Detailed documentation for the Exo3D project

This document explains all the complex mechanisms we implemented into the project, including the reasoning needed to come to these solutions.

Animation : how to move the objects

The obvious choice to make several objects move independently would be to have a time-based simulator. Each object gets a parametric equation to describe its movement and nothing else than time is required. This also would allow very useful features such as rewinding the simulation at any point in time or placing the planets exactly where they are supposed to be at a certain time. Unfortunately, this is highly inefficient from the Babylon perspective, and results in either an enormous imprecision or needing a really large memory.

The reason of this is the fact that Babylon cannot give a parametric formula to an object and expect it to move exactly like an ellipse. You have to give a certain number of points and tell the engine that you want the object to change its position every X milliseconds. This also conflicts with the need of manipulating the emulation speed, and the objects' movement is terrible in this system because it can only teleport, so there is no continuous trajectory. As you can read, parametric solution led us to a lot of bigger issues and we could not afford to fix them - it would have taken too much time and concessions. It is still nonetheless the "right" solution, so maybe in the future if the need of a parametric solution comes in, we will consider it again.

Instead, what we choose to do is rely on a much precise and malleable system : the Babylon Animation. This object includes all sorts of tools that fit almost everything we want by default : precision with a low amount of data, continuous movement and easy speed control for all objects. Here is how it works : give any number of keys to an animation (a key consists of a frame and its associated value : position in our case ; see figure 1) and launch the animation. Babylon will automatically interpolate between those points and the object will move smoothly - without teleporting ! - between each of these points, looping infinitely. We don't have to do anything else, not even manage the time flow (see figure 2).
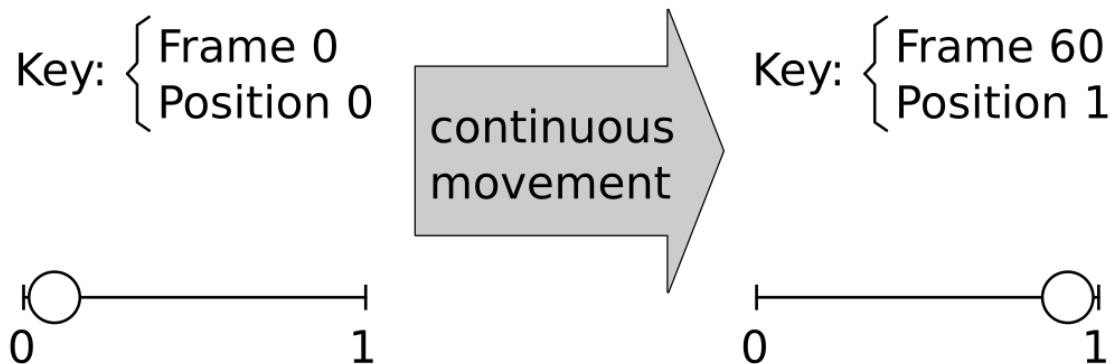

Figure 1: Animations between keys

But by default, Babylon interpolates linearly, so with a defined number of points it will have a polygon-shaped movement, like the diagram above. Once again, Babylon has a tool for this : non-linear interpolation is possible if you give to the key the tangents of the point in addition to the frame and the position. It does the interpolation automatically, so there is nothing further to do. The default interpolation corresponding to our parameters is the Hermite one, and is provided by the Babylon library (see figure 3).
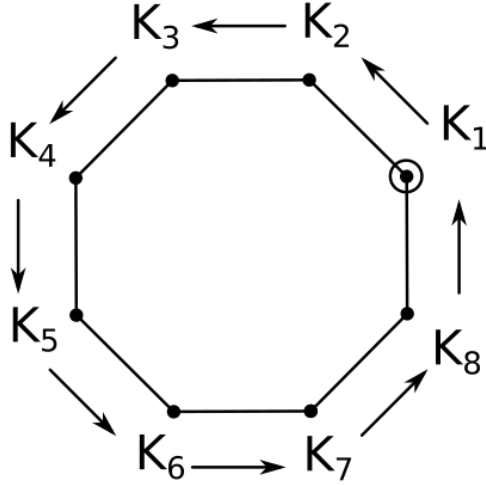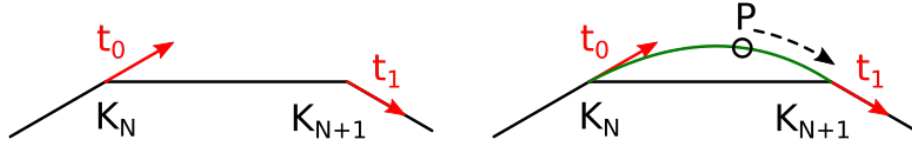
Figure 2: Looping

Figure 3: Interpolation between keys

What we obtain is a quite accurate movement between two points, and once applied to a few hundred points around the trajectory of the movement, the result is so much closer to an ellipse than the parametric solution that we had to keep the animation by keys. Last side note about the animation system : once any animation is launched, it grants itself a `speedRatio` parameter, which tells how it fast it should animate. Given keys at frame 0, 30 and 60 (for 30 frames per second), a normal `speedRatio` of 1 will make the animation loop every 2 seconds. Make it 2 instead and it will go twice as fast - a 1-second loop, then. We just need to stock all animations in an array, and cycle through it to manipulate all `speedRatio` at once.

Trajectory : calculations and 2D-solutions in 3D-space

Before we can move the objects, we need to know where they are supposed to move. The vast majority of all planets follow elliptic trajectories (the exceptions consisting of three-body problems and other chaotic solutions, which are not our concern for this project). But there are a few different ways to calculate an ellipse and only one of them is appropriate for our problem : the focus-centered ellipses. The reference for the origin is always the left or the right focus of the ellipse (in our case, we arbitrarily chose the left focus), and the position is calculated with polar coordinates : the distance r between the object and the focus, and the true anomaly ν which represent the angle between the object and the apoapsis of the trajectory (see figure 4).
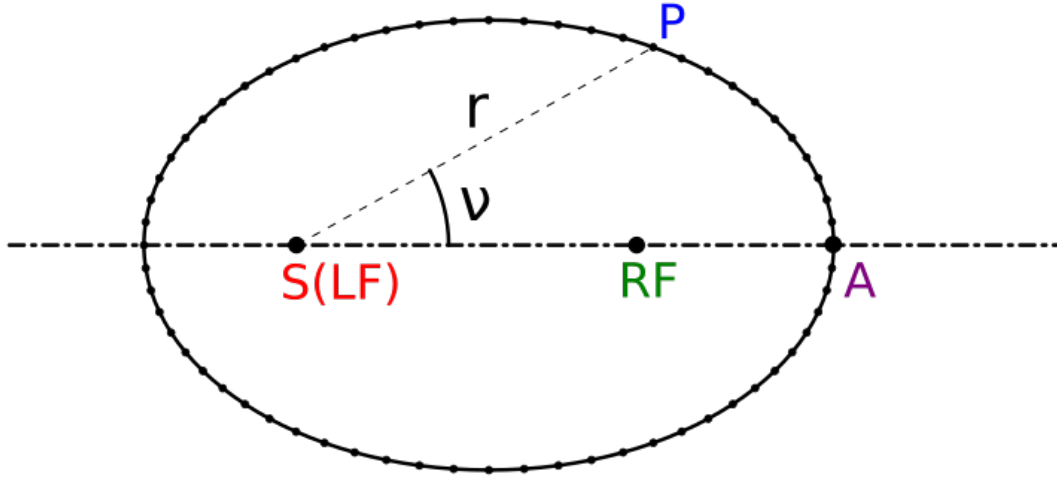

Figure 4: Trajectory centered on the left-focus

The distance r is given by the following formula :

$$r = a\sqrt{(\frac{1 - e^2}{1 + e\cos(\nu)})}$$

where :

- a is the semimajor axis of the trajectory
- e its eccentricity
- ν the true anomaly.

Because a and e are always known and fixed for any ellipse, the only parameter is ν. We do have its exact formula, but I have to be honest here : I do not understand half of it and I am more a developer than a physicist. To implement it, I would need much more knowledge on how it works, so I can translate it accurately from a harsh mathematical formula to proper lines of code. The ν formula also depends on the time, which we already decided to exclude from our solutions. We may choose to finally use this formula in the future, but as for now we consider that ν grows linearly, and therefore each point is equally separated from each other in terms of angles.

This approximation is not really that inaccurate, because when the object approaches the apoapsis in this formula, it goes faster (and slower when approaching the periapsis). If we choose the star as the left focus,

the formula is right but the movement is wrong : planets will accelerate when being far from the star and decelerate when approaching it - and the correct behaviour is the opposite ! Well then that's easy : the star can be the right focus of the ellipse. Yes, this is pure cheating but mathematically this is still right. Sorry to all astrophysicists, please do not throw rocks at me, we had to keep the simulation easy to develop.

As we mentionned it in the animation part, the animation only need a few hundred positions. We need points for a closed trajectory, so ν goes from 0 rad to 2π rad. Therefore, X points are separated from each other by an angle of $\frac{2\pi}{X}$, as you can see in the diagram above.

But this diagram still shows a trajectory in a plane. This formula implies that the object's position is on the plane formed by the focus, the apoapsis and itself. If converting polar coordinates to Cartesian coordinates is quite simple in a plane, it is much harder to convert from 2D polar coordinates to 3D Cartesian coordinates. We would need more angles, complex trigonometric functions and maybe matrix manipulation. Instead, we cheated a bit - once again - with what you will read in the very next part.

Inclination on the ecliptic plane and self-inclination

In our solar system, the plane formed by the Sun, the Earth, and its apoapsis is considered the reference for placing objects in the system. We are then flat (talking about the plane, not the Earth, begone flat-earthers !) and everything else is inclined. This plane is called the ecliptic, and I will refer to this term a few times even for outer systems. I am then acknowledging our plane can be transposed anywhere and should be the reference.

The vast majority of planets do not have their plane in common, actually they all have different inclinations, so they would all need Cartesian coordinates in that 3D-space. But what if we could still fake being in a plane for all planets ? This is possible thanks to the concept of parenting in OOP. When you create an object, it has some attributes, maybe methods, and it is independent of anything. Although you can choose to give a parent to that object, and it will inherit its attributes and methods. Not to make a course on OOP, but just to give a bit of context on what we will talk about.

The important thing to know here is how parenting objects - and more specifically, the mesh of the object - works really well in our case. In Babylon, giving a mesh "A" another mesh "B" as parent make it become its new reference. If "A" has (1, 0, 0) for coordinates, but "B" has (2, 5, 5), then "A" has for *absolute coordinates* (3, 5, 5). And even more interesting, applying any rotation on "B" is reflected on its whole plane (X, Z) ! In practice, the "B" object will have the coordinates of the center of the system - most probably the star - then take the value of the inclination to the ecliptic of the "A" object and apply a rotation on the X-axis of the "B" object (Z-axis would work as well but either of those two options is still missing an angle which we will not talk about here : the argument of periapsis). The result is our elliptical trajectory, still in 2D, but "tilted" to match its inclination in space. No further calculation is needed, just the value of the inclination to the ecliptic plane (see figure 5). It is just that easy, and honestly it feels like cheating but it definitely works.
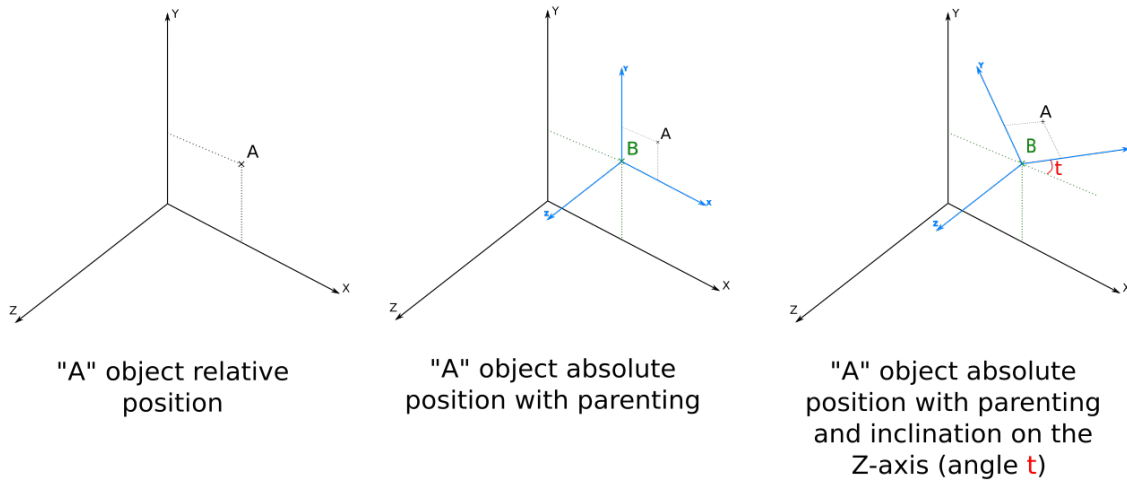

Figure 5: How parenting meshes affect their position

Self-inclination is how much the planet is inclined on its own X-axis (or Z-axis on the diagram below, X or Z does not matter that much). At first I tried to simply apply that inclination to the mesh of the object, but it conflicted with the fact that all objects revolve around themselves (on their Y-axis). If you keep it

that way, the mesh of the object will revolve around the Y-axis of its own space (which isn't rotated). To give an example, take a spinning top, tilt it a bit on the side and make it spin. Will it spin exactly around its center axis ? Not at all, instead it will spin around the Y-axis of the ground and have a weird-looking movement. That is exactly what our meshes do when going through this method (see figure 6).
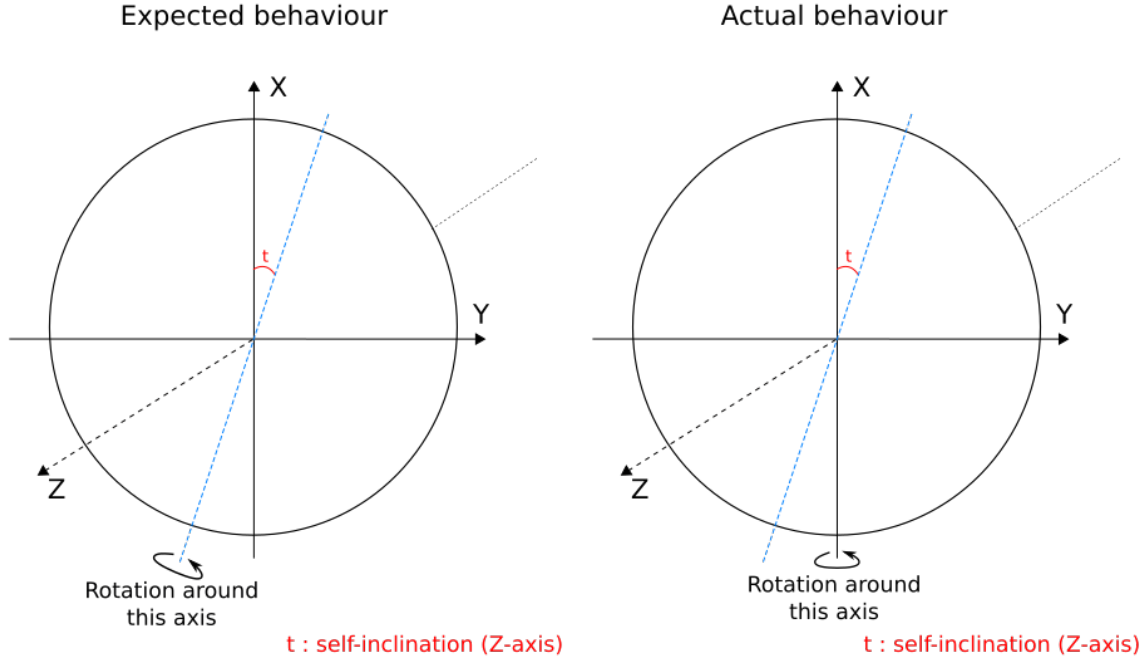

Figure 6: Expected behaviour and actual behaviour

This is why we need another parent, following the object this time. That parent will take the inclination, and the actual object will be tilted accordingly. Applying the rotation on the Y-axis will no longer cause issues and will give the expected result above. Because we then have two layers of parenting, we have to visualize where exactly is our object and what it does. The highest parent does not move and takes the inclination to the ecliptic, the lower parent has to move alongside the planet, so it carries the movement in place of the object (in addition to the self-inclination it has) and the object itself only revolves around its Y-axis (see figure 7).

Tilting planes is essential for our objects to behave normally, and it avoids atrocious calculations and matrix conversions. Although do not try this at home, you would not like to fall at 45° on the ground.

Highest parent

Lower parent

- center of the system
- invisible planet parent and its plane
- object child in the plane of its parent

- planet itself, Y-axis in red
- another invisible object, parent of the planet, following the trajectory and inclining the planet
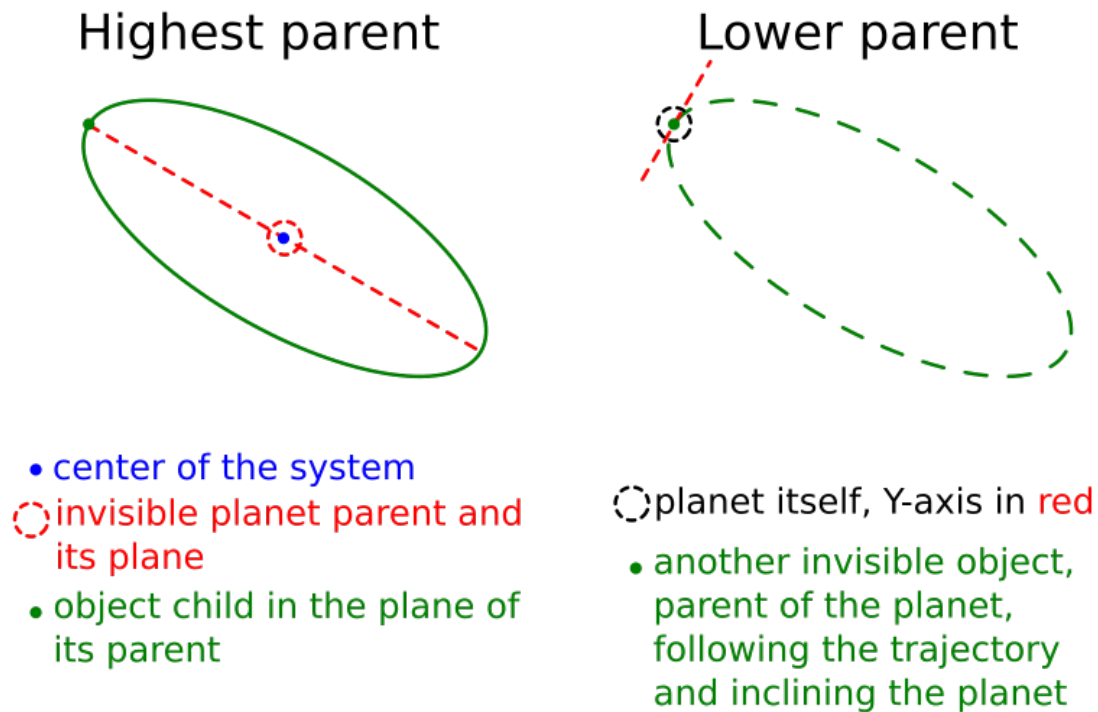
Figure 7: Parent hierarchy and roles

Camera placement and targets

Three types of cameras are available in the simulator : the system camera, the planet camera and the free camera. Each of those serves a different purpose, and accordingly needs different treatment. The system camera is centered on - you guessed it - the whole system, most likely the star. By default, it is far enough from its target to see every trajectory in the system, but you can zoom in to take a better look at the star. Dragging the mouse allow you to move around the star as if you were attached to it by a cable, in other words you can move in a sphere (whose diameter is your distance to the star). The planet camera has the same type of movement, but it has any planet for target. You can change which one you want to look at with the UI. And the free camera can move everywhere and has no particular target (see figure 8).

There used to be a transition system, which made the immersion sharply better, but it was totally incompatible with the previously mentionned parenting system. The type of camera we use for the system and the planet (called ArcRotateCamera) needs a specific position for its target. It isn't impossible to use the same trick as we did above, by including the camera in the plane of the highest parent. But this means we are literally changing planes when transitioning from an object to another, which brings in all the bad calculations we wanted to avoid. This is an absolute calamity to handle, and I never want to hear the concept of transition between planes, ever.

So as I mentionned it, we actually use the same system of parenting for planet camera (system camera doesn't need it because its plane is not inclined). To make the spectator feel the inclination of the planet, the camera spawns by default exactly parallel to the ecliptic but still pointing at the planet. Going from a planet to another also means changing planes, but this is fine because it is immediate so there are no calculations, just changing the parent of the camera instantly.
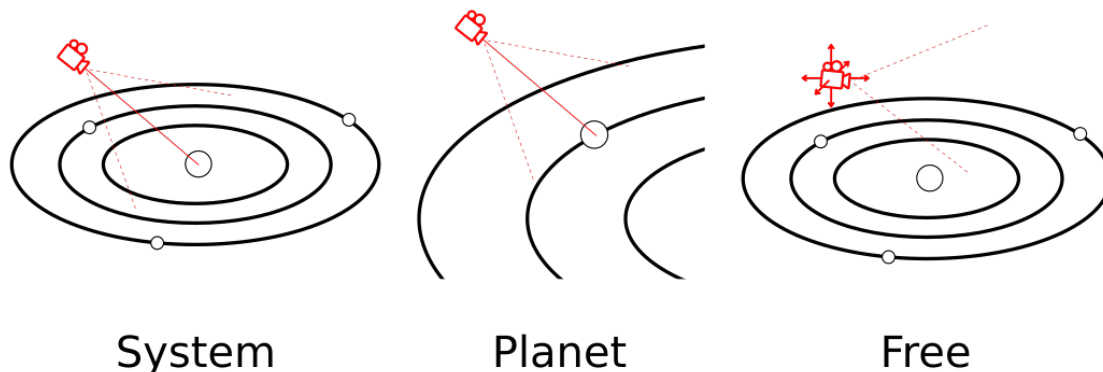
Figure 8: Cameras

Object scaling

About the placement of camera, it is quite nice to have different views on the whole system... but we are missing one big problem : everything is scaled correctly by default. Just think about it, can you see a 1 centimeter ball from a 118 meters distance ? That is exactly the kind of ratio we are talking about when trying to see the Earth from the Sun. Objects in space are so ridiculously far from each other compared to their size that it is impossible to see them in the same view, which is pretty disappointing. The biggest exoplanet we ever found - as I am writing this - is about 80 times the size of the Earth, which is still smaller than our Sun (~100 times the diameter of Earth). In a universe where we speak in terms of astronomical units or even light-years, seeking planets become a real challenge.

It can be really useful to represent planets in a neighbouring space, where you really can have a good idea of the size of everything. That is genuinely one of the most important goals of this project and we took a moment to find a solution to this problem. How can we scale up objects to see them at an interesting, didactic view ? Well, the first important thing to keep in mind is that we want to scale objects at a higher size but they shall keep their proportions relative to each other - i.e. Jupiter staying 11 times bigger than the Earth, among many. The challenge here is to scale up every object as much as possible, without them colliding. We can achieve this by comparing planets' diameter and their respective orbit. By calculating the smallest distance between them, we can then enlarge these two planets by the same factor until their surfaces eventually collide. If any two planets of the same radius of 5 000 km, are at minimum far from each other by 1 million kilometers, then we can deduce that the wanted ratio to make them touch is $\frac{1000000}{5000+5000} = 100$: each planet can be 100 times larger (see figure 9).

The general formula for this is as follows :

$$ratio = \frac{d}{r_1 + r_2}$$

where :

- d is the smallest distance between the two orbits
- r_1 and r_2 are the radius of each planet.

This formula is applied to every planet and their direct exterior neighbour, and the smallest value obtained will be the ratio chosen for the entire system (except the star). With this solution, two and only two planets touch each other, while the other planets don't. To avoid those two planets from touching each other - and to keep the view somewhat comfortable - we simply multiply this ratio down to about 40-50% of its value. This results in a really pleasant view of the system, where you can see everything without zooming for 1
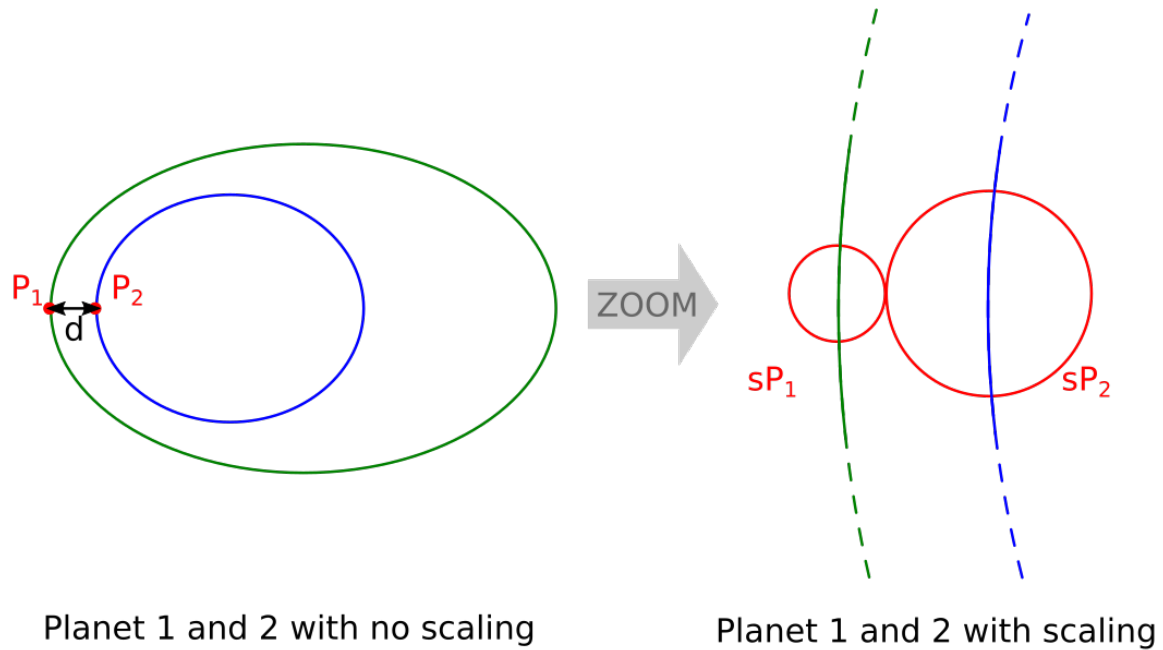
Objects scaling:

Figure 9: Scaling up two planets of different sizes

hour long. But what about the star ? That is a particular case, because the star is always larger than its planets (much larger actually) and if you apply that same ratio to the star, it would most probably consume two or three planets - that is the case in the Solar System, with this formula and the ratio of the planets, the Sun devours Venus, Mercury and the Earth. This is why we calculate the ratio of the star afterwards, once we already scaled up every planet. In that case, we apply the same formula to the star and the first planet of the system but with its scaled up radius. And instead of 40-50% for the star, we put it at 90% because we want the star as large as possible. With its own ratio, the star may appear smaller than some of its planets - Solar System again : the Sun is smaller than every giant planet with this formula.

About the cameras and handling of that change of scale : cameras have a defined minimum radius which they cannot go under, otherwise they would be too close to the object they are looking at (that ratio is often 3 times the diameter of the object). Then, we also have to keep in mind that the cameras have to eventually zoom out of their object, otherwise they could be stuck inside it if it grows too much. Basically :

- the camera zooms out if its current distance is not enough to be outside the object
- the camera doesn't zoom out if it is far enough
- the camera always zooms in when the object scales down

And the zoom in/out preserves the visual ratio of the object you are looking at : if you suddenly change the size of the planets when looking at the Earth while it took a quarter of your screen, the scaled-up Earth will still take a quarter of your screen, though you may now see Venus or Mars around it where you previously didn't.

Ratios in the Solar System

The default unit we are using is the Babylon unit. We chose to make the diameter of the Earth equal to 1 Babylon unit, and scale everything else based on this. This implies really high distances for the engine (we are talking in millions of units) and at first we were unsure that the engine could even handle it. The problem is that if we take the problem from the other side - by taking huge distances as a unit, like 1 AU = 1 Babylon unit - this creates tiny objects, going from 5×10^{-8} to 0.01 units. The Earth is at a microscopic 0.008 and already makes the scene go completely wrong : skybox freezing, incorrect proportions and wrong camera placement. This is why we take the planets as reference and not the distances.

For objects, the diameter is then calculated in "Earth diameters", which only means if the planet is 10 times larger, it will have a diameter of 10 units (in realistic view anyway, as didactic view will scale it up as we already mentionned). For distances however it is more interesting to use AU (which have a defined constant, approximately equal to 11 700 units) and then tell the distance in multiples of this unit ; I made the simulation convert AU into Babylon units anyway. Here you can see the detail of the different ratios used in the Solar System.

Planet / Object	Diameter (in kilometers)	Semimajor axis (in kilometers)	Diameter (in Babylon Units)	Semi-major axis (in AU)
Sun	1 392 684	None	109.179	None
Mercury	4 878	57 909 050	0.383	0.387
Venus	12 100	108 209 500	0.949	0.723
Earth	12 756	149 597 887	1.000	1.000
Moon	3 474	384 399 (to the Earth)	0.273	0.00257
Mars	6 792	227 944 000	0.533	1.523
Jupiter	142 984	778 340 000	11.209	5.202
Saturn	120 536	1 426 700 000	9.449	9.537
Uranus	51 118	2 870 700 000	4.007	19.189
Neptune	49 528	4 498 400 000	3.883	30.070

To give an idea of what kind of distances we are talking about, Neptune and its 30 AU from the Sun represents about 351 800 Babylon units, which was enough to cause severe rendering issues - nothing unfixable fortunately, we just had to implement an algorithmic depth buffer. Now I'll give this information : the largest orbit recorded for any exoplanet is around 9 900 AU, more than 100 000 000 Babylon units. This is a tough challenge to represent this kind of unbounded distances, but that is the whole point of the project : showing to everyone how vast the universe is besides our little planet !

Time in the simulation

Enough talking of easy 3 dimensions problems, let us be real physicists and add a fourth dimension : the time. As we already told it in the very first part, time flow is something difficult to manage in our simulation. We are using Babylon's animation system, which provides *timelines* for each object. This means every object can be slowed down, stopped, or rewound to any point of its trajectory (in theory only, we didn't try this yet but Babylon provides the means to do so). There isn't a global clock, checking when is every object, instead they all have their own clock. While Mercury is at 10% of its 3rd revolution since the beginning of the simulation, Venus would be only at 82% of its 1st revolution, the Earth at 50% and Neptune at a measly 0.3% (see figure 10).
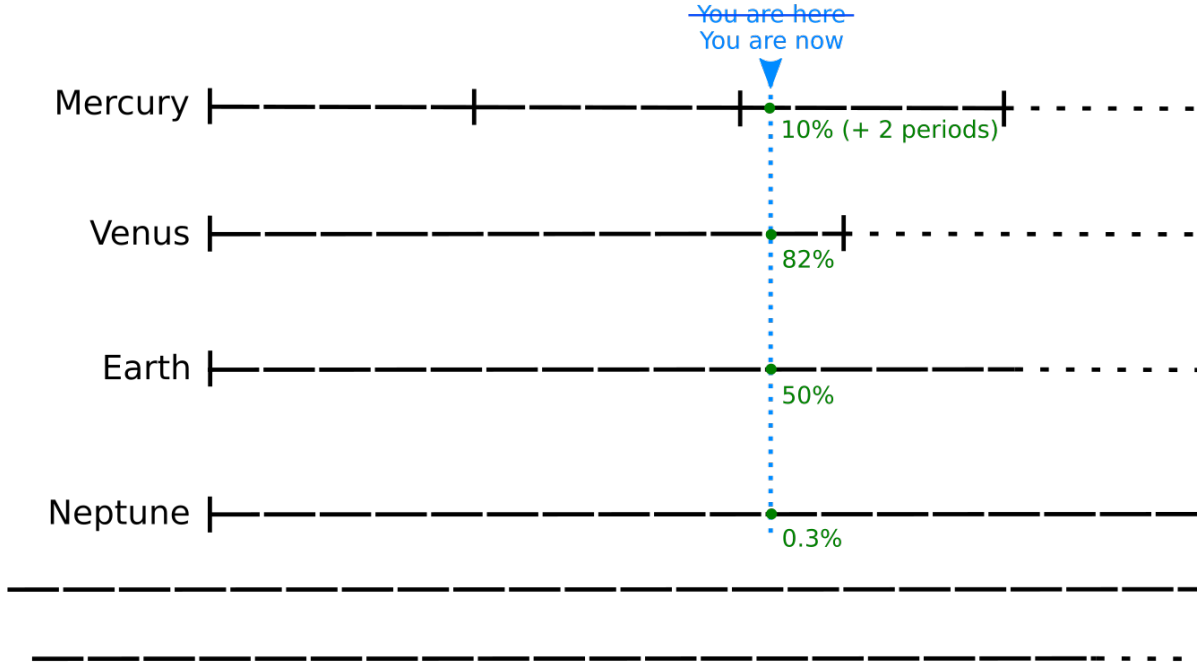

Figure 10: Different animation timelines in parallel

Those different periods all coexist nonetheless, and by putting all the animations and their timelines in an Array, it is possible to affect all timelines at once (for example by increasing or decreasing the emulation speed). By default, the only speeds allowed are 0 (pause), 0.5 (slowed), 1 (normal, default) and 2 (accelerated). This brings in a new issue : as we saw just above, planets can have extremely different periods of revolution. Let's say we want Mercury to revolve in 5 seconds. This means Venus will revolve in about 12.5 seconds, Earth in 20.5 seconds... and Neptune in 3420 seconds (57 minutes). If you put it the other way - Neptune in 5 seconds - then Mercury will revolve every 7 milliseconds. We can't cover such a large difference of periods with only 0 to 2 times speed. And as we can't know how long can be a period, we can't choose a random high value. Also, we don't want to let people figure out that the right speed to see Saturn move at the bare eye is 12 200%.

Planet / Object	Revolution time (in days)	Self-revolution time (in days)	Revolution time (relative to Mercury)	Self-revolution time (relative to Mercury)
Sun	None	26.919	None	0.306
Mercury	87.969	58.846	1	0.669
Venus	224.667	-243.023[1]	2.554	-2.763[1]
Earth	365.256	0.997	4.152	0.011
Mars	686.885 (1.88 years)	1.026	7.808	0.012
Jupiter	4 332.01 (11.86 years)	0.414	49.245	0.0047
Saturn	10 754 (29.44 years)	0.448	122.248	0.0051
Uranus	30 698 (84.05 years)	-0.718[1]	348.964	-0.0082[1]
Neptune	60 216 (164.86 years)	0.671	684.514	0.0076

The solution to this problem lies in relativity. Here you saw the periods relative to Mercury, which is the reference at the start of the simulation. Mercury is what we could call a "time-reference planet" (let's say TRP), the TRP defines how fast should move the other planets and always revolve around the star in 5 seconds. And we can change that TRP at any time with the UI : you can choose to make every object go faster or slower, to keep up with a new TRP. If the Earth is the TRP, then it needs 5 seconds to revolve while Mercury is now at around 400% speed and revolves in 1.2 seconds.

Finally, we find it interesting to show in the simulator how much time has passed - this is notably frightening when you see Neptune revolve once and literally one century and a half is gone. We then need to effectively check how much time passed thanks to some recursive timeouts. First, we call a Timeout every X seconds, and we tell it that the TRP is Mercury. Knowing this information, the Timeout can check how long is the real period of Mercury and update the number of days simulated. Whenever the TRP changes, the Timeout adapts its calculation and updates the day at the new rate, based on the information of that new TRP. For every X seconds passed, the Timeout updates the number of days following this formula :

$$d_{i+1} = d_i + \frac{s_{cur} * rev_{P1}}{t_{sim} * X}$$

where :

- d is the number of days
- s_{cur} is the global speed ratio of the animation
- rev_{P1} is the revolution period of the first planet (in days)
- t_{sim} is the number of seconds defined for the period of the TRP
- X is the Timeout duration in seconds.

In practice, X is a parameter given in milliseconds and the formula is adapted accordingly, but the result is the same.

---

[1]Minus values for self-revolution are for planets with a self-inclination superior to 90°, in which case they are not rotating in a prograde motion but a retrograde motion instead.

JSONs and dictionaries

For the longest time, we used custom local objects to store the properties of the spatial objects. As the project goes on, we want to include more and more support for the rest of the exoplanet project. This is why we had to convert those local objects into proper external files - YAML files to be precise, generated with the work of Ulysse Chosson, engineer on the project. We didn't need to work with YAML, so JSON format was enough, as both can be converted to each other. There begins the adventure : make spatial object information go into a JSON file and read it from there.

The first trouble we had to face was how JSON files are handled in JavaScript. You need to wait some time before JS can give you the corresponding object, and you cannot assign it to a variable or start interacting with it until that time is over. This led us to introduce `await` and `async` support in various modules, due to them needing to interact with the information contained in the JSON. This was tricky, but not impossible, and the result allows to stop the instructions of the program while the JSON hasn't been fully accessed.

Once that step was completed, we also had to think of formatting the data. Ulysse will give us files which are built in a very special way, and from now on we have to work around that. The YAML/JSON files containing information regroup all spatial objects in a system. The first half of the file contains the raw information on these (name, status, size, trajectory, etc.) and the second half describes how they interact with each other by creating sub gravitational systems ("sg" in short most of the time). You can see an example below.
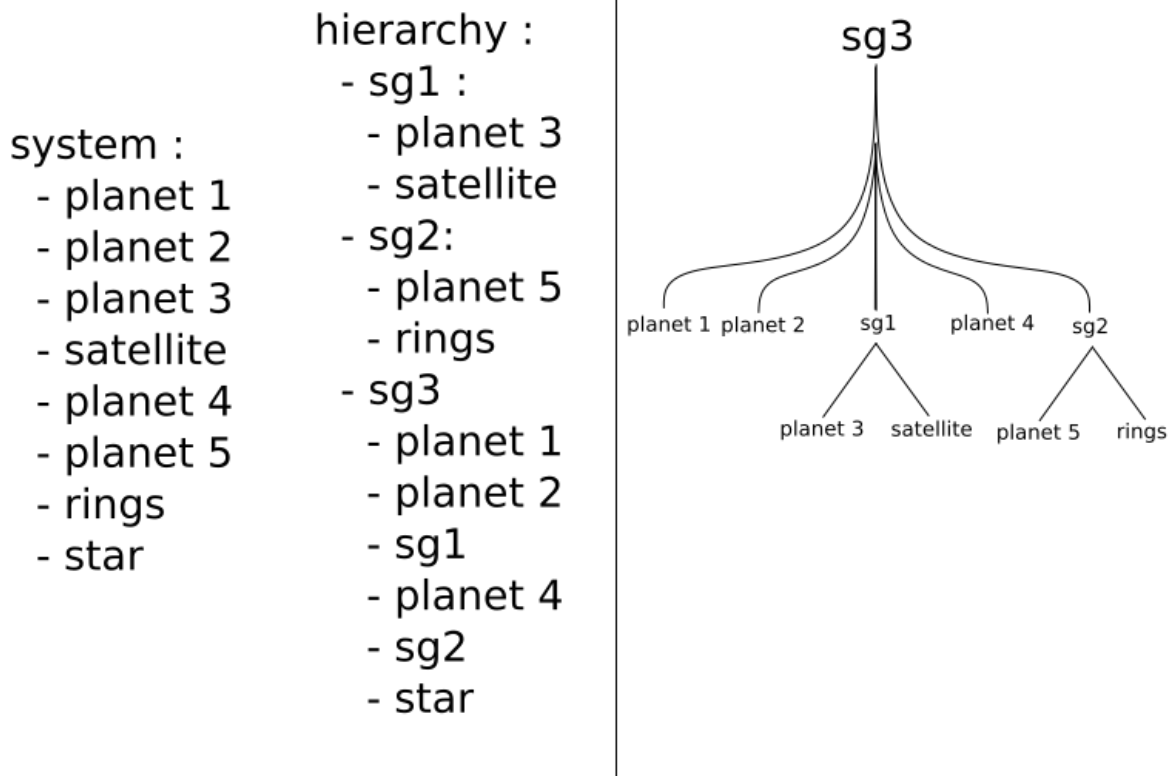
To explore this file, we look first at the main subsystem : the one containing the star(s). From there, we implement an algorithm that travels across this subsystem and tries to recognize elements in it. If the algorithm finds a spatial object, it can take its information from the 'system' part and give them to the system builder for later use. If it is another subsystem from the 'hierarchy' part, the algorithm explores that subsystem and tries to find other spatial objects. But we had a huge problem : those JSON files have their contents in lists, and that makes the 'system' part and the 'hierarchy' part really hard to connect. That would be much easier if all those elements were attributes of their respective parts, because JavaScript

has plenty of support for identifying attributes in separate locations. This is why we had to transform the raw JSON file into a dictionary version of it. And now, we are ready to go.

Three major assumptions were raised to make this algorithm work :

- The last subsystem is necessarily the one at the root of the system. It contains the star(s) and every other subsystem/spatial object
- Satellites and rings are, by definition, associated to one and only one planet. If a subsystem contains either of those two, then there is a planet in that subsystem, and that planet is their parent
- A spatial object cannot appear more than once in the hierarchy.

Those three conditions make sense in regard of the organization of a system, and they are rules applied on Ulysse's work. If everything above is respected, then the algorithm works fine and extracts correctly all the information we need to build the system.