EPITA

UNDERGRADUATE (FIRST YEAR) PROJECT

25.05.2018

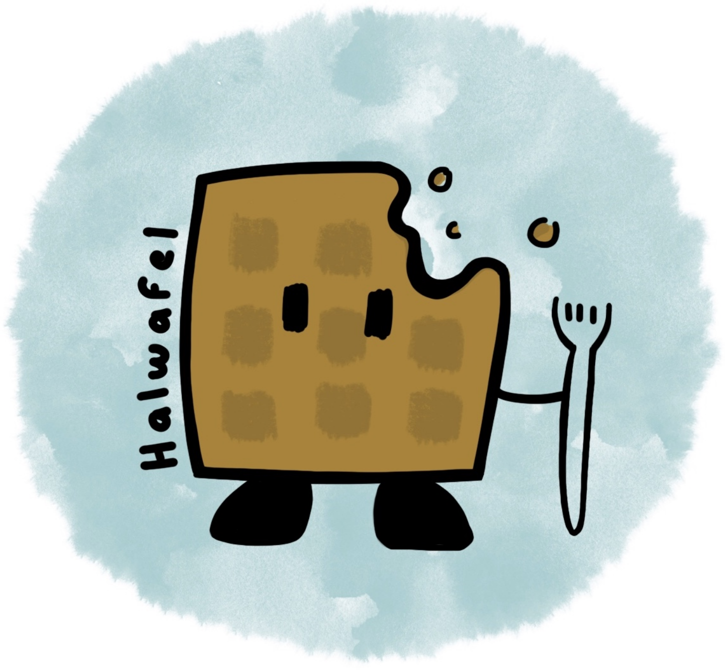
3RD PRESENTATION REPORT

THE FRAME WORKSHOP

3D ANIMATION SOFTWARE

HALWAFEL

Brian Bang, Etienne Benrey, Léane Duchet, Julie Hazan



Contents

Introduction

Halwafel is a group made of four members, Brian Bang, Etienne Benrey, Léane Duchet and Julie Hazan. It was founded on October 16th, 2017. The name of the group is based on a stylized contraction of the two words half and waffle. This report will present in detail our software project, **The Frame Workshop**, from its initial design to the fine details of its construction, without neglecting the difficultes and successes of the undertaking of such a project. We will first present Halwafel as a group, then explain the concept of our project, what were able to accomplished, before delving into the details of the functionalities and the implementation of our work.

1 Presentation of the group members and project

1.1 The group

Halwafel is, as mentioned previously, a group constituted of four members of EPITA's first-year students. We each come from different backgrounds, with different experiences, and were able to bring something new to the table.

Brian Bang

Brian was a student from the Lycée Albert-de-Mun. He already had a good basis when it comes to writing code, having already done a code project in the context of his Terminale year's ISN class, in which he coded a full MasterMind in Python. Despite not having any experience when it comes to editing, or using programs such as the one we implemented, his great competence in website design and construction made him have a skill that no one else had truly acquired in their previous experiences.

Etienne Benrey

Etienne was previously a student from the Lycée International de Saint-Germain-en-Laye. He had already coded in C++ regarding his Terminale's ISN year. The project he worked on that year was the card game called Président. He also had an optional course which was "AP Informatique" giving him coding basics in addition to his ISN project. His large experience in video editing helped him having a quite good vision of our project even if he had no experience in animation.

Léane Duchet (Project Manager)

Léane comes from the Lycée Sainte Thérèse. With limited, but non-negligible coding experience in Python thanks to Prologin's GirlsCanCode training course, and a good understanding of what a simple animation program looks like, she was the member of the group who could always keep us in check. Her rigor was a very helpful asset throughout the project.

Julie Hazan

Julie was with Etienne at the Lycée International de Saint-Germain-en-Laye. Her first coding experience was for her Première's TPE which consisted in developing a encrypting and decrypting algorithms in Visual Basic (Caesar, Vigenere, RSA). She also used Blender a little which brought an artistic touch to our group.

1.2 Project

The first idea we had was a RTS (real time strategy) game. We then decided that we did not want to make a game especially since some of us do not play games regularly and hence lacked motivation for such a project.

The second idea was to construct a machine learning neural network. The most obvious problem that this method could solve would have been a maze. We gave up this idea after learning that this would be done in S3, as it would not be interesting to do it twice.

The third idea was making a GPS-like path finding system using A* method. This idea came up after we watched a series of computer file videos, which explain various path-finding methods. It was also interesting because it was correlated with maze solving, so went back to our previous idea. We gave up path finding because using a preexisting method meant that the project could be done too easily.

The next idea was map generation using fractals. This included the generation of mountains and forests. This main drawback to this idea was the computing power required to render these maps, we were afraid our computers would not be powerful enough.

Then, we thought we could try to do something with house automation. The idea was to process information from sensors placed in a house, use it to track problems (such as floods) and display them on a map of the house in order to be able to take action as quickly as possible. The drawbacks were that if we wanted to test our software in a real situation, we would have needed to buy sensors, which we heard was prohibited by the project rules.

We then had a spontaneous idea, which was to do a Tetris-like game. This would imply creating a 3D multiplayer Tetris in which players could affect other adjacent players' boards. We then realized that this was going back to our original game idea. Furthermore, this idea was too simplistic and had more creativity obstacles than programming obstacles.

Finally, we got to our final idea, which originated from one of our members' father's business project, which required the creation of a 3D or 2D animation video. We hence got the idea of developing a 3D animation-editing program. This seemed like a good contender for our project, since if this could be useful to one's business, it could be useful in a more general context. For example, our program could be used for trailers, marketing videos, animation films, advertisements or even for video-game cutscenes. Since we could use Unity, the 3D aspect of this project was feasible.

The Frame Workshop is an animation software, which allows a user to make his own animations, the user is able to import his own objects from Blender or other similar modeling programs, or use Unity's assets. The user can then create a skeleton to different meshes. Moreover many tools are provided to the user in a GUI (stands for Graphic User Interface) to manipulate every object as efficiently as possible : timelines for objects, cameras, visual effects, sound synchronization.

Cf. Annexe 1

2 Book of specification follow-up

For the next table, what is in parenthesis is what was supposed to be done by the presentation in the column.

Part of the project	**15 March 2018 (S1)**	**30 April 2018 (S2)**	**11 June 2018 (S3)**
Mesh + Sound import	100% (100%)	100% (100%)	100% (100%)
Skeleton creation	50% (100%)	50% (100%)	50% (100%)
Graphic User Interface (GUI)	30% (30%)	85% (80%)	100% (100%)
Creation of motion	0% (0%)	55% (55%)	100% (100%)
Background imaging + sound	0% (0%)	0% (0%)	100% (100%)
Particle animation	0% (0%)	0% (0%)	100% (100%)
Website	50% (40%)	90% (70%)	100% (100%)

Part of the project	**15 March 2018 (S1)**	**30 April 2018 (S2)**	**11 June 2018 (S3)**
Mesh + Sound import	Julie Brian		
Manual skeleton creation	Julie		
Automatic skeleton creation	Léane Etienne		
Graphic User Interface (GUI) Movement in 3D space and First GUI layout	Julie		
Graphic User Interface (GUI) Timelines, option button, properties and inventory		Etienne	
Graphic User Interface (GUI) Toolbar		Léane	
Creation of motion manual skinning and motion		Brian	
Creation of motion automatic skinning and motion		Julie	
Background imaging + sound			Etienne Léane
Particle animation			Etienne Léane
Website	Brian		

Parts of the project	Done	Modified	Abandoned
Mesh + Sound import	✓		
Manual skeleton creation	✓	✓	
Automatic skeleton creation			✓
Graphic User Interface (GUI)	✓	✓	
Creation of motion	✓		
Background imaging + sound	✓		
Particle animation	✓		
Website	✓		

All detailed explanation about the abandoned and modified parts as well as how they were done will be in the third part : Achievements. There are also some added functionalities that will also be detailed in the third part.

3 Achievements

3.1 GUI (Graphic User Interface)

First, we handled the general aspect of the interface, organizing the window and deciding where to place the buttons and other UI elements. Then, another important part of the work was to facilitate the user's direct interaction with the scene and its objects. Meaning how we can modify the view in the 3D scene, move an object and interact with those.

Cf. Annexe 2

3.1.1 Creation of interface using UI elements

The interface's general aspect was made using UI panels, whose position depends on the camera. For any movement of the camera, the interface would move accordingly. Panels also possess a useful property which is the adaptation to the size of the screen. The sizes of the panels will be stretched if the program is opened on a larger screen or on the contrary, they will shrink if the main window is reduced.

Our interface consists of four panels: a tool bar, two aligned panels on the right for the inventory (up right) and one for the interaction with the scenes (down right) and one last panel for the time lines. The rest of the screen is occupied by the 3D view. The interface is accessible as a set of independent windows that can be dragged around and resized by the user.

The interface also needed some elements to allow the user to interact with the program, so several UI objects were implemented such as buttons, a text box, sliders and many more. Those elements do not have the same distortion properties as the panel: they are fixed. This means that their size won't be affected by a change of size of the screen. Toolbar buttons are an exception as they are not fixed vertically, their height adapts to the one of the tool bar panel. The anchor tool of unity was very useful at this moment: it determines the level of distortion of an element, relatively to an axis and the parent of the object.

The tool bar is composed of five buttons : File, Edit, Insert, View and Options

3.1.1.1 Toolbar

For each of the following button, a panel appear when the user click on them, on those panels, there are the different buttons that are explained in each part.

Insert

This button gives access to five other buttons : the Import button, the Load Object and Load Sound buttons, the Remove button and the particles button.

The Import button, its use and how it was implemented will be explained later on.

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. There are two input fields that appear when this button is clicked, the first one for the objects that need to move during the animation, the second one for the objects that do not move overtime.

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).

The particle button allows the user to load preset particles onto the scene. There are five types of particles : the flare, the fireworks, the steam, the smoke and a dust storm. As long as the play button is not clicked, the particles stay onto the scene. Once it is pressed, the particles appear at the time the user want them to and disappear when he wants them to disappear.

The Load Sound button is used to load sound onto the scene. It is added to the sound timeline so that the user can choose when it should be played during the scene.

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.

Edit

The buttons used for the manual skeleton are placed under this button, the functionalities provided by this button will be explained in the ”manual skeleton” part.

File

The file button contains four buttons : the New project button, the Save project button, the Load project button and the Render 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, we did 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. This was removed as our tree class was not useful anymore for the creation of motion. The nodes are now GameObjects and the hierarchy is made using the parent/child relation of Unity, so there was a parser for those that also put them and their children into lists that can be parsed in Xml. Then, the bones needed to be serialized as they contain the skinning of the mesh, that is represented by some vector3 of Unity (it needs to be saved as there is manual skinning and not only automatic one). The serialization 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 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.

The **Render button** is used to transform the animation created by the user into a video of a given format. This is done by running the whole animation on a secondary camera, and recording in real-time the animation seen by the camera. To do so, we decided to use a Unity Asset called FrameRecorder, which is a generic basis for various recording scripts. It allows for a lot of flexibility in its use, while offering a simple solution for recording the user's screen in run-time. It is divided into three main parts : the media input, the recorder and the settings, or base structure. This follows the general structure we see in many programs of inputing data, processing the data, and then bringing it to the user in the form of a GUI. In our case, the input data is the camera in the scene, the processing is transforming the data into a video format. The GUI component is hidden from the user of our program, but represents the settings that are inputed into the recorder. Each part is composed of one or more classes, but we will consider each part as one over-arching class, despite this being a slight simplification.

The media input is responsible for retrieving all the information that needs to be recorded at a given frame, and transforming that information into usable data for the recorder. This runs at every frame, and retrieves either the audio, the bitmap representing all the pixels of the screen, or both. Certain settings affect the input, such as the demanded screen resolution. The framerate or other settings regarding the video itself have no affect, as the input is called once per frame at the start of the frame, desregarding anything about the sequence of frames we want to retreive. It focuses solely on the given frame. The settings for the input stay fixed throughout the recording, implying that they will keep the settings given at the start of the recording. An editor class is implimented alongside the inputs in order to modify these initial settings.

The recorder class is the most complex of the three parts, as well as being the essential part of the recorder. Its goal is to have a generic structure for all types of recorders which take data from the input, and transform it into a different format. It is in essence a formatter for the data we retreived. In order to accomplish this, the recorder is first notified of the start of th recording. This instantiates all necessary elements for the data transfor to occur. Similarly to the input, it is then called at every frame to process the new inputs of that frame. A recorder can take as many inputs as is implemented, but, unless specifically implemented, is not ready to handle a different number of inputs than what is expected. One again, since the recorder is called every frame, it is in no way influenced by the sequence of frames that is called. It is not dependant on the input's settings either, as these are completely distict parts of the asset. Furthermore, the output file that the recorder creates is not saved in memory, and is only a temporary file. It hence needs to be handled by a more general structure, that encompasses both the input and the recorder.

The final part is the base structure, which is a generalisation of what is implemented as the "Support" and the multiple "Editors", which combine to make the two previous parts function correctly. This is what takes all the remaining settings, most notably the frame rates, whether they be constant or not, the recording start, and the recording end. The asset allows for many more settings to be modified if implemented, but are not necessary for our uses of a renderer. When the recording is activated, this base structure takes in all the required arguments, starts up the input and the recorder, then handles each frame, calling first the input, then transfering the data to the recorder. When the recording ends, this same structure takes the temporary file given by the recorder, and saves it to the desired file path.

The FrameRecorder asset is hence a huge help when it comes to recording the screen and turning the wanted data into a video format. However, it does not offer a recording asset, simply the general, unimplemented classes that would be required to perform such a task. We hence had to use this basic structure to implement our recorder. This was done with the help of a second asset : FrameCapturer. This asset is what allowed us to use the FrameRecorder to transform the screen bitmaps into an mp4 format, allowing us to output a video file to a designated path. It is made to function alongside FrameRecorder, allowing us to easily implement all the required elements for the FrameRecorder. We set the features of the recordings to have a constant frame rate of 30fps, which we deem to be sufficient for the type of rendering that can be done with our software. With the help of these two tools, the rendering can be done smoothly, through a real-time recording of animation.

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. Finally, 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 will be described in the next part.

3.1.1.2 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.

3.1.1.3 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.

3.1.1.4 Timelines

Implementation

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 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 four types of keys, through two 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, thus facilitating the placement of keys at precise times. The four types of keys can be divided into two main catégories : translation keys, and motion keys.

Any key needs to be linked to an object of the scene, with the exception of keys relating to sound or particles. 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.

Each model of a translation 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 linearity, 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.

Motion keys are implemented in a very similar fashion, as a lot of the required information is the same. Each one of theses keys has a gameobject attached to it, which is once again given by the timeline it is attached to, and a list. This list is the equivalent of the value for translation keys, but allows for multiple articulations to be handled by the same key. This means that making one smooth movement that involves several articulations is facilitated, as the user does not need to always make sure all the keys are aligned. Each element of the list has its associated articulation, the rotation we want that articulation to be at when we get to the studied key, a speed curve, and a linearity boolean. Once again, the value of the rotation is given by the rotation of an empty object. The linearity and speed curve are once again coded into the name of the empty object, with the added information of the serialized number of the articulation. The latter allows for easy access to the wanted articulation simply from the empty object.

There are finally three special types of keys, one type for each pre-implemented timeline : camera keys, particle keys, and sound keys. Camera keys are implemented in that same way as translation keys, but are limited to position and rotation, with a addition of two booleans indicating if the camera focuses or follows a given gameobject. The gameobject in question is given by its position as a child, and is stored in the name in the same way as Motion keys have articulations stored. Particle and sound keys have a single time attribute, as all the other attributes are either unnecessary, or already given but the position of the particle in the scene.

As mentioned earlier, each key, no matter its type, has a time attribute which is given by its position on the timeline. The minimal time is always 0, 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 placed 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.

Application

With all the tools for storage in place, the user needs to be able to use the timelines as he wishes. All timelines have the same structure. On the left part, a header allows the user to create keys as he needs them.

When a translation key of a given type is created, its value, so position, rotation or scale, is immediately stored. If the user wants to modify this value, he or she will have to create a new key with the wanted value. This is to avoid extensive cluttering of the GUI. Creating such a key is done by clicking the top button on the timeline. To choose the type of key, the user slides a button between three positions. The current type selected is both written under the slider, and indicated by a color code. Modifying its other attributes is as easy as right-clicking on the key to change is linearity, or Alt-right-clicking on it to cycle through its various preset speed curves. Any change will be marked at the bottom right of the panel, and can also be seen in the Properties Panel which, as mentioned previously, is accessible through a simple left-click of the key.

Motion keys are slightly more complicated to use, as they can hold any number of articulations. The input of theses articulations and their values is hence not given at the creation of the key, but after. Creating a motion key is done by clicking on the bottom button of the timeline. In order to start selecting and editing articulations, the user Ctrl+right-clicks on the key to enter an edit mode. Once the edit mode is started, the object becomes invisible, revealing its skeleton. Clicking on any of the articulations selects the articulation, automatically setting it as an articulation of the key. You can then modify the rotation of this articulation through the minimizable popup panel above the timelines, which allows you to rotate in all three spacial dimensions. Dragging the first coordinate all the way to one side will rotate the articulation by 90 degrees in the X coordinate, before coming back to the center. This is the same for the second and third coordinate, which correspond to Y and Z respectively. Exiting the edit mode will remove the popup panel and make the object visible again.

The special case of the Camera timeline is implemented in the same way as translation keys with regards to creation of the key and selection of its type. However, the selection of a gameobject to follow is done in the same way as the selection of articulations for motion keys, if the user chooses to have the camera follow an object.

Keys for sound and particles are different from the other types of keys, since the user cannot create any. They are automatically created when a sound or a paticle is imported into the scene through the toolbar. There is hence no special feature needed to be implemented for these timelines.

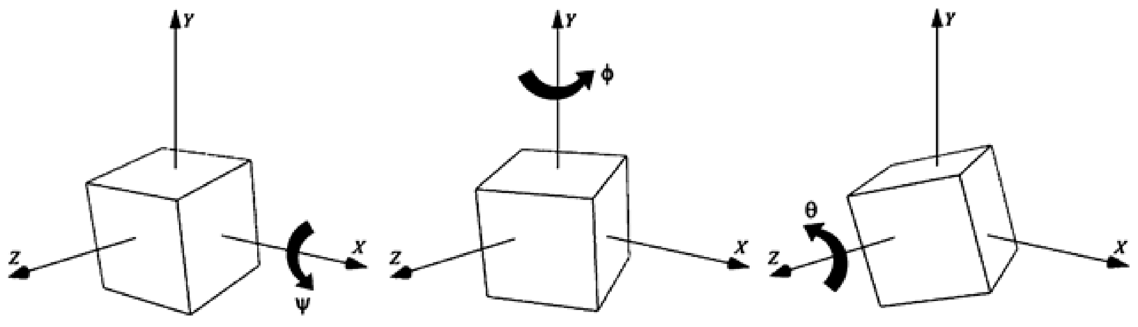
All keys, no matter what timeline they are on, have a time component that needs to be modified. When a key is created, it is but in the center of the timeline, at the middle point between the time 0 and the maximal time. In order to modify when the key is activated, the user simply clicks and drags the key to the left or to the right as he or she pleases. Putting the key all the way to the left will activate it instantaneously, while putting it all the way to the right will activate it at the very end of the movement. Note that dragging the key requires the user to left-click it, hence conveniently activating the Properties panel. The maximal time can be changed through a slider at the top of the timelines panel, giving the user a certain amount of flexibility in this regard.

The timelines are in essence what connect the user to the motion, in the same way that the rest of the GUI connects the objects to the scene.

3.1.2 Camera movement

To allow the user to set up his scene and move around in the 3D view, we had to manage camera displacement and rotation. A first attempt was to synchronize the mouse movements with the one of the camera. This method was very intuitive but could not allow many displacement possibilities : the Y axis wasn't taken into account. The second idea was to take the same displacement commands as Unity. This means the camera movement had to be decomposed into three sections : rotation (right click and dragging), side movements (scroll wheel click and dragging) and zoom (scroll wheel).

The rotation of an object in Unity can be identified by the Euler angles. This method compares the local axes of a rigid body with fixed axes. Any orientation can be described by three elementary rotations: the three Euler angles. this will allow us to affect the three dimensional orientation of the camera by applying a 3D vector.



Rotation of cube with Euler angle representation

Here, we can see the three components determining the orientation of the cube. The three Euler angles are Ψ, Φ and Θ, respectively along the x, y and z axes.

In our case, the rotation affects only the x and y axes of the camera, we don't want the camera rolling on the side. The program starts when the scroll wheel is pressed down. At this moment, the position of the mouse is stored as the initial position. If the mouse is dragging, we compute the difference between the initial position and the current position of the mouse, this will give us the two dimensional vector of rotation (the last coordinate, z, is set to zero) . The vector is then added to the Euler angles of the camera, multiplied by the time variable and the chosen speed.

The side movements was the simplest section to implement. The program must detect anytime the scroll wheel is being pressed and subtract the position of the mouse (multiplied by the speed and the time variable) to the camera position. Only the x and y axes need to be updated as side movements only deal with parallel displacements to the screen.

Finally, the zoom section consists of a linear displacement along the z axis (in Unity, the z axis represents the depth of the scene). However, we had to take into account the orientation of the camera: the local z axis of the camera may not be aligned with the absolute z axis. A solution to this problem was to link the camera to an empty game object, a parent, that would always keep the same orientation and will follow the camera into its movements. to obtain the z camera axis, we simply have to get the localEulerAngles of the camera and only take the z component. Therefore, every position changes should be applied to the parent of the camera (in this way, the local camera position is always the zero vector) and any rotation should be applied to the local camera Euler angles, not affecting its parent.

3.1.3 Drag and Drop of objects

Another important element that allows the user to interact with the objects in the scene is to be able to drag and drop them around in the 3D view. We need first to have a function that returns the selected object. This is done by using the raycast Unity tool. It draws a perpendicular line to the screen, with its origin located on the pointer. The generated ray then intersects with the collider of the object and returns the associated game object.

When the mouse is dragging, its position is being added to the one of the target. In addition to this, we wanted the dragging of the object to be more precise and also to be able to move the object along the z axis (which could not be done with the mouse movements) so the idea was to decompose the movements of the target along the three absolute axes. In this way, if the x key is pressed down, the object must only move along the x axis (respectively with the y and z keys).

3.1.3.1 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.

3.2 Sound and Mesh import

This part was complicated, at the beginning our goal was to open the file dialog and let the user choose the file he wanted, the program was supposed to check whether the file was an audio file or a 3D model. When we started, the first issue that occurred was that we didn't know where the chosen file was going, so we created two files Models and Sounds and changed the target path depending on the extension of the chosen file. Afterwards, while we were running the program it took between one and two minutes to do the whole copying process. Also during Runtime mode we don't have access to the UnityEditor class which is the class we need to open the file panel to let the user pick the file he needs so we had to opt for another method.

The other method may not be the most efficient considering that the user has to seek for his files, indeed what we did is asking for the path to the file the user wants to import and then we copy it in the resource folder (we realized only one folder is needed). Since the latter was quicker than the first method we used, we decided to keep this one despite its difficulty to use.

3.3 Skeleton

3.3.1 Manual skeleton creation

The user has the possibility to generate and place himself the points of the skeleton of a specified object. The process begins by clicking on the "Manual skeleton" button, the user must also select an object on the scene by clicking on it. This part uses the raycast functionality of Unity which allows the program to identify an object on the scene located under the pointer, when the left mouse button is activated. A red sphere is then added to the scene which represents the root of the skeleton. It is set as a child of the object which means every movement of the parent will affect the position of the root, therefore the position of the skeleton.

The rest of the skeleton consists of a general tree, with each node a sphere with its position. There are two ways to complete the tree: by pressing the E key, the user will be able to add a child to the last created sphere. The user can also simply select a sphere from the skeleton and chose to add a child to it, to drag it around or add a child to its parent.

Due to the relation child parent between spheres, there exists a hierarchy of movement. Thus, a change in the position of a sphere will affect all its children but the position of the parent will remain unchanged. This is very useful when thinking of the implementation of the animation of the skeleton.

3.3.2 Automatic skeleton

The automatic skeleton has not been 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 the second presentation, we first tried to understand the concept of mesh in Unity and how they are made.

We will first explain the ideas we implemented and then detail why they did not work.

For the automatic skeleton, two classes were needed, the point class and the tree class, used to make the hierarchy of the tree. As the automatic skeleton is not used anymore, they have been removed.

Point class

This class is used to make the nodes of the skeleton. It takes a name and a sphere GameObject as parameters. The first parameter is made to be able to have a name for each point in Unity and the second to place it considering the axis of Unity.

The Point class has different methods so that the rotation and the position can be modified by adding to the coordinates. Each point has to be added in a tree, and thus has its position in the tree modified at the creation of the tree it is part of.

Tree class

This class is used to make the links between the points. It is an implementation of a general tree, with the root and a list containing its subtrees. We can either add a full subtree or a point to that tree (if a point was added, a tree was first created and then added to the list of subtrees).

Once we had this, we had to implement the creation of the skeleton itself.

Finding collisions with the mesh

One of the most important features we had to implement in order to automatically create a skeleton is finding the collisions between various line segments and the object's mesh. This allows us to know when points in 3D space are inside or outside the mesh, if a line segment is fully included in the mesh, etc. Since there are no pre-implemented methods that fill this role, we had to implement one ourselves. The method functions in the following way: for each triangle of the object's mesh, the program determines if the line segment intersects the triangle. If so, that intersection point is added to a list in the form of an object of class Point. This is done in four consecutive steps:

1. First, the function computes the intersection between the line and the plane defined by the three points of the triangle. This intersection can either be empty, in which case we can ignore this triangle, a single point, or an infinity of points. The latter is extremely rare, as even if the line and the triangle are parallel and aligned, the lack of precision of a floating-point number will slightly skew this result. If such a case does happen, it can be ignored, as one of the triangles connected to this one will also be intersected by the line. This means that all cases in which the intersection is not a single point can be ignored.

   Finding the intersection between the plane and the line requires the plane Cartesian equation of the plane, so a normal vector to the plane, and a parametric equation of the line. This leads to solving the following system:

$$S = \begin{cases} a * x + b * y + c * z + d = 0 \\ x = \alpha * t + x_M \\ y = \beta * t + y_M \\ z = \gamma * t + z_M \end{cases}$$

with $M \begin{pmatrix} x_M \\ y_M \\ z_M \end{pmatrix}$ a point of the line and $\vec{n} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ a normal vector to the plane defined by the points of the triangle.

With a little computation, we get the resulting solution for $I \begin{pmatrix} x_I \\ y_I \\ z_I \end{pmatrix}$, the intersection between the line and the plane:

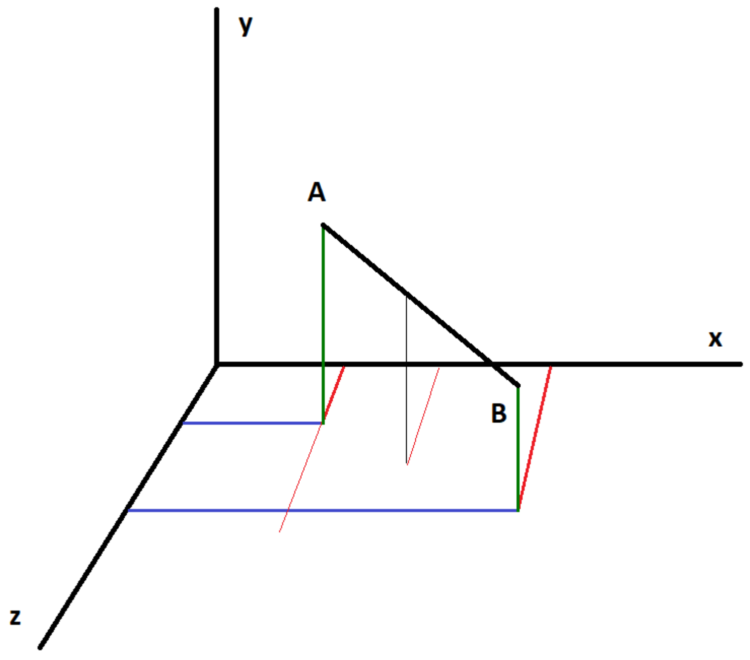
$$S = \begin{cases} t = \frac{-(a*x_A+b*y_A+c*y_A+d)}{a(x_B-x_A)+b(a(y_B-y_A)+c(a(z_B-z_A)} \\ x_I = (x_B - x_A) * t + x_M \\ y_I = (y_B - y_A) * t + y_M \\ z_I = (z_B - z_A) * t + z_M \end{cases}$$

with $A \begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix}$ and $B \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix}$ the two points defining the line.

If the denominator of t is null, then we are in one of the cases in which the intersection is not a single point, so we can simply make an exception for when this is the case. This allows for a clear and unique formula for each one of the coordinates of I.

2. Second, the program checks if this intersection point is included in the triangle. The method used is to check, for each of the three sides of the triangle, if the intersection point is on the same side of the line (defined by the studied side) as the vertex not included in the line. This is done by multiplying the coordinates of the vector product of the vectors connecting both points (the intersection point and the triangle's vertex that is not included in the line) with one of the points of the line, and the vector connecting the two points defining the line. If the two points are on the same side of the line, then the vector products will have the some orientation, and the scalar multiplication of their coefficients will be positive. If they are on opposite sides, implying that the intersection point is not included in the triangle, then the scalar multiplication of their coefficients will be negative. Testing this for all three sides determines if the point is in the triangle or not.

3. We now have a list with all the intersections between the line and the mesh. However, what is requested is only the ones between the two given points, not on the whole line. As such, the next step is to remove all points that are not in this range. This is done by looking at simply one coordinate at a time. Assuming that the difference between the x coordinates of the two points is not null, any point between the two points will have their x coordinates between the x coordinates of the two points. In order to avoid problems with float precision, the program does this for the coordinate (x, y or z) for which the difference is the greatest, ensuring that we do not miss any elements.

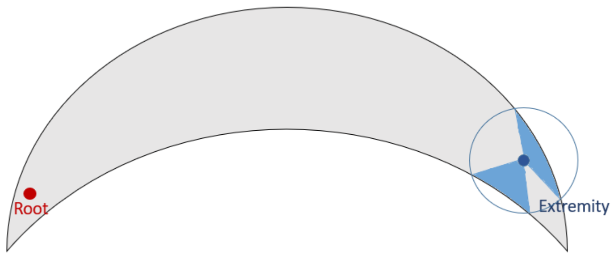


Identification of point in line segment

4. The final step is sorting the points, so that they are in order. This essentially means sorting them by increasing distance to one of the points. To do so, the program generates a list containing tuples (implemented as arrays of size 2) with the distance and the index of point in the original list of points. A simple bubble sort is applied to this list of distances, and then a new list is created and returned, according to the new order of indexes. At this point, the functions has finished running and returns the resulting list of 3D points.

Finding the extremities

Now that the tools are in place to start finding points, the program can attempt to find the extremities of the mesh. To do so, the program traverses through the whole inside of the mesh, looking at points at regular intervals. Each one of these points is analyzed to know whether it is an extremity or not. Within a radius determined by the size of the object and the user-defined density, the program checks how much of the point is surrounded by mesh, and how much of it is not. This is done by approximation, looking at 26 different angles, all at an angle $\frac{\pi}{2}$ of one another. If the point is considered to be an extremity, it is added to the list of points that will be the list of extremities. Since the positions that are looked at are spaced out by a distance that is proportional to the radius of the circle, overlapping points that would correspond to the same extremity are avoided.



Finding extremities

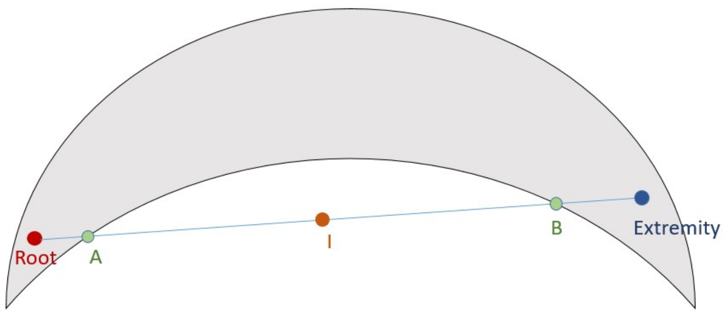
Creation of points and links between the root and the extremities

The next thing that had to be done was the links between the root chosen by the user and each one of the extremities of the mesh. For each one of the ends, the goal was to either directly link the root with the extremity or to find intermediate points to be able to link them.

If it is possible to link them directly, that is to say that the function determining the collisions did not find any collision with the mesh, then the extremity is added in a tree that will become a subtree of the root.

Otherwise, we start to search for possible intermediate points that allow us to create links. We will do this in four steps.

1. If the point cannot be linked to the extremity that means that there should be at least two points of collision with the mesh in between those two, so we take those two points that will be called A and B, and we first search for the middle I of the segment [AB].



Finding the center of AB

2. After finding I, the points of the circle of center I and a given radius R (big enough so that there could possibly be collisions with the mesh between those points and I) have to be found. This circle is the intersection of the sphere of center I and radius R with the plane containing I and orthogonal to AB. The points of this circle are given by :

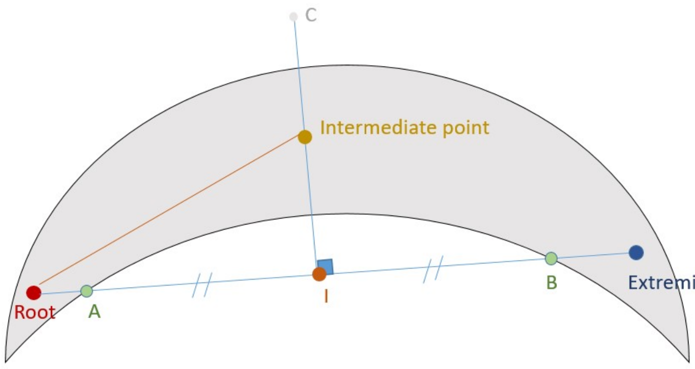
$$C : \begin{cases} x = \frac{-(2*y_I+R*sin(\Phi)*sin(\theta))*(y_B-y_A)-2*z_I+R*cos(\Phi))*(z_B-z_A)}{x_B-x_A} \\ y = y_I + R * sin(\Phi) * sin(\theta) \\ z = z_I + R * cos(\Phi) \end{cases}$$

with $0 \leq \Phi \leq 2\pi$ and $0 \leq \theta \leq \pi$.

If we have $x_B - x_A = 0$ then the intersection is computed differently and will give formulas with either $y_B - y_A$ or $z_B - z_A$ as the divisors. As A and B can't be the same point, we will always have at least one those divisors different from 0, giving us all the possibilities for the difference between A and B. We then check for different angles of θ and Φ whether we have collisions with the mesh between I and the computed point. If we do not we just continue to search computing another point's coordinates until we either find one that crosses the mesh at least twice or we have done a complete circle, that last case means we have no way of creating an intermediate point using this point I so the first step is redone on the right and on the left of I (using I as A or B, like in a binary search).

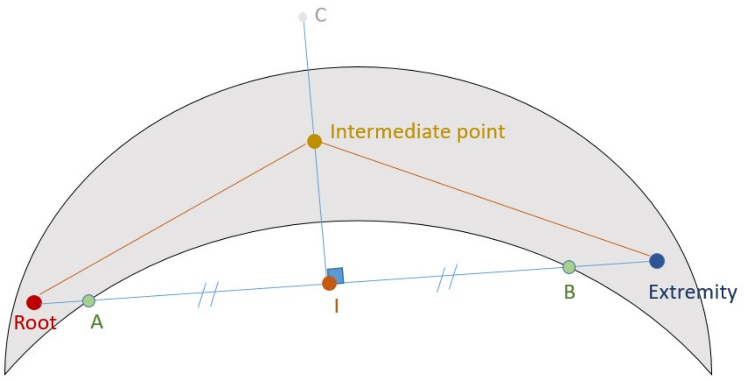
3. If a point for which the mesh is crossed at least twice was found, then a point that is in the mesh is taken (between the first and second collision with the mesh). Then, if this point can be linked to the root, a point of the Point class of the skeleton is created and is linked to the root using a Tree (it becomes a subtree of the root). If not, other points are tried until one

that works is found or until too many tries have be done. If no possibility has been found then we go back to the first step.



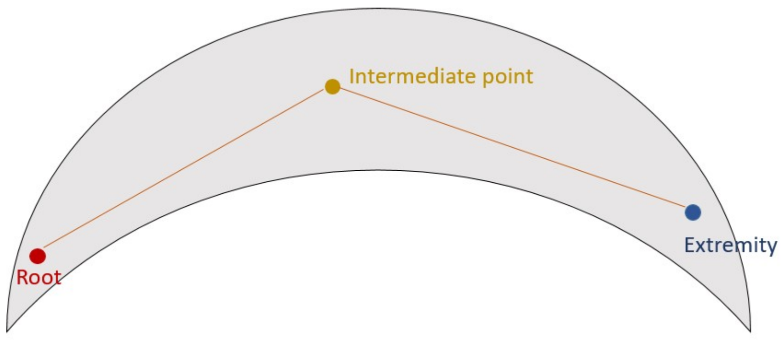
Finding a point in the mesh and linking it to the root

4. If the root has successfully been linked to an intermediate point, the extremity is linked to that point if it is possible, else the first step is redone, using the intermediate point just created as the root. The other extremities are tested to see if it is possible to link them to this intermediate point. If it is, then it is done else, the whole four steps are done on the next extremity, if it is not already linked to the root.



Linking the intermediate point to the extremity

The skeleton that would have been obtained would have been composed of the root which would be linked to all the extremities through the different intermediate points.



Final skeleton

The details as to why we did not do the automatic skeleton will now be explained.

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 get 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.

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.

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

3.4 Creation of motion

We had to differentiate two types of motion : the displacement of objects in space and the articulation of the model. The latter implies the deformation of the mesh and the use of the skeleton to create the illusion that the model is animated.

3.4.1 Displacement of objects

3.4.1.1 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.

3.4.1.2 Draw the trajectory

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 would 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.

Cf. Annexe 3

3.4.1.3 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.

Cf. Annexe 4

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.

Cf. Annexe 5

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 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.

Cf. Annexe 6

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.

Cf. Annexe 7

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.

3.4.1.4 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 vision.

3.4.2 Internal motion

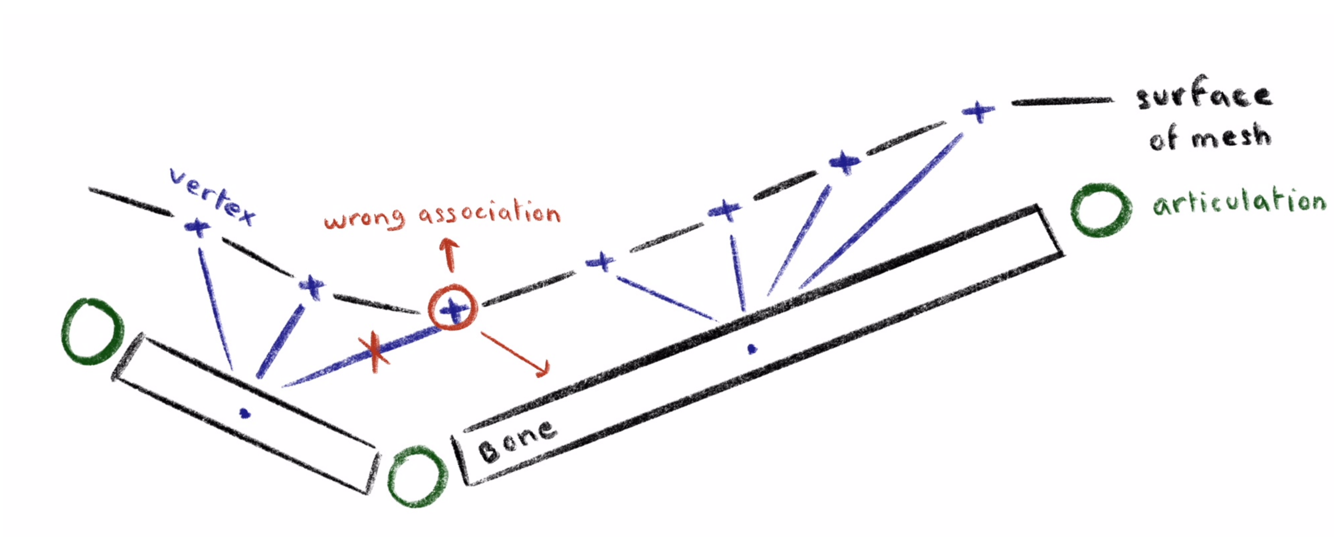
The internal motion of a model can be decomposed into two steps of creation. First, the skinning of the model, which means that each vertex of the mesh will be associated to a bone of the skeleton. Therefore, each time the bone is moved, the associated vertices will follow the movement. The second step of the internal motion is the articulation of the model: using a modified version of the motion keys, the user will be able to move the articulations and determine their movement on the timeline.

Skinning

In many other animation softwares, the skinning process is available in two options : manual or automatic. We decided to implement both types of skinning so the user will have the choice between selecting manually groups of vertices and then link them to bones, or launch the automatic process which will automatically create these groups.

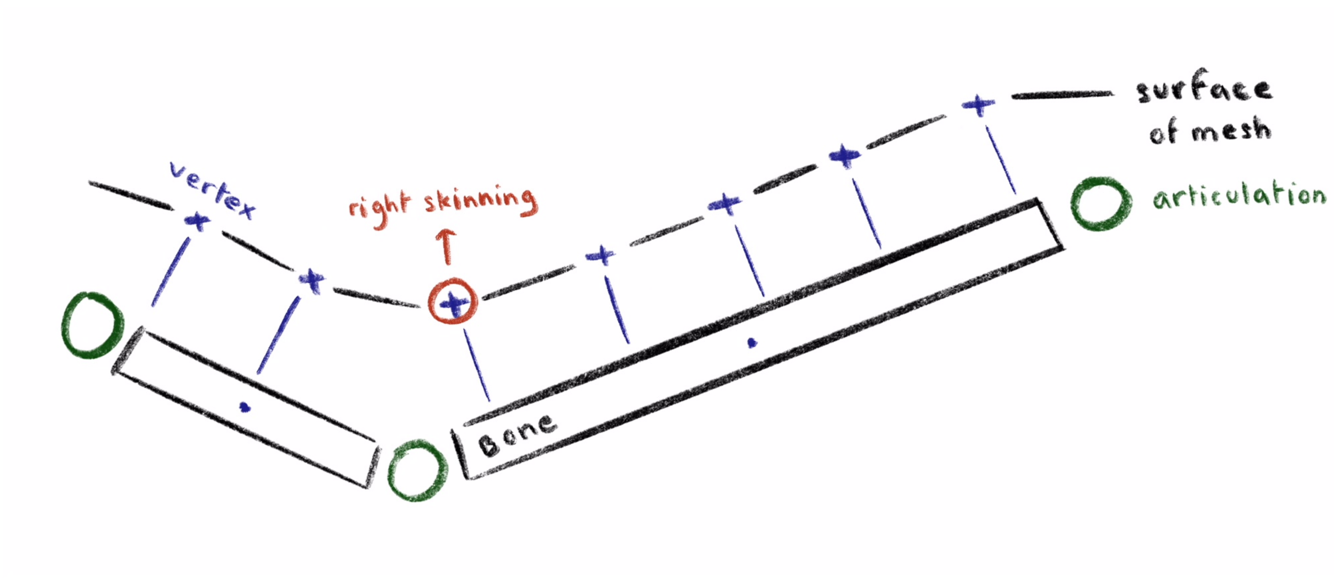
First of all, to manipulate vertices, we had to understand how Unity manages them. This was the same concern than with the automatic creation of the skeleton. The vertices of a mesh can be accessed using the GetVertices method which returns a list of Vector3s that represent the coordinates of the vertices. For rendering reasons, Unity duplicates all the vertices of a mesh by three, which means that there will always be three vertices at the same location. For example, a cube would normally possess 8 vertices but the GetVertices function gives back a list of 24 Vector3s with the duplicated vertices, in no particular order. If we want to move one of them, the other two must follow as well. Then, if we want to move vertices we should be able to modify their values but GetVertices only returns a list of Vector3s, not the actual vertices of the mesh, their coordinates could not be modified this way. Actually, a vertex is not an object so it cannot be set as a child of a bone and its value is read-only. I we want to modify on vertex we will have to modify the entire mesh. This was an important problem as the deformation of a mesh implied the modification of the vertices' coordinates. A solution was to create a script that would manage both problems : it will synchronize the vertices located at the same emplacement (if one is moved, the other two follow) and represent each triplet of vertices by a gameObject sphere. The fact that we represent the vertices by a gameObject makes it much easier to manipulate them and locate them into space. This sphere stores as attributes, three indexes which are used to access the vertices in the GetVertices list. At each frame, the program creates a new list of vertices depending on the location of the spheres and give back this list as the new set of vertices of the mesh.

For the skinning to be efficient, a vertex will generally be associated to the closest bone in the skeleton. The automatic skinning relies on this principle, it will travel the list of vertices, find the closest bone and make the sphere representing the triplet of vertices as a child of this bone. There is no predefined function in Unity to access the closest gameObject so our first idea was to go through the list of bones of the skeleton, compute the distance between the vertex and the bone and find out which distance is the smallest. However this method would not work in all cases. The location of a bone is determined by its center but if a vertex is located at an extremity of this bone, next to a smaller bone, the center of the second bone will be closer than the first. The vertex will then be associated to the wrong bone.



In this example, we see that the vertex circled in red is slightly closer to the left bone's center even if it is closer to the right bone. As a result it is paired to the left bone. With this method, in some cases, we will have wrong associations.

A better way to find the closest bone should be to consider the closest point on the bone to the vertex and take this distance to compare with the other bones. For this new method, colliders were very useful. Our strategy was to draw a sphere around the vertex with a small radius, at first. The radius increases at every frame. While the sphere doesn't collide with another bone, the sphere keeps growing. Once it has collided, the sphere representing the vertex is made child of the bone, the skinning is done. In this way, the distance vertex-bone is computed not according to the bone's center but to the point on the surface's bone that is the closest to the vertex. This method, contrarily to our first idea, works in every case.



If we take back the previous example, the second method has solved the problem of wrong association: the vertex is now attached to the right bone.

Articulation

The implementation of the articulations helps us to determine bones to which the vertices of a given mesh will be linked allowing to have a mesh deformation for the creation of the internal motion.

Two classes have been implemented : ArticulationKey and Articulation. The two classes combines the attributes from the Key Class which were a GameObject, a type, a value (corresponding to the new position, rotation or scale), a speed curve, a time information and a boolean to know whether the movement will be linear. However, creating a key for each articulation would mean having sub-timelines for every single articulation of a GameObject that might result into an overloaded screen. That's why we more or less splitted the Key Class attributes between these two new classes. ArticulationKeys will have as attributes the main GameObject we want to move the articulations, a time information and a list of Articulations. While the Articulations will have as attributes a GameObject that represents the articulation which is a sphere, a value which will be a rotation value, a speed curve, and finally the boolean indicating the linearity of the movement.

For the motion, an articulation will only be able to move around its parent and will follow it. For each articulation, an empty GameObject which has the same position as the articulation's parent will be assigned as new parent and this empty GameObject's parent will be assigned the former parent as new parent allowing us to handle the rotation. When this is done, every articulation is dealt with like any other GameObject.

3.5 Background imaging and sound

This allows the user to add objects that are not linked to a timeline and sound. The objects are just added onto the scene and are moveable on the scene, but during playtime, these objects stay still as opposed to the objects who have a timeline. The sound is linked to a timeline, so that it can be played at the right moment.

3.6 Particle animation

This allows the user to load particles onto the scene. These particles are preset and are from the particle system package of Unity. There are five kinds of particles, which are activated thanks to a timeline.

3.7 Website

The website was begun for the first presentation, the means we used to create it are the basic ones: HTML which stands for Hypertext Markup Language, used for the content of a page, and CSS which stands for Cascading Style Sheets, used to the handle the layout. We didn't use CSS much since we didn't have to build our website from the ground and were allowed to use templates that we could find on the Internet. Hence, the template we chose has only been slightly modified.

For the first presentation, the main layout had been set with the different pages we needed, each of them corresponding to a given section : Homepage, Book of Specifications, About Us, FAQs and Contact. Every page has an English version and a French version. Also, we have been able to host the website using a service provided by Github : Github Pages. This service allowed us to update the website easily.

Then for the second presentation, the Book of Specifications section had been replaced by a Download section. In this section our book of specifications, the different reports we made through the semester and the software are now available for download. Even though the pages have been translated, it is not the case for the different documents, every one of them but this report are solely in English. In addition to this, we decided to have a log book in order to keep track of what have already been done, the latter has been made using Google Sheets. Another change that had been brought to the website is the background music we added in order to create a relaxing atmosphere.

Cf. Annexe 8

Being aware the background music may not have the expected effect for everyone, a musicless version of the website has been created for the last presentation. Moreover the software and the manual have been added to the dedicated download section. You can check our website at : https://kairew.github.io/

4 Personal thoughts on the project

Brian

In my opinion, this project was a nice first step towards the future in which we may have to work with people we don't know. Because indeed, at the beginning of the year we didn't really know each others and throughout time from complete strangers we became teammates, and even more, friends.

We worked fine as a team, from time to time ideas about how to implement things may have diverged but it never ended up into a fight and we were always able to find a common ground in the end. This is also something I learnt from this project, diverging ideas don't mean someone is right or wrong it just shows us there is not only one way to think.

On another note, this project was quite unusual compared to any other group project I have ever taken part in. Most of them were only research work like the TPE where the aim was only to gather information, sort them out and finally present them whereas for this project, even though it needed this research process as well, it also required some kind of creativity in its construction. Not as much creativity as for games, but creativity in problem solving, every time an issue was encountered we had to look for a way to fix it even if it forced us to lower our expectations.

Since the type of project could be freely chosen, the choice of something other than a game was quite bold I believe, and we saw it early enough while wondering more than often why we didn't choose the game instead. It was even bolder and challenging if we consider that, for all of us, this project was one of the first - if not the first - coding project ever undertaken. All this considered, I think we can say we did well and I'm glad we're still alive.

Etienne

As I expected coming into this project, it was a huge learning experience. I came into this project very excited, and somewhat over-ambitious. Having a lot of great ideas and beginnings of answers regarding how to go about applying these ideas, I felt like I could finally do something big, accomplishing a project I had spent a lot of time on, and that I could be proud of. I knew my dreams would never hold, and I did lower my expectations, but it was definitely difficult for me to accept the fact that a first-year project cannot be as impressive as a fully fledged animation program.

Once I was able to get myself over this hurdle, the project was a fun and enjoyable project to work on. I learned a lot about my own working habits when it comes to working on code, despite the fact that this is far from the first long-term project I have had to work on. This project was somewhat different to everything I had experienced before. I believe what I learned the most from was to navigate between having high or low expectations for my coworkers, and enforcing these expectations. Overall I have little to no complains about their personal work or involvement in the project, but I believe that I could have done a better job making sure everyone was on the same page, and that we we're expecting group members to do something as they implemented something similar, but different enough to slow us down. Having these slight difficulties allowed me to further learn to work in a group, and especially in a situation where the work must be very well organized in order for the work process the function correctly, to hope to end up with a good result.

The project as a whole turned out as I expected it to after limitations of the Unity graphical user interface. This made for a GUI that is a lot less clean and user-friendly than what I hoped for, but I now realize I need to do more research on a tool before expecting certain results from it. The rest of the project is functional, which is to me a feat I am happy to have accomplished, as I had never attempted, nor even started to attempt, anything as complicated as a 3D animation program, even with the help of the Unity Engine and coordinates.

Léane

This project made us learn many things from how to use Unity to work in teams. It has been quite different to the projects that I had done so far as it gave us more freedom on how to organize our work, but we thus had more responsibilities. We had to do many research on how Unity works and how to make things work in Unity, but the use we had for it was quite different from making a game. There were similar parts to what a game should do but other parts were quite different and we had to think a lot about how we could take inspiration from functions others made in order to get the results we wanted. We also had to do a lot of research on the built-in methods of Unity, some of which did not give the results we expected, mostly because we did not understand the explanation given on the doc of Unity.

During the projects, I would say I was going from being in high spirit to being completely down. I often went from thinking our project was a good idea to thinking it was the worst idea we ever had and that we should have made a game. Even though there were hard moments when we all were questioning our choice, we still all went through it until the end. While a game is an appealing project, I am still glad we did an animation software because we had to think a lot about how we could distort Unity's original use to what we needed, and we had to be quite creative to do so.

The only thing I am sad about in this project is the fact that we did not complete everything we wanted to make. This was mostly because we were too ambitious and overestimated a bit our capacities compared to the time we had.

Our project was an interesting challenge for all of us, even more when thinking that some of us were not coding regularly before coming to EPITA, and none of us had done C# or try to use Unity before this year. At no point we got into fights, even when we had different ideas on how our software should be made, apart from some problems of communications that we had sometimes, for the most part we had good teamwork during the whole project. I think we managed quite well, we might not have done everything we wanted to do, but we did what we could to make the software as we imagined it and all of us made it until the end.

Julie

Working on the Frame Workshop was a new experience. Apart from the TPE, which was mainly about researches, I had never been part of a common project that implied coding. I had done some animation before this project and I was interested in understanding the code behind these animations. I felt some apprehension, at the beginning, as this was completely new and so different from what we have been asked to do until then, we had a lot more freedom and time than the regular programming TPs. This assignment asked a lot of work, most of all a good planning and teamwork.

From the start, this project was a challenge and we had sometimes to ask ourselves if it was possible. First year students generally choose a game and that is why we wanted to find a different idea, something that would stand out. First, none of us had experienced a coding project before this one and the main complexity came from the fact it wasn't a game. It was indeed very complex to adapt Unity to our needs as the software was entirely designed for creating videogames. Most of the tutorials found online were also dedicated to games and weren't of much help. In that sense, our project asked a lot of creativity to implement the software the way we imagined it. We sometime asked ourselves why we didn't choose a game after all.

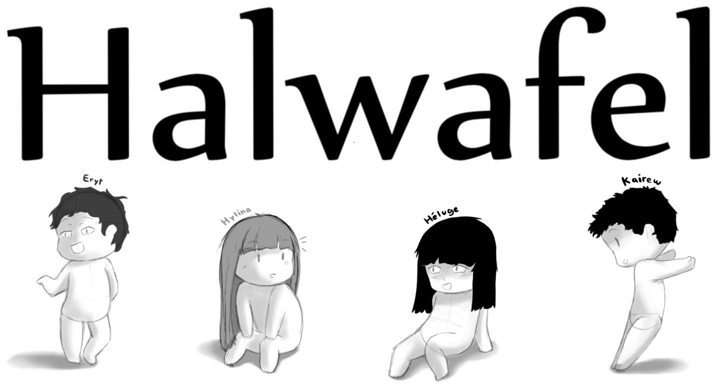
We hadn't much difficulty, on the other hand, about the teamwork. Every member of the group was very invested and cooperative. I am grateful that we had to work with each other. We had the work divided into different sections which correspond to different functionalities of the program, with people in charge of the GUI and others working on the animation, for example. This was a clever organization: each member was responsible of a part of the program and in this way, none of us was working on the same field as another person, therefore avoiding tension.

The choice of an animation software was not a bad idea, after all. It was possibly more complex to implement but I think it forced us to learn by ourselves how to use Unity. We were continually experimenting diverse ways to implement our ideas, and this helped us to fully understand Unity. We finally succeeded in almost every part of our project.

Conclusion

Our goal we set out to accomplish was to create a 3D animation software by using Unity's built-in engine to aid us along the way. Over the course of the last three months, we have had some great successes, as well as times of struggle with impossible problems, but despite all the ups and downs, we can say that we succeeded in accomplishing our goal. We have built a 3D animation software.

The windy path that lead us here had us go through notable problems with the automatic generation of the skeleton, which we sadly had to leave behind as it was causing to many problems for our project. This same path lead us to great success to a working motion and a bug-free GUI. Our determination and our will to do well kept us going through this treacherous way.



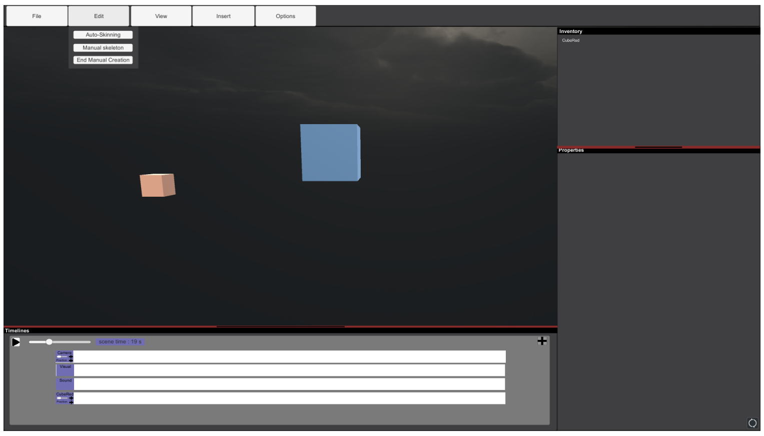
Halwafel is not a group, Halwafel is not a project; Halwafel is a way of life.

Thanks to every parent, friend and pet for their moral support.

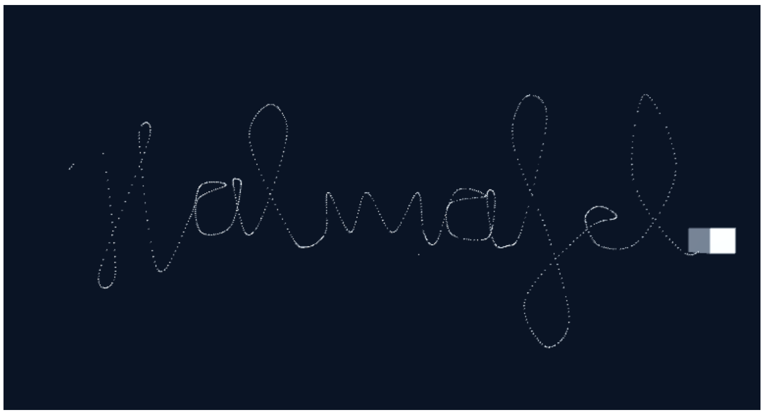
Annexes

The Frame Workshop

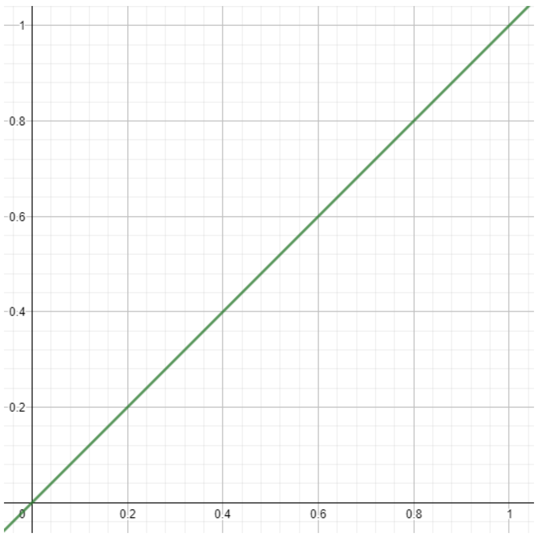
Annexe 1



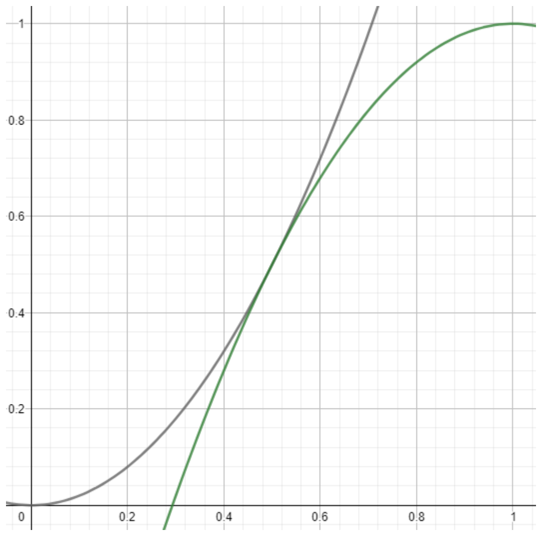
Annexe 2



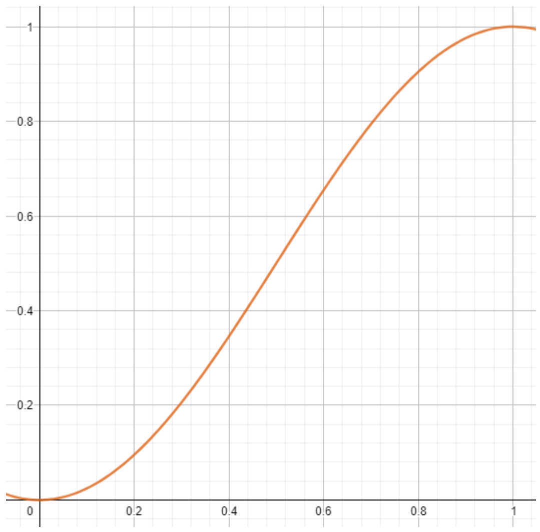
Annexe 3



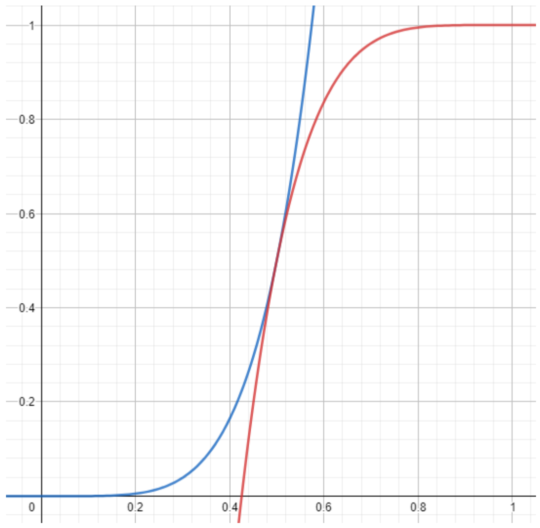
Annexe 4



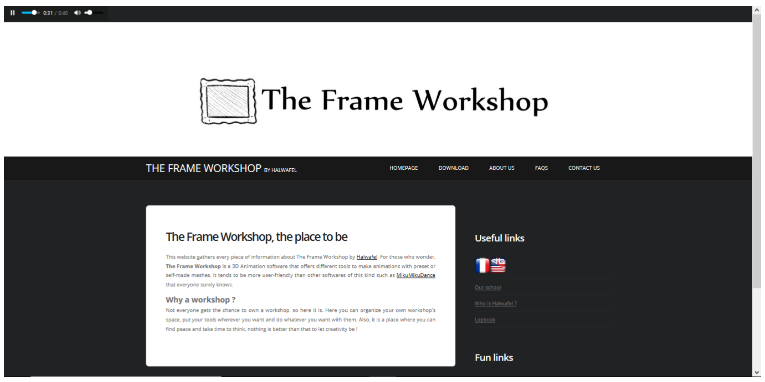
Annexe 5



Annexe 6



Annexe 7



The Frame Workshop, the place to be

This website gathers every piece of information about The Frame Workshop by Halwafel. For those who wonder, **The Frame Workshop** is a 3D Animation software that offers different tools to make animations with preset or self-made meshes. It tends to be more user-friendly than other softwares of this kind such as MikuMikuDance that everyone surely knows.

Why a workshop ?

Not everyone gets the chance to own a workshop, so here it is. Here you can organize your own workshop's space, put your tools wherever you want and do whatever you want with them. Also, it is a place where you can find peace and take time to think, nothing is better than that to let creativity be !

Annexe 8

Webographie :

- answers.unity.com
- autodesk.eu
- blender.org
- docs.unity3d.com
- ffmpeg.org
- forum.unity.org
- github.com
- learnmmd.com
- msdn.microsoft.com
- pixelplacement.com
- stackoverflow.com
- unity3d.com
- wikipedia.com
- youtube.com