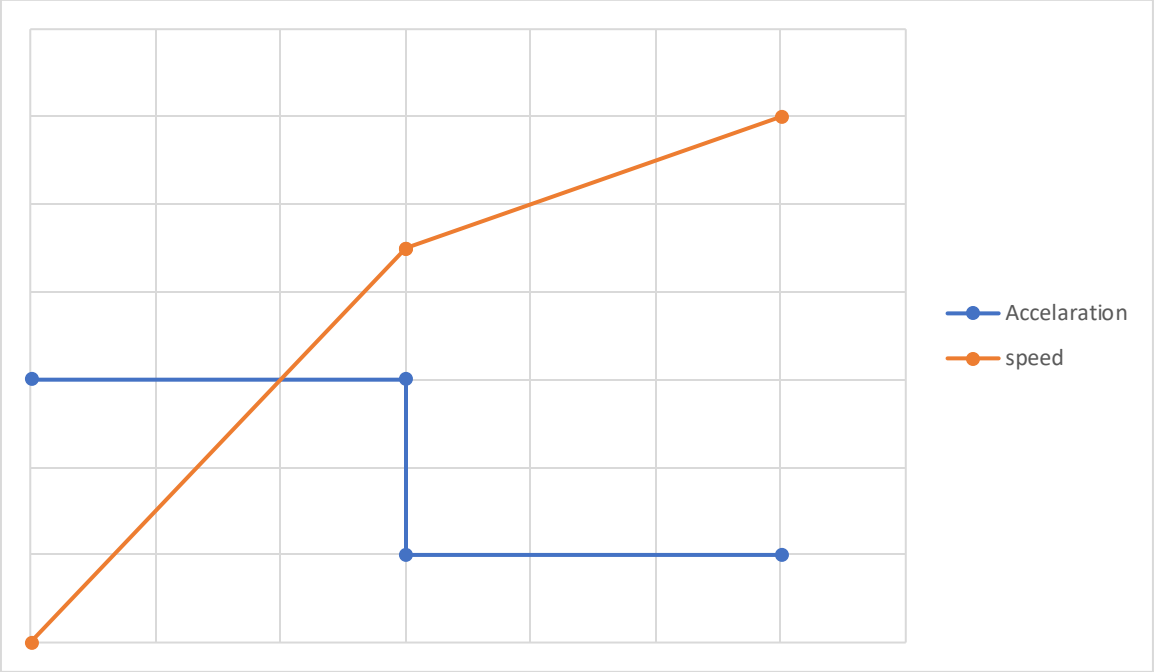
Decisions and process

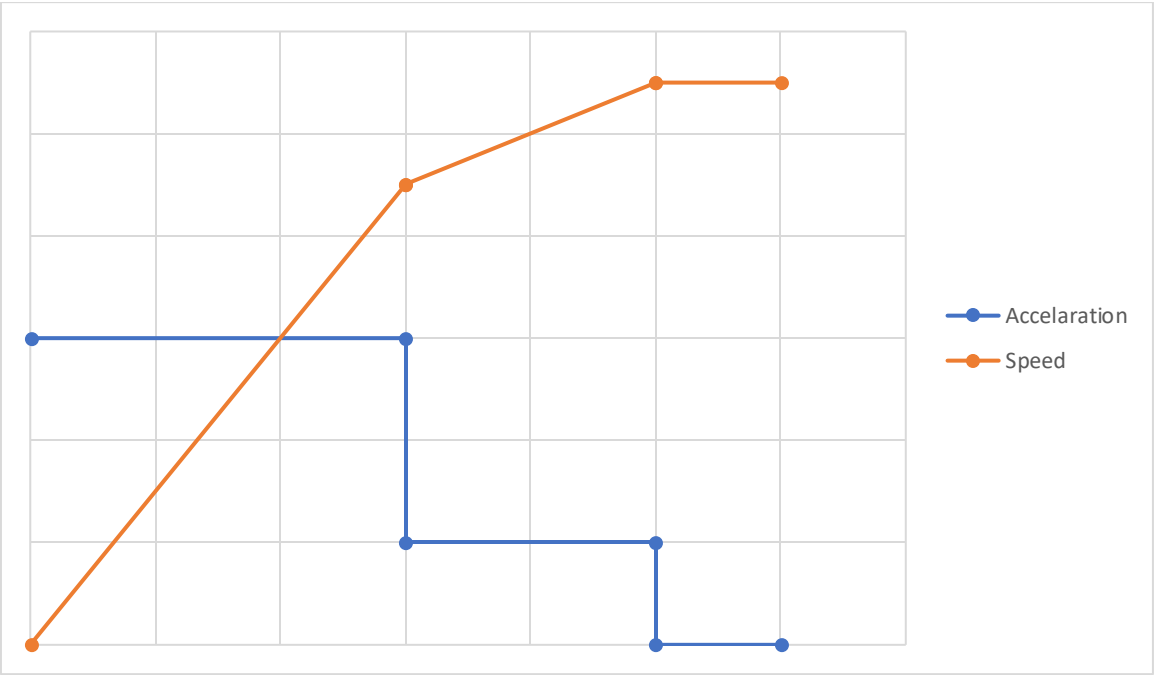
Physics

We will use acceleration variables to create dynamic speeds.



We use a chart in the style of the above. The acceleration is constant up to a certain speed, after which it steps up or down. (In the game, the values differ, this graph explains it clearest. For horizontal speed and acceleration, the acceleration steps up from a certain speed).

We also implement a max speed: this would be the resulting graph:



We use this concept separately for horizontal and vertical acceleration and speed.

If you want to change direction at high speeds, we want that to be snappy. For this, we make the deceleration higher. If we wanted the ground to feel “slippery”, we could turn this deceleration down.

Save Data and Level files.

In save.txt, we save in this order:

- Levelnumber
- Player x
- Player y
- X0 (for the camera)

An empty file will start the game from the beginning, a file with only a level number will spawn the player at the default location.

The level files describe a level with:

- “RB” for a row of blocks
- “CB” for a column of blocks
- “RS” for a row of spikes
- “CS” for a column of spikes
- “ED” for the end door

The first two following numbers are the x and y coordinate of:

- Collums: the “lowest” block, highest in value.
- Rows: the leftmost block, lowest in value.
- Door: the x and y coordinate.

The third number is the length/height of the row/column.

The game only reads 3 level files, with the names:

Level1.txt, level2.txt, level3.txt.

If you want to increase this, you need to change some variables in the code.

Collision detection

The following is the checklist where we based the collision detection on.

If a side-key is pressed:

- check which side
- How much do you move? (speed)
- Check if you collide
- if collide → move as far as you can go
- if you don’t collide → move

If up-key is pressed:

- check if on ground
- start jump
- You have to be able to execute other functions during jump
- if you collide with something: stop jump
- If Jump is done: start falling

If you walk of platform:

- start falling
- during fall: be able to execute other functions

Fall:

- Check if you can fall the max fall distance
- If not, move as far as you can
    - Stop the fall
- If you can, keep falling

On ground check has to be checked all the time

When you move while you jump, you don't have to account for the y-direction when checking for collisions in the x-direction, and vice-versa. This is because we implement the functionality of moving as far as you can go if a collision is detected on the intended trajectory.

> → if it would happen that you can overshoot the collision because of your increasing/decreasing your y-position, the game would receive a new keyEvent, check for collisions again and not detect a collision anymore.

When falling, we found that you do have to check for collisions in the x and y direction simultaneously. We use the same for-loop principles with moving as far as the player can, as in the directions separately, but combine them now. We check this in the move function to be sure it is executed.

Main class

At first, we restarted the game by creating a new game object, and disposing the frame of the original game. We soon found, however, that the game started to lag quite a bit after restarting or going to the next level.

After some testing, we concluded that this happened because of the multiple game objects that existed together. To solve this problem, we changed the structure of our code, by introducing a "Main" class, which could function as a sort of constructor of the Game class. Now we could keep using one game instance, and reset the values on restarting, or going to the next level.

Bugs

We had some issues with the JPanel, where the code runs too fast to dispose the frame and to create a new frame. To resolve this issue we added a sleep statement. If the problem still persists, it is easily resolved by restarting the code.