flip()
    if userquit:
        running = False

# Close window
view.closewindow()
```

Below you can see a template for the `view.py` file, describing the shape and goal of the functions you will make in this file. In the following steps you will fill these functions with code, using chapter 12 of the reader and the pygame documentation for details on the functions (`https://www.pygame.org/docs/`).

The `view.py` template:

view.py

```python
import pygame as pg
# Colours
white = (255, 255, 255)
black = (0, 0, 0)
red = (255, 0, 0)

def openwindow(xmax, ymax):
    """ Init pygame, set up window, return scr (window Surface) """
    scr = ...
    return scr

def processevents():
    """ Let PyGame process events, and detect keypresses. """
    dalpha = 0              # -1 or 0 or 1
    dthrottle = 0           # -1 or 0 or 1
    dflaps = 0              # -1 or 0 or 1
    gearpressed = False     # True or False
    brakepressed = False    # True or False
    userquit = False        # True or False

    return dalpha, dthrottle, dflaps, gearpressed, brakepressed,
     ↪  userquit

def clr(scr):
    """Clears surface, fill with black"""
    ...
    return

def flip():
    """Flip (update) display"""
    ...
    return

def closewindow():
    """Close window, quit pygame"""
    ...
    return

def elev2y(elev, ymax, minelev, maxelev):
    """Scale an elevation angle to y-pixels"""
    y = ...
    return y

def azi2x(azi, xmax, minazi, maxazi):
    """Scale an azimuth angle to x-pixels"""
    x = ...
    return x

def drawhor(scr, theta, xmax, ymax, minelev, maxelev):
    """Draw horizon for pitch angle theta[deg]"""
    ...
    return
```

Fill in the `view.py` functions in the following steps:

1. **`openwindow()`**, **`processevents()`**, **`clr()`**, **`flip()`**, and **`closewindow()`**: These five functions should implement the basic PyGame functions to open a window, process events and detect keypresses, fill the screen with a single colour, flip the display, and close and quit PyGame. Have a look at the examples in chapter 12 of the reader (or your own work in assignment 4!) to find out what PyGame functions are needed. For the user inputs consider that:

   - Alpha, throttle, and flaps can both increase and decrease, so you need two keys for each.
   - Gear only extends, so you need one key
   - Brakes extend instantaneously, for which you need one key
   - A user quit event can be triggered by the escape key, and the user clicking the window close button.

   The `processevents()` example in the `view.py` template above gives an example of what values you could return for each control. A suggestion for which keys you could use for the aircraft controls is given below in Step 4, but you can also choose your own!

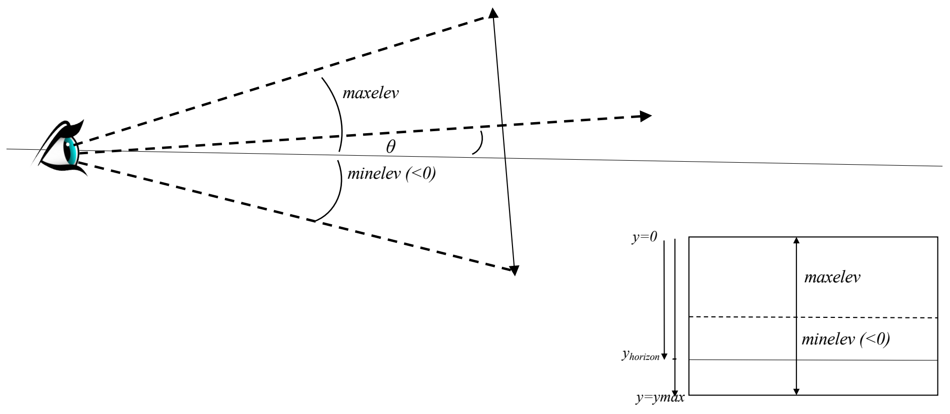
2. **`elev2y()`** and **`azi2x()`**: For these scaling functions you need to convert degrees of viewing angle to pixels in your pygame window (with y running from 0 top to ymax bottom). For this you can use the size of the viewport:

   $$x = x_{max}\frac{azi - azi_{min}}{azi_{max} - azi_{min}} \qquad\qquad y = y_{max} - y_{max}\frac{elev - elev_{min}}{elev_{max} - elev_{min}}$$

   Check your functions by running view.py and calling elev2y and azi2x in the shell, for example:

   ```
   >>> from view import azi2x, elev2y
   >>> azi2x(-10, 500, -20, 20)
   125.0
   >>> elev2y(-10, 500, -15, 35)
   450.0
   ```

3. **`drawhor()`**: To see where you need to draw our horizon, see the figure below:

   The elevation is zero when you look straight ahead, along our pitch angle theta. When you pitch up (positive theta), the horizon should move down (negative elevation), and vice versa. With this in mind, use the conversion function `elev2y()` to calculate the y-coordinate of the horizon line. With this y-coordinate draw a line from $x = 0$ to $x = x_{max}$.

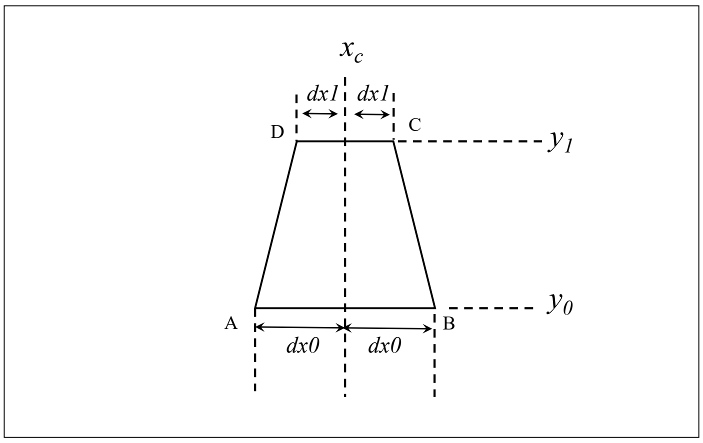
Test the conversion and drawing functions by using the test program at the beginning of the assignment and enter different values for the pitch angle to see whether the horizon makes sense. For negative pitch angles, so when looking down, or entering a negative pitch angle the horizon should go up!

Step 2: Add the runway

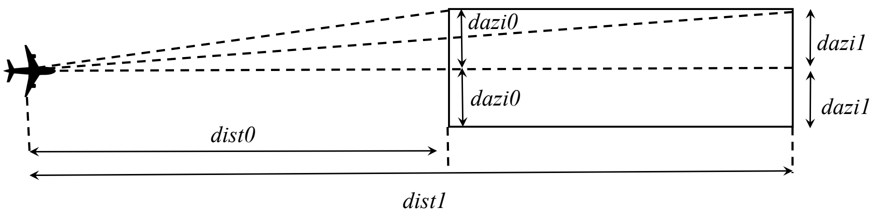
When this works you will add the runway, which has the following dimensions: a width of 60 meters, and a length of 3000 meters. You will add a new function to view.py:

```
def drawrunway(scr, theta, x, y, xmax, ymax,
               minazi, maxazi, minelev, maxelev):
```

For the calculation, you will use the fact that you will only simulate in two dimensions: you will always fly on the centerline, so no bank angle or course changes. This means your runway will always look like a trapezium ABCD with a top (far) side that is smaller than the bottom (close) side. The goal is to calculate its dimensions Δx_0 and Δx_1 and the coordinates y_0, y_1 and x_c. See the figure below for the meaning of these variables.



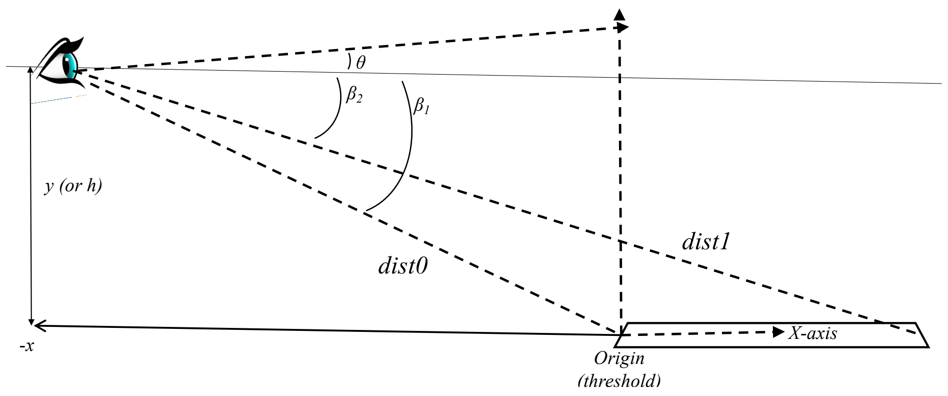
So for the horizontal coordinates, you only need to calculate the width in pixels as it is symmetric around azimuth angle zero, so around the center of the window. First you need to calculate the half width of the runway at the closer side (threshold) and similarly at the far side. Use the figure below and your conversion functions to calculate the half width in azimuth degrees.



$$dazi_0 = \arctan\left(\frac{\frac{1}{2}runwaywidth}{dist_0}\right) \qquad dazi_1 = \arctan\left(\frac{\frac{1}{2}runwaywidth}{dist_1}\right)$$

Convert these angles to the unit degrees, as the viewing port $minazi$, $maxazi$ are also in degrees.

For the distances $dist_0$ and $dist_1$, you need to use the Pythagorean distances, so including the altitude, see the figure below:



For the elevation angular coordinate do a similar exercise. Remember that you have to do this relative to minus θ:

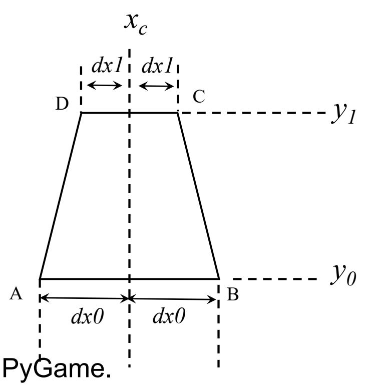
$$elev_0 = -\theta - \arctan(\frac{y}{-x}) \qquad elev_1 = -\theta - \arctan(\frac{y}{-x+runwaylength})$$

With the angular azimuth and elevation coordinates known, you need to convert everything to pixels. As the half width of the runway is not a position you should scale it with:

$$\Delta x_0 = x_{max}\frac{dazi_0}{maxazi-minazi} \qquad \Delta x_1 = x_{max}\frac{dazi_1}{maxazi-minazi}$$

The center x-coordinate x-c and y0 and y1 can be calculated with the conversion functions:

```python
import view
xc = view.azi2x(0, xmax, minazi, maxazi)
y0 = view.elev2y(elev0, ymax, minelev, maxelev)
y1 = view.elev2y(elev1, ymax, minelev, maxelev)
```



With these you can finally define the four corners of the trapezium ABCD:

$A(xc - dx0, y0) \quad C(xc + dx1, y1)$
$B(xc + dx0, y0) \quad D(xc - dx1, y1)$

These can then be used to draw the line pieces: AB, BC, CD, DA with PyGame.

Step 3: Adding motion

To make your model moving you first only implement the motion. Check chapter 12 and make a running game loop with your horizon. Take the following values as initial condition:

$$\begin{aligned} \gamma &= -3^\circ \\ \alpha &= 3^\circ \\ V &= 220\text{kts} \\ x &= -3000\text{m} \\ y &= 200\text{m} \end{aligned}$$

And use before the simple integration, this to calculate the speeds:

$$\begin{aligned} \theta &= \alpha + \gamma \\ v_x &= V\cos\gamma \\ v_y &= V\sin\gamma \end{aligned}$$

This should result in a descending flight towards the runway. Play around with some other values for speed and the angles to check your program. Note, though, that the equation for $elev_0$ doesn't allow for positions beyond the runway threshold, so let your simulation stop when x reaches zero.

Step 4: Add an aircraft model

Now you can finally add our aircraft model. First you will make a drag polar function flaps gear and brake extensions all run from 0.0 to 1.0:

```python
def CLCD(alpha, dflaps, dgear, dbrake):
    ...
    return CL, CD
```

Use the following model equations for the CL,CD function:

$$
\begin{aligned}
C_{L_{max}} &= 1.5 + 1.4\delta_{flaps} \\
C_L &= \min(0.1(\alpha + 3) + \delta_{flaps},\ C_{L_{max}}) \\
C_{D_0} &= 0.021 + 0.020\delta_{gear} + 0.120\delta_{flaps} + 0.4\delta_{brakes} \\
C_D &= C_{D_0} + 0.0365C_L^2
\end{aligned}
$$

You will also need the following model parameters:

```python
# Model parameters
mzerofuel = 40000.0 #[kg] mass aircraft + payload excl fuel
mfuel = 5000.0      # [kg]
S = 102.0           #[m2]
Tmax = 200 * 1e3    #[N]
CT = 17.8 * 1e6     # [kg/Ns] kg/s per N thrustSpecific fuel consumption
rho = 1.225         # [kg/m3] air density (constant or use ISA)
g = 9.81            # [m/s2] gravitational constant

throttledot = 0.1  # [1/s] throttle speed (spool up/down speed)
alphadot = 1.0  # [deg/s] alpha change due to control
flapsdot = 0.2  # [1/s] flap deflection speed
alphamin = -10.0  # [deg]
alphamax = 20.0  # [deg]
```

Controls

You control your aircraft with alpha, throttle, flaps, gear, and airbrakes. Use `pg.key.get_pressed()` and different keypresses to implement these controls (Pygame docs provide the correct index to this Boolean array to check whether a key is pressed or not: `https://www.pygame.org/docs/ref/key.html#key-constants-label`. See chapter 12 on how to do this.):

- **alpha**: Use the up- and down-key to increment/decrement alpha with a rate of `alphadot` degrees per second. Keep alpha between α_{min} and α_{max}.
- **throttle**: Use two other keys (e.g., w and s) to increment/decrement the throttle with a rate of `throttledot`. Keep the throttle between 0 and 1.
- **Flaps (optional)**: Use two keys (e.g., q and a) to increment/decrement the flaps with a rate of `flapsdot`. Keep the flap setting between 0 and 1.
- **Gear (optional)**: Assume that gear extends instantaneously. Your code should extend the gear once the letter ‘g’ has been pressed and released again.
- **Brake (optional)**: Assume that brakes extend and retract instantaneously. Brakes are extended when the ‘b’ key is pressed.

To calculate the acceleration, first you need to calculate the forces:

$$\begin{aligned}
C_L, C_D &= f_{CLCD}(\alpha, \delta_{flaps}, \delta_{gear}, \delta_{brake}) \\
L &= C_L \frac{1}{2} \rho V^2 S \\
D &= C_D \frac{1}{2} \rho V^2 S \\
W &= m \cdot g \qquad \text{with } m = m_{zerofuel} + m_{fuel} \\
T &= \delta_{throttle} T_{max} \\
\dot{m}_{fuel} &= -C_T \cdot T
\end{aligned}$$

Using the equations of motion from the course Introduction to Aerospace Engineering I. you can then calculate the accelerations in the direction of the speed and orthogonal to it, so the rate of change for the speed and γ:

$$\begin{aligned}
\frac{dV}{dt} &= \frac{T \cos\alpha - D - W \sin\gamma}{m} \\
\frac{d\gamma}{dt} &= \frac{L - W \cos\gamma + T \sin\alpha}{mV}
\end{aligned}$$

Integrating this will give the new V and the new flight path angle γ:

$$\begin{aligned}
V(t + \Delta t) &= V(t) + \frac{dV}{dt} \Delta t \\
\gamma(t + \Delta t) &= \gamma(t) + \frac{d\gamma}{dt} \Delta t
\end{aligned}$$

With the γ and airspeed, calculate your v_x and v_y.

This now allows you to land an aircraft. With keyboard control of throttle and alpha. This is sufficient for the assignment. You can add more controls for flaps, gear and speed brakes, as described above. Also adding indications using pygame fonts allows you make a sort of HUD view.

Optional extra's

- Record time histories of the aircraft states in lists or arrays, and plot them in graphs after completing a descent, to analyse your landing performance.
- Reduce ymax for OTWV drawing and add a cockpit panel in the lower part of window by loading an image and blitting the resulting surface there and then draw indicators and dials on it. Or blitting with digital nice images for indicators, flap setting, throttle, gear, fuel etc.
- Or on the OTWV HUD display: add cross at origin and fpv (flight path vector) alpha degrees lower (= FPV and AoA indicator)