Analysis and Functionality Report

Multidisciplinary Engineering Project

Information Technology

Implement a non-trivial algorithm

Project Stars

by

group 9

Bryan Van Huyneghem
Jonathan Van Damme
Michiel Mortier
Robin Goussey
Jelle Hamerlinck

Coaches: Veerle Ongenae & Leen Brouns

University of Ghent
Faculty of Engineering and Architecture
Bachelor of Science in Information Engineering Technology

Academic year 2016-2017

“Are you living in a computer simulation?”

Niklas Boström

Table of contents

List of figures and/or list of tables

List of abbreviations

CSV	A CSV is a comma separated values file which allows data to be saved in a table structured format. CSVs look like a garden-variety spreadsheet but with a .csv extension. Traditionally they take the form of a text file containing information separated by commas, hence the name. [4]
GUI	The GUI or Graphical User Interface is a graphical (rather than purely textual) user interface to a computer.

List of explanatory vocabulary

Activity diagram	Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system. [2][7][17]
Class diagram	Shows a collection of static model elements such as classes and types, their contents, and their relationships. [2][7]
High-level programming language	A programming language with strong abstraction from the details of the computer. It may use natural language elements and be easier to use than low-level programming languages, which are closer to a computer's instruction set architecture.
Object oriented	Object-oriented programming is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate, rather than the logic required to manipulate them. [14]
Use-case diagram	Shows use cases, actors, and their relationships. They are developed from the point of view of the user and are therefore not very technical. [2][7]
Thread	A thread can be seen as a smaller process inside a process There can be multiple threads inside the process, and they all have access to the same sources as the other threads inside the process. [15]

Definition list for project-specific vocabulary

Agriculture	*Agriculture* is one of four *technologies* and aids the *organism* in their combat against *disasters* such as famine. *Agriculture* indirectly influences the *life quality* of an *organism* through *usable landmass*.
Architecture	*Architecture* is one of four *technologies* and aids the *organism* in their combat against *disasters such as natural disasters*. *Architecture* directly influences the *life quality* of an *organism* and the *progression*.
Atmosphere	The *atmosphere* of a *planet* determines how well the composition is suited for the *organism* and, combined with *distance*, they play a vital role in determining a *planet*'s *temperature*.
Attribute	An *attribute* is a characteristic for a *planet* or *organism*. *Attributes* are collected and displayed as a whole in the *information panel*.
Breakthrough	A *breakthrough* is part of the *mechanic events* and is Project Stars approach to simulate reality by adding the randomness and unpredictability of daily life. *Breakthroughs* have positive effects on the *planet* and *organism*. They appear in a wide variety and their frequency and gravity depend on how many *turns* have passed.
Cap	The highest amount of *technology* points an *organism* can own. This *cap* is set at 15 for the *technologies agriculture*, *architecture* and *medicine*, and is set at 30 for the *technology engineering*.
Disaster	A *disaster* is part of the *mechanic events* and is Project Stars' approach to simulate reality by adding the randomness and unpredictability of daily life. *Disasters* have negative effects on the *planet* and *organism*. They appear in a wide variety and their frequency and gravity depend on how many *turns* have passed.
Distance (from star)	The *distance* between a *planet* and a *star* is set by Project Stars to be between 15,0000,000 km and 360,000,000 km.
Engineering	*Engineering* is one of four *technologies* and aids the *organism* in their combat against *disasters*. *Engineering* directly influences the *life quality* of an *organism* and is a major component of the *progression mechanic*.
Event	An *event* is Project Stars approach to simulate reality by adding the randomness and unpredictability of daily life. *Events* can occur as being beneficial or harmful to the *planet* and *organism*, the former being a *breakthrough* and latter being a *disaster*. A full list of all *disasters* and all *breakthroughs* with their corresponding effects can be found in *Appendix C*.
Goal	The *user*'s *organism* has successfully survived and reached another *planet*, if and only if it reaches a *progression* of 1000.
GZ *or* **Goldilocks' Zone**	The *Goldilocks' Zone* -- or in short *GZ* -- is a zone at a set *distance* from its *star* that has (easy) optimised *planet attributes* for an *organism*'s survival. Project Stars' *GZ* starts at 135 million km and ends at 180 million km, with its centre set at 150 million km.

Home Planet *or* **Main Planet**	The single *planet* that is inhabited by the *organism* whom is trying to escape said *planet* and near-imminent destruction.
Information Panel	The *information panel* collects and displays all *attributes* for a *planet*. The *main planet* displays its own *attributes* as well as its *organism*'s *attributes*.
Landmass	The *landmass* of a *planet* is a value between 10 and 100, and is defined as the percentage of land on a *planet*. The complement of *landmass* would be the percentage of water on the *planet*. *Landmass* is one of the *planet attributes* that is used to calculate the *usable landmass* of a *planet*.
Landmass (usable)	The amount of *usable landmass* of a *planet* is a percentage value calculated through the total *landmass* of a *planet*, the *technology* level in *agriculture* and the *technology* level in *architecture*.
Life Quality	The *life quality* is the combination of the *landmass*, *usable landmass*, average surface *temperature, population health* and the *technology* level in *architecture* and *engineering*. This *life quality* is a factor that is used to calculate the *total population* for the *organism*.
Main Planet *or* **Home Planet**	The single *planet* that is inhabited by the *organism* whom is trying to escape said *planet* near-imminent destruction.
Mechanic	A mechanic is a construct of rules or methods designed for interaction with the simulation, thus providing progression throughout said simulation. Different theories and styles with relation to the mechanic differ as to their ultimate importance in the simulation.
Medicine	*Medicine* is one of four *technologies* and aids the *organism* in their combat against *disasters* such as diseases. *Medicine* directly influences the regeneration of the *population health* of an *organism* and indirectly influences the *life quality* of said *organism*.
Multiplier	A *multiplier* is a factor that is multiplied with a *planet attribute*, such as *usable landmass*, or an *organism attribute* such as *population health* or *total population* to either increase or decrease said *attributes*.
Organism	The *organism* is the key ingredient in the *simulation* and is what *progresses* during one-thousand years to escape its *home planet*. It is hindered by *disasters* and aided by *breakthroughs*. *User* choice also impacts the *organism*'s well-being (*see more at*: *population health* and *life quality*).
Planet	A *planet* is part of the *star system* and has five characteristics or *attributes* that define what it is like: its (1) planet name, (2) *distance*, (3) *atmosphere*, (4) *landmass*, (5) *temperature* and its (6) *planet quality*.
Planet Quality	The *quality of a planet* is a value (score) between 0 and 100 that indicates how well-suited a planet is to be inhabited by an *organism*. It is initially presented to the user in an effort to guide them in their (easy) selection of a *main planet*. The quality of a *planet* is determined by the amount of *landmass* and its average surface *temperature* (which is determined through *distance* and *atmosphere*). As such, it covers all *planet attributes*.

Population (health)	The *population health* (0-100) of an *organism* is an indicator for its well-being and its regeneration is dependent on the *technology* level in *medicine*. It is initially set to 100 and is possibly negatively affected by the health *multiplier* as a result of *disasters*.
Population (total)	The *total population* of the *organism* is the total amount of species of said *organism* that are living on the *main planet*.
Progression	The *progression* mechanic is what indicates the *organism's* level of sophistication on a scale of 0 to 1,000 and is what ultimately leads to the *organism*'s escape from its *home planet* -- in other words the successful finalisation of the *simulation*. If the population is not on the decline, the *progression* is dependent on the *technology* level in *medicine, architecture, engineering, life quality* and the *total population*. However, in a scenario where *population* has decreased in relation to the previous *turn*, *progression* is solely dependent on *life quality* and the difference in *population* between this *turn* and the previous. *Progression* is visualised as a progression bar that tells the *user* what the total *progression* is and how much *progression* will be gained or lost upon ending the current *turn*.
Radius (planet)	The *radius* of a planet is determined through the *distance* of the *planet* to its *star*, but it is only used in the GUI to draw the planets.
Research (focus)	An *organism*'s *research* is a way to passively gain a point in a particular *technology* every five *turns*.
Rings (planet)	A *planet ring* is a possible location for a *planet* around its *star*. Project Stars has a set total of eleven *rings* that it chooses at random *distances* from the *star*, but allows for expandability for even more *rings*. The *user* is guaranteed to have at least 3 *rings* (or possible *planet locations*) in the *GZ*.
Simulation	Project Stars is a *simulation* that shows the user's *organism's star system* and the *progression* that is made by the *organism* towards its ultimate *goal*.
Star	The *star system*'s *star* is the *organism's* driving force before behind the reasoning to escape its *planet*. In precisely one-thousand years, it will engulf the *planet* in massive amounts of solar winds and solar radiation, annihilating anything that lives.
Star System	A *star system* is the environment in which the *simulation* takes place. It contains a single, near-death *star* and a number of *planets* between 5 and 7.
Technology	*Technology* is the *mechanic* that protects the *organism* from complete annihilation. There are four technologies: agriculture, architecture, medicine and engineering *(see more at: medicine, agriculture, architecture, engineering*). At the start of the *simulation*, the *user* is allowed twelve points which can be spent on the three first technologies to their heart's content. *Technology* can be advanced passively via the *mechanic research* focus every x amount of turns or via a *Breakthrough*.

Temperature (surface)	The *surface temperature* of a planet in °C is defined per formula using the *distance* between star and planet, and the planet's *atmosphere* quality.
Turn	Project Stars limits the amount of turns to 100, which means that each *turn* is equal to exactly 10 years. A *turn* is the mechanic of continuing in the *simulation* and progressing through the one-thousand final years that the *organism* has to escape its *home planet*.
User	The *user* is the person who chooses the main planet, spends the initial twelve *technology* points and takes critical decisions before, during and after *disasters* and *breakthroughs* which both alter the *progression* speed of the *organism* through *technologies*.

Introduction

Welcome to Project Stars' Analysis and Functionality Report. Project Stars is a simulation by Bryan Van Huyneghem, Michiel Mortier, Jonathan Van Damme, Robin Goussey and Jelle Hamerlinck. This simulation was commissioned by the University of Ghent as part of the Bachelor of Science in Information Engineering Technology education programme, under the subject Multidisciplinary Engineering Project. The exact assignment stated:

"The students have to create an application that uses a non-trivial algorithm in a Python version that exceeds version 3.0. This algorithm needs to be constructed in such a way such that it can be recycled for future work, and such that other engineers with a similar background can understand and edit the code. Furthermore, there must be enough theoretical support through Analysis and Design diagrams such that the general structure and functionality can be understood in a short period of time."

An estimated 90 hours work per member was expected to be spent on this project. A good chunk of this time was spent at University campus inside the classroom, though most work was done at home where the team collaborated together through Teamviewer, Skype, GitHub and Google Docs or worked individually on a person-specific task.

Within said assignment, the group quickly found common ground in our passion for games. After a short brainstorm, it was agreed that the creation of an interactive simulation of a star system was going to be the main goal.

The central star of the system is on the brink of dying, forcing the local intelligent species on a human-controlled planet to commence a race against the clock to evacuate its population from the planet and settle on a new one. The user is in control of the decisions the species make in their final one thousand years, all the while random events will either help or hinder the progression they make to escape fate.

The algorithm creates an environment where all kinds of planet properties, species properties and technologies, and random events interact with each other. After each turn in the simulation, every change is calculated and then presented to the user through the visual interface. In this interface, the user can adjust the research focus of their species, according to what seems necessary.

This Analysis and Functionality Report offers the reader with an extensive and in-depth look at Project Stars. Chapter 1 briefly describes the concept behind the simulation, whilst Chapter 2 provides the reader with a detailed, initial analysis of the project and work to be done. Chapter 3 solidifies the functionality and mechanics behind the simulation, whilst Chapter 4 will describe how these were implemented. Lastly, conclusions are drawn in regards to the end result and future work. Appendices include a Manual of the simulation, Technical Documentation and a list of all Events.

1. Concept

The Project Stars simulation follows the progression of an organism in its race against time and destruction. Its main goal is to escape its home planet and find a new, more distant world where its civilisation can further improve itself and continue its space quest. The organism's final one-thousand years of civilisation before catastrophe hits, symbolises the inevitable confrontation of a race with its annihilation.

At the start of each simulation, Project Stars offers the user a star system consisting of one star and a number of planets to choose from. These planets are generated on so called planet rings, which are possible locations for planets to exist at, each at a particular distance from the system's star. It should be noted that the Goldilocks' Zone or GZ contains the planet rings that spawn the most optimal and easiest planets for the organism to survive on. Similar to how Earth is a planet within the GZ, these planets come with very strong and positive attributes.

Each planet is equipped with an information panel where these attributes are listed, allowing the user to make an informed and optimal selection based on said information -- if they so choose to. The user's planet of choice turns this planet into the Main Planet or home planet.

Initially, a planet has (1) a distance from its star, (2) a planet radius, (3) the percentage of landmass, (4) a value indicating the quality of the atmosphere, and (5) an average surface temperature. Finally, values (3) and (5) are combined into one attribute that indicates the overall quality of the planet, which can be seen as an overall score for that planet.

It is assumed that the planet's organism commences its journey with the technology and wisdom of a humanlike civilisation during the 17th century..

As the simulation -- and therefore the organism -- progresses, a planet will also indicate what (6) the organism's overall health (population health) is and what (7) the total amount of living organisms (total population) is. The simulation progresses through what are called turns. One turn equals ten years, so the simulation supports a grand total of one-hundred turns before the organism's time runs out.

The organism has access to four technologies: agriculture, medicine, architecture and engineering. The user is allowed to spend a total of twelve points in the first three technologies to boost the initial values. These will further increase as the user's organism progresses throughout the simulation to a cap of 15 for the first three technologies and a cap of 30 for engineering.

Technologies play a major role in the survivability of the organism, as they will directly influence the susceptibility of the organism to disasters. The organism is allowed to research passively into one technology at a time, as such generating one point per five turns. Furthermore, they influence

the life quality, which is a grand total of the usable landmass, average surface temperature and population health.

Disasters are one of two, the other one being Breakthroughs, simulation mechanics that are part of the Events. An event is Project Stars approach to simulate reality by adding the randomness and unpredictability of daily life. Disasters have negative effects on the planet and organism, whilst Breakthroughs have positive effects. Both of them appear in wide variety. The frequency is set and the kind of event (out of 102) that occurs depends on the organism's progression. A full list of all disasters and all breakthroughs with their corresponding effects can be found in *Appendix C*.

The ultimate goal of the simulation is to escape the organism's home planet within one-thousand years. This can be achieved by reaching a progression of 1,000. Progression is the simulation's way of showing the user how advanced the organism has become. It is based on the amount of organisms alive – and therefore the life quality – and the engineering technology.

2. Analysis

It is vital that computer scientists and engineers are aware of the latest concepts and techniques with regards to computer programming, and software design and development. This knowledge and information allows them to think up and develop new ideas, algorithms and software through a concise, comprehensible and step by step approach [3].

Such an approach is called a *Systems Development Life Cycle* (SDLC), also referred to as the *Application Development Life Cycle* (Figure 1), and was first coined in the 1960s to "develop large scale functional business systems in an age of large scale business conglomerates for information systems activities revolving around heavy data processing and number crunching routines" [3]. It is used in software engineering and information systems to describe a process for planning, creating, testing and deploying a system or a piece of software. This cycle, as shown in Figure 1, below, consists of several phases and has proven to be an important component throughout the process and development of this project.

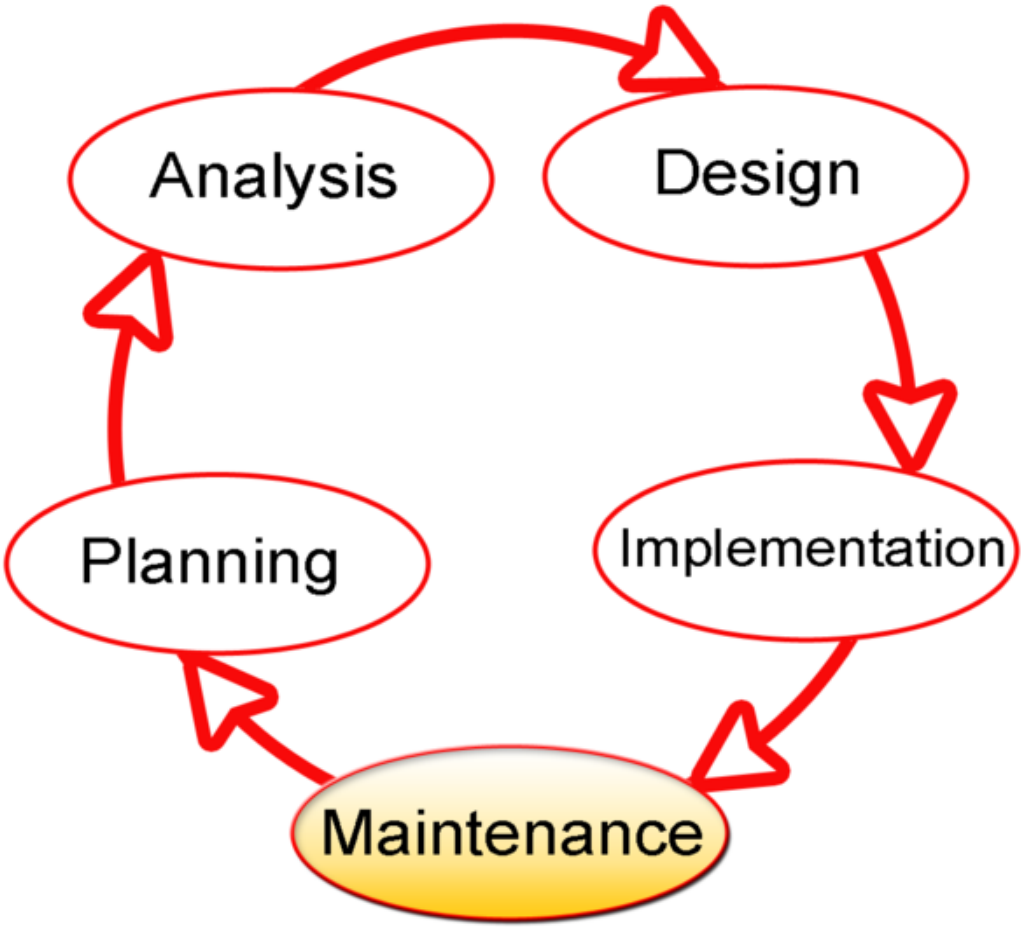

Figure 1. Systems Development Life Cycle (SDLC). The process for planning, creating, testing and deploying a system or piece of software. [3]

The first step that is to be taken upon the completion of one or more brainstorming sessions (*Planning*), is one that requires developers to sketch their ideas in such a way that their problems are, firstly, sharply defined and, secondly, their needs become apparent. The cycle refers to this as *Analysis*. This chapter on analysis offers the reader a closer look at the overall structure of the project [9]. *Chapter 3: Functionality and simulation mechanics* shall talk more in-depth about the *design* and functionality of the simulation itself. In *Chapter 4: Implementation*, the reader shall be guided through the actual code and *implementation* and GUI design. Finally, *conclusion and future work* will briefly mention a few things about the *maintenance* aspect of the project, however this will be touched upon in *Chapter 4: implementation*, as well.

One way of offering the reader with a comprehensible analysis of Project Stars is through the *Unified Modelling Language* (UML). UML is a general-purpose, development, modelling language whose intent is to provide a standard way to visualise the design of a system [3]. This part is vital in understanding the full picture of the project, because it introduces the reader with a simple overview of the choices that were made, whilst explaining the design through diagrams.

The goal of this analysis is to determine the problem, namely the implementation of a non-trivial algorithm through a simulation, game or something similar, as well as break down the notes that were taken throughout the Planning phase, into different pieces in an attempt to define all needs.

UML diagrams allow for a detailed, but easy to understand, way of offering information about the initial design and requirements, and how to deliver the required functionality. It quickly became apparent that the following three diagrams would suffice our needs towards visualising the main concept: a class diagram, an activity diagram and a use case diagram.

Although development and analysis are close friends, *Chapter 2: Analysis* is to be viewed as the skeleton to *Chapter 3: Functionality and simulation mechanics*. *Chapter 4: Implementation* will eventually give greater depth and clarification to the variables and methods that are introduced during Chapter 2. A brief and practical manual is provided to the user in *Appendix A: Manual*.

2.1 UML: Class diagram

A class diagram (Figure 2) shows the classes of a system, their inter-relationships and the operations and attributes of the classes [3]. This kind of diagram allows for conceptual analysis, giving the reader a detailed depiction of the design of the *object-oriented* project. Project Stars has been designed in the widely used *high-level programming language* Python. Python is an object-orientated language and as such, allows the usage of classes.

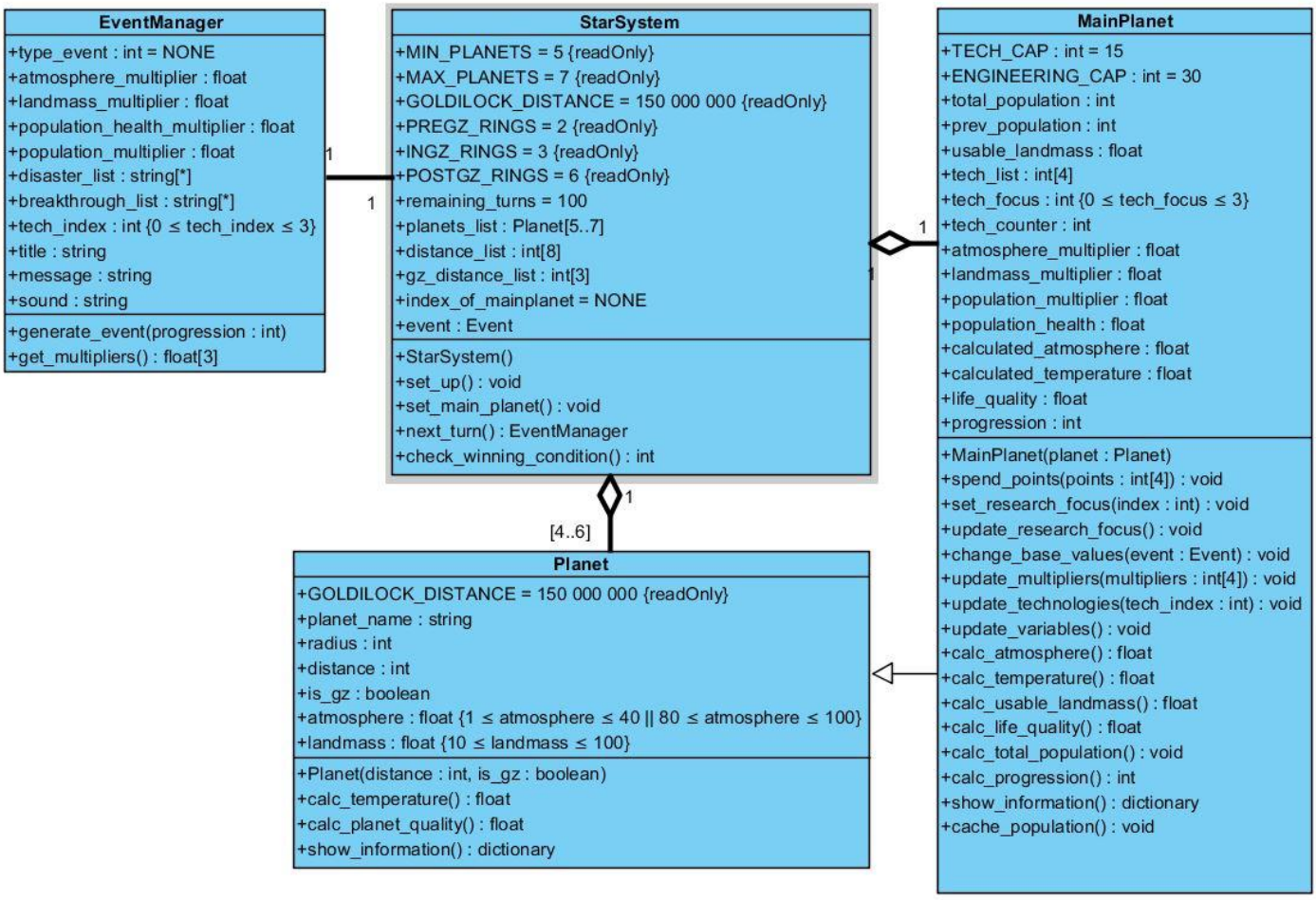

Figure 2. Class Diagram for Project Stars. The relation between Project Stars' classes and their methods and attributes.

It is not required to define a type when declaring a variable or object in Python. Instead, dynamic type checking, i.e. the process of verifying the type safety of a program at runtime, performs all the work for you. Each runtime object is associated with a type tag[1] that contains its type information. In other words, the compiler can tell which type a variable refers to. However, Python is nonetheless referred to as a strongly-typed language, because it is restrictive about how types can be intermingled [13].

Whilst it may at first be strange why Python is both strongly-typed and dynamically typed, the preceding explanation should resolve this confusion. To avoid confusion, overall, and remain ever

[1] A reference to a type

so comprehensible, the appropriate variable types that the compiler would determine, have been included in the class diagram on the previous page (Figure 2).

2.1.1 Class StarSystem

The class StarSystem should be viewed as a central class in Project Stars. Many methods were initially one of a kind, but were later expanded upon in the GUI classes to be handled. These classes will be analysed closely in *section 4.6* of *Chapter 4: Implementation*. A very brief introduction to this class will be given in *section 2.1.5* of this chapter.

The class StarSystem contains objects of type Planet, which are stores in a list (`planets_list`). It creates 11 fictitious rings at randomly generated distances, that are possible locations for planets to reside at. It also sets the Goldilocks' Zone's centre in `GOLDILOCK_DISTANCE` and limits the amount of planets that are initially created to a minimum of 5 and a maximum of 7 through the two constants `MIN_PLANETS` and `MAX_PLANETS`. These planets will reside on the previously mentioned rings. Some rings will remain empty and will henceforth no longer take part in the project. A separated list, called `gz_distance_list`, stores and guarantees the creation of three rings that reside in the Goldilocks' Zone.

StarSystem has a very clear set of instructions and methods that are executed in its initialise phase, ergo upon the construction of an object of this class. First of all, the amount of remaining turns is set to 100. During the method set-up, planet names are collected from a *Comma Separated Values* (CSV) document and stored in a list. This list is defined within the scope of the method `set_up()` and is as such not mentioned in the class diagram.

Next, within the same method, distances are randomly generated through formulas that define the distance as either preceding, residing in, or succeeding the Goldilocks' Zone. Via the random method, it will select names and distances from their respective lists and add these to the objects of type Planet that have been created and stored in the `planets_list`. The creation of these objects and their details will be discussed in *section 2.1.2* of this chapter.

Initially (and in the prototype), a method `show_planets()` would take care of a text-based display of all objects of type Planet, but this was later removed from the end version. The method's function was passed on to another method with the same name in the class MainPage in Gui.py.

The user is allowed to set their main planet via the method `set_main_planet()`, which will fill the variable `index_of_main_planet` with the value that is the index of the selected planet in `planets_list`. This selection will be achieved via a user click in the GUI.

The method `next_turn()` is the ultimate driving force behind the simulation and will simulate the next turn, which calls all the methods to update or store their values.

Eventually, the method `check_winning_condition()` is called to check whether or not the user has achieved a total progression of 1000 or if the `remaining_turns` variable has reached 0. It will return either a success or failure message, respectively upon successfully or unsuccessfully completing the simulation, or return `None` and as such indicate to the programme that the user has not yet finished the simulation. It should continue executing the actual simulation in this instance.

2.1.2 Class Planet

The class Planet stores all basic information about a planet and therefore represents a planet as an object of this class. The class stores a constant `GOLDILOCK_DISTANCE`, which is set at 150,000,000 (km) and is defined as the centre of the Goldilocks' Zone.

A planet has six attributes: `planet_name`, `distance`, `radius`, `is_gz`, `atmosphere` and `landmass`. It receives its distance and its planet name from the class StarSystem. Its radius is determined via a formula (cf. *Chapter 4: Implementation*) that generates bigger radii if the planet is further away from its star, or in other words has a bigger distance value. If the planet's attribute `is_gz` is set to True, the atmosphere will be a randomly generated number between 80 and 100. If `is_gz` is False, the atmosphere will be a randomly generated number between 1 and 40. As such, planets that reside in the Goldilocks' Zone will have an initially better atmosphere than planets that are outside this zone.

The methods `calc_temperature()` and `calc_planet_quality()` determine a planet's temperature attribute as well as the overall planet quality. Planet quality is a score out of 100 that is given to a planet, based on its initial attributes. It provides the user with an easy, overall number that shows them how decent the planet is. This plays a part when the user selects their main planet.

The method `show_information()` displays the information of an individual planet object. This method is overwritten in the class MainPlanet.

2.1.3 Class MainPlanet

This class uses the copy-constructor to copy the attributes that belong to its parent class Planet. MainPlanet is, as such, a child class or sub class to Planet. Functionally, this class supplies an object of itself that stores all the information that its super class Planet already has, as well as information that belongs to the organism. This information encompasses the following organism attributes: `total_population`, `prev_population` (acquired through the method `cache_population()`, which stores the current `total_population` before StarSystem executes a `next_turn()` method), `population_health`, `tech_counter`, `tech_focus`, `tech_list`, `life_quality` and `progression`. These concepts are explained and defined in greater detail in the definition list for project-specific vocabulary, which can be found in the initial lists that precede this main body of text. The attribute `usable_landmass` is a planet attribute, but is specific to the class MainPlanet; in other words, only an object of type MainPlanet has this

attribute. This class also has a number of multipliers, namely `atmosphere_multiplier`, `landmass_multiplier` and `population_multiplier`. Multipliers are Project Stars way of altering planet and organism attributes when a disaster (cf. *section 2.1.4* EventManager) has occurred.

The opposite event to a disaster is a breakthrough. These provide the user with bonus points to their technologies. These technologies – of which there are four – are stored in a list `tech_list`. At the start of the simulation, the user is allowed to spend 12 points. The method `spend_points()` allows the user to spend these technology points and should only be called at the very beginning of the simulation. The user is not allowed to spend technology points in the fourth technology, engineering. The GUI will check for the values that are entered by the user and will provide them with an error message in case the total amount of points do not add up to 12. It will also check for negative numbers, and provide the user with an error message if they try to close the window without entering any numbers.

The user is, furthermore, allowed to set a research focus, done through the method `set_research_focus()` that requires the index of the technology in the tech_list that it should set its focus to. Setting its focus, in this context, translates to a mechanic that allows the user to passively gain one point in a technology of preference, every time five turns have gone by. This turn value is stored in `tech_counter` and is checked in the method `update_research_focus()`, which subtracts 1 from `tech_counter` every time a turn passes. The `tech_counter` is initially 5 and when it reaches 1, it should call the method `update_technologies()`, add one to the appropriate technology, and reset the `tech_counter` value to 5 so it can start all over again. The user is allowed to switch research focus, but will lose their progress and as such reset the value to 5 upon every focus switch.

The method `change_base_values()` is called in StarSystem during every turn and checks the type of event (cf. EventManager); disaster or breakthrough. It will update the multipliers if the event is a disaster and it will do so with the method `update_multipliers`, which is provided a parameters that gets the multipliers from class EventManager (`get_multipliers`). In the scenario of a breakthrough, the method `change_base_values()` will instead call the method `update_technologies()` and, similar to the passive gain of one technology point via `set_research_focus()`, add one point to the technology that is specified in the breakthrough event.

The method `update_variables()` is a method that updates all variables separately through its correct methods. These updates include: atmosphere (via the method `calc_atmosphere()`), temperature (via the method `calc_temperature()`), usable landmass (via the method `calc_usable_landmass()`), population health (regeneration happens per formula via the technology medicine), life quality (via the method `calc_life_quality()`), total population (via the method `calc_total_population()`) and progression (via the method `calc_progression()`).

In a final instance, the method `show_information()` from Planet is overwritten by the method `show_information()` in MainPlanet. This method prints the information about the planet and organism, i.e. their attributes, in the GUI.

2.1.4 Class EventManager

An event is Project Stars' way of simulating the randomness of real life by adding some exciting and some unpredictability to the simulation. An object of type EventManager stores its type of event, disaster or breakthrough, in `type_event`. All disasters and breakthroughs are read into the system from a CSV file, and are stored in respectively a `disaster_list` or `breakthrough_list`.

This CSV file has the following information which is in turn stored in the appropriate variables of the class: the type of event (`type_event`), progression start and end (to be checked in the method `generate_event()`), the sound file name (`sound_file_name`), an event title (`title`) and an event message (`message`). These two attributes are the same for both events. A disaster, however, has four multipliers included in the CSV file, whereas a breakthrough has a technology index instead of these multipliers.

These multipliers are stored in `atmosphere_multiplier`, `landmass_multiplier`, `population_health_multiplier` and `population_multiplier`. The technology index is stored in the variable `tech_index` and is a number between 0 and 3, both included. This number indicates the index of the technology in `tech_list` in the class MainPlanet that is to be updated with one point in the method `update_technologies()` in the class MainPlanet.

The complete format for events in the CSV file, is as followed: `type_event;start;end;sound_file_name;“title”;...`

- Disaster:
  - `type_event` = 0
  - `0;start;end;"sound_file_name";"title";"message";atmosphere_multiplier;landmass_multiplier;population_health_multiplier;population_multiplier`
- Breakthrough:
  - `type_event` = 1
  - `1;start;end;"sound_file_name";"title";"message";tech_index`

The class itself will generate an event when an object of its class is made. This happens via the method `generate_event()`, which takes one parameter: progression. As mentioned before, the CSV file has a progression start and end, which are two numbers between 0 and 1,000 (borders included) that indicate when the event can occur. The method `generate_event()` is what

checks and sets all of the above mentioned attributes of class EventManager, depending on the type of event.

2.2 UML: Sequence diagram

The sequence diagram (Figure 3) visualises in which ways objects, applications and methods interact with each other. This diagram describes how Project Stars first creates a StarSystem object per application request (start), which in turns creates a list of planets (of type Planet). Next, the application asks each Planet object for their planet information (attributes), which the Planet returns as a list. Upon doing so, the application asks the StarSystem to create an object of type MainPlanet, which occurs. All Main Planet attributes are calculated, and the StarSystem deletes the old Planet object and replaces it with an object of type MainPlanet (at the same index, as per front-end user choice).

The bread and butter of the application is the method `next_turn()` in StarSystem, called by the application itself. This method is repeated whilst the winning condition is false or whilst the user has not yet lost. StarSystem responds by first storing its current total population via the method `cache_population()` in MainPlanet, which it asks the object of type MainPlanet to do. This object replies by returning the total population value to StarSystem.

If conditions are met, i.e. the chance for an event to occur, StarSystem creates an object of type EventManager and uses it to call the method `generate_event()`, passing on the `progression` of the object of type MainPlanet as a parameter of this method. It returns an event (for the GUI's ease) which prompts StarSystem's method `change_base_values()` to respond and be executed. The returned event is passed an a parameter to this method.

The event can either be a disaster or a breakthrough. Each has their own path and methods involved.

- ➔ A disaster event will call the method `get_multipliers()` on its object event in MainPlanet and this method will respond by returning them. Next, the multipliers are updated through the method `update_multipliers()`.
- ➔ A breakthrough event will get the `tech_index` involved and return this to the object event of type EventManager in MainPlanet. Next, the technology is updated through the method `update_tech()` which has one parameter, namely the `tech_index`.
- ➔ If no event occurs, these steps are all skipped. Hence, they are marked as optional in the diagram.

Lastly, the method `update_variables()` is called by StarSystem on the object of type MainPlanet, requesting the object to recalculate and update all necessary variables. Eventually, all information is returned to the application to be handled by the GUI classes in Gui.py and as such be displayed correctly to the user. The end of the application depends on when the GUI closes.

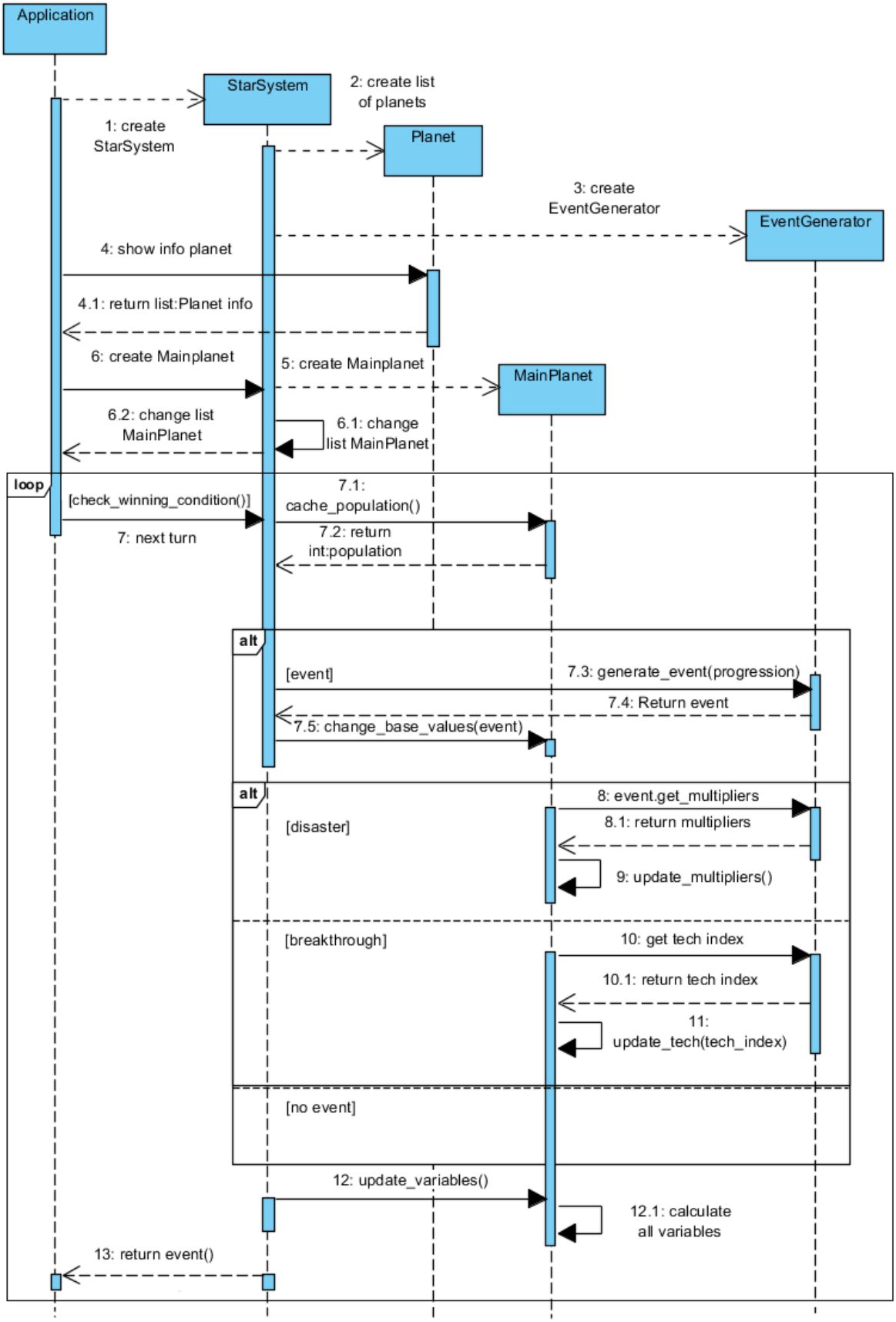

Figure 3. Sequence Diagram for Project Stars. The steps taken when the user is shown planet information, choses a Main Planet and progresses through the simulation via the next_turn() mechanic.

2.3 UML: Activity diagram

The next diagram of choice to better visualise Project Stars is the activity diagram, which can be found on the next page in Figure 4. This diagram depicts high-level processes, including flow or logic within a system [17], in this case the simulation.

The diagram starts at the initialisation of the simulation and walks the reader through the various procedures that are performed or executed in order to set up the simulation.

This includes:

- the generations of planets;
- the choice of a main planet from said planets;
- the ability to show information about the planet in order to support to user in their choice of main planet;
- spending twelve points;
- an iterative that simulates one-hundred turns (`next_turn()`);
- the user's choice to either change or keep their research focus – they must select a research focus in turn 1;
- a simulation finalisation after 100 turns have gone by, or if the `total_population` attribute to the MainPlanet object detects a value lower than or equal to 1. In these scenario's, the user has been unsuccessful in completing the simulation. On the other hand, the user has successfully completed the simulation, if the Star System's `check_winning_condition()` detects a progression of 1,000 in one of its turns.

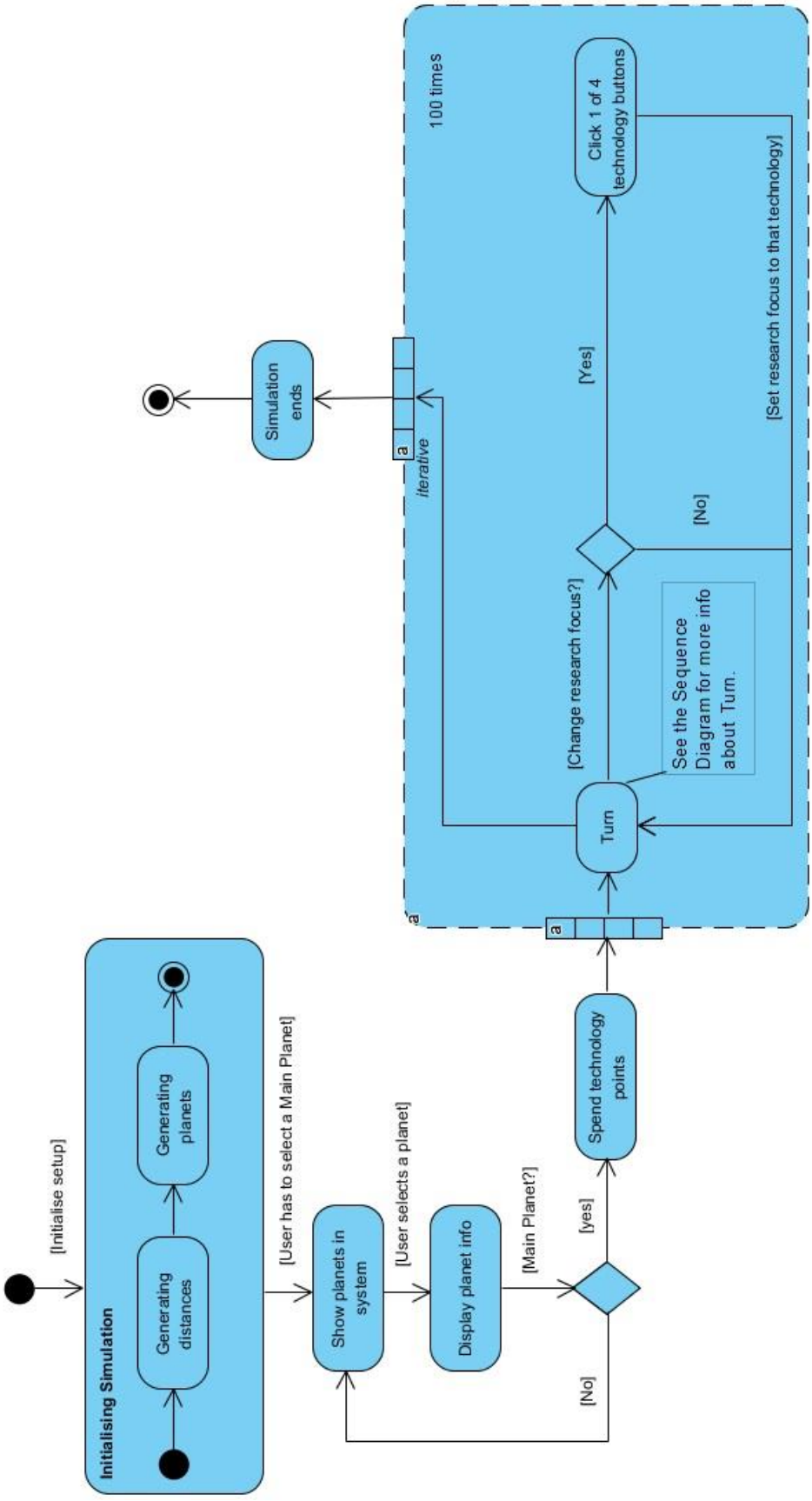

Figure 4. Activity Diagram for Project Stars. The diagram shows the steps a user takes through the simulation.

2.4 UML: Use-case diagram

The third and final diagram to visualise Project Stars is the use-case diagram (Figure 5). The main purpose of a use-case diagram is to provide the glue that keeps the requirements model together. They are developed from the point of view of the user and are therefore not very technical. This use-case diagram describes the interaction between the single user and the simulation.

A user (technical term: actor) can:

- ✓ look at the data of planets;
- ✓ choose (select) a main planet;
- ✓ invest (spend) points in technologies;
- ✓ set a research focus of a technology;
- ✓ look at the data of their organism and main planet;
- ✓ increase the turn.

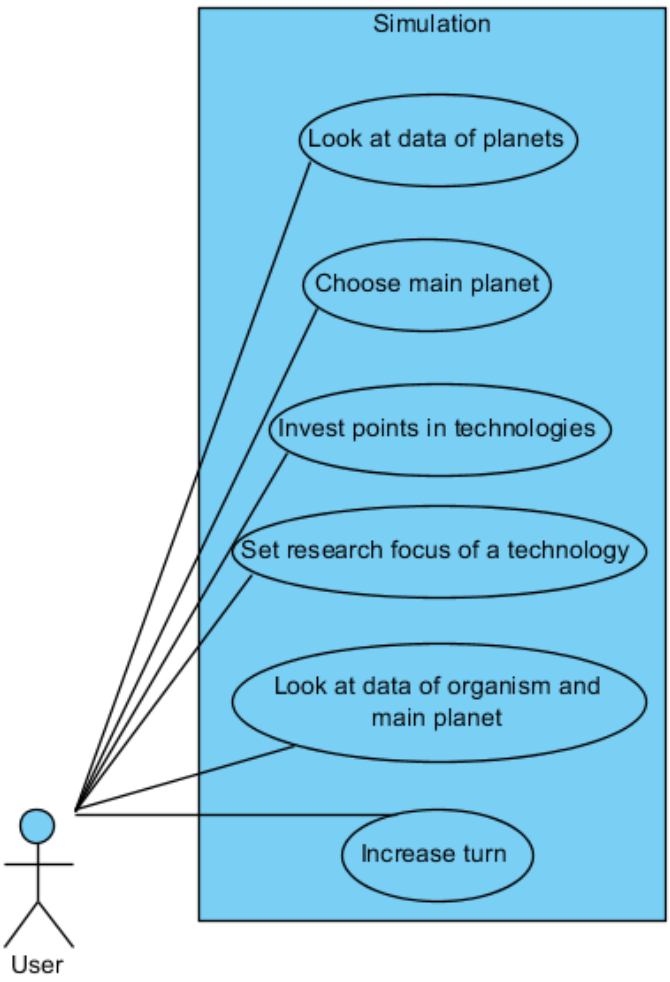

Figure 5. Use-case Diagram for Project Stars. The diagram shows the possible interactions between user and application (simulation

3. Functionality and simulation mechanics

This chapter will go into the functional detail of every aspect and mechanic of the simulation. This includes the star system, the planets and their attributes, the organism and their attributes, events and the GUI. This chapter includes many references to *Chapter 4: Implementation*, in an attempt to link variable and method choice as explained in *Chapter 2: Analysis* to the actual explanation and functionality behind them (this chapter), combined with Chapter 4's formula analysis.

While a textual version of the connections between planet attributes, organism attributes and technologies is provided within this chapter, this chapter also contains a complete, visual overview of these relations with legend. This overview can be found in *3.4: Visual overview of attributes and technologies.*

3.1 Star System

A star system is the environment in which the simulation takes place. It contains a single, near-death star and a number of planets between five and seven. These planets are generated on so-called planet rings. A planet ring is a possible location for a planet around its star. Project Stars has a set total of eleven rings that it chooses at random distances from the star, but allows for expandability for even more rings. The user is guaranteed three rings (or possible planet locations) in the so-called Goldilocks' Zone, of which one to three will be active.

The star system can be divided into three main zones: the near-star zone, the Goldilocks' Zone or GZ, and the distant-star zone. The Goldilocks' Zone is the most optimised region for a planet to live at. It is a zone at a set distance from its star that has optimised planet attributes for an organism's survival, thus providing the user with a higher chance to successfully complete the simulation.

Project Stars' GZ starts at 135 million km and ends at 180 million km. The predefined centre of the GZ is set at 150 million km, which is the real-life definition of one AU or Astronomical Unit. The near-star zone (pre-GZ) spans from 10% to 90% of the GZ centre – in other words 15 million km through 135 million km. The distant-star zone (post-GZ) spans from 120% to 240% or the GZ centre – in other words 180 million km through 360 million km.

The star system is acted upon by the operator turn. The turn mechanic is a key functionality that allows the user to progress throughout the timeline, with each turn being ten years out of a total of 1000 years. The simulation ends when 100 turns have been completed.

3.2 Planets

Initial planet attributes are randomly generated, whilst still taking some predefined limitations into consideration. A planet has four initial characteristics that define what it is like: its (1) planet name, (2) distance, (3) atmosphere and its (4) landmass. A planet also has two calculated attributes that are determined through three out of four of its initial attributes (planet name is merely an aesthetic function). These two calculated attributes are (5) the (average surface) temperature and the (6) the planet quality. Planet attributes are displayed in the information panel (cf. *section 3.6.2*) together with the organism attributes.

A planet has a seventh attribute, its radius, which is determined through the distance of the planet to its star, but it is only used in the GUI to draw the planets. It holds no value elsewhere in terms of calculations.

3.2.1 Determining the six planet attributes

Below is a list of the six planet attributes and in which way they are determined or calculated.

3.2.1.1 Planet name

A planet's name is randomly fetched from a CSV file that contains a wide array of available, existing planet names.

3.2.1.2 Distance

A planet's distance, from its star, is calculated in the star system and can range anywhere between 15 million km and 360 million km (cf. *section 4.2* to view the formulas that generate the distances).

3.2.1.3 Atmosphere

A planet's atmosphere is a randomly generated integer between 80 and 100 for planets in the GZ, and a randomly generated integer between 1 and 40 for planets outside of the GZ, i.e. the near-star zone or distant-star zone.

3.2.1.4 Landmass

A planet's landmass is a randomly generated integer between 10 and 100 that determines the total amount of landmass that is available to a planet.

3.2.1.5 (Average surface) Temperature

A planet's average surface temperature is calculated through its distance and atmosphere. Planets that are further away from their star are generally colder and vice versa. A planet that is generated inside the GZ has a temperature ranging from -25 °C to 50 °C, whereas a planet outside this zone has a temperature of -250 °C to 500 °C (cf. *section 4.3.2* to view the formulas that calculate the temperature).

3.2.1.6 Planet Quality

Planet quality is Project Stars' way of showing the user an average score between 0 and 100 and covers all of the above planet attributes. It is initially presented to the user and indicates how well-suited a planet is to be inhabited by an organism, in an effort to guide them in their selection of a main planet (cf. *section 4.3.2.2* to view the formulas that calculate the planet quality).

3.3 The Main Planet

A main planet is the user's planet of choice to harbour life for its organism. It keeps all planet attributes it had prior to being selected as a main planet, but gains one planet attribute and a number of extra attributes that belong to the organism that lives on it. These organism attributes are: (1) the total population of the organism, (2) four technologies, (3) population health, (4) life quality, (5) a research focus and (6) progression. The planet attribute that is gained, is (1) usable landmass and is unique to the main planet.

3.3.1 Determining the additional planet attribute

3.3.1.1 Usable landmass

A main planet's usable landmass is the planet's landmass that is fit for building and farming. It therefore depends on agriculture and architecture and is calculated through these two technologies along with the total landmass (cf. *section 3.2.1.4*) a planet has (cf. *section 4.4.1.2* to view the formula that calculates the usable landmass).

3.3.2 Determining the six organism attributes

Below is a list of the six organism attributes and in which way they are determined or calculated.

3.3.2.1 Total population

The total population of the organism is set to 100,000 at the start of Project Stars' simulation, but can easily be altered. It is calculated via the life quality of the organism. The main planet also keeps track of the total population in the system's previous turn, i.e. the previous population, and uses

their difference to determine the approach to the calculation of the progression (cf. *section 4.4.1.2* to view the formulas that calculate the total population).

3.3.2.2 Technologies

An organism's technologies shields it from disaster and functionally influence a whole array of other attributes that belong to the organism. The four available technologies are (1) medicine, (2) agriculture, (3) architecture and (4) engineering. The user is allowed to spend a total of twelve points to their heart's content in the first three technologies. The higher a technology is, the more effective it will protect the organism from disaster and the more it will positively influence other organism attributes.

1) *Medicine* aids the organism in their combat against disasters such as diseases. Medicine directly influences the regeneration of the population health of an organism and directly influences the life quality of said organism.
2) *Agriculture* aids the organism in their combat against disasters such as diseases and famine. Agriculture indirectly influences the life quality of an organism.
3) *Architecture* aids the organism in their combat against disasters such as natural disasters. Architecture directly influences the life quality of an organism and its progression.
4) *Engineering* aids the organism in their combat against disasters. Engineering directly influences the life quality of an organism and is a major component of the progression mechanic.

Technologies have a built-in cap, which means they cannot increase indefinitely. The first three technologies have a cap of 15, whereas Engineering has a cap of 30 (cf. *section 4.4.1.1* to view the formulas that update the technologies). Functionally, engineering is allowed a higher cap, because it has a significant weight factor in the calculation of the progression. Overall, this higher cap has a positive effect on the progression mechanic.

3.3.2.3 Population health

The population health (0-100) of an organism is an indicator for its well-being and its regeneration is dependent on the technology level in medicine. It is initially set to 100 and is possibly negatively affected by the health multiplier as a result of disasters (cf. *section 4.4.1.1*). Population health is visually represented by a keyword that is determined as followed: 100-70: Healthy, 70-40: Average health, <40: Bad health.

3.3.2.4 Life Quality

The life quality is the combination of the landmass, usable landmass, average surface temperature, population health and the technology level in architecture and engineering. Its functionality can be viewed as similar to planet quality: a simple percentage that indicates the how good the overall state

of the organism is (cf. *section 4.4.1.2* to view the formulas that calculate the life quality). This life quality is a factor that is used to calculate the total population for the organism.

3.3.2.5 Research Focus

Research focus or technology focus is the organism's way to passively gain a point in a particular technology every five turns. The user has a choice to alter the organism's technology focus every turn, but that will reset and nullify the previous turns that had been spent towards a technology point (cf. *section 4.4.1.1* to view the method that determines the research focus).

3.3.2.6 Progression

The progression mechanic is what indicates the organism's level of sophistication on a scale of 0 to 1000 and is what ultimately leads to the organism's escape from its home planet -- in other words the successful finalisation of the simulation. Progression is visualised as a progression bar that tells the user what the total progression is and how much progression was gained or lost upon ending the previous turn.

If the population is not on the decline, the progression is dependent on the technology level in medicine, architecture, engineering, life quality and the total population. However, in a scenario where population has decreased in relation to the previous turn, progression is no longer dependent on the before-mentioned technologies, but solely dependent on life quality and the difference in population between this turn and the previous (cf. *section 4.4.1.2* to view the formulas that calculate the progression). By removing the influence of technologies in this scenario, the difficulty curve slightly raises and forces the user to make choices that improve said life quality and population.

3.4 Visual overview of attributes and technologies

The figure on the next page (Figure 5) shows an overview of all the planet and organism attributes, and their intricate relationships to each other. Planet attributes are listed on the left-hand side, technologies are listed in the middle and organism attributes are listed on the right-hand side.

Each rectangle, as indicated in the legend, has two bubbles: one green and one red. A green bubble is the receiving end of the rectangle and a red bubble is the transmitting end of the rectangle. There are two kinds of arrows: (1) a purple arrow between two rectangles indicates that the transmitting rectangle affects the receiving rectangle; (2) an orange arrow between two rectangles indicates that the transmitting rectangle affects the multiplier that belongs to the receiving rectangle.

The image also contains two portals: one blue and one orange. They are solely used to bridge the gap from the left side to right side and thus prevent the intermingling of too many arrows. The blue portal is a starting point, whereas the orange portal is an ending point – as indicated by the legend and the direction of the arrows going through or coming out of the portals.

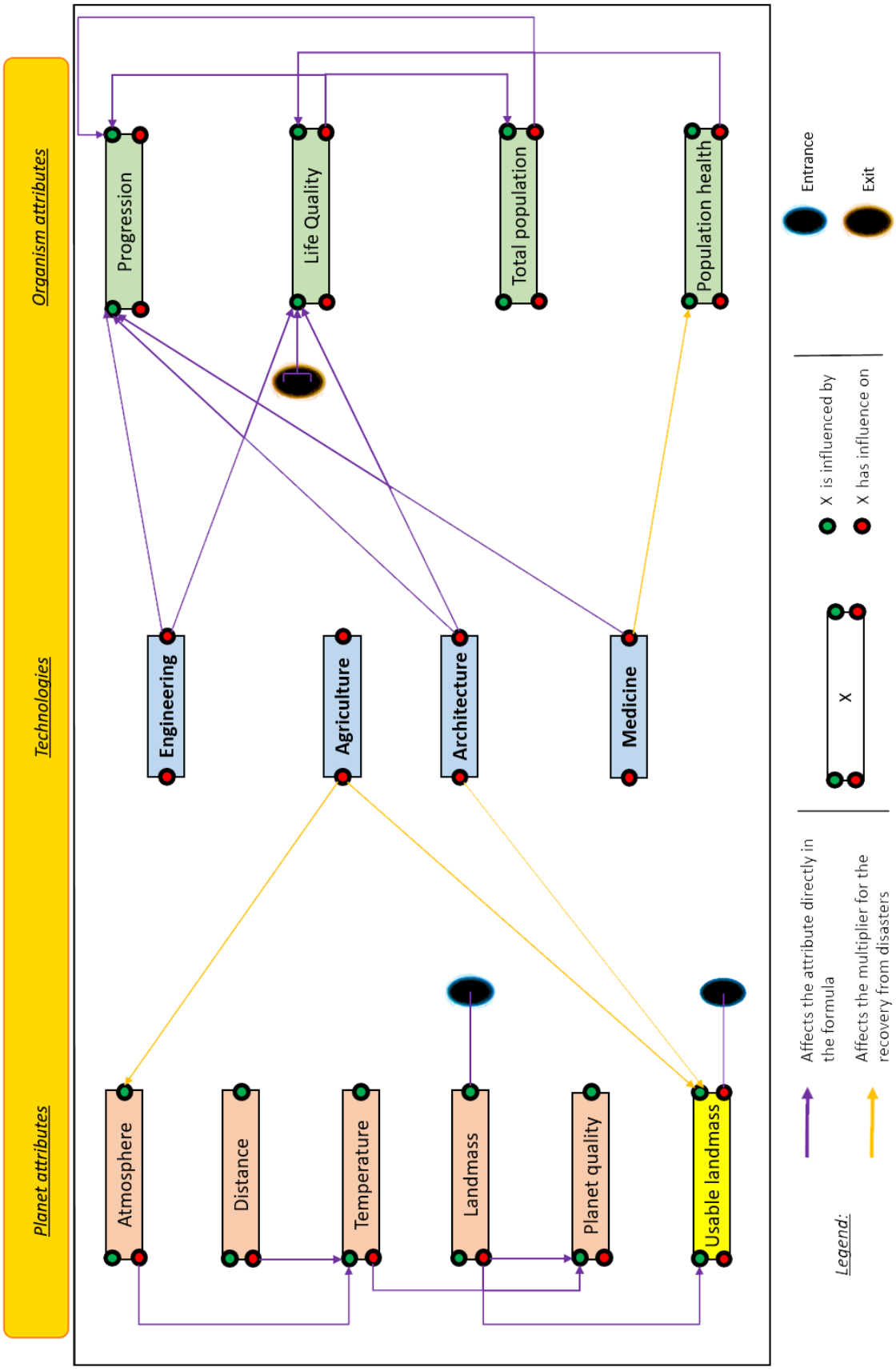

Figure 6: Attribute and Technology Relation Diagram. A full overview of planet and organism attributes, technologies and their mutual relationships.

3.5 Events

An event is Project Stars' approach to simulate reality by adding the randomness and unpredictability of daily life. Events can occur as being beneficial or harmful to the planet and organism, the former being a breakthrough and latter being a disaster.

Disasters are based on a multiplier-effect approach, which means that they change the current value of an attribute by multiplying it with the multiplier. These multiplier will always be less than 1 but higher than 0, and as such effect it negatively. Breakthroughs, on the other hands, are not based on a multiplier-effect approach and as such they will always add one point to one of your technologies. The technology that is increased is determined by the breakthrough. A full list of all disasters and all breakthroughs with their corresponding effects can be found in *Appendix B: List of Events*.

3.6 Visualising Project Stars (GUI)

The GUI is Project Stars' window to the user and is constructed independently, which means it implements classes from the provided code. The module that was used to construct the window, is the *module* Tkinter [16] and is provided by Python by default.

3.6.1 Starting screen

Figure 7 shows the layout of the *Starting screen*, which consists of a few buttons, an image, and a menu bar. The menu bar contains two dropdown-menus: *File* and *Options*. *File* allows the user in to save a simulation [12], load a previously saved simulation, or exit the application.

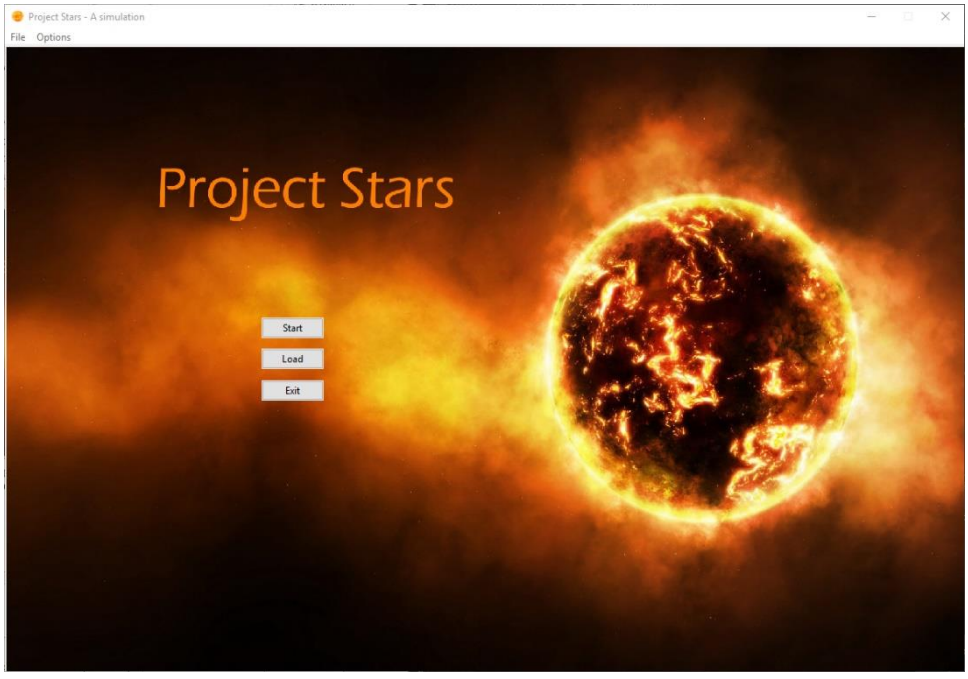

Figure 7. Project Stars' Starting Screen. The screen presented to the user upon executing the application.

Exiting is also possible by clicking the exit cross at the top right of the screen. *Options* allows the user to mute sounds and music. These two options can be ticked or unticked and are, as such, by definition not mutually exclusive.

3.6.2 Application window

The application window, as shown in Figure 8, is one big *container* on the surface or first level. This container is separated into two frames (left and right) that contain one or more frames with multiple *widgets* on lower levels. The left frame remains one single frame with widgets, but the rightmost frame is separated into three separate frames, each with their own widgets. Throughout the application, the *grid-layout* is the only layout used.

3.6.2.1 Left frame

The left frame contains three widgets. The *canvas* (the large star system), the progression label and the progress bar. The canvas widget is used to draw moving objects (here: planets), whereas the progression label and progress bar represent how advanced the species are (cf. *section 3.3.2.6*). Notice how progression has a number between bracket. This is referred to as the rate of change and is the amount of progression that was gained or lost between the previous and current turn.

3.6.2.2 Right frame

The right frame is divided into a top, centre and bottom frame. Firstly, the upper frame consists of four buttons, in which the user can set their research focus. Next, the middle frame initially provides the user with instructions to the simulation, and later visualises the events that occur throughout the simulation. This includes a short text describing the event, and, if the event is a breakthrough, the technology that has gained one point.

Lastly, the bottom frame initially supplies the user with information about the selected planet and later on shows a detailed description of planet and organism (cf. *sections 3.2.1, 3.3.1* and *3.3.2* on planet and organism attributes). These, too, use the rate of change mechanic and, as was previously mentioned, provide the user a visual way to see the attribute's change between the previous and current turn.

3.6.3 End screen

Finally, the user is provided with an end screen (Figure 9) upon successful completion of the simulation. This shows a congratulation message and their final total population. The final total population and cause of death is shown, if the user was unsuccessful in completing the simulation.

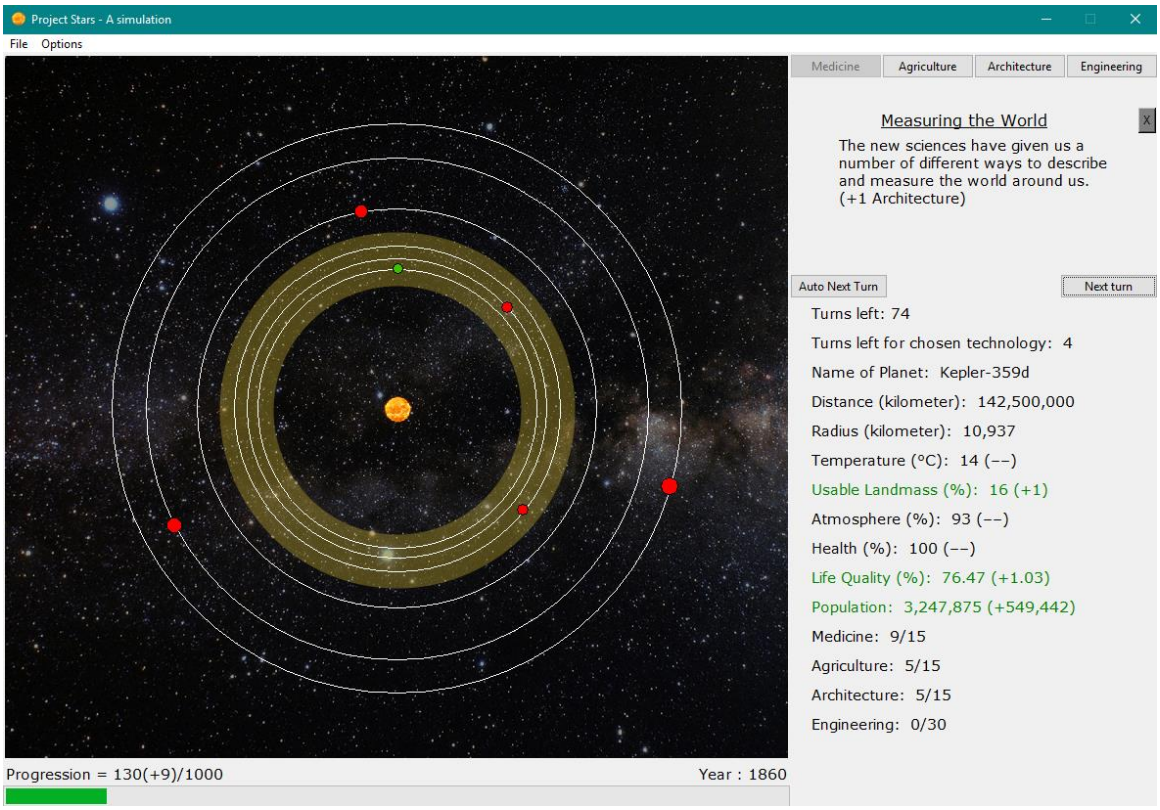

Figure 8. The Application's Main Page. The Main Page presents the user with all useful information on its Main Planet and organism, as well as a visual representation of the Star System.

Figure 9. The Application's End Page. This is the final window and shows whether the stimulation has been successful or not.

4. Implementation

In *Chapter 2: Analysis*, a close look at the tools, methods, variables and techniques provided a clear-cut image of what is to be expected of Project Stars [5]. *Chapter 3: Functionality and simulation mechanics* offered the reader an intricate look at the connections between and inner-relations of all planet and organism attributes, as well as the technologies and their workings, and the Event mechanic. Generally speaking, *Chapter 3: Functionality and simulation mechanics* described the functionality of all the before-mentioned aspects of Project Stars on a surface level design. This indicates that there is also a level of design that is focused on the actual implementation of this functionality and its design. Chapter 4: Implementation provides this, as such.

This chapter features many aspects and details that have already been mentioned in *Chapter 2: Analysis*, and should, therefore, be viewed as an in-depth expansion in terms of explaining code, side by side with the formulas and calculations behind the mechanics mentioned in *Chapter 3: Functionality and simulation mechanics*.

However, this chapter does not explain code line by line, but rather, explains large chunks of code and either workings, or, if needed, cuts these chunks up into smaller bits first.

4.1 Project Stars' Prototype

An initial design for Project Stars featured a text-based way of interacting with the user. A broad implementation of the classes StarSystem, Planet, MainPlanet and EventManager allowed for a test phase run of some of the formulas that had been designed in theory. These formulas had been subject to change throughout the entire development phase of this project, featuring minor tweaks, nerfs or buffs to many simulation mechanics. The prototype allowed for debugging and helped solve many issues that were initially obscure.

It was important that the values for planet and organism attributes that the user would see, actually made sense. Although the team was quite successful in developing formulas that would fit their needs – and as such, closely resembling real life scenarios – some simplifications took place. One example is the planet attribute temperature. Its formula scales with distance and appears quite accurate, though it is assumed that the planet does not have a day and night cycle. The temperature is a static value and does not change, unless it is subject to outside change through events. This means that real life scenarios, such as a planet that is close to its star (and therefore boiling hot) but has its temperature drop below 0 °C at night (e.g. Venus), are out of the equation. Venus also has a higher average temperature than Mercury despite being further away from the Sun, due to the greenhouse effect on Venus. Project Stars' atmosphere does not account for different molecules and merely offers the user with a percentage value of how liveable it allows the planet to be.

Despite these simplifications, Project Stars has done a good job at simulating change through controlled (but still random) nature, using some 102 different kinds of events.

4.2 Star System

As mentioned in Chapter 3, Star System is the environment in which the simulation takes place. It contains a number of planets and a star and is represented by the appropriately named class StarSystem.

The class' main function is to initialise said environment, take care of the turns that allow the user to progress through the simulation and keep track of the (Main) Planet object(s) and its attributes. The class creates 11 rings at randomly generated distances, with each ring being a possible location for a planet to reside at. Furthermore, it initialises the value for the Goldilocks' Zone' centre at 150 million km (`GOLDILOCK_DISTANCE`) and limits the amount of planets that are created at the beginning of the simulation to a number between 5 (`MIN_PLANETS`) and 7 (`MAX_PLANETS`).

Each one of these planets will reside at one and only one of the before-mentioned rings. Considering that there are more rings than initial Planet objects, rings that are not filled with a Planet object will no longer partake in the simulation. Each ring is defined at a set distance from its star and the class StarSystem keeps track of all these distances using a list. To ensure that the user will have at least one Planet object in the GZ to choose from, the class guarantees the generation of three rings that reside in the GZ and keeps track of these in `gz_distance_list`. Theoretically, the possibility to have all three of these rings still remaining upon the start-up of the simulation (after generation has taken place) is low, but does exist.

Every class features an `__init__()` which initialises an object of this class and its variables, as well as runs developer-chosen methods that are required upon the start-up of the class. In this case, StarSystem initialises: `remaining_turns` to 100, an empty list to store all planets (`planets_list`), an empty list to store all planet distances (`distance_list`), the previously-mentioned list to store the GZ distances (`gz_distance_list`), the index of the main planet (`index_of_main_planet`), an object of type EventManager (`event`) with a parameter of type string ("Events.csv") and lastly, the `set_up()` method.

Firstly, this method opens the CSV file that contains all planet names and stores them in a local variable `planet_names`. It then commences the generation of planet distances using a for-loop for each zone (3) in the star system: a zone that precedes the GZ, the GZ itself and a zone that succeeds the GZ. These zones, respectively, contain 2, 3 and 6 rings, which are stored in `PREGZ_RINGS`, `INGZ_RINGS` and `POSTGZ_RINGS`.

As mentioned in *section 3.1*, these zones have set ranges:

- pre-GZ: [0.1, 0.9[multiplied by the `GOLDILOCK_DISTANCE`;
- GZ: [0.9, 1.2] multiplied by the `GOLDILOCK_DISTANCE`;
- post-GZ:]1.2, 2.4] multiplied by the `GOLDILOCK_DISTANCE`.

The following formulas (1) through (3) calculate the distances that are to be stored in the `distance_list` or `gz_distance_list` depending on their nature.

$$PREGZ\_DIS = \left(iter \cdot \left(\frac{0.8}{PREGZ\_RINGS}\right) + \left(\frac{fitter}{100}\right)\right) \cdot GOLDILOCK\_DISTANCE \quad \textbf{(1)}$$

$$GZ\_DIS = \left(iter \cdot \left(\frac{0.3}{INGZ\_RINGS}\right) + \left(\frac{fitter}{100}\right)\right) \cdot GOLDILOCK\_DISTANCE \quad \textbf{(2)}$$

$$POSTGZ\_DIS = \left(iter \cdot \left(\frac{1.2}{POSTGZ\_RINGS}\right) + \left(\frac{fitter}{100}\right)\right) \cdot GOLDILOCK\_DISTANCE \quad \textbf{(3)}$$

These three formulas are very similar in their function, the only differences being the values for two variables (`fitter` and `iter`[2]) and a constant value (0.8, 0.3 or 1.2). Therefore, the randomness of these distances is found in these variables and constant, and as such allow the complete formula's limits to fit to the before-mentioned zone ranges. All three formulas have the same structure: two terms that are summed and then multiplied by the constant `GOLDILOCK_DISTANCE`. As mentioned in *section 2.1.1*, some rings will remain empty and will henceforth no longer take part in the project.

The first term contains the `iter` (used as for-loop variable, i.e. iterator) has a range from 0 to `PREGZ_RINGS`, `INGZ_RINGS` or `POSTGZ_RINGS` (depending on the formula). The `iter` is multiplied with what could be seen as a constant that contains `PREGZ_RINGS`, `INGZ_RINGS` or `POSTGZ_RINGS`. This is done to create subzones with equal size that will contain one ring each.

The second term contains the `fitter`, which is a randomly generated number with a range from:

- 10 *to* $10 + \frac{70}{PREGZ\_RINGS}$;
- 90 *to* $90 + \frac{20}{INGZ\_RINGS}$;
- 120 *to* $120 + \frac{100}{POSRGZ\_RINGS}$.

These are the ranges per zone, for each zone respectively.

The frequent use of `PREGZ_RINGS`, `INGZ_RINGS` or `POSTGZ_RINGS` (depending on the formula) is justified by the idea of creating a formula that easily adapts to a higher number of rings in a zone.

Next, another for-loop uses a local variable `temp_name` to store a randomly selected planet name from the list `planet_names` and a randomly selected index (`distance_number`) from the `distance_list`. Using `temp_name` and `distance_number`, a Planet object is created and stored in the `planets_list`. Lastly, the selected distance is removed from the list to avoid it

[2] Iter is short for iterator.

being selected again. A second for-loop completes an identical process, but using the distances stored in `gz_distance_list`.

The class StarSystem concludes its function with three more methods: `set_main_planet()`, `next_turn()` and `check_winning_condition()`. These will be now be examined closely.

The first of these methods has one parameter of type Planet. The class MainPage of Gui.py calls this method in its own `set_main_planet()` method and provides it with this parameter of type Planet – the partim about the GUI (cf. *section 4.6*) will examine how this providing is done. The planet's index is searched for in `planets_list` and set as the `index_of_main_planet`. Next, the Planet object at that particular index is overwritten by an object of class MainPlanet. The copy-constructor of this class is called to designate the Planet object's attributes to the object of class MainPlanet.

The second of these methods, `next_turn()`, simulates the next turn in the simulation. It first executes the `cache_population()` on an object of type MainPlanet to store the turn's current population. Then, a random number is generated from 0 to 3 (3 not included) to determine whether or not an event will take place. If the chances, which are 1/3, are right, the method calls for the methods of class MainPlanet, namely `calc_progression()`, `change_base_values()`, to update their values. Next, the method `update_variables()` is called on the object of class MainPlanet. A more in depth look at what this method does, is provided in *section 4.4* of this chapter. Finally, the remaining turns (`remaining_turns`) is updated – lowered by one each time `next_turn()` is executed. The method `next_turn()` will return either `None` or an object of class EventManager, depending on whether or not the 1/3 chance of an `Event` was met. The method is called in the method `next_turn()`in the class MainPage of Gui.py to pass on this `None` or `EventManager`.

The third and last of these methods, `check_winning_condition()`, is called in the class MainPage of Gui.py on an object of class StarSystem to check the return value. This method returns 0, 1 or 2, respectively, when Progression is equal to or higher than 1000, if the remaining turns are 0 or if the total population has hit 1 or 0.

4.3 Planet

An object of class Planet stores all information about a planet and therefore represents one. The class Planet is a parent class to the class MainPlanet. Planets are generated at the very beginning of every simulation, as explained in *section 4.2* of this chapter.

A planet has one constant, namely GOLDILOCK_DISTANCE, which is set to 150,000,000. This is the GZ's centre.

4.3.1 Setting or calculating planet attributes

Upon initialisation of an object through the method `__init()__`, a planet has its planet attributes set or calculated. These include `planet_name`, `distance`, `radius`, `is_gz`, `atmosphere`, and `landmass`. Firstly, a planet object receives its `planet_name`, `distance` and `is_gz` values from StarSystem through its parameters in said initialisation phase.

Secondly, the `radius` of a planet is calculated through the `distance` via the following formula (Formula 4):

$$radius = \left(5 + \frac{distance \cdot 15}{360{,}000{,}000}\right) \cdot 1{,}000 \quad \textbf{(4)}$$

This formula allows the radius to be directly proportional to the distance, which means that smaller planets reside closer to their star and bigger planets further from their star. The distance is multiplied by 15 to scale it upwards and then divided by 360,000,000. The latter value is the maximum distance a planet (ring) can occur at. This total value is added to 5 and multiplied by 1,000 (for a more realistic value). As such, the lower limit (where `distance` is theoretically 0) is set to 5,000 and the upper limit (where `distance` is theoretically 360,000,000) is set to 20,000. Before this calculated value is assigned to the variable `radius`, it is cast to an integer, allowing it to be ready for display in the GUI.

Next, the `atmosphere` will be a randomly generated integer depending on the `Boolean` value of `is_gz.` This random value will either be between 80 and 100 if `is_gz` is `True`, or, between 1 and 40 if `is_gz` is `False`.

Finally, a planet's `landmass` will always be a randomly generated integer between 10 and 100.

4.3.2 Planet methods

A planet has three more methods: (1) `calc_temperature()`, (2) `planet_quality()` and (3) `show_information()`.

4.3.2.1 calc_temperature()

The method `calc_temperature()` calculates the temperature of a planet depending on the distance and atmosphere through one of two formulas. The reader is reminded that the Star System is divided into three zones: a zone preceding the GZ, the GZ itself and a zone succeeding the GZ. This is important, because the temperature is significantly different for each one of these zones. Two formulas were designed through trial and error.

The first formula (Formula 5) calculates the temperature for the GZ planets[3].

$$temperature = 250 - \left(\frac{distance}{600{,}000}\right) \cdot \left(\frac{100}{atmosphere}\right) \quad \textbf{(5)}$$

A constant first term is lowered by a second term that is scaled by distance and atmosphere. Four limits can be identified:

- distance and atmosphere are both maxima: a temperature of -50 °C is returned;
- distance and atmosphere are both minima: -31.25 °C;
- distance is maximal and atmosphere is minimal: - 125 °C;
- distance is minimal and atmosphere is maximal: 25 °C.

These temperature values confirm that the formula accounts for more distant planets and planets with lesser quality atmospheres as being colder, and vice versa.

The second formula (Formula 6) calculates the temperature for a planet that is not in the GZ. The variable `const` is equal to `-1.0` if a planet precedes the GZ, or is equal to `0.5` if a planet succeeds the GZ. There is a valid reason for this distinction: theoretically, there is no upper limit for temperature. A lower limit does exist, however. This temperature is approximately -273 °C (absolute zero). Setting `const` to `-1.0` for planets that precede the GZ, allows their temperature to increase much more significantly, because the distances here are relatively low. A value `0.5` for `const` dictates that the temperature decreases more slowly for more distant planets, whilst still respecting the lower limit for the temperature. A planet that resides at the most distant ring, i.e. a distance of 360 million km, can therefore never have a temperature below absolute zero.[4]

$$temperature = 230 + \frac{3.5}{const} - const \cdot (-const * atmosphere + 300 + \frac{const \cdot distance}{1{,}000{,}000} \cdot 4) \quad \textbf{(6)}$$

4.3.2.2 planet_quality()

In *chapter 3: Functionality and simulation mechanics*, the planet quality was defined as being a way to show the user an average score between 0 and 100 and cover all planet attributes. The method `planet_quality()` calculates this overall score, using a formula for each of the three distinct scenarios.

Formula 7 is used to calculate the planet quality:

$$planet\_quality = \frac{1}{3} \cdot scaled\_temperature + 0.15 \cdot landmass + 0.35 \cdot atmosphere \quad \textbf{(7)}$$

[3] As a reminder, the minimum and maximum values for distance and atmosphere are, respectively, [135,180] million km and [80,100] %.

[4] As a reminder, the minimum and maximum values for distance and atmosphere for a planet preceding the GZ are, respectively, [15, 135[million km and [1,40] %. Analogously, a planet succeeding the GZ has distance and atmosphere values of, respectively,]180,360] million km and [1,40] %.

The local variable `scaled_temperature` signifies a temperature, but has three scenarios through which it is determined. The temperature itself is calculated through the method `calc_temperature()`, but is altered slightly, depending on the region in which the temperature resides.

Firstly, if the temperature is in the region]-25,50[°C, `scaled_temperature` is calculated via a parabola, Formula 8:

$$scaled\_temperature = -0.06 \cdot (temperature + 35) \cdot (temperature - 65) \quad \textbf{(8)}$$

Secondly, if the temperature is in the region [-273,-25] °C, `scaled_temperature` is calculated via an exponential, Formula 9:

$$scaled\_temperature = e^{\frac{temperature + 104.75}{20}} \quad \textbf{(9)}$$

Thirdly, if the temperature is 50 °C or higher, `scaled_temperature` is calculated via an exponential, Formula 10:

$$scaled\_temperature = e^{-\frac{temperature + 136.75}{20}} \quad \textbf{(10)}$$

As such, the linear combination between the `scaled_temperature`, `landmass` and `atmosphere` results in the planet quality, returned by the method `planet_quality()`.

4.3.2.3 show_information()

The method `show_information()` returns all the information of an individual planet object (name, distance, radius, landmass, atmosphere, temperature and planet quality). This method is overwritten in the class MainPlanet.

4.4 Main Planet

The class MainPlanet is a subclass to the class Planet. Firstly, the copy-constructor is used to set the `planet_name`, `landmass`, `distance`, `is_gz`, `atmosphere` and `radius`. The `total_population` is set to 100,000 by choice but can easily be changed. The previous population is saved in `prev_population` to calculate the rate of change, which is the loss or gain in population between two turns. This rate of change is simply the difference between the `total_population` and `prev_population`, which have the same value in turn 1 and therefore yield a difference of 0. The `total_population` of a turn is stored in `prev_population` via the method `cache_population()`, through simple assignment.

Furthermore, the `population_health` is set to 100 (a percentage) and the `tech_counter` to 5. This is the amount of turns that will have to pass before the user gains a technology point. The

`tech_focus` variable is initially `None`, as there is no technology focus just yet, but will change to an index of the array `tech_list` as per user-choice. This list contains the four technologies: medicine, agriculture, architecture and engineering. All multipliers, namely `atmosphere_multiplier`, `landmass_multiplier` and `population_multiplier` are set to value 1. The variable `progression`, which will be checked as winning condition, is set to 0. The class has two constant values, `TECH_CAP` and `ENGINEERING_CAP`, which define the highest possible value for the three basic technologies and the engineering technology, 15 and 30 respectively.

4.4.1 MainPlanet methods

A MainPlanet has 14 methods as shown in Table 1. This section will first take a closer look at the eight, basic methods. Each method that requires formulas to calculate a value will be discussed separately and are often more complex. They will be referred to as advanced methods. It should be noted that these advanced methods always take the current values in the current turn into account to calculate (and thus, return) the values for the next turn. In other words, these methods calculate the new values that belong to the next turn.

Table 1:
The 14 methods in the class MainPlanet

Basic methods	Advanced methods
`spend_points()`	`calc_atmosphere()`
`set_research_focus()`	`calc_temperature()`
`update_research_focus()`	`calc_usable_landmass()`
`change_base_values()`	`calc_life_quality()`
`update_multipliers()`	`calc_total_population()`
`update_technologies()`	`calc_progression()`
`update_variables()`	
`show_information()`	

4.4.1.1 Eight basic methods

The first eight methods are `spend_points()`, `set_research_focus()`, `update_research_focus()`, `change_base_values()`, `update_multipliers()`, `update_technologies()`, `update_variables()` and `show_information()`. These methods perform rather straight-forward tasks.

The method `spend_points()` has one parameter, the list `points`. Upon selecting a Main Planet in the GUI, the user is prompted with a window that asks them to spend 12 points on the three basic technologies. The entered values are stored in this list and assigned to the corresponding index in the list `tech_list` (index 0: medicine, index 1: agriculture, index 2:

architecture, index 3: engineering). As such, the object of type MainPlanet knows what the user's level in a particular technology is.

The method `set_research_focus()` has one parameter, `index`. This method is called in the method `research_focus()` in the class MainPage in Gui.py. This method will fetch the value for index from `btn_list` and pass it on to `set_research_focus()`. The latter method will check whether or not the `tech_focus` is equal to `index`. If this is not the case, the `tech_counter` will be reset to 5 and the `tech_focus` set to the value of `index`. As such, whenever the user decides to change their technology focus, the `tech_counter` is reset.

The method `update_research_focus()` checks whether or not the research is complete. If the `tech_counter` is not 1 and the `tech_focus` is set (not `None`), the `tech_counter` is reduced by 1. If `tech_counter` is 1, the `method update_technologies()` will be called to add a point to the technology that is selected for research. As such, the method `update_technologies()` will update the technologies.

The method `change_base_values()` has one parameter: `event`. This object of type EventManager will call one of two methods depending on the type of event (`type_event`). If the `type_event` is 0, this indicates that the event is a disaster; the method `update_multipliers()` is called. If the `type_event` is 1, this indicates that the event is a breakthrough; the method `update_technologies()` is called.

The method `update_multipliers()` will, as mentioned above, be called if a disaster occurs. It has one parameter, the list `multipliers`, which stores the four multipliers that are fetched from the CSV file (cf. *section 2.1.4*). It will multiply the each of the multipliers (`atmosphere_multiplier`, `landmass_multiplier`, `population_health` and `population_multiplier`) with the corresponding index in the list `multipliers` to calculate the new multipliers' values.

The method `update_variables()` will be called if attributes need to be updated. This method will, therefore, call all advanced methods (cf. *section 4.4.1.2*). Each variable (atmosphere, temperature, usable landmass and population health) will be changed separately, through its correct methods. Life quality, total population, progression and the research focus are also updated if the appropriate qualifications or needs are met.

Finally, the eighth basic method is `show_information()` also exists in MainPlanet's parent class (Planet). Planet's `show_information()` is overridden in this class to display additional attributes, specific to the Main Planet. These additional attributes are usable landmass, population health, life quality, total population, medicine, agriculture, architecture and engineering. Two attributes, specific to Planet only, are omitted: landmass and planet quality.

4.4.1.2 Six advanced methods

The six last methods are more advanced, because they use formulas that need a word of explanation to accompany them. These methods are: `calc_atmosphere()`, `calc_temperature()`, `calc_usable_landmass()`, `calc_life_quality()`, `calc_total_population()` and `calc_progression()`.

a) calc_atmosphere()

The method `calc_atmosphere()` calculates the atmosphere through its original value (`atmosphere`) and its multiplier (`atmosphere_multiplier`). The atmosphere is also dependent on the agriculture technology (`tech_list[1]`). The method `calc_atmosphere()` returns the new value for the atmosphere, but before this is calculated, the method checks your technology level in agriculture. The `atmosphere_multiplier` will be reset to its base value of 1, if the sum between the multiplier and the level in agriculture divided by 150 is greater than or equal to 1 (Formula 11).

$$atmosphere\_multiplier + \frac{agriculture}{150} \geq 1 \quad \textbf{(11)}$$

If this condition is not met, the agriculture technology will still aid to help compensate the negatively effecting multiplier by setting the `atmosphere_multiplier` to the sum of `atmosphere_multiplier` and agriculture divided by 150 (Formula 12).

$$atmosphere\_multiplier = atmosphere\_multiplier + \frac{agriculture}{150} \quad \textbf{(12)}$$

b) calc_temperature()

The method `calc_temperature()` calculates the temperature, which is dependent on the atmosphere and distance. This method also exists in the class Planet, but is overridden in this class (MainPlanet). If the Main Planet resides in the GZ, it is determined as followed, as per Formula 13:

$$temperature = \left(250 - \frac{distance}{600{,}000}\right) \cdot \frac{100}{atmosphere} \quad \textbf{(13)}$$

If the Main Planet does not reside in the GZ, it is calculated as per Formula 10 (cf. *section 4.3.2.1*), with one minor change: the factor `atmosphere` is changed to `calculated_atmosphere`, to take possible negative effects by the `atmosphere_multiplier` into account. This `calculated_atmosphere` is set in the method `update_variables()` to the value returned from the method `calc_atmosphere()`. Again, the value for `const` is either `-1` or `0.5`, depending on whether or not the Main Planet respectively precedes or succeeds the GZ.

c) calc_usable_landmass()

The method `calc_usable_landmass()` calculates the usable landmass as per Formula 14. Before any changes to the `usable_landmass` are applied, a temporary variable that is between 0 and 1 is calculated (Formula 11). It uses both the tech points of architecture and agriculture.

$$tech\_variable = \frac{architecture \cdot 0.5 + agriculture \cdot 0.5}{15} \quad \textbf{(14)}$$

Similar to the calculation of atmosphere, the user technologies protect them versus negatively affecting multipliers. If the following conditions are met, the `landmass_multiplier` will be reset to its original value of 1 (Formula 15)

$$landmass\_multiplier + \frac{tech\_variable}{10} \geq 1 \quad \textbf{(15)}$$

However, if they are not met, the `landmass_multiplier` will be calculated as followed (Formula 16):

$$landmass\_multiplier = landmass\_multiplier + \frac{tech\_variable}{10} \quad \textbf{(16)}$$

Once the multiplier has been determined, the `usable_landmass` is calculated as per Formula 17:

$$usable\_landmass = landmass\_multiplier \cdot landmass \cdot (0.8 \cdot tech\_variable + 0.2) \quad \textbf{(17)}$$

d) calc_life_quality()

The method `calc_life_quality()` calculates the life quality (a number between 0 and 100) and is dependent on five factors: landmass, usable landmass, architecture, engineering and population health. Temperature plays a two roles: choosing the formula through which the life quality is calculated (role 1) and a factor in these formulas that contains the `calculated_temperature` (role 2).

Different temperature ranges go hand in hand with these different formulas and as such three major scenarios can be distinguished in role 1: (1) the temperature lies in the interval]-25,50[°C; (2) the temperature is lower than -25 °C; (3) the temperature is above 50 °C.

The first (1) scenario describes the GZ and calculates the life quality via Formula 18.

$$life_{\text{quality}} = \left(\frac{engineering}{6}\right) + 35 \cdot \frac{usable_{landmass}}{landmass} + \left[\frac{(1-x) \cdot architecture}{TECH\_CAP} + y\right] \cdot 20 + 0.4 \cdot \text{population\_health} \quad \textbf{(18)}$$

Life quality is a number between 0 and 100, dependant on the five before-mentioned factors plus temperature. Looking at the terms in Formula 15 from left to right:

- Term 1: engineering is a number from 0 to 30 (due to the special technology cap for engineering). This value for engineering was divided by 30 and represents 5% of the life quality, so it is multiplied with 5 to make it a number between 0 and 5. This division by 30 and multiplication by 5 all together is the same as $\frac{engineering}{6}$, as shown in the formula.
- Term 2: usable landmass is divided by landmass to yield a percentage value. This value multiplied by 35 gives a number between 0 and 35, and represents 35% of the life quality.
- Term 3: The y value can be determined per Formula 19. Since the temperature for an organism ideally needs to be between -25 °C and 50 °C, it can be seen as a parabola with a top at ($\frac{25}{2}$,-1). The parabola has intersections with the x-axis at the points -25 and 50. This causes the value to be best at a temperature of $\frac{25}{2}$ °C and worse the lower it goes.

$$y = \frac{(calculated\_temperature + 25)\cdot(calculated\_temperature - 50)}{1406.25} \quad \textbf{(19)}$$

  Term 3 clearly shows that a higher level in architecture will result in a higher life quality. Poor temperatures in the interval]-25,50[will be nerfed via this architecture tech and will negate some effects of the temperature. When x, for example, is a number of 0.25, and the architecture is 15/15 (maxed out), term 3 will have the full boost on life quality, which is 20% of it. The constant `TECH_CAP` is 15 for the three basic technologies.
- Term 4: the population health counts for 40% of the life quality. Healthier people are generally happier. If the population health is at 100, term 4 results in 40% of the life quality.

The second (2) and third (3) scenarios, respectively below -25 °C and above 50 °C calculate life quality in a similar way, via Formula 20.

$$life\_quality = \left(\frac{Engineering}{6}\right) + 35 \cdot \frac{usable\_landmass}{landmass} + \left(1 \cdot \frac{architecture}{TECH\_CAP}\right) \cdot 20 + 0.4 \cdot population\_health) \quad \textbf{(20)}$$

This formula looks identical to Formula 18, yet it is not. For the sake of visibility, term 3 has been multiplied by 1 to indicate the difference between the two formulas. Scenario 2 and 3 do not account for a variable x that is based on the calculated temperature. Instead, the calculated value for the life quality is multiplied by a factor that punishes this extreme temperature. Formula 21 and Formula 22 are respectively for cold temperatures (scenario 2, below or equal to -25 °C) and for hot temperatures (scenario 3, above or equal to 50 °C).

$$life\_quality \mathrel{*}= 1 - \frac{(calculated\_temperature + 25)}{-248} \quad \textbf{(21)}$$

$$life\_quality \mathrel{*}= 1 - \frac{(calculated\_temperature - 50)}{500} \quad \textbf{(22)}$$

e) calc_total_population()

The method `calc_total_population()` calculates the total population of the user's species and is dependent on life quality and the current (total) population in the current turn.

To calculate the (new) total population, multiple phases of the population are taken into account. When the population is less than 5000, the formula below (Formula 23) is used:

$$total\_population = total\_population + \left(\frac{life\_quality}{100} - 0.5\right) \cdot 3 \cdot total\_population \quad \textbf{(23)}$$

When the population is greater than one billion (1,000,000,000), the formula below (Formula 24) is used:

$$total\_population = 1{,}000{,}000{,}000 + \left(\frac{life\_quality}{100} - 0.5\right) \cdot \left(\frac{1{,}000{,}000{,}000}{1.3}\right) \quad \textbf{(24)}$$

When the population is in between, so [5000;1,000,000,000] the following formula (Formula 25) is used:

$$total\_population = total\_population + \left(\frac{life\_quality}{100} - 0.5\right) \cdot \left(\frac{total\_population}{1.3}\right) \quad \textbf{(25)}$$

As mentioned earlier in this section, life quality is a percentage value between 0 and 100, so it is divided by 100 to yield a number between 0 and 1. It is subtracted with 0 and, as such, the total population can decrease or increase depending on life quality. This term is then either multiplied by $3 * total\_population$, $\frac{1{,}000{,}000{,}000}{1.3}$ or $\frac{total\_population}{1.3}$. The values 3 and 1.3 are experimental values.

In short, if the current (total) population is lower than one billion, it will use the value for the current population. If it is not, then it will nerf the formula by using a flat value of one billion to prevent the equation from being overpowered. If the population were to ever dip below 0, the population will automatically be set to 0. This will result in an unsuccessful run of the simulation, as one of the losing conditions is a population equal to or below 1.

Lastly, the `total_population` is multiplied by its own multiplier to account for possible events.

$$total\_population \mathrel{*}= population\_multiplier \quad \textbf{(26)}$$

The method `calc_progression()` calculates the progression of the user's species. It is calculated through several formulas. Formula 27 is used when there is an increase in population. The local variable `calc_progression` is the return value to `calc_progression()`.

$$calc\_progression = progression - \left(1 - \frac{life\ quality}{100}\right) \cdot \frac{rate\ of\ change(population)}{250{,}000} \quad \textbf{(27)}$$

As mentioned earlier in this section, life quality is a percentage value between 0 and 100, so it is divided by 100 to yield a number between 0 and 1. It is subtracted from 1 to create a fraction value. As such, the progression can decrease or increase depending on life quality. It is then also multiplied with the rate of change in population, which is a fancy way for the difference between the previous population and the current population (total population), and divided by a number that came from experimentation. If the calculated progression is more than $\frac{progression}{1.5}$, it is capped at that value. Note that, here, progression indicates the current progression in the current turn.

$$calculated\ progression = \frac{progression}{1.5}$$

If the population is too big, as in bigger than 1,000,000, the population to calculate the progression is capped at said one million. This helps prevent the calculated progression (Formula 28) from exploding in extreme proportions.

$$calc\_progression = progression + (engineering + 0.25 \cdot architecture + 0.1 \cdot medicine + 10 \cdot \left(\frac{life\ quality}{100} - 0.5\right) \cdot \left(\frac{population}{500{,}000}\right) \quad \textbf{(28)}$$

The familiar life quality divided by 100 is, so it is a number between 0 and 1, and is then subtracted minus one half, so when life quality is too low (under 50), the progression will start decreasing. The factors 10 and 500,000 are experimental factors to help balance the equations. Furthermore, three technologies are also factored in:

- engineering, because this is important to progression in general;
- architecture, because you need decent infrastructure to design your space ships;
- medicine, because you have to research how the human body is effected by long-time exposure to radiation in space.

Agriculture is not involved, because farming potatoes and building a space ship are two entirely different things. Granted, your species have to eat in order to continue their work, but this train of thought is implemented via life quality, as this is covers a broader sense of said factor of comfort in food and health.

4.5 EventManager

The class EventManager handles all the events and has, aside from its constructor `__init()__`, only two methods. These methods are `get_multipliers()` and `generate_event()`.

The method `get_multipliers()` will return a list, with all its necessary multipliers that were fetched from the CSV file. The method `generate_event()` only has one parameter, which is the `progression` (of the user's species). It will first calculate the chance of a disaster, depending on the `progression`. Then, two different things may happen. If the event takes place and is a disaster, it will pop an event from the `disaster_list` and fill all the necessary variables (as fetched from the CSV file, cf. *section 2.1.4*). Similarly, if the event is a breakthrough, it will pop an event from the `breakthrough_list` and fill all necessary variables.

4.6 GUI and its classes

The project can be sliced in two halves: the model and the view. The view or GUI serves as the window to the user, whereas all the previous classes that have been examined in depth serve as the model. Although this section will take a closer look at the implementation of the GUI, it will not be discussed as thoroughly as the model [12]. Mainly because many of these methods are fairly straight-forward and use existing methods from packages that add buttons, labels, music etc. The initial assignment focuses on the implementation of a non-trivial algorithm rather than the design and development behind a GUI.

The Python file GUI contains a total of six classes that all play their own specific part in the visualisation of Project Stars: `Application`, `StartPage`, `MainPage`, `PlanetDrawing`, `MyPopupWindow` and `EndPage`.

4.6.1 Application

The class Application uses the method `update_music()` to update the music. The `start_startscreen()` method creates an object of StartPage and displays it. `start_simulation()` will create an object of MainPage and display it. Finally, the `window_size()` will change the size of the container object.

The user might want to save mid-simulation. To be able to do this, a function to save and load is implemented. The simulation can only be saved when a main planet has been chosen, and a focus has been set. To do this, the method `save_file()` was created. The object StarSystem is given as a parameter with the method, and that object is written into a file. This method uses the built in module Pickle and is, therefore, saved as a PKL file.

To load that file again, the method `load_file()` is used. This time the frame is the parameter of the method. First, the object StarSystem is loaded from the save file [12]. Next, the method `load_startpage()` is called from the object, of type MainPage, frame to update the GUI. StarSystem is the only parameter of this method. The method load_startpage() is explained in *section 4.6.2* of this chapter.

4.6.2 StartPage, MainPage and EndPage

These pages represent the three different frames or windows. The StartPage will show three buttons and play music in the background. The MainPage is the frame that contains the simulation itself. It shows the planets, all the information, the events and the buttons. The EndPage shows whether or not the simulation was successful and displays a small frame that provides the user with their total_population upon completing the simulation. EndPage uses the method `show_end()`, while StartPage does everything in its `init()`.

The MainPage consists of multiple methods, which are each used for a separate purpose. The method `research_focus()` will check if the thread for drawing is already live. Depending on if it is, the program will first ask the user to either choose a focus through the code in `instruction_path()`, or change the focus at the moment and change the related buttons, this being the technology buttons at the top. The `instruction_path()` method is the method that forces the user to start a simulation, this being choosing a Main planet etc.

A rather peculiar method is `thread_make()`. The method will create a separate thread for controlling and altering the canvas. The code that the thread will run is the method `update_canvas()`. This methods makes the planets move. The reason the method is made a separate thread, is that without it, there would be two unfortunate choices. Either `sleep()` is not used, and the drawing of the planets happens too fast (and rotate at the speed of light), or either `sleep()` is used, but you can almost never use the buttons on the application, because the whole code is in sleep mode. So the choice was made to use a separate thread for the drawing, so `sleep()` could be used.

The method `create_canvas()` will create the canvas, and through that set its background, size and location. It will also call the method `show_planets()`, which in turn will display the planets on the canvas.

The method `update_canvas()`, as mentioned earlier, will move every planet by calling the `planet.move_obj()` (from the class PlanetDrawing) for each planet in the list of star system, which it holds as a parameter. This method will then use the `time.sleep()`. All of this code will be put into an infinite loop, so it will loop until either the separate thread is deleted, or the application is stopped.

`show_planets()`, as previously mentioned, is the method that will display the planets on the canvas. The parameters of this method are a list of the planets and the canvas. It will then draw the planets, using the provided list which generates objects of the type PlanetDrawing. The method `get_random_angle()` will generate a list of random angles, 20 or more degrees separated each, and as much as there are planets. These are then placed together on the canvas, depending on the angles, added with the associated movement circles.

`create_progressbar`, `create_message_window` and `create_info_frame_planet` are all pretty self-explanatory methods. They create and draw the progress bar, message window and the frame with all planet and organism information (planet and organism attributes). The `create_info_frame_planet()` method will use an auto scale to scale all elements to show.

The method `update_info_frame_label` will update the information labels of the planet and organism attributes. It will also show the difference (rate of change) with the previous value (in the previous turn), to provide the user with a sense of visible change.

The method `set_main_planet()` was created to select the Main Planet. It asks the user upon pressing the button to set a selected planet as their Main Planet whether or not this is the planet the user would like to select. If the user agrees and presses yes, the button to set the Main Planet will be disabled and a short instruction will be displayed, wishes the user good luck. It passes the selected planet on to Star System, where Star System's `set_main_planet()` will perform all the hard work and calculations. Next, it calls the methods `thread_make()` to perform its part and the method `create_info_frame_planet()` to create the frame that displays all necessary information, as previously mentioned. It will also prompt the user with a window that allows the user to spend their 12 initial technology points through `spend_points()`.

The method `next_turn()` is most likely the key method of the whole simulation. It will call all the calculations in the right order and interact with the GUI elements to display everything to the user. These calculations, the formulas, and the reasoning behind them are all explained in *sections 4.3* and *4.4* of this chapter.

In `load_startpage()` the StarSystem of frame is updated to the most up to date values for all information, labels, bars, the year, planet drawings etc. This method is only used when loading another star system from the save file. Because of this other star system, all its parameters have to be changed and updated. The frame needs to know that there is another Main Planet, and so the value of `main_planet` is changed to the current Main Planet. The method `create_canvas()` will be called to create the canvas with all the planets and star. Next, the information window is updated with the right information, which includes and covers the technologies, turns etc. A couple more things need to happen: all technology buttons are updated such that the right one is selected; the message window is updated; the music is enabled; and finally the method `update_canvas()` is called to rotate the planets.

The methods `auto_update_turn()` and `auto_update_change()` are used to have the simulation automatically click the next turn and to enable and disable the auto turn feature.

Finally, the method `planet_color()` is used for changing the colour of the planet which, as previously mentioned, is determined through the life quality of the organism.

4.6.3 PlanetDrawing

The class PlanetDrawing contains all the planet rings and the background image. It also contains the method `move_obj()` which moves the planet in a circle around its star. The methods `show_planet_info()` and `change_color()` are self-explanatory. The first method will call the Planet method `show_planet_info()`. The second method will change the colour, depending on the life quality of the population to visually show the user the species' life quality.

4.6.4 MyPopupWindow

Once the user has selected their Main Planet, they are allowed to spend a total of 12 technology points. A window will pop up, prompting the user to spend their points. This class is used for this input of technology points at the start of the simulation. It represents a window with 3 labels and 3 text fields, and an ok-button It uses the method `ok_enter()` to show this. The method `ok()` checks whether or not the entered values are numbers and add up to a total of 12.

Conclusion and future work

The main purpose of this multidisciplinary engineering project was to implement a non-trivial algorithm in Python and as such create a game, or a simulation etc. Group 9 chose to create a simulation of a planetary system with one central star. The main goal of this simulation was to enable the escape of the user's organism from its home planet, and in doing so, guarantee its survival. This document provided the reader with a step by step record of the process and steps taken throughout the development of Project Stars.

The SDLC, which details the steps taken throughout the development of said simulation, allowed the group to better structure and organise the tasks at hand. The first phase, planning, resulted in the outlines of the simulation in terms of functionality and described this in great detail from the get go. Many of the formulas were designed in this phase and reflect a simplified version of reality. These formulas were tweaked throughout the implementation phase if tests would show that an attribute or a technology was too powerful or too underwhelming.

The second phase, analysis, allowed the group to clearly state the various needs to undertake this project. This was done through UML diagrams, which provided a clear-cut view of the interaction between the objects that resulted from phase one.

Phase three and four, design and implementation, consisted of the actual coding of the project, as well as the development of the GUI, manual and Analysis & Functionality Report. The application, named Project Stars, was initially developed as a prototype that provided the developers with prompt-based results (i.e. no GUI), and allowed them to tweak formulas and errors in the calculations of the various (planet and organism) attributes. This prototype still exists and would allow the parsing of all data in case a user or developer would like to see every variable every single step of the way. The final version, of course, comes with a fully designed GUI, which was made possible through Python's Tkinter.

Project Stars' strengths lie in the fact that each simulation will have its own uniqueness. Two simulations may be similar, but they will never be the same. The many attributes working together, as well as the mechanic Events, has helped create a rather challenging simulation, if the user so choses it to be.

Looking forward, it would not be difficult to tweak formulas or values and improve the simulation, due to the way it was programmed and designed. Future versions of this simulation could have the organism start from its new planet, with its obtained technologies readily available. As such, a quest towards continuous survival, space exploration/domination could very much become a reality.

There were some difficulties in creating a functioning EXE file towards the final stages of the project, but these issues were eventually resolved. Project Stars has proven to be a challenging but very fruitful learning experience for every team member in group 9.

References

A

[1] Almodovar, J. S. [Jeremy Shane Almodovar]. (2017, February 3). *Interstellar - Do Not Go Gentle/No Time For Caution* [Video File]. Retrieved from https://www.youtube.com/watch?v=gSkIwwdT79E&t=176s

[2] Ambler, Scott W. (2004). *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge: Cambridge University Press.

[3] Ambler, Scott W. (2012). The Agile System Development Life Cycle (SDLC). Retrieved from http://www.ambysoft.com/essays/agileLifecycle.html

B

[4] Big Support. *What is a CSV file and how do I save my spreadsheet as one?*. Retrieved from https://support.bigcommerce.com/articles/Public/What-is-a-CSV-file-and-how-do-I-save-my-spreadsheet-as-one

C

[5] CMS & Department of Health & Human Services USA. (2008). Selecting A Development Approach [PDF file]. Retrieved from https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf

D

[6] Derek & Brandon Fiechter. (2017, April 18). *Dark Space Music - Black Hole* [Video file]. Retrieved from https://www.youtube.com/watch?v=bR_TenG3RJk

N

[7] Naessens, H. & Cnops, J. (2017). Softwareontwikkeling I: Analyse & Ontwerp [Lecture notes]. Retrieved from Minerva Online Course Material.

[8] Nasa Exoplanet Archive. (2016, December 14). Retrieved from https://exoplanetarchive.ipac.caltech.edu/

O

[9] Oberguggenberger, M. & Ostermann, A. (2011). *Analysis for Computer Scientists*. London: Springer-Verlag London.

P

[10] Phillips, D. (2010). *Python 3 Object Oriented Programming* [PDF file]. Birmingham: Packt Publishing, Retrieved from https://docs.google.com/file/d/0By4GdMmzUrGAWmF4SzZNVDIwUEE/view

[11] Pointal, L. (2015). *Python 3 Cheat Sheet* [PDF file]. Retrieved from https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf

[12]Python Software Foundation. (2017). *12.1. pickle – Python object serialization*. Retrieved from https://docs.python.org/3/library/pickle.html

[13]Python Wiki. (2012). *Why is Python a dynamic language and also a strongly typed language?*. Retrieved from https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

R

[14]Rouse, M. (Microservices)(2008). *Object-Oriented Programming (OOP)*. Retrieved from http://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP

[15]Rouse, M. & Martinez, R. (Tech Target)(2005). *Definition: thread*. Retrieved from http://whatis.techtarget.com/definition/thread

S

[16]Shipman, John W. (2013, December 31). *Tkinter 8.5 reference: a GUI for Python* [PDF file]. New Mexico Tech Computer Center. Retrieved from http://infohost.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf

T

[17]Tutorials Point. (2017). *UML – Activity Diagrams*. Retrieved from https://www.tutorialspoint.com/uml/uml_activity_diagram.htm

V

[18]Visual Paradigm. (2016, December 15). *Visual Paradigm Quick Start* [PDF file]. Retrieved from https://d1dlalugb0z2hd.cloudfront.net/quickstart/quickstart.pdf

Appendix A. Project Stars Manual

While the Project Stars simulation does not have a steep learning curve, a slight introduction can be useful to anyone who would like to attempt a successful first simulation. This manual provides you with a quick five minute summary of the simulation and teaches you about the various windows, buttons and mechanics that Project Stars has to offer.

Starting up

When you start up Project Stars, you will start with a similar screen as the one down below (Image 1). The three available buttons are pretty self-explanatory: **Start** will initiate a new simulation, **Load** will start up the simulation from where it was previously left – provided that you have a previously saved simulation – and finally, **Exit** will close the screen. Let's click on start and get going.

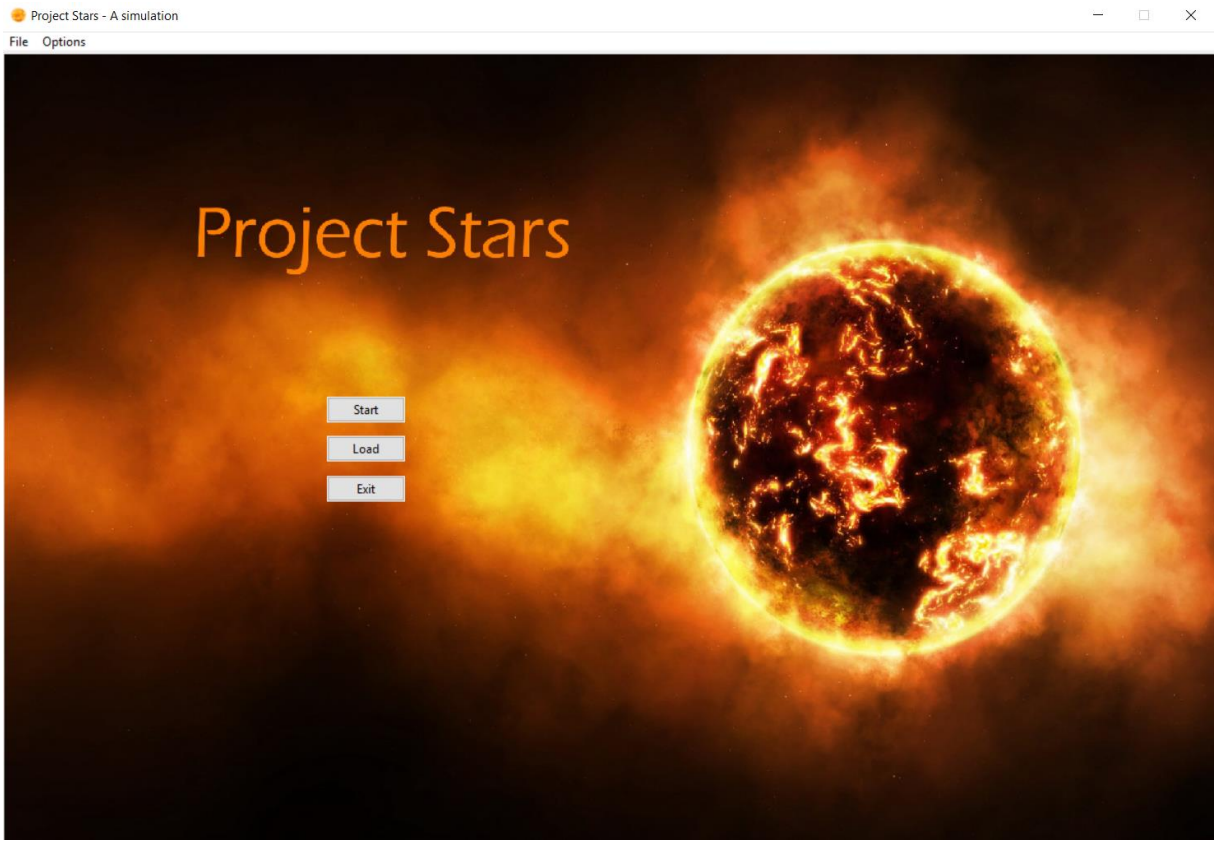

Image 1: Main menu

On the left hand side of the screen, you can see the Project Stars star system and the planets orbiting it. This star system has been randomly generated for you. As the instruction message states, you can click on the **planets** (the rotating red orbs on the white circles) to see their properties.

(1) In the top left corner, a **file** button is located, which contains the buttons **save** and **load**. Saving is only possible when the actual simulation has begun, so only the load button will be highlighted.

(2) On the right side, a number of buttons can be seen, but these are not highlighted at the moment. They will become available as the simulation commences.

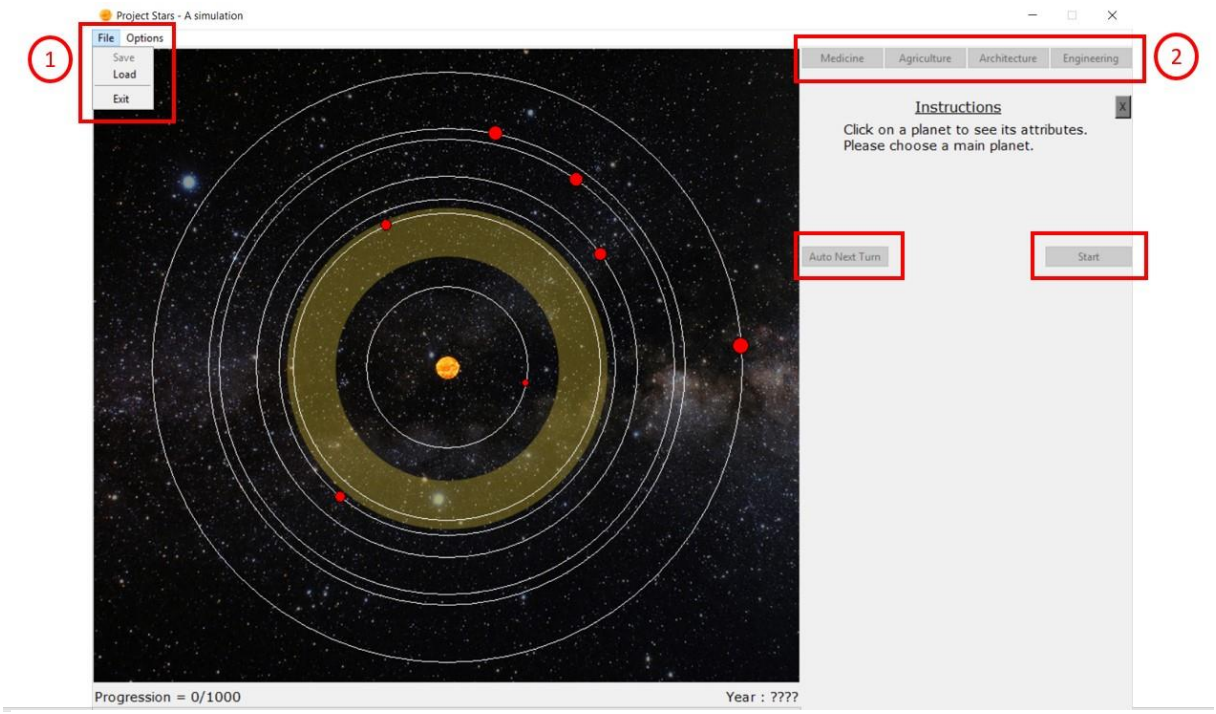

Image 2: Starting screen

Planet selecting

When a planet has been selected, all of the basic information will be shown to you in a new **information panel** on the right (Image 3). This contains the planet's name, distance to the star, radius, percentage of surface that is habitable landmass, a percentage of how good the atmosphere is in comparison to an ideal atmosphere, temperature in degrees Celsius and a planet quality. This percentage is a combination of all the above, which determines the overall quality of the planet.

The golden zone in the centre is known as **Goldilocks' Zone**. Because of the distance from this zone to the star, the circumstances of the planets inside tend to be better suitable to sustain life than the planets that are closer or further. For the first few simulations, we highly recommend using one of these planets.

When the planet of your choice has been selected, click the button underneath the attributes to confirm this as your **Main Planet**. A pop-up screen will appear to confirm your choice. From this moment on, this planet is your Main Planet.

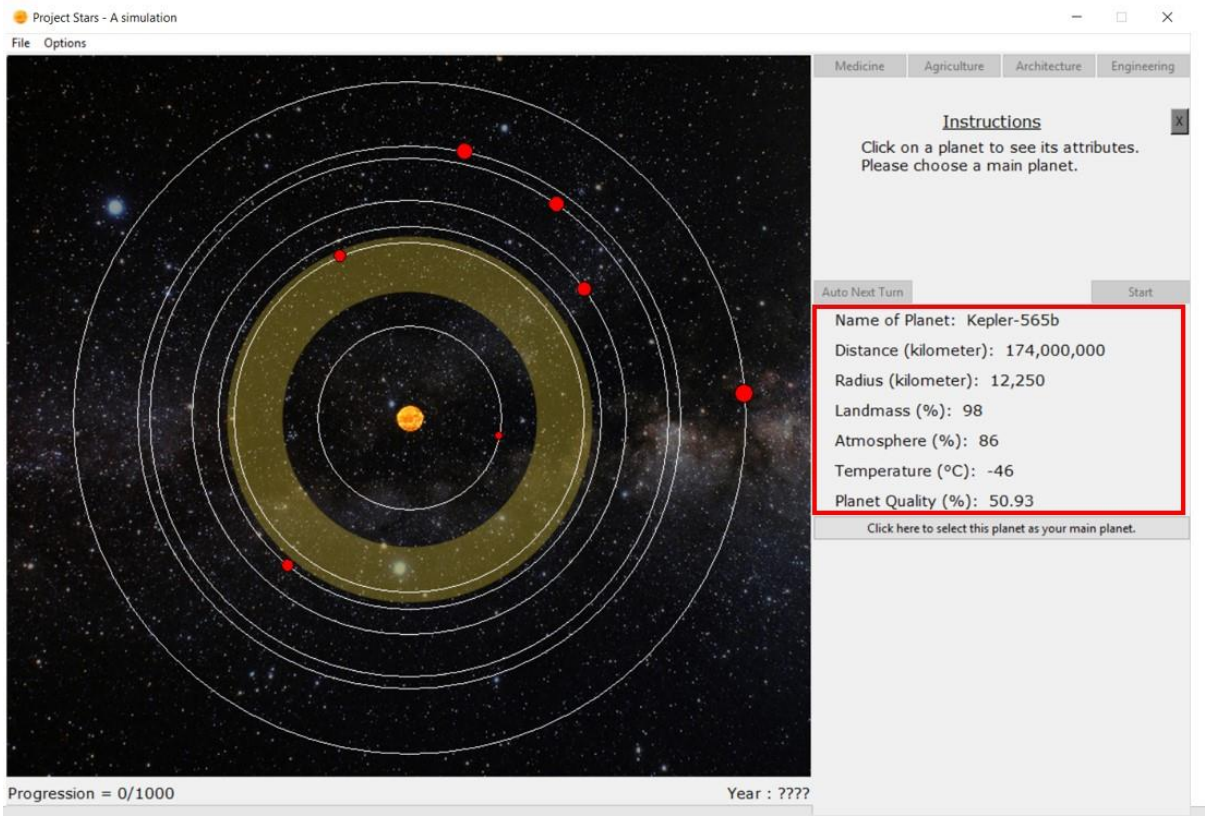

Image 3: Planet attributes

Setting technologies

Once the Main Planet has been set, another pop-up screen will appear (Image 4). This will ask you to give your organism a basic 12 points in the **technologies** that can be seen on the starting screen (notice that engineering is not included). Technologies provide assistance to the organism, be it by improving the organism or gaining resistance against the possible disasters that could strike the planet. The three "supportive" technologies (medicine, agriculture and architecture) have a cap of 15, while the "main" technology engineering has a cap of 30. Each technology can be further researched as the simulation progresses. The influence technologies have are explained in the following table.

Medicine	Mostly used to keep your organism healthy, medicine is the only technology that will increase your population health, thus resulting in a higher life quality. Also responsible for a small fraction of your progression.
Agriculture	As your organism needs to feed itself to survive, it will produce food on the usable landmass of the planet. Upgrading agriculture will make the organism more capable of recovering the landmass, should it be affected by a disaster. More food means that more productive organisms can live, thus increasing your population. Also gives knowledge about how to fight disasters that damage the atmosphere.
Architecture	Similar to agriculture, architecture ensures the organism can recover faster from disasters that affect the usable landmass. Also has a decent impact on the progression, not as much as engineering, but more significant than medicine.
Engineering	The main technology of the simulation. Engineering unlocks secrets of the universe for the organism, needed to create a spacecraft that can evacuate the species from the planet before it's too late. Has a massive impact on the progression in comparison to the other technologies, while also slightly boosting the life quality though luxury products.

It is a wise decision to already start **preparing** your organism, depending on the planet that was chosen. Living on a small planet with a limited amount of landmass? Focus on agriculture and architecture to prevent hunger. Poor atmosphere? Agriculture will prevent a complete breakdown, should a disaster hit the atmosphere, etc.

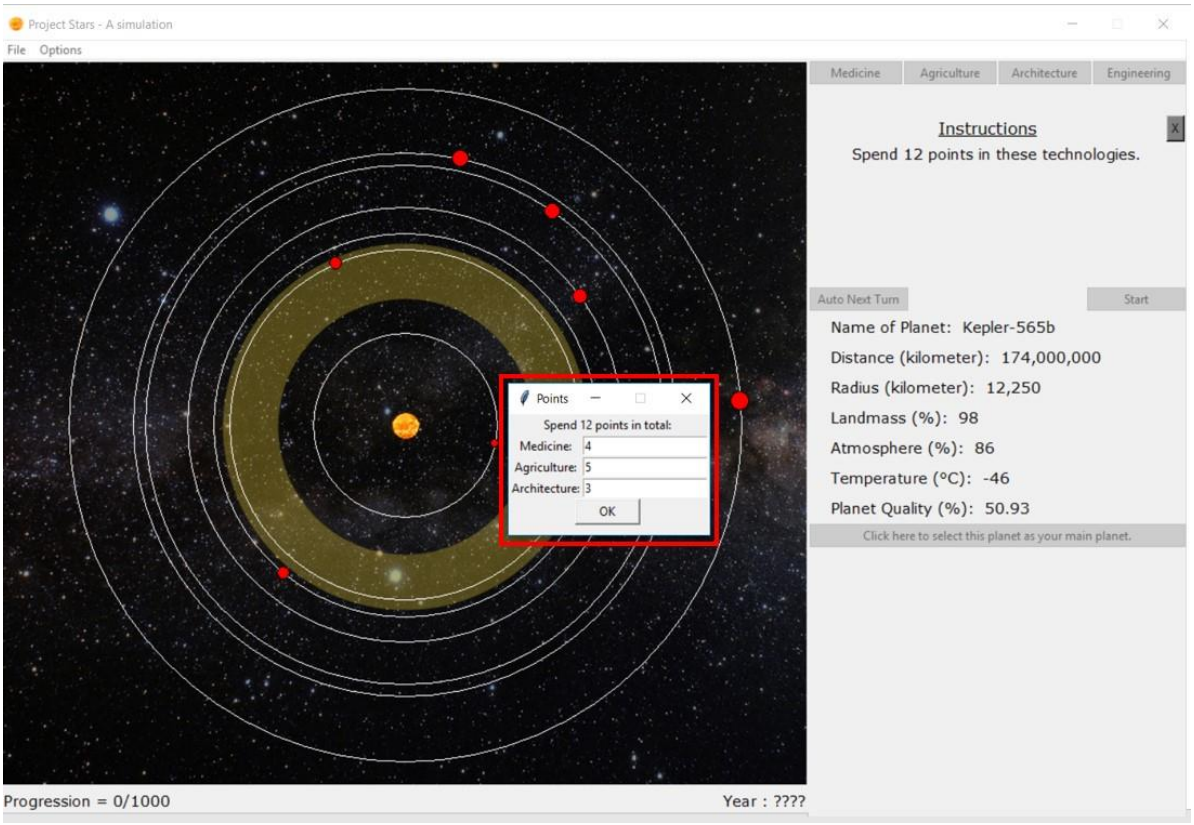

Image 4: Setting starting technologies

Setting a research focus

Now that the base technologies of the organism have been set, the starting **research focus** must be appointed. Click on one of the technology buttons in the upper right corner, which are now available (Image 5).

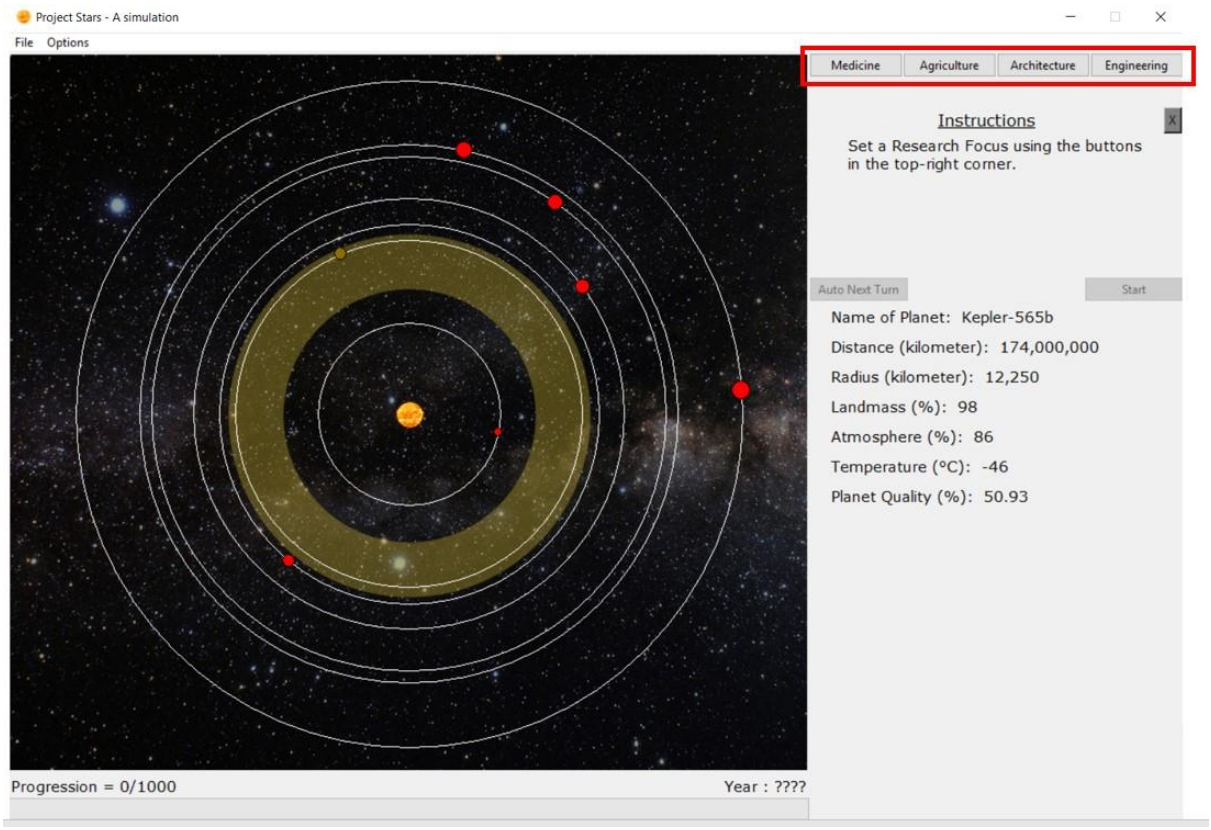

Image 5: Setting research focus

Simulation start

Now that everything is ready, the simulation is can start. The **start-button** is now available – click it. Some changes were made: the planets started rotating in their respective rings and some **additional information** has been made available: the amount of turns remaining, turns left for technology advancement, the usable landmass, (population) health, life quality, population and all of the technologies (Image 6)

- Turns remaining: Each simulation consists of 100 turns. With each turn, the population will grow or shrink, life quality can change and all kinds of events can happen. The turns will only progress when the "next turn" button is pressed, so you have as much time as you like to adjust to the new situation.

- Turns left for technological advancement: As explained above, technologies can increase over the course of the simulation. It takes 5 turns of consecutive research to elevate a technology to a higher level (so should the research focus change from one technology to another, all progress towards another level in the former technology will be permanently lost). This number shows the amount of turns left until the elevation takes place.
- Usable landmass: This number represents the percentage of the landmass that the organism currently has available.
- Population health: The health of the population. Disaster can bring this number down, while it will slowly recover afterwards (depending on the medicine technology). Since a sick population has a hard time working on a way to escape from the planet, bad health will cause a slowdown in the progression.
- Life quality: Perhaps the most important value in the simulation. Life quality depends on your usable landmass, health, architecture and engineering. It represents the mental state of your organism as a whole. This is the most important factor of your population growth, while also playing a major role in the progression.
- Population: The current population.
- Technologies: The current technology levels

Next to some of the attributes, there will be brackets with two minus signs inside. As the progression runs, the **changes** relative to the previous turn will be shown between these brackets.

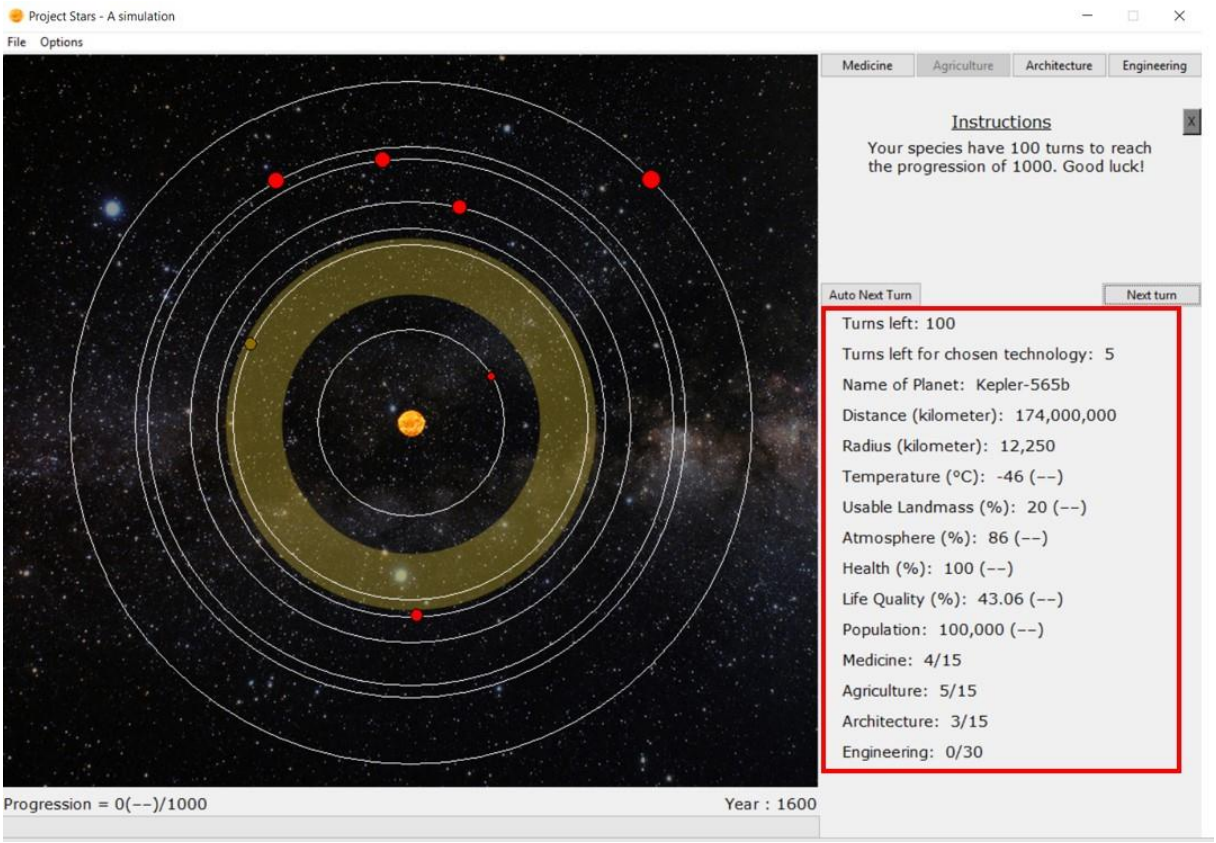

Image 6: Additional attributes

Events

As the simulation progresses, certain **events** will randomly occur on your planet or in your star system. Some of these are useful (Breakthroughs, Image 7), others will have rather annoying (or devastating, depending on the event) consequences (Disasters, Image 8).

Breakthroughs are events are helpful to the organism. Depending on the event, one of the four technologies will get an instantaneous boost of 1. This is completely independent from the current research or tech levels already accumulated.

Disasters are events will set the organism back a few steps. It is the simulations way of trying to prevent you from getting of the planet. Simulations where the progression has furthered a lot will encounter more and deadlier disasters. While the main objective is to escape, rushing it might leave you with little defence against the disasters that might lie ahead.

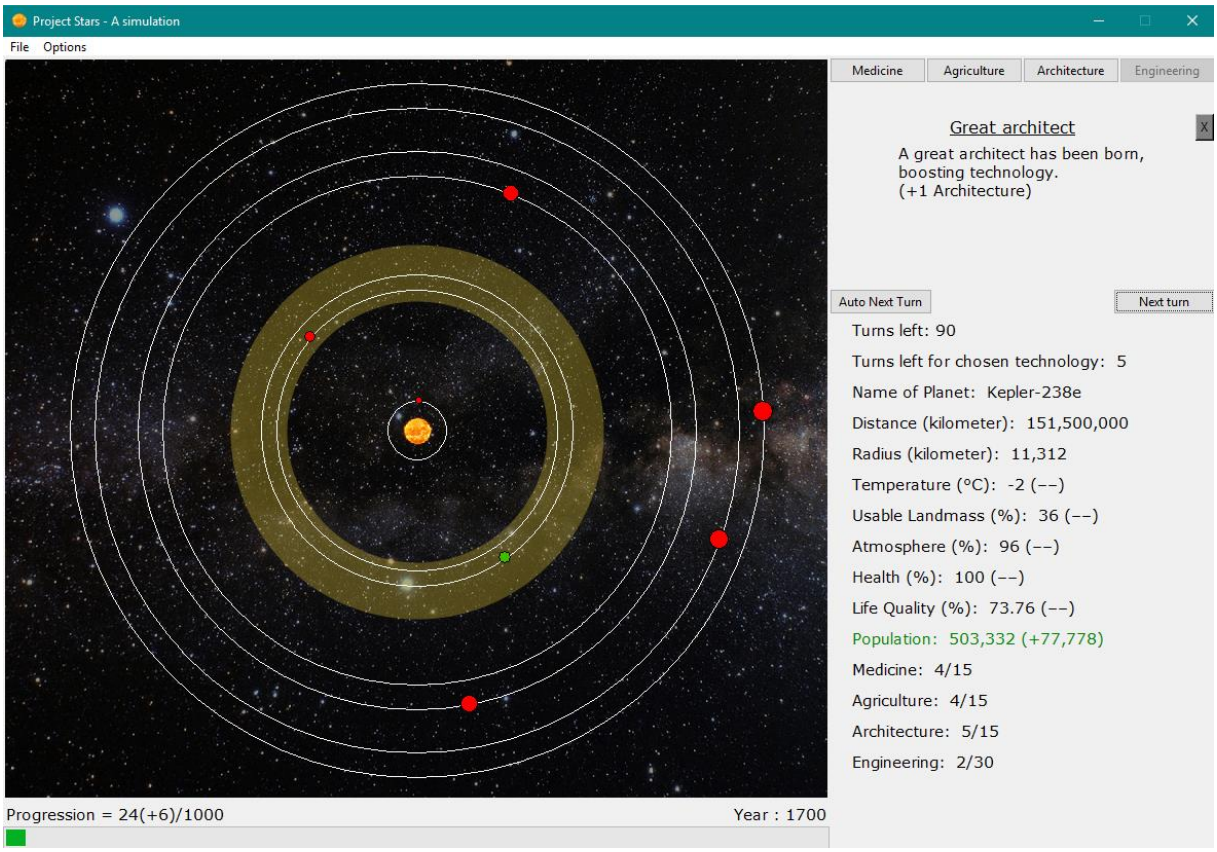

Image 7: Example of a breakthrough

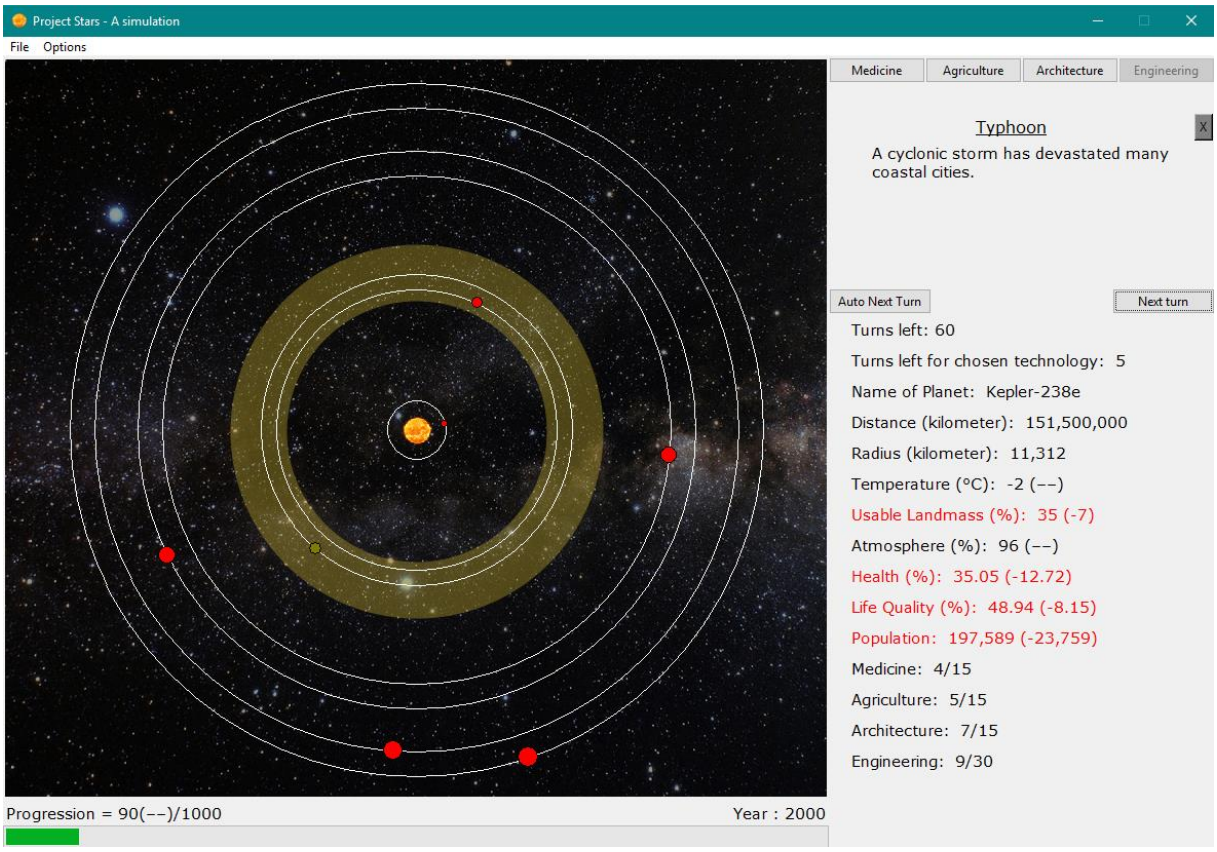

Image 8: Example of a disaster

Appendix B. Technical Documentation

Class StarSystem:

Class that holds all info, planets and key methods.

```
def set_up(self):
```

Setup, read in all planet names and give planets different distances.

```
def set_main_planet(self, planet):
```

Sets the main planet as per user-choice and stores it in the appropriate list and index.

```
def next_turn(self):
```

Key method that simulates the next turn and calls all methods to update their values.

```
def check_winning_condition(self):
```

Checks whether the game is either won or lost, depending on the progression and population.

Class Planet:

Class that holds basic info about a planet and therefore represents one.

```
def calc_temperature(self):
```

Calculate the temperature depending on distance and atmosphere.

```
def planet_quality(self):
```

Calculate the planet quality, depending on the temperature, atmosphere and landmass.

```
def show_information(self):
```

Display the information of the planet, like name, radius and more useful information.
Overridden in class MainPlanet.

Class MainPlanet(Planet):

Class that holds all the info about the organism and is more detailed than Planet.

```
def spend_points(self, points):
```

Spend the technology points at the start of the simulation.

```
def set_research_focus(self, index):
```

Method to choose research focus and passively upgrade technologies.

```
def update_research_focus(self):
```

Check if research is complete or not, and if yes, update the technologies and reset counter.

```
def change_base_values(self, event):
```

Choose the type of method to call, depending on the type of event.

`def update_multipliers(self, multipliers):`

After a disaster, update the multipliers.

`def update_technologies(self, tech_index):`

When a breakthrough has occurred, update the right technology.

`def update_variables(self):`

Key method to update all variables separately, through its correct methods.

`def calc_atmosphere(self):`

Calculate the atmosphere, and regenerate the multiplier.

`def calc_temperature(self):`

Calculate the temperature depending on the atmosphere and distance.

`def calc_usable_landmass(self):`

Calculate the usable landmass, depending on architecture, agriculture, and its multiplier.

`def calc_life_quality(self):`

Calculate the life quality, depending on the temperature, landmass, usable landmass, architecture, engineering and population health.

`def calc_total_population(self):`

Calculate the total population, depending on life quality and its current population.

`def calc_progression(self):`

Calculate the progression, depending on the population, the life quality, and technologies.

`def show_information(self):`

Print the information about the planet, like distance, name of planet, and more relevant information.

`def cache_population(self):`

Save the population, to calculate the difference in the next turn.

Class EventManager:

Class that reads event file, manages events and generates a random event.

`def generate_event(self, progression)`:

Method to generate a random event and select one, depending on chance and progression.

`def get_multipliers(self)`:

Return the multipliers of the occurred disaster.

Class Application(tk.Tk):

Class Application is the main application of the Gui file.

```
def update_music(self):
```

Play the music. Music can also be muted.

```
def start_startscreen(self):
```

Show the start screen.

```
def start_simulation(self):
```

Show the simulation panel.

```
def load_simulation(self):
```

Load the start screen from a previous, saved simulation.

```
def save_file(self, ss):
```

Save the current simulation.

```
def end_frame(self, index, information):
```

Show the appropriate end screen.

```
def load_file(self, frame):
```

Load the StarSystem object from the save file.

```
def enable_save_file(self, x):
```

Enable the save button.

```
def window_size(self, width, height):
```

Sets the size of the window.

```
def on_exit(self):
```

Confirmation window which is called upon closure of window.

Class StartPage(tk.Frame):

This class is the start window.

Class MainPage(tk.Frame):

This is the class with all other canvasses and objects.

```
def load_startpage(self, ss):
```

Load the start page.

```
def research_focus(self, index):
```

Enable all buttons except the current focus.

```
def create_canvas(self):
```

Create window to draw figures on.

```
def thread_make(self):
```

Create a thread for drawing the planets.

```
def show_planets(self, obj, planets):
```

Make new PlanetDrawing objects who draw themselves.

```
def get_random_angle(self, number):
```

Generate number of angles who differ 20 degrees with each other.

```
def update_canvas(self):
```

Make planets move.

```
def auto_update_change(self):
```

Change state when button is pressed.

```
def create_progressbar(self):
```

Create the progress bar.

```
def create_message_window(self):
```

Create message window. Used for communication with user.

```
def show_message_window(self, title, body_text):
```

Show message window. Used for communication with user.

```
def create_info_frame_planet(self, startinfo, main=False):
```

Show planet information, if MainPlanet set, only this information will be shown.

```
def update_info_frame_label(self, startinfo, planet=None):
```

Update the information labels.

```
def set_main_planet(self, planet):
```

Remember the main planet and asks for points set.

```
def next_turn(self):
```

Generate next turn.

```
def planet_color(self):
```

Create colour code.

```
def instruction_path(self, x):
```

Show instructions to guide the user.

Class PlanetDrawing:

Class object that contains the planet drawing and the planet ring.

```
def move_obj(self):
```

Set drawing on new coordinates.

```
def show_planet_info(self, event):
```

Get information about the planet.

```
def change_color(self, color):
```

Change colour of planetdrawing; only used for main planet.

Class MyPopupWindow:

Asks user for tech point distribution.

```
def ok_enter(self, event):
```

Calls method ok when used.

```
def ok(self):
```

Checks if the values are valid.

Class EndPage(tk.Frame):

Shows the end frame.

```
def show_end(self, index, information):
```

Show end text.

Appendix C. List of Events

All Events are stored in Events.csv, and are read and managed by the EventManager class. The header is as followed:

Disaster header:

```
0;start;end;"sound_file_name";"title";"message";atmosphere_multiplier;landmass_multiplier;population_health_multiplier;population_multiplier
```

0;0;250;CrowdPanic;Thieves and Murderers;A large group has started raiding and murdering people!;0.85;0.8;0.8;0.9
0;0;350;CrowdPanic;Black Death;An infectious bacterial disease has spread all over the world and has decreased the health of your population.;1;1;0.3;0.4
0;0;1000;Earthquake;Earthquake;An earthquake has occurred and has damaged your infrastructure!;0.8;0.6;0.7;0.75
0;0;1000;Hunger;Starvation;Famine sweeps across the globe!;1;1;0.4;0.7
0;0;1000;Volcano;Volcanic eruption;Death Mountain has erupted and has filled the atmosphere with ashes.;0.4;0.8;0.6;0.9
0;0;1000;Tsunami;Tsunami;A giant 10-meter tall tsunami destroyed one of the biggest capitals.;1;0.8;0.5;0.7
0;0;1000;Disaster;Environmental disaster;Something has caused a very strange atmospheric change -- maybe the CPU is overheating?;0.2;0.5;0.8;1
0;0;1000;Fire;Forest fire;Panic arises, many farmlands are lost and pollution levels are skyrocketing.;0.7;0.5;0.7;0.95
0;0;1000;SnowStorm;Blizzard;A severe blizzard has made certain crop-fields unusable and many people have gotten the common cold.;1;0.9;0.6;0.95
0;0;1000;Typhoon;Typhoon;A cyclonic storm has devastated many coastal cities.;1;0.8;0.65;0.9
0;0;1000;Drought;Drought;Many crops have died and the scorching heat has made life difficult.;0.95;0.7;0.75;1
0;0;1000;Tornado;Tornado;A violent and dangerous tornado has destroyed many houses.;1;0.95;0.7;0.9
0;0;1000;Asteroid;Asteroid;A huge pile or rocks has crash landed on the populated planet, thousands might die.;1;0.95;0.25;0.5
0;0;1000;Flu;Jungle Fever;People are sick and dying.;0.95;1;0.5;0.97
0;0;1000;Flu;Ebola;The disease is spreading rapidly!;0.95;1;0.4;0.97
0;0;300;Gargamel;Gargamel;Gargamel is trying to catch some smurfs with Azrael, but he keeps on failing. So he isn't really affecting anything...;1;1;1;1
0;100;300;MedievalWar;The Great War;Global tensions between different cultures have thrown the planet into a World War.;1;0.9;0.7;0.5
0;100;400;Disaster;My God or Your God;Religious tensions have reached a new boiling point, causing people to fight each other. Is there truly a Creator?;0;1;0.9;0.8

0;150;1000;Flu;The flu;Influenza is on the rise due to a movement's refusal to vaccinate!;0.9;1;0.8;0.6
0;250;750;Bankruptcy;Bankruptcy;People can't pay for food anymore and the economic system is failing!;1;1;0.8;1
0;300;700;Snake;Land of Nopes;Someone brought their Nopes from Australia, and they have started killing people.;1;1;0.75;1
0;300;900;Tobacco;Tobacco;Somebody inhaled the fumes of some burning leaves and, sadly, enjoyed it.;0.6;0.9;0.4;1
0;450;1000;RedMustang;Red Mustang Pollution;One of your species has bought a Red Mustang and it has dramatically decreased the quality of your atmosphere.;0.5;1;0.75;1
0;450;1000;Zombie;Zombie outbreak;A lot of species have transformed into zombies. A cure is being researched...;0.8;0.9;0.3;0.4
0;450;1000;PoliticalCrisis;Political Crisis;People are rioting because a dictator has taken over parts of the world!;0.8;0.9;0.9;0.9
0;700;900;Kardashians;The Kardashians;Too many were born, and ruined a bit of everything -- from North to West.;0.8;0.9;0.95;1
0;500;1000;TonaldDrump;New Leader: Tonald Drump;Your species have elected a new leader named Tonald Drump. He has caused a lot of pollution and decreased overall healthcare.;0.75;1;0.75;1
0;500;1000;GreatWar;World War;People are dying all over the world.;0.95;0.90;0.40;0.5
0;600;700;BillNye;The planet is not round!;Some idiots are claiming the Earth is flat and have caused quite some havoc among the people.;1;1;0.95;1
0;600;900;Disaster;Warning! Toxic!;Toxic waste is being handle irresponsibly. Organisations are outraged!;0.8;0.7;0.8;1
0;600;1000;AtomicBomb;The Atomic bomb;A country has harnessed the power of the atom and developed and used a nuclear bomb in combat. ;0.6;0.8;0.5;0.6
0;700;1000;Disaster;Nuclear meltdown;Do you know Chernobyl? They've made the same mistake.;0.7;0.9;0.7;0.75
0;700;1000;Protection;Prevention!;People have started using condoms as a means of birth control.;1;1;1;0.95
0;700;1000;monster;Genetically Modified Monsters;Uhm, we've created something -- and it's kind of wrecking cities and its inhabitants. Woops!;1;0.9;0.9;0.9
0;750;1000;Monkey;Planet of The Apes;Looks like it wasn't a film after all.;1;1;0.5;0.6
0;800;1000;Disaster;Terrorism!;Terrorism is on the rise and is causing havoc in your cities.;1;1;0.7;0.8
0;800;1000;Disaster;The River of Blood;A massive fire at an industrial complex has released toxic agrochemicals into the air and resulted in tons of pollutants entering the closest rivers, turning it red.;0.6;0.75;0.7;0.95
0;850;1000;Disaster;Blimey, riots!;Strict politics have stirred emotions and riot outbreaks are on the rise.;1;1;0.7;1
0;850;1000;Exams;Exams, exams everywhere;We exceeded expectations just by turning up for the exams!;1;1;0.7;1

0;900;1000;Cocaine;Crack 'n Caine;Cocaine is like really evil coffee. Your species have legalised the use of drugs.;1;1;0.3;0.8
0;950;1000;AlienInvasion;Alien Invasion;Aliens have invaded and terraformed your planet to mine valuable ores.;1;0.5;0.7;0.4
0;950;1000;IonCannon;Ion Cannons;Fanatics have managed to seize control of the D.O.D.'s Ion Cannon and have obliterated an entire country based on race.;0.8;0.7;0.6;0.65
0;990;1000;Matrix;Has science gone too far?;Scientists have conducted research towards the meaning of life. People have realised they are in a simulation.;1;1;1;0.05
0;981;1000;DoIt;The Empire strikes back;The Death Star has destroyed the entire planet! => Simulation over.;0;0;0;0

Breakthrough header:

`1;start;end;"sound_file_name";"title";"message";tech_index`

1;0;400;Breakthrough;Understanding of the menstruation cycle;People now fully understand the menstruation cycle -- Population is booming!\n(+1 Medicine);0;;;
1;0;600;Breakthrough;Organism Dissection;After long discussion, the people have decided that species dissection is not immoral and for the greater good.\n(+1 Medicine);0;;;
1;200;800;Breakthrough;Jungle species;A special kind of species that is immune to many diseases has been discovered in a jungle.\n(+1 Medicine);0;;;
1;500;700;Breakthrough;On The Origin Of Species;A theory of Evolution?! The basis to the modern evolutionary synthesis.\n(+1 Medicine);0;;;
1;500;800;Breakthrough;Anesthetic;The discovery of anesthesia has improved overall medical care.\n(+1 Medicine);0;;;
1;500;1000;Breakthrough;Morphine;Friedrich Serturner has invented morphine.\n(+1 Medicine);0;;;
1;600;1000;Breakthrough;Penicillin;Alexander Fleming has helped create penicillin.\n(+1 Medicine);0;;;
1;700;800;Breakthrough;Heart Transplant;The first heart transplant has been achieved.\n(+1 Medicine);0;;;
1;750;1000;Breakthrough;X-ray;X-rays have proven useful in the analysis of many injuries.\n(+1 Medicine);0;;;
1;750;1000;Breakthrough;Prosthetics;Advanced prosthetics have been discovered.\n(+1 Medicine);0;;;
1;800;1000;Breakthrough;DNA;DNA has been discovered!\n(+1 Medicine);0;;;
1;900;1000;Breakthrough;Augmented Species;Augmentation to the organism body has increased overall health and performance.\n(+1 Medicine);0;;;
1;0;400;Breakthrough;Banana;Your species have discovered bananas and who doesn't like bananas!\n(+1 Agriculture);1;;;
1;0;500;Breakthrough;Fertilizer;Fertilizer has been discovered.\n(+1 Agriculture);1;;;
1;0;500;Breakthrough;Potato;The potato plant has been discovered.\n(+1 Agriculture);1;;;
1;0;500;Breakthrough;Crop rotation;Apparently the soil prefers multiple kinds of crops. Interesting.\n(+1 Agriculture);1;;;

1;0;600;Breakthrough;Irrigation;By irrigating the fields, agriculture has improved.\n(+1 Agriculture);1;;;
1;400;600;Breakthrough;The Spaghetti Tree;People have tried making a spaghetti tree -- they failed -- but still.\n(+1 Agriculture);1;;;
1;400;800;Breakthrough;Machinery;Machines were made to help plow the fields!\n(+1 Agriculture);1;;;
1;550;700;Breakthrough;Benjamin Holt;This man invented the tractor!\n(+1 Agriculture);1;;;
1;600;700;MomsSpaghetti;Mom's spaghetti;Knees weak, arms are heavy, Agriculture +1 already.\n(+1 Agriculture);1;;;
1;800;1000;Breakthrough;Plant genetics;Using DNA to genetically modify plants.\n(+1 Agriculture);1;;;
1;800;1000;Breakthrough;One-meal tomatoes;A new kind of tomato has been made that contains enough nutrients to replace an entire meal!\n(+1 Agriculture);1;;;
1;0;150;Breakthrough;Modern Pyramids;In an attempt of Renaissance for Egyptian culture, pyramids have been built.\n(+1 Architecture);2;;;
1;0;200;Breakthrough;The Mixture;People are now able to build houses cheaper and more efficient.\n(+1 Architecture);2;;;
1;0;300;Breakthrough;Pantheon;Your species have renovated the great Pantheon.\n(+1 Architecture);2;;;
1;0;300;Breakthrough;Statue of Zeus;People have turned towards ancient gods and have built a great wonder for Zeus.\n(+1 Architecture);2;;;
1;0;1000;Breakthrough;Great architect;A great architect has been born, boosting technology.\n(+1 Architecture);2;;;
1;0;200;Breakthrough;Reinforced Concrete;Someone mixed sand, water, steel and more, and got an awesome building material.\n(+1 Architecture);2;;;
1;100;500;Breakthrough;Measuring the World;The new sciences have given us a number of different ways to describe and measure the world around us.\n(+1 Architecture);2;;;
1;100;400;Breakthrough;Visionary Thinkers;Our world is a bastion of intellectual thought and our universities produce some of the most visionary thinkers ever when it comes to materials and construction.\n(+1 Architecture);2;;;
1;400;800;Breakthrough;Wine Cellar;People are now able to build underground! All hail decadence!\n(+1 Architecture);2;;;
1;500;800;Breakthrough;The Statue of Liberty;The Statue of Liberty has been constructed.\n(+1 Architecture);2;;;
1;550;700;Breakthrough;Urban Planning;Cities are now constructed with regard to efficiency and beauty.\n(+1 Architecture);2;;;
1;600;750;Breakthrough;The Eiffel Tower;The Eiffel Tower has been built in Paris.\n(+1 Architecture);2;;;
1;600;1000;Breakthrough;Kim Ragaert;Kim Ragaert has been born, instantly boosting Architecture through material sciences!\n(+1 Architecture);2;;;
1;800;1000;Breakthrough;Burj Khalifa;One of the tallest buildings has been constructed.\n(+1 Architecture);2;;;
1;980;1000;jedi;The Force;The Jedi Temple has been discovered.\n(+1 Architecture);2;;;

1;0;250;Breakthrough;Leonardo Da Vinci;Being a master in many subjects of science and art does have its perks.\n(+1 Engineering);3;;;
1;0;300;Breakthrough;Mathematics Boom;Your species have been uncovering many of nature's secrets and have developed a new art to describe them.\n(+1 Engineering);3;;;
1;0;1000;Breakthrough;Astronomy;Look up -- the Stars await us.\n(+1 Engineering);3;;;
1;300;450;Breakthrough;Galileo Galilei;This person has played a major role in the contributions of observational astronomy.\n(+1 Engineering);3;;;
1;300;500;Breakthrough;Apples and Isaac;Isaac Newton's Principia formulated the laws of motion and universal gravitation.\n(+1 Engineering);3;;;
1;300;1000;Breakthrough;Scientific Experimentation;While it is true that we stand on the shoulders of our ancestors, it is high time that we start to learn about the world from our own, well documented observations.\n(+1 Engineering);3;;;
1;400;700;Breakthrough;Printing Press;Printing has been made easier to spread more wisdom.\n(+1 Engineering);3;;;
1;500;1000;Breakthrough;We've hit oil, boys!;A large reservoir of oil has been found. James Young -- where are y'at?\n(+1 Engineering);3;;;
1;500;700;Breakthrough;The Education Act;Henceforth education is compulsory for those who are to govern our legal system.\n(+1 Engineering);3;;;
1;550;950;Breakthrough;Chemical Wizard;Chemistry has opened many new doors: a whole array of new molecules and compounds are now being developed.\n(+1 Engineering);3;;;
1;600;800;Breakthrough;The Transistor;The transistor has been developed. Immense progress in computing has been announced!\n(+1 Engineering);3;;;
1;600;1000;ExtraBreakthrough;A great Engineer;Wim has been born! A great day for Engineering!\n(+1 Engineering);3;;;
1;800;1000;Breakthrough;The Internet;The Internet:Lolcats and memes for days! Oh, and sharing information.\n(+1 Engineering);3;;;
1;950;1000;ExtraBreakthrough;Nanotechnology;Excuse me, where is the nanotechnology department? - You just stepped on it.\n(+1 Engineering);3;;;
1;900;1000;Breakthrough;Pizza;Your organism has created the recipe for pizza -- what a great day for your organism! Engineers wouldn't survive without pizza.\n(+1 Engineering);3;;;
1;900;1000;ExtraBreakthrough;Boris Boreau;The sandals make the man! Or something along those lines.\n(+1 engineering);3;;;
1;900;1000;Breakthrough;Quantum Computing;"""If you think you understand quantum mechanics, you don't understand quantum mechanics.""\n(+1 Engineering)";3;;;
1;900;1000;Breakthrough;Nuclear Fusion Tech;At long last, the power of a star is within our grasp.\n(+1 Engineering);3;;;
1;950;1000;Breakthrough;FTL? FTL!; Light may be fast, but it finds the darkness has always got there first, and is waiting for it.\n(+1 Engineering);3;;;
1;999;1000;ExtraBreakthrough;Future tech;Mass relays are feats of Prothean engineering advanced far beyond the technology of any living species.\n(+1 Engineering);1;;;