

UNIVERSIDAD DE COSTA RICA
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica

IE0499 – Proyecto Eléctrico

**Diseño e implementación de un algoritmo para la
corrección de *glitches* de *MoCap* basado en goniometría**

por

Mario Alberto Castresana Avendaño

Ciudad Universitaria Rodrigo Facio

Julio de 2017

Diseño e implementación de un algoritmo para la corrección de *glitches* de MoCap basado en goniometría

por

Mario Alberto Castresana Avendaño

A41267

IE0499 – Proyecto Eléctrico

Aprobado por

Dr. rer. nat Francisco Siles Canales

Profesor guía

Ing. José Fernando Salas Fumero

Profesor lector

Ricardo Román Brenes, M. Sc.

Profesor lector

Julio de 2017

Resumen

Diseño e implementación de un algoritmo para la corrección de *glitches* de MoCap basado en goniometría

por

Mario Alberto Castresana Avendaño

Universidad de Costa Rica

Escuela de Ingeniería Eléctrica

Profesor guía: Dr. rer. nat Francisco Siles Canales

Julio de 2017

En la actualidad, la tecnología de captura de movimiento o *Motion Capture*, se utiliza en producciones de todo nivel en Animación Digital. El concepto en el que se basa dicha tecnología, consiste en la elaboración de modelos 3D a partir de datos numéricos interpretados por un conjunto de cámaras infrarrojas y sensores. El problema que presenta esta tecnología es, que el procesamiento de los datos arrojados por estos sensores no siempre es correcto, lo cual hace que existan pérdidas de datos de posición que resultan en modelos 3D defectuosos.

Este proyecto, consiste en el diseño e implementación de un programa de *software* capaz de corregir dichos errores de procesamiento por parte de las cámaras, y con esto, generar modelos de animación 3D corregidos mediante el uso de algoritmos de interpolación en 3D.

Los recursos teóricos para proponer una solución a este problema, se basan en estudios de anatomía y goniometría, los cuales son comúnmente utilizados en Terapia Física y Ortopedia. Para la implementación del programa como tal, se decidió utilizar Python, ya que presenta varias ventajas de integración con otros programas de Animación Digital, y todas las herramientas necesarias para montar un *framework* de trabajo en el laboratorio de investigación en reconocimiento de patrones y sistemas inteligentes (*PRIS-Lab*).

El resultado que se espera obtener, es un programa capaz de recibir un archivo de captura de movimiento como argumento de entrada, y producir a la salida, un archivo corregido tipo *Biovision Hierarchical Data*.

Palabras claves: Animación Digital, Motion Capture, Goniometría, Computer Graphics.

Acerca de IE0499 – Proyecto Eléctrico

El Proyecto Eléctrico es un curso semestral bajo la modalidad de trabajo individual supervisado, con el propósito de aplicar estrategias de diseño y análisis a un problema de temática abierta de la ingeniería eléctrica. Es un requisito de graduación para el grado de bachiller en Ingeniería Eléctrica de la Universidad de Costa Rica.

Abstract

Diseño e implementación de un algoritmo para la corrección de *glitches* de MoCap basado en goniometría

Original in Spanish. Translated as: “Design And Implementation Of A Motion Capture Glitches Correction Algorithm Based On Goniometry”

by

Mario Alberto Castresana Avendaño

University of Costa Rica

Department of Electrical Engineering

Tutor: Dr. rer. nat Francisco Siles Canales

July of 2017

Nowdays, motion capture technology, is used in productions of all levels in Digital Animation. The concept on which this technology is based on, consists of the elaboration of 3D models from numerical data interpreted by a set of infrared cameras and sensors. The problem with this technology is that the processing of the data generated by these sensors is not always accurate, causing loss of position data which results in corrupted 3D models.

This project consists of the design and implementation of a software program capable of correcting such camera processing errors, in order to generate accurate 3D models, by using goniometry and articular phisiology to understand human movements and correct those that are incorrect.

The theoretical resources to propose a solution to this problem, are based on anatomy and goniometry studies commonly used in physiotherapy and orthopedics. For the software implementation, it was decided to use Python, as it presents several integration advantages with Animation software and compatibility with other research tools, currently being developed in the Pattern Recognition and Intelligent Systems Laboratory (PRIS-Lab).

The expected result is a program capable of receiving a motion capture file as the input argument, and outputting a corrected BioVision Hierarchical Data file.

Keywords: Digital Animation, Motion Capture, Goniometry, Computer Graphics.

About IE0499 – Proyecto Eléctrico (“Electrical Project”)

The “Electrical Project” is a course of supervised individual work of one semester, with the purpose of applying design and analysis strategies to a problem in an open topic in electrical engineering. It is a requisite of graduation for the Bachelor of Science in Electrical Engineering, granted by the University of Costa Rica.

Dedicado a mi familia y amigos.

Agradecimientos

Quiero empezar por agradecerle a los profesores de esta Escuela que más me han apoyado a lo largo de mi carrera universitaria, ya que todos le agregaron un valor incalculable a mi experiencia de estudio. A mi tutor Francisco Siles Canales, le quiero agradecer por todo su apoyo brindado a lo largo de este proyecto y mi carrera. A Elvira Chaves, cuyos aportes en Fisioterapia y goniometría, fueron vitales para lograr concretar este proyecto. Un agradecimiento especial a Josué Vargas y Andrés Mora, por ayudarme a desarrollar el marco teórico del presente trabajo. Finalmente, quiero agradecer a Ricardo Román y a Fernando Salas por sus excelentes aportes y el apoyo brindado. Me siento honrado de haber sido apoyado por gente talentosa.

Índice general

Índice general	xii
Índice de figuras	xiv
Índice de tablas	xv
Nomenclatura	xvii
1 Introducción	1
1.1. Alcances del proyecto	2
1.1.1. Problema a resolver	2
1.1.2. Herramientas a utilizar	3
1.2. Justificación	4
1.3. Objetivos	4
1.3.1. Objetivo general	4
1.3.2. Objetivos específicos	4
1.3.3. Actividades importantes del proyecto	5
1.4. Metodología	5
1.4.1. Planificación	6
2 Marco Teórico	9
2.1. Aspectos a utilizar de animación 3D	9
2.1.1. Modelado jerárquico	9
2.1.2. <i>Motion Capture</i>	11
2.2. Movimiento humano	12
2.2.1. Goniometría	13
2.2.2. Nomenclatura de movimientos basada en goniometría	15
2.2.3. Taxonomía de <i>glitches</i>	16
2.3. Reconocimiento de patrones	19
2.3.1. Sensores	20
2.3.2. Vector de características	21
2.3.3. Umbralización	21

2.3.4. Detección	21
2.3.5. Segmentación	21
2.3.6. Corrección	22
3 Diseño e implementación del algoritmo	23
3.1. Introducción	23
3.2. Levantamiento de requisitos	24
3.3. Arquitectura del programa y tareas principales	24
3.3.1. Acerca de los archivos <i>bvh</i>	26
3.3.2. Leer el <i>bvh</i> y crear una jerarquía de <i>bones</i>	27
3.3.3. Asignar un <i>bone</i> a cada entrada del vector de movimiento	28
3.3.4. Aplicación de los estudios de goniometría	29
3.3.5. Corrección del archivo de <i>MoCap</i>	30
3.4. Interfaz de entrada/salida	34
3.5. Implementación de clases y subclases	34
3.5.1. Clase <i>Bone</i>	35
3.5.2. Implementación de subclases	37
3.6. Recursos en línea	42
4 Validación	43
4.1. Introducción	43
4.2. Prueba sintética 1	44
4.2.1. Detección y localización	44
4.2.2. Corrección	45
4.3. Prueba sintética 2	47
4.3.1. Detección y localización	47
4.3.2. Corrección	49
4.4. <i>Precision, Recall</i> y <i>F1-Score</i>	50
4.5. Pruebas de laboratorio	52
4.5.1. escena 2.2 esqueleto 9	52
5 Conclusiones y recomendaciones	55
5.1. Conclusiones	55
5.2. Recomendaciones	57
A Programa principal y su diagrama de flujo	59
A.1. Código fuente del programa <i>BVHTuneUP</i>	59
A.2. Diagrama de flujo del programa principal	63
A.3. Diagrama de flujo para la función <i>Goniometry_check</i>	64
B Diseño del programa	65
B.1. Arquitectura del programa	65

B.2. Diagrama UML para las clases	66
C Código de las clases	67
C.1. Clase padre <i>Bone</i>	67
C.2. La subclase <i>Arm</i>	70
C.3. La subclase <i>ForeArm</i>	73
C.4. La subclase <i>Leg</i>	76
C.5. La subclase <i>UpLeg</i>	80
C.6. La subclase <i>Spine</i>	84
C.7. La subclase <i>Wrist</i>	87
Bibliografía	91

Índice de figuras

1.1.	<i>Glitch de captura de movimiento.</i>	3
1.2.	Diagrama de Gantt Marzo-Abril	7
1.3.	Diagrama de Gantt Abril-Mayo	7
1.4.	Diagrama de Gantt Junio-Julio	8
2.1.	Jerarquía de un <i>bvh</i> .	10
2.2.	Estructura de datos de <i>MoCap</i>	11
2.3.	Ejemplo de goniometría.	13
2.4.	Planimetría para el cuerpo humano.	14
2.5.	Movimientos asociados a los <i>glitches</i> anatómicos.	16
2.6.	Articulación escapulohumeral.	17
2.7.	Taxonomía de <i>glitches</i> .	18
2.8.	<i>Glitches</i> que el algoritmo puede arreglar.	19
2.9.	Proceso de RP para este proyecto.	20
3.1.	Arquitectura del programa.	25
3.2.	Proceso de umbralización.	31
3.3.	Diagrama de flujo del programa principal.	33
3.4.	Interfaces de I/O por tarea.	34
3.5.	Diagrama UML de clases.	35
4.1.	ECM para la prueba sintética 1.	46
4.2.	ECM para la prueba sintética 2.	49
4.3.	Pérdida de orientación de la rodilla en Y.	49
4.4.	<i>Glitch</i> de rotación interna del tobillo.	52
4.5.	<i>Glitch</i> de flexión de codo del <i>frame</i> 274-404.	53
4.6.	<i>Glitch</i> de flexión de codo en el <i>frame</i> 394.	53
4.7.	<i>Glitch</i> de flexión de codo por pérdida de ejes.	54
4.8.	<i>Glitch</i> de flexión lateral de columna.	54

Índice de tablas

3.1.	Jerarquía de <i>bones</i>	28
3.2.	Ejes y sus respectivos movimientos.	29
3.3.	Límites de movilidad del cuerpo humano.	32
4.1.	<i>Glitches de la prueba sintética 1</i>	44
4.2.	<i>Glitches para la prueba sintética 2</i>	47
4.3.	<i>Precision, Recall y F1-Score por prueba</i>	51

Nomenclatura

bone	Proveniente del inglés, significa hueso. Elemento perteneciente a un esqueleto de un archivo de captura de movimiento. También llamado <i>JOINT</i> en los archivos <i>bvh</i>
BVH	formato de los archivos de captura de movimiento. Proveniente del inglés <i>BioVision Hierarchy</i>
ECM	Error Cuadrático Medio
EIE	Escuela de Ingeniería Eléctrica de la Universidad de Costa Rica
frame	cuadro de un vídeo
GdL	Grados de libertad de una articulación del cuerpo humano
glitch	Error de <i>software</i> o <i>hardware</i> . También llamado <i>bug</i> . Palabra de origen desconocido, fue usada en el idioma inglés desde 1960 en el programa espacial de EEUU; se cree que proviene del inglés <i>malfunction, hitch</i>
IEEE	Instituto de Ingenieros Eléctricos y Electrónicos (del inglés <i>Institute of Electrical and Electronics Engineers</i>)
MoCap	Captura de movimiento (del inglés <i>Motion Capture</i>)
PRIS-lab	Laboratorio de reconocimiento de patrones y sistemas inteligentes (del inglés <i>Pattern Recognition And Intelligent Systems Laboratory</i>)
árbol	Tipo abstracto de datos, el cual simula una estructura jerárquica. Se define como una colección de nodos, en donde cada nodo es una estructura de datos que contiene valores y referencias a nodos hijos dentro de la estructura. Todo esto con la restricción de que ninguna referencia está duplicada y ninguna apunta al nodo raíz.

Capítulo 1

Introducción

Actualmente, en el laboratorio de investigación en reconocimiento de patrones y sistemas inteligentes (*PRIS-Lab*), se utilizan cámaras infrarrojas de captura de movimiento para el estudio del movimiento humano. Dichas cámaras son capaces de generar un modelo 3D básico del esqueleto humano, que se representa en archivos del tipo *Biovision Hierarchical Data* o **BVH**. Los archivos BVH contienen los datos necesarios para generar un modelo 3D que captura fielmente todos los movimientos hechos por un actor enfrente de las cámaras y se usan frecuentemente en Animación Digital y otras áreas afines.

El problema con estas cámaras es que, por lo general, las grabaciones presentan una cantidad considerable de errores de captura o *glitches*, en los cuales la posición de los sensores infrarrojos no es capturada de manera correcta y eso deriva en modelos en los que las articulaciones se deforman o hacen movimientos antinaturales [Parent, 2012]. El presente proyecto, consiste en el diseño e implementación de un algoritmo capaz de eliminar esos errores de captura de movimiento de un archivo BVH, y así generar una animación 3D más correcta y fluida.

En general, este proyecto se mantiene dentro del área de la computación y otras ramas de investigación afines al *PRIS-Lab*, particularmente los grupos conocido como MOVE y DAWN. Ambos grupos, hacen uso de un avanzado sistema de *Motion Capture* para diferentes propósitos de investigación y también han tenido que lidiar con los errores propios de este sistema.

En el caso de grupo MOVE, éste se dedica al análisis tridimensional del movimiento humano y el desarrollo de sistemas de modelado computacional para el análisis biomecánico-cinemático de movimientos humanos, en particular para optimizar el desempeño deportivo y el mejoramiento de la salud.

En lo que respecta a DAWN, éste grupo desarrolla proyectos de apoyo a la investigación y a las labores generales del laboratorio. Haciendo uso del enorme recurso tecnológico, exploran aplicaciones de la animación digital partiendo desde el diseño conceptual, el modelado y el *rigging*, hasta la renderización de alta calidad.

Es el propósito de este proyecto, desarrollar e implementar una aplicación de *software* capaz de ayudar a estos grupos a depurar sus archivos de captura de movimiento, con el fin de facilitar las labores de investigación. Como nota final, este proyecto de investigación, se suma a los múltiples desarrollos que se dan actualmente en el PRIS-lab, y constituye parte de un trabajo en progreso mucho más amplio y profundo.

1.1. Alcances del proyecto

Los archivos de tipo **BVH**, por lo general presentan varios tipos de errores de captura de movimiento. Como parte de los objetivos de este proyecto, se pretende definir una taxonomía de errores adecuada, con el fin de delimitar puntualmente qué tipo de eventos se consideran errores y cuáles de éstos se van a corregir.

De forma general, el presente proyecto desarrolla una solución de *software* capaz de llevar a cabo la detección, localización y corrección de errores de captura conocidos como *glitches*. El programa que se obtuvo como resultado, presenta mejoras sustanciales en los vídeos de *MoCap*, sin embargo, no se pretende crear un programa que sustituya en su totalidad las capacidades de un animador profesional.

El programa representa una primera aproximación en la corrección automatizada de errores, sin embargo, éste no pretende arreglar una captura en su totalidad, ya que no todos los *glitches* se pueden corregir mediante *software*. Será responsabilidad del animador, retomar el archivo y completar la corrección hasta ésta quede preparada para la siguiente fase de producción.

1.1.1. Problema a resolver

Cuando se capturan datos de movimiento por medio de cámaras infrarrojas, existen ocasiones en las que los sensores y las cámaras no son capaces de obtener la posición correcta de cada parte del cuerpo humano en el espacio, y debido a esto, se obtienen tomas con movimientos animados inconsistentes. Usualmente, se les llama a estos errores *glitches*, palabra que proviene del inglés *hitch* y que significa dificultad o problema temporal.

Normalmente, los sistemas de captura de movimiento ópticos son muy vulnerables al ruido y por eso son propensos a este tipo de errores [Parent, 2012]. La razón por la que estos errores ocurren, tiene que ver con la cantidad de luz en el estudio en el momento de la grabación o interferencia entre varios sensores que las cámaras pierden momentáneamente [OptiTrack, 2017].

En la figura 1.1, se muestra un vídeo de captura de movimiento visto en en el *software* de animación 3D, Blender. Note como el codo derecho del esqueleto que representa al actor, está torcido de forma antinatural.



Figura 1.1: *Glitch* de captura de movimiento.

Debido a esta limitante, muchos errores como estos se presentan con frecuencia en grabaciones de *MoCap*. Lo que se pretende en este proyecto, es arreglar los *glitches* mediante un algoritmo, de tal manera, que los movimientos de los esqueletos en 3D como el de la figura, se vean más naturales.

1.1.2. Herramientas a utilizar

La manipulación de datos puros para generar un modelo en 3D, requiere de un conjunto de bibliotecas estandarizadas de **OpenGL** para funcionar de manera eficiente. La disponibilidad de dichos recursos, depende del lenguaje de programación que se escoja para la implementación del programa. Dicho esto, se escogió desarrollar el programa en Python, debido fundamentalmente, a las herramientas que actualmente se desarrollan en el *PRIS-Lab* y a las ventajas de compatibilidad con otros programas de animación 3D.

Inicialmente, se planteó trabajar con bibliotecas de *OpenGL* en C/C++, sin embargo, una vez que se investigó a profundidad la forma en la que operan los archivos de *MoCap* tipo BVH, se decidió prescindir de las bibliotecas de *OpenGL* y manejar los datos de manera directa.

Siendo este el caso, las únicas herramientas de desarrollo que se escogieron para trabajar en este proyecto fueron:

- Python 3.5.2
- IDE Visual Studio Code 1.13.1
- el módulo CSV de Python para generar archivos de tipo Excel (muy útil para *Debugging* y validación de datos)
- *Blender* como software de animación 3D para comprobar los archivos BVH generados.

Dentro de los aspectos a omitir en este proyecto, se encuentran el diseño de una interfaz gráfica para el programa, ya que se supone que éste correrá en el *cluster* del *PRIS-Lab*.

1.2. Justificación

La tecnología de captura de movimiento o *Motion Capture*, es muy utilizada en áreas como la Animación Digital para diferentes tipos de producciones audiovisuales, que van desde los videojuegos y comerciales de televisión, hasta producciones cinematográficas. Normalmente, muchos animadores se basan en modelos 3D obtenidos con esta tecnología para producir todo tipo de animaciones [Parent, 2012].

El problema que este proyecto pretende resolver, es importante porque en el área de Animación Digital, corregir los archivos de *MoCap* representa una inversión significativa en cualquier producción. Poner a un animador profesional a corregir manualmente, cuadro por cuadro, un archivo **BVH** puede costar \$10 por segundo, por personaje animado.

Esto quiere decir, que una producción pequeña como un comercial o un corto animado, puede llegar a costar miles de dólares, solo por el hecho de solicitar a un animador depurar un archivo de *MoCap*. Eso sin mencionar, que es un proceso muy tedioso.

Revisando las herramientas comerciales disponibles y la literatura existente, no se pudo encontrar una herramienta de *software* que eliminara errores de captura de un archivo de *Motion Capture*. Obviamente, no se han podido hacer programas que sustituyan a los animadores. Lo que sí se puede hacer, es un programa que elimine los errores más comunes de una captura de movimiento y produzca un archivo **BVH** de mejor calidad.

En este sentido, el aporte principal del presente proyecto, se basa en crear dicho programa, y con eso, tratar de aligerar la carga que supone para un grupo de producción o investigación, el lidiar con errores de captura de movimiento.

Cabe mencionar, que una de las razones que motivaron este desarrollo, fue el ayudar a equipos de investigación del PRIS-lab que utilizan sistemas de *MoCap* regularmente. El beneficio que grupos como MOVE o DAWN obtienen de este proyecto, es un programa que facilite los procesos de producción e investigación, al tener un medio por el cual corregir *MoCaps* defectuosos.

1.3. Objetivos

1.3.1. Objetivo general

- Diseñar e implementar un algoritmo de corrección de *glitches* de *MoCap* basado en goniometría.

1.3.2. Objetivos específicos

- Realizar una investigación bibliográfica referente a los algoritmos más comunes para la generación de curvas de animación.
- Seleccionar los algoritmos e infraestructura de *software* para la resolución del problema.
- Implementar de dichos algoritmos en Python.

- Validar la implementación en pruebas de ejecución.
- Escribir y enviar a revisión un artículo para la conferencia estudiantil de la IEEE (CONESCAPAN).

1.3.3. Actividades importantes del proyecto

- Definir una taxonomía de *glitches* para delimitar qué tipo de errores el programa será capaz de corregir.
- Definir interfaz de entradas y salidas del programa.
- Definir la representación numérica más conveniente para los datos de movimiento.
- Documentar toda la información necesaria para la implementación del programa en Python.

1.4. Metodología

El diseño del programa, se trabajará con metodologías de desarrollo ágiles (*Agile*), en las cuales, se definirán un conjunto de tareas, con sus respectivos avances por semana. Los avances se darán a conocer semanalmente en las reuniones de PRIS-ProSeminar, donde se dará retroalimentación de cada tarea desarrollada, así como la revisión de objetivos y del trabajo escrito.

En lo que respecta al desarrollo de los objetivos, se planteó iniciar con una búsqueda bibliográfica relacionada con los siguientes campos:

- Animación Digital
- *Computer Graphics*
- Goniometría
- Fisiología articular

Los dos primeros campos se consultaron con el objetivo de conocer las herramientas de producción disponibles en el mercado actualmente; como parte de esta investigación, se buscaron referencias sobre herramientas, algoritmos o propuestas existentes para resolver el problema planteado en este proyecto. Los otros dos campos, más relacionados al área de Ciencias Médicas, se consultaron para conocer más acerca de la mecánica humana.

A la fecha de escritura de este trabajo, no se encontró ninguna herramienta de *software* que corrija errores de *Motion Capture*. En lo que respecta a algoritmos y métodos de estimación de posición, sí existe una variedad de métodos en la literatura consultada, sin embargo, en este proyecto se trabajará con estudios de goniometría para definir los límites de movimiento humano y corregir las capturas por medio de éstos.

La búsqueda bibliográfica hecha anteriormente, permite diseñar con mucho más criterio los algoritmos de corrección necesarios para estimar la trayectoria correcta de un movimiento corregido en 3D.

Como consecuencia del cumplimiento del primer objetivo, se espera conocer de forma general cuáles algoritmos conviene seleccionar para manipular archivos de captura de movimiento.

La forma en la que se planea seleccionar la infraestructura adecuada de *software*, es mediante la aplicación de los conceptos expuestos por Rick Parent en su libro *Computer Animation: Algorithms & techniques*, en el cual se exponen los algoritmos más comunes para la manipulación de información en *MoCaps* y en programas de animación digital.

Uno de los primeros pasos de desarrollo, será definir una taxonomía de *glitches* basada en fisiología articular y goniometría según Kathryn Luttgens [Luttgens et al., 2011]. La intención detrás de esto, es definir qué tipo de errores se pretenden corregir, y con esto, definir la mejor manera de resolver dichos errores. No todos los errores de captura se pueden resolver a partir de goniometría, pero ésta constituye una fuente de información muy precisa sobre los límites de movimiento humano. De ahí la importancia de la taxonomía.

En lo que respecta a la implementación de los algoritmos seleccionados, lo primero es hacer un levantamiento de requisitos y definir la arquitectura del programa en alto nivel. Luego se procederá a dividir el algoritmo en una serie de tareas. Cada tarea tendrá sus respectivas condiciones de frontera y diagramas de flujo. Finalmente, se procedió a definir las interfaces de Entrada/Salida del programa junto con sus estructuras de datos, para dar paso al diseño de clases y sus respectivos métodos.

Tal y como se muestra en uno de los objetivos específicos, será necesario decidir qué tipo de representación se utilizará para manipular los datos de movimiento. Por lo general, se utilizan ángulos de Euler y coordenadas cartesianas en *x*, *y*, *z* en los programas de animación 3D, así que se tomarán en cuenta los estándares actuales para manipular datos.

Finalmente, para cumplir el objetivo de validación, se pretende desarrollar una serie de prueba sintéticas y de laboratorio. En las primeras, se usarán archivos de *MoCap* con errores específicos en condiciones muy controladas, mientras que en las pruebas de laboratorio, se utilizarán capturas reales de producción. Mediante una comparación basada en el Error Cuadrático Medio (ECM) por *frame*, se pretende establecer qué tan preciso es el algoritmo en su proceso de corrección.

1.4.1. Planificación

El curso de Proyecto Eléctrico (IE-0499), se desarrolla a lo largo de un semestre ordinario en la EIE. Es por esto, que la planificación inicial del proyecto, contempla 16 semanas de tiempo de desarrollo. En estas semanas, se pretende desarrollar el proyecto basándose en el diagrama de Gantt mostrado en la página siguiente. Tome en cuenta, que los tiempos de desarrollo por cada objetivo, son estimados según en la experiencia del autor en proyectos previos.

1.4. Metodología

7

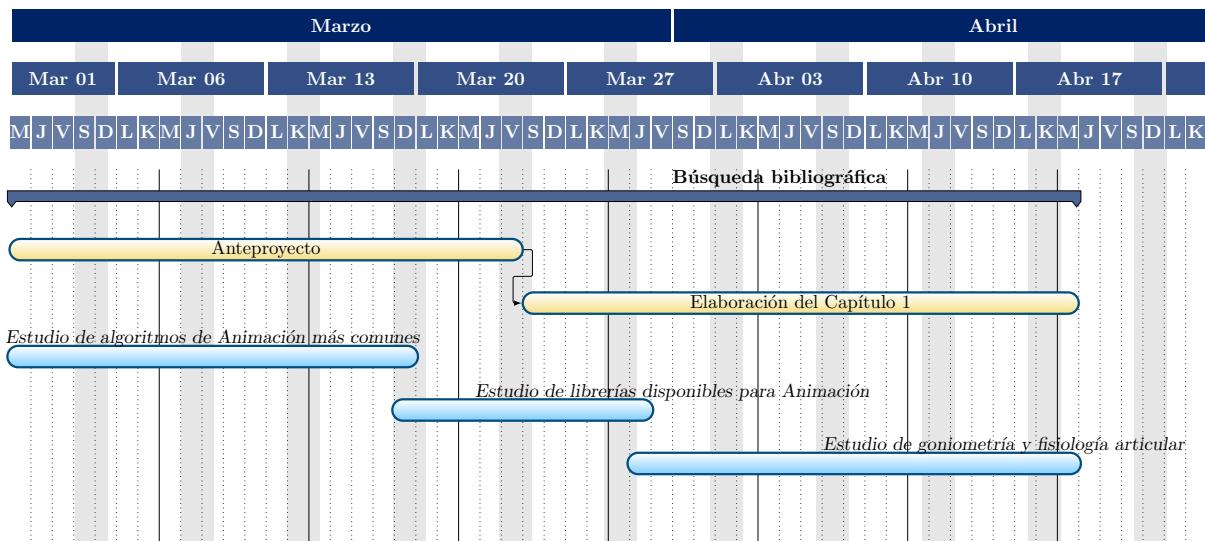


Figura 1.2: Diagrama de Gantt que muestra la planificación de Marzo-Abril para la investigación bibliográfica.

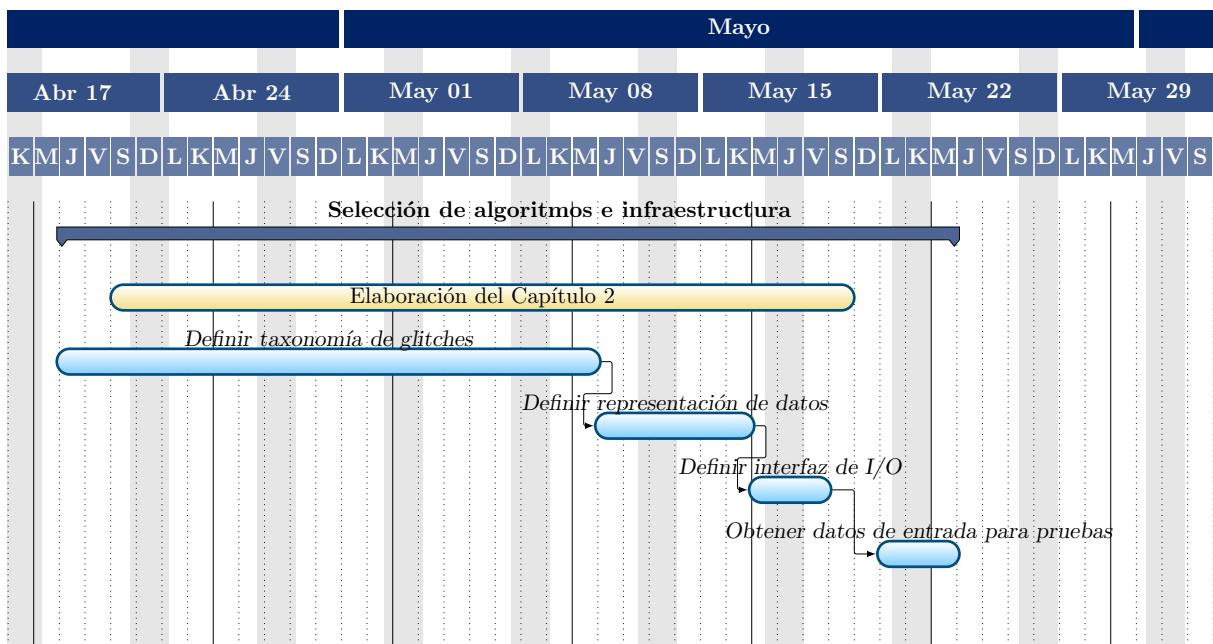


Figura 1.3: Diagrama de Gantt que muestra la planificación Abril-Mayo para la selección de algoritmos e infraestructura.

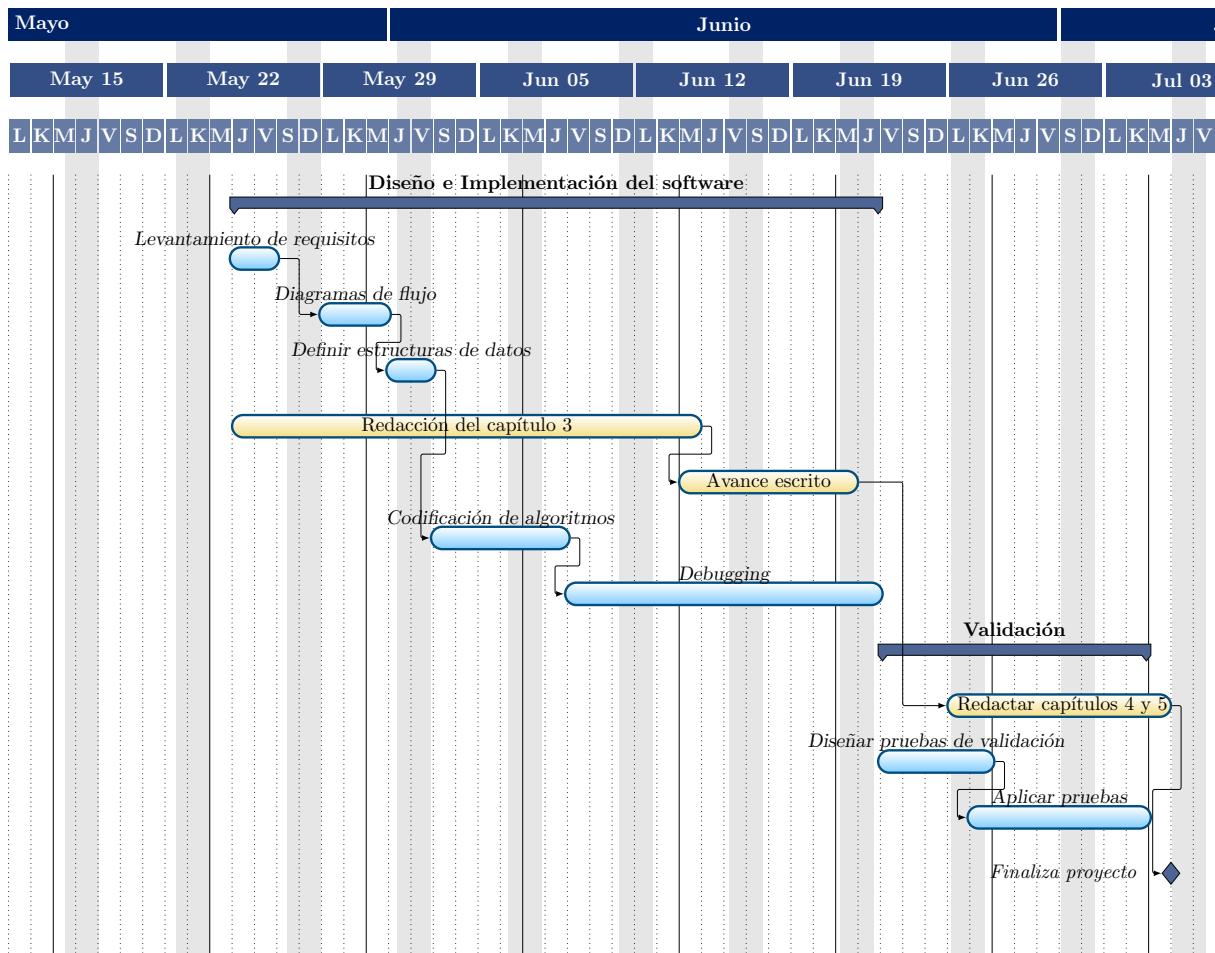


Figura 1.4: Diagrama de Gantt que muestra la planificación para el diseño, implementación y validación del algoritmo desarrollado.

Capítulo 2

Marco Teórico

En esta sección del proyecto, se tratarán temas relacionados a la teoría que se usa como base en todos los programas de Animación Digital para producir movimiento. Aunque no es de sorprenderse, que mucha de la teoría de *Computer Graphics* que se usa para animar en 3D, se basa en principios de robótica como *Inverse Kinematics* o *Forward Kinematics*, combinadas con *Hierarchical Modeling*.

2.1. Aspectos a utilizar de animación 3D

Para el presente proyecto, se hará uso de cierto conceptos de Animación Digital, en particular, modelado jerárquico y *Motion Capture*. Esto es así, debido a que principalmente, se trabajó con *hardware* de captura de movimiento y *software* de animación 3D.

Existe una variedad de formatos para representar los archivos de captura de movimiento, sin embargo, se escogió trabajar con *bvh* ya que éste es uno de los más comúnmente utilizados. De igual manera, existen una variedad de sistemas de captura de movimiento, siendo Optitrack el escogido para este proyecto.

2.1.1. Modelado jerárquico

Al abrir un archivo de captura de movimiento en un programa de animación 3D, como *Blender* o *Maya*, lo primero que notará, es que la captura se representa como un esqueleto en 3D capaz de moverse. Cada sección del esqueleto es llamada hueso o *bone*, y la totalidad de estos elementos se representan como una jerarquía por parte del programa. En la figura 2.1 se puede apreciar dicha jerarquía, la cual suele representarse en los programas como una estructura de datos llamada árbol.

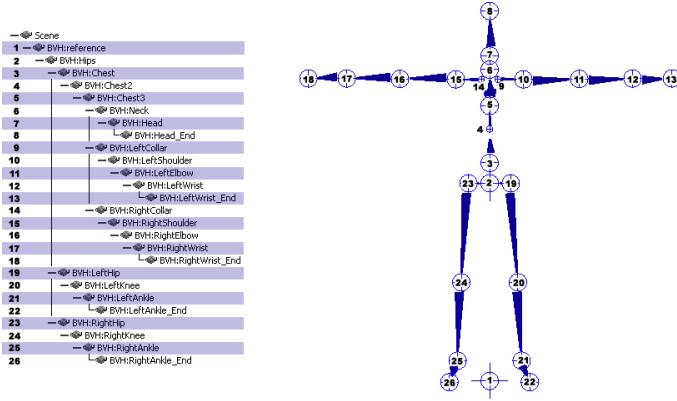


Figura 2.1: Esqueleto de un archivo *bvh* con su respectiva jerarquía. Tomado de CGTalk [[CGTalk, 2017](#)].

El modelado jerárquico consiste en la aplicación de restricciones de movilidad relativa entre diferentes elementos organizados en una estructura similar a un árbol [Parent, 2012]. En otras palabras, cuando se tiene una figura articulada, el modelado jerárquico es la técnica mediante la cual, se le aplican restricciones de movilidad a cada elemento de la figura, para asegurarse que ésta se mueva como una sola cosa y no se desarme.

Usar esta técnica tiene sus ventajas desde el punto de vista computacional. Internamente, cuando los programas de animación quieren representar el movimiento de una figura articulada, recurren al modelado jerárquico, debido a que los árboles facilitan en gran medida los cálculos intermedios para estimar el movimiento en 3D. [Parent, 2012]

Los árboles, son muy comunes en los programas de animación para representar humanos y animales. Lo que normalmente se hace para representar movimiento humano, es definir una estructura similar a la que se muestra figura 2.1. Cada hueso de esta jerarquía se puede representar en un árbol como un nodo, y cada nodo se conecta con otro a través de un arco (véase figura 2.2).

La idea detrás de usar un árbol, es facilitar los cálculos de animación. Para animar cada hueso del cuerpo humano que se mostró anteriormente, el programa de animación, por medio de *OpenGL*, debe calcular una serie de matrices por cada hueso del cuerpo humano. Calcular cada matriz por cada hueso de la jerarquía, cada vez que haya movimiento es muy costoso a nivel computacional.

Dada la complejidad de estos cálculos, lo que se hizo para poder animar más fluidamente, fue utilizar un concepto proveniente de la robótica: *cadenas cinemáticas* [Parent, 2012]. Partiendo de este concepto y la estructura de arbol de la figura 2.2, lo que se hace es asignar a cada arco del árbol una matriz de transformación que afecta al hueso que esté apuntando. Cada una de estas matrices, es responsable de mover cada hueso de forma independiente, con respecto al hueso padre de la jerarquía.

Si cada hueso se mueve con respecto al elemento anterior al que estaba conectado en la jerarquía, no es necesario hacer una gran cantidad de cálculos por cada hueso. La utilidad de las cadenas cinemáticas, es que una vez que se hace el cálculo de una matriz para un hueso, esta se reutiliza para calcular la posición del siguiente *bone* de la jerarquía, hasta llegar al final. Es una forma eficiente de reutilizar

cálculos previamente hechos.

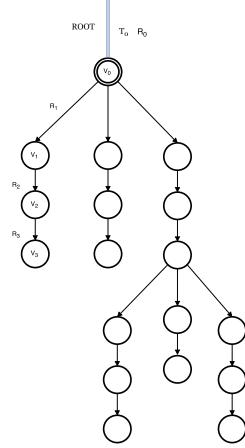


Figura 2.2: Árbol que respresenta un esqueleto. Cada nodo representa un *bone* y cada arco, contiene la matriz de transformación para nodo que toca. Al nodo V_1 le corresponde la matriz R_1 , al V_2 la R_2 y así sucesivamente. Al nodo de Root, por lo general se le asignan dos matrices R_0 y T_0 .

Así pues, si se desea conocer la posición final que tendrá el *bone* representado por V_3 en la figura 2.2, sólo es necesario aplicar todas las transformaciones en cadena que se encuentren en el camino, desde *Root* hasta V_3 :

$$V'_3 = T_0 R_0 R_1 R_2 R_3 V_3. \quad (2.1)$$

Siendo V'_3 la nueva posición de *bone* representado por V_3 .

El modelado jerárquico es utilizado en *Motion Capture* para definir las jerarquías de huesos que posteriormente terminará en el archivo *bvh*, y consecuentemente, la forma en que se configuran los vectores de movimiento, está hecha para que sea más fácil el construir las matrices de transformación para cada hueso del cuerpo.

2.1.2. Motion Capture

Motion Capture, se define como el proceso mediante el cual se graba el movimiento real de un ser humano y luego éste es mapeado a un objeto sintético (modelo 3D particular). Normalmente llamado *MoCap* en el medio, este proceso involucra la detección, digitalización y captura del movimiento de un objeto. El concepto por lo general aplica a capturar movimiento humano, pero en realidad se puede expandir a cualquier objeto [Parent, 2012].

La extracción del movimiento puro a partir de una señal de vídeo, es una tarea bastante difícil que actualmente, es un campo de investigación en si. Es por esto que el proceso de *MoCap* no se vale solamente de una señal de vídeo para reconstruir el movimiento humano en un modelo 3D. Los sistemas actuales de captura de movimiento, se valen de varios tipos de sensores para poder transformar una señal de vídeo, en datos puros que puedan ser usados para animación digital.

Tecnologías de *Motion Capture*

La diferencia entre las tecnologías existentes de *Motion Capture*, radica en qué tipo de sensores se estén utilizando [Parent, 2012]. Actualmente, hay dos tecnologías bien definidas:

1. Sensores electromagnéticos
2. Sensores ópticos.

Los sensores electromagnéticos, dan paso a una tecnología comúnmente denominada *magnetic tracking*. En este tipo de sistemas, los actores a filmar, deben usar sensores magnéticos en sus articulaciones. Estos sensores transmiten su posición a una unidad central de procesamiento y ésta captura los movimientos del actor. La gran ventaja de este sistema, es su gran precisión, al menos a nivel teórico y el hecho, de que la captura de movimiento se puede desplegar en tiempo real.

Para funcionar correctamente, este sistema debe estar en un cuarto aislado, sin ningún tipo de interferencia electromagnética en escena. Esto representa una desventaja, dado que aislar un cuarto de toda interferencia no es barato. Otro pequeño inconveniente, lo presentan los sensores magnéticos. Existen los sensores que transmiten su información vía cables, lo cual fuerza al equipo de producción a utilizar arneses para asegurar los cables. Esto limita el movimiento del actor.

La otra opción que resulta más cómoda, son los sensores inalámbricos, sin embargo, éstos requieren que el actor lleve consigo un paquete de baterías y en ambos tipos de sensores, existen limitaciones de rango y precisión del campo magnético.

La segunda tecnología, es la de sensores ópticos. Los sensores ópticos son mucho más simples y no requieren cables o energía, son solo marcadores reflectores que reflejan la luz. Esta tecnología, utiliza señales de vídeo para grabar imágenes de una persona en movimiento y dado que los datos de posición no son generados de forma directa, se necesitan más sensores entre cada articulación.

Los sistemas ópticos tienen la ventaja de que son de bajo presupuesto en comparación a los magnéticos, sin embargo, una desventaja que presentan, es que son particularmente vulnerables al ruido generado por el exceso de luz [OptiTrack, 2017] y además, son propensos a errores. Los sistemas básicos se construyen con ocho cámaras infrarrojas y de ahí, se puede escalar el sistema hasta usar 16 de ellas.

2.2. Movimiento humano

Ante la gran variedad de movimientos no válidos que pueden presentarse en un vídeo de *MoCap*, es difícil usar términos simples para nombrar un *glitch* cada vez que éste ocurre; cuando se trabaja con un vídeo es necesario poder decir, qué tipo de error se está corrigiendo y dónde. Es por esto, que cada vez fue cobrando más importancia el diseñar alguna manera de clasificar los *glitches* de *MoCap*.

Una clasificación, no puede inventar nombres solo para dividir diferentes categorías. En este proyecto, la idea de crear una taxonomía, es generar información útil por medio de ella. Una lista de características que indican qué tipo de información se quiere obtener por medio de esta clasificación sería:

- El tipo de movimiento.

- ¿En qué dirección se dió?
- ¿A qué miembro pertenece?

Dicha información se puede obtener por medio de la fisiología articular y la goniometría. Dos áreas de las Ciencias Médicas que comparten una amplia cantidad de estudios, que permiten elaborar una clasificación de errores de *MoCap* basada en mecánica humana.

La ventaja de hacer la taxonomía basada en estas dos áreas, es que no se inventan nombres de ningún tipo. Todos están previamente definidos en los libros de Ciencias Médicas de estas áreas y cada nombre que se usa, habla de una articulación y un movimiento en particular.

2.2.1. Goniometría

Goniometría deriva del griego *gonion* ('ángulo') y *metron* ('medición'), es decir: «disciplina que se encarga de estudiar la medición de los ángulos». En las Ciencias Médicas, es la técnica de medición de los ángulos creados por la intersección de los ejes longitudinales de los huesos a nivel de las articulaciones [Taboadela, 2007] (véase figura 2.3).

Según Claudio Taboadela [Taboadela, 2007], en el campo de la Medicina, la goniometría tiene 2 dos objetivos principales:

1. Evaluar la posición de una articulación en el espacio, esto como procedimiento para cuantificar y objetivizar la ausencia de movilidad de una articulación.
2. Evaluar el arco de movimiento de una articulación en cada uno de los tres planos del espacio, esto como procedimiento para cuantificar y objetivizar la movilidad de una articulación.

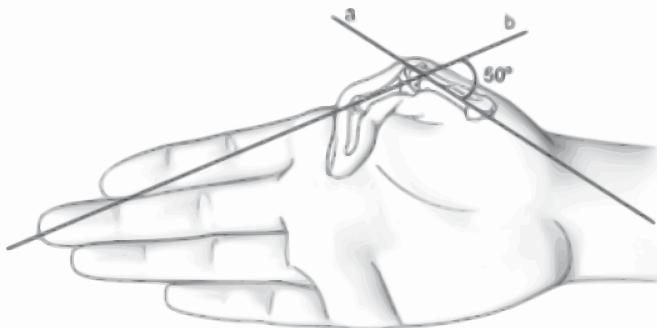


Figura 2.3: En este caso, la intersección de la prolongación de los ejes longitudinales del primer metacarpiano (a) y de la primera falange (b) del pulgar determina un ángulo de 50° a nivel de la articulación metacarpofalángica. Figura y texto tomado de [Taboadela, 2007].

Para efectos del proyecto, el segundo objetivo será el más importante, ya que éste da lugar a estudios de movilidad que se pueden usar para clasificar los movimientos. Los movimientos en los estudios de goniometría, se basan en diferentes planos para definir claramente las trayectorias de los arcos. La planimetría para clasificar los movimientos se muestra en la figura 2.4. A continuación, se definen cada uno de los movimientos de la clasificación según Taboaleda [Taboaleda, 2007]:

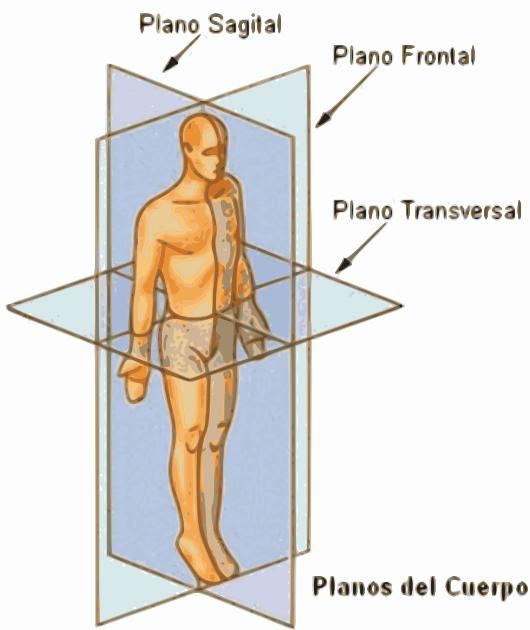


Figura 2.4: Distribución de cada plano en el cuerpo humano. Cada uno de estos planos se usa para describir los movimientos en los estudios de goniometría. Tomado de [Planimetria, 2017].

- **Flexión/Dorsiflexión:** Es todo movimiento en el plano sagital que desplaza una parte del cuerpo hacia delante de la posición anatómica.
- **Extensión o hiperextensión:** Es todo movimiento en el plano sagital que desplaza una parte del cuerpo hacia atrás de la posición anatómica.
- **Abducción:** Es todo movimiento en el plano frontal que aleja una parte del cuerpo de la línea media.
- **Aducción:** Es todo movimiento que en el plano frontal acerca una parte del cuerpo a la línea media.
- **Desviación radial/ulnar:** En la muñeca, la línea media corresponde a la prolongación de la línea media del tercer dedo con la línea media del antebrazo. Cuando la mano se desplaza hacia

la apófisis estiloides del radio, se denomina desviación radial, y cuando lo hace hacia la apófisis estiloides del cúbito, desviación ulnar.

- **Rotación interna/externa:** El movimiento en el plano transversal que desplaza una parte del cuerpo hacia fuera se llama rotación externa, en cambio, cuando la desplaza hacia dentro, se denomina rotación interna.
- **Pronación-supinación:** En el antebrazo, se observa un movimiento de rotación denominado pronación-supinación. En la pronación, el antebrazo gira hacia dentro llevando la palma de la mano hacia abajo, y en la supinación, gira hacia fuera llevando la palma de la mano hacia arriba.

2.2.2. Nomenclatura de movimientos basada en goniometría

Partiendo de la definición de movimientos presentada anteriormente, se puede empezar a construir una clasificación *glitches* de movimiento basado en el nombre de la articulación afectada, junto con el nombre del movimiento no válido. Así, si se tiene un movimiento no válido, éste se puede asociar tanto a un miembro, como a una dirección.

Tome por ejemplo un *glitch* de extensión de codo izquierdo. Con ese nombre, ya se sabe que el error se encuentra en el codo izquierdo y que está en dirección del plano sagital. Concretamente, hacia atrás del plano sagital y más allá del límite permitido de extensión del codo.

Esto representa una ventaja muy grande en los videos de *MoCap*, ya que éstos se basan en ángulos de Euler en los planos sagital, frontal y transversal, para describir los movimientos de cada hueso. El hecho de tener una taxonomía que asocie los errores con planos (y en última instancia ejes (*Z, X, Y*)), es un punto de partida perfecto para localizar y corregir un daño en una articulación.

La idea detrás de localizar el daño de un *glitch* de movimiento de forma precisa, es poder asociarlo a los ángulos de Euler (o ángulos de rotación) de esa articulación en particular. Haciendo hecho esta asociación, es muy fácil aplicar límites de movimiento a sus ángulos para determinar si el hueso está en una posición válida o no. De no estarlo, es muy fácil detectar un *glitch* por medio de *software* usando esta taxonomía.

Para empezar a definir claramente una clasificación de *glitches*, se empezó por determinar dos grandes categorías. La primera la constituyen los *glitches* anatómicos, los cuales responden a eventos en los cuales una articulación del *MoCap* realiza un movimiento anatómicamente imposible. Es decir, excede los límites de los arcos de movimiento humano.

La segunda categoría, viene dada por los *glitches* de ruptura, los cuales pertenecen a un evento muy raro que se da en las capturas de movimiento, cuando al esqueleto de *MoCap* se le separa un hueso del resto del cuerpo. Estos errores no serán objeto de corrección en este proyecto, por razones de tiempo de desarrollo. La figura 2.5 muestra todos los movimientos asociados a los *glitches* anatómicos.

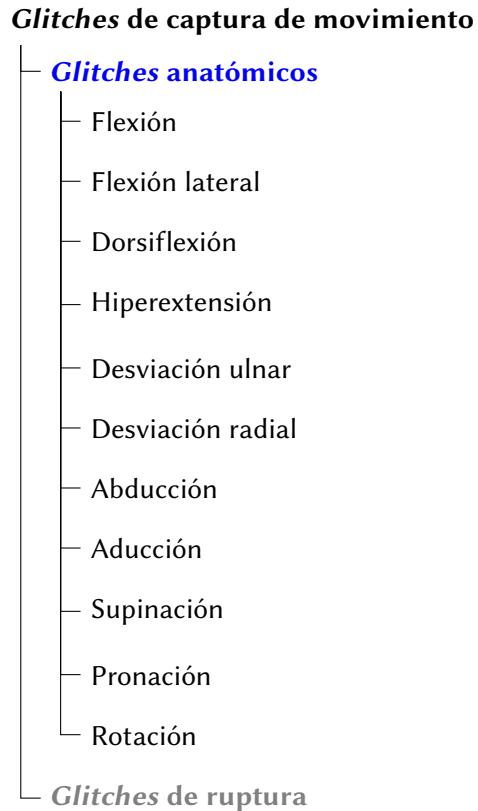


Figura 2.5: Movimientos asociados a cada *glitch*.

Los *glitches* anatómicos contemplan a todas las articulaciones del cuerpo humano y sus movimientos previamente definidos. Como se mostró en el ejemplo anterior del codo, la idea es componer el nombre del *glitch* usando el nombre del movimiento, más el nombre de la articulación afectada. Con esto se puede establecer un mecanismo de detección, localización y corrección de errores de captura de movimiento.

2.2.3. Taxonomía de *glitches*

Habiendo establecido las bases para una taxonomía de *glitches* en la figura 2.5, se procederá a definir todos los errores que se encuentran en una captura de movimiento y cuáles son relevantes para el presente proyecto.

Para empezar, el programa toma en cuenta los errores que se generan en las siguientes articulaciones:

- Espina (lumbar-torácica)
- Espina (cervical)

- Muñeca
- Codo
- Hombro
- Cadera
- Rodilla
- Tobillo

Los *glitches* que no se tomarán en cuenta, serán aquellos que pertenezcan a las manos y los dedos de los pies. De igual manera, se ignoran los *glitches* de las clavículas, ya que arreglar errores que incluyan la clavícula y el hombro, resultó muy complejo de implementar.

El nombre correcto de esta articulación es *articulación escapulohumeral*. Según los libros de anatomía, estas articulaciones poseen movimientos poliaxiales y multiplanares, lo cual indica que los movimientos no se producen en planos y ejes aislados, sino que ocurren en múltiples ejes y planos del espacio a la vez [Taboadela, 2007] (véase figura 2.6). Es por esto, que la articulación escapulohumeral se modeló como una articulación simple de hombro con 3GdL y se le asignaron los límites máximos de la tabla 3.3.

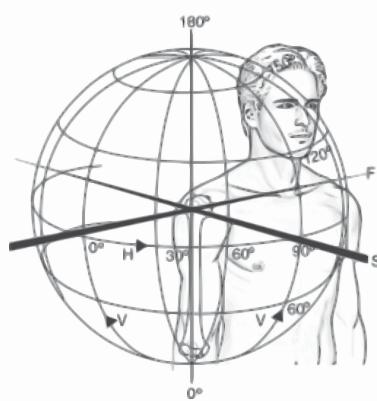


Figura 2.6: Articulación escapulohumeral con su arco de movimiento poliaxial y multiplanar. Tomado de [Taboadela, 2007].

Partiendo de las articulaciones que se tomarán en cuenta, se procederá a establecer la taxonomía de *glitches* que utilizará el programa. Como se expuso anteriormente, se utilizarán los movimientos explicados en la sección 2.2.2 para denominar cada *glitch*. La idea de la nomenclatura establecida, es permitir asociar el *glitch* con un ángulo de Euler en particular, perteneciente a los archivos de *MoCap*. La figura 2.7, sintetiza todos los errores que aparecen en un *MoCap*.



Figura 2.7: Taxonomía de *glitches* del algoritmo. Se pueden apreciar todos los errores junto con los respectivos ángulos de Euler del MoCap que se ven afectados. Los *glitches* marcados en rojo, aplican a todas las articulaciones. Los azules, aplican solo a las muñecas, tobillos o columna vertebral.

Con el objetivo de aclarar cuáles de estos *glitches* aplican a cada articulación, se creó la figura 2.8, la cual contiene una descripción de todos los errores que arreglará el algoritmo, separados por el plano al que pertenecen. La idea del algoritmo, es clasificar cada uno de los *glitches* del *MoCap* en alguna de estas categorías, mediante un proceso de **Reconocimiento de Patrones**.



Figura 2.8: *Glitches que el algoritmo puede arreglar, separados por el plano al que pertenecen.*

2.3. Reconocimiento de patrones

El Reconocimiento de Patrones (RP) es el proceso de identificar objetos o tomar decisiones, a partir de datos puros. El reconocimiento, se lleva a cabo buscando la correspondencia entre un patrón dado y un conjunto de patrones predefinidos [Lifshits et al., 2004] [Duda et al., 2000]. En el contexto del presente proyecto, el proceso de RP consiste en la **clasificación** de los *frames* dentro de un archivo de *Motion Capture*.

Tal y como se aprecia en la figura 2.9, el proceso de RP se separó en tres etapas bien definidas, dentro de las seis etapas del algoritmo en general. Piense en el algoritmo, como un Sistema de Reconocimiento de Patrones (SRP) que recibe datos puros de *MoCap* a la entrada, y a la salida, genera un archivo *bvh* corregido.

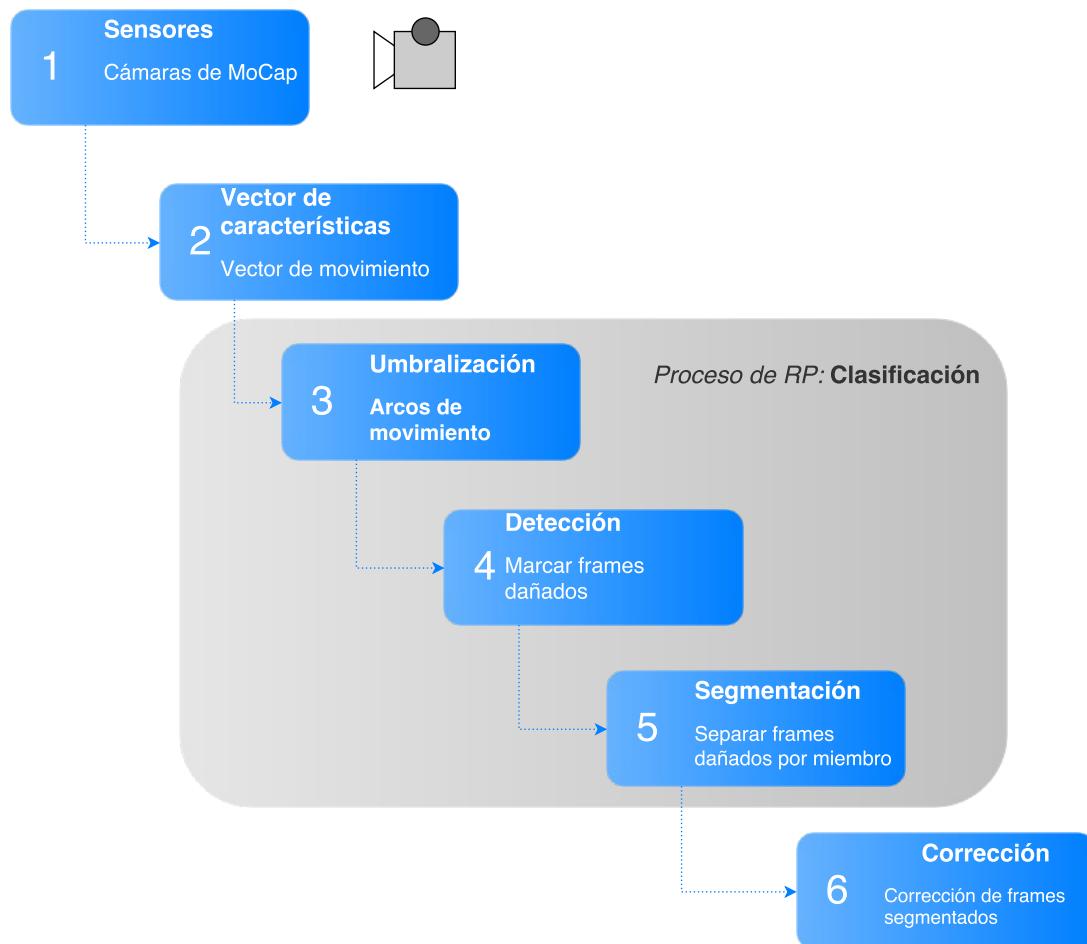


Figura 2.9: Proceso de Reconocimiento de Patrones llevado a cabo por el programa.

Para explicar el proceso general que lleva a cabo el algoritmo, se explicará cada etapa por separado:

2.3.1. Sensores

En esta aplicación en particular, los sensores del SRP son las cámaras infrarrojas del sistema de *Motion Capture*. Estas cámaras generan un conjunto de datos en un archivo CSV, el cual contiene todas las coordenadas de cada miembro del cuerpo humano por *frame*.

El procesamiento de este archivo por parte del *software* de producción, convierte este archivo *csv* en un archivo *bvh*, el cual se utiliza para generar el *vector de características* o *features*.

2.3.2. Vector de características

Dentro de cada archivo *bvh*, existe una matriz de grandes dimensiones llamada MOTION. Esta matriz de movimiento, esta conformada por todos los vectores necesarios para animar el esqueleto de *MoCap* en un espacio 3D. La forma en que los programas leen esta matriz, es separándola fila por fila, y utilizando cada una como un vector de movimiento.

Cada vector de movimiento, contiene los ángulos de traslación y rotación necesarios para mover todos huesos del cuerpo humano en un *frame*. La idea del algoritmo, es extraer esta información del archivo *bvh* y formar el vector de características a partir de cada línea de la matriz de MOTION. Con esto, el algoritmo tendrá toda la información que necesita para iniciar el proceso de clasificación.

2.3.3. Umbralización

El primer paso para realizar el proceso de RP, es aplicar un algoritmo de umbralización al vector de características. Se sabe que las articulaciones tienen límites de movimiento, expresados en arcos de movilidad. Dichos arcos de movilidad, están bien definidos en los estudios de goniometría. Partiendo de dichos estudios, lo que hace el algoritmo de umbralización, es comparar los límites máximos de movimiento de cada hueso, con los ángulos de Euler de cada articulación en el vector de características.

Dado que los límites están expresados en ángulos de Euler, que son las mismas unidades del vector de características, no hace falta ningún cambio de representación de los datos. La comparación es directa. El propósito del algoritmo es determinar si hubo o no, alguna violación de los límites de movilidad humana, por parte del esqueleto de *MoCap*, en el *frame* actual.

2.3.4. Detección

La detección, es el siguiente paso a ejecutar después de que el algoritmo de umbralización encuentra una violación de los límites de movilidad. Se define **detección** como la identificación de elementos, eventos u observaciones, que no se conforman a un comportamiento esperado o a otros elementos dentro de un conjunto de datos [Chandola et al., 2009].

La esencia del paso de detección, es que en el momento en que los límites de movilidad se ven violentados, el algoritmo considera el evento como un *glitch* y procede a identificar el presente *frame* como defectuoso. El algoritmo aplica la umbralización y la detección a cada vector de características que extraído del *MoCap*, y para cuando éste termina el proceso, se obtiene una clasificación que divide a los *frames* de un *MoCap* en dos grupos: defectuosos y no defectuosos.

2.3.5. Segmentación

La clasificación de los *frames* en defectuoso o no, permite llevar a cabo otro proceso importante, en el cual, se dan a conocer las secuencias de *frames* consecutivos en las cuales se detectó un *glitch* en la misma articulación. Este proceso se conoce como **segmentación**. En este sistema de RP, la segmentación se da gracias a dos subproductos de la clasificación: la localización temporal y la localización espacial.

Localización espacial

La capacidad de detección del algoritmo, puede decirnos que se encontró un error en el *MoCap*, sin embargo, no es capaz de decirnos por si sola, en cuál articulación está ubicado el error. La capacidad del algoritmo de estimar en qué articulación se presentó un error, se conoce como *localización espacial*.

Para lograr ubicar un error a nivel espacial, el programa se vale de una de las características más relevantes del formato *bvh*, el modelado jerárquico [Biovision BVH, 2017]. En este tipo de modelado, una figura articulada compleja, se subdivide en una serie de elementos que se pueden manipular por aparte.

Esta técnica se vale de estructuras de datos conocidas como *árboles*, en las cuales cada hueso del *MoCap* es un nodo del árbol, y a su vez, cada nodo contiene su propia información de movimiento [Parent, 2012]. Si algún hueso viola alguna restricción de movimiento, es muy fácil decir cuál de todos los nodos tiene el error.

Localización temporal

Por otra parte, la localización temporal se lleva a cabo de forma simple: si se encuentra *glitch* en alguna parte de la captura, la información de cada hueso del esqueleto del *bvh*, se encuentra consignada en un vector de movimiento que pertenece a una matriz. Al determinar cuál fila de la matriz de movimiento contiene el vector, se sabrá en cuál *frame* del vídeo ocurrió el error.

Al llevar un control de cada *frame* y qué *glitches* presentó cada uno, se cuenta con una forma de localizar temporalmente los errores en el *MoCap*.

2.3.6. Corrección

Finalmente, al conocer la ubicación del error tanto espacial como temporalmente, se puede proceder a modificar las coordenadas del vector de movimiento correspondiente. El concepto de corrección, significa alterar los datos de movimiento en una articulación particular, en un *frame* dado, usando como referencia los máximos valores permitidos para un movimiento en particular.

La lista de movimientos y sus valores máximos permitidos por articulación, se pueden encontrar en el capítulo 3 en la tabla 3.3. Para determinar con precisión qué eje de rotación se debe corregir, se creó la taxonomía de *glitches* expuesta con anterioridad, la cual permite asociar ejes con movimientos tal y como se explicará en el capítulo 3 mediante la tabla 3.2.

Capítulo 3

Diseño e implementación del algoritmo

3.1. Introducción

En este capítulo, se habla de todo lo relacionado con el diseño e implementación del programa en Python. Se cuenta con una serie de secciones que hablan de la arquitectura del programa, la cantidad de tareas, diagramas de flujo, etc. Todo con la intención de documentar bien el *framework* desarrollado para la resolución del problema. Antes de iniciar con la primera sección, conviene hablar del problema de separar la detección, localización y corrección, y de cómo se resolvió.

En los inicios del proyecto, se propuso la idea de realizar un programa que fuera capaz de corregir videos que tuvieran errores de captura de movimiento, tales como, movimientos no válidos o articulaciones quebradas. Tal enfoque se limitaba solamente a la **corrección** de errores y dependía de otros programas para la detección y localización de errores en un archivo de *Motion Capture*, sin embargo, se tuvo que desechar esa idea, por los inconvenientes a la hora de abordar la solución de *software*.

Lo primero, es que no resulta buena idea la separación entre las tres tareas, debido a la estructura que presentan los archivos *Biovision Hierarchy*. Estos archivos, representan todos los datos de movimiento en una sola matriz de grandes dimensiones. Si se pretende corregir una captura, hay que **localizar** la fila y la columna de esa matriz que contiene el error, para después **corregirla**.

Aunque esto no parece de mayor complejidad, las cosas cambian cuando se usan *MoCaps* reales, en los cuales la cantidad de entradas en una matriz puede sobrepasar fácilmente las 65000. Un programa que corrija un BVH no solo debe ser capaz de darse cuenta que hubo un error. Debe localizar las articulaciones dañadas y en cuál cuadro ocurrieron, para así dar con la entrada correcta de la matriz de movimiento.

La razón más importante por la que la separación de tareas se tuvo que desechar, fue debido a que hacer la detección y localización a mano en un archivo real, es sumamente tedioso. Tomar una matriz de mas de 65000 entradas y revisarla elemento por elemento, para decirle al programa donde están los *glitches*, no tiene sentido. La idea del *software* es automatizar las tareas, no hacerlas peor.

Es por esto, que en el levantamiento de requisitos, se propuso desarrollar un programa que abordara las tres tareas de detección, localización y corrección. Partiendo de estas tres cosas, el programa tiene mejor desempeño y capacidad para arreglar los archivos dañados sin sacrificar la automatización de las tareas. En la siguiente sección, se hablará más en detalle de qué requisitos se tomaron en cuenta.

3.2. Levantamiento de requisitos

Al inicio del diseño, se estableció la siguiente lista de requisitos que el programa debe cumplir:

- El programa deberá ser capaz de leer un archivo BVH con *Root Translation only* y representación de Euler (Z, X, Y).
- El algoritmo debe recibir un archivo BVH a la entrada y generar un nuevo archivo corregido tipo BVH a la salida.
- El programa debe de ser capaz de detectar que hubo errores de movimiento en el archivo BVH de entrada.
- El programa debe ser capaz de localizar en qué articulación se dió el *glitch* detectado (localización espacial).
- El *software* debe ser capaz de localizar en qué *frame* o cuadro del vídeo ocurrió un error de *MoCap* (localización temporal).
- Basándose en la localización espacial y temporal, el programa tiene que ser capaz de identificar en qué dirección no válida se dio el movimiento, para que éste pueda ser corregido en el eje correcto (Z, X o Y).
- El programa debe ser capaz de entender los límites de movimiento humano y aplicarlos a la hora de corregir un error.
- El programa deberá ser capaz de corregir *glitches* que sean de un *frame* de duración o que se prolonguen por varios *frames*.

Partiendo de esta lista de requisitos, se tiene un programa bastante completo para resolver el problema propuesto. Una cosa que vale la pena mencionar, es qué tipos de errores no se tomarán en cuenta en este programa.

Dado que hay errores que no se pueden corregir por factores externos, se decidió no incluir los *glitches* de manos en el programa. La razón, es que muchos de los errores que existen en las manos, se dan por interferencia de sensores causada por exceso de luz en el cuarto de filmación [OptiTrack, 2017] y, por lo general, los equipos de producción optan por animar las manos de un personaje manualmente. Los *MoCaps* más comunes hechos en el laboratorio, no toman en cuenta los dedos de las manos.

3.3. Arquitectura del programa y tareas principales

En esta parte del diseño, se dividió el algoritmo a programar en 4 grandes tareas. A cada tarea le corresponde una parte importante relacionada a la detección, localización y corrección de errores. En esencia, el algoritmo lleva a cabo las siguientes tareas:

1. Leer el archivo de captura de movimiento y crear una jerarquía de *bones* basado en el apartado HIERARCHY.
2. Asignar un *bone* a cada entrada del vector de movimiento. Existe un vector de movimiento por cada *frame* del vídeo.
3. Aplicar estudios de goniometría a cada hueso de la jerarquía e identificar cuáles movimientos no son válidos en el vector.
4. Una vez corregida toda la matriz de movimiento, reescribirla en el archivo BVH corregido.

La figura 3.1 muestra las cuatro tareas anteriores asociadas con los tres conceptos más importantes que debe seguir el programa: detección, localización y corrección. Cada una de estas tareas tiene su propia sección para ser desarrollada en detalle a continuación. Para facilitar la explicación de cada tarea, se incluirá un apartado en el cual se explicará el formato de los archivos *bvh*.

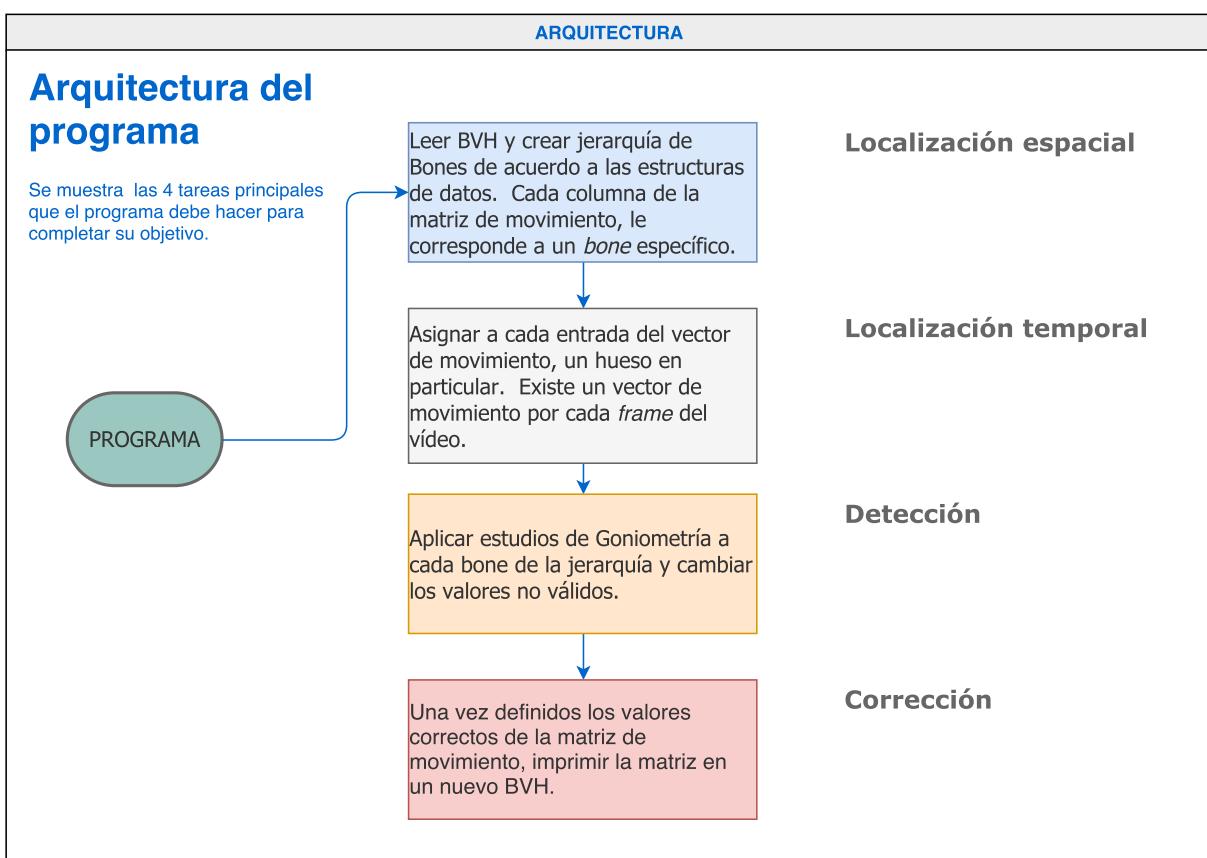


Figura 3.1: Arquitectura del programa.

3.3.1. Acerca de los archivos *bvh*

El formato **Biovision Hierarchical Data** fue creado por la empresa Biovision, la cual se dedicada a brindar servicios de *Motion Capture* en su momento, y necesitaba una forma de brindarle datos de *MoCap* a sus clientes [Biovision BVH, 2017]. Es un formato de archivo de texto simple. Contiene dos grandes secciones claramente definidas: el apartado **HIERARCHY** y la sección de **MOTION**. La sección de **HIERARCHY** define el esqueleto humano como una jerarquía, mientras que la sección de **MOTION** lo mueve.

En general, los programas de animación pueden utilizar estructuras jerárquicas para representar figuras articuladas, como un esqueleto, y ser capaz de animarlo. El modelado jerárquico consiste en la aplicación de restricciones de movimiento a todo un conjunto de elementos en una estructura jerárquica similar a un árbol [Parent, 2012].

Otra forma de verlo, es que el modelado jerárquico le permite a un animador animar un conjunto completo de objetos (por ejemplo huesos) como si fueran una sola cosa, sin preocuparse de que éstos se separen a la hora de moverse, ya que la unión entre ellos es parte del modelo [Parent, 2012].

Dicho esto, la sección de **HIERARCHY** define el cuerpo humano como un solo modelo y lo que se obtiene a la hora de leer esa información son todos los huesos de esqueleto a animar. La sección de **HIERARCHY** se ve así en todos los archivos BVH:

```
HIERARCHY
ROOT Hips
{
    OFFSET      0.00      0.00      0.00
    CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
    JOINT Chest
    {
        OFFSET      0.00      5.21      0.00
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT Neck
        {...}
    }
}
```

El nodo o hueso raíz contiene siempre la palabra clave **ROOT**, en este punto vale la pena anotar, que los huesos de un *bvh* pueden ser tratados como nodos de un árbol, ya que estamos hablando de una estructura jerárquica. La forma en la ésta se arma, consiste en seguir el orden que le da el *bvh*.

Como puede notar, cada hueso (nodo) de la estructura está definido por la palabra **ROOT**, en el caso del nodo raíz, o **JOINT** para todos los demás. Cada hueso contiene una lista de parámetros que definen una serie de características importantes. A saber:

- **OFFSET:** define la distancia del nodo actual con respecto al nodo padre en la jerarquía.
- **CHANNELS:** define los grados de libertad que posee un hueso en particular. Por lo general todos tienen 3 GdL, solo el nodo **ROOT** posee 6 porque incluye las coordenadas de traslación x, y, z.

Vale aclarar, que el orden en el que vienen los datos en cada línea de la sección de MOTION, corresponde con el orden en el que se muestran los CHANNELS para cada articulación [Biovision BVH, 2017].

Partiendo de esto, si el nodo ROOT está primero y tiene 6 GdL (Xposition, Yposition, Zposition, Zrotation, Xrotation, Yrotation), los primeros seis valores del vector de MOTION corresponderán al hueso ROOT, luego vendrán tres valores seguidos que representarán (Zrotation, Xrotation, Yrotation) para el hueso *Chest*, luego los 3 valores (Zrotation, Xrotation, Yrotation) para el siguiente hueso, y así hasta terminar la jerarquía.

La sección de MOTION es mucho más simple. Consiste en una matriz extensa en donde cada fila tiene un vector de 156 posiciones, que representa un *frame* del vídeo. Como ya se explicó, a cada posición de esas 156, le corresponde un valor de Zrotation, Xrotation, Yrotation de algún hueso de la jerarquía. Como cada hueso tiene 3 GdL, el vector de MOTION se reparte de tres en tres, es decir, a cada hueso le corresponden tres valores seguidos del vector de movimiento.

Siguiendo con el *bvh* de ejemplo, la sección de MOTION se muestra a continuación:

```
MOTION
Frames:      2
Frame Time: 0.033333
8.03 35.01 88.36 -3.41 14.78 -164.35 13.09 40.30 -24.60 ...
7.81 35.10 86.47 -3.78 12.94 -166.97 12.64 42.57 -22.34 ...
```

La constante **Frames** indica la cantidad de cuadros que tiene el vídeo, dos para este caso; siguiendo la lógica del formato, la cantidad de *frames* será igual a la cantidad de líneas de MOTION. **Frame Time** indica cuántos segundos tarda un *frame* del video, aunque comúnmente, lo que se usa es el inverso de este valor para conocer los *fps* (*frames per second*) de la captura.

El orden en que se interpretan los valores de cada fila ya se explicó con anterioridad, sin embargo, hay un detalle final por explicar. En la sección de HIERARCHY, se puede apreciar que en cada CHANNEL el orden de las coordenadas por lo general viene dado por Z, X, Y. Estos valores, son ángulos de Euler expresados en grados, y siempre se deben considerar como ángulos pertenecientes a un sistema de coordenadas local, por cada *bone*.

3.3.2. Leer el *bvh* y crear una jerarquía de *bones*

Habiendo estudiado el formato BVH para capturas de movimiento, se facilita mucho la interpretación de la información que éste contiene. El primer paso que ejecuta el programa es crear una jerarquía de *bones* basándose en la sección de MOTION del *bvh*. Utilizando los nombres de los JOINT o huesos de la sección HIERARCHY, se puede crear una jerarquía que le asigne a cada hueso del cuerpo, una entrada del vector de MOTION.

Para crear dicha jerarquía, se lee el archivo *bvh* mediante una línea de código muy sencilla en Python 3:

```
BVHfile = open(sys.argv[1], 'r')
```

```
# Con este ciclo, se itera línea por línea del archivo bvh
for line in BVHfile.readlines():
    ...

```

Dado que esta instrucción es muy simple, se puede ubicar en el diagrama de flujo del programa principal ubicado en el apéndice A. Se puede identificar como la primera tarea del mismo.

Esta parte del programa tiene condiciones de frontera muy sencillas. Toma por entrada un archivo de texto y genera al terminar un *file object* perteneciente a la interfaz *TextIOWrapper*, la cual permite manipular texto muy fácilmente, como si se tratara de arreglos comunes; en este caso, el programa separa el *bvh* en líneas.

3.3.3. Asignar un *bone* a cada entrada del vector de movimiento

Para explicar mejor este paso, vale la pena hacer una tabla que diga el nombre que le asignamos a cada hueso y las tres entradas que le corresponden del vector de MOTION. Note que los nombres de los huesos, y las clases que los representan, se basan en los nombres que les da el archivo *MoCap*.¹.

La definición de la jerarquía de *bones* se basa en la tabla 3.1

Hueso	Articulación	Índices de MOTION		
		Z	X	Y
LeftUpLeg	Cadera Izquierda	132	133	134
RightUpLeg	Cadera Derecha	144	145	146
LeftLeg	Rodilla Izquierda	135	136	137
RightLeg	Rodilla Derecha	147	148	149
LeftFoot	Pie Izquierdo	138	139	140
RightFoot	Pie Derecho	150	151	152
LeftArm	Hombro Izquierdo	21	22	23
RightArm	Hombro Derecho	78	79	80
LeftForeArm	Codo Izquierdo	24	25	26
RightForeArm	Codo Derecho	81	82	83
LeftHand	Muñeca Izquierda	27	28	29
RightHand	Muñeca Derecha	84	85	86
Spine1	Columna (lumbar-torácica)	9	10	11
Neck	Columna (Cervical)	12	13	14

Tabla 3.1: Jerarquía de huesos creada por el programa. Se muestran los índices de MOTION que le corresponden a cada uno y a su vez, la articulación real a la que corresponden. No todos los índices del vector de movimiento se utilizan.

Una vez definida la jerarquía, ya se cuenta con una forma de localizar espacialmente los *glitches* de la captura; note que cada vector (*Z*, *X*, *Y*) se puede asociar con un movimiento en particular y con un

¹Para más información sobre el diseño de clases, consulte el apéndice B

miembro del cuerpo, ya que el vector (Z, X, Y) representa los ángulos de rotación de la articulación, en un sistema de coordenadas local.

Las condiciones de frontera de esta tarea son bastante simples: no se reciben datos de entrada para iniciar esta tarea y a la salida, se genera un conjunto de objetos pertenecientes a la subclases **Arm**, **ForeArm**, **Wrist**, **Foot**, **Leg**, **UpLeg** y **Spine**. Cada uno de estos objetos contiene la información de la tabla 3.1, con la cual se pueden asociar los errores detectados en MOTION, con un *bone* en particular. En la sección 3.5 se hablará más del diseño de cada una de estas subclases.

3.3.4. Aplicación de los estudios de goniometría

Los estudios de goniometría, consisten en una serie de límites de movilidad para cada una de las articulaciones del cuerpo humano. Para este proyecto, se utilizaron los estudios de movimiento hechos por Kathryn Luttgens [Luttgens et al., 2011], ya que presentan la ventaja de que cada movimiento está claramente definido y puede ser asociado a un eje de coordenadas en particular.

Esto representa una gran ventaja para el diseño del programa. El hecho de que cada movimiento listado en la tabla 3.3, se pueda asociar a un eje de rotación Z , X o Y en el vector MOTION, facilita mucho la localización de errores.

La tabla 3.2 muestra la asociación entre ejes y movimientos para cada articulación. La combinación de la jerarquía de *bones* junto con la aplicación de estos estudios, crea la interfaz perfecta para llevar a cabo la detección y localización de *glitches* por parte del algoritmo.

Movimiento	Eje asociado
Aducción / Abducción	Z
Flexión lateral	Z
Flexión / extensión palmar	Z
Flexión / Hiperextensión	X
Dorsiflexión	X
Desviación	X
Rotación	Y
Pronación / Supinación	Y

Tabla 3.2: Ejes Z , X , Y asociados a movimientos del cuerpo humano encontrados en la tabla 3.3.

La idea del programa, es usar los límites de la tabla 3.3 para la detección de *glitches* de movimiento por medio de la función *Goniometry_check*². Esta función o método, consiste en analizar los tres valores del vector de MOTION que pertenezcan a una articulación en particular y separarlo en sus componentes Z , X , Y .

Una vez hecho esto, si algún valor de rotación se excede de los límites permitidos, hay que cambiarlo por el valor máximo que el cuerpo permita. Así si el valor máximo de flexión de un codo son 145° , cualquier codo que se flexione más de eso, solo se le permitirá flexionarse hasta 145° .

²En el apéndice A se encontrará un diagrama de flujo detallado de la función.

El concepto general es muy simple. Si ya se sabe qué partes del vector de movimiento le pertenecen a cada articulación, eso quiere decir, que ya se conoce qué límites aplicarle a cada entrada del vector MOTION. Aquellas entradas que no cumplan las restricciones de movimiento serán cambiadas de una vez.

En términos del sistema de Reconocimiento de Patrones explicado en la sección 2.3, la función *Goniometry_check* aplica el algoritmo de umbralización necesario para poder clasificar los *frames*. En general el proceso inicia con la umbralización, la cual permite encontrar los errores. La detección permite clasificar el *frame*. La clasificación da paso a la segmentación, la cual se basa en localización espacial y temporal de *glitches*, y una vez que se conocen todos los *glitches* se procede a arreglarlos. La figura 3.2 muestra el proceso de umbralización.

Las condiciones de frontera de esta tarea, se pueden resumir así: las entradas serían la jerarquía de *bones* y el vector de MOTION sin alteraciones. A la salida de esta tarea, lo que se genera es un vector nuevo de MOTION, que se grabará en posteriormente en el archivo *bvh* corregido.

3.3.5. Corrección del archivo de MoCap

Finalmente, esta sería la última tarea del programa en la cual, se recibe el vector de movimiento con sus 156 entradas corregidas, y se imprime en el archivo *bvh* corregido. Esta tarea se lleva a cabo mediante una sola instrucción de código llamada *outputBVH.write(salida)*, donde *outputBVH* es otro *file object* al cual se le escribe el vector de movimiento. Lo anterior, describe las condiciones de frontera, en las cuales se puede ver que la salida de esta tarea, es la salida del programa como tal.

Diagrama de flujo del programa principal

En términos generales, las secciones 3.3.2 a la 3.3.5, describen las tareas principales del algoritmo. Al poner toda esta información en un diagrama de flujo, se obtiene un diagrama que describe al programa principal en su totalidad. La figura 3.3, contiene dicho diagrama junto con sus estructuras de datos.

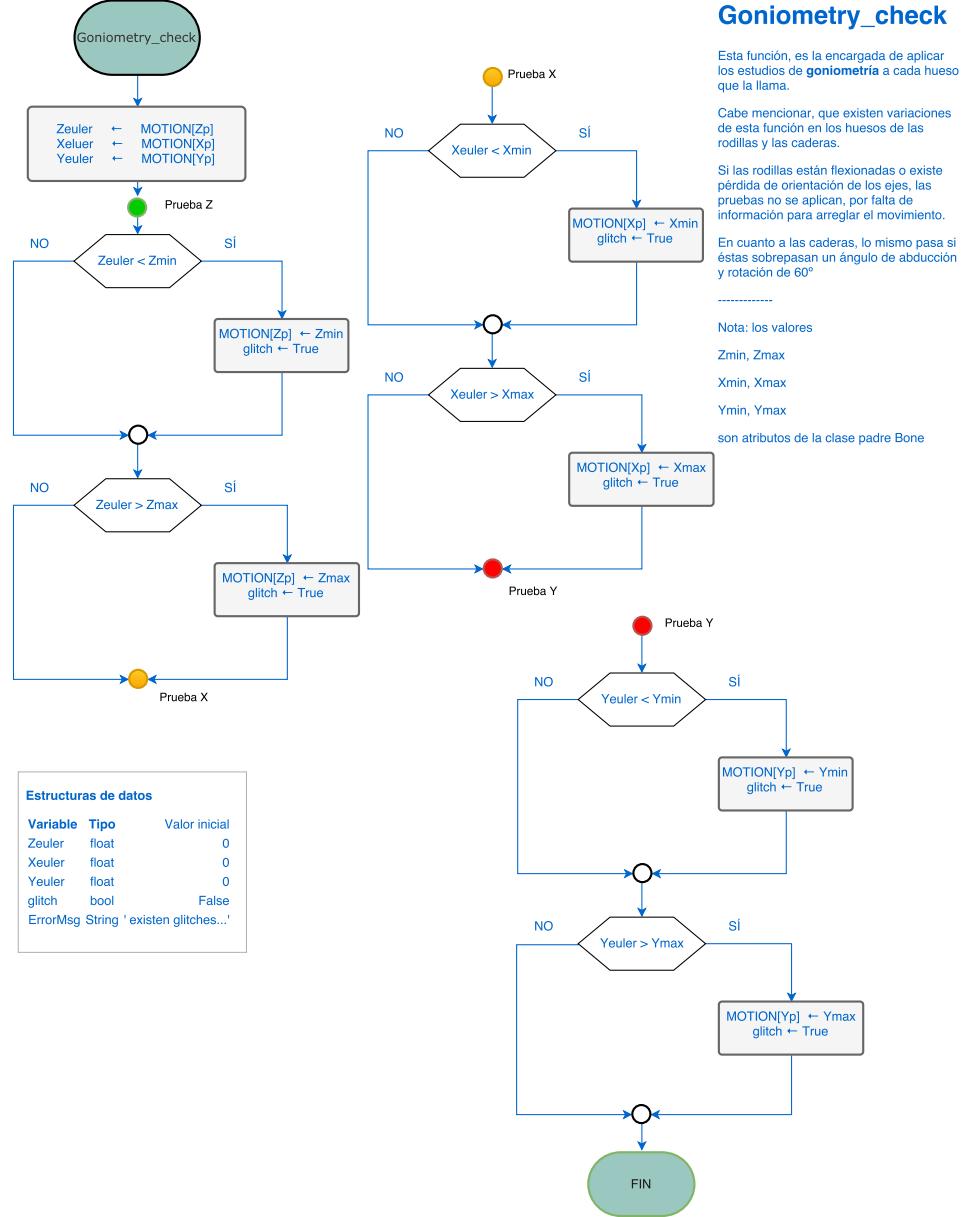
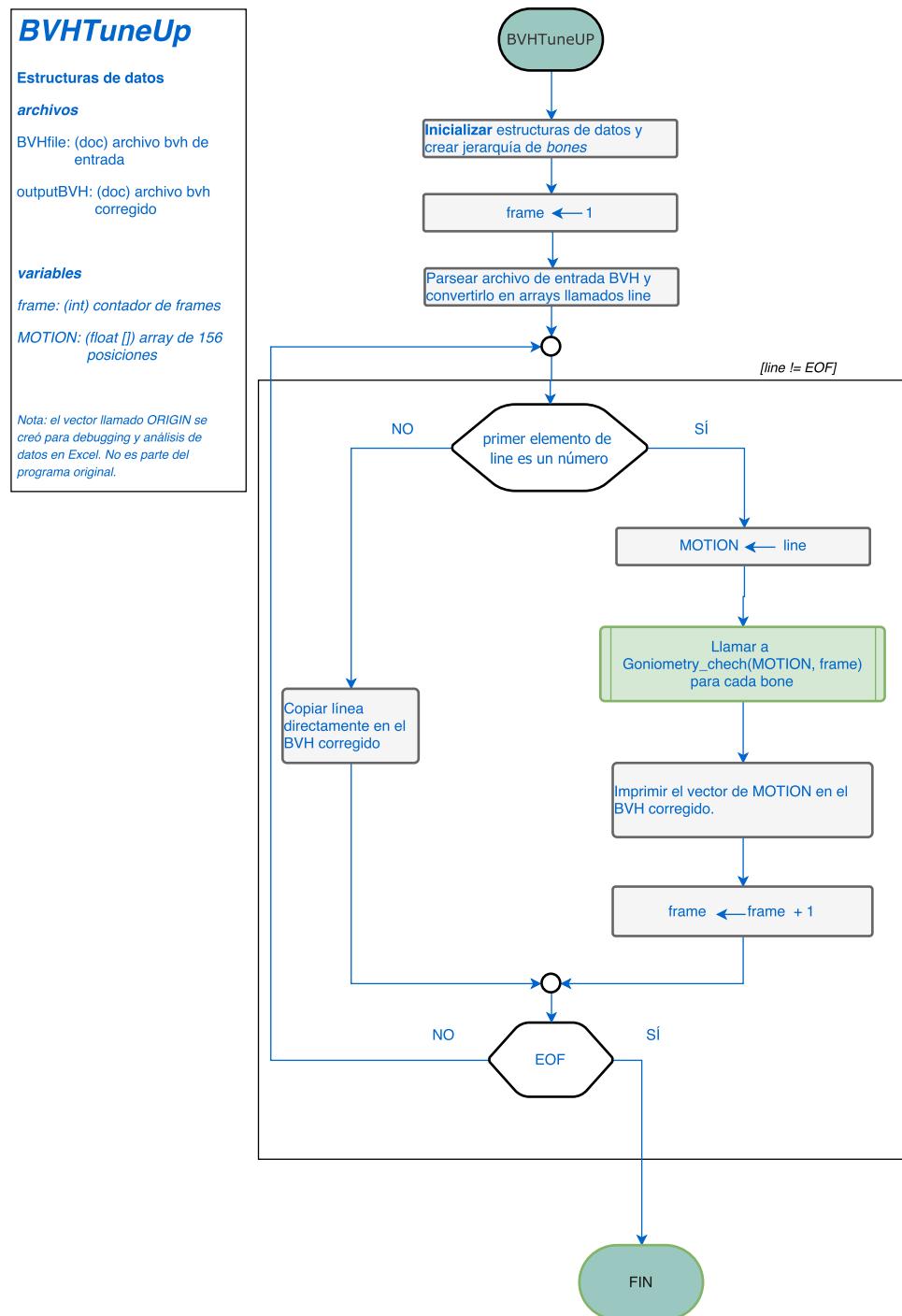


Figura 3.2: Diagrama de flujo de la función *Goniometry_check* que implementa el algoritmo de umbralización. Note como una violación de los límites, se arregla inmediatamente en el vector de características MOTION.

Articulación	Movimiento	Ángulos [°]			
		fuente 1	fuente 2	fuente 3	fuente 4
Codo	Flexión	140	145	145	145
	Hiperextensión	0	0	0	0-10
Brazo	Pronación	80	90	90	80
	Supinación	80	85	90	90
Muñeca	Extensión (Dorsiflexion)	60	70	70	50
	Flexión (flexión palmar)	60	90	-	60
	Desviación Radial	20	20	20	20
	Desviación Ulnar	30	30	35	30
Hombro	Flexión	180	170	130	180
	Hiperxtensión	50	30	80	60
	Abducción	180	170	180	180
	Aducción	50	-	-	-
Hombro con brazo abducido	Rotación interna	90	90	70	60-90
	Rotación externa	90	90	70	90
	Abducción horizontal	-	-	-	135
	Aducción horizontal	-	-	-	45
Cadera	Flexión	100	120	125	120
	Hiperxtensión	30	10	10	30
	Abducción	40	45	45	45
	Aducción	20	-	10	0-25
Cadera extendida	Rotación interna	40	35	45	40-45
	Rotación externa	50	45	45	45
Rodilla	Flexión	150	120	140	130
Tobillo	Flexión plantar	20	45	45	50
	Dorsiflexión	30	15	20	20
Espina (cervical)	Flexión	60	-	-	40
	Hiperxtensión	75	-	-	40
	Flexión lateral	45	-	-	45
	Rotación	80	-	-	50
Espina Lumbartorácica	Flexión	45-50	-	-	45
	Hiperxtensión	25	-	-	20-35
	Flexión lateral	25	-	-	30
	Rotación	30	-	-	45

Tabla 3.3: Límites de movilidad del cuerpo humano según Luttgens [Luttgens et al., 2011]. Estos valores límite son los que se usan de referencia a la hora de instanciar las clases. Tomada de *AOKHealth.com* [AOK, 2017].

Figura 3.3: Diagrama de flujo del programa principal *BVHTuneUP*.

En la sección 3.5, se explica cómo se implementaron las estructuras de datos por medio de un esquema de clases y herencia. Las ventajas que ofrece utilizar Programación Orientada a Objetos, radica en la flexibilidad que presenta el paradigma para definir distintas clases con diferentes versiones del algoritmo de umbralización para cada *bone*, y diferentes atributos para cada *bone* de la jerarquía del *bvh*.

3.4. Interfaz de entrada/salida

En esta sección, se explica la interfaz de entrada/salida del programa con un poco más de detalles. La figura 3.4, muestra cómo un archivo de texto se interpreta para sacar de éste una serie de estructuras de datos que se transforman en información útil para resolver el problema.

En el proceso de detección, localización y corrección, se generan estructuras de datos intermedias que generan información útil para la siguiente etapa. En la sección 3.3, se habló en detalle de las condiciones de frontera por cada tarea y la figura 3.4, representa un resumen de los datos que genera cada una de las tareas expuestas anteriormente.



Figura 3.4: Diagrama de interfaces de entrada salida separado por tareas básicas.

3.5. Implementación de clases y subclases

En esta sección, se expone el código implementado en Python para mostrar los detalles finales del proyecto en sí. No existe mucha diferencia entre los diagramas de flujo expuestos en el apéndice A y el código presentado en esta sección, dado que Python permite implementar las instrucciones de los diagramas muy fácilmente. Para ver la totalidad del código, consulte el apéndice C.

El paradigma que se escogió para diseñar el algoritmo fue el de Programación Orientada a Objetos. El diseño de las clases, se llevó a cabo siguiendo el diagrama UML de la figura 3.5. En este diseño, se creó una superclase llamada *Bone*, la cual contiene todos los atributos generales encontrados en cualquier hueso de la jerarquía. La idea del diseño, es que todas las subclases hereden los atributos de la superclase *Bone*, y que a su vez, cada subclase implemente sus propia versión del algoritmo de umbralización.

Esto es así, debido a que cada subclase representará un *bone* diferente de la jerarquía, y cada *bone* posee límites de movilidad distintos; otro rasgo exclusivo de cada hueso, son las condiciones de excepción, en las cuales, se le deben aplicar pruebas de goniometría distintas a la articulación, dependiendo de su posición.

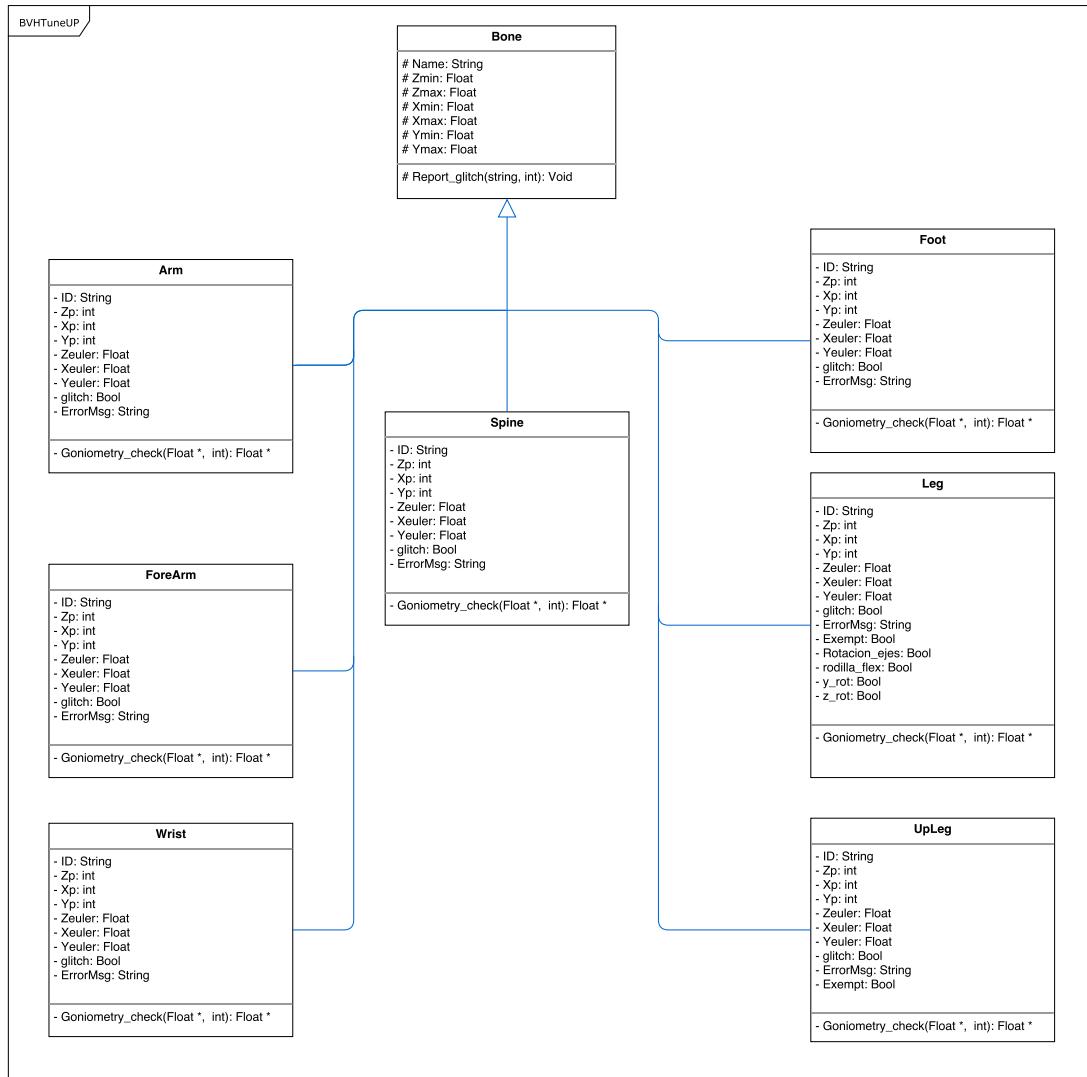


Figura 3.5: Diagrama UML para superclases y subclases.

Debido a las características que requiere implementar cada hueso, se decidió utilizar herencia desde un principio. Esto no solo facilita la implementación, sino también, le permite al programa agregar nuevas características en medio del desarrollo, algo muy común en estos proyectos. Eso sin mencionar, que el mantenimiento posterior del código se facilita aún más.

3.5.1. Clase **Bone**

Esta es la clase más básica y general de todas, de ahí que sea la super clase de la que todas heredan. **Bone**, contiene todas las características que comparten cualquiera de los huesos que existan en la jerarquía,

tales como:

- Name: nombre del hueso
- Zmax, Zmin : valores máximos y mínimos en Z
- Xmax, Xmin : valores máximos y mínimos en X
- Ymax, Ymin : valores máximos y mínimos en Y

Además, comparte una función con todas sus clases hijas, que se llama Report_glitch. Tal y como es debido, estos atributos y métodos, se consideran como **protégidos** según los modificadores de acceso.

En el constructor de la clase, se puede apreciar que una vez que se crea un objeto tipo *Bone*, se confieren todos los límites de movimiento consignados en los estudios de goniometría, ya que los límites de movimiento se le pasan como parámetros. He aquí un extracto del constructor de esta clase:

```
class Bone:
    """
    Esta clase contiene todos los métodos necesarios para manipular la
    información de cualquier hueso de la jerarquía proveniente de la sección
    HIERARCHY del archivo BVH.
    """
    def __init__(self,
                 Name='GenericBone',
                 Zmin=0,
                 Zmax=0,
                 Xmin=0,
                 Xmax=0,
                 Ymin=0,
                 Ymax=0):
        """
        Se inicializa cada hueso con sus valores por default:
        Name: nombre del hueso
        Valores Goniométricos para este hueso:
        Zmax, Zmin : valores máximos y mínimos en Z
        Xmax, Xmin : valores máximos y mínimos en X
        Ymax, Ymin : valores máximos y mínimos en Y
        """
        self.Name = Name
        self.Zmin = Zmin
        self.Zmax = Zmax
        self.Xmin = Xmin
        self.Xmax = Xmax
```

```
    self.Ymin = Ymin
    self.Ymax = Ymax
```

A esta clase, se le une la función para reportar *glitches* llamada Report_glitch, la cual reporta en un archivo de texto, los errores que se van encontrando, junto con los respectivos *bones* y *frames* del vídeo.

```
def Report_glitch(self, errorMsg='Hay un glitch de movimiento aquí', frame=0):
    """
    Descripción:
    Esta función se encarga de reportar los glitches que se encuentra y en
    qué frame en específico está. Cada glitch se reporta en un archivo de
    texto llamado Glitch_Report

    argumentos:
    errorMsg: string que contiene un mensaje de error
    frame: cuadro donde se produce el glitch
    """
    #Creamos el archivo de reporte
    glitch = open('Glitch_Report', 'a')
    errorString = 'Frame: ' + str(frame) + ' bone: ' + self.Name + ' ' + self.ID
    + ' | ' + errorMsg + '\n'
    #se escribe el string de error y se libera la memoria
    glitch.write(errorString)
    glitch.close()
```

Esta función reporta los *glitches* detectados, al igual que la localización espacial y temporal de cada uno. Cabe destacar, que la función no genera esta información, solamente la recibe como parámetros.

3.5.2. Implementación de subclases

En el diagrama UML mostrado en el apéndice B.2, se puede apreciar que todas las subclases heredan los métodos y atributos de la clase *Bone*. Por lo que esta clase, puede llamarse la clase padre y todos sus atributos se consideran protegidos.

Todas las subclases poseen una estructura similar, en la cual primero se llama al constructor de la clase padre *Bone* para definir los límites de movilidad y después, se llama al constructor de la subclase en cuestión. Una vez llamado el constructor de la subclase, se procede a definir los valores numéricos que permiten localizar al hueso dentro de la jerarquía. He aquí un ejemplo con la clase *Arm*.

```
class Arm(Bone):
    """
    Esta subclase implementa el estudio de goniometría para los brazos en
    el esqueleto del BVH. La jerarquía los llama "Arm".
```

```

"""
def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):
    """
    Se inicializa este hueso con los siguientes parámetros
    ID: identificador del bone. Ej: izquierdo/derecho

    Cada posición del hueso se define con un vector de ángulos de Euler
    (Z, X, Y) los cuales tienen una posición específica dentro del array
    de la sección MOTION del BVH
    Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso
    Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso
    Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso
    """
    self.ID = ID
    self.Zp = Zp
    self.Xp = Xp
    self.Yp = Yp
    #se llama al constructor de la super clase para acceder a todos los atributos
    #de goniometría
    Bone.__init__(self,
                  Name='Brazo',
                  Zmin=-90.000000,
                  Zmax=90.000000,
                  Xmin=-45.000000,
                  Xmax=135.000000,
                  Ymin=-90.000000,
                  Ymax=90.000000)

```

La gran diferencia que existe entre las subclases, es la forma en la manejan su versión de *Goniometry_check*, ya que hay algunos huesos, que presentan una variación de este método.

Las cámaras de captura, presentan un problema a la hora de grabar ciertos tipos de movimientos de personas muy flexibles. A veces, los ejes de las rodillas tienden a perder la orientación del eje de rotación Y y eso hace que las cámaras terminen reportando los movimientos del eje Z en el eje X, y viceversa. Esto resulta muy inconveniente, ya que si se presenta un *glitch* en ese instante, dependiendo de qué tan grave sea la pérdida de orientación del eje, puede ser que el *glitch* pase inadvertido.

Con el objetivo de tomar en cuenta todos los casos posibles, se incluyó una pequeña variación en la función *Goniometry_check*, en la cual, mediante un simple función lógica, se decide si se aplican o no las pruebas goniométricas. Si *Rotacion_ejes* y *rodilla_flex* son verdaderas, las pruebas no se aplicarán, ya que puede que los movimientos registrados por las cámaras sean correctos, pero los ángulos con que los registra están reportados en los ejes que no corresponden. He aquí el código de esta pequeña función lógica:

```
#Variables para probar si hubo rotacion de ejes y el esqueleto esta agachado
rodilla_flex = Xeluer > 13.0 or Xeluer < -15.0
y_rot = Yeuler > 20.0 or Yeuler < -20.0
z_rot = Zeuler > 40.0 or Zeuler < -40.0

Rotacion_ejes = y_rot or z_rot
```

La función más elaborada de Goniometry_check está en la subclase Leg, y muestra a continuación. Note, que se aplica la función lógica anterior primero, y después se aplican las pruebas goniométricas de acuerdo al diagrama de flujo del apéndice A.3.

```
def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que un hueso posee en un frame determinado, con el valor de los límites goniométricos de ese hueso en particular. Si algún ángulo de Euler excede los límites del movimiento humano, se reportará un glitch en ese frame y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler para cada hueso en un frame dado. El orden de cada hueso viene dado por la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    #Exempt es una variable que se activa cuando detecta problemas de rotacion
    #de ejes Z y Y en las rodillas
    Exempt = False
    ErrorMsg = ' existen glitches de '

    #Variables para probar si hubo rotación de ejes y el esqueleto está agachado
    rodilla_flex = Xeluer > 13.0 or Xeluer < -15.0
    y_rot = Yeuler > 20.0 or Yeuler < -20.0
    z_rot = Zeuler > 40.0 or Zeuler < -40.0

    Rotacion_ejes = y_rot or z_rot
```

```

if rodilla_flex and Rotacion_ejes:
    Exempt = True

if Exempt:
    #Existen dos pruebas goniométricas distintas de acuerdo al nivel de flexión de las
    #rodillas. En el caso de que las rodillas tengan un ángulo de flexión mayor a 45º o
    #exista una rotación de los ejes Z y Y, debemos incrementar los límites de movilidad.
    #en Z y Y. Esto debido al comportamiento de los huesos en el BVH, los cuales rotan
    #los ejes Y y Z para representar movimientos de un esqueleto agachado.

    #Esto ocurre debido a la pérdida de orientación del hueso, por parte de las cámaras
    #en los ejes Z y Y.

    #probamos límites nuevos en Z
    if Zeuler < -160.000000:
        #MOTION[self.Zp] no se le aplica restricción en Z
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Z- | '

    if Zeuler > 160.000000:
        #MOTION[self.Zp] no se le aplica restricción en Z
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Z+ | '

    #aquí probamos nuevos límites en X
    if Xeluer < -150.000000:
        #MOTION[self.Xp] no se le aplica restricción en X
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en X- | '

    if Xeluer > 150.000000:
        #MOTION[self.Xp] no se le aplica restricción en X
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en X+ | '

    #aquí probamos nuevos límites en Y
    if Yeuler < -105.000000:
        #MOTION[self.Yp] no se le aplica restricción en Y
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Y- | '

    if Yeuler > 105.000000:
        #MOTION[self.Yp] no se le aplica restricción en Y

```

```
glitch = True
ErrorMsg += 'pérdida de orientación de los sensores en Y+ | '

else:
    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'torsión | '

    if Zeuler > self.Zmax:
        MOTION[self.Zp] = self.Zmax
        glitch = True
        ErrorMsg += 'torsión | '

    #aquí probamos límites en X
    if Xeluer < self.Xmin:
        MOTION[self.Xp] = self.Xmin
        glitch = True
        ErrorMsg += 'extension | '
    if Xeluer > self.Xmax:
        MOTION[self.Xp] = self.Xmax
        glitch = True
        ErrorMsg += 'flexion | '

    #aquí probamos límites en Y
    if Yeuler < self.Ymin:
        MOTION[self.Yp] = self.Ymin
        glitch = True
        ErrorMsg += 'rotacion interna | '
    if Yeuler > self.Ymax:
        MOTION[self.Yp] = self.Ymax
        glitch = True
        ErrorMsg += 'rotacion externa | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```

3.6. Recursos en línea

La implementación final del programa está expuesta en los apéndices A y C. Si se quiere bajar y estudiar el código fuente, así como el material del apoyo, acceda a el repositorio en **GitHub**:

<https://github.com/mario23285/ProyectoElectrico.git>

Capítulo 4

Validación

En esta sección del proyecto, se presentan una variedad de pruebas cuantitativas que reflejan las capacidades del algoritmo para la detección, localización y corrección de *glitches* en un archivo *BVH*. En general, se presentarán dos tipos de pruebas:

- Pruebas sintéticas
- Pruebas de laboratorio

La diferencia entre ambas, radica en que las pruebas sintéticas se realizaron con archivos *BVH* en circunstancias muy controladas y las pruebas de laboratorio, fueron hechas con vídeos normales de producción. La justificación de separar las pruebas en dos grupos, viene dada por la necesidad de ilustrar distintas capacidades que posee el algoritmo y, al mismo tiempo, presentar las limitaciones que éste posee.

4.1. Introducción

Las pruebas propuestas para el presente proyecto se centraron en las principales cualidades que debe poseer el algoritmo de corrección para poder corregir un archivo de *MoCap* de manera óptima:

- Detección
- Localización
- Corrección

Retomando estos conceptos del marco teórico, la **detección** se refiere a la capacidad que tiene el algoritmo de identificar que ocurrió un error en la captura de *MoCap* y reportarlo como tal.

La **Localización** en nuestro caso, se divide en dos conceptos, la localización espacial, la cual se refiere a la capacidad de asociar un evento detectado (un *glitch*) a una articulación del esqueleto en particular (sea el codo, la rodilla, etc.) y la localización temporal, la cual consiste en la capacidad de asociar un *glitch* con un *frame* en particular.

Finalmente, la corrección tiene que ver con la capacidad del programa de estimar de los valores correctos que debería tener el archivo de *MoCap* para generar un modelo corregido.

Las pruebas sintéticas, se desarrollaron para poder probar adecuadamente las capacidades de corrección del programa. El concepto detrás de estas pruebas es muy simple: existen dos versiones de un mismo archivo BVH, una sin errores (considerada el *Ground Truth*) y otra con errores en *frames* específicos. La idea de la prueba en sí, es darle el BVH con errores al algoritmo de corrección y después comparar la salida del algoritmo con el *Ground Truth* y ver qué tanto se parecen entre sí.

4.2. Prueba sintética 1

La prueba sintética 1, consistió en tomar un archivo de *MoCap* con 310 *frames* grabado a 25fps y crear dos versiones de ese archivo. Como ya se mencionó, existe la versión *Ground Truth* y la versión con errores, la cual tiene los *glitches* listados en la tabla 4.1. Note que dicha tabla incluye una columna en la que se marcaron los errores que fueron detectados con éxito por el algoritmo después de su ejecución.

Cabe mencionar, que los errores fueron incorporados de manera paulatina. Es decir, se comenzó sobre pasando de 1° en 1° los límites permitidos en cada articulación. De tal manera, que el algoritmo tuvo que lidiar con toda clase de errores, desde sutiles hasta pronunciados (90° o más). Con esto, se puede evaluar la sensibilidad del algoritmo mediante las pruebas de *Precision*, *Recall* y *F1-Score*, las cuales se encuentran en la sección 4.4.

4.2.1. Detección y localización

Glitches	frame inicial	frame final	<i>Glitch</i> detectado
Rotación de tobillo derecho	15	20	Sí
Torsión y rotación de rodilla derecha	20	39	Sí
Rotación de tobillo izquierdo	42	54	Sí
Torsión de rodilla izquierda	120	132	Sí
Abducción de hombro izquierdo	200	210	Sí
Abducción de hombro izquierdo (rotando hombro)	261	296	Sí

Tabla 4.1: *Glitches* agregados a la prueba sintética 1. Cada uno de estos errores fue introducido a mano en el archivo BVH para ser corregido posteriormente.

La tabla 4.1 muestra la localización de cada *glitch* en el archivo BVH dañado. Continuando con la prueba, se procedió a corregir el BVH dañado por medio del programa y éste publicó el reporte de eventos que se muestra a continuación. Cabe mencionar, que el informe aquí mostrado es un extracto, ya que el original contiene más información por *frame* y es mucho más extenso. El propósito de este informe, es ilustrar cómo el algoritmo es capaz de detectar y localizar los errores.

```
---- Version simplificada de la salida del reporte de Glitches ----
2017/06/17 19:01 >> Prueba sintetica 1
```

```

Frame: 15 bone: Tobillo Derecho | existen glitches de rotacion externa
al
Frame: 20 bone: Tobillo Derecho | existen glitches de rotacion externa
---
Frame: 20 bone: Rodilla Derecha | existen glitches de torsion y rotacion interna
al
Frame: 39 bone: Rodilla Derecha | existen glitches de torsion y rotacion interna
---
Frame: 40 bone: Tobillo Izquierdo | existen glitches de rotacion interna y torsion
externa
al
Frame: 57 bone: Tobillo Izquierdo | existen glitches de rotacion interna y torsion
externa
---
Frame: 120 bone: Rodilla Izquierda | existen glitches de torsion | rotacion interna
al
Frame: 132 bone: Rodilla Izquierda | existen glitches de torsion | rotacion interna
---
Frame: 200 bone: Brazo Izquierdo | existen glitches de abduccion horizontal
al
Frame: 210 bone: Brazo Izquierdo | existen glitches de abduccion horizontal
---
Frame: 261 bone: Brazo Izquierdo | existen glitches de abduccion frontal
al
Frame: 296 bone: Brazo Izquierdo | existen glitches de abduccion frontal
---

```

Al concluir la ejecución del programa, se logró detectar en un 100 % todos los errores del *MoCap* dañado y cada error fue localizado con éxito a nivel espacial y temporal.

4.2.2. Corrección

En lo que respecta al nivel de corrección, se necesita algún criterio que logre comparar el archivo generado por el algoritmo con un archivo perfecto. Inicialmente, se había pensado en un criterio basado en el Error Cuadrático Medio por *frame*, pero aplicar este criterio para comparar el archivo perfecto con el dañado, no da la cantidad de información suficiente para probar qué tan precisa fue la corrección hecha por el programa.

Ante esta situación, se planteó hacer un experimento sencillo, pero efectivo. Primero se obtuvo el error cuadrático medio por *frame* entre el *Ground Truth* y el archivo dañado. Lo que resulta de esto, es que la curva de error es cero en los *frames* que son iguales entre ambos archivos y distinta de cero en los *frames* que son diferentes, es decir, se generó una curva de error que indica dónde están los *glitches*.

El segundo paso, consistió en reparar el archivo dañado usando el algoritmo y obtener un *bvh* corregido. Finalmente, el tercer paso fue obtener la curva del error cuadrático medio por *frame* entre el archivo *Ground Truth* y el *bvh* corregido. El resultado de este experimento se encuentra en la figura 4.1. En la ecuación 4.1 se puede apreciar la fórmula del ECM que se usó para estos experimentos.

$$ECM = \frac{1}{n} \cdot \sum_{i=1}^n (\bar{O}_i - \bar{G}_i)^2 \quad (4.1)$$

Donde \bar{O}_i representa el vector de MOTION *Ground Truth* para el i-ésimo *frame*, y \bar{G}_i , representa el vector de MOTION dañado para el mismo *frame*. Luego, se repitió la misma fórmula, sustituyendo el vector \bar{G}_i , por el vector \bar{C}_i , el cual representa el vector de MOTION corregido para el i-ésimo *frame*.

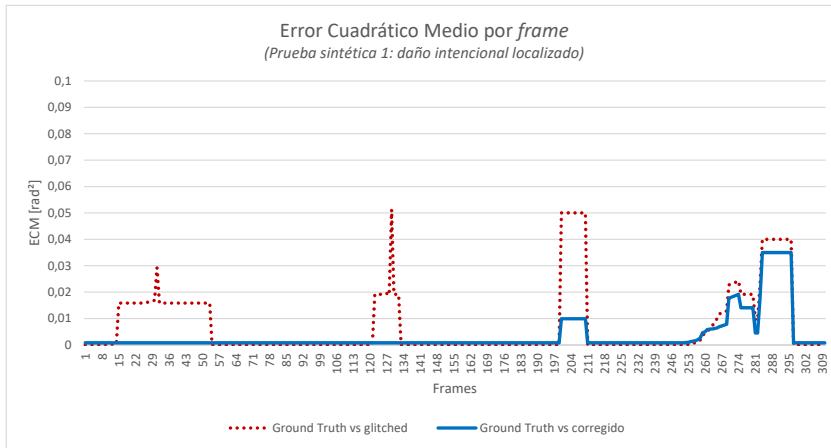


Figura 4.1: Prueba sintética 1: la curva roja, representa el ECM por *frame* entre el *Ground Truth* y el archivo dañado (*Glitches*). La curva azul representa la diferencia entre el *Ground Truth* y el corregido. Note que entre más cercana a cero es la curva azul, mejor es la corrección.

El resultado de este experimento muestra de una forma gráfica qué tan acertado fue el programa a la hora de corregir los errores de *MoCap*. La curva roja, representa la diferencia que existe entre el *Ground Truth* y el archivo dañado. Si lo que se quiere es corregir el archivo dañado, hay que reducir la curva roja a cero lo más posible, de manera que la diferencia entre ambos sea mínima.

La curva azul, muestra la efectividad del proceso de corrección a lo largo de cada *frame* de la escena. Se puede apreciar que los *glitches* de los cuadros 15 al 54, se corrigieron de manera perfecta, mientras que los errores del cuadro 120 al 132 (Torsión de rodilla izquierda) se mejoraron sustancialmente.

Como nota final de este experimento, vale la pena señalar que los *glitches* de la última sección de la escena (cuadros 200-296), están clasificados como errores de abducción de hombro. La articulación del hombro es, por mucho, la más complicada del cuerpo humano. Posee 6 grados de libertad y diferentes límites de movilidad dependiendo del nivel de abducción del hombro [Kapandji and Lacomba, 2006].

Aunque el algoritmo es capaz de arreglar estos errores con cierta precisión, siempre es difícil calcular la posición correcta del brazo del esqueleto, si se tiene un error de hombro que involucre rotación y abducción al mismo tiempo.

4.3. Prueba sintética 2

La segunda prueba sintética es más interesante que la prueba anterior, ya que ésta se llevó a cabo con una animación hecha en el laboratorio. El concepto aplicado de prueba sintética fue el mismo, sin embargo, la particularidad de esta prueba es que contiene un *glitch* que no pudo ser detectado debido a las limitaciones de la información contenida en el archivo BVH dañado.

4.3.1. Detección y localización

Al igual que en la prueba 1, los *glitches* se introdujeron de manera paulatina, comenzando con errores sutiles, e incrementando los ángulos a medida que transcurre el intervalo de *frames* de un *glitch* en particular. La razón por la que se hizo de esta forma, es porque el interés de esta prueba no es solo corregir los errores, sino también, evaluar cuántos Falsos Positivos y Falsos Negativos obtendrá el algoritmo.

Glitches	frame inicial	frame final	<i>Glitch</i> detectado
Rotación de tobillo derecho	1	11	Sí
Torsión de rodilla derecha	12	22	Sí
Torsión de rodilla derecha	52	62	Sí
Torsión de rodilla izquierda	63	73	Sí
Rotación de tobillo izquierdo	74	84	Sí
Torsión de rodilla derecha	241	264	No
Rotación de tobillo derecho	435	445	Sí

Tabla 4.2: Listado de *glitches* para el segundo experimento. Note que esta vez se detectaron 6 de los 7 errores.

Al igual que el experimento anterior, se procedió a dañar un archivo BVH perfecto con el listado de errores mostrado en la tabla 4.2 y luego, se le aplicó el programa a dicho archivo para obtener el reporte de *glitches*.

```
----Version simplificada de la salida del reporte de Glitches----
2017/06/17 22:37 >> Prueba sintetica 2

Frame: 1 bone: Tobillo Derecho | existen glitches de rotacion interna
al
Frame: 12 bone: Tobillo Derecho | existen glitches de rotacion interna
---
Frame: 13 bone: Rodilla Derecha | existen glitches de torsion |
al
Frame: 61 bone: Rodilla Derecha | existen glitches de torsion |
---
Frame: 62 bone: Rodilla Izquierda | existen glitches de torsion |
al
Frame: 71 bone: Rodilla Izquierda | existen glitches de torsion |
--
```

```

Frame: 72 bone: Tobillo Izquierdo | existen glitches de rotacion interna |
al
Frame: 81 bone: Tobillo Izquierdo | existen glitches de rotacion interna |
-----
Frame: 314 bone: Cadera Derecha | existen glitches de perdida de orientacion de los
sensores en X-
Frame: 317 bone: Cadera Derecha | existen glitches de perdida de orientacion de los
sensores en X-
-----
Frame: 435 bone: Tobillo Derecho | existen glitches de rotacion externa |
al
Frame: 445 bone: Tobillo Derecho | existen glitches de rotacion externa |

```

Una de las cosas que salta a la vista, es que todos los errores de torsión de rodilla, se detectaron sin problemas, con excepción del error de torsión de rodilla derecha que va del *frame* 241 al 264 (véase figura 4.2). Existe una razón para este resultado.

Hay ocasiones en las que las cámaras de Captura de Movimiento, no son capaces de calcular bien el ángulo de rotación de un hueso, ni tampoco la posición de éste en el espacio, y como resultado, pierden la orientación del mismo, haciendo que el sistema de coordenadas (Z, X, Y) rote y se intercambien los ejes entre sí. Lo cual quiere decir, que los movimientos que se supone van en el eje Z, terminan reportados en el eje X.

Otra forma de decir lo anterior es que, a veces, los movimientos del esqueleto se ven correctos en el vídeo, pero internamente los ángulos que se usaron para construir las matrices de movimiento de OpenGL, no son correctos según los estudios de goniometría. Esto ocurre cuando los cálculos de posición de los sensores del traje de *MoCap*, no convergen a un valor confiable.

Normalmente, esto es difícil de apreciar en los vídeos, ya que los huesos de un archivo BVH tienen, todos, la misma forma tetraédrica. No poseen una parte frontal o lateral y todos tienen 3 grados de libertad de movimiento sin importar donde estén en el cuerpo. Esto hace que se puedan tener movimientos que se ven normales, pero que internamente (en los vectores de movimiento) tienen ángulos no válidos de rotación.

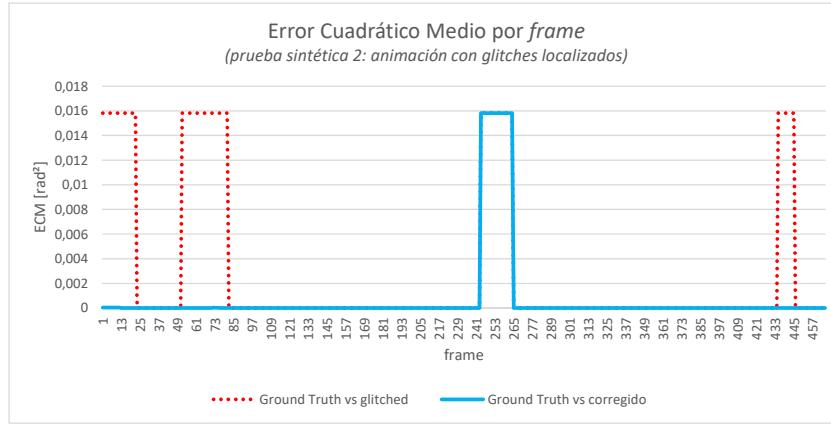


Figura 4.2: Prueba sintética 2. Las curvas de ECM muestran la efectividad de la corrección hecha. Casi todos los errores fueron corregidos en un 100 %.

4.3.2. Corrección

El hecho de que los ejes de rotación se puedan perder ocasionalmente para un hueso, aumenta la probabilidad de que se den *glitches*, pero esto ocurre bajo circunstancias especiales. En la mayoría de experimentos hechos en el laboratorio, se observó que las personas muy flexibles, cuando hacen movimientos poco comunes, tienen una alta probabilidad de confundir a los sensores y hacer que las cámaras pierdan la orientación de las rodillas. Un experimento hecho con una persona muy flexible, puede verse en la figura 4.3. Como se puede apreciar, las cámaras llegan a confundirse al punto de reportar ángulos de 120° o más para una rodilla que no puede moverse en esa dirección.

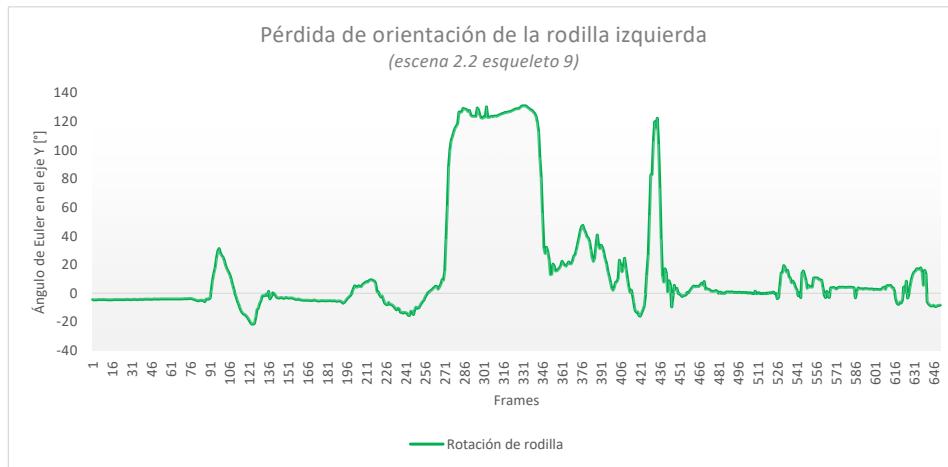


Figura 4.3: Pérdida de orientación de la rodilla a lo largo del tiempo. El ángulo anatómicamente correcto para la rotación de rodilla debería ser siempre 0°, porque ésta no rota sobre sí misma.

Siendo este el caso, hay *glitches* que no se pueden detectar, localizar ni corregir, cuando se pierde la orientación de los ejes de rotación de un hueso, ya que al cambiar los ejes de posición, no tiene sentido aplicar los estudios de goniometría tradicionales. En estos casos, el programa se ve forzado a realizar pruebas de goniometría laxas. Esto quiere decir, que hay límites que no se aplican en los ejes Z, X o Y, dándole más libertad de movimiento al esqueleto en circunstancias poco usuales.

En estos casos, la información proporcionada por los estudios de goniometría, no es suficiente para corregir el archivo de captura de movimiento. Es por esto, que en la sección de recomendaciones, se habla de utilizar una fuente de información secundaria, a parte de los estudios de goniometría, para corregir un *MoCap* dañado.

Una sugerencia a tener en cuenta, es utilizar los archivos CSV generados por las cámaras infrarrojas. Estos datos, contienen información sobre la posición de cada uno de los sensores del traje de *MoCap* en un *frame* dado y pueden ser de utilidad para determinar la posición de un hueso cuya orientación se haya alterado.

4.4. **Precision, Recall y F1-Score**

Como paso final de la validación, es importante incluir las pruebas comúnmente denominadas *Precision*, *Recall* y *F1-Score*, esto con el objetivo de validar el proceso de RP del algoritmo y su sensibilidad; el proceso de RP en cuestión, es un proceso de clasificación mediante el cual se separan los *frames* defectuosos, de los no defectuosos. La definición de dichas pruebas parte de los siguientes términos [Albon, 2016]:

- **Positive (P)**: se refiere a una observación positiva. Ejemplo: *frame* defectuoso.
- **Negative (N)**: observación negativa. Ejemplo: *frame* no defectuoso.
- **True Positive (TP)**: la observación es positiva, y es marcada por el algoritmo como positiva.
- **False Negative (FN)**: la observación es positiva, sin embargo, el algoritmo la marca como negativa.
- **True Negative (TN)**: la observación es negativa y es marcada por el algoritmo como negativa.
- **False Positive (FP)**: la observación es falsa, pero se marca como positiva.

Precision : proporción de las predicciones positivas que son correctas. La precisión es una medida, de cuántas predicciones positivas fueron, de hecho, observaciones positivas.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4.2)$$

Recall : también llamada *True Positive Rate* (TPR), es una proporción de todas las observaciones positivas que se predicen correctamente. TPR, es una medida de cuán bueno es un modelo para predecir casos positivos.

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P} \quad (4.3)$$

F1-Score : es la media armónica entre entre *Recall* y *Precision*. *F1-Score* es un promedio de *Precision* y *Recall*, sin embargo, se utiliza la media armónica porque es la manera apropiada de promediar las proporciones (*ratios*).

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (4.4)$$

Partiendo de estas definiciones, se calcularon cada uno de estos indicadores para las pruebas sintéticas 1 y 2, mostradas anteriormente. Los resultados se pueden apreciar en la tabla 4.3.

Prueba	Precision	Recall	F1-Score
Prueba sintética 1	0.949	1.00	0.974
Prueba sintética 2	0.954	0.922	0.938

Tabla 4.3: *Precision, Recall y F1-Score por prueba*.

Note como *Recall* baja a 0,922 para la segunda prueba sintética, dada la cantidad de Falsos Negativos que obtuvo la prueba. En esta prueba, existe un *glitch* de torsión de rodilla derecha que no fue corregido, pero sí detectado. Este error se dio por la rotación de ejes anteriormente expuesta, la cual inhibe las pruebas goniométricas para la rodilla en ese lapso de 23 *frames*. Tales situaciones, no las puede corregir el algoritmo, por la falta de información para determinar la posición correcta de la rodilla.

Los Falsos Negativos, por lo general, se manifiestan cuando los errores del *MoCap* en un *frame* específico, son considerados muy sutiles. En estos casos el SRP marca el *frame* como no defectuoso. En lo que respecta a los Falsos Positivos, estos surgen cuando existe mucha diferencia entre dos *frames* consecutivos, en los cuales, uno tiene un *glitch* pronunciado y el otro *frame* no. Es en estos casos, donde el algoritmo marca estos *frames* como defectuosos y procede a arreglarlos.

Otro caso que puede favorecer la aparición de FN o FP, puede ser el traslape de dos *glitches* pertenecientes a distintas articulaciones, en el mismo *frame*. Esto hace, que la segmentación falle por un pequeño margen de 1 ó 2 *frames*.

En general, el algoritmo goza de una buena precisión a la hora de clasificar los *glitches*, minimizando la cantidad de Falsos Negativos y Falsos Positivos por medio del algoritmo de umbralización. Éste, es capaz de aplicar la variedad de pruebas necesarias, para determinar si un *frame* está defectuoso o no, dando paso a una segmentación lo suficientemente precisa, para corregir la mayor cantidad de *frames* posibles.

4.5. Pruebas de laboratorio

En esta sección de validación, se analizaron imágenes de un archivo corregido por el algoritmo. Se utilizó un video tomado en el laboratorio bajo las mismas circunstancias que en una producción de Animación Digital normal y luego, se procedió a usar el algoritmo para corregir los errores de la captura.

La siguiente sección habla de la escena filmada y los errores corregidos. Se utilizó el programa de animación **Blender** para mostrar una comparación lado a lado, entre el esqueleto original con *glitches* y el esqueleto corregido por el algoritmo. Como ayuda visual, el esqueleto corregido siempre se muestra en color amarillo, mientras que el original (dañado) se muestra en color gris.

4.5.1. escena 2.2 esqueleto 9

Esta escena se filmó en el laboratorio con la ayuda de un practicante de Kung-Fu. Al ser una persona muy flexible, se esperaban errores en los brazos y piernas, lo cual sucedió. En la figura siguiente, se muestra el primer *glitch* encontrado en la captura.

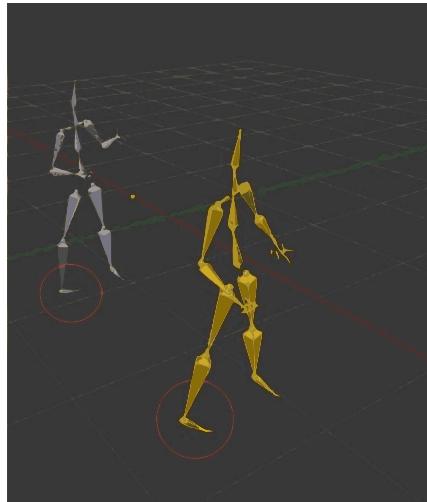


Figura 4.4: Primer *glitch* en el frame 140. Note la rotación interna del tobillo y cómo fue corregida.

En los segundos siguientes, se detectó un *glitch* complejo en el codo. Dicho evento, incluye errores de supinación, movimientos no válidos en el eje Z (debido a la pérdida de orientación de los ejes), y consecuentemente, la información del eje X no es válida tampoco, por un período de tiempo largo (*frames*: 274 al 404). La corrección hecha por el algoritmo se puede apreciar en la figura 4.5

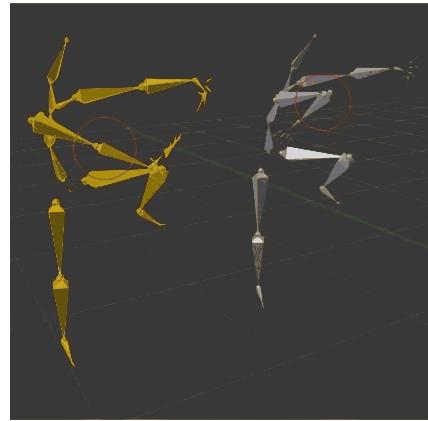


Figura 4.5: Primer *glitch* complejo en el codo: *frames*:274-404. El algoritmo usa las referencias de goniometría para determinar la posición correcta del codo y así extenderlo en la dirección correcta.

El problema con este error del codo, fue que se prolongó por mucho tiempo y este tipo de errores son difíciles de corregir. No obstante, la animación final resultó ser mejor que la original. En la figura siguiente, se puede apreciar el *frame* 394 en el que la posición del brazo fue calculada correctamente aunque existía un *glitch* previo.



Figura 4.6: *Frame* 394: la posición del codo fue calculada con éxito.

La razón por la cual el programa pudo calcular correctamente la posición del codo y determinar que éste debía estar extendido, es porque la mayoría de los movimientos de flexión que estaban en el vídeo original, fueron reportados en el eje Z. El codo no tiene movimiento en este eje, por lo que el programa fácilmente determinó que el ángulo correcto en Z era 0° , lo cual corresponde a un brazo extendido.



Figura 4.7: La posición del codo se pudo arreglar con éxito debido al error encontrado en el eje Z en este cuadro del video.

Finalmente, otro error importante que pudo ser corregido corresponde a una flexión lateral de la espina dorsal que no era válida. La figura 4.8 muestra el error corregido con éxito.

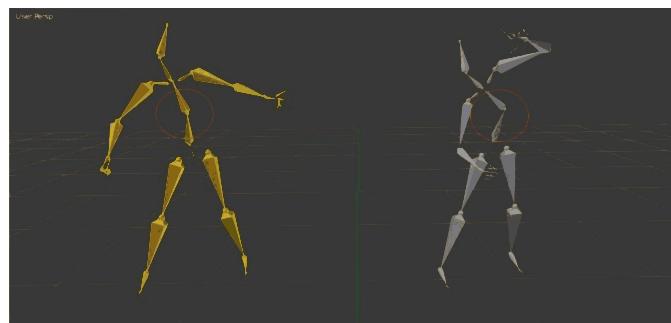


Figura 4.8: Error en la columna vertebral corregido en el cuadro 416.

Las pruebas mostradas en estos experimentos, ilustran de manera cuantitativa, qué tan precisa resulta ser la corrección hecha por el *software*. En ausencia de un criterio que permita comparar ángulos de Euler de manera directa, se tuvo que desarrollar una metodología que permitiera demostrar la efectividad de los resultados. Es por esto que se implementaron las pruebas sintéticas con curvas de ECM para poder tener un punto de comparación entre una animación que es correcta y otra que no lo es.

En general, las curvas de ECM muestran que el algoritmo mejora de manera sustancial una captura dañada, demostrando que éste tiene un buen desempeño en la corrección de animaciones, facilitando así el proceso de producción para animadores profesionales y gente afín a la producción.

Capítulo 5

Conclusiones y recomendaciones

La conclusión de este proyecto, dejó ver los desafíos constantes que existen en el campo de la Animación Digital con *Motion Capture*. Actualmente, los sistemas de captura de movimiento se encuentran en un proceso de mejoramiento constante, lo cual se puede ver en la gran cantidad de trabajos de investigación, orientados a mejorar los diseños de *hardware* de las cámaras y sensores para obtener animaciones de mejor calidad.

A la fecha de finalización del presente proyecto, no deja de sorprender el hecho de que no se encontraron trabajos previos en el área de *software*, que estuvieran orientados a resolver el problema de arreglar los *glitches* de *MoCap* por medio de un algoritmo.

5.1. Conclusiones

Dada la situación, de que no se encontraron trabajos previos en el área de *software* que resolvieran este problema, es válido decir que el programa desarrollado en este proyecto, es el aporte novedoso que se esperaba desarrollar. Existen muchos retos en Animación Digital y muchas formas de abordarlos. En el caso particular de este proyecto, se escogió una idea poco explorada y se desarrolló una solución lo más completa posible, en un período de cuatro meses.

Algunos puntos importantes que destacar al finalizar este proyecto son:

1. Los archivos BVH no están tan estandarizados como se esperaba. Existen algunas versiones que alteran ligeramente el orden de las jerarquías o los vectores de ángulos de Euler, lo cual fuerza al diseñador a escoger un formato particular para tratar los datos.
2. En los casos en los que se pierde la orientación de los ejes Z, X o Y en una captura, es más difícil arreglar los *glitches* de *MoCap* debido a que la información disponible no es suficiente para calcular la posición precisa de cada hueso del esqueleto humano.
3. La goniometría resultó ser un excelente primer paso para la corrección de errores de *MoCap*, al deshacerse eficientemente de los movimientos no válidos en una animación.

4. La taxonomía basada en fisiología articular y goniometría representó una valiosa herramienta, al permitir no sólo la localización de los errores de forma precisa, sino también facilitar la corrección de los mismos por medio de los estudios de movilidad. Todo esto fue posible, sin necesidad de inventar nombres para los movimientos o complicar las cosas usando otras clasificaciones.
5. El problema de utilizar interpolación para generar *frames* en un archivo BVH, es cómo sincronizar los movimientos entre las matrices de traslación y rotación de los huesos del esqueleto, para hacer lucir el movimiento como algo natural. Tal y como lo señala Rick Parent [Parent, 2012], hacer lucir un movimiento natural requiere de mucho talento por parte de un animador, no solo interpolación.

Todas estas conclusiones, se derivan del cumplimiento de los objetivos originales que se planificaron al inicio del proyecto en cuestión.

Los objetivos propuestos se lograron concretar a tiempo. Estableciendo conclusiones por cada uno, el primero consistió en la investigación bibliográfica que se debe llevar a cabo al inicio de todo proyecto. Aquí no solo se investigó la forma en la que operan los programas de Animación Digital más usados, sino también la forma en la que operan las cámaras de captura de movimiento y los formatos utilizados por éstas para representar los datos, como los archivos BVH.

Después de realizar una extensa búsqueda, se concluyó que no existían proyectos que abordaran el problema de corregir capturas de movimiento vía *software*. La gran mayoría de trabajos de investigación que se encontraron, están enfocados en el mejoramiento de sensores y cámaras de captura de movimiento. Es por esto, que el principal aporte de este proyecto es llenar ese vacío.

El segundo objetivo resultó ser más complicado. Originalmente, se propuso utilizar bibliotecas de OpenGL en C/C++ para la implementación del proyecto, ya que éstas contienen funciones incluidas para todo lo relacionado con la animación 3D. También se propuso usar interpolación 3D para resolver el problema de los errores de *Motion Capture*, sin embargo, a la mitad de proyecto, se tuvo desechar todas estas ideas y se optó por un método que utilizara goniometría en vez de interpolación y se cambió el lenguaje C++ por Python.

La idea de desechar la interpolación como método para arreglar *glitches*, surgió debido a las complicaciones que presentó este método desde el punto de vista de la animación digital. Internamente los programas de animación como Maya o Blender, dividen el movimiento en espacios 3D en matrices de rotación y traslación. El verdadero desafío en estas circunstancias, es cómo hacer para sincronizar las matrices de traslación con las de rotación para hacer que un movimiento luzca natural.

La interpolación por sí sola no puede llevar a cabo esta sincronización y, por eso, hace lucir a los *glitches* aún más extraños de lo que ya son. Siendo este el caso, se desecharon esta idea después de varias pruebas de concepto hechas en Blender.

En lo que respecta al cambio de lenguaje de C++ a Python, éste obedeció a que las bibliotecas de OpenGL que originalmente se iban a usar para interpolación, se desecharon. Se hizo el cambio a Python, debido a que éste lenguaje es altamente compatible con Maya y Blender, dos programas de animación muy utilizados tanto en el PRIS-lab, como en producciones de alto nivel.

Finalmente, la implementación fue hecha en Python utilizando el paradigma de Programación Orientada a Objetos, tal y como se explicó en el capítulo 3. Se escogió este paradigma debido a que es mucho más fácil crear nuevos módulos e integrarlos, con miras a trabajo futuro.

Este problema probó ser muy desafiante y requiere de más recursos para resolverlo mejor. Se han encontrado una variedad de métodos de estimación de posición que podría ser útil implementar, es por esto que el desarrollo del programa debe verse de manera modular, para ir agregando nuevos elementos en el futuro.

Es por esto que se concluyó, que es mejor usar recursos como programación orientada a objetos y lenguajes altamente soportados, como Python, para implementar más módulos más adelante, ya que la solución a este problema de *Motion Capture* puede terminar siendo una combinación de varias herramientas teóricas.

Un aspecto de la validación, que vale la pena incluir en estas conclusiones, es que no hay criterios establecidos para comparar ángulos entre dos esqueletos pertenecientes al mismo *frame* de un *MoCap*. Se tuvo que lidiar con este hecho comparando dos variedades de curvas ECM, sin embargo, este criterio podría no ser el más óptimo, ya que existen ángulos que aunque sean numéricamente distintos, a nivel posicional son lo mismo (e.g. 0° y 360°). Con el ECM, se suman diferencias que en realidad no existen.

En lo que respecta al último objetivo, el artículo aún se encuentra en escritura pero estará listo a finales de julio, para su posterior envío y revisión para CONESCAPAN en su siguiente edición.

5.2. Recomendaciones

1. En el caso de que existan errores que no puedan ser corregidos con goniometría, se podrían utilizar los archivos CSV generados por las cámaras como una segunda fuente de información. Estos archivos contienen información sobre la posición de cada sensor encontrado en el traje que se usa para *Motion Capture*, además de un índice de confiabilidad por cada cálculo realizado. Se podría usar como un control secundario para determinar la posición de un hueso con los ejes rotados.
2. Otra aproximación al problema, sería tratar de corregir un *glitch* a partir de una estimación de la posición correcta que debería tener el hueso afectado. La forma de llevar a cabo esta estimación podría ser utilizando *Sequential Pose Estimation* tal y como lo plantean Drews *et al* en su artículo *Sequential pose estimation using linearized rotation matrices* [Drews et al., 2013]. Este método usa una forma interesante de filtros de Kalman extendidos que podrían ser de utilidad en este caso.
3. Como trabajo futuro, es recomendable revisar el diseño del programa con miras a hacerlo más eficiente y procurar que el diseño mantenga la facilidad de mantenimiento para futuros desarrollos.
4. Tratar de implementar funciones que trabajen con una variedad mayor de formatos BVH en el futuro, para evitar limitarse a sólo una versión del formato.

Apéndice A

Programa principal y su diagrama de flujo

A.1. Código fuente del programa **BVHTuneUP**

```
# -*- coding: utf-8 -*-

"""
UNIVERSIDAD DE COSTA RICA                                     Escuela de Ingeniería Eléctrica

IE0499 / Proyecto Eléctrico
Mario Alberto Castresana Avendaño
A41267

Programa: BVH_TuneUp
-----


archivo: BVH_TuneUp.py
descripción:
Este es el programa principal. Su función consiste en reparar un archivo BVH
que tenga glitches de MoCap. Para tal efecto, se vale de las estructuras de
datos contenidas en la sección HIERARCHY del archivo BVH y un algoritmo que
compara estudios de goniometría con los movimientos descritos en los vectores
de MOTION del archivo de MoCap.

La documentación del algoritmo se puede encontrar en el github

https://github.com/mario23285/ProyectoElectrico.git

"""

#Lista de clases y módulos a importar

#módulos de sistema
```

```

import sys
import csv
import re

#módulos de la jerarquía de huesos
from Foot import Foot
from Leg import Leg
from Arm import Arm
from ForeArm import ForeArm
from UpLeg import UpLeg
from Wrist import Wrist
from Spine import Spine

#-----ESTRUCTURAS DE DATOS Y OBJETOS-----
#Creación de la jerarquía de Bones del MoCap
#cada objeto se inicializa con un ID (nombre identificador) y las coordenadas
#dentro de la sección MOTION

#miembros inferiores
LeftUpLeg = UpLeg('Izquierda', 132, 133, 134)
RightUpLeg = UpLeg('Derecha', 144, 145, 146)

LeftLeg = Leg('Izquierda', 135, 136, 137)
RightLeg = Leg('Derecha', 147, 148, 149)

Leftfoot = Foot('Izquierdo', 138, 139, 140)
Rightfoot = Foot('Derecho', 150, 151, 152)

#tronco superior
LeftArm = Arm('Izquierdo', 21, 22, 23)
RightArm = Arm('Derecho', 78, 79, 80)
LeftForeArm = ForeArm('Izquierdo', 24, 25, 26)
RightForeArm = ForeArm('Derecho', 81, 82, 83)
LeftHand = Wrist('Izquierda', 27, 28, 29)
RightHand = Wrist('Derecha', 84, 85, 86)

Spine1 = Spine('lumbar-toracica', 9, 10, 11)
Neck = Spine('Cervical', 12, 13, 14)

#-----FIN DE ESTRUCTURAS DE DATOS Y OBJETOS-----

#Archivos de entrada (BVHfile) y salida (outputBVH)

```

```
BVHfile = open(sys.argv[1], 'r')
outputBVH = open(sys.argv[2], 'w+')

#Archivo de Excel para analizar error cuadrático medio
ECM = open('ECM.csv', 'w+')
ecm_csv = csv.writer(ECM, dialect='excel')

#Contador de frames (cuadros del MoCap)
frame = 1

#Inicio-----
print('Inicializando BVH TuneUp... \nCreando jerarquía de Bones...')

#con el método isdigit() determinamos si el primer o segundo elemento de la línea
#parseada es un dígito, si lo es, esta línea de seguro es de MOTION por tener números
for line in BVHfile.readlines():
    if line[0].isdigit() or line[1].isdigit():

        #Aquí hay que separar la línea parseada y crear un arreglo de MOTION válido
        line = re.split('\s+|\n', line)
        #eliminar el ultimo elemento de line '\n' y convertir a float todo
        line.pop()
        MOTION = [float(nums) for nums in line]
        print('Procesando movimientos del frame: ' + str(frame))
        #Preservuamos vector de MOTION orginal para su posterior análisis y validación
        ORIGIN = MOTION
        ecm_csv.writerow(ORIGIN)

        #Aquí se aplican los estudios de goniometría a cada Bone-----
        Leftfoot.Goniometry_check(MOTION, frame)
        Rightfoot.Goniometry_check(MOTION, frame)
        LeftLeg.Goniometry_check(MOTION, frame)
        RightLeg.Goniometry_check(MOTION, frame)
        LeftUpLeg.Goniometry_check(MOTION, frame)
        RightUpLeg.Goniometry_check(MOTION, frame)

        Neck.Goniometry_check(MOTION, frame)
        LeftArm.Goniometry_check(MOTION, frame)
        RightArm.Goniometry_check(MOTION, frame)
        LeftForeArm.Goniometry_check(MOTION, frame)
        RightForeArm.Goniometry_check(MOTION, frame)
        LeftHand.Goniometry_check(MOTION, frame)
```

```

RightHand.Goniometry_check(MOTION, frame)
Spine1.Goniometry_check(MOTION, frame)

#escriba el vector de MOTION al ECM.csv para validación de datos
ecm_csv.writerow(MOTION)

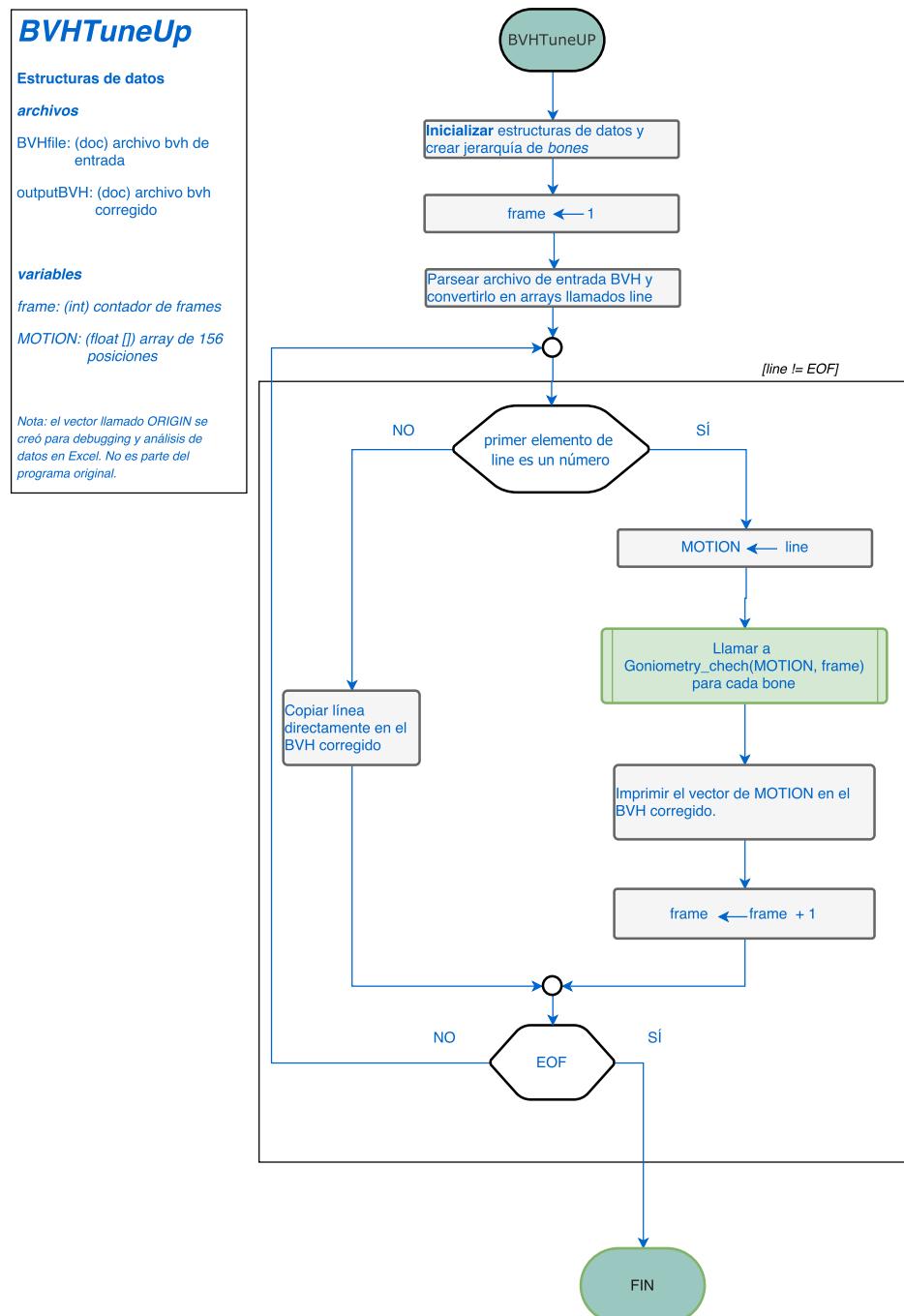
#aqui se debe escribir el arreglo de MOTION al outputBVH convirtiéndolo en string
outputMotion = [str(nums) for nums in MOTION]
#se convierte el arreglo de una lista de floats a un string
salida = ' '.join(outputMotion)
outputBVH.write(salida + '\n')

#incrementar el contador de frames
frame += 1
else:
    outputBVH.write(line)

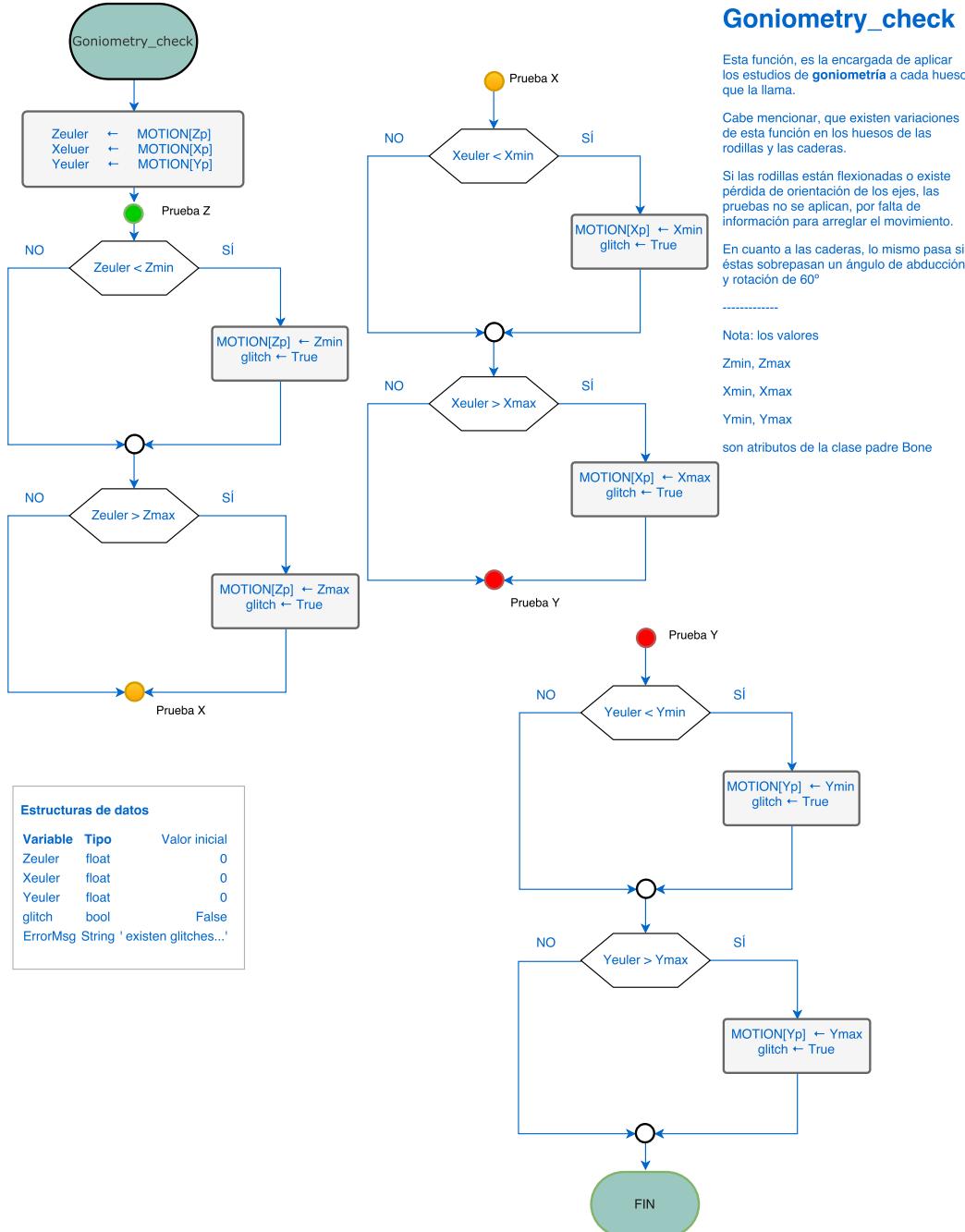
---final del script se libera la memoria---
print('Creando Glitch_Report...\\nCreando archivo BVH corregido en este directorio...\\n')
print('Liberando memoria...\\n')
print('BVH_TuneUp finalizó con éxito. Que tenga un buen día.')
BVHfile.close()
outputBVH.close()
ECM.close()

```

A.2. Diagrama de flujo del programa principal



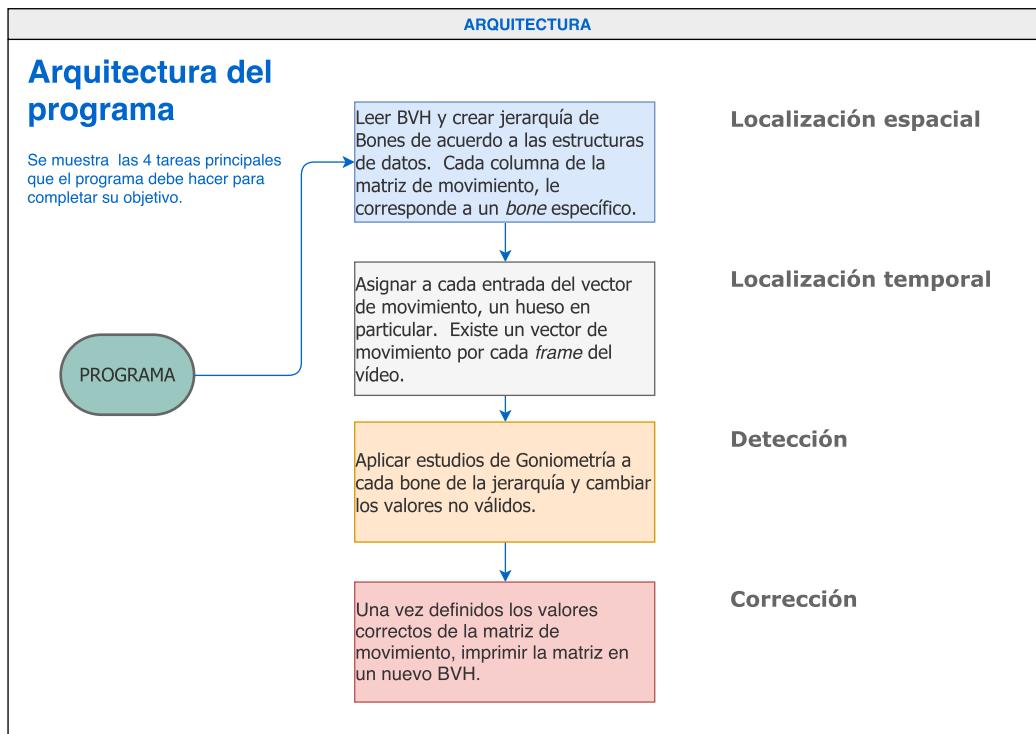
A.3. Diagrama de flujo para la función Goniometry_check



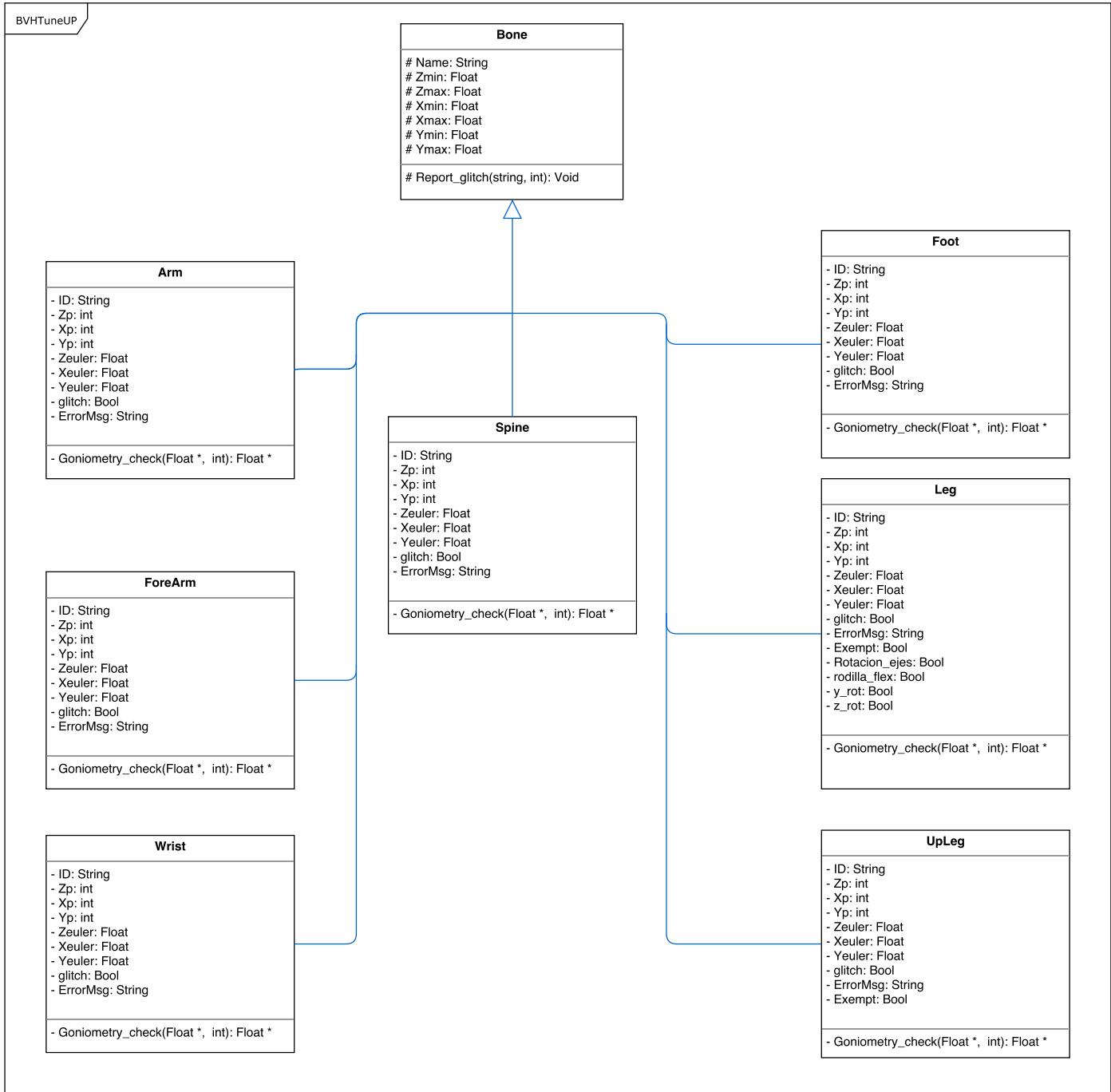
Apéndice B

Diseño del programa

B.1. Arquitectura del programa



B.2. Diagrama UML para las clases



Apéndice C

Código de las clases

C.1. Clase padre **Bone**

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

*IE0499 / Proyecto Eléctrico
Mario Alberto Castresana Avendaño
A41267*

Programa: BVH_TuneUp

archivo: Bone.py

descripción:

Este archivo contiene la clase Bone, utilizada para manejar toda la información contenida en el archivo BVH.

A partir de esta clase, se definen todos los huesos con el nombre correspondiente dentro de la sección HIERARCHY del BVH y se agregan las características goniométricas de cada hueso del esqueleto humano, utilizando un esquema de herencia.

La instanciación de cada hueso basado en esta clase, da como resultado la construcción de todo el esqueleto representado por la sección HIERARCHY del BVH.

"""

`class Bone:`

"""

Esta clase contiene todos los métodos necesarios para manipular la información de cualquier hueso de la jerarquía proveniente de la sección

```

HIERARCHY del archivo BVH.
"""

def __init__(self,
             Name='GenericBone',
             Zmin=0,
             Zmax=0,
             Xmin=0,
             Xmax=0,
             Ymin=0,
             Ymax=0):
    """
    Se inicializa cada hueso con sus valores por default:
    Name: nombre del hueso
    Valores Goniométricos para este hueso:
    Zmax, Zmin : valores máximos y mínimos en Z
    Xmax, Xmin : valores máximos y mínimos en X
    Ymax, Ymin : valores máximos y mínimos en Y
    """
    self.Name = Name
    self.Zmin = Zmin
    self.Zmax = Zmax
    self.Xmin = Xmin
    self.Xmax = Xmax
    self.Ymin = Ymin
    self.Ymax = Ymax

def Report_glitch(self, errorMsg='Hay un glitch de movimiento aquí', frame=0):
    """
    Descripción:
    Esta función se encarga de reportar los glitches que se encuentra y en
    qué frame en específico está. Cada glitch se reporta en un archivo de
    texto llamado Glitch_Report

    argumentos:
    errorMsg: string que contiene un mensaje de error
    frame: cuadro donde se produce el glitch
    """
    #Creamos el archivo de reporte
    glitch = open('Glitch_Report', 'a')
    errorString = 'Frame: ' + str(frame) + ' bone: ' + self.Name + ' ' + self.ID + ' | ' + errorMsg
    #se escribe el string de error y se libera la memoria
    glitch.write(errorString)

```

```
glitch.close()
```

C.2. La subclase **Arm**

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

IE0499 / Proyecto Eléctrico

Mario Alberto Castresana Avendaño

A41267

Programa: BVH_TuneUp

archivo: Arm.py

descripción:

Este archivo contiene la clase Arm, la cual se utiliza para implementar el brazo izquierdo y el derecho. Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z aducción(+) y abducción(-) horizontales

X aducción(+) y abducción(-) plano frontal

Y rotación interna(+) y rotación externa(-)

"""

```
from Bone import Bone
```

```
class Arm(Bone):
```

"""

Esta subclase implementa el estudio de goniometría para los brazos en el esqueleto del BVH. La jerarquía los llama "Arm".

"""

```
def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):
```

"""

Se inicializa este hueso con los siguientes parámetros

ID: identificador del bone. Ej: izquierdo/derecho

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

```
self.ID = ID
```

```
self.Zp = Zp
```

```

self.Xp = Xp
self.Yp = Yp
#se llama al constructor de la super clase para acceder a todos los atributos
#de goniometría
Bone.__init__(self,
               Name='Brazo',
               Zmin=-90.000000,
               Zmax=90.000000,
               Xmin=-45.000000,
               Xmax=135.000000,
               Ymin=-90.000000,
               Ymax=90.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    ErrorMsg = ' existen glitches de '

    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'abduccion horizontal | '

    if Zeuler > self.Zmax:

```

```
MOTION[self.Zp] = self.Zmax
glitch = True
ErrorMsg += 'aduccion horizontal | '

#aquí probamos límites en X
if Xeluer < self.Xmin:
    MOTION[self.Xp] = self.Xmin
    glitch = True
    ErrorMsg += 'abduccion frontal | '
if Xeluer > self.Xmax:
    MOTION[self.Xp] = self.Xmax
    glitch = True
    ErrorMsg += 'aduccion frontal| '

#aquí probamos límites en Y
if Yeuler < self.Ymin:
    MOTION[self.Yp] = self.Ymin
    glitch = True
    ErrorMsg += 'rotacion externa | '
if Yeuler > self.Ymax:
    MOTION[self.Yp] = self.Ymax
    glitch = True
    ErrorMsg += 'rotacion interna | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```

C.3. La subclase **ForeArm**

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

*IE0499 / Proyecto Eléctrico
Mario Alberto Castresana Avendaño
A41267*

Programa: BVH_TuneUp

archivo: ForeArm.py

descripción:

Este archivo contiene la clase ForeArm, la cual se utiliza para implementar el codo izquierdo y el derecho. Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z no hay movimiento en el codo

X pronación (+) y supinación (-)

Y flexión (+) y extensión (-)

"""

```
from Bone import Bone
```

```
class ForeArm(Bone):
```

"""

Esta subclase implementa el estudio de goniometría para los codos en el esqueleto del BVH. La jerarquía los llama "ForeArm".

"""

```
def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):
```

"""

Se inicializa este hueso con los siguientes parámetros

ID: identificador del bone. Ej: izquierdo/derecho

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

```
self.ID = ID
```

```
self.Zp = Zp
```

```

self.Xp = Xp
self.Yp = Yp
#se llama al constructor de la super clase para acceder a todos los atributos
#de goniometría
Bone.__init__(self,
              Name='Codo',
              Zmin=0.000000,
              Zmax=0.000000,
              Xmin=-90.000000,
              Xmax=90.000000,
              Ymin=-10.000000,
              Ymax=140.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    ErrorMsg = ' existen glitches de '

    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'Movimiento no valido en el eje z | '

    if Zeuler > self.Zmax:

```

```
MOTION[self.Zp] = self.Zmax
glitch = True
ErrorMsg += 'Movimiento no valido en el eje z | '

#aquí probamos límites en X
if Xeluer < self.Xmin:
    MOTION[self.Xp] = self.Xmin
    glitch = True
    ErrorMsg += 'supinacion | '
if Xeluer > self.Xmax:
    MOTION[self.Xp] = self.Xmax
    glitch = True
    ErrorMsg += 'pronacion| '

#aquí probamos límites en Y
if Yeuler < self.Ymin:
    MOTION[self.Yp] = self.Ymin
    glitch = True
    ErrorMsg += 'extension | '
if Yeuler > self.Ymax:
    MOTION[self.Yp] = self.Ymax
    glitch = True
    ErrorMsg += 'flexion | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```

C.4. La subclase Leg

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

*IE0499 / Proyecto Eléctrico
Mario Alberto Castresana Avendaño
A41267*

Programa: BVH_TuneUp

archivo: Leg.py

descripción:

Este archivo contiene la clase Leg, la cual se utiliza para implementar la rodilla izquierda y la derecha. Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z torsión no válida

X Flexión + y extensión -

Y rotación no válida

"""

```
from Bone import Bone
```

```
class Leg(Bone):
```

"""

Esta subclase implementa el estudio de goniometría para las rodillas en el esqueleto del BVH. La jerarquía los llama "Leg".

"""

```
def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):
```

"""

Se inicializa este hueso con los siguientes parámetros

ID: identificador del bone. Ej: izquierdo/derecho

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

```
self.ID = ID
```

```
self.Zp = Zp
```

```

    self.Xp = Xp
    self.Yp = Yp
    #se llama al constructor de la super clase para acceder a todos los atributos
    #de goniometría
    Bone.__init__(self,
                  Name='Rodilla',
                  Zmin=-0.200000,
                  Zmax=0.200000,
                  Xmin=0.000000,
                  Xmax=150.000000,
                  Ymin=-1.000000,
                  Ymax=1.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    #Exempt es una variable que se activa cuando detecta problemas de rotación
    #de ejes Z y Y en las rodillas
    Exempt = False
    ErrorMsg = ' existen glitches de '

    #Variables para probar si hubo rotación de ejes y el esqueleto está agachado
    rodilla_flex = Xeluer > 13.0 or Xeluer < -15.0
    y_rot = Yeuler > 20.0 or Yeuler < -20.0
    z_rot = Zeuler > 40.0 or Zeuler < -40.0

```

```

Rotacion_ejes = y_rot or z_rot

if rodilla_flex and Rotacion_ejes:
    Exempt = True

if Exempt:
    #Existen dos pruebas goniométricas distintas de acuerdo al nivel de flexión de las
    #rodillas. En el caso de que las rodillas tengan un ángulo de flexión mayor a 45º o
    #exista una rotación de los ejes Z y Y, debemos incrementar los límites de movilidad.
    #en Z y Y. Esto debido al comportamiento de los huesos en el BVH, los cuales rotan
    #los ejes Y y Z para representar movimientos de un esqueleto agachado.

    #Esto ocurre debido a la pérdida de orientación del hueso, por parte de las cámaras
    #en los ejes Z y Y.

    #probamos límites nuevos en Z
    if Zeuler < -160.000000:
        #MOTION[self.Zp] no se le aplica restricción en Z
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Z- | '

    if Zeuler > 160.000000:
        #MOTION[self.Zp] no se le aplica restricción en Z
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Z+ | '

    #aquí probamos nuevos límites en X
    if Xeluer < -150.000000:
        #MOTION[self.Xp] no se le aplica restricción en X
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en X- | '

    if Xeluer > 150.000000:
        #MOTION[self.Xp] no se le aplica restricción en X
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en X+ | '

    #aquí probamos nuevos límites en Y
    if Yeuler < -105.000000:
        #MOTION[self.Yp] no se le aplica restricción en Y
        glitch = True
       ErrorMsg += 'pérdida de orientación de los sensores en Y- | '

```

```

    if Yeuler > 105.000000:
        #MOTION[self.Yp] no se le aplica restricción en Y
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en Y+ | '

else:
    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'torsión | '

    if Zeuler > self.Zmax:
        MOTION[self.Zp] = self.Zmax
        glitch = True
        ErrorMsg += 'torsión | '

    #aquí probamos límites en X
    if Xeluer < self.Xmin:
        MOTION[self.Xp] = self.Xmin
        glitch = True
        ErrorMsg += 'extension | '
    if Xeluer > self.Xmax:
        MOTION[self.Xp] = self.Xmax
        glitch = True
        ErrorMsg += 'flexion | '

    #aquí probamos límites en Y
    if Yeuler < self.Ymin:
        MOTION[self.Yp] = self.Ymin
        glitch = True
        ErrorMsg += 'rotacion interna | '
    if Yeuler > self.Ymax:
        MOTION[self.Yp] = self.Ymax
        glitch = True
        ErrorMsg += 'rotacion externa | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)

```

C.5. La subclase *UpLeg*

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

IE0499 / Proyecto Eléctrico

Mario Alberto Castresana Avendaño

A41267

Programa: BVH_TuneUp

archivo: UpLeg.py

descripción:

Este archivo contiene la clase UpLeg, la cual se utiliza para implementar las caderas. Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z aducción + y abducción -

X Flexión + y Extensión -

Y rotación externa + y rotación interna -

"""

`from Bone import Bone`

`class UpLeg(Bone):`

"""

Esta subclase implementa el estudio de goniometría para las caderas en el esqueleto del BVH. La jerarquía los llama "UpLeg".

"""

`def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):`

"""

Se inicializa este hueso con los siguientes parámetros

ID: identificador del bone. Ej: izquierdo/derecho

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

`self.ID = ID`

`self.Zp = Zp`

```

self.Xp = Xp
self.Yp = Yp
#se llama al constructor de la super clase para acceder a todos los atributos
#de goniometría
Bone.__init__(self,
               Name='Cadera',
               Zmin=-60.000000,
               Zmax=60.000000,
               Xmin=-35.000000,
               Xmax=165.000000,
               Ymin=-50.000000,
               Ymax=50.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    #Exempt es una variable que se activa cuando detecta problemas de rotación
    #de ejes Z y Y en las rodillas
    Exempt = False
    ErrorMsg = ' existen glitches de '

    if Yeuler > 60.0 or Yeuler < -60.0:
        Exempt = True
    if Zeuler > 60.0 or Zeuler < -60.0:
        Exempt = True

```

```

if Exempt:
    #Existen dos pruebas goniométricas distintas de acuerdo al nivel de flexión de las
    #caderas. En el caso de que las caderas tengan un ángulo de abducción mayor a 60º o
    #exista una rotación de los ejes Z y Y, debemos incrementar los límites de movilidad.
    #en Z y Y. Esto debido al comportamiento de los huesos en el BVH, los cuales rotan
    #los ejes Y y Z para representar movimientos de un esqueleto agachado.

    #Esto ocurre debido a la pérdida de orientación del hueso, por parte de las cámaras
    #en los ejes Z y Y.

    #probamos límites nuevos en Z
    if Zeuler < -160.000000:
        #MOTION[self.Zp] = -150.000000
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en Z- | '

    if Zeuler > 160.000000:
        #MOTION[self.Zp] = 150.000000
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en Z+ | '

    #aquí probamos los mismos límites en X
    if Xeluer < -45.000000:
        #MOTION[self.Xp] = self.Xmin
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en X-'
    if Xeluer > 180.000000:
        #MOTION[self.Xp] = self.Xmax
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en X+'

    #aquí probamos nuevos límites en Y
    if Yeuler < -105.000000:
        #MOTION[self.Yp] = -105.000000
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en Y- | '
    if Yeuler > 105.000000:
        #MOTION[self.Yp] = 105.000000
        glitch = True
        ErrorMsg += 'pérdida de orientación de los sensores en Y+ | '

```

```
else:
    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'abduccion | '

    if Zeuler > self.Zmax:
        MOTION[self.Zp] = self.Zmax
        glitch = True
        ErrorMsg += 'aducción | '

    #aquí probamos límites en X
    if Xeluer < self.Xmin:
        MOTION[self.Xp] = self.Xmin
        glitch = True
        ErrorMsg += 'extensión | '
    if Xeluer > self.Xmax:
        MOTION[self.Xp] = self.Xmax
        glitch = True
        ErrorMsg += 'flexión | '

    #aquí probamos límites en Y
    if Yeuler < self.Ymin:
        MOTION[self.Yp] = self.Ymin
        glitch = True
        ErrorMsg += 'rotación interna | '
    if Yeuler > self.Ymax:
        MOTION[self.Yp] = self.Ymax
        glitch = True
        ErrorMsg += 'rotación externa | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```

C.6. La subclase *Spine*

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

IE0499 / Proyecto Eléctrico

Mario Alberto Castresana Avendaño

A41267

Programa: BVH_TuneUp

archivo: Spine.py

descripción:

Este archivo contiene la clase Spine, la cual se utiliza para implementar la columna vertebral (sección lumbar-torácica). Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z flexión lateral derecha + e izquierda -

X flexión + y extensión -

Y rotación izquierda + y derecha -

"""

`from Bone import Bone`

`class Spine(Bone):`

"""

Esta subclase implementa el estudio de goniometría para las muñecas en el esqueleto del BVH. La jerarquía los llama "Hand".

"""

`def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):`

"""

Se inicializa este hueso con los siguientes parámetros

ID: identificador del bone. Ej: izquierdo/derecho

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

`self.ID = ID`

`self.Zp = Zp`

```

self.Xp = Xp
self.Yp = Yp
#se llama al constructor de la super clase para acceder a todos los atributos
#de goniometría
Bone.__init__(self,
              Name='Espina',
              Zmin=-25000000,
              Zmax=25.000000,
              Xmin=-25.000000,
              Xmax=45.000000,
              Ymin=-45.000000,
              Ymax=45.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    ErrorMsg = ' existen glitches de '

    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'flexion lateral | '

    if Zeuler > self.Zmax:

```

```
MOTION[self.Zp] = self.Zmax
glitch = True
ErrorMsg += 'flexion lateral | '

#aquí probamos límites en X
if Xeluer < self.Xmin:
    MOTION[self.Xp] = self.Xmin
    glitch = True
    ErrorMsg += 'hiperextension | '
if Xeluer > self.Xmax:
    MOTION[self.Xp] = self.Xmax
    glitch = True
    ErrorMsg += 'flexion | '

#aquí probamos límites en Y
if Yeuler < self.Ymin:
    MOTION[self.Yp] = self.Ymin
    glitch = True
    ErrorMsg += 'rotacion excesiva a la derecha | '
if Yeuler > self.Ymax:
    MOTION[self.Yp] = self.Ymax
    glitch = True
    ErrorMsg += 'rotacion excesiva a la izquierda | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```

C.7. La subclase Wrist

"""

UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

*IE0499 / Proyecto Eléctrico
Mario Alberto Castresana Avendaño
A41267*

Programa: BVH_TuneUp

archivo: Wrist.py

descripción:

Este archivo contiene la clase Wrist, la cual se utiliza para implementar las muñecas. Los estudios de goniometría para este hueso se basan en los siguientes límites de los ángulos de Euler:

Z flexión palmar + y extensión (dorsiflexión) -

X flexión radial + y Flexión ulnar -

Y rotación no válida

"""

`from Bone import Bone`

`class Wrist(Bone):`

"""

Esta subclase implementa el estudio de goniometría para las muñecas en el esqueleto del BVH. La jerarquía los llama "Hand".

"""

`def __init__(self, ID=' ', Zp=0, Xp=0, Yp=0):`

"""

*Se inicializa este hueso con los siguientes parámetros
ID: identificador del bone. Ej: izquierdo/derecho*

Cada posición del hueso se define con un vector de ángulos de Euler (Z, X, Y) los cuales tienen una posición específica dentro del array de la sección MOTION del BVH

Zp: índice del array MOTION que contiene el angulo de euler Z para ese hueso

Xp: índice del array MOTION que contiene el angulo de euler X para ese hueso

Yp: índice del array MOTION que contiene el angulo de euler Y para ese hueso

"""

`self.ID = ID`

`self.Zp = Zp`

```

self.Xp = Xp
self.Yp = Yp
#se llama al constructor de la super clase para acceder a todos los atributos
#de goniometría
Bone.__init__(self,
              Name='Muñeca',
              Zmin=-70.000000,
              Zmax=90.000000,
              Xmin=-30.000000,
              Xmax=20.000000,
              Ymin=0.000000,
              Ymax=0.000000)

def Goniometry_check(self, MOTION, frame):
    """
    Descripción:
    Esta función se encarga de comparar el valor de los ángulos de Euler que
    un hueso posee en un frame determinado, con el valor de los límites
    goniométricos de ese hueso en particular. Si algún ángulo de Euler excede
    los límites del movimiento humano, se reportará un glitch en ese frame
    y se procederá a corregirlo en el arreglo MOTION.

    argumentos:
    MOTION: arreglo de 156 posiciones que contiene todos los ángulos de Euler
    para cada hueso en un frame dado. El orden de cada hueso viene dado por
    la sección HIERARCHY del BVH.
    frame: cuadro del video de MoCap que se está analizando
    """
    #Primero, definimos los valores de cada ángulo de Euler
    Zeuler = MOTION[self.Zp]
    Xeluer = MOTION[self.Xp]
    Yeuler = MOTION[self.Yp]
    glitch = False
    ErrorMsg = ' existen glitches de '

    #probamos límites en Z
    if Zeuler < self.Zmin:
        MOTION[self.Zp] = self.Zmin
        glitch = True
        ErrorMsg += 'dorsiflexión | '

    if Zeuler > self.Zmax:

```

```
MOTION[self.Zp] = self.Zmax
glitch = True
ErrorMsg += 'flexión palmar | '

#aquí probamos límites en X
if Xeluer < self.Xmin:
    MOTION[self.Xp] = self.Xmin
    glitch = True
    ErrorMsg += 'flexion ulnar | '
if Xeluer > self.Xmax:
    MOTION[self.Xp] = self.Xmax
    glitch = True
    ErrorMsg += 'flexion radial| '

#aquí probamos límites en Y
if Yeuler < self.Ymin:
    MOTION[self.Yp] = self.Ymin
    glitch = True
    ErrorMsg += 'rotacion no valida en Y | '
if Yeuler > self.Ymax:
    MOTION[self.Yp] = self.Ymax
    glitch = True
    ErrorMsg += 'rotacion no válida en Y | '

if glitch:
    self.Report_glitch(ErrorMsg, frame)
```


Bibliografía

- [Albon, 2016] Albon, C. (2016). Precision, Recall and F1 Scores. Tomado de https://chrisalbon.com/machine-learning/precision_recall_and_F1_scores.html.
- [AOK, 2017] AOK (2017). Joint Range of Motion Data Using a Goniometer. Tomado de https://aokhealth.securestand.com/xq/ASP/ProductID.614/qx/PDF/Using_a_Goniometer_Effectively.pdf.
- [Biovision BVH, 2017] Biovision BVH (2017). Biovision BVH. Tomado de <https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html>.
- [CGTalk, 2017] CGTalk (2017). Cgtalk: Bone set-up bvh file. Tomado de http://lookslikematt.com/cgtalk/mb_bone-set-up_bvh_characterstudio.jpg.
- [Chandola et al., 2009] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(September):1–58.
- [Drews et al., 2013] Drews, T. M., Kry, P. G., Forbes, J. R., and Verbrugge, C. (2013). Sequential pose estimation using linearized rotation matrices. *Proceedings - 2013 International Conference on Computer and Robot Vision, CRV 2013*, pages 113–120.
- [Duda et al., 2000] Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification*. Wiley-Interscience, New York, USA, 2nd editio edition.
- [Kapandji and Lacomba, 2006] Kapandji, A. and Lacomba, M. (2006). *Fisiología articular: esquemas comentados de mecánica humana*. Number v. 1 in Fisiología articular. Tomo 1. Hombro, codo, pronosupinación, muñeca, mano. Editorial Médica Panamericana.
- [Lifshits et al., 2004] Lifshits, M., Goldenberg, R., Rivlin, E., and Rudzsky, M. (2004). Using Pattern Recognition for Self-Localization in Semiconductor Manufacturing Systems. pages 520–527.
- [Luttgens et al., 2011] Luttgens, K., Hamilton, N., and Weimar, W. (2011). *Kinesiology: Scientific Basis of Human Motion*. Brown & Benchmark, Dubuque, IA, twelfth ed edition.
- [OptiTrack, 2017] OptiTrack (2017). Optitrack - general faqs. Tomado de <http://optitrack.com/support/faq/general.html>.

[Parent, 2012] Parent, R. (2012). *Computer Animation: Algorithms and Techniques*. Elsevier Science, Waltham, MA, 3rd editio edition.

[Planimetría, 2017] Planimetría (2017). Planimetría. Tomado de
<https://sites.google.com/site/cienciasdelasalud1bloque2/ciencias-de-la-salud-bloque2/planimetria>.

[Taboadela, 2007] Taboadela, C. H. (2007). *Goniometria una herramienta para la evaluacion de las incapacidades*. ASOCIART, S.A., Buenos Aires, Argentina.