TroMotion: Skeletal animation library

Bachelor's Thesis

Design and Development of Videogames

Surname: Mas Ortega Name: Iban

Pla: 2014

Director: Díaz García, Jesús

Index

Summary

Software development is one of the most common things nowadays. Mostly all that surrounds us have software integrated. To create that software, people work daily and create tools to ease the creation of new software.

The entertainment industry is one that most software develops every year. From special effects to videogames, software is key to create all films, adds and games we consume daily.

That industry relies on lots of libraries and applications created by third parties to bring alive their works and one of the most important tools they have for that is animations.

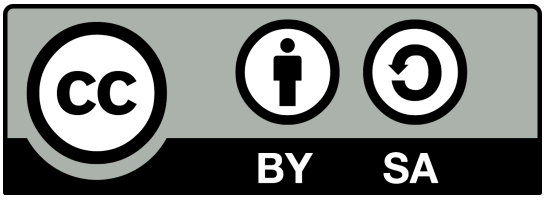
The animation system, against others, doesn't have a variety of third-party solutions to it since it usually relies on the core functionality of the base application.

TroMotion is an approach to bring a solution to the animation system focused on being easy to integrate into any source code and use the already developed solutions of that code to improve performance and resource usage. The library gives a solution to the most basic and common applications of animation such as blending, transitions and data management for skeletal animations.

Even though the specific approach will always give better performance issues, creating a common base that can be easily extended with extra features, is a good starting point for small companies and independent developers, as well as for students learning about the animation subsystem.

Key Words
3D animation, skeletal animation, library, C++, animation blending

Links
GitHub: https://github.com/Trodek/TroMotion



Index of tables

Index of figures

Glossary

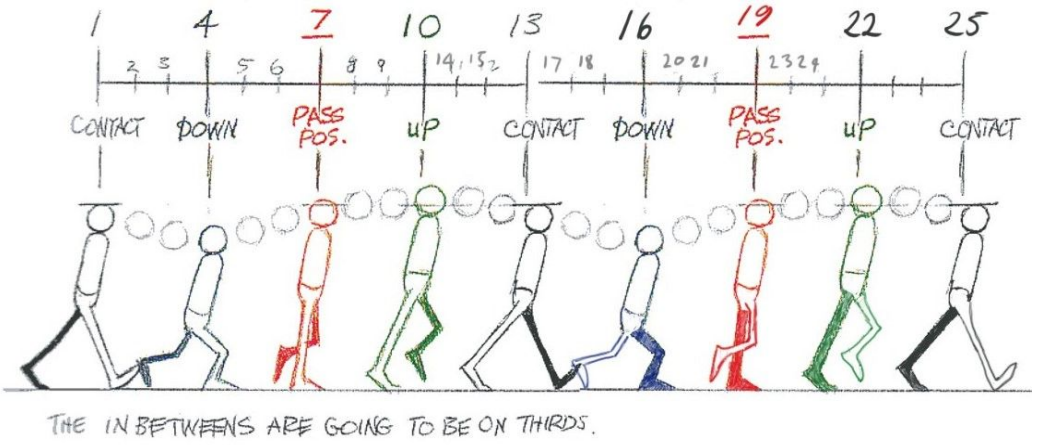
- **3D rendering:** Set of processes needed to convert a CGI, represented by mathematical points, into an image on a screen.
- **Agile methodology:** Approach to project management consisting of creating the project into small tasks assigned to specific members of the team.
- **Animation blending:** Technique used to mix two or more animations into one.
- **Animation layer:** Method that allows playing different animations on different parts of the same skeleton.
- **Animation retargeting:** Technique used to play an animation into a skeleton different from the one used to create it.
- **Animation system:** The subsystem in charge of handling all tasks related to moving a CGI.
- **Animation transition:** Procedure to change between two animations that allow effects to control that change.
- **Bind pose:** The position a 3D model has when creating the skeleton for it
- **Bones:** Empty space between two joints. Usually represented to show the relation between joints.
- **CGI:** Computer Graphics Imagery. Set of points and transformations represented mathematically that can be used by a computer to create images.
- **Feature-driven development:** Type of agile methodology on which the tasks are the features needed to develop.
- **Game engine:** Software with all the tools needed to create a video game.
- **GPU:** Graphics processor unit. The computer component in charge of performing all computations related to graphics.
- **Inverse kinematics:** Technique used to adjust the transformation of a set of joints to match a target.
- **Joints:** Each of the points used in skeletal animation to move a certain part of a CGI.
- **Key pose:** Specific pose used to create animation clips.
- **Metachannels:** Set of information non-related to the original objective used to identify and track specific aspects of the original one.
- **Phenakistoscope:** Circular object with images inside and holes in the sides that plays an animation when rotated at a certain speed.
- **Pose:** Unmovable position represented into a skeleton.
- **Rag doll:** Technique that simulates physical reaction for elements not intended to be physically realistic.
- **Real-time software:** Application that calculates and produces the image to show in a screen fast enough to fool the human brain to think there is no delay between them.
- **Rigging:** Name of the process of creating a skeleton for a 3D model.
- **Serialization:** Process to store data into files and allows load them for later use.
- **Skeletal animation:** Technology that allows creating motion for computer-generated imagery (CGI) which consists of moving an internal set of joints that conforms a skeleton.
- **Skinning:** Technique used in animation to associate skeleton joints to CGI objects.

- **Subsystem:** A part of a software that takes care of a specific group of tasks from the same area.

1. Introduction

In current audiovisual media, and thanks to the evolution of technology, the use of special effects and fantasy creatures is very common. That allows creative directors and artists to create settings and environments that some years ago only existed in literature. To achieve that, lots of different technologies work together from rendering pipelines, to 3D creation software.
One of the more important parts of creating scenes relies on the movement of the elements that appear on them. Animation techniques are the ones responsible to do that.
Through animation, the models created that, at first, are static, become dynamic recreating the movements of a living entity, such as a human or animal.

The techniques used for 3D animation are the evolution of the previously existing ones for 2D animation. In traditional 2D animation, the sensation of movement is created by the change between images fast enough to fool the brain, which perceives the trick as the movement. The image F1.1 is an example of how walking images are created. This idea is translated similarly to the 3D animation but since the technology used to create these 3D elements is different, the way to create the illusion of movement is also different.



F1.1. Walk cycle in 2D animation - The animator's survival kit, Richard Williams

This thesis focus on one of these techniques, that is called skeleton animation. The name represents how the technique works. Using a set of joints, also called bones, an inner structure that represents how the model should move is created. This structure is called a skeleton. Then, the model is associated with this skeleton and then the skeleton is the one, as in 2D animation, changed to the different positions that create the illusion of movement.

In order to do that, different data structures and mathematical operations are needed, which, usually, are dependent on the other software that uses them.

This project consists of a possible solution to make those structures and operations the most independent possible of the software and what are the pros and cons of this approach against the existing ones.

1.1 Motivation

The idea of developing this project came to me while developing my own engine for a university subject. During the subject, we were exploring all the different subsystems and improvements that can be incorporated in a game engine and for most of them, there was already a library that could be used to help with each subsystem. But once we reached the animation subsystem, the teacher didn't recommend the use of any of the existing ones for this subsystem since it is very reliant on core features of the engine.

Since then, the idea of creating a library able to handle animation for most game engines and software has been something I was thinking of doing by myself and therefore I decided to do it.

Creating a library is also a challenging feat that will allow me to grow not only as a programmer but as a professional since it requires to check the stability of it and correct functionality in different configurations.

1.2 Problem

When any programmer or company aims to create any software for 3D rendering, there is a point in the development that have to make himself a question: how will I treat animations?

The animation system is one of the core functionalities of any 3D engine or software, and that is also what makes it one of the most challenging subsystems since it requires to be integrated with most of the core elements of the software such as the transformations, model data and rendering systems.

Current solutions to this, usually include own solutions to some elements that the base software already has solved, which leads to code duplication and low performance. For that, current libraries are not used and their own solutions are developed since they create performance issues and integration with the base code is complex.

The main focus of this library is to solve these issues and bring a base solution for the animation system, which can be expanded and modified depending on the purpose of each user.

1.3 General goals

This thesis main objective is to provide a software-independent solution for skeletal animation in 3D graphics. That includes the creation of data structures for all the elements that take part in skeletal animation.

Objectives:

1. Design and develop a skeletal animation library
2. Allow the user to play and interact with animations
3. Create tools to personalize how the animations work
4. Release the library under GNU General Public Licence
5. Analyze the pros and cons of this solution against others
6. Develop an example engine to integrate the library

These objectives will create a library with the characteristics described in the table below:

No base code dependent	
Description	The library will provide all the necessary information and structures to allow it to be integrated into most of the current software.
Consequences	The developers will have access to a kit of tools that will allow them to use already created solutions and optimizations working inside the library
Personalized animations	
Description	The user can decide how to reproduce the animation, how to mix them and when to be notified by the animation system.
Consequences	The animations can be reused in any way possible, different interactions can be triggered when the animation starts, ends or on a personalized time.
Real-time	
Description	The animation system will be able to work on real-time software and engines
Consequences	The developer will not see any performance issue between before and after using the animation library
Able to be recreated	
Description	All structures will provide a serialization mechanism to allow them to be reused in the future
Consequences	The developer can save all the work done into files and load them to continue working with them

T1.1 Capabilities of the library

1.4 Specific goals

From the previous general objectives, the following is a more detailed list of objectives needed to achieve each of the goals:

Base code independence

To separate the base code from the library, but to use the optimizations already existing:

1. Math abstraction: provide a file that allows the use of the base code math solution in the animation library.
2. Data types: the creation of all different data types needed to store and use the animation data.

Personalized animations

To allow the user to create different effects from a set of animations, the library provides:

1. Transitions: allow to define how one animation changes to another one
2. Blending: Animations can be mixed to create a specific one (ex. change from walk to run)
3. Layers: the layers system allow to divide how animations are applied and to use different animations on specific parts of the model

Real-Time

In order to allow the library to perform the better possible, it will feature:

1. GPU computed animation: most of the computation will be done on the graphics card, which allows better performances
2. Data selection: to reduce the amount of data needed to send to the graphics card, the library will check what is needed at each point and only send the required data

Able to be recreated

One of the most important points on the whole system is for it to be able to store and load the configurations and all previous objectives.

1. Serialization: include native serialization for all the components on the animation system and provide a guide for creating own save files.

Easy to use

One of the main objectives of a library is to simplify the implementation of software:

1. Documentation: information on each component and how to set up the library guide will be provided to the users.

1.5 Project scope

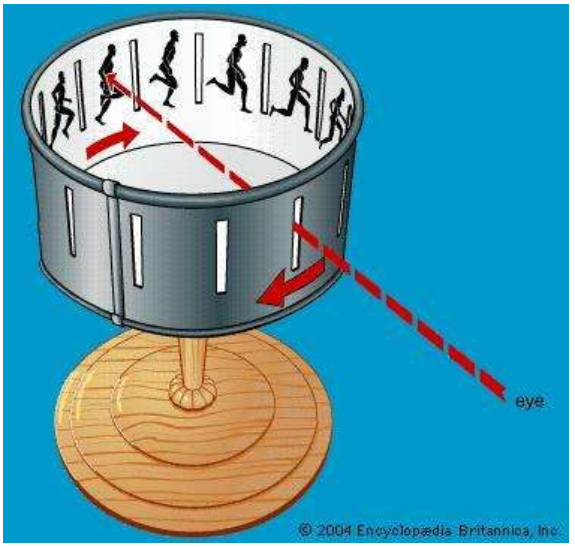
Animation is one of the key systems in any 3D software. Nowadays, it is used from video games to the advertisement industry. Creating a full animation subsystem inside this project is not possible because there are a lot of improvements and additional elements that animation systems include nowadays.

This library aims to ease the creation of any of the tools used with 3D rendering, more specifically for educational purposes or small companies since big companies already have their own solution to this system.

Companies and more specifically programmers are dealing with animations each time they create any new tool or engine. The library aims to create a base from which other improvements can be created and added. Also, it can be used as a base to develop more complex animations systems on top of it.

2. State of the art

The animation is one of the core techniques used nowadays in cinema, games, and advertising among other fields. The techniques used for animations are related to the evolution of cinema and, more specifically, cartoons.



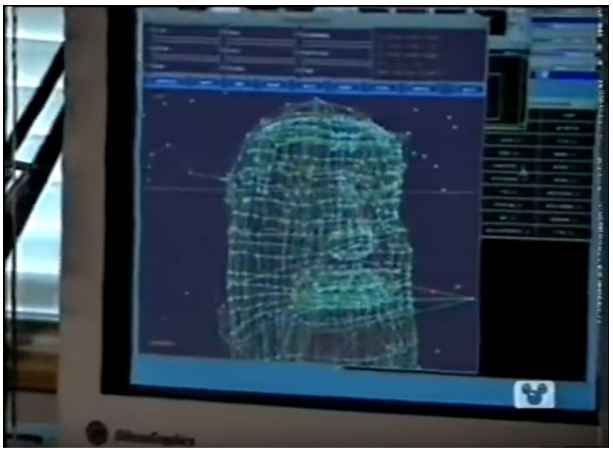
F2.1. Phenakistoscope - www.britannica.com/art/animation

The basics of animation consist of fooling the brain to think that a series of static images are creating motion. Since prehistoric times, we can find references to that art in some paintings but to really start perceiving the motion we need to look to closer to our days. The first we can consider an animation was the phenakistoscope(4) (see figure F2.1), a disk with images inside, which create the effect of motion when spinning.

After that, the invention of cinema allowed the evolution of animation in different fields that lead to the creation of some techniques that are still used nowadays.

But wasn't till Walt Disneys' Mickey Mouse that animations start seeing a huge change. Till then, more and more techniques and technologies had been used in animation until computers allow to create animations.

When talking about computer animations, the first thing that comes to anyone's mind is Pixar and Toy Story and that is not rare since it was the first movie created with computers using what is known as CGI (Computer-generated imagery), as shown in F2.2. But the process to develop that kind of animations with computers was still only available to a few projects.


F2.2. CGI from Toy Story - The making of Toy Story, 1995

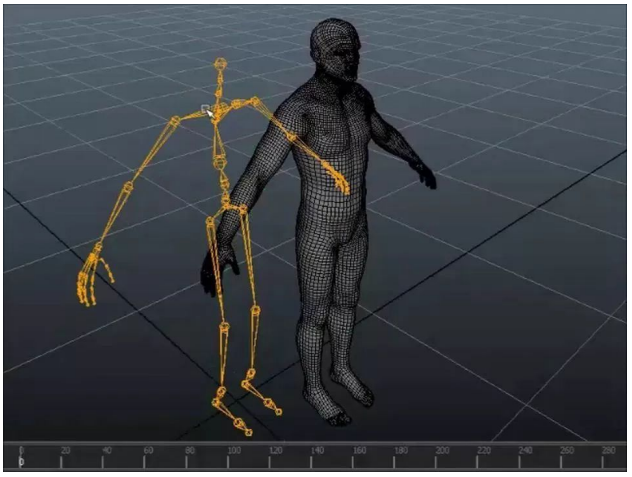
With the evolution of technology and computer science, the creation of animations like Toy Story starts becoming more usual.

2.1 Skeletal animation

The evolution in technology allowed to introduce animation in other fields, not just cinema. That also helps to develop more techniques to create animations. Till the point that animations, nowadays, are processed in real time.

The technique this project focus is one of those techniques and the most used nowadays, skeletal animation.

Skeletal animation tries to reproduce what a human or animal skeleton does. In biology, all vertebrate animals have a skeleton, which holds all the parts of the body and allows for other parts of the body to create motion. Taking this idea, skeletal animation relies on creating a skeleton for the computed generated model creating bones and joints.



F2.3. Model and Skeleton - http://www.cgmeetup.net

Joints are a set of key points on the model, which allow creating the motion.

Bones represent the empty spaces between joints and how these bones are connected.

The process of creating a full skeleton for a specific model is called rigging. An example of that is the F2.3 where there are a model and the skeleton next to it.

Then, through a process called skinning, the model vertices are assigned to one of these joints, which will be the ones moved and, therefore the vertices themselves.

As Jason Gregory stated in the book Game Engines Architecture(7), in skeletal animation, the pose of the skeleton directly controls the vertices of the mesh, and posing is the animator's primary tool for breathing life into characters.

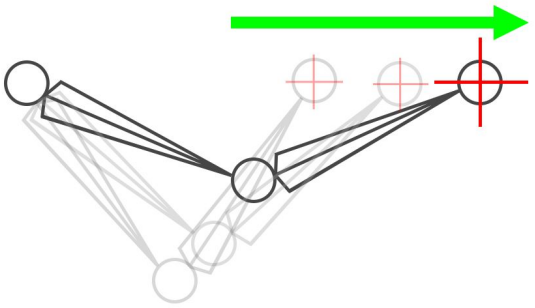
With that poses, the final animation is created. To create them, the animator creates a clip, which consists of a series of consecutive poses that perform one motion. Traditionally, to create the illusion of movement, between 30-60 poses per second are needed.

Creating that amount of poses is not efficient nor for the animator or the software. In order to solve that, forward kinematics are used. This technique consists of knowing the positions for a pose and the positions for the next one, then using mathematical interpolations, the software can reproduce the in-betweens. The poses used for that technique are called key poses.

2.1.1 Related Technologies

Skeletal animation itself will not allow creating what we see in current films, video games, and entertainment media. having skeletal animation as the base, several other techniques were developed.

1. Blending: Blending allows to mix two or more animations, which is useful to adjust animations to different situations.
2. Partial Blending or Layering: the layer system allows to play different animations at the same time for different parts of the skeleton. A clear example of this will be an animation to hold and a run animation. Using layers we can create a run holding animation.
3. Inverse Kinematics (IK): Inverse Kinematic (see F2.4) allows to modify the animation calculating the correct position of animation for a specific moment. A grab animation is usually modified with IK to go always to the target but the original animation is always the same.



F2.4 IK Example - https://docs.unrealengine.com

4. Animation Retargeting: Until now, skeletons and animations had been related one to another, and usually that is the case. Animation retargeting allows playing an animation made for a skeleton into another skeleton with a similar structure. This technique allows saving time when creating new characters or elements with animations.
5. Metachannels: Sometimes, other elements of the software need to be synchronized with animations. Metachannels allow that notifying the software when something happens on the animation, for example when a clip starts or ends.
6. Rag dolls: when a character or element needs to behave following the rules of physics instead of preset animations, rag dolls are used. Rag dolls simulate how the character will move according to physics, it is used when a character dies or goes limp among other cases.

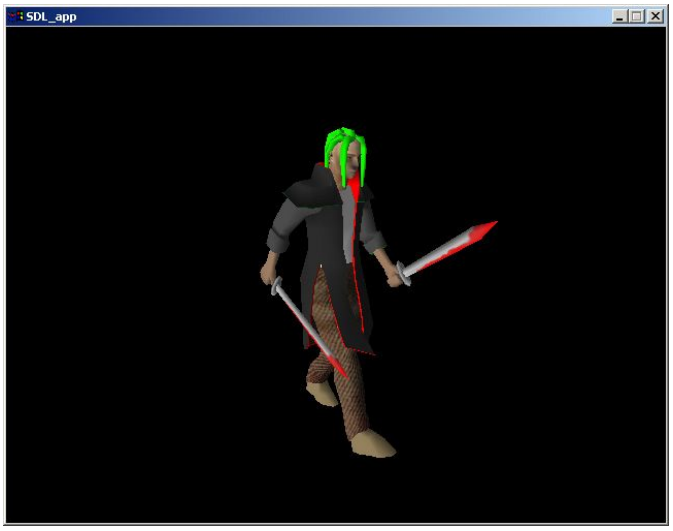
2.2 Existing solutions

When talking about existing solutions for animation systems, we have to divide them into two groups: libraries, and specific solutions. Libraries try to give a general solution to animation while specific solutions only work for the software or engine implementing them. In the paragraphs below I will list some of the existing solutions for each one of these two categories.

2.2.1 Libraries

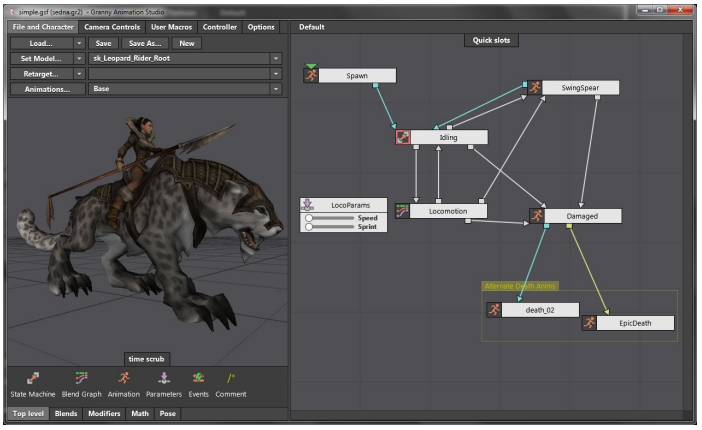
There aren't many libraries that try to solve the animation system, the most important ones are:

1. Ozz-animation: This is a skeletal animation library that focuses on character animation functionalities. Ozz-animation provides tools to load skeletons and models from most common files, reduce keyframes and compress the data.
   Also includes a mathematics solution to handle the needed operations. Ozz-animation introduces solutions for systems that other software usually have solved previously, such as model loading and mathematics, which will reduce the performance in the long term. The library hasn't been updated for about a year.
2. Animadead: Animadead (F2.5) is another library that gives a solution to skeletal animation, but it only handles the joint structure, gives a solution to load models, and some basic blending. Also, the library has its own mathematical solution. Same as Ozz-animation, including features that base code usually handles previously and introducing more than one solution to the same system (mathematics and model loading) affects the performance of the application. This library hasn't been updated since March 2006.



F2.5 Animadead Example - http://animadead.sourceforge.net/demoss.shtml

3. Granny 3D: Professional animation library from RadGameTools. It provides a full library with an external tool (F2.6) to import animations from the most used third-party tools (Maya, Max, and XSI). Their developers describe their piece of software as follows: "*Granny is a powerful toolkit for building all kinds of interactive 3D applications. We built Granny to be the most efficient and flexible animation system in the industry, but she also features a powerful set of exporters and data manipulation tools." [13]*

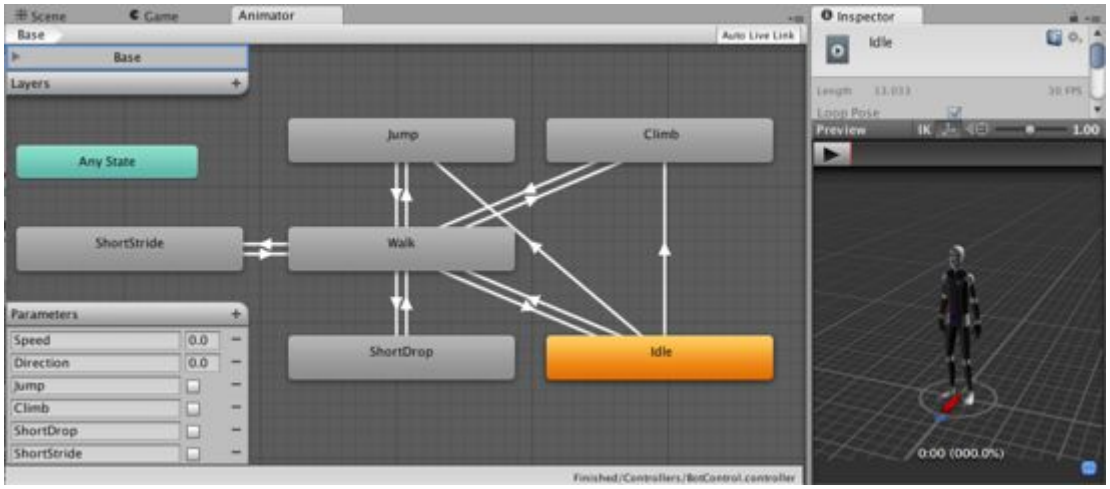


F2.6 Granny 3D Animation Studio. At the left the model, at the right a tool to interact with the animations - http://www.radgametools.com/granny.html

2.2.2 Specific Solutions

Nowadays, all software that features 3D rendering has a solution for animations. Most of this software creates its own solution for the animation system:

1. Game engines: one of the most important entertainment industries nowadays are video games. All characters in the games use animations to represent what is happening in the game. In order to create these games, game engines appeared some years ago and each engine has its solution to the animation system. Some of the most popular engines are Unity 3D, Unreal Engine, Godot, Frostbite, and CryEngine. All of them provide an animation system that gives all the functionalities talked previously to the users to create their games.



F2.7 Unity Animation. The image shows different animations and the transitions - https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html

2. 3D Modeling software: Another branch is the software used to create models and animations. This software also features animation systems that allow the creation of clips. The base for their animation system is the same as for game engines but, usually, they provide more tools to edit the joints and associate the

model to the skeleton. Among these applications, the most important ones are Blender, Autodesk Maya, and Autodesk 3DS Max.

2.3 TroMotion in the market

In the previous pages, the state of the art in skeletal animation has been defined. After analyzing the existing solutions we can set the bases for the work proposed for this bachelor's thesis.

Current solutions for this system providing the necessary tools for videogame development exist in two different flavors from the point of view of code independency. Some of them are designed to work as autonomous libraries that can be reused by custom game engines not integrating this functionality a priori. On the other hand, some others come integrated into existing game engines in a monolithic fashion. As we aim at reusability and code independence, we focus on the first type of solutions.

TroMotion idea is similar in spirit to Granny 3D: a library providing the essentials to solve the skeletal animation problem, that can be easily integrated into any game engine. However, Granny 3D is a commercial piece of software and for certain independent developers or students, that type of solution is out of reach.

Currently, it does not exist any free software that provides a base solution for integrating skeletal animation into a game engine. Furthermore, it does not exist any solution of this type, focusing on its easy integration into an existing game engine, and not redefining (i.e. reusing) its core tools such as its mathematics library or model loading system.

The aim of this thesis is the implementation of free software that provides a solution for the aforementioned issues. TroMotion is a free and open-source skeletal animation library, focused on its easy integration into an existing code base and easily extensible and adapted to the specific needs of its users.

3. Project management

Developing a library is not an easy task. Not only for the challenge in programming but also because it needs documentation to support it. During the development, this document has also been produced to reflect the success and issues faced.

To succeed in the development, planning of the time is mandatory. For that, the library development has been divided into three parts: documentation, the library, and the example.

Each of these parts has been divided into smaller parts corresponding to different stages of the development. The tasks for each part are in the following table:

Documentation	**Library**	**Example**
Introduction	Skeleton	Base Engine
Animation Research	Animation Clip	Library Integration
Development Diary	Animation Montage	
Library Documentation	Animator	
	Math Abstraction	
	Extra Features	

T3.1 Grouped tasks

All the tasks above are how the project has been divided. Each of the tasks can have smaller subtasks that will be specified in the project development section.

3.1 Procedure and tools for monitoring the project

To monitor and follow the correct development of the project, two tools will be used: the Gantt chart and Trello.

When working with software, the most common methodology to use is Agile. Agile allows tracking more efficiently in which tasks are done, which are ready to be reviewed and which are left in the backlog.

The Gantt chart is useful to have a global view of the entire project and check at any point if the development goes well or needs to be adjusted to fill in the schedule.

For more detailed tracking on each of the parts, the Agile methodology works better since it allows to track each task and the goals in a short period of time.

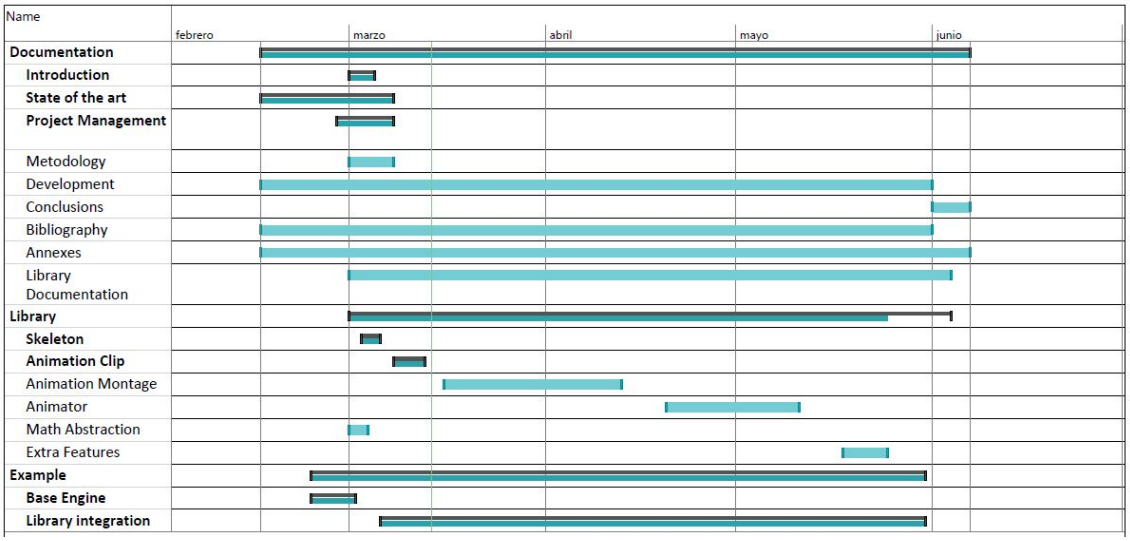
3.1.1 GANTT

In order to develop the project, first, a research period is needed. During this period, the work is focused on documentation. At this stage, the state of animation and how the project is approached is decided.

After the research, the development of the project starts where all the parts of the library are created as well as the example and the development diary.

Finally, the lasts two weeks are reserved for closing the project and elaborate the conclusions of it.

All this can be seen in the following Gantt table:



F3.1 Gantt chart for the project

3.1.2 Trello

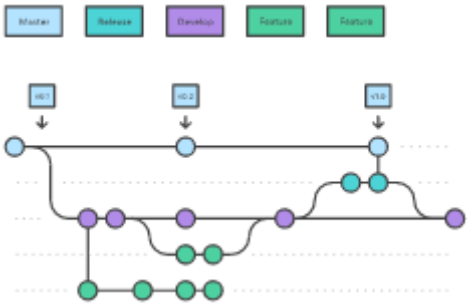
Trello is a great tool for Agile methodology. A Trello board with all the tasks of the project is used to track and manage the development of the project. The Trello board can be found in the Links section.

3.1.3 GitHub repository and version control tools.

A library is a set of code files, which can be precompiled or not, ready to integrate into any codebase. To develop code, version control tools help managing the versions and handling the changes into it. For that reason, a GitHub repository is used to host the code and documentation of the library in GitHub Wiki.

These tools provide workflows to ease the development of software and collaboration between developers:

- Commit: a group of changes done in the project that are uploaded together into the repository.
- Branches: A branch holds the state of the project from a specific point in time, branches can be created at any point from a previous one. All repositories have a default branch called master.



F3.2 Git Workflow. Different branches and their relations - https://es.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

The link to the GitHub repository can be found in the Links section.

3.2 Validation tools

The most important part of any development is to assure that the result works properly and is free of issues, which can cause problems for any user of the product.

Checking each of the features that the library has is key to ensure its correct behavior. For that, an example is developed along with the library. The example will use all the features of the library to show how they work and help to implement them in users' projects.

In order to test them, a model and animations are required. To ensure the quality and prevent any issue due to the errors on models or animations, the ones used for testing purposes came from Mixamo, a tool from Autodesk to create some basic animations to models. With the tool, they provide example models that are tested and checked by professional artists and animators.

During the development of the library, periodically checks had been done to ensure all features working together correctly. These checks lead to the creation of an additional tool with tests for the library that allow users to check for any possible error on their implementation.

More in detail validation processes are included in the project development section for the more critical parts of the project.

3.3. SWOT

Analyzing what the project will bring to the market is one of the key points to know if the development time and cost are worth it. SWOT is one of the most used tools for that that allow comparing form an internal and external point of view the positive and negative things the project will bring.

The table T3.2 analyzes the strengths, weaknesses, opportunities, and threads that the library can face.

	Positive	**Negative**
Internal	**Strengths** Prepared for base code integration Easy to include new features Well documented with examples Can be used for educational purpose	**Weaknesses** Requires more time to integrate Knowledge of the system is required User Interface has to be developed for each implementation
External	**Opportunities** Other libraries don't integrate with base code There aren't many animation libraries Most solutions are product specific	**Threats** Big companies have their own solutions No precedents on libraries being used in many products Can become outdated due to technology improvements

T3.2 SWOT

3.4. Risks and contingencies

Developing a project requires time and during that period, some issues can make the project development be slower or even stop it. In order to minimize the issues the project can face, T3.3 some of the main problems the project can face.

Risk	**Solution**
Computer issues.	A second personal computer is available to create the project.
Code corruption or lost code	A version control repository is used to have access to all changes and revert possible issues
Documentation corruption	The documentation is also saved in the version control and in cloud services
Time is not correctly estimated	A free week is counted into the schedule for extra features that can be used for delays reducing the number of features

T3.3 Risks and contingencies

3.5. Cost analysis

Even though the library is an open-source project and its goal is not making money, the creation of it requires some resources and time. In table T3.4, there is an estimation of time and the approximate cost of developing it.

	Estimated hours	Potential deviation	Planned hours (with deviation)	Cost
Documentation	93		135,25	2.028,75 €
Introduction	2	Low	2	30,00 €
State of the art	5	Average	6,25	93,75 €
Methodology	4	Average	5	75,00 €
Development	40	High	60	900,00 €
Conclusions	2	Low	2	30,00 €
Library Documentation	40	High	60	900,00 €
Library	72		104	1.560,00 €
Skeleton	2	Average	2,5	37,50 €
Animation Clip	6	Average	7,5	112,50 €
Animation Montage	30	High	45	675,00 €
Animator	15	High	22,5	337,50 €
Math Abstraction	4	Low	4	60,00 €
Extra Features	15	High	22,5	337,50 €
Example	70		105	1.575,00 €
Base Engine	30	High	45	675,00 €
Library Integration	40	High	60	900,00 €
Total	235		344,25	5.163,75 €

T3.4 Development cost by tasks

Taking that table into account and adding the equipment and services needed for the development of the library, the table T3.5 gives an approximation of the total costs of the development of the project.

Most of the costs for the project, come from the time invested in the development of it. The costs on software are zero since the tools used have a free service.

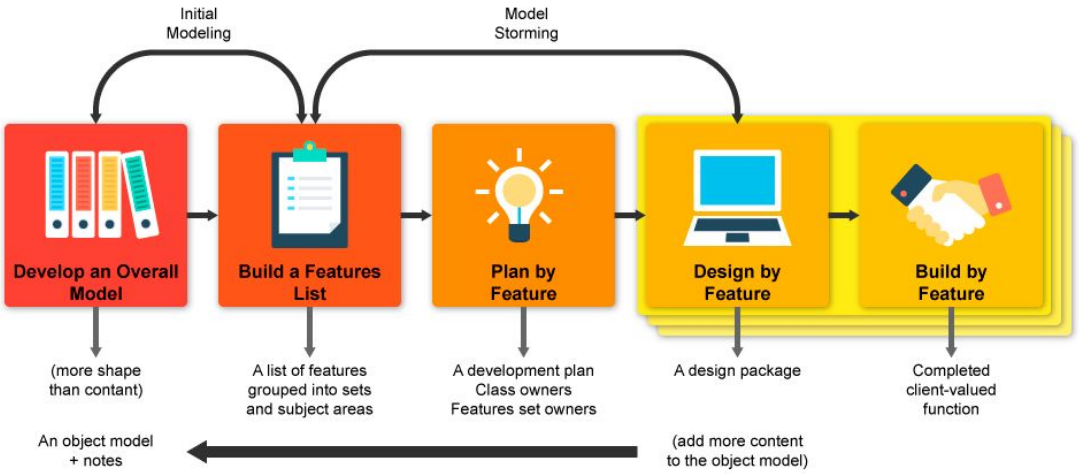
The equipment needed has a decent amortization time, which reduces significantly the impact it has on the total price.

Type	Subject	Price	Type	Amortization	Total price
Direct Costs					
Personal	Salary	5.163,75 €	Total		5.163,75 €
Equipment	Desk	150,00 €	Amortization	5	12,50 €
	Chair	100,00 €	Amortization	5	8,33 €
	Computer	1.250,00 €	Amortization	4	130,21 €
	Screen	130,00 €	Amortization	4	13,54 €
	Mouse	25,00 €	Amortization	3	3,47 €
	Keyboard	45,00 €	Amortization	3	6,25 €
Consumables	Notebook	4,00 €	Unique		4,00 €
	Pens	2,00 €	Monthly		10,00 €
Software	Visual Studio	0,00 €	Monthly		0,00 €
	GitHub	0,00 €	Monthly		0,00 €
	Trello	0,00 €	Monthly		0,00 €
Indirect costs					
Services	Electricity	20,00 €	Monthly		100,00 €
	Water	11,00 €	Monthly		55,00 €
	Food	25,00 €	Monthly		125,00 €
Total					**5.632,06 €**

T3.5 Total costs for the project

4. Methodology

To produce the library, as stated previously in the project management section, an agile methodology has been used. More exactly, a “Feature-driven development” methodology.



F4.1 Feature-driven development - newline.tech/blog/feature-driven-development-methodology/

This methodology focuses on developing a big image of the project itself and then breaking it into features that can be individually designed and developed. That allows creating smaller goals that are easily achieved and reduce the overwhelming of facing a big project.

4.1 Feature-Driven Development integration

This methodology relies on creating features that relate to a big picture. For that reason, the project has been divided into three parts.

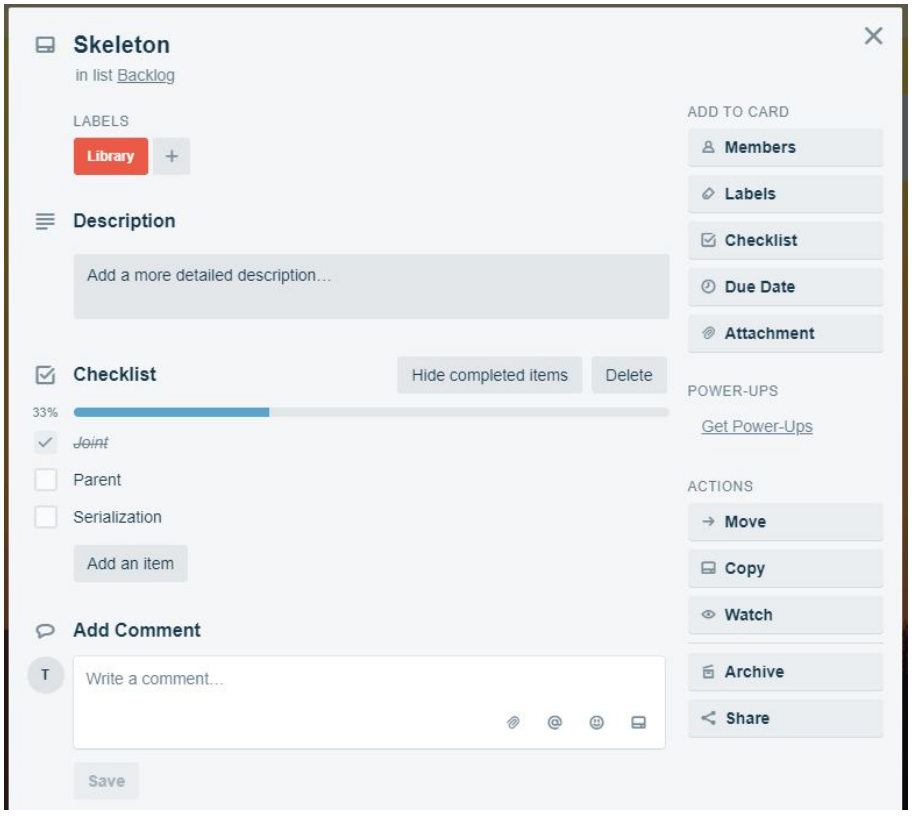
- On the documentation part, all tasks will be related to creating documentation of the library and the project report.
- The library part contains all the features of the animation system.
- The example part contains the ones related to providing an example and testing the library.

For each of the parts, the different tasks assigned need to be designed and adjusted to the schedule taking into account that some of them are related and some other dependants between them. As an example, when creating the blending between animations, the implementation of the example will be dependent and related to the library task and once it is tested, the documentation added to the documentation. Also, the explanation of the creation and decisions will be added to the project development section.

4.1.1 Tracking Process

To track the development of the features, Trello will be used. Being the first thing the creation of the task in Trello.

In Trello, the task will be classified into one of the three parts of the project. Each of the parts will have a color tag to make easier to visualize to which one it belongs and a checklist of the items to develop for that task. A created task is placed into the backlog.



F4.2 Trello task example

The backlog holds all tasks pending to be developed. Usually, for a library task, the corresponding task to implement it into the example and create the documentation about it will also be created.

When a task is taken to be developed, it is moved to the “In Progress” column at the Trello board. Each time one of the items of the checklist is finished, it will be checked and once all are checked, the task is moved “Ready to Review”

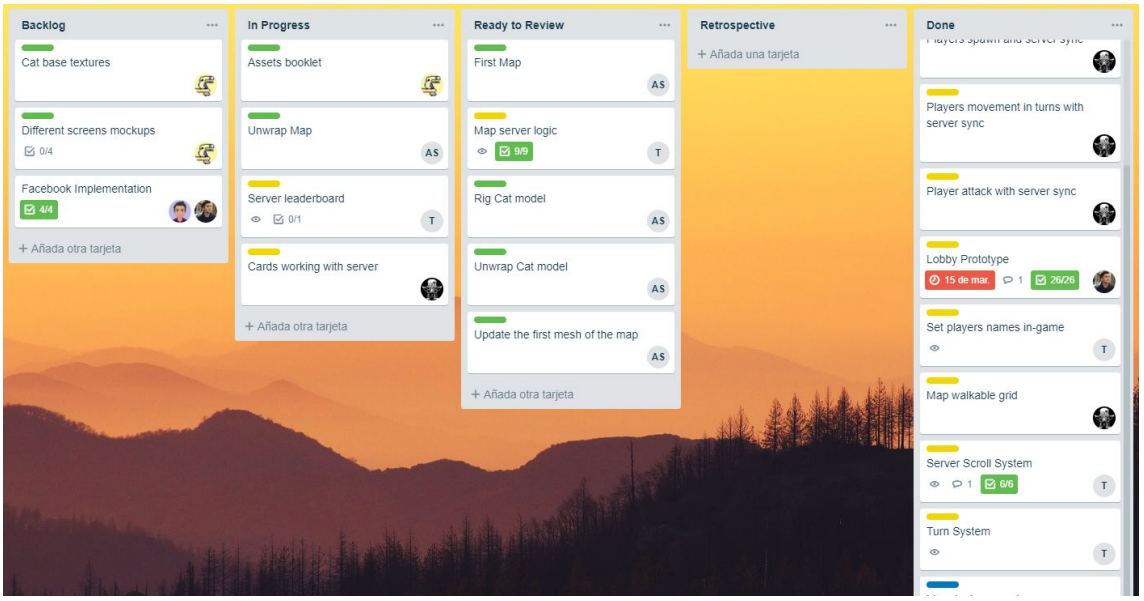
The ready to review column holds all the task finished but that still need to be tested. Being a solo developer, testing all the features in the library is not an easy task. So to consider a task reviewed, the following procedure is followed:

- A completed task is moved to ready to review.
- The implementation of the feature into the example is the next task developed.
- Once the integration is developed, all the features in the task are check individually.

- A fast check on all previous features is done to ensure the new one has not created any bug.
- If any issue is found, the tasks affected are moved to the “Retrospective” column with a comment about the issue.
- If there aren’t any issues, the tasks being reviewed are moved to “Done”

This flow leads us to two new columns of the Trello board. The retrospective column holds all tasks already developed affected by issues. While there are tasks in the retrospective column, no new features enter the development face because adding more features could make the issue more difficult to fix.

The “Done” column holds all the tasks already tested. Usually, tasks in this column shouldn’t move back again to other columns, but due to the continuous integration of features, some of them can be affected by new issues. A task in the done column that is affected by an issue is moved to the retrospective column with the comment about the issue.



F4.3 Trello board example

4.1.2 Project schedule

The feature-driven development, focuses more on each feature that in the big picture of the project and schedule. For that reason, a weekly check of the task completed against the Gantt chart of the full project gives a realistic view about the time left and time needed.

That will allow adjusting the schedule and tasks that will be implemented to fulfill the project scope.

5. Project Development

The entertainment industry has evolved into different branches. These branches include from cinema to lecture. Videogames are one of those branches and maybe one of the newer ones.

Videogames provide users with interactive experiences where they can become the hero of a fantasy story or test environments for professional training (flight simulators). Those experiences have all common technology that can be used to create thousands and thousands of different ones. The applications that contain these technologies are game engines.

A game engine is a software application where users can create their own videogames. These applications are complex and have lots of systems working together in order to allow users to create the game they want.

The animation system developed in this project is one of the systems any game engine includes. The system relies on some of the functionality the engine provides.

5.1 Testing Game Engine

During the development process, a continuous testing process is one of the main things needed to ensure the correct functioning of any piece of software. When developing any software application, the programmers have to ensure that the code of that application will work on any situation the application encounters. To do that, usually, the application goes through a Quality Assurance process. This process consists of testing each of the functionalities the application has and users can experience.

For the development of this library a way of testing it's functionality to ensure the users it fulfills its purpose is needed. This will require a game engine where to implement the library, but commercial game engines usually don't provide their base code to the user. For this reason, a small testing game engine will be created to fulfill this purpose.

5.1.1 Game Engine Structure

The first thing when talking about the structure of a game engine we need to know is the two different parts they contain.

The first of them is usually called the core of the engine. This part is the one responsible for the most basic things the engine must allow and holds the solutions for the management of the memory and elements the engine will use. The core of the engine handles how the resources are loaded and saved from and to files. It defines

the rendering functionality for the game and the structure that each element of the game will follow and how they are related.

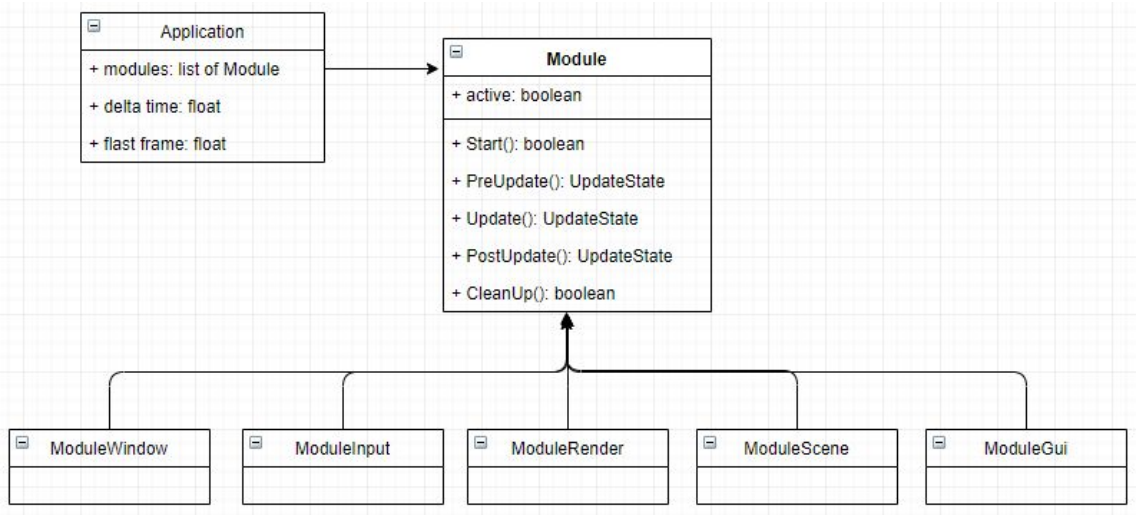
The second part of the engine are the subsystems. Each subsystem is responsible to add more functionality to the game engine and allow the user to create better experiences. The animation system this project is about is one of these subsystems. Some examples of subsystems are the GUI (Graphic user interface), the audio system and the particle system. All of them use the core of the engine functionalities to improve the capabilities the games created will have.

5.1.2 Implementation of the Game Engine

As stated before, the animation is a subsystem of the game engine. To create an animation system, the core capabilities of a game engine are needed. These capabilities are the ones that will be implemented in the testing game engine, which will allow proving the correct functionality of the library and help in the development of it.

In order to speed up this part of the development, since it is not completely related to the animation system itself, the resources from the webpage www.learnopengl.com will be used. These resources allow us to speed up some of the more tedious parts of the creation of the application.

The engine will have a simple but scalable design (F1.1) that allows implementing the animation subsystem as each of the other subsystems a commercial engine have. In the following sections, the integration of the library into this system is detailed.



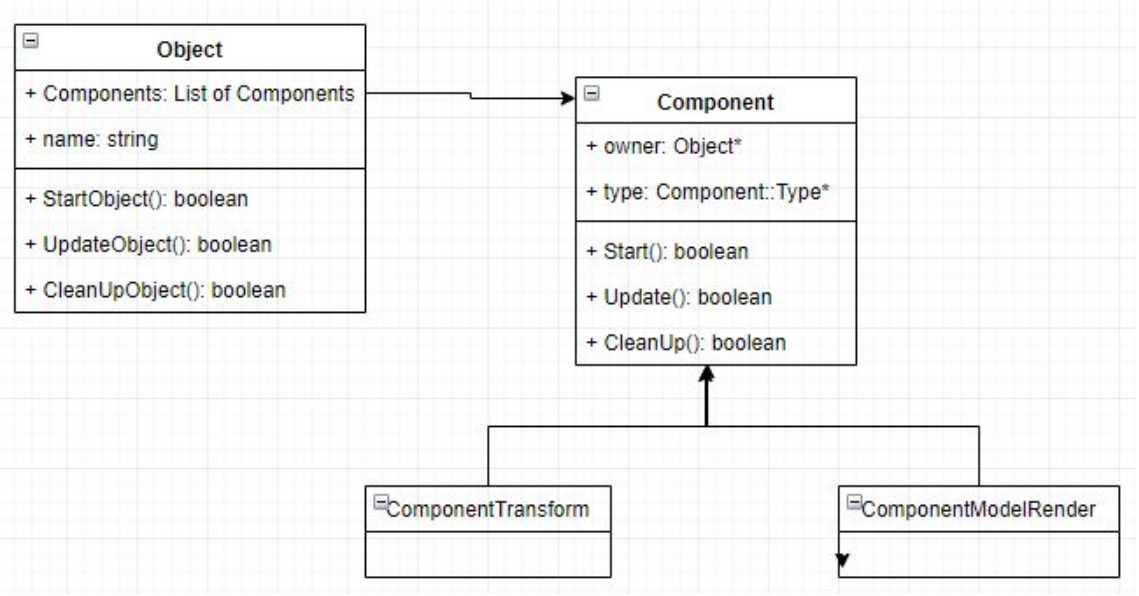
F5.1 Engine Structure

As the image F1.1 shows, the structure of the testing engine has as its main component the application itself. The application is the one responsible for managing all the systems of the engine. Each of these systems is called "Module" and are the ones that manage the logic for each of the necessary tasks the core engine does. In the concrete case of this engine, there will be no system managing the resources because for the purpose of this engine that concrete system is not required.

- **Application:** The role of the application can be defined as a cluster. The main purpose of this element of the engine is holding all the systems the engine has and to manage when each of them performs its functionalities.
  The application also records the time each frame of execution takes and to store this value.
- **Modules:** A module in each of the systems the game engine has. These modules can be active or inactive.
  All modules have some basic functionality the application uses to manage them. These are the five methods listed in the figure F1.1 in the module box: Start, PreUpdate, Update, PostUpdate, CleanUp.
  The Start method holds the logic for the initialization of the system.
  PreUpdate, Update and PostUpdate methods hold the logic that each module will execute each frame the game engine is running.
  CleanUp has the different steps needed to ensure the module frees all memory from the platform used.
    - **ModuleWindow**: This system is in charge of creating the window where the engine will run and handling the input from the window. The window consists of the space on the screen the application occupies and the buttons on top of it, as well as the title and icon.
    - **ModuleInput:** When the user presses a key or moves the mouse inside the space of the window, the module input registers these events and allows the game engine to react to them. In commercial engines, this system supports all kinds of input depending on the platform such as touch for mobile and controllers for consoles.
    - **ModuleRender:** One of the more important parts of the game engine is being able to represent the resources on the screen and that is the job of the module render. This module contains the functionality to convert the space represented mathematically in the computer into images on the screen. In this testing game engine also contains the camera, that allows changing the point of view of the user.
    - **ModuleScene:** This module is the one in charge of holding and managing the elements of the game. The structure this module uses is deeply explained later in this section.
    - **ModuleGui:** This module handles the different graphic elements the engine uses to display information of the current state of the engine and elements to the user.

When all the modules work together, the base of the game engine is capable of creating an environment where we can add elements to it. That elements are the ones that create the final game.

The elements have also been designed to be scalable and easy to use and follow the structure of figure F5.2.



F5.2 Object Structure

Each of the elements that compose the game is called an object. These objects are capable of holding different components, which add utility to them.

Similar to how the modules work, each component has basic functionality that the object uses to interact with them: Start, Update and CleanUp.

As the functionality of the module Start initializes the component, Update is executed each frame and CleanUp removes the memory used by the component.

The core engine has two components:

- **ComponentTransform:** The transform of the object represents the position, the rotation and the scale of that element in the game. Modifying the values this component holds, we can translate, rotate and scale any of the objects in the game. This component is mandatory and all elements in the game must have it.
- **ComponentModelRender:** As we have seen before, the engine is capable of representing the objects into images on a screen. In order to decide what should be rendered, this component holds the data of the resources that need to be drawn on the screen. The elements that are drawn on the screen are known as models.

Combining the different two different components the testing game engine has, the user is able to place a model into the application scene and see it on the screen.

5.1.3 Resource handling

For the purpose of this testing engine, there is no need for creating a whole system to manage the resources of the engine since it is created just to test the animation library.

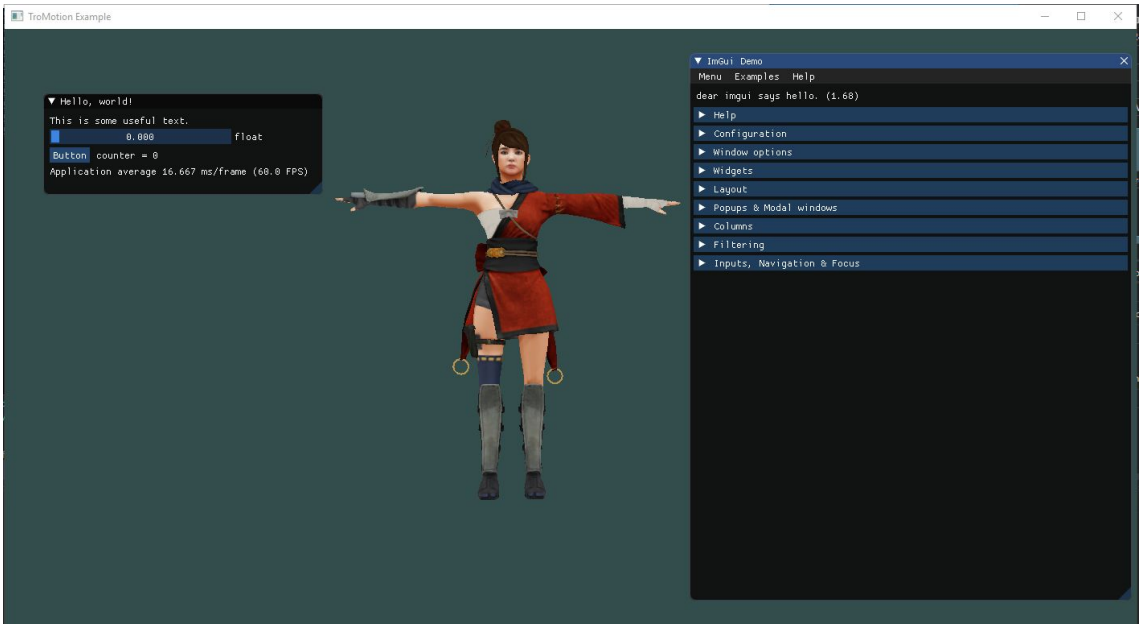
The handling of the resources is directly done by the modules of the game engine. The scene is the one holding the resources for the game objects and the render handles the shaders.

The core structure of the game engine handles three types of resources: models, textures and shaders.

- **Models:** Models are the points in the space that represent an element of the game. These points are used by the model render to create the representation on the screen.
- **Textures:** the models only contain information regarding the shape of the element. To allow the application to draw the correct color in the correct position textures are used. they store information about color and other details of the model which can not be stored on the model itself.
- **Shaders:** Shaders are small programs that the graphics card execute when the module renders draw each model. They allow personalizing how the model is drawn on the screen.

To handle these resources third-party libraries are used:

- **Assimp** (10): allows loading model information from different types of files.
- **stb_image** (11): this library loads images and textures in different formats both compressed and not compressed ones.



F5.3 Core engine working

5.2 Animation library

The main purpose of this project is developing an animation library to handle the animation system of any game engine. To do that, the animation library should be easy to integrate and provide an easy to use solution.

This chapter details all the elements and concerns the library is designed in order to fulfill the goals of the project.

5.2.1 Integration concerns

Integration is one of the most key topics the library has to handle. The integration of it should try to use any solutions the core engine already provides and not introduce custom solutions for them.

One of the most important points of any system a game engine has should be optimization. When the developers use third-party libraries and systems, usually, the performance these external tools will have is lower than a custom made solution due to the fact that a custom made solution will use the already existing systems and optimizations the game engine provides. On the other hand, external solutions for a system will provide their own approaches for systems that the core game engine already handles and generally, having more than one solution for a system working at the same time will result in lower performances.

To ensure that the library will not affect the performance of the game engine, identifying the systems the animations will use is key.

Shortly, the animation system is based on a structure made out of joints. Each join represents how a set of vertices from the model is placed. Having different placements for these joints allow us to create the animation. With this fast summary of what the animation consists of, the key elements that need to be handled can be determined.

First of all, the concept of a structure. The elements the animation system relies on are organized in a tree structure. To represent that, each branch of the tree stores the element they are related to. The structure is named the skeleton.

Joints are the next important word that appears in the definition. Joints are each of the elements the structure is made of. Following the definition, these joints represent the placement of a group of vertices. In a 3D environment, to mathematically represent the placement of an element, three properties are used: Position, rotation, and scale, and for three-dimensional space, each of them will have at least three numerical representations, one for each of the axis.

The concept of representing the placement of a group of vertices is what in animation is called skinning. Vertebrate beings have a skeleton that sustains their bodies and allows them to move. Using this concept from a digital point of view, the skeleton of a vertebrate is the skeleton of the animation system and the skin for a vertebrate is the vertices for a model. Then, skinning consists of assigning to the vertices one or more joins that will change its position.

After analyzing that simple definition of the animation system, it is clear that the system needs a way to mathematically represent the different properties stated above.

Mathematics is something that any game engine handles. As stated in the previous section 5.1 Testing Game Engine, the engine already uses mathematics to represent the position, rotation, and scale of an object.

In order to use mathematics in the library, there are two possible approaches: use a library-specific system or use the one that the game engine is already using.

To improve performance, the library will use the one the game engine has, but that creates a new problem: each engine will have its own mathematics library. Abstraction is a technique that can be used to handle that.

5.2.2 Mathematics library abstraction

The mathematics library holds representations for types such as vectors and matrices and how to operate with them.

Each library has its own nomenclature for the types and different ways to do the operations with them since that depends on how the library is programmed.

Creating variants of the code to handle different libraries is not something feasible nor optimal due to the fact that there are lots of different mathematics libraries and some of them are not available since they are owned by companies.

To be able to use the engine mathematics library and implement the animation system, a mathematics abstraction is needed.

An abstraction(12) allows handling complex code in a simpler way where the details are hidden and only the functionality is available. Abstractions can be done with classes and with headers.

- Class Abstraction: consists of using a structure of C++ called a class that allows to package functionality under it.
- Header Abstraction: a header is a file that contains code. To use a header as an abstraction, the header has functions that relate to the code we are creating an abstraction of.

Since at first there's no way of knowing how the mathematics library will work, a header approach is more convenient in this scenario. The header allows packing each of the functionality under a name and then editing that functionality depending on how the mathematical library works.

For the different types that the library defines, in the header file of the mathematics abstraction, a rename of the type can be set so the animation library will always use the same name. This is done using the keyword typedef form the C++ programing language, which allows calling an existing type by another keyword that the programmer assigns to it.

Then different functions will handle the complexity of dealing with the functionality of the mathematics library. This functionality ranges from creating a variable of a specific type to operate with them.

Not all of the functionality of the mathematics library from the game engine is used for the project. Only the needed parts are included in the header (F5.4). These types and functionalities are:

- Types
  - Vectors
  - Quaternions
  - Matrices: specifically 4x4 matrices
- Functionalities:
  - Type creation
  - Operations for each type
  - Operation between types
  - Conversion between types

```
namespace TroMo {

    typedef unsigned int uint;

    using TroMoMat4x4 = float4x4; // <math_lib_mat4x4>;
    using TroMoVec3 = float3; // <math_lib_vec3>;
    using TroMoVec4 = float4; // <math_lib_vec4>;
    using TroMoQuat = Quat; // <math_lib_quat>;

    inline TroMoMat4x4 CreateMat4x4(TroMoVec3 position, TroMoVec3 scale, TroMoQuat rotation)
    {
        return float4x4::FromTRS(position, rotation, scale);
    }

    inline TroMoVec3 CreateVec3(float x, float y, float z)
    {
        return float3(x, y, z);
    }

    inline TroMoVec4 CreateVec4(float x, float y, float z, float w)
    {
        return float4(x, y, z, w);
    }

    inline TroMoQuat CreateQuat(float x, float y, float z, float w)
    {
        return Quat(x, y, z, w);
    }

    inline TroMoVec3 TroMoLerpVec3(TroMoVec3 vec1, TroMoVec3 vec2, float t)
    {
        return vec1.Lerp(vec2, t);
    }
```

F5.4 TroMotion Math Implementation Code (Fragment) - The image shows the code used to implement the library used in the example application.

5.2.3 Skeleton and joints

As has been stated in different parts of the project, the animation systems rely on two different technologies that work together, the skeleton and the skinning.

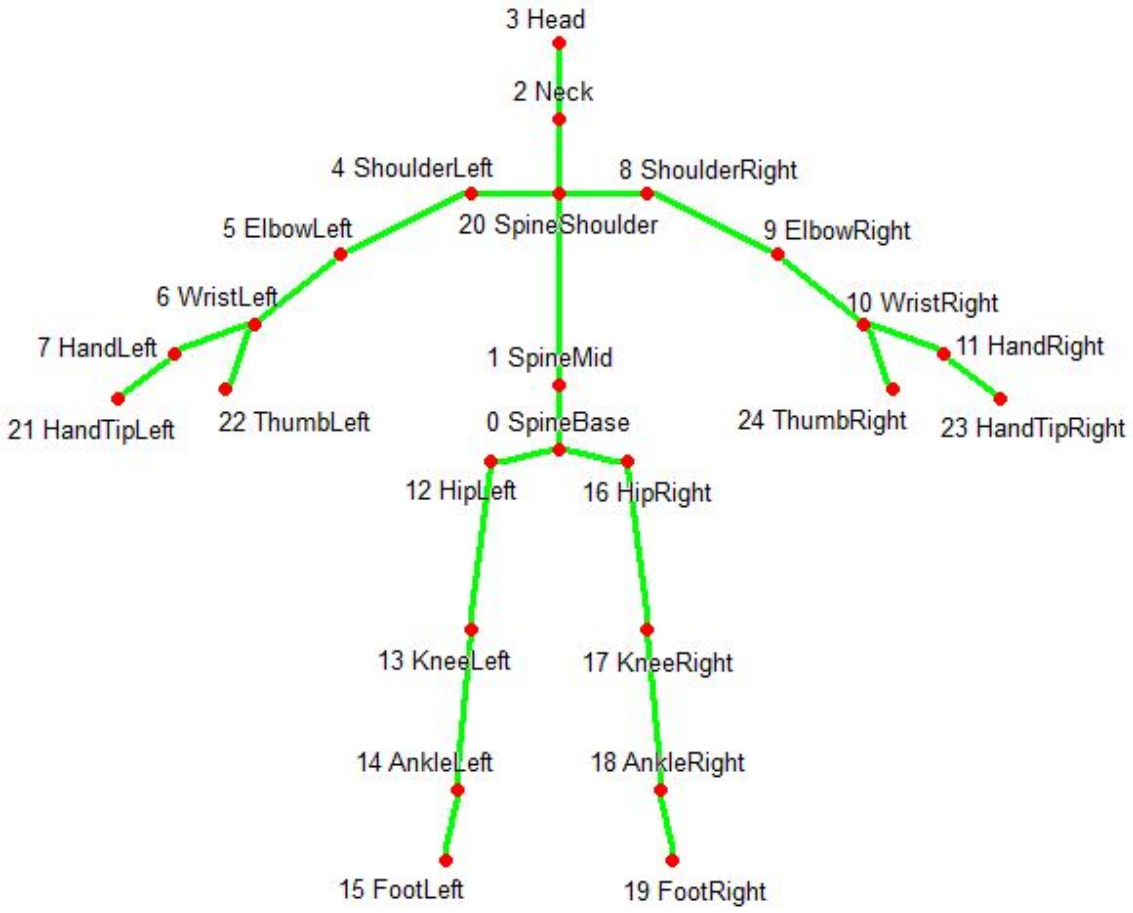
Skeletons are a simplified representation of a model. The most important part of this representation are the joints, which create the skeleton. These joints are organized in a

hierarchy that defines the relation between them. A joint contains the position, rotation, and scale of that part of the structure.

A typical structure for a skeleton hierarchy is the following one:

- Pelvis
  - Lower Spine
    - Middle Spine
      - Upper Spine
        - Right Shoulder
          - Right Elbow
            - Right Hand
              - Right Thumb
              - Right Index Finger
              - Right Middle Finger
              - Right Ring Finger
              - Right Pinky Finger
        - Left Shoulder
          - Left Elbow
            - Left Hand
              - Left Thumb
              - Left Index Finger
              - Left Middle Finger
              - Left Ring Finger
              - Left Pinky Finger
        - Neck
          - Head
            - Left Eye
            - Right Eye
            - various face joints
  - Right Thigh
    - Right Knee
      - Right Ankle
  - Left Thigh
    - Left Knee
      - Left Ankle

This skeleton structure and the one in the figure F5.4 are meant for humanoid-like models. Other type models such as animals can define a different structure for the skeleton. The creation of the skeleton for a model is called rigging.



F5.5 Skeleton with joints - The image shows an skeleton with its joints names - https://www.sealeftstudios.com/blog/blog20160708.php

To represent a skeleton and the joints in the library, the structure used contains the following properties:

- Inverse Bind Pose: the inverse bind pose is a matrix that stores the position, rotation, and scale of the joint at the moment it was attached to the model. This matrix is stored inverted to ease the operations the system will perform with it. The matrix transforms from the bone coordinates space to the model coordinates space.
- Name: Represents the name of the joint. That allows identifying easily the joint while working with it. F5.5 shows an example of joint names.
- Parent: corresponds to the index of the joint the current one is attached to. In figure F5.5, the parent of the joint SimeShoulder is 1, and the parent of HandTipRight is 11.

With the previous structure of a joint, we can define a skeleton as a group of joints, which will be represented in the memory with the next structure:

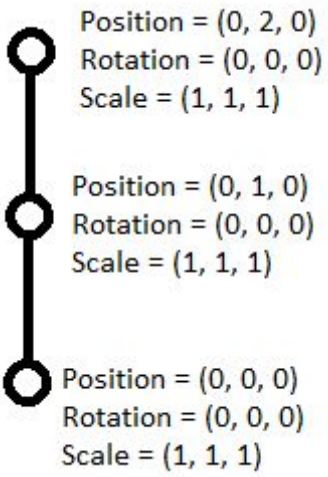
- Joint count: number of joints the skeleton has.
- Joint array: the data for all the joints that form the skeleton.

The code in figure F5.6 is the one used in the library.

```
namespace TroMo {

    struct Joint
    {
        TroMoMat4x4 inversePose; //inverse bind pose transform. This matrix converts from bone space to model space.
        char* name; // joint name
        uint parentIndex; // parent index id

    };

    struct Skeleton
    {
        uint numJoints;
        Joint* joints;
        uint id; // skeleton identifier

        // Returns the index of the joint with name <jointName>
        uint GetJointIndex(const char* jointName)
        {
            std::string name = jointName;
            for (int i = 0; i < numJoints; ++i)
            {
                if (jointName == std::string(joints[i].name))
                    return i;
            }
            return numJoints;
        }
    };
```

F5.6 Joint and Skeleton Code - The image contains the code used in the library to represent joints and skeletons.

To check that the structure works correctly, the following test skeleton is set-up manually:



F5.7 Test Skeleton Structure - Joints with their corresponding transformations for test skeleton

Using the skeleton structure in figure F5.7, access to the different members of the structure as well as the functionality to get the joint index is tested correctly without any issues.

5.2.3 Poses

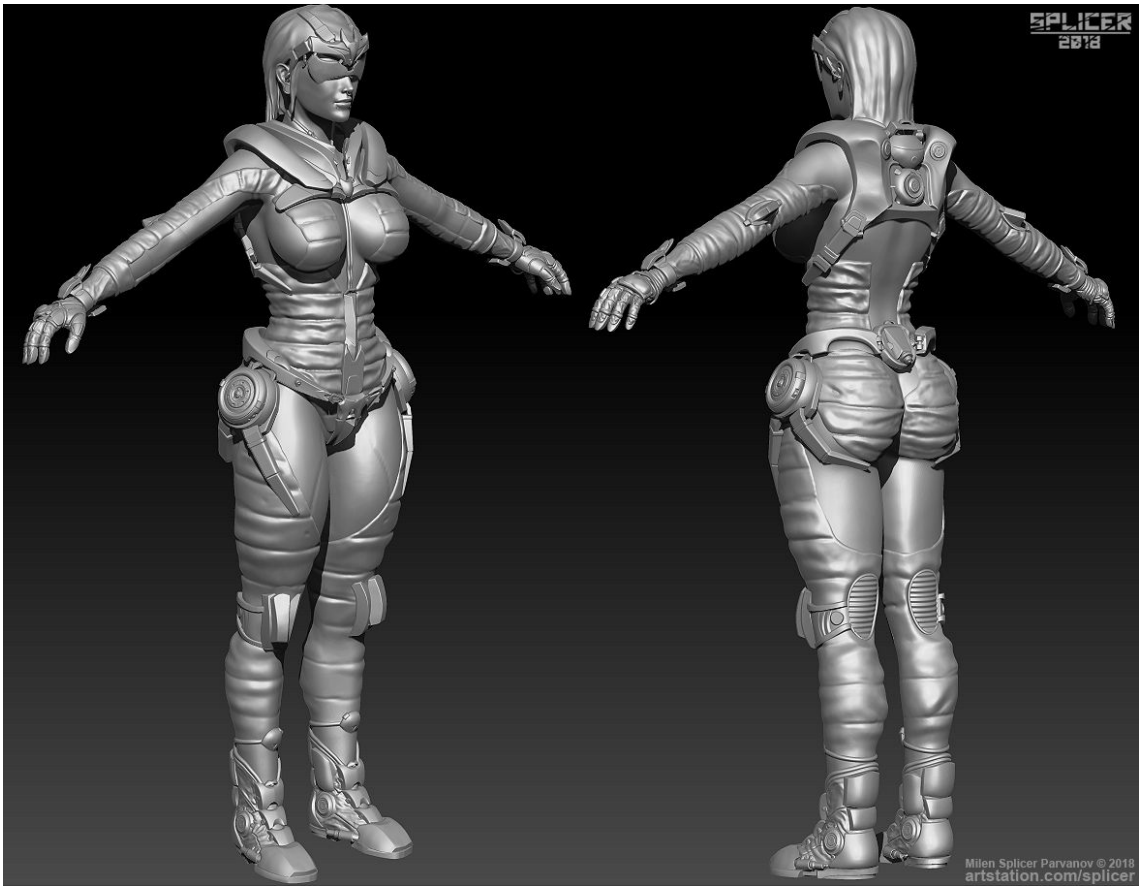
An animation consists of the illusion of movement by changing the position of each part of a model in a small period of time, which makes the brain think the model is moving.

These movements can be created using different methodologies, from manually moving each joint to capturing the movement of an actor. Independently of the methodology used to create the animation, these movements can be divided into poses.

Poses are the state of animation for a certain moment on time. In computer graphics, each of the moments of time that is represented into the screen is called a frame. In animation, each pose is related to a frame, for that reason, the number of poses per second is usually called frames per second (FPS).

When talking about poses, there is a special one that needs to be taken into account which is called **Bind Pose**.

The bind pose is a special pose since is the one used to bound the skeleton. This pose, which is also called reference pose or rest pose, has special characteristics the other ones don't need to have, being the most important one that in this pose, all model articulations are the most extent possible. In this pose the model, if it is a humanoid, has the arms stretched and the feet slightly separated what remains to the letter **T**, hence the name T-pose (F5.8).



F5.8: T-Pose Example - Model of a character posed in T-Pose - https://polycount.com/discussion/202303/to-t-pose-or-not-to-t-pose

Before going to the actual implementation of the poses into the animation system, there's something to take into consideration. For most of the cases, the scale of the joints in poses can be considered uniform. That will simplify the structure of the joint pose in memory. For the library implementation, this will not be used since some animators can use the scale component to create their animations.

The only consideration about memory, since memory is packet in groups of 4, the scale and translation will be stored with 4 variables each one instead of 3.

The pose implementation has two parts, the transformation of a joint and the pose itself.

For the specific joints transformations in the pose:

- Rotation: a quaternion will store the local rotation of each joint.
- Translation: local position stored in a vector of 4 values.
- Scale: local scale stored in a vector of 4 values.

With this structure for the joints in the pose, the following structure defines the full pose:

- Local Joint Poses: an array of all the joints for the pose
- Global Pose: matrix array precalculated to transform into that joint from world space.
- Skeleton: reference to the skeleton this pose is done for.
- Frame: the frame in the full clip

5.2.4 Animation Clips

Once the system is able to handle the data needed, the next step is to combine it and add some control variables to it.

That is the meaning of the animation clip. An animation clip represents an animation. A clip can be a walking cycle that loops and then starts back or a full animation containing all the movements for a character (run, jump, attack…). Initially, that depends on what the animator has created it.

add pose key info and justify why pose not used

Animation clips are the first element of the system that allows the user of the game engine to personalize how the animation will play and how it will look like in the final game. That is done through the basic properties any clip has:

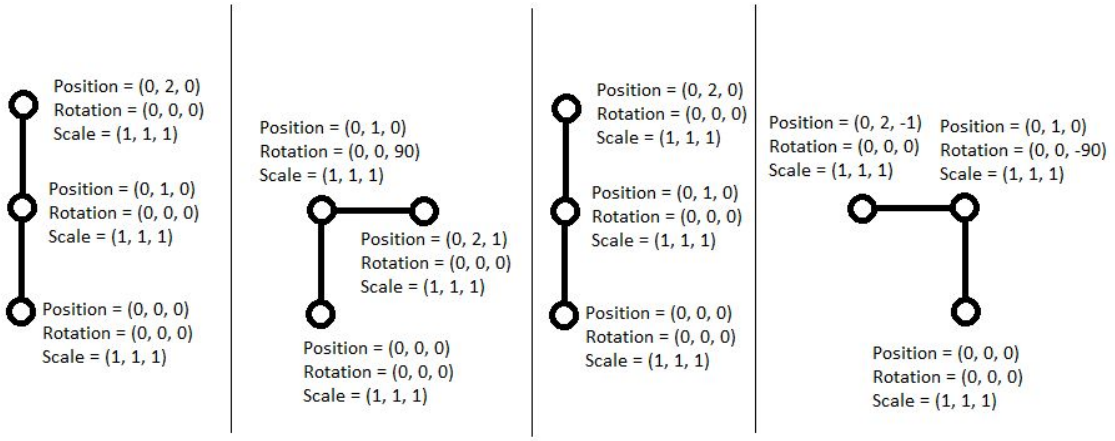
- Poses: the array of all the poses that clip has.
- Speed: the reproduction speed of the clip
- Poses count: the number of poses the clip contains
- Loop: handles if the animation should loop or not.
- Root Motion: some animations have motion associated with the root bone. This motion sometimes needs to be used on the game and others not.

Only with this basic information, the user can already modify some of the values and personalize how the animation will behave. The behaviors that can be modified in the clip, affect the clip for all its uses. Increasing or decreasing the Speed parameter will make the clip reproduce faster or slower and the loop value allows us to play it multiple times or just once.

Sometimes the animations are not prepared for games or the art team prefers to create all the animations in a single file, which creates a clip with all the animations together.

For that reason, the animation clip implementation includes a tool to create sub-clips from one. This tool will split a clip into clips with the same properties but that only contains the selected poses from the original clip.

In order to test the functionalities, an animation clip with the frames in the following figure (F5.9) is set-up:



F5.9 Test Animation Clip - The keyframes for the test animation and the transformations for each joint

Using the keyframes and the skeleton related to the clip all the joints data is accessible and the clip can be divided into 2 clips with 2 keyframes each one.

5.2.5 Skinning

Once all the animation data referring to the skeleton is ready to be used, the other important part of the animation system needs to handle how this data related to the skeleton is going to be used in the models.

During the development of the project, the skinning technique has been mentioned before in the integration concerns chapter. As previously explained, the skinning technique attaches the skeleton to the model itself. That is done with the addition of more data to the vertices of the model.

Usually, to represent a model that is not skinned in memory, the system only needs to store the position of the vertex in the 3D space, the normal of that vertex and the texture coordinates. Briefly explained, the position represents where in the world the vertex is, the normal represents the direction the vertex faces and the texture coordinates are used to check the color the vertex has inside the texture resources.

When we need to handle skeletal animation, apart from the information stated before, there are two more properties that the vertices need to know:

- Joint indexes: These are the indexes a vertex is attached to. a vertex can be attached to more than one joint, that way, the final movement and placement of it will be more accurate. The number of joints a vertex can be attached to is not a set number, but the more joints, the more expensive the calculations will be. Usually, the numbers of joints used for a vertex is 4 and that is the amount the library will support.

- Joint weights: a weight is how much a value affects an operation. In the case of the joint weight, they represent how much the vertex is affected by each of the joints. The sum of all the weights must always be 1.

Testing the skinning requires a model associated with the skeleton. For that, the example engine was used as explained in 5.3.1.2 Loading de skinning data.

5.2.6 Animation Montage

After collecting all the data needed for the library to work, the animation library needs a member that allows users to customize how they want to use the animation clips. To fulfill this purpose, the animation montage provides them with the tools needed to customize how an animation will play, combine them and get information about them.

The animation montage always has the position the skeleton joints have at that specific point of time. Using these values the users are able to modify the final position of the model vertices to represent the animation.

The montage contains the state machine (explained at 5.2.7 Animation State Machine) that handles the different clips and transitions and allows the users to interact with them. It also handles the reproduction state of the animation: Playing, Paused, Stopped.

This data is present in the library using the following structure:

- CalculatedPose: represents the transformations of the joints in the skeleton for a specific point in time.
- StateMachine: contains the state machine that controls the animation behavior.
- State: an enumeration which defines the state of the animation system
- Skeleton: the skeleton this montage is referred to.

The different functionalities of the animation montage allow changing the state machine associated with the montage and update the state of the animation itself.

The montage is responsible for preparing the final joints transformations to be then collected by the user and send them to de shader. This process consists of multiplying the skeleton inverse pose of the joint by the transformation matrix in the current pose. That converts the transformation matrix to one that transforms the vertex position to the new one. That is because when the skeleton was created, the transformation of the vertex was already defined and the skeleton takes into account that transformation.

Figure F5.10 shows the code used in the library to fulfill the requirements stated before.

```
namespace TroMo
{
    class AnimationMontage
    {
    public:
        enum AnimationState
        {
            Stopped,
            Playing,
            Paused,
        };
    public:
        AnimationMontage(Skeleton* skeleton){ ... }

        void UpdatePose(float deltaTime){ ... }

        void SetStateMachine(AnimationStateMachine* stateMachine){ ... }

    private:


    public:
        TroMoMat4x4* calculatedPose;
        AnimationStateMachine* stateMachine;


    private:
        Skeleton* skeleton = nullptr;

        AnimationState state = Stopped;
    };
}
```

F5.10 Animation Montage Code Structure - The image shows the code used in the library to fulfill the requirements of any montage.

In order to test the animation montage, the other member that creates the montage is needed. Assuming the state machine works correctly, the only thing remaining to test in the montage class is the reproduction state and that the state machine joints transformations are correctly converted by the skeleton inverse pose.

To set a test environment, a state machine with the clip used previously in the Animation clip test is used. Then the possibility to trigger the play, stop, and pause functions are added when pressing the keyboard keys 1, 2 and 3. With that set-up, when pressing the different keys the montage correctly plays, stops and pauses the clip reproduction and the transformation for the joints is correct.

5.2.7 Animation State Machine

When working with animations, one of the goals is being able to associate them with the different states of the character they are bound to. This is used to reduce the number of animations needed and create animations for specific situations. A common example of that is a character that is stopped, starts to run and jumps. Without the possibility to associate different animations, the sequence should be created together

but this tool allows us to create different animations such as Idle, Run, Jump, Fall and then play and mix them according to the situation.

The member of the animation library in charge of that feature is the Animation State Machine. This member contains all the functionality a user needs to correlate different animations and when the transition between them should happen.

The state machine manages different elements of the library and makes the work together creating the final result:

- Animation state: (See 5.2.8) A state handles the reproduction of an animation clip and allows it to modify some parameters of it.
- State machine variables: (See 5.2.9) define user-defined parameters that allow controlling how the animation behaves
- Transitions: (See 5.2.10) Defines when and how animations change from one to another.

For each of the previous elements, the animation state machine defines different functionalities such as add new elements, modify them or remove them.

```
class AnimationStateMachine
{
public:
    AnimationStateMachine(Skeleton* skeleton) : skeleton(skeleton)
    {
        currentPose = new TroMoMat4x4[skeleton->numJoints];
    }

    void UpdateState(float dt){ ... }

    int AddState(AnimationClip* clip){ ... }

    void RemoveState(int id){ ... }

    void SetInitialState(int stateId){ ... }

    void AddVariable(const char* name){ ... }

    void AddVariable(const char* name, float value){ ... }

    void RemoveVariable(const char* name){ ... }

    void SetVariableValue(const char* name, float value){ ... }

    AnimationStateMachineVariable GetVariable(const char* name){ ... }

    void Reset(){ ... }

private:

public:
    std::vector<AnimationState*> states;
    std::vector<AnimationStateMachineVariable> variables;
    std::vector<AnimationTransition*> transitions;

    AnimationState* initialState = nullptr;
    AnimationState* currentState = nullptr;

    TroMoMat4x4* currentPose = nullptr;

private:
    int stateId = 0;
    Skeleton* skeleton = nullptr;
};
```

F5.11 Animation State Machine Code - Structure of the code used to implement the animation state machine.

The main goal of the animation state machine is to compute the final pose corresponding to the time the animation is on. That is done through the update pose function. That function takes the time since the last frame and adds that to the Animation State corresponding to the current animation. Once the current animation is updated, the transitions for the current state are checked to know if any of them are fulfilled. If a transition is fulfilled, the animation state machine changes its state to the destination clip. Another case that the animation state machine also checks is related to blending clips (see 5.2.10.2 Animation Blender) the state machine asks the blender to create the blended pose in the cases where it is needed. From those calculations, the current position for the animation is created and ready to use.

To ensure that functionality works, two clips are needed. For testing purposes, the clip used to test animation clips is divided into two, then two different states are created and added to the state machine with a transition from one to another. This process allows testing not only the sate machine but all the other elements involved with the state machine.

Using the set-up described previously, the following cases are tested and checked:

- Clip reproduction: updating a single clip reproduces correctly the animation.
- Transition to another clip: the state machine transitions to the other clip in any of the cases a transition can have.
- Blending between two clips: the clips correctly transitions from one to another while blending during the transition.

5.2.8 Animation State

As stated in the state machine section (5.2.7) the animation state is one of the key components of the animation library since it allows us to modify and reproduce a clip.

The animation clip allowed users to personalize the reproduction speed of a clip and loop, but those settings will be applied for all the users of the clip. Sometimes, these options need to be used for a specific moment or just for a specific group or actions. The animation montage provides additional configuration for that purpose:

- Reproduction speed: a number that defines how fast the montage is played. A value of 0 means stopped time, 1 normal speed, any variation modifies the speed. The speed can also be negative, which will make the clip play in reverse.
- Loop: the loop value overrides the one from the clip, that way a clip not supposed to loop will loop or another way. The loop is applied to the whole montage. a montage with two clips will loop at the end of the second one.
- RootMotion: this parameter defines if the clip should use the translation values of the first joint in the skeleton.

That functionality allows modifying how the clip is played for each specific moment.

To reproduce a clip, the animation state handles the reproduction time for the specific clip. Whenever the montage asks the clip to update, the time value is increased or decreased according to the delta time and the speed value of the clip.

Then the animation clip calculates the skeleton transformation for that specific moment in time.

5.2.8.1 Skeleton transformation calculation.

One of the most critical points in the animation library is reproducing correctly the clips. As stated in 5.2.4 Animation Clips, a clip contains the different poses for specific points in time, the keyframes. These keyframes are used to reproduce the animation.

Each of the keyframes contains the data for all the joints at a specific time. When reproducing the animation, the time doesn't need to match exactly any of these keyframes times. Therefore, the position, rotation, and scale of the joint needs to be calculated from the keyframes for the time requested.

To do so, a mathematic technique called interpolation is used, creating an approximated point between the two initial ones.

In animation, there are two types of interpolations that are usually used:

- Linear interpolation (Lerp): the obtained value can be represented in a line between the two values.
- Spherical Interpolation (Slerp): the obtained value follows an arc between the two values. Mostly used for rotations.

To calculate an interpolation, the equation needs the initial value, the end value, and the progress through the end value represented in the interval 0 to 1, where 0 means initial value and 1 the end value.

That data is available at any point in time using the keyframes data and the time. Using these values the library obtains the value for any point of time.

Usually, the math library the game engine uses have functionality for those interpolations. To reuse that functionality, TroMotion defines in the math abstraction a function the user can modify to perform both Lerp and Slerp using the math library. In the library configuration, there's a parameter that allows choosing the interpolation method used in rotations.

When the values for the specific joint are calculated, the transform matrix is created for the specific joint. This is also usually present in the math library so a function in the math abstraction allows the user to use the math library one.

Finally, the transformation matrix is multiplied by the parent transformation and stored to be used from the animation state machine and animation montage.

5.2.9 Animation State Machine Variables

Defining what an animation state machine does to handle the clips reproduction requires two additional elements. One that defines how to change the behavior and one to allow users to interact with these changes. The goal of the animation state machine variables is the last one.

An animation state machine variable defines a property in the state machine that the user can assign a value to. That value can be changed at any point during the reproduction of animations which allow to control the behavior of the state machine.

Variables in computer programing can represent different types of data. From characters to decimal numbers and true or false expressions. The most useful one is the decimal number since animations usually are tied to the character movement velocity. For that reason, the variable uses a floating-point data type. That functionality can be expanded in future work or by any user of the library.

To allow users to modify and identify the variables, they have a name the user can set to them. Using that name, the other elements of the animation state machine can check and modify the value of the variable.

The variable has the following functionalities:

- Set the variable name: allows to change the name after setting it for the first time.
- Modify the value: allows changing the value at any point in the reproduction.
- Check variable value against another value: allows comparing the variable to another one to check if it is bigger, equal, or smaller. This is useful for triggering transitions (see 5.2.10) at specific moments.

In the following figure (F5.12) the code used in the library to represent the variables is shown.

```
namespace TroMo
{
    class AnimationStateMachineVariable
    {
    public:
        AnimationStateMachineVariable(const char* name) : name(name){}
        AnimationStateMachineVariable(const char* name, float value) : name(name), value(value) {}

        void SetValue(float value){ ... }

        void SetName(const char* name){ ... }

        const char* GetName()const{ ... }

        bool IsBigger(float value){ ... }

        bool IsSmaller(float value){ ... }

        bool IsEqual(float value){ ... }

    private:
        float value = 0.f;
        const char* name = nullptr;
    };
}
```

F5.12 Animation State Machine Variable Code - The image shows the code used to represent a variable in the animation library.

5.2.10 Animation Transitions

One of the topics that have appeared during the last points in the thesis is the possibility of mixing animations to create a sequence. That sequence should be able to adapt depending on the state of the application. To fulfill that, the animation transitions are added to the library.

A transition changes the current animation to another one. This allows creating sequences combining different animations. These sequences don't need to be always the same and can change depending on what the user decides.

Transitions allow defining how the change from one clip to another happens. Sometimes the user prefers for the transition to happen at the end of the clip that is currently being reproduced, when a variable value matches a condition or blending the two clips involved.

To be able to achieve that, the animation transition contains a reference to the clip where the transition starts, a reference to the clip it changes to and the conditions for the change to happen.

The other important element in the transitions is the blending between the two clips affected by the transition. A parameter that the user can modify allows enabling the blending functionality and setting the time for the blending to start. The time is used to know at which point before the end of the clip the blending needs to start. Another time parameter allows setting how much of the next clip needs to be overlapped with the previous one.

5.2.10.1 Animation Transition Conditions

The element in charge of controlling the changes between the two clips in the transition is the transition condition. This element checks if the transition should change to the next clip or not. A transition can have more than one condition.

The condition has some parameters to define what condition is defined on it. These parameters allow setting the condition to a variable or to the end of the clip:

- Transition at the end: this configuration makes the clip change to the next one when the clip finishes its reproduction.
- Variable condition: this configuration allows us to check if one of the variables in the state machine matches the defined condition against a user inputted value.

With the condition set, when the transition is being checked, the conditions tell the transition if they are fulfilled or not.

```
class AnimationCondition
{
public:
    enum Type
    {
        CheckVaraible,
        ClipEnd
    };
    enum VariableCheckType
    {
        Bigger,
        Smaller,
        Equal,
    };
public:
    AnimationCondition() {}

    void SetType(Type type) { ... }
    Type GetType()const { ... }

    bool IsFulfilled(AnimationStateMachine* state) { ... }

private:

public:
    VariableCheckType checkType = Equal;
    const char* variableName = nullptr;
    float variableValue = 0.0f;

private:
    Type type = CheckVaraible;
};
```

F5.13 Animation Condition Code - Code structure used to implement the animation conditions into the library.

Using the set-up created to test the animation state machine the following cases are tested:

- Clip End: the transition happens when a clip ends
- The variable is smaller: the transition happens if the variable value is smaller than the state machine variable with the same name.
- The variable is equal: the transition takes place if the variable value is the same as the state machine variable it is related to.
- The variable is bigger: when the variable value is bigger than the state machine variable value the transition is triggered.

5.2.10.2 Animation Blender

One of the modes the transition has, allows the library to blend between the two clips involved. Blending is usually used in multiple places in the animation systems allowing to create complex animation sets depending on multiple variables. That functionality is out of the scope of the thesis. In order to show how blending works, the most simple approach is implemented in the blender.

The blender allows creating smoother transitions between the two clips involved. The blending limits the capability of transitions forcing them to change at the end of the clip.

To properly blend a clip, the transition has two different time parameters that indicate to the blender when to start the blend and the progress of it. This progress is used to create a weight for each of the clips and calculate a transformation matrix out of them.

The blender is a structure that allows calculating the matrix. It takes the two skeleton poses and the weight of it and then returns a pose blended. The matrices for each of the joints are calculated using a weighted sum.

5.2.11 Animation Manager

All the systems that have been explained until now, work together into the library to create and reproduce animations. To handle the animations and the date they use, the animation manager is used. the manager is accessible from any point in the system since it uses a Singleton pattern that allows that. This is not a required element and the functionality it does can be handled by the library user's resource manager.

The animation manager has all the functionality to create and store the different data parts of the animation library:

- Skeletons
- Animation Clips

The manager adds them to a list of elements and ensures that at the end of their use are correctly deleted and don't leave any memory in use.

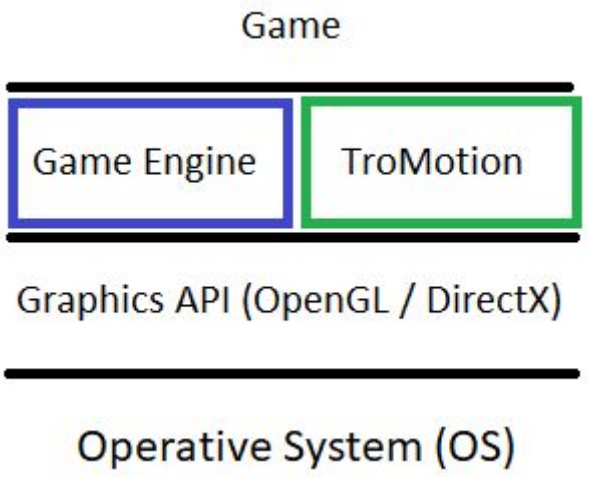
```
namespace TroMo
{
    class AnimationManager
    {
    public:
        //Get manager instance
        static AnimationManager& getInstance()
        {
            static AnimationManager instance;

            return instance;
        }

        // Creates an Skeleton and assigns an id for it
        Skeleton* CreateSkeleton() { ... }
        //Creates a new Animation Clip
        AnimationClip* CreateAnimationClip() { ... }
        //Creates a new Pose for a skeleton
        SkeletonPose* CreatePose() { ... }

    private:
        //prevent instancing
        AnimationManager() {}
        AnimationManager(const AnimationManager&) = delete;
        AnimationManager& operator=(const AnimationManager&) = delete;


    private:

        uint skeletonId = 0;
        std::list<Skeleton*> skeletons;
        std::list<SkeletonPose*> poses;
        std::list<AnimationClip*> animationClips;

    };
}
```

F5.14 Animation Manager - Code used to handle the animation resources created

5.3 Implementing the library into the example engine

Previously in the previous parts of this thesis, the example game engine and the library have been explained themselves. Now that both the engine and the library have been described, let's take a look at how they interact together.

To understand better the relationship between them, in figure F5.15, a diagram of the structure of the application is defined. The figure shows how the different parts build one on top of the other, starting with the Operative System as the base. On top of it, a graphics API is responsible for rendering images to the screen. Build on top of the graphics API, the game engine working with the library together to create the top part, the game.



F5.15 Layer relation diagram - The figure shows how the different parts build-up to create the final game

5.3.1 Loading the data for the library

On point 5.2 Animation Library, different data types were defined to hold the data for the animation library. These data need to be loaded by the engine developer.

More specifically, the data types that need to be loaded in order to use the library are the following:

- Skeletons
- Vertex skinning data
- Animation clips

Loading these data is handled using the third party library Assimp(10) but, even though Assimp loads the data from the file, he uses his own data structures to store the data.

To be able to use the data, the engine needs to convert it to the data types used by the animation library. Then store them in a resource manager to be able to access them. The example engine doesn't have a resources manager due to its simplicity. The clips and skeletons will be directly stored in the model data for ease of access.

5.3.1.1 Loading the skeleton data

Assimp library treats each of the elements on a file like nodes. Nodes have children nodes that define their hierarchy with respect to the other nodes. Nodes can contain mesh (3D models) information or just their name and position. To be able to identify the joints ones from the mesh ones, Assimp has build-in functionality to differentiate them.

The joints' names are also not loaded directly into the node. Assimp loads them as a child node for the one with the joint transformation. In order to load it correctly, looking

at the children of the node to find one with an identity transformation and store its name for the joint.

5.3.1.2 Loading the vertex skinning data

The data related to the vertices skinning data is stored together with the mesh data in Assimp. Additional information is loaded in the vertex data for the meshes. The data from Assimp then needs to be loaded into the skinning data from the library.

Assimp loads the name of the joint. For the library skinning data, the index of the joint in the skeleton is needed. To convert the name to the index, the skeleton structure has a function that returns the index. Using that functionality, the data can be loaded from Assimp.

The data is then loaded to the graphics card with the other mesh data to use later in the rendering process.

5.3.1.3 Loading Animation clips

Animation clips are the most important element in the library in order to reproduce animations since they store the data for the animation. Again, to load them the Assimp library is used. Assimp loads the animation in separated nodes dedicated to animations. These nodes contain the information for the joints positions at a specific time point, the duration, and the frames per second of the animation.

Differently, Assimp loads the data by joint instead of by keyframe. Thatdifficults a bit the loading of the data since the data needs to be taken for each joint and sort it in keyframes.

This part has consumed a lot of time from the development of the thesis since not only sorting the data required a specific algorithm to create the keyframes at first and then fill them with the correct data when the joints where loaded but the number of keyframes didn't match the ones from the animation. After some investigation on the issue and attempts to sort the clip data, the issue was related to the Assimp library and not something related to the implementation of it. The library has an issue loading the transformation values for some animation keyframes. The issue always happens with the FBX file type and sometimes with the Collada file type. For the issue present in the third-party library, the current version of the application is not capable of showing the models available to test performing correctly.

In order to ensure the application issue is related to this library issue, the animation library has been tested separatedly as explained on each of the animation library elements. Then the following procedure has been followed to try to make the library work together with the engine:

1. Ensure the skeleton pose works correctly: The first step to identify the problem was related to the keyframes transformations, the skeleton has been loaded.

Then these skeleton positions have been used to pose the character. That worked correctly as can be seen in figure F5.16.

F5.16 Model using skeleton - The image shows the model using the skeleton transformations.

2. Ensure the library code works properly using the skeleton as a clip. That consisted of creating an animation clip using only the skeleton as the keyframes. This test worked properly and the result is the same as it was in the previous case (F5.16). That discards any issue in the library code that had been tested previously.
3. Load the keyframes from Assimp and the model file. The last point to check was using the transformations loaded from the third-party library used to load the model, skeleton, and animations. Once the animation clips are loaded, they are reproduced using the same code used in the previous test that worked correctly. Nevertheless, the result was not the same. In this case, the skeleton transformations didn't work properly and the model is completely displaced from the correct position (F5.17)

F5.17 Model animated incorrectly - The image shows the result of using the transformations loaded from the third party library.

5.3.2 Using the library features - Animation component

Once the data is loaded into the engine, the next step is being able to use it. As explained in 5.1.2, the engine uses a component system. That system allowed to add functionalities to the objects in the engine.

Using the component system of the engine, an animation component is created to interact with the library. This component will contain an animation montage that uses the same skeleton that the model present in the object as seen in figure F5.18.

```
#include "Component.h"
#include "AnimationMontage.h"

class Object;

class Animation : public Component
{
public:
    Animation(Object* parent);

    bool Update();
    bool CleanUp();

private:

public:
    TroMo::AnimationMontage* montage = nullptr;

private:

};
```

F5.18 - Animation Component - Code used for the animation component. The only element required is an Animation Montage.

As the point 5.2.6 Animation Montage explains, the animation montage contains all the functionality of the library linked. Then the user must ensure to update the state of the animation using the functionality the library provides and retrieve the data through the montage to display the character animated.

5.3.3 Displaying the character using the animation

At his point, all the elements needed to use the animation system are included in the engine and ready to be used. To use them the user needs to send the data the animation library calculates to a shader.

A shader is a software that is run directly into the graphics card. graphics cards have advantages over CPUs to calculate things faster when the operations needed are not complex. Since the example engine uses the graphics API OpenGL, the shader is written in GLSL, the language OpenGL defines for shaders.

The data the shader needs to use the calculated joints positions are the skinning data, which includes the joint indexes and the weight for the vertex, the calculated joints and the number of joints (F5.19).

One thing to take into account is that OpenGL doesn't allow to allocate memory dynamically into the graphics card. When sending the joints data to the shader, it should have reserved enough space to handle the bones your animations are using.

```
// == LIBRARY IMPLEMENTATION ==
// send joint data to shader
Animation* anim = (Animation*)GetComponentOfType(Component::C_Animation);
if (anim != nullptr)
{
    App->render->GetShader()->setInt("numJoints", mr->GetModel()->skeleton->numJoints);
    for(int i = 0; i < mr->GetModel()->skeleton->numJoints; ++i)
    {
        std::string uniform_name = "Joints[" + std::to_string(i) + "]";
        float4x4 mat = anim->montage->calculatedPose[i];
        App->render->GetShader()->setMat4(uniform_name, mat.Transposed());
    }
}
// =====
```

F5.19 Sending data to shader -The image shows the code used to send the joints data to the shader

Once the data is sent to the shader, the shader code is able to use it to transform the vertex position to the one corresponding to the animation. That is done multiplying the joint matrices that affect the vertex by its weight and adding them. The resulting matrix is then used to transform the vertex position for the final image to be displayed correctly.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 5) in vec4 jointsIDs;
layout (location = 6) in vec4 jointsWeights;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

const int MAX_JOINTS = 100;
uniform mat4 Joints[MAX_JOINTS];
uniform int numJoints;

void main()
{
    mat4 JointTransform = Joints[int(jointsIDs[0])] * jointsWeights[0];
    for(int i = 1; i < 4; i++)
    {
        if(jointsIDs[i] < numJoints)
        {
            JointTransform += Joints[int(jointsIDs[i])] * jointsWeights[i];
        }
    }

    TexCoords = aTexCoords;

    gl_Position = projection * view * model * JointTransform * vec4(aPos, 1.0);
}
```

F5.20 Shader code - The image shows the code used inside the vertex shader to use the joints transformations.

5.3.4 Engine Interface to work with the library

Previously, all the points in the implementation of the library into the engine, have talked about what is needed to use the library. This part is not mandatory to have but it is very useful to allow non-programmers members of a team to work with the animation library.

To ease the use of the animation library, engines provide some kind of user interface to create the transitions and states that define the animation behavior. These user interface elements are completely dependent on how uses or create them.

The example engine, as explained in the 5.1 Example Engine part, uses the library ImGui to create the user interface. This library allows creating all kind of personalized UI elements that adapts to the user necessities.

Due to the issues found while implementing the library with the third-party library Assimp, this part of the development couldn't be completely finished.

The elements implemented in the example engine are two:

- Skeleton hierarchy: this user interface element shows the complete skeleton loaded and the relation between the joints in it. (F5.21)

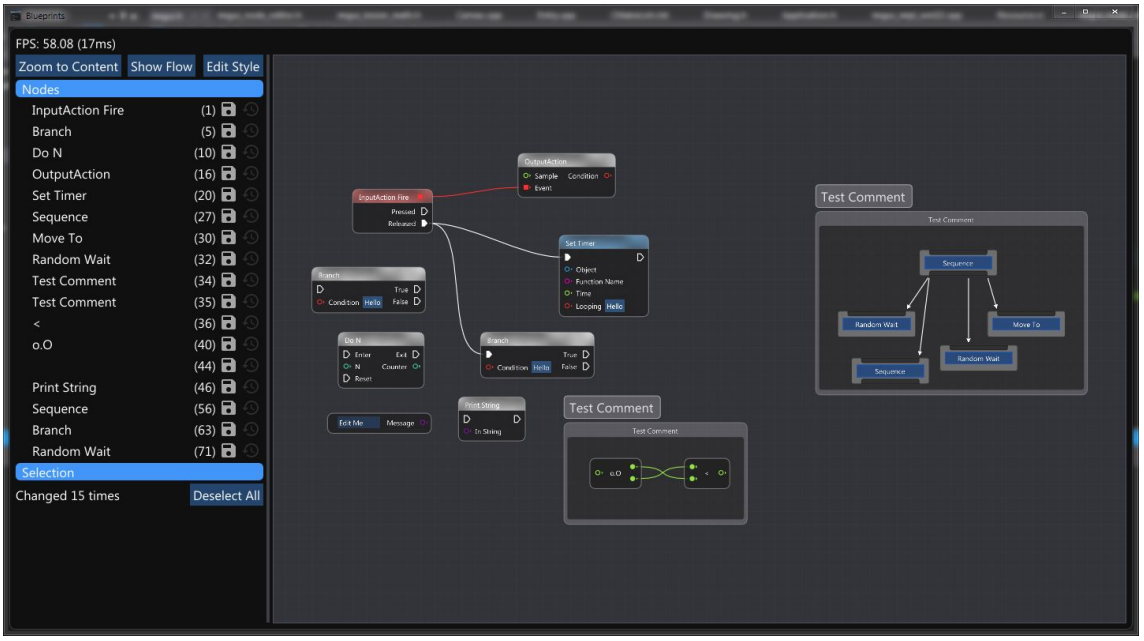
F5.21 Skeleton hierarchy - Shows the relation between the different joints in the skeleton

- Reproduction control panel: a simple panel with the functionality to play, pause, and stop the reproduction of the animation (F5.22).

F5.22 Reproduction control panel

The possibilities to create different UI tools with this library allow us to create complex editors that will allow displaying personalized interfaces for different components.

During the development of the library, a special useful tool that can help develop a node graph similar to the one used on professional engines such as unity (F2.7) was found. This tool is related to the ImGui library but developed as an extension that can be found in its own repository(14). Even though the tool has not been used in the final app due to the issues previously mentioned, the tool has been tried and is highly recommended to be used in future work or for other projects. The image F5.23 shows an example of the tool being used to replicate the Unreal Engine blueprints system.



F5.23 Node Graph tool - the image shows the tool being used to replicate the Unreal Engine blueprint system - https://github.com/thedmd/imgui-node-editor

6. Conclusions

An animation library consists of a lot of different elements that allow games and other graphics applications to reproduce animations and adapt them to the situation required. The skeleton, together with the 3D model allow artists to easily create animations and save them into animation clips. These clips contain the animation keyframes that are used by other applications to reproduce the animations.

The current situation of animation systems in the market relies on companies being able to create their own solutions or paying for a professional solution that is out of reach for small companies or students. For that reason, TroMotion can bring a solution to those users that look for an open-source solution that wasn't available until now in the market, since the few existing animation libraries don't focus on ease of implementation nor reutilize the developer software solutions for other systems.

During the development of the thesis, lots of inconveniences happened that made the development of the library slower than expected. Some of them being related to issues found during the development and some others to external work such as finishing university projects, personal necessities, and an internship.

Despite the development issues and the full schedule that made the production of TroMotion not as fluid as intended, most of the general objectives in the thesis have been fulfilled. Those objectives included the library being independent of the source code of the user as the main key point which is completely accomplished through the mathematic library abstraction developed.

The system also aimed to allow users to use different animations and mix them as required. Thanks to the animation state machine working together with the sates and transition are possible to create personalized animation sequences. This objected also included the user-defined event. The events required additional abstractions to allow the user to personalize them enough to be useful. After reviewing the available time and the remaining tasks for the library, this feature was the first to be taken out of the backlog due to the time investment required for it against the final utility.

Being able to be used in real-time environments was also a concern in the development of the library, the library works correctly in real-time thanks to the combination between precalculating the transformation matrices before sending them to the shader and the shader itself that optimize the time needed to calculate the final position for the vertices.

Regarding the possibility to be able to replicate the state, this feature was not implemented at the end due to the lack of time and other features were prioritized to allow the user being able to reproduce animations and personalize them. This feature should be one of the top priorities to improve the animation library potential.

One of the key points in the development was creating an abstraction for the different systems the library relied on the existing software. That opens the to any application to easily integrate it and use its features. On the other hand, some of the third-party tools used to create the example engine, particularly the Assimp library, created more problems that required extra time to try to solve and reduced the amount of time

available to the development of the library. Other model loading tools are available that could maybe be more useful than the Assimp library. When the issues with the Assimp library appeared, the time needed to redo the example engine with another library exceeded the remaining time available for the implementation of the library.

As future work, changing the current example engine to allow to correctly prove the library potential is essential. Once that is achieved, the first thing to add to the library is the serialization part. That will help users to save and load faster the animations they create and configure. Also, allow creating personalized events to improve the possibilities when working with it.

Existing animation systems have support for lots of different functions such as layers that allow using different animations for different parts of the body. Others also allow creating blending between multiple animations depending on more than one variable. And most of them also support inverse kinematics. Those features can also be added into the library in the future to extend the base system that is now present.

Even though the library is not as complete as other professional solutions, I believe it is a good starting point for small teams and students to understand and use the animation system and have a solid base to improve it and develop its own features to expand it. Being an open-source library it can be the first step to develop an open-source animation library with other collaborations that bring a real solution for the animation system.

7. Bibliography

1. Williams, Richard. 2001. *The animator's survival kit.* ISBN 9780865478978
2. GNU General Public License. www.gnu.org/licenses/gpl-3.0.en.html. Checked 12 March 2019
3. History of animation, https://www.nyfa.edu/student-resources/quick-history-animation/, checked 13 March 2019
4. Animation, https://www.britannica.com/art/animation, checked 13 March 2019
5. Animadead, http://animadead.sourceforge.net/features.shtml, checked 10 March 2019
6. Ozz-Animation, http://guillaumeblanc.github.io/ozz-animation/, checked 11 March 2019
7. Gregory, Jason. 2013. *Game Engine Architecture*. CRC Press. ISBN 978-1-4665-6001-7
8. Feature-driven development, http://newline.tech/blog/feature-driven-development-methodology/, checked 14 March 2019
9. Trello, https://trello.com/, checked 15 March 2019
10. stb library, https://github.com/nothings/stb, checked 30 March 2019
11. Assimp, http://www.assimp.org/, checked 6 April 2019
12. Abstraction in C++, https://www.geeksforgeeks.org/abstraction-in-c/ checked 17 April 2019
13. Granny 3D. RadGameTools, http://www.radgametools.com/granny.html, checked 05 June 2019
14. ImGui Node Library, https://github.com/thedmd/imgui-node-editor, checked 18 August 2019