EPITA

UNDERGRADUATE (FIRST YEAR) PROJECT

30.04.2018

2ND PRESENTATION REPORT

---

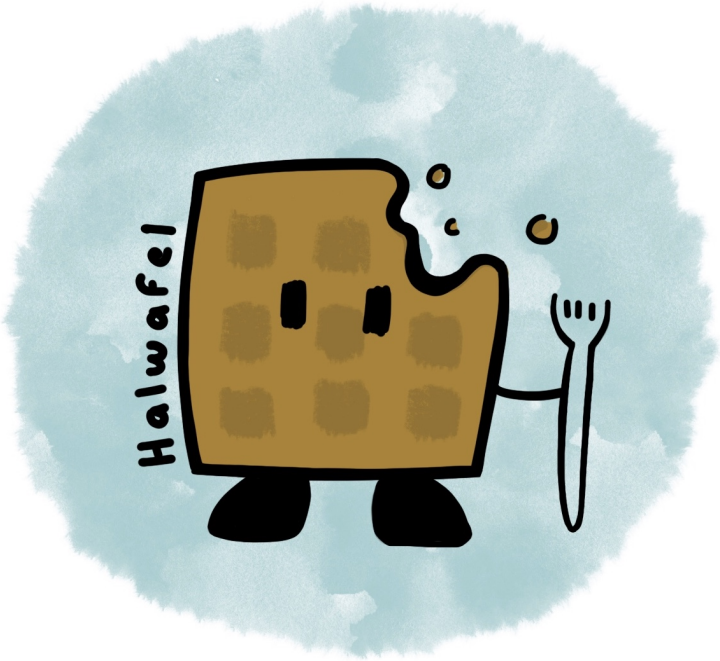
# THE FRAME WORKSHOP

---

## 3D ANIMATION SOFTWARE

HALWAFEL

Brian Bang, Etienne Benrey, Léane Duchet, Julie Hazan



# Contents

# Introduction

As a reminder, The Frame Workshop is a 3D animation software. In this report you will find the information about what has been done since the first presentation, the main concerns will be the GUI, the creation of motion and the website. Then you will find explanations about what will be done for the final presentation.

# 1 Summary of what was done for the first presentation

- Skeletons: we are now able to create manual skeletons, an attempt to do an automatic skeleton has been started during the first presentation.
- GUI: The general layout of the GUI has been set up with the main panels. A drag-and-drop and a scene manipulation methods have been implemented.
- Import: the user has to input the path to the file he wants to import in an input field. Since neither OpenFileDialog() from C# nor OpenFilePanel() from Unity were usable this was the solution we ended up with.
- Website: the website was started for the first presentation and was hosted using Github Pages.

# 2 Achievements

| **Part of the project** | **What had to be done by the second presentation** | **Done parts of the project** |
|---|---|---|
| **Mesh + Sound import** | 100% | 100% |
| **Skeleton creation** | 100% | 50% |
| **Graphic User Interface (GUI)** | 80% | 80% |
| **Creation of motion** | 55% | 60% |
| **Background imaging + sound** | 0% | 0% |
| **Particle animation** | 0% | 0% |
| **Website** | 70% | 90% |

| Done parts of the project | Brian Bang | Etienne Benrey | Léane Duchet | Julie Hazan |
|---|---|---|---|---|
| **GUI** | | Timelines<br>Option | Insert<br>View<br>File | |
| **Creation of motion** | X | | | X |
| **Skeleton creation** | | X | X | |

## 2.1 Automatic skeleton

The automatic skeleton is not done and will not be done as problems arose during the coding of the functions.

For the first presentation, we had quite a clear concept on how it was going to be made, but there were problems with the function managing the collisions with the mesh. In fact, we did not properly understand how the mesh is built. So for this presentation, we first tried to understand the concept of mesh in Unity and how they are made.

At first, after reading the documentation on the construction of the list of vertices that can be retrieved from a Unity GameObject, we thought they could be easily handled by converting them into lists of triangles in 3-dimensional space. We were quite surprised to discover that we would get very different results from what we expected, whether we tried using Unity's local to world coordinate functions, or making our own. We then later discovered that the vertices list was not constructed in the way we had initially understood when reading the documentation, as each vertex is tripled, but their placement in the list is seemingly random, within each third of the list. We were unsuccessful in finding information about how this list is constructed by Unity, and making the task much more complicated than anything we expected.

After some research on the subject, we discovered the function RayCast, but learning that it only returned the first object hit, we quickly disregarded it. By the first presentation, we had looked at RayCast, but dismissed it, and figured out that our problem came from getting the correct position to look at, and hence having wrong inputs for our functions. Through testing, we know that the functions would have worked if we had correct inputs, but this route was not working for us.

Some time after the first presentation, we rediscovered RayCast, or more specifically RayCastAll, which returns multiple intersections. We hoped that using such a function would allow us to not have to use any vertex, and simply

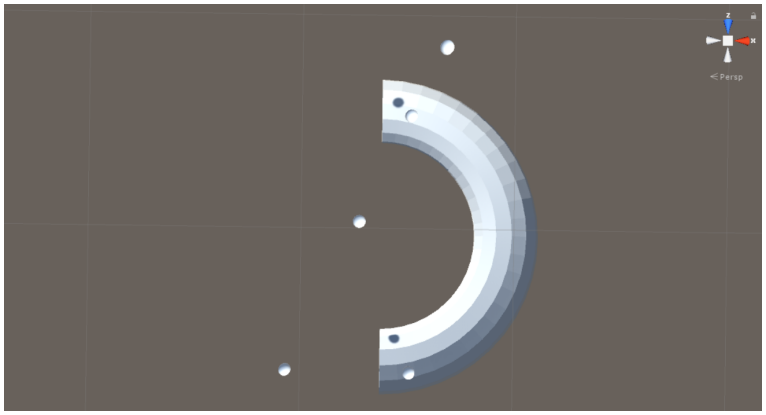
get the intersections between the mesh and the ray we wanted. This was almost perfect, but we quickly learned that RayCastAll did not return multiple points for multiple collisions with the same object. We hence had to go back to the drawing board, to find new solutions, which came in the form of using the simple RayCast function multiple times, each time starting where the last one left off. This, once again had its major flaws, as it does not detect collisions with the object in which we cast from. This means that, no matter what we do, we will be missing collisions. First, we will miss the first collision, as we start from inside the object we want to detect, and then we will miss half of the remaining collisions, as they will be coming from the interior of the object, and not the exterior.

A potential solution is to have two loops of RayCasts, that are in opposite directions, and both start outside of the object's mesh. This means that we would have to first calculate the total distance covered until we get to the first point that we want, ignoring all collisions until then. Then we need to store all the collisions until we get to the second wanted point, and finally either stop the program, or ignore the remaining collisions. This needs to be done for both directions, and then the resulting lists of collisions need to be merged, at which point we have a functional list. However, this would be very complicated to handle, without even taking into consideration that calculating the distance traveled by the ray would lack a lot of precision, as we are continually adding greater and greater error margins with each collision, which are due to the nature of floats. This method poses one last problem, which is finding the outer edges of the object, in order to be able to start the cast outside of the object's mesh. We will explain later why this was problematic. To avoid this, we could simply choose an arbitrary distance from the initial point to start from, but this could give errors when using large objects, or would have to be large enough that in any semi-complex scene, the program would have a complexity such that it would drastically slow down the program, which would render the functionality unusable, and utterly useless.

The function managing collisions not being functional means the skeleton cannot be done, as both function actually building the skeleton need the collisions with the mesh.

In addition to the function used for the collisions, there is a function that is used to get the extremities of the mesh. To do so, there is a need to find the dimension of the mesh, which yet again was not nearly as straightforward as we expected. Our first thoughts were to take the full list of vertices of the mesh and to do a simple minimum and maximum search for each dimension. As stated earlier, this could not work due to our problems with correctly retrieving the vertices' positions. Testing confirmed this, as we would some unexpected results. We hence decided to switch to a different method, which consisted in getting the dimensions of the object we were studying. Along with the center

of the object, we could get the edges of the mesh of the object. This seemed to work at first, but we discovered that getting the center of the object is impossible, since the information is not given by Unity. Using the absolute position of the object does not solve this problem, as position is never placed in the same place relative to the center of the object. We hence did not have any solution for finding the dimensions of a mesh, or the maximal and minimal outer edges of its mesh.



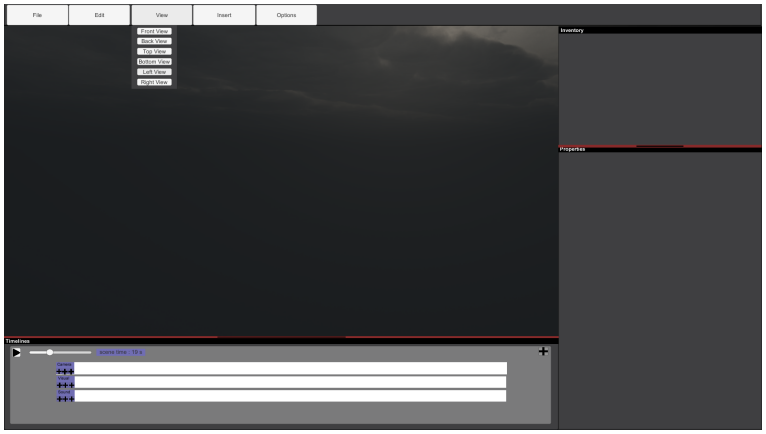
We ran out of possibilities to try and eventually decided to give up on the automatic skeleton so that we could concentrate on the other part of the project, as building a skeleton is not the most important functionality of our program.

Moreover, even though the automatic skeleton is not working, there still is a manual skeleton which is completely functional and will be used as a base for the creation of the motion.

## 2.2 Graphic User Interface (GUI)

### 2.2.1 General aspect

The general aspect of the GUI was slightly changed to make it look more appealing as well as add more functionalities to it. Furthermore, all the new elements that needed to be implemented to be able to use the new functionalities of the program, most notably the first part of creation of motion.



### 2.2.2 Inventory

The inventory panel is quite simple in its design, as its goal is simply to show the user all the objects that are currently in the scene. This is done by the manipulation of a string that is presented in a text Unity UI element. Adding an element in the scene appends its name to the end of the string, along with a new line. Deleting removes the name from the string, and shifts all names that were previously under the deleted object's name up to follow the original format of the inventory. In order to avoid having the list be cut-off due to the user having too many objects in the scene, the bottom of the panel can be dragged down to show more of the list. This will be explained later, as it was implemented alongside the Options button.

### 2.2.3 Properties

The properties panel is linked back to the timelines, as it shows the properties of any key that is left clicked (this will hence show the properties of a key that is being dragged, or simply clicked on). Releasing the left click mouse button clears the panel. The panel gives the following information : the object it is linked to, the type of key (either position, rotation or scale), the type of movement (either linear or curved), the speed curve, and the time at which it will be executed. The latter changes in real-time as the key is dragged along the timeline.

### 2.2.4 Insert

This button gives access to three other buttons : the Import button, the Load button and the Remove button.

The Import button was already implemented for the last presentation, what changed is the destination folder of what we want to import. This destination is now the resource folder, that was changed so that loading the object on the scene was possible.

The Load button allows the user to load an object onto the scene by giving the name of the object the user wants to add to the scene in an input space. The Resource. Load function of Unity was used in order to load the object in Unity. To make the object visible, it has to be copied/instantiated in a GameObject (from the class of the same name of Unity). This GameObject will be stored in a list with all its information and a timeline will be created for the object, this will be explained later on.

The Remove button is used to destroy a GameObject from the scene. It uses the function destroy of Unity. To destroy an object, the user just needs to

input the name and the object and all its information will be destroyed from the scene (the GameObject will remain in the resource folder).

### 2.2.5 View

This section contains six different preset views : Front, Back, Top, Bottom, Left and Right. The Front view is the one set when a new project is started. These views change the position of the camera depending on the position of an invisible plane.

### 2.2.6 Edit

The buttons used for the manual skeleton have been placed under this button.

### 2.2.7 File

The file button contains three buttons : the New project button, the Save project button and the Load project button.

The New project loads a basic empty scene, with only the GUI elements.

The Save button allows the user to save the GameObjects he imported into the scene and their information (such as their skeleton). To do so, the user will only need to input the name of the file he wants to save in in the input field of the button.

For this button, different methods of serialization have been explored. The first method was to use the serialization function build in Unity, which would, according to their documentation, allows us to save GameObjects and other classes unique to Unity. After some research and experimentation, it seemed like GameObjects were not supported by Unity's serialization function, as saving and loading a simple GameObject was not working. As Unity serialization used binary formatting, it was difficult to really understand the way Unity saved the information, thus the method to save was changed. There were two major choices to save the file which were Xml or Json. Having little knowledge on how both works, we went for the first choice, as it was easy to understand and use.

The first thing that had to be done were parsers for our custom classes. First, the skeleton class, which is a general tree. To save it, it is put into a list containing all the nodes, the nodes being a structure that a Xml serializer can easily save. The same was done with the timelines which contain the movement and for the GameObject class (as a parser for those is not implemented with Xml functions). To save each object and its information, a class was made containing everything that was needed, that class contains the usable versions of the information (so a tree for the skeleton for example, and not a list of all the nodes) and when the user wants to save, it will transform into structures that

can be saved in Xml.

The Load button will be used to load a saved project. To do so, the user just need to input the name of the file (without the extension).

The process to make this function was almost the same as for the save button. From the serializable types, the load function recreates the whole scene where it was left. First, the game objects are all reloaded onto the scene with the same method as for the load function. First, the GameObject is taken from the resource folder (as each saved object had to be imported in this folder and then loaded from there, each object should still be in there). Then, its skeleton is recreated from the list that was saved and its timelines and keys are rebuilt to get the motion of the objects. This is done for each object that was loaded onto the scene. The camera the user see the scene through is also saved, so that when the user continues one of his projects, it is recreated exactly like it was left.

### 2.2.8 Options

The Options button allows the user to freely move panels around the screen, even putting them aside if necessary. When the button is clicked, the panels go into a free-movement state, in which they can be clicked and dragged around. Clicking the button a second time locks the panels in their respective positions. This operation can be executed as many times as the user chooses. To implement this, the program detects what panel the user has his mouse over when he tries to click one, then repeatedly sets the coordinates of the panel to the coordinates of the mouse, which are transfered from the screen coordinates to world coordinates. Final, the panel stops following the mouse when the button is released.

Along with this functionality, we chose to implement a system to resize the windows when they are in their default positions. This is done by dragging the edges of the panels that are marked as being able to be dragged through clear red lines. This functionality is disabled when the options button is used, but the panels retain their user-given sizes, giving the user full control over the heights of the panels, and as a second step, full control over their positions. This is true for all panels, with the exception of the Inventory panel that has a preset size due to its sliding nature that was described earlier.

### 2.2.9 Timelines

This panel of the GUI is what links the motion of objects to the interface we see. Its aim is to allow the user to freely choose what the program does with objects as they move around. To do so, a timeline system was implemented, which let users put keys corresponding to the various positions, rotations, or sizes of a given object. The implementation of such keys is explained in a later

part. We have three base timelines, Camera, Visual and Sound, that will be respectively for camera movement, extra visuals, and sound implementation. For every object added to the scene, a new timeline is automatically created during the loading under all previous timelines. This is added to what was first said, as initially all movements of all objects were to be placed in the Visual timeline. We decided to go past these initial expectations, in order for the program to be much more intuitive for the user. In order to ensure usability, when the size of the timelines window is too small to fit all the timelines at once, the panel becomes scrollable through left-click and dragging either up or down. This functionality is disabled when the panel is large enough to fit all timelines at once.

When adding an object, the new timeline needs to be placed underneath the rest of of the timeline, which is done by first making the parent panel longer, then adding duplicating a hidden, base timeline, which is then moved to the adequate position for the screen size. This base timeline has all functionalities implemented in it, except for the link to an object to the scene. This means that all timelines are simple copies of the this base template.

When deleting an object, the corresponding timeline needs to be deleted. When doing so, all the timelines that are lower need to be shifted up one position. This implies calculating the amount by which it needs to be translated upwards, which is dependent on the screen size. Once all the timelines are moved, the panel than contains them needs to be shortened in order to compensate for the smaller number of timelines. This is important for making the panel not scrollable when all timelines are visible at once. A slight margin if left, in case the user still wants to move the timelines up slightly, if the lowest one is very close to the bottom of the screen.

All timelines have the option to add all three types of keys, through three buttons placed on the left-most section of the timeline corresponding to the wanted object. These are initially placed in the center of the timeline, and can be moved along the timeline, corresponding to changes in the time of the key. Moving any key on any timeline temporarily disables the vertical scrolling of the panel, this facilitates the placement of keys at precise times. Each model of a key has an empty GameObject child that stores all needed information about the key, besides time and the object of the scene that it corresponds to. Its type (location, rotation or scale), its state as last element, and speed curve are stored in the empty object's name, with the following format : "R012". The first character corresponds to the type of key, with L being Location, or Position, R being Rotation, and finally S being Scale. The second character gives a boolean giving information about whether the movement is to be linear or curved. If it is curved, the character will be '1', hence the boolean being true, otherwise, it will be '0', hence a false boolean, corresponding to a linear movement. The last two characters are digits making the numbers 0 through 13. Each number corresponds to a preset speed curve, that is converted from an integer to a curve type. Their implementation is explained in the next part. The position, rotation

or scale of the key are given by the position, rotation or scale of this child, depending on the type of key.

As mentioned earlier, each key has a time attribute, which is given by its position on the timeline. The minimal time is always , but the maximal time varies. It is given by a slider placed above the timelines. The movements will be scaled with this value given by the user. In order to retrieve the time corresponding to the position on the timeline, two invisible objects are places at the beginning and at the end of the timeline. We then calculate the X coordinate as a percentage of the distance from the first blank object to the second blank object. It is then multiplied by the coefficient that gives the time indicated by the slider, which is found by finding the time of the second blank element, using this same function.

A key also needs to be linked to an object of the scene, with the exception of keys relating to sound or extra visuals, which are not implemented yet, as they are to be taken care of for the last presentation. One way to link objects together is to make one the parent of the other. This however, would mean that the coordinates of the object are dependent on the coordinates of the timeline, which is not desirable. We hence chose to make all objects in the scene children of an empty object that is at the origin of Unity coordinate system. The order of the children is the same order as the one of the timelines in the GUI, making it easy to find the wanted object from the timeline. All keys of a timeline are hence linked to their object through this empty object.

Overall, all the functionalities of the timelines that were needed for what is done for the motion are implemented as expected. We also added slight modifications to the structure of the timelines to make the them a lot easier to use. We hence accomplished, an even surpassed what was to be done for this presentation.

### 2.2.10 Additional UI elements

As the GUI was taking shape, we realized that a few extra elements should be added. First of all, the drag and drop system used to move the objects in the scene needed to be extended to rotation and scale. We hence implemented these in exactly the same fashion. In order to activate them, the user simply needs to press the 'R' for rotation, or 'S' for scale, at the same time as the usual 'X', 'Y' or 'Z'. We also added the system to make panels bigger, through a slider system. This was already explained in detail when explaining the 'Options' button.

## 2.3 Creation of motion

### 2.3.1 Key class

The animation of an object is decomposed into three components of movement: the position of the object, its rotation and its scale. The user should be able to decide the values of these components at any time during the animation

and in all three space dimensions. Our system that handles this information was inspired by Blender's management of keys.

A key is an object placed on the timeline by the user that stores a specific data about an object: either its location, rotation or scale (depending on the type of the key). In this way, the user gives fixed values at certain moments in the animation and the program will automatically compute the intermediary steps between these values. For a linear movement from A to B in 10 seconds, the user will have to create a location key at point A, at time 0, then another key, 10 seconds later on the timeline, that stores the coordinates of point B. It is the same principle for rotation and scale: the program will compute the changes from one rotation or scale state to another, using keys.

Therefore, we had to implement the Key class. Each object of this class is characterized by six attributes: the type, the Game Object it belongs to, a speed curve, a value, a time information and a boolean which tells us if the object will follow a linear path toward the next key or not. The GameObject is the object on the scene that will be animated. Every object has its own timeline with its own keys. There are three types of Key, one for the position that we called Loc, one for the rotation called Rot and one for the scale called Scale. These types allow us to determine which information of the object will be modified among the location, the rotation and the scale. The time in seconds corresponds to the moment at which the object's state will correspond to the value of the key. The value is a Vector3 variable as we need to express the motion components in three dimensions. The use of the speed curves will be later explained.

### 2.3.2 Actual motion

Now that we have three lists of keys per object, we need to process this information and compute the trajectory. At first, the aim was to allow the user to draw Bézier curves in space that will represent the path of motion. Curves are however complex to code and are not necessarily intuitive for the user.

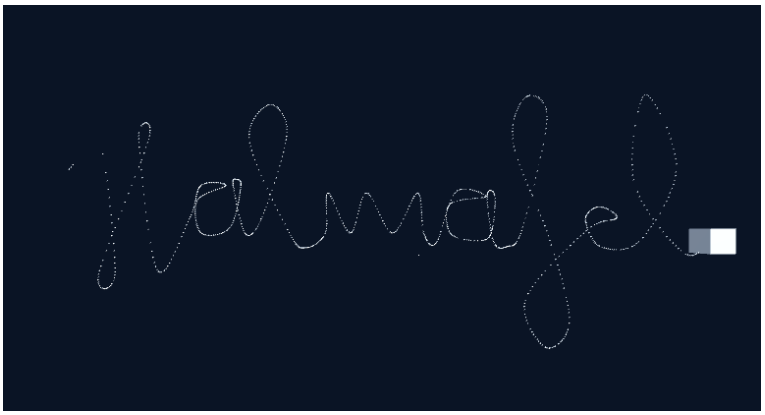
We then realized that there exist several assets in Unity's Asset store that make the creation of motion easier. We used a library asset called iTween Editor allowing us to handle movement, rotation and scale modification. Our first idea was to use iTween to move the objects around as it contains functions that, given two values and a time interval, modify the object's state along time. However, as there are three different functions dedicated to the three motion components (position, rotation and scale), it was impossible to modify at the same time, the rotation, the scale and position. Therefore several modifications couldn't be done simultaneously on three different parameters.

Our second idea was to use the Update() function which is called every frame. This allowed us to update the objects' states at every moment, for every parameter.

The state of the object is described at every moment by a path function that receives as parameters the value and the time of both keys. There exist two

trajectory options. First, a linear path, the keys are linked by a linear trajectory and the equation of motion is an affine function of the form f(x) = ax + b where a is the difference of coordinates between both keys divided by the difference of time and b is the initial coordinate. Then, there is the curved path option which is partly handled by iTween. We used the function PutOnPath() which receives three parameters: a GameObject (the mobile), an array of coordinates and a number between 0 and 1 that represents the percentage of the path on which the object will be placed. It is equal to the current time divided by the total time of the animation. At every frame, the percentage varies and gives the illusion of movement.

The difficulty was about managing three types of data at the same time. The rotation and scale parameters also have the option of a linear or a curved progression. The same affine function was implemented for linear progression. However, iTween's PutOnPath only works for a position path. A solution was to call this function with scale and rotation coordinates on two empty GameObjects. The empty GameObjects will be placed on the path and the main object will take the position of the empty GameObjects as rotation or scale values. Therefore the user would have the choice of a constant change of rotation/scale or an accelerated change followed by a deceleration.



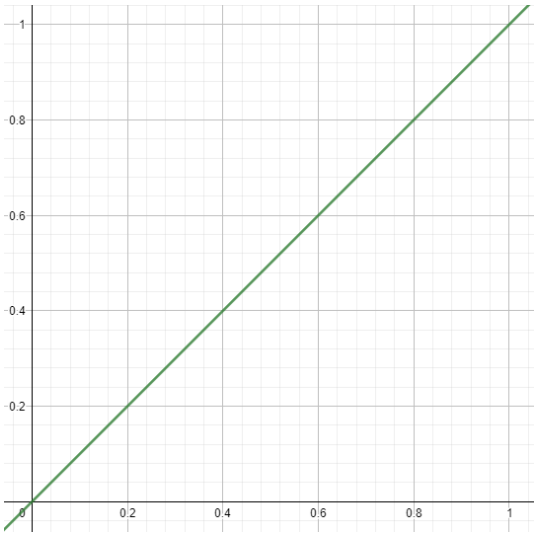
### 2.3.3 Speed curve

After having set the trajectory of the object, the user should be able to control the speed along this trajectory. This is why we needed speed curves to define the variation of speed along the path, between two keys. As for the implementation of the trajectory, our first idea was to build a Bézier curve of speed that the user would be able to modify. In this way , the user could decide for example to have a slow increasing motion at the beginning then a decreasing speed at the end of the motion.

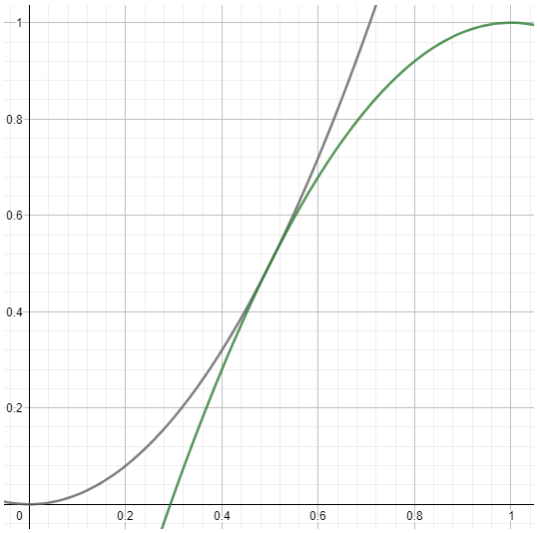
iTween owns a set of predefined speed curves, so instead of using Bézier curves, we decided to use this set of curves. It would allow the user to choose one curve among this set that could possibly correspond to the expected motion speed. A problem was that these speed curves are used in iTween's functions that we couldn't use because of the reason explained above so these presets

were inaccessible for our system. However, even if we couldn't use them we could take the functions of these curves and implement them in the Update() function. iTween's curves are based on Robert Penner's open source easing equations.

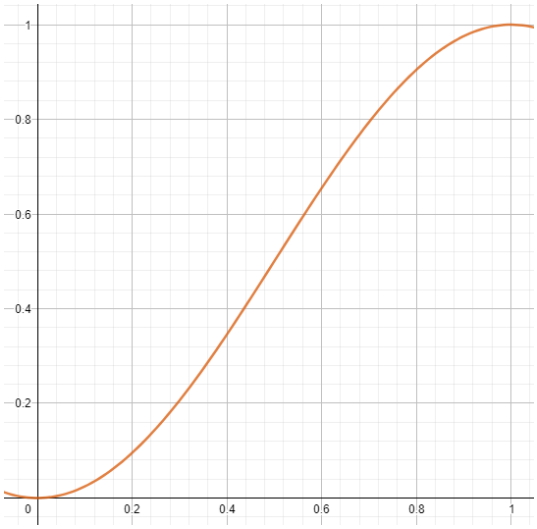
There exist three types of easing: ease-in (slow at the beginning then speed up), ease-out (starts fast and slows at a stop) and ease-in-out (slow at the beginning and at the end). Then, we must define the amount of easing : slow or fast acceleration/deceleration. iTween gives access to 32 easing functions but we decided to implement only 10.



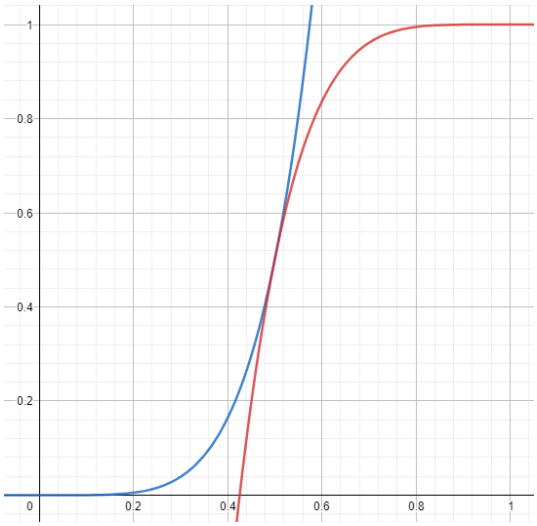

The simplest curve is the linear one, that is to say the speed is constant along the path. No need to implement an in/out acceleration as the acceleration is constant.




Then, we chose the quadratic easing. Its equation is based on a square variable (time in our case). The curve represents half a parabola. The easing-in and is expressed as $f(t) = c*(t/d)*(t/d)+b$ where t is time, b is the the initial coordinate, c is the total change of coordinate and d is the duration of the movement. For the out-easing, the function is : $f(t) = -c * (t/d) * ((t/d) - 2) + b$ with the same parameters. The in-out function takes the first or the second formula depending on the progress of the animation.




The third easing is the sinusoidal easing, it is based on a sine or a cosine function. The easing-in function is expressed as follows : $f(t) = -c * cos(t/d * (PI/2)) + c + b$ and the easing-out : $f(t) = c * sin(t/d * (PI/2)) + b$. The produced curve will have a smoother increase than the quadratic one.




Finally, the quintic easing relies on a fifth power variable and is the curve with the sharpest increase between the speed curve options. Its expression is : $f(t) = c * (t/d)^5 + b$ and the easing out : $f(t) = c * ((t/d)^5 + 1) + b$.

Therefore, for the speed variations of position, rotation and scale, the user will have the choice between a linear progression or a curved one, knowing that the curved option is accessible in three versions. This may looks like we are narrowing the user's possibilities, however our concern is to make the program ready-to-use. Although it seems to be an issue, the preset speed curves we have chosen still represent a large range of choices.

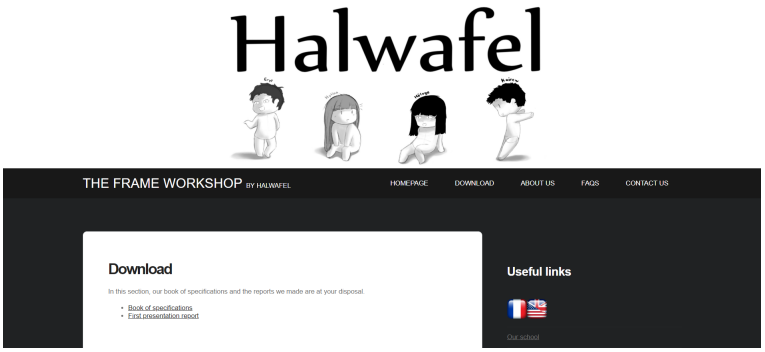
### 2.3.4 Camera

To do animation it is necessary to have cameras. The final render will use the cameras added by the user. For that reason, its movement during the animation is an important point. The camera will have the same system of

displacement as the other objects in scene. They will have a set of keys since they are GameObjects, they also have a position, a rotation, but no scale. In addition to this, we thought that we might add some functionalities that could be useful for camera movements. Methods have been implemented to allow the camera to follow a path of his own and adapt its focus towards a Game Object which could be invisible since the user may don't want the focus to be exactly on the object he animates. Another specific feature is the displacement of the camera with an animated object: so that the animated GameObject will always stay in the camera's field of view.

## 2.4 Website

Since the first presentation, the website has been slightly modified, as said the Book of specifications section has been replaced with a Download section where the Book of Specifications, the first presentation report and this report itself will be available as PDF. Hence, it will be possible to download all of these documents from there. Of course The Frame Workshop will also be available for download. Finally a background music and a logbook have been added. We kept the logbook simple, we basically just put what we had to be be done and then just tick it once it was. The website is still hosted using GitHub Pages and is available at https://kairew.github.io.



# 3 To be done by the next presentation

## 3.1 Graphic User Interface (GUI)

The GUI is finished for the most part as many of the buttons are implemented. It will now mostly be about linking the new functionalities to UI panels so that there are usable by the user.

Here is what need to be implemented :

- A button to be able to add particles effect to the scene.
- A button that will allow the user to load a sound onto the scene.
- Background images will be in the scene.
- The timeline containing the sound.
- The script associated with the button render will be written.
- The save and load buttons will have to be modified so that they can save and load the added functionalities.
- Under the insert button, a second load button will be added for objects that do not need to move during playtime.

## 3.2 Creation of motion

The goal for the next presentation will be to implement internal motion, that is to say, mesh deformation. To do so we will made possible to articulate the skeleton, every vertex of the mesh will be linked to a bone so that when the points move the mesh is moving along with it. To do so we will need to handle two things:

- Skinning: Define for each bone of the skeleton which point of the mesh is going to animate. This will be done with two methods:
  - The first one allows the user select a group of vertices on the mesh and assign it to a certain bone of the skeleton. The bone's movements should affect the movements of all the vertices linked to it.
  - The second one will automatically assign every vertex of the mesh to the closest bone.
- Articulation: relying on the hierarchy between the nodes of the tree representing the skeleton, an animated point will only move its children. The animation of a bone will be made by the rotation of an articulation point, it will therefore use the same principle than the other objects (keys) but only for the rotation.

## 3.3 Background imaging and Sound

The Background images are images or GameObjects that will automatically be loaded in the scene when a new project is started.

The Sound can already be imported but not yet loaded in the scene and linked to a timeline. The load button for sounds will be similar to the button used to load objects onto the scene. The sound will be added to a timeline once it is loaded.

## 3.4 Particle animations

With the particle animation, the user will be able to load particles onto the scene. To do this, the in-built particle system of unity will be used. The user will be able to choose between preset colors and sizes of particles and load them onto the scene as well as make them disappear from the scene at the time he decides.

## 3.5 Website

The website can be considered almost finished. It now has a background music which may be bothering while we go from a page to another, so we will find a way to solve this, maybe by duplicating the website into a soundless version. Among other things, everything that has been asked for the final presentation to be available for download will be added in the Download section of the website.

# Conclusion

For this second presentation, the goals for GUI, creation of motion and website have been achieved. The GUI has integrated all the new functionalities that we implemented about motion and save/load. The motion of objects in the scene is ready, with the possibility to modify the trajectory and speed variation along time. The website is constantly being improved and updated. The automatic skeleton which has been started before the first presentation has been given up, however the manual skeleton is functional and will be used for the implementation of the internal motion, that is to say the articulation of the models.

To conclude, what had to be done has been done, we are aware it is not over yet and many challenges are awaiting on our path.