# OGP Assignment 2018-2019: **Jumping Alien** (Part II)

This text describes the second part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. Therefore, all grades are scored based on this assignment. The assignment is preferably taken in groups consisting of two students. It is, however, possible to do the project individually. In that case you need not implement some (small) parts of the assignment, as indicated in the following section. In principle, you should work out your solutions for the second and third parts of the assignment with the partner you chose for the first part. You are, however, allowed to start working with a new partner, or to work out the rest of the project on your own. Changes must be reported to ogp-inschrijven@cs.kuleuven.be before March 31, 2019.

If during the semester conflicts arise within a group, this should be reported to ogp-inschrijven@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

The assignment consists of three parts. The first part focusses on a single class, the second part on associations between classes, and the third part on inheritance and generics. After handing in the third part, the entire solution must be defended before Professor Steegmans or Professor Jacobs.

A number of teaching assistants (TAs) will advise the students and answer their questions. More specifically, each team has a number of hours where the members can ask questions to a TA. The TA plays the role of consultant who can be hired for a limited time. In particular, students may ask the TA to clarify the assignment or the course material, and discuss alternative designs and solutions. However, the TA will not work on the assignment him/herself. Consultations will generally be held in English. Thus, your project documentation, specifications, and identifiers in the source code should be written in English. Teams may arrange consultation sessions by email to ogp-project@cs.kuleuven.be. Please outline your questions and propose a few possible time slots when signing up for a consultation appointment.

To keep track of your development process, and mainly for your own convenience, we encourage you to use the *Git* version control system. Instructions on how to obtain a private repository on GitHub, already populated with the provided GUI code (see section 4), as well as a short tutorial, will appear in a separate document on Toledo.

The goal of this assignment is to test your understanding of the concepts introduced in this course. For that reason, we provide a graphical user interface and it is up to the teams to implement the requested functionality. The requested functionality is described at a high level in this document and it is up to the student to design and implement one or more classes that provide this functionality. The grades for this assignment do not depend only on functional requirements. We will also pay attention to documentation, accurate specifications, re-usability and adaptability.

This text extends the assignment for the first part of the project. Portions of the original assignment that have not been changed are colored blue. Portions of the original assignment that have been changed are colored red. New parts are simply printed in black.

# 1 Assignment

This assignment aims to create a platform video game that is loosely based on the Super Mario series by Nintendo. In **Jumping Alien**, the player controls a little green character called `Mazub`. The goal of the game is to move `Mazub` safely through a hostile two-dimensional game world, avoiding or destroying enemies and collecting items. In the first part of the assignment we focused on a single class `Mazub` that implements the player character with the ability to jump and run to the left and right in a simplistic game world. In this second part we extend `Mazub`, introduce a more complex game world and further game objects that interact with `Mazub`. Thus, the focus of this part of the assignment is on associations between classes. Your solution may contain additional helper classes (in particular classes marked *@Value*) and may already make use of concepts such as inheritance. In the remainder of this section, we describe the classes `World`, `Mazub` and `Plant` in more detail. **All aspects of the class `Mazub` shall be specified both formally and informally. All aspects of the class `World` and the class `Plant` shall only be specified in a formal way.**

## 1.1 The Class `World`

**Jumping Alien** is played in a rectangular game world that is composed of a fixed number of $X$ times $Y$ adjointly positioned, non-overlapping *pixels*.

Both $X$ and $Y$ shall be positive. Each pixel is located at a fixed position $(x, y)$. The position of the bottom-left pixel of the game world shall be $(0, 0)$. The position of the top-right pixel of the game world shall be $(x_{max}, y_{max}) = (X - 1, Y - 1)$. All pixels are square in shape. For the purpose of calculating locations, distances and velocities of game objects, each pixel shall be assumed to have a side length of $1\ cm = 0.01\ m$. The class `World` shall provide methods to inspect the game world's dimensions $X$ and $Y$ and the *length* of its tiles.

### 1.1.1 Tiles

Pixels are grouped together to `tiles`. All tiles are square and of the same size, given as the *length* of all the tiles' sides in pixels, i.e. each tile consists of *length* times *length* pixels. As a result of this, the number of horizontal pixels ($X$) of a world, as well as its number of vertical pixels ($Y$) must be divisible by *length* without remainder. Each tile is located at a fixed position $(x_T, y_T)$. Similar to pixels, the position of the bottom-left tile of the game world shall be denoted as $(0, 0)$, and the position of the top-right tile of the game world shall be $((X/length) - 1, (Y/length) - 1)$.

A game world shall have geological features, including passable terrain (air, water, magma) and impassable terrain (solid ground). The position of geological features of the game world shall always be determined by means of the position of a tile $(x_T, y_T)$ bearing the feature. The feature then affects all pixels belonging to that tile. If a tile of the game world is not assigned a feature explicitly, "air" should be used as the default. The class `World` shall provide a method to ask for the feature whose bottom-left pixel is positioned on a given position. That method must return its result in (nearly constant) constant time.

All aspects concerning tiles and their geological features shall be worked out in a **total way**.

### 1.1.2 Game Objects

The world shall contain game objects such as the player character `Mazub`, enemy characters and collectable items. In this version of the game, the game world can only contain `Mazub`s and plants. The next and final version of the game will introduce other types of game objects. Notice that a game world can have several `Mazub`s, only one of which is controlled by the player. Game objects are always rectangular and occupy a number of pixels. The position of game objects shall always be determined by the position $(x, y)$ of the pixel that is occupied by the bottom-left pixel of the game object. The presence of game objects in a tile only affects those pixels that are actually occupied by the game object.

Game objects may occupy any pixel of passable terrain tiles. Game objects may only occupy the top-most row of pixels of solid terrain tiles. Finally, some game objects may not overlap with other game objects. In this version of the game, `Mazub`s are not allowed to overlap with other `Mazub`s. Plants, on the other hand, may overlap with other plants and with `Mazub`s (although they may be eaten in that case as explained in section 1.2.10). Game objects overlap each other if and only if their inner layers overlap. This means that game objects do not overlap each other if only pixels in the outer layer of the one game object overlap pixels in the outer layer of the other game object (and vice versa). For a rectangular game object at position $(x, y)$ that occupies an area of $X_G$ times $Y_G$ pixels, its outer layer (also referred to as its perimeter) is defined by the left and right side of the game object, comprising of the coordinates $(x, y+1..y+Y_G-2)$ and $(x+X_G-1, y+1..y+Y_G-2)$, and the bottom and top sides of the game object, comprising of the coordinates $(x..x+X_G-1, y)$ and $(x..x+X_G-1, y+Y_G-1)$. An algorithm that performs sufficiently fine-grained collision detection is outlined in section 2.

Before the start of a new game, a game world is set up such that it contains at least one player character `Mazub` and no more than 100 other game objects (which may also be other `Mazub`s). That limit may change in future versions of the game, and may differ from world to world. The first `Mazub` added to a game world will be the one that can be controlled by the end user. Once the game is started, no other objects can be added to the game world. The initial position of all game objects is passed explicitly to those game objects at the time of their creation.

Game objects are to be killed as soon as their bottom-left position $(x, y)$ would leave the boundaries $(0, 0)..(x_{max}, y_{max})$ of the game world, if any, to which they belong. Further conditions for the "death" of a game object may be specified for each of the different types of game objects discussed below. As an example, `Mazub` is considered dead as soon as it has no hit points left. If a game object's death conditions are met while the object is still located within the boundaries of a game world, the dead game object shall be removed from the game world with a delay of $0.6s$ of game time. During this time the game object is not moving but may still passively interact with other game objects.

### 1.1.3 Target Tile

The ultimate goal of the game is to pilot `Mazub` towards a given target tile. The game is won when `Mazub` reaches that target tile, i.e, as soon as one of `Mazub`'s pixels overlaps with the given target tile. The game is lost as soon as `Mazub` is removed from the game world because `Mazub` has died or because `Mazub` has been positioned outside the boundaries of its game world. The target tile is set at the time a game world is created. It may change afterwards, even at times the game is actually being played. Notice that the target tile may be outside the boundaries of the game world to which it applies. It may also be located in impassable terrain. In those cases, `Mazub` may not be able to win the game because the target tile is out of reach.

All aspects concerning the target tile shall be worked out in a **nominal way**.

### 1.1.4 Visible Window

The Graphical User Interface (GUI) for **Jumping Alien** will, in most cases, only display a relatively small rectangular window of the game world. The exact dimensions of this window shall be given in game-world pixels at the time the world is created. They are fixed from that moment on. The visible window for a game world shall never be bigger than the game world itself.

The window shall surround the player object `Mazub` as far as possible. More specifically, if the horizontal side of the window is at least 400 pixels larger than the horizontal side of `Mazub`, and `Mazub` is positioned at least 200 pixels from the the left and right borders of the game world, the window shall be positioned so that there are at least 200 pixels between all pixels occupied by `Mazub` and the left and right borders of the visible window. The same rule applies to the vertical positioning of `Mazub` in the window. If the above condition is not satisfied for some direction, the window must be positioned such that it completely fits in the world to which it applies, and such that at least some pixels of `Mazub` are in its range. If there is no `Mazub` in the game world, the visible window will have position $(0, 0)$.

*Students that are working out the project on their own, must not support the repositioning of the visible window as described above. The visible window in their game will always be positioned 100 pixels to the left and 50 pixels below the bottom-left pixel of `Mazub`. If that position is outside the boundaries of the game world, it is changed to coincide with the left side, respectively the bottom side of the game world.*

The class `World` shall provide methods to inspect the position (bottom-left corner) and the height and width of the visible window. All aspects concerning the visible window shall be worked out in a **defensive way**.

### 1.1.5 Advance Time

`World` shall further implement a method `advanceTime` to advance the state of the world over a given time duration $\Delta t$ in seconds. The duration $\Delta t$ shall never be less than zero and always be smaller than $0.2s$. The method iteratively invokes `advanceTime` on all game objects inhabiting the world, starting with the player object `Mazub`. It will handle all contacts with non-air terrain tiles, as well as all overlaps between `Mazub`s and plants. Finally, will also see to it that the visible window is adjusted appropriately *(not for students working on their own)*. The method `advanceTime` shall be worked out in a **defensive way**. No documentation must be worked out for this method.

## 1.2 The Class `Mazub`

The player character `Mazub` is a rectangular object of varying size that can move within a game world.

### 1.2.1 Position

As illustrated in Figure 1, `Mazub` occupies an area of $X_P$ times $Y_P$ pixels of a game world and `Mazub`'s position is given as $(x, y)$, denoting the pixel of the game world that is occupied by `Mazub`'s bottom-left pixel. `Mazub`s should never be positioned completely outside their game world. More precisely, the position of the bottom-left pixel must always be inside the boundaries of the world. Notice that this does not prevent `Mazub`s to partially cross the right and top border of the game world. `Mazub`s that are not part of a game world, are assumed to be positioned in a universe that is composed of pixels in the same way game worlds are composed of pixels. The universe is an area that spreads both horizontally and vertically from `0.0` to `Double.POSITIVE_INFINITY`.

When displaying `Mazub`s, their position will always be aligned with pixels of their game world. In other words, `Mazub`s will not be displayed as partially occupying some pixels of the game world. They either fully occupy a game world's pixel, or they do not occupy such a pixel at all. Nevertheless, the actual position of a `Mazub` is the position of its bottom-left corner in the game world. That actual position is given as $(x, y)$, in which both $x$ and $y$ are double precision floating-point values expressing the distance in meters of the bottom-left corner of `Mazub` to the bottom-left corner of the game world. The actual position of `Mazub` is not aligned with the game world, meaning that the actual position can be any position within the pixel in the game world occupied by the left-bottom pixel of `Mazub`.

All aspects concerning the position of `Mazub` shall be worked out defensively.

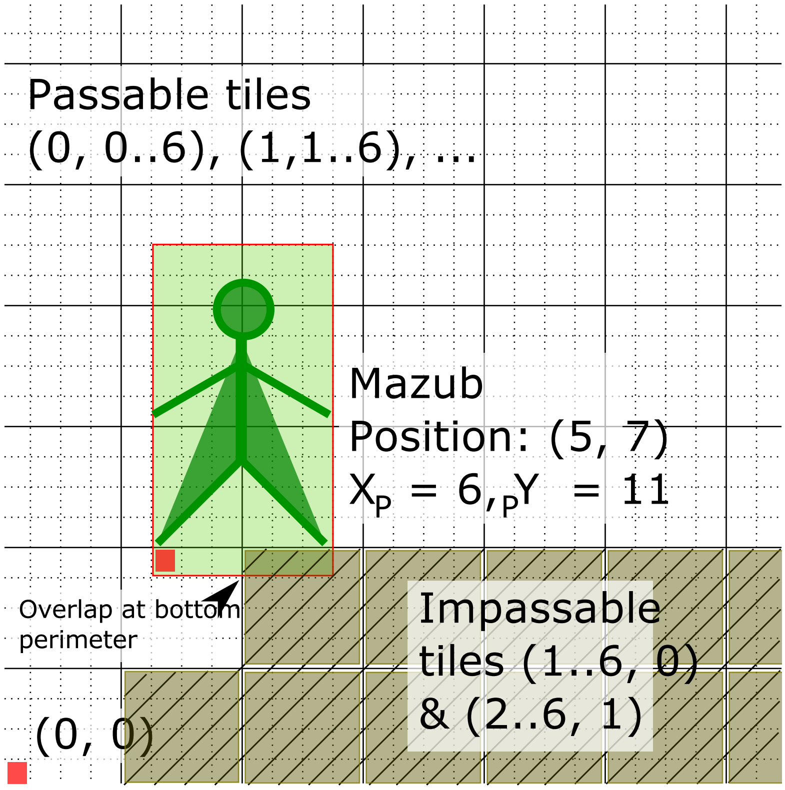


Figure 1: **Jumping Alien**: The game world with pixels, tiles and the player character `Mazub`. `Mazub` is located in passable terrain tiles, standing on the impassable tile $(2, 1)$.

### 1.2.2 Orientation

`Mazub` is oriented either to the left, to the right or to the front. When `Mazub` is moving, its orientation will be either to the left or to the right. When `Mazub` is not moving, and at least 1 second has passed since its last move has ended, `Mazub` will be oriented to the front.

All aspects concerning the orientation of `Mazub` shall be worked out nominally.

### 1.2.3 Horizontal velocity

`Mazub` is able to move horizontally in both directions. The horizontal velocity is negative if `Mazub` is moving to the left; it is positive if `Mazub` is moving to the right. The horizontal velocity of `Mazub` is expressed in meters per seconds and is constrained by a minimum value and a maximum value. If `Mazub` is standing straight while moving horizontally, the magnitude o its horizontal velocity (= absolute value) will not be below a minimum value of 1.0 m/s and will not exceed a maximum value of $3.0m/s$. If `Mazub` is ducking while moving horizontally, its horizontal velocity will have a constant value that is equal to $1.0m/s$ or $-1.0m/s$ depending on the direction in which `Mazub` is moving. Obviously, if `Mazub` is not moving horizontally, its horizontal velocity is equal to $0.0m/s$.

The minimum value and the maximum value for the horizontal velocity do not change during the game. However, in the future, the actual values for each of them may change both in case `Mazub` is standing straight and in case `Mazub` is ducking. The minimum value for the horizontal velocity will never be changed to a value below $1.0m/s$. The maximum value for the horizontal velocity will never be changed to a value below the minimum horizontal velocity. In future versions of the game, it must be possible to have different `Mazubs` with different minimum values and maximum values for their horizontal velocity.

All aspects concerning the horizontal velocity of `Mazub` shall be worked out totally.

### 1.2.4 Vertical velocity

`Mazub` is able to move vertically in both directions. The vertical velocity of `Mazub` is expressed in meters per seconds and will never exceed $8m/s$. That value will always be the same for all `Mazubs`, and will never change in future versions of the game. A positive value for the vertical velocity means that `Mazub` is moving upwards. With a negative vertical velocity, `Mazub` falls down. There is no specific lower bound for the vertical velocity. Obviously, if `Mazub` is not moving vertically, its vertical velocity is equal to $0.0m/s$.

All aspects concerning the vertical velocity of `Mazub` shall be worked out totally.

### 1.2.5 Running

The player character may run to the right side or to the left side of the game world, The class `Mazub` shall provide methods `startMove` and `endMove` to initiate, respectively to stop movement in a given direction. Obviously, you can not start moving if you are already moving, and you can not stop moving if you are not already moving. Moreover, dead `Mazubs` can also not start to move. These methods must be worked out nominally.

`Mazubs` can only move through passable terrain. However, as explained in section 1.1.2, the bottom side of `Mazub` may overlap with the top side of solid terrain tiles. Moreover, while `Mazub` is moving in its game world, only its outer side may overlap with the outer side of other `Mazubs` in that world.

Once `startMove` has been invoked, `Mazub` starts moving in the given direction with a horizontal velocity equal to its minimum value, accelerating with $a_x = 0.9m/s^2$ in that direction until the velocity reaches its maximum value. `Mazub`'s horizontal velocity after some $\Delta t$ seconds can be computed as $v_{x_{new}} = v_{x_{current}} + a_x \Delta t$, where $v_{x_{current}}$ is `Mazub`'s current horizontal velocity. Once $v_x$ equals its maximum value, `Mazub`'s horizontal velocity shall remain constant at that maximum value. The velocity of `Mazub` shall drop to zero immediately as `endMove` is invoked.

It must be possible to change the horizontal acceleration in future versions of the game. All `Mazubs` will always have the same value for the horizontal acceleration.

### 1.2.6 Jumping and Falling

The player character may also *jump*. Similar to running, the class `Mazub` shall provide methods `startJump` and `endJump` to initiate or stop jumping. These methods must be worked out defensively. Obviously, `Mazub` can not start jumping if it is already jumping, and `Mazub` can not stop jumping if it is not already jumping.

Once `startJump` has been invoked, `Mazub` starts moving with a velocity of $v_y = 8\ m/s$ in positive y-direction (i.e. upwards). Invoking the method `endJump` shall set `Mazub`'s vertical velocity to zero if the current vertical velocity is greater than zero. Additionally, if `Mazub`'s top perimeter or side perimeter overlaps with impassable terrain or with other `Mazubs` while `Mazub`'s vertical velocity is greater than zero, `Mazub`'s vertical velocity shall be set to zero, effectively ending the jump. Notice that `Mazub` can start jumping even when it is not located on top of solid ground or on top of another game object.

When `Mazub`'s bottom perimeter does not overlap with pixels belonging to impassable tiles or other `Mazubs`, `Mazub` shall *fall*. More specifically, `Mazub` shall accelerate with $a_y = -10\ m/s^2$ in positive y-direction until `Mazub`'s bottom perimeter reaches the top-most row of pixels of an impassable tile or of another `Mazub`, or when `Mazub` leaves the map. As `Mazub` stands on impassable terrain or another `Mazub`, $a_y$ shall be set to zero.

All methods that concern the vertical acceleration of `Mazub` shall be worked out using total programming. The values for the initial vertical velocity and the vertical acceleration do not change during the game, and will not change in future versions of the game.

### 1.2.7 Ducking

The player character may also duck so as to decrease its size. In future phases of the game this will be used to avoid enemies or to access narrow passages. `Mazub` can duck while standing still, while moving or while jumping. The class `Mazub` shall provide methods `startDuck` and `endDuck` to initiate and stop ducking. These methods are are to be implemented totally. Ducking affects the $X_P$ and $Y_P$ attributes of `Mazub` as explained in Section 1.2.9. Ducking also restricts $v_{xmax}$ to $1m/s$. In other words, `Mazub` moves at a constant horizontal velocity as long as `Mazub` is ducking.

For your implementation you may safely assume that $Y_P$ for a ducking `Mazub` is smaller than $Y_P$ for a `Mazub` who is not ducking. If `endDuck` is invoked in a location where the appropriate non-ducking $Y_P$ would result in `Mazub` overlapping with impassable terrain or with some other `Mazub`, `Mazub` shall continue to duck until appropriate space is available.

### 1.2.8 AdvanceTime

The class `Mazub` shall provide a method `advanceTime` to update the position and the velocity of `Mazub` based on its current position, its current velocity, its current acceleration and a given time duration $\Delta t$ in seconds. This duration $\Delta t$ shall never be less than zero and always be smaller that $0.2s$. The method `advanceTime` will also update the image to be used for displaying `Mazub`, as we will explain in section Section 1.2.9.

The horizontal velocity of `Mazub` may be computed as $v_{x_{new}} = v_{x_{current}} + a_x \Delta t$, in which $v_{x_{current}}$ denotes the current horizontal velocity of `Mazub`, and $a_x$ its horizontal acceleration. The horizontal position of `Mazub` may be computed as $x_{new} = x_{current} + v_{x_{current}} \Delta t + \frac{1}{2} a_x \Delta t^2$, in which $x_{current}$ denotes the current horizontal position of `Mazub`. Similar formulae hold for computing the vertical velocity and the vertical position of `Mazub` after some period of time. The method `advanceTime` must ensure that on the trajectory leading to the final position, `Mazub` never overlaps with impassable terrain or with other `Mazub`s in other ways than allowed as discussed in section Section 1.1.2. Moreover, the method `advanceTime` must remove `Mazub` from its world, as soon as its bottom left pixel leaves the boundaries ot that world.

You may safely assume that no in-game time elapses between the last invocation of `advanceTime` and a subsequent invocation of the methods `startMove`, `endMove` or any of the `start...` and `end...` methods specified in this section. As for the corresponding method in the class `World`, no documentation must be worked out for the method `advanceTime` in the class `Mazub`. The method `advanceTime` shall be worked out in a **defensive**

**way**.

### 1.2.9 Character Size and Animation

*Students that are working out the project on their own, must not support an array of sprites. Their constructor of the class `Mazub` will completely ignore the array of sprites supplied to it. Instead, they will always use the default sprite as the current sprite to display `Mazub`.*

To display the player character in a visualisation of the game world, `Mazub` will have an array of sprites. A sprite is an image of $X_P$ times $Y_P$ pixels. A class `Sprite` is provided with the assignment. It offers a.o. the methods `getHeight` and `getWidth` to inspect the size of a sprite. The constructor of `Mazub` shall have a parameter to accept an array of `sprites`. That array must contain an even number of at least 10 images. Moreover, the array of sprites must have an equal number of images for running to the left and for running to the right.

The class `Mazub` shall further provide a method `getCurrentSprite` that returns the image to be used for displaying the player character. The image to be displayed must reflect the current state of the player character, as described in Table 1. If there are multiple such images (i.e., $m > 0$), these images shall be used alternating. Starting with the image $\texttt{images}_i$ with the smallest $i$ appropriate for the current action, a different image shall be selected every $75ms$. As `Mazub` continues to run, $\texttt{images}_{i+1}$ shall be displayed, followed by $\texttt{images}_{i+2}$, and so on. Once the set of images for the current action is exhausted, i.e., $\texttt{images}_{i+m}$ has been displayed and `Mazub` is still running, the above procedure repeats starting from $\texttt{images}_i$.

Importantly, each $\texttt{images}_i$ may have different dimensions. Independently of whether `getCurrentSprite` is invoked, `Mazub`'s $X_P$ and $Y_P$ must always be reported by the inspectors as the dimensions of the image that is appropriate with respect to `Mazub`'s current state. Whenever `Mazub` should turn to a new (larger) sprite, and `Mazub` would overlap with impassable terrain or with some other `Mazub`, `Mazub` will keep the old sprite until the switch becomes possible. For your implementation and for the documentation you may safely assume that no in-game time elapses between the last invocation of `advanceTime` and a subsequent invocation of the methods `getCurrentSprite` or `getHeight` and `getWidth`.

All aspects concerning sprites must be worked out in a defensive way. No documentation is required for methods or aspects of methods related to sprites.

### 1.2.10 Hit-Points, Metabolism and Enemy Interaction

`Mazub` is assigned a number of hit-points. All numerical aspects related to these hit-points shall be worked out using integer numbers and total programming. At the beginning of a game, `Mazub` is assigned 100 hit-points. The current number of hit-points may change during the game as a response to actions performed by `Mazub`. It shall, however, never be lower than 0, indicating `Mazub`'s death, and never be greater than 500.

`Mazub` can gain hit-points by consuming Alien `Plant` objects. As any of `Mazub`'s perimeters overlap with a living `Plant` object while `Mazub` has less than 500 hit-points, `Mazub`'s hit-points shall be increased by 50 up to a maximum of 500 . Otherwise the living `Plant` is not affected by the contact. If any of `Mazub`'s perimeters overlap with a dead `Plant` object its hit points shall be diminished with 20, and the dead `Plant` object shall be removed from the game world.

`Mazub` can lose hit-points due to contact with water or magma. As long as any of `Mazub`'s pixels overlap with a terrain tile containing water, `Mazub`'s hit-points shall be decreased by 2 per 0.2 $s$. If `Mazub` remains in contact with water for less than 0.2 $s$, no hit-points shall be deduced. As long as any of `Mazub`'s pixels overlap with a terrain tile containing magma, `Mazub`'s hit-points shall be decreased by 50 per 0.2 $s$. Any contact with magma shall immediately incur the loss of hit-points but no more than 50 hit-points shall be deduced per 0.2 $s$. If `Mazub`'s pixels overlap with water and magma at the same time, `Mazub` shall only lose hit-points because of the overlap with magma.

The class `Mazub` shall provide a method to inspect the current number of hit-points.

## 1.3 Plants

Alien `Plants` primarily act as food for `Mazub`. They neither jump nor fall nor duck but are capable of hovering on *passable and impassable* terrain. `Plants` possess one hit-point and are destroyed upon contact with a hungry `Mazub`. Plants die after 10 seconds of game time, meaning that their hit points are set to 0. The number of seconds plants live may change in future versions of the game. It may even differ from plant to plant. As for `Mazubs`, dead plants shall be removed from their game world with a delay of 0.6 seconds.

Until they die, `Plants` shall move to the left and right with a constant horizontal velocity of 0.5 $m/s$ for 0.5 $s$ of game time, alternating and starting to the left. As for `Mazubs`, plants will be destroyed as soon as they leave the boundaries of their world. Contact with other plants does not affect `Plants` and they also do not lose hit-points when making contact with water or magma.

Plants are created with an array of two sprites that are to be returned by `getCurrentSprite` for movement to the left (default, index 0) and to the right (index 1). *As for `Mazub`, the constructor of the class `Plant` worked out by students working on their own, will completely ignore the array of sprites supplied to it. Instead, they will always use the default sprite as the current sprite to display a plant.*

All aspects of the class `Plant` shall be specified only in a formal way.

# 2 Collision Detection

Your implementation may employ the left, right, top and bottom perimeters of game objects to determine if a game object in its current position is capable of conducting certain actions, such as moving or jumping. Specific interactions of a game object with overlapping terrain or with other game objects have been specified in the previous sections.

This section describes an idea for a simple algorithm to detect collisions between game objects on the one hand, and between game objects and terrain features on the other hand. The focus of the approach presented here is on efficiency and on achieving a high enough precision for the purpose of your game implementation. The algorithm does, by no means, aim at realistically modelling physics and you are free to opt for another algorithm in your implementation of the functional requirements of **Jumping Alien**.

Our algorithm relies strongly on features of the game world described above: all geological features and game objects are rectangular in shape and in-game-positions are based on square pixel coordinates. Furthermore, game objects move in isolation, one after another, while all other game objects do not move. Fig. 2 illustrates this setup: `Mazub` positioned at $(x, y)$ and `advanceTime` is invoked with some $\Delta t$ that, given `Mazub`'s current velocity and acceleration would result in `Mazub` moving to $(x', y')$. However, we wish to detect if `Mazub` overlaps with any of the terrain features or game objects (blue rectangles) on its way from $(x, y)$ to $(x', y')$.

Intuitively our algorithm aims to slice the time advancement $\Delta t$ into smaller fractions $dt$ of time that ensure that a game object moves approximately 1 $cm$, the side length of a pixel, per time slice. This allows us to split the entire movement into fractional movements from one pixel to another, followed by immediate checks for overlapping of the moving game object with the surrounding terrain features as well as with other game objects. Indeed, after moving for seven of these time slices, we see that `Mazub` overlaps with one of the blue rectangles (overlapping pixels marked in dark blue) and take

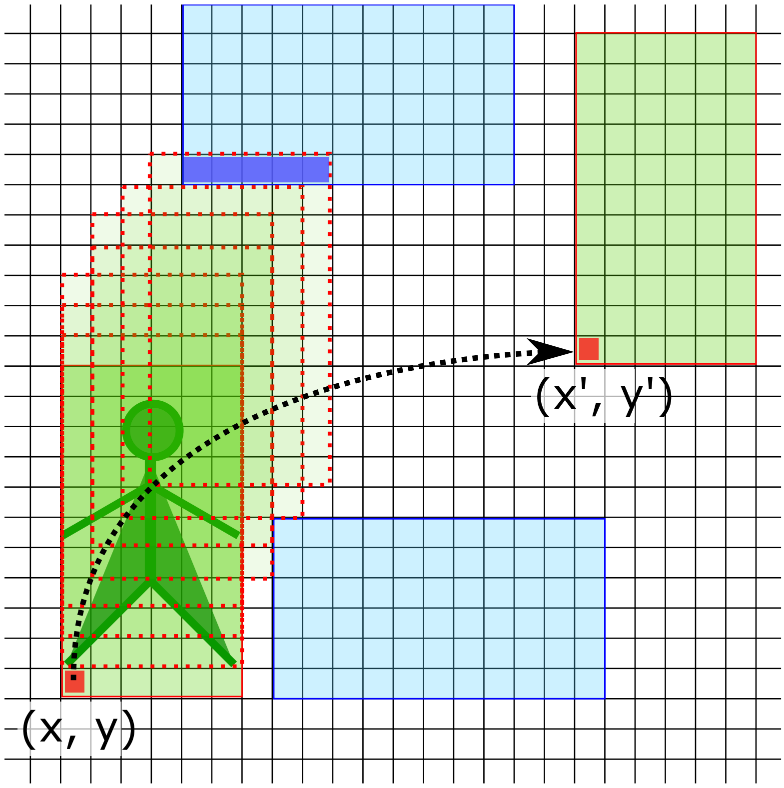


Figure 2: **Jumping Alien**: Collision detection.

actions as specified in the previous section.

As can be seen, choosing a reasonably small $dt$ is key for achieving acceptable performance and precision. We propose to determine $dt$ as the time needed to travel 0.01 $m$ at the current velocity of the game object that is to be moved. Specifically, we compute $dt$ as follows:

$$dt = \frac{0.01}{\sqrt{{v_x}^2 + {v_y}^2} + \sqrt{{a_x}^2 + {a_y}^2} * \Delta t}$$

Now we iteratively advance time for $dt$ (using the effective velocities *and* acceleration) and compute the in-game-position of the moved game object. Based on the in-game-position we can determine if some object $((x, y), X_P, Y_P)$ does not overlap with another object $((x', y'), X'_P, Y'_P)$: the two objects do not overlap if $(x + (X_P - 1) < x' \vee x' + (X'_P - 1) < x \vee y + (Y_P - 1) < y' \vee y' + (Y'_P - 1) < y)$; otherwise they do overlap.

The game object will keep its position if something hampers its movement over the smaller time step $dt$. In other words, you must not try to bring the moving game object as close as possible to the obstacle. In such a case, the smaller time step *dt* is not counted as being passed. Instead, the moving game object will change its state (e.g., stop jumping), and the calculated dt will be applied again to the moving object in that new state.

As other strategies to detect collisions are allowed, methods for collision detection shall not reveal in their documentation any internal details concerning the actual strategy. However, this does not apply to other methods that you may introduce, such as a method `collidesWith(other)` that must reveal in its documentation when "this" collides with "other".

# 3 Storing and Manipulating Real Numbers as Floating-Point Numbers

In your program, you shall use type **double** as the type for variables that conceptually need to be able to store arbitrary real numbers, and as the return type for methods that conceptually need to be able to return arbitrary real numbers.

Note, however, that variables of type **double** can only store values that are in a particular subset of the real numbers (specifically: the values that can be written as $m \cdot 2^e$ where $m, e \in \mathbb{Z}$ and $|m| < 2^{53}$ and $-1074 \leq e \leq 970$), as well as positive infinity (written as `Double.POSITIVE_INFINITY`) and negative infinity (written as `Double.NEGATIVE_INFINITY`). (These variables can additionally store some special values called *Not-a-Number* values, which are used as the result of operations whose value is mathematically undefined such as 0/0; see method `Double.isNaN`.) Therefore, arithmetic operations on expressions of type **double**, whose result type is also **double**, must generally perform *rounding* of their mathematically correct value to obtain a result value of type **double**. For example, the result of the Java expression `1.0/5.0` is not the number 0.2, but the number[1]

$$0.200000000000000011102230246251565404236316680908203125$$

When performing complex computations in type **double**, rounding errors can accumulate and become arbitrarily large. The art and science of analysing computations in floating-point types (such as **double**) to determine bounds on the resulting error is studied in the scientific field of *numerical analysis*.

However, numerical analysis is outside the scope of this course; therefore, for this assignment we will be targeting not Java but *idealised Java*, a programming language that is entirely identical to Java except that in idealised Java, the values of type **double** are exactly the extended real numbers plus

[1]You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`.

some nonempty set of *Not-a-Number* values:

$$\textbf{double} = \mathbb{R} \cup \{-\infty, +\infty\} \cup \mathit{NaNs}$$

Therefore, in idealised Java, operations in type **double** perform no rounding and have the same meaning as in regular mathematics. Your solution should be correct when interpreting both your code and your formal documentation as statements and expressions of idealised Java.

So, this means that for reasoning about the correctness of your program you can ignore rounding issues. However, when testing your program, of course you cannot ignore these. The presence of rounding means that it is unrealistic to expect that when you call your methods in your test cases, they will produce the exact correct result. Instead of testing for exactly correct results, it makes more sense to test that the results are within an acceptable distance from the correct result. What "acceptable distance" means, depends on the particular case. For example, in many cases, for a nonzero expected value, if the relative error (the value $|r-e|/|e|$ where $r$ and $e$ are the observed and expected results, respectively) is less than 0.01%, then that is an acceptable result. You can use JUnit's `assertEquals(double, double, double)` method to test for an acceptable distance.
http://introcs.cs.princeton.edu/java/91float/.

# 4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on `Mazub`. The user interface is part of the Eclipse project that you can get from Github. You will need to pull the updated GUI for this part from Github, as explained on Toledo.

You will find a folder `src-provided` that contains the source code of the user interface, the `Util` and `Sprite` class and further helper classes. Generally, the files in this folder require no modification from your side. The classes that you develop must be placed in the folders `src` (implementation classes) and `tests` (test classes).

To connect your implementation to the GUI, update your class `Facade` in package `jumpingalien.facade` from part 1 to implement the updated `IFacade` interface. `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, you may execute the `main` method in the provided class `jumpingalien.JumpingAlien`.

After starting the program, you can press keys to modify the state of the program. Commands are issued by pressing the `left`, `right`, `up` and `down` arrow keys to start running to the left, right, and to start jumping and ducking, respectively. That is, pressing the above keys will invoke `startMoveLeft`, `startMoveRight`, `startJump` or `startDuck` on your Facade. Releasing these keys invokes `endMoveLeft`, `endMoveRight`, `endJump` and `endDuck` accordingly. Pressing `Esc` terminates the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Mazub`. No additional grades will be awarded for changing the GUI.

## 5 Testing

We will verify that your implementation works properly by running a elaborated number of JUnit tests against your implementation of `Facade`. As described in the documentation of `IFacade`, the methods of your `Facade` class shall only throw `ModelException`. The test suite that we will use to evaluate part 2 is distributed along with the assignment. You will find it in the folder `tests`. If you run the suite, it will yield a total score that will be integrated in your final score for the course.

## 6 Submitting

The solution must be submitted via Toledo as a jar file individually by all team members before the 15th of April 2019 at noon. Follow the instructions on Toledo (under Project:Assignment) to submit a proper JAR.

## 7 Feedback

There will be no feedback session to discuss this part of the project. Your solution for this part will be evaluated during the final defense. You are free to change your solution for this part while working out the final part of the project.

Table 1: Association between sprite index and character behaviour.

| **Index** | **To be displayed if `Mazub`...** |
|---|---|
| 0 | is not moving horizontally, has not moved horizontally within the last second of in-game time and is not ducking. |
| 1 | is not moving horizontally, has not moved horizontally within the last second of in-game time and is ducking. |
| 2 | is not moving horizontally but its last horizontal movement was to the right (within 1s), and the character is not ducking. |
| 3 | is not moving horizontally but its last horizontal movement was to the left (within 1s), and the character is not ducking. |
| 4 | is moving to the right and jumping and not ducking. |
| 5 | is moving to the left and jumping and not ducking. |
| 6 | is ducking and moving to the right or was moving to the right (within 1s). |
| 7 | is ducking and moving to the left or was moving to the left (within 1s). |
| $8..(8+m)$ | the character is neither ducking nor jumping and moving to the right. |
| $(9+m)..(9+2m)$ | the character is neither ducking nor jumping and moving to the left. |