Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Animación Esquelética con datos de captura de movimiento

Trabajo Especial de Grado presentado ante la Ilustre
Universidad Central de Venezuela
por el Br. Dayrut D Simancas M
para optar al título de Licenciado en Computación.

Tutor:
Prof. Rhadamés Carmona

Caracas, Septiembre del 2019

Caracas, 20 de Septiembre de 2019
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Dayrut David Simancas Martínez C.I.: V-21.412.683, con el título "Animación esquelética con datos de captura de movimiento", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 20 de Septiembre de 2019, a las 9am, para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual este realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo con una calificación de dieciocho (18) puntos.

En fe de lo cual se levanta la presente acta, en Caracas el viernes veinte (20) de septiembre de 2019 dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Tutor Rhadamés Carmona.

REPUBLICA DE VENEZUELA
Facultad de Ciencias
Escuela de Computación
Universidad Central - Caracas

Prof. Rhadamés Carmona
(Tutor)

Prof. Walter Hernández
(Jurado Principal)

Jesús Cibeira

Prof. Jesús Cibeira
(Jurado Principal)

Resumen

La animación de personajes 3D articulados dentro de aplicaciones convencionales de animación esquelética por lo general requieren un sistema de control (rig) manual para especificar la estructura esquelética interna de los modelos y para definir como su superficie geométrica (malla o skin) debe deformarse durante cada fotograma de una animación. Para este trabajo se desarrolló el prototipo de una aplicación web usando WebGL y Three.js la cual dada una malla de triángulos 3D de un personaje humanoide como parámetro de entrada adapta un esqueleto (jerarquía de articulaciones) dentro de la superficie geométrica del modelo y después de asociarlos permite que el uso de datos de captura de movimiento animen el personaje. El procedimiento de adaptación de una jerarquía de articulaciones dentro de un modelo es conocido como incrustación de esqueleto, y dentro de la aplicación desarrollada se incluyó un módulo basado en la biblioteca Pinocchio (Baran y Popović [3]) el cual facilita el proceso de rigging automático de los personajes cargados.

Palabras Clave: Animación esquelética, datos de captura de movimiento, rigging automático, WebGL, three.js.

Índice general

Introducción

El diseño y modelado de personajes 3D articulados se ha vuelto mucho más sencillo que antes con la introducción de sistemas fáciles de utilizar, aplicaciones destinadas al uso de principiantes que han hecho bastante accesible la creación y manipulación de personajes 3D. Sin embargo, dar vida a estos modelos estáticos no resulta ser del todo fácil. En aplicaciones convencionales de animación esquelética, el usuario debe manipular el personaje directamente, colocar las articulaciones (joints) del esqueleto, especificar que partes de la malla (superficie geométrica) corresponden a cada hueso y luego posicionar el modelo de distintas maneras para dotarlo de movimiento. Estos pasos son parte del procedimiento típico asociado con un segundo de animación en la pantalla, haciendo que el proceso de animación/visualización de acciones simples de los personajes resulte ser más tediosos y difícil de lo que podría ser.

Las ventajas de llevar a cabo el proceso de animación de dicha manera radican en el control preciso que los animadores tienen sobre los personajes durante cada uno de los fotogramas de las animaciones realizadas, y en la oportunidad de no limitar los movimientos creados con las mismas reglas que rigen el mundo real. Sin embargo, los inconvenientes también son significativos. Primero, tal proceso requiere cierta cantidad de conocimientos específicos, y la falta de estos imposibilita animar los modelos 3D de personajes. En segundo lugar, los movimientos de los personajes animados manualmente suelen ser propensos a errores y presentar interrupciones, siendo necesario revisar de forma continua la consistencia de su animación. Tercero, en la mayoría de los casos, los personajes que se animan presentan esqueletos similares y el comportamiento deseado en estos al moverse suele ser aproximadamente igual. Todo esto causa que reiterar el proceso de animación para múltiples modelos 3D conlleve un esfuerzo manual significativo y sin el uso de herramientas para automatizar algunos de los pasos involucrados se consumiría bastante tiempo.

Hay técnicas que tienen como principal enfoque la generación automática de jerarquías de articulaciones (extracción de esqueletos) para su animación, pero estas en su mayoría requieren más de una pose estática como entrada para poder producir esqueletos cuyo uso no sea limitado. Aquí es donde el método conocido como incrustación de esqueleto (skeleton embedding) resulta de gran utilidad: ya que este al redimensionar y posicionar un esqueleto dado para que encaje dentro de un modelo 3D, permite que un único esqueleto pueda usarse con una amplia gama de personajes, y junto a una colección de datos de movimientos para unos pocos esqueletos, es posible establecer un sistema de animación fácil de usar.

Las animaciones producidas para personajes 3D por lo general solo son compatibles con el esqueleto para el que fueron creadas, pero se puede hacer una excepción a esta regla para los esqueletos que están estrechamente relacionados. Aplicando técnicas de reorientación de animaciones (animation retargeting), es posible utilizar los datos de movimiento ligados al esqueleto de un personaje para mover el esqueleto de otro y transferir la animación esquelética correspondiente a una nueva malla sin un esfuerzo manual significativo.

En tal sentido, el objetivo fundamental de este Trabajo Especial de Grado es aplicar los principios de incrustación de esqueletos (skeleton embedding) y reorientación de animaciones (animation retargeting) para desarrollar una aplicación prototipo utilizando WebGL y three.js (API y biblioteca JavaScript) la cual simplifique el proceso de animación de personajes en un entorno web.

Objetivos

Objetivo General

Desarrollar una aplicación usando WebGL y three.js que permita visualizar animaciones sencillas de personajes a partir de datos de captura de movimiento, adaptando un esqueleto genérico a las mallas estáticas de distintos modelos 3D.

Objetivos Específicos

- Estudiar librerías existentes que permitan crear escena para la visualización de modelos 3D e incluir interfaz gráfica de usuario para controlar elementos visuales dentro de la misma.

- Seleccionar método y herramientas tecnológicas que permitan integrar módulo de software en la aplicación para incluir la funcionalidad de incrustación de esqueleto.
- Aplicar técnica de incrustación de esqueleto para insertar y ajustar jerarquías de articulaciones dentro de modelos 3D.
- Aplicar técnica de reorientación de animaciones para animar los modelos 3D de personajes a partir de datos de captura de movimiento.
- Evaluar la correctitud de los esqueletos incrustados y la calidad de las animaciones producidas.

Alcance

Los modelos de prueba a utilizar en la aplicación para evaluar la correctitud de los esqueletos incrustados serán 16 mallas (archivos .obj)[1] construidas por un artista usando Cosmic Blobs® software (desarrollado por Dassault Systèmes SolidWorks Corp), los mismos modelos usados en el trabajo de Baran y Popović [3] para evaluar su sistema.

El Usuario de la aplicación tendrá la opción de cargar cualquier modelo en formato .obj que desee, pero para que el personaje especificado en el archivo sea procesado exitosamente y pueda ser visualizado correctamente, este debe cumplir ciertos requisitos que evitan inconvenientes y resultados no deseados:

- **La malla debe ser única y cerrada:** la superficie geométrica en el archivo debe ser un solo componente conectado. Evitar modelos de personajes con múltiples partes que no crean un volumen compacto.
- **La malla debe tener su volumen bien definido:** la superficie geométrica debe cumplir con las siguientes propiedades: 1. Cada arista pertenece a dos caras. 2. Cada vértice está rodeado por una secuencia de aristas y caras. 3. Las caras solo se intersecan entre sí en bordes y vértices comunes. Otra forma de expresarlo, el modelo debe tener una forma que pueda manufacturarse en la vida real y pueda desplegarse en una pieza plana (el modelo debe ser *manifold*).

---

[1] Modelos 3D de personajes usados para evaluar correctitud de esqueletos incrustados en la aplicación – `http://www.mit.edu/~ibaran/autorig/pinocchio.html`

- **La malla debe estar orientada correctamente:** el modelo de personaje debe estar erguido y frente al usuario.

- **El modelo debe ser proporcionalmente similar al esqueleto inicial:** el modelo de personaje debe poseer una anatomía aproximadamente antropomórfica.

Las animaciones de los personajes serán impulsadas por un conjunto de datos de captura de movimiento humanos obtenidos del sitio `https://sites.google.com/a/cgspeed.com/cgspeed/motion-capture/cmu-bvh-conversion`, una colección de archivos .bvh basados en la base de datos de captura de movimiento de la universidad Carnegie Mellon [2]

La aplicación utilizará WebGL y Three.js, herramientas tecnológicas usadas para crear y mostrar gráficos de computadora 3D en navegadores web. Por lo que será necesario el uso de algún navegador que soporte ambos: Google Chrome 9+, Firefox 4+, Opera 15+, Safari 5.1+, Internet Explorer 11 y Microsoft Edge [3].

Para facilitar la comprensión del lector, este documento se ha estructurado en los siguientes capítulos:

Capítulo 1 - Marco Teórico: presenta los fundamentos conceptuales que sustentan el dominio y contexto bajo los cuales se desarrolló el presente trabajo.

Capítulo 2 - Herramientas y Tecnologías utilizadas: presenta cada uno de los componentes, herramientas y tecnologías que permitieron crear y desarrollar la aplicación.

Capítulo 3 - Implementación de la Aplicación: describe los métodos y componentes involucrados en la creación y ejecución de la aplicación desarrollada, presentando una visión general de su funcionamiento.

Capítulo 4 - Resultados: describe los criterios usados para evaluar el desempeño de la aplicación.

Y finalmente se presentan las Conclusiones, Recomendaciones para futuros trabajos relacionados y las Referencias Bibliográficas y electrónicas consultadas durante la investigación.

---

[2] CMU Graphics Lab Motion Capture Database - `http://mocap.cs.cmu.edu/`

[3] Navegadores web que soportan WebGL - `https://threejs.org/docs/#manual/en/introduction/Browser-support`

Capítulo 1

Marco Teórico

Hay algunos conceptos básicos e importantes que son considerados necesarios para comprender el contexto en el cual se desarrolló el presente Trabajo Especial de Grado. Este capítulo sirve como un resumen para presentar las bases conceptuales fundamentales que sustentan el dominio del trabajo de investigación realizado y la aplicación desarrollada para el mismo.

1.1. Puntos y Coordenadas Cartesianas

Técnicamente hablando, un punto es una ubicación en un espacio n-dimensional, donde n suele ser igual a 2 o 3. Un sistema de coordenadas, es un sistema que utiliza uno o más números (coordenadas) para determinar unívocamente la posición de un punto o de otro objeto geométrico; en el caso del sistema de coordenadas cartesianas, se utilizan dos o tres ejes mutuamente perpendiculares para especificar una posición en un espacio 2D o 3D. Por lo que, un punto P es representado por un par o tripleta de números reales, (P_x, P_y) o (P_x, P_y, P_z) [12] (Figura 1.1).

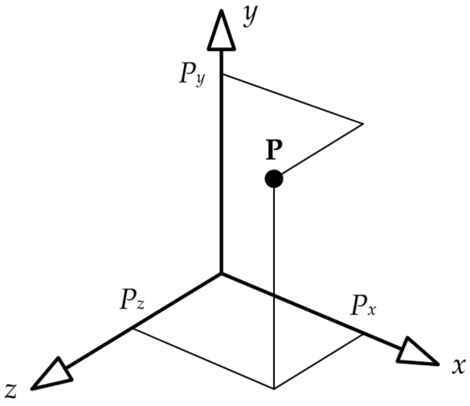


Figura 1.1: Un punto representado en coordenadas cartesianas [12].

1.2. Vectores

Un vector geométrico v es una entidad que tiene tanto una magnitud (también llamada longitud) como una dirección en un espacio n-dimensional. Dicho vector puede ser representado gráficamente como un segmento de línea dirigida que se extiende desde un punto hasta otro con una punta de flecha (Figura 1.2); normalmente utilizado para denotar una dirección o un cambio de posición [26].

Un vector 3D puede ser representado por un conjunto de tres escalares (x, y, z), de forma similar a un punto, con la diferencia de que el vector siempre será un desplazamiento relativo a algún otro punto conocido. Es por esto que un vector puede ser utilizado para representar un punto en el espacio (vector de posición), siempre y cuando este tenga su extremo inicial fijado en el origen del sistema de coordenadas en uso [12].

Se pueden realizar operaciones aritméticas con vectores de la misma manera que se puede con números reales. Una operación básica es la adición, geométricamente, la suma combina dos vectores juntos en un nuevo vector. Si se piensa en un vector como un agente que cambia de posición, entonces dados dos vectores u y v, el nuevo vector $w = u + v$ combina el efecto de cambio de posición de u y v en una misma entidad (ejemplo en Figura 1.3).

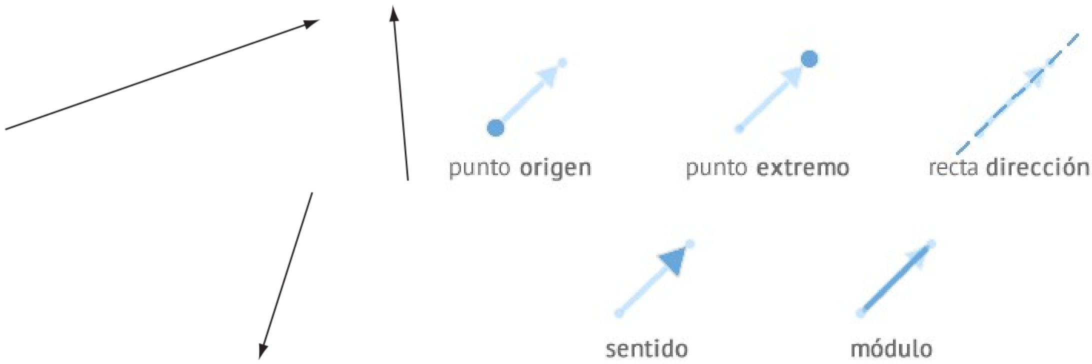


Figura 1.2: Representación gráfica de vectores y visualización de sus características

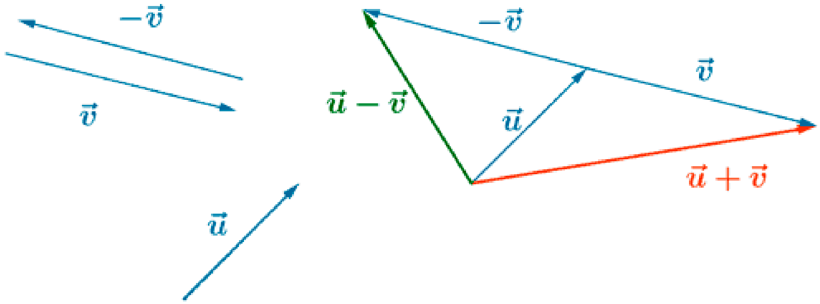


Figura 1.3: Visualización de las propiedades de la suma de vectores en el espacio.

1.3. Mallas y Modelos 3D

Una malla (mesh) es una forma compleja compuesta de triángulos y vértices generalmente utilizada para representar superficies, por lo que una malla no es solo una colección de triángulos no relacionados, sino más bien una red de triángulos que se conectan entre sí a través de vértices y bordes (*edges*) compartidos para formar una única superficie continua [25] (Figura 1.4).

La información mínima requerida para una malla triangular es un conjunto de triángulos (tríos de vértices) y las posiciones (en el espacio 3D) de sus vértices, aunque la mayoría de los programas de modelado 3D utilizados además requieren la capacidad de almacenar datos adicionales en los vértices, bordes o caras de los triángulos para admitir mapas de textura, sombreado, animación, etc. Es por esto que el término "malla" comúnmente es usado para referirse únicamente a la superficie geométrica que puede ser renderizada, mientras que los objetos compuestos que pueden contener múltiples mallas, otros tipos de datos y metadatos adicionales son los llamados “modelos”.

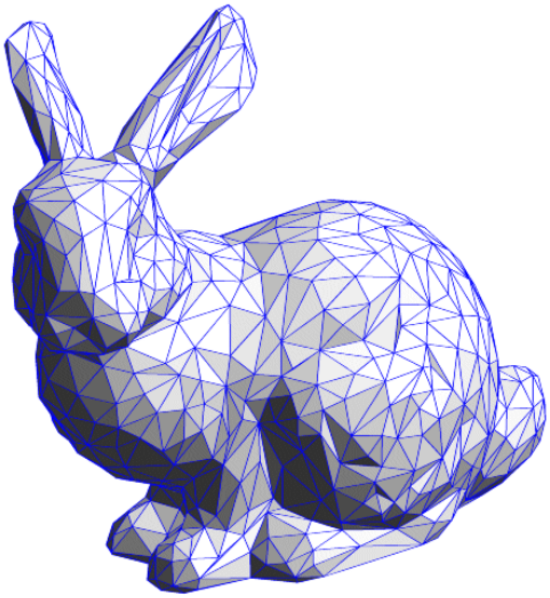


Figura 1.4: Malla triangular simplificada del modelo “Stanford Bunny”.

1.4. Esqueleto, Huesos y Articulaciones

Un esqueleto es una representación abstracta de las distintas maneras que la malla de un modelo 3D puede ser subdividida en distintos grupos de vértices, donde cada grupo corresponde a una parte móvil (hueso) de su cuerpo. Por ejemplo, en la Figura 1.5 se muestra como la malla de un modelo humano podría agruparse en diez partes (huesos) diferentes: cabeza (HEA), torso (TOR), brazo izquierdo superior (LUA), brazo izquierdo inferior (LLA), brazo derecho superior (RUA), brazo derecho inferior (RLA), pierna izquierda superior (LUL), pierna izquierda inferior (LLL), pierna derecha superior (RUL), pierna derecha inferior (RLL).

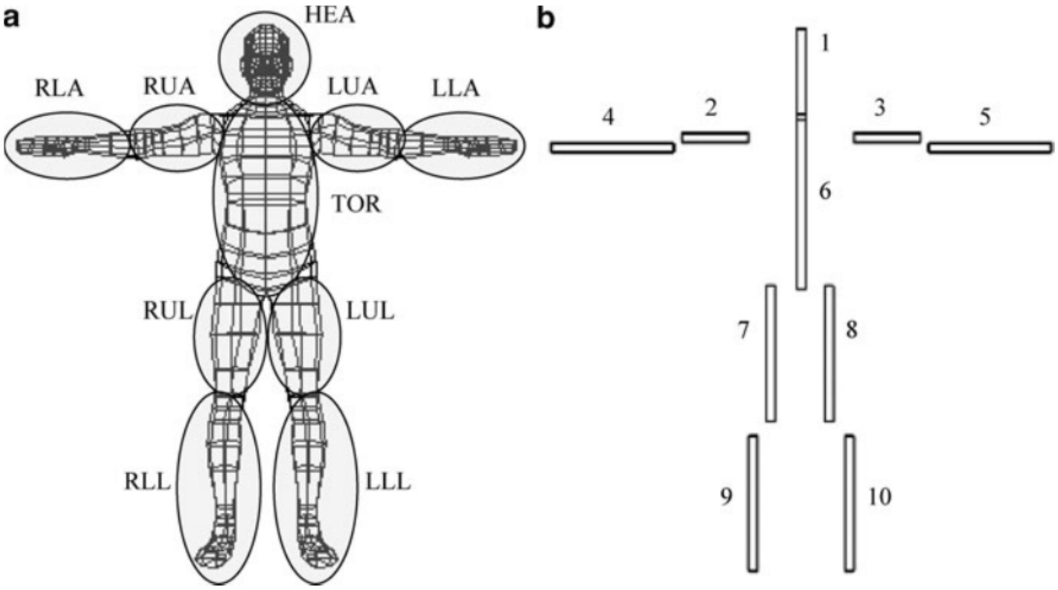


Figura 1.5: (a)Los vértices en un modelo de malla se agrupan en partes que se mueven una respecto a la otra. (b)Una definición de esqueleto formada en base a una agrupación de vértices [21].

Un esqueleto se compone de una jerarquía de piezas rígidas conocidas como articulaciones (*joints*); se selecciona una articulación como el nodo raíz de la estructura de árbol y todos los demás joints son sus hijos, nietos, y así sucesivamente. En el caso del modelo humano, este puede ser modelado como una colección de diferentes partes del cuerpo, con articulaciones en el cuello, hombros, codos, muñecas, caderas, rodillas, y los tobillos.

A menudo los términos "articulación" y "hueso" son utilizados de manera intercambiable, pero el termino hueso (bone) es un término inapropiado. Técnicamente, las articulaciones son los objetos manipulados directamente, mientras que los huesos son simplemente espacios vacíos entre las articulaciones (Figura 1.6); es decir, cada componente hueso/bone es una estructura abstracta, no una primitiva gráfica [12].

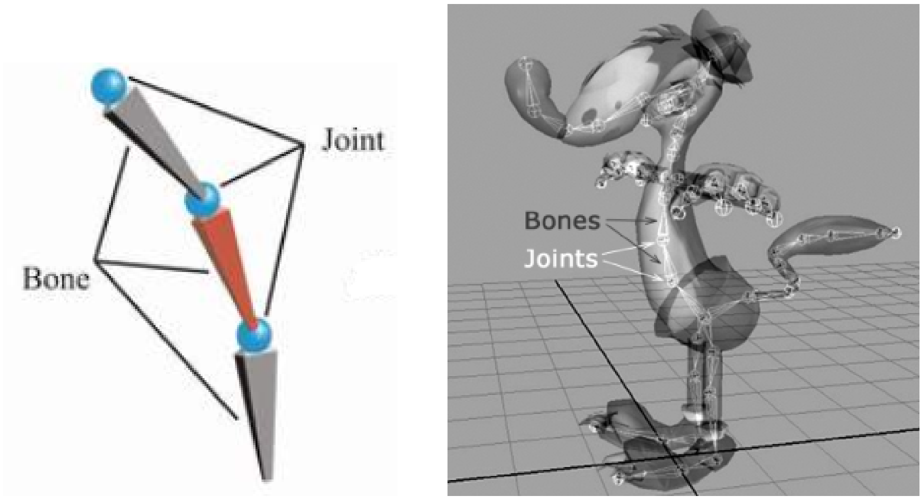


Figura 1.6: Distinción entre los términos bone/hueso y joint/articulación. [1] y [12]

1.5. Rotaciones y Grados de Libertad

El término "grados de libertad" (*degrees of freedom* – DOF) se refiere al número de formas mutuamente independientes en que el estado físico de un objeto (posición y orientación) puede cambiar [12]. Dado un objeto tridimensional cuyo movimiento no está restringido artificialmente, este tiene tres grados de libertad para sus traslaciones (a lo largo de los ejes xyz) y tres grados de libertad para sus rotaciones (alrededor de los ejes xyz), para un total de seis grados de libertad (Figura 1.7).

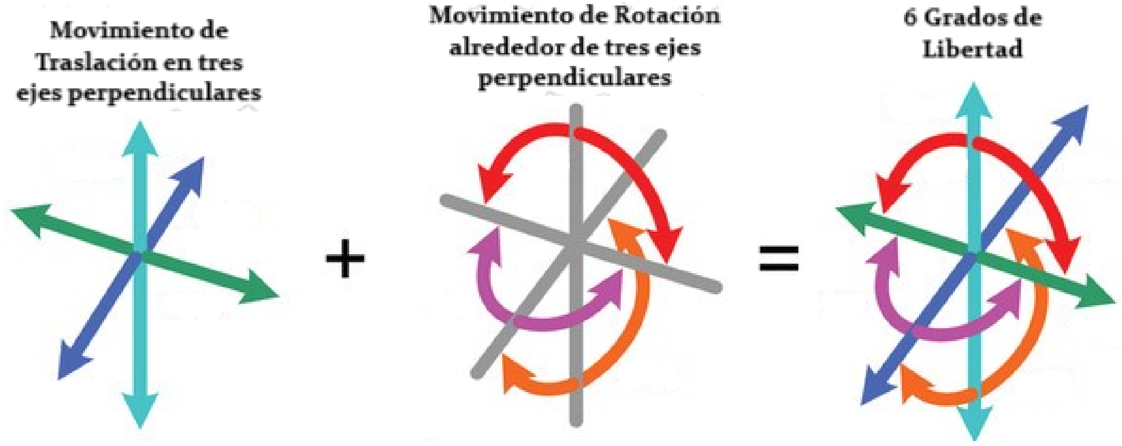


Figura 1.7: Visualización de los 6 posibles grados de libertad (sin restricciones) de un objeto en tres dimensiones.

1.6. Animación

Es el proceso mediante el cual se muestran un grupo de imágenes fijas (ligeramente diferentes entre sí) en orden secuencial y a una velocidad suficiente, para crear la ilusión en los ojos del espectador que algo se esta moviendo. La frecuencia a la que se muestran dichas imágenes consecutivamente, se le conoce como *frame rate*; expresado en cuadros o fotogramas por segundo (FPS o *frames per second*). Figura 1.8

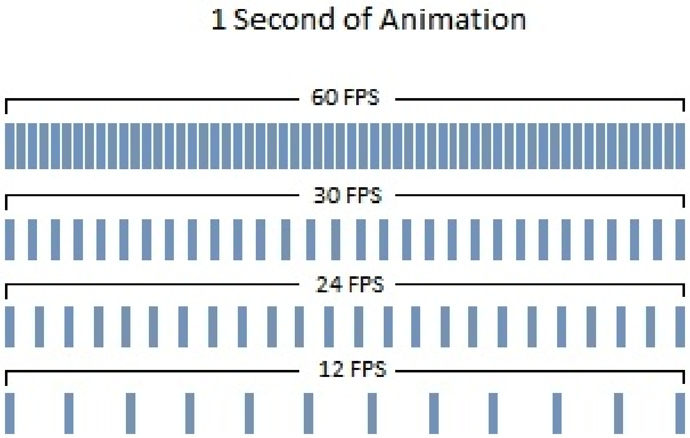


Figura 1.8: Visualización de cantidad de fotogramas en un segundo [6].

1.6.1. Animación de fotogramas clave / *Keyframed Animation*

La forma más simple de animar un objeto es basándose en la noción de que dicho objeto tiene un estado o condición inicial y este cambiará a lo largo del tiempo, en posición, forma, color, luminosidad o cualquier otra propiedad, a alguna forma final diferente. La animación por fotogramas claves adopta el principio de que solo se necesitan mostrar las imágenes/poses/escenas "claves/key" que afectan a la transformación de este objeto, y que todas las demás posiciones intermedias se pueden deducir a partir de estos (Figura 1.9).

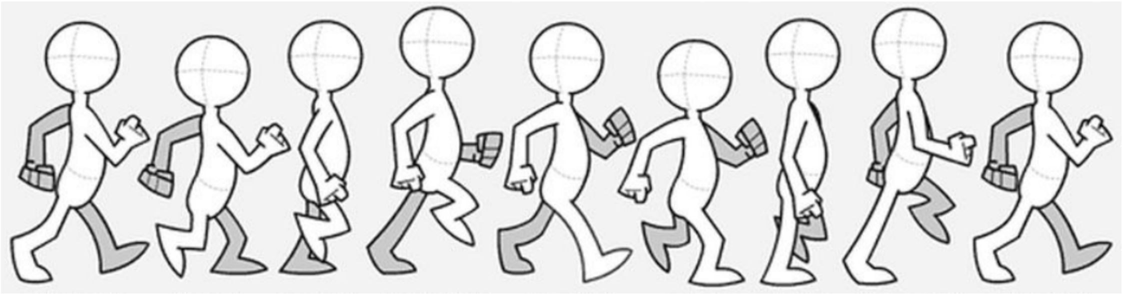


Figura 1.9: Ilustración de fotogramas claves para la animación de un personaje caminando.

El **Fotograma clave / *Keyframe*** es la herramienta más básica de cualquier animador. El fotograma clave es un dibujo o una pose que un animador crea para mostrarse en un cuadro determinado, el cual contienen la información sobre los parámetros requeridos para la transformación de las articulaciones en ciertos puntos discretos en el tiempo. En la animación 3D, un fotograma clave actúa como un marcador colocado en una línea de tiempo, por lo tanto, los fotogramas clave de un animador son las ubicaciones clave de las posturas o posiciones de su objeto/personaje durante una secuencia de movimiento. El computador luego es el encargado de interpolar la posición del objeto/personaje a lo largo del tiempo con los datos contenidos en cada fotograma, completando las secuencias de movimiento con fotogramas intermedios [5].

1.7. Animación Esquelética

Animación esquelética o animación basada en huesos, es una técnica de animación por computadora, en la cual el modelo (personaje u otro objeto articulado) a animar se encuentra compuesto por dos partes principales: a) una representación superficial usada para dibujar el objeto (conocida como la skin/piel o mesh/malla) y b) un conjunto jerárquico de huesos interconectados (conocido como el esqueleto o rig) empleados para animar la skin [7].

La malla triangular continua se vincula a las articulaciones del esqueleto; de tal manera que los vértices de la malla se coordinan para seguir los movimientos de las articulaciones y cada vértice puede ser influenciado/ponderado por múltiples articulaciones a la vez, gracias a lo cual la skin se puede estirar de forma natural a medida que los huesos se mueven [12] (Figura 1.10).

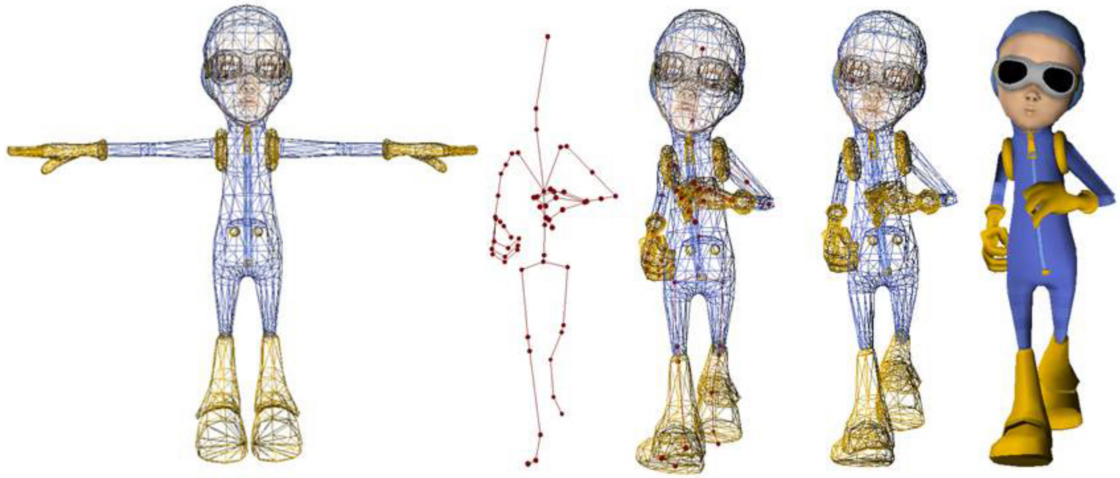


Figura 1.10: Malla vinculada a esqueleto (de izquierda a derecha): 3D mesh en Pose de referencia. Esqueleto. Vértices del modelo ajustados al esqueleto. Modelo renderizado: sin el esqueleto y usando texturas y materiales [2].

1.7.1. Poses

Sin importar que técnica se emplee para producir una animación, cada una de ellas tiene lugar a lo largo del tiempo. La ilusión de movimiento de un personaje se logra al organizar el cuerpo del mismo en una secuencia de poses discretas-estáticas para luego mostrarlas en sucesión rápida (generalmente a una velocidad de 30 o 60 poses por segundo).

La pose de un esqueleto se obtiene al rotar, trasladar y escalar sus articulaciones de maneras arbitrarias. La pose de una articulación esta definida por su posición, orientación y escala relativa a algún un marco de referencia. De esta manera la pose de una articulación usualmente se representa por una matriz 4x4 o 4x3, o por una estructura de datos SQT (S = factor o vector escalar, Q = cuaternión para rotación y T = vector de traslación) [12].

La pose más utilizada para presentar las mallas 3D antes de ser asociadas a un esqueleto es la conocida como Pose de referencia o Bind Pose; caracterizada por mantener las extremidades del modelo alejadas una de las otras y separadas del cuerpo (facilitando el proceso de asociación entre vértices y articulaciones). Figura 1.11

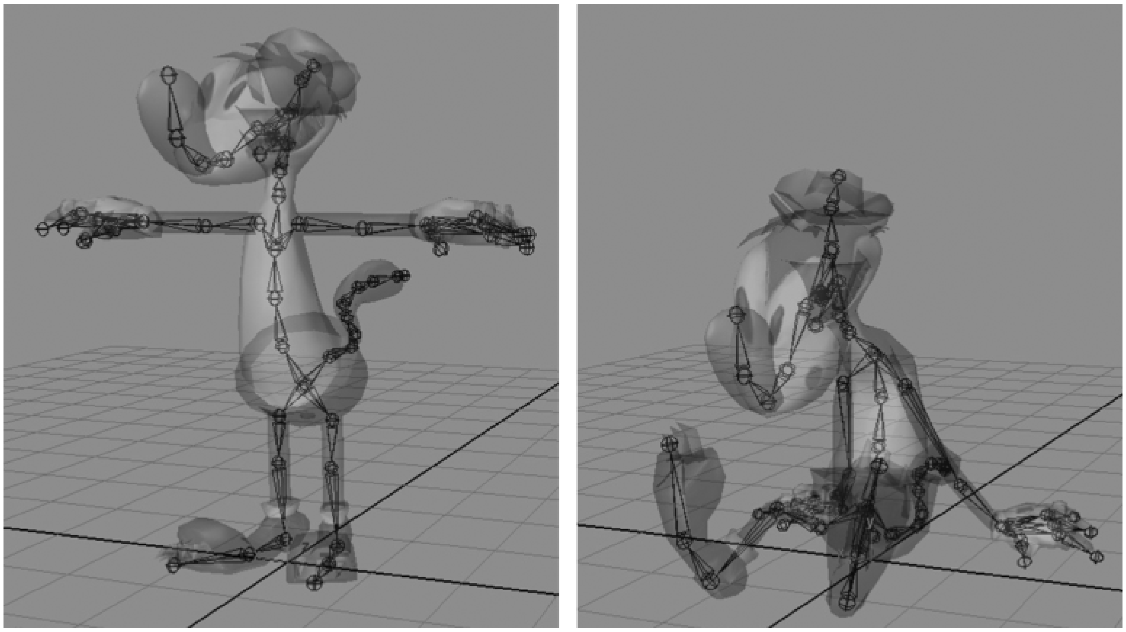


Figura 1.11: Dos poses distintas del mismo esqueleto. La pose de la izquierda es la conocida Bind Pose (Pose-T) [12].

1.8. Animación por Captura de Movimiento (*motion-capture*)

Captura de movimiento se autodefine por su nombre: captura/capture = tomar en posesión, apoderarse, adquirir; y movimiento/motion = el acto de cambiar físicamente de ubicación. Entonces animación por captura de movimiento (*MoCap Animation*), es la técnica que emplea un sistema (tanto hardware como software) para registrar el movimiento de una persona real, para luego transferir los datos obtenidos a un paquete de software de animación en 3D y poder aplicarlos a un personaje digital (Figura 1.12). Hay dos tipos principales de tecnologías utilizadas con este enfoque, los sistemas con y sin marcadores:

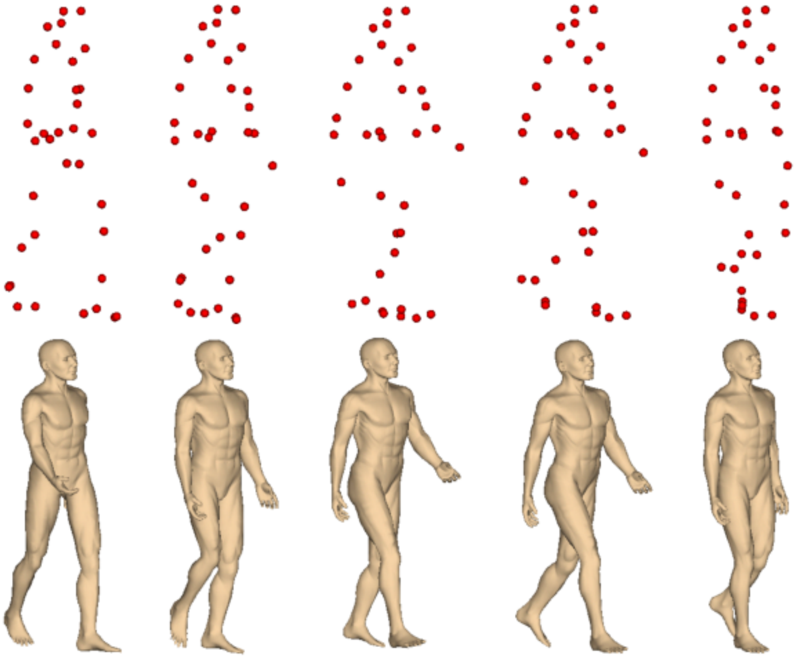


Figura 1.12: Reconstrucción del movimiento de caminar a partir de datos de captura de movimiento (mocap data). (arriba) Datos originales de Mocap. (abajo) Secuencia de animación. [15].

1.8.1. Sistemas con marcadores:

Estos consisten en aplicar un conjunto de marcadores (*markers*) sobre y alrededor de las articulaciones de un actor, para que luego un sistemas de cámaras realice un seguimiento de datos tridimensional, triangulando cada marcador colocado [5]. Triangulación: es el proceso de determinar la ubicación de un punto en espacios tridimensionales midiéndolos desde puntos fijos en el espacio.

Estos marcadores pueden ser pasivos o activos, según el sistema. Los marcadores pasivos están hechos de un material retrorreflectante, el cual refleja la luz devuelta a su punto de origen con muy poca dispersión, por lo que las cámaras suelen tener diodos emisores de luz (LED). Mientras que los marcadores activos son LED que se auto iluminan para que el sistema óptico de la cámara pueda rastrearlos (Figura 1.13).

El sistema de marcadores utiliza un volumen de captura, un espacio virtual tridimensional que se crea colocando varias cámaras alrededor de un espacio de actuación. Se crea un espacio $(0, 0, 0)$ (la ubicación XYZ central para un sistema de coordenadas cartesianas) dentro del software para capturar los marcadores. Luego un artista entra en el volumen de captura y realiza una pose inicial para una captura de marcadores preliminar. El software toma la captura inicial y codifica cada marcador con un nombre para poder rastrearlos a partir de ese momento. Se procede a capturar los movimientos por parte del interprete y el software asigna valores de traslación y rotación de manera continua los cuales pueden ser aplicados sobre el esqueleto de algún personaje posteriormente.

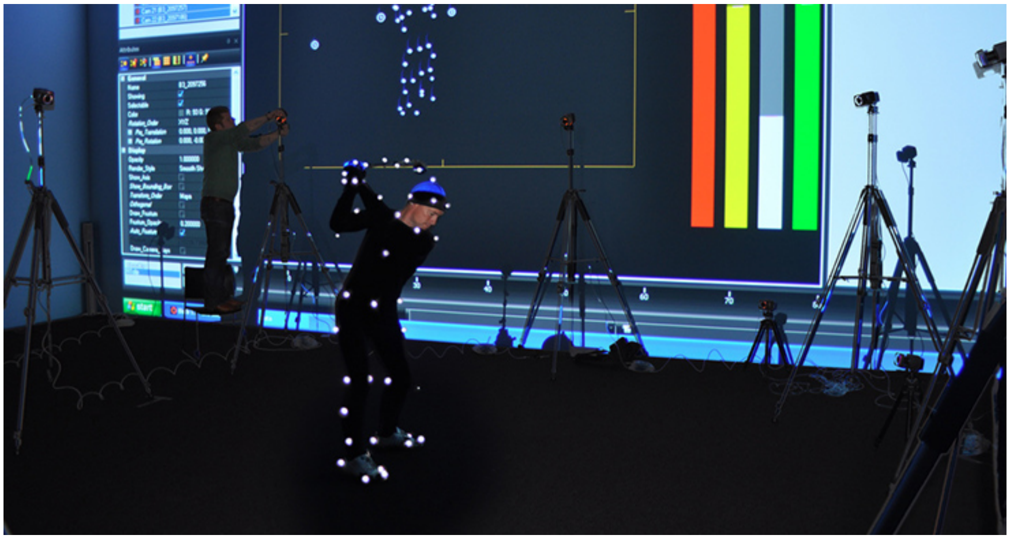


Figura 1.13: Sistema óptico de captura de movimiento con marcadores – PGA Golf Swing Capture. fuente: `https://www.flickr.com/photos/bale_kim/6188385934`

1.8.2. Sistemas sin marcadores:

Los sistemas sin marcadores generalmente permiten al usuario tener más control sobre dónde puede ocurrir la captura, porque el espacio de captura puede ser más grande y no limitarse a un escenario. El sistema sin marcadores típico emplea un traje para la captura, los interpretes usan el traje como una capa exterior robótica, que mide las rotaciones de cada articulación del cuerpo. Estos sistemas pueden usar cámaras, pero a menudo ninguna cámara está involucrada en absoluto, en cambio, el traje puede transmitir datos directamente a una computadora (Figura 1.14). Los trajes proporcionan captura de movimiento que es lo suficientemente precisa para la industria del entretenimiento, pero no se usan comúnmente en aplicaciones médicas o biomecánicas.

Los sistemas de captura sin marcadores basados en cámaras son similares a los sistemas de marcadores, ya que utilizan cámaras calibradas a un punto específico para capturar el movimiento de un intérprete dentro de un volumen. La ventaja de los sistemas sin marcadores de cámara es que los interpretes pueden usar casi cualquier tipo de ropa que deseen ya que no requieren el uso de trajes especiales para rastrear sus movimientos (Figura 1.15). En su lugar se emplean algoritmos especiales de visión por computadora, diseñados para permitir que el sistema analice múltiples flujos de datos ópticos de entrada e identificar formas humanas, descomponiéndolas en partes constituyentes para su seguimiento.

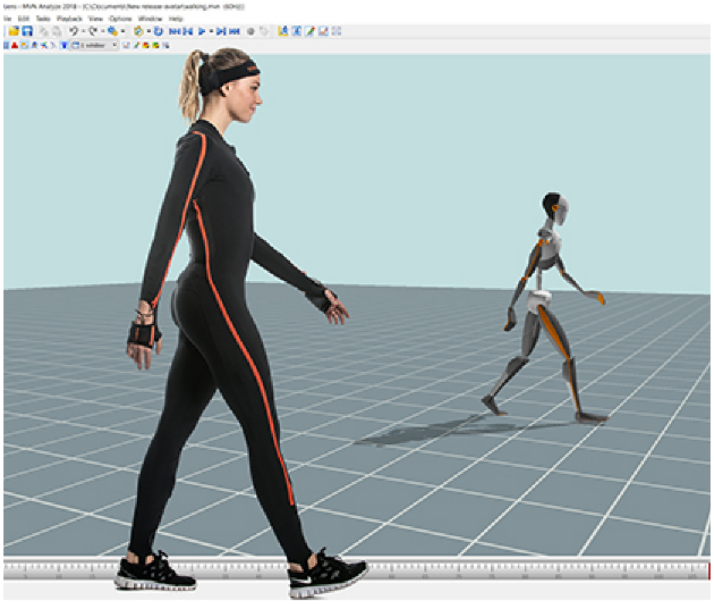


Figura 1.14: Sistema de captura de movimiento sin marcadores – Traje transmite datos directamente a un programa ©Xsens.

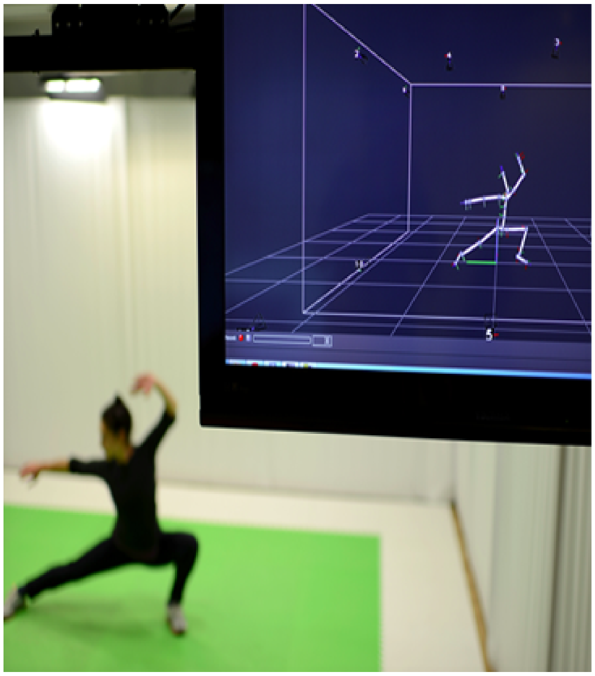


Figura 1.15: Tecnología de visión artificial para identificar y rastrear sujetos sin la necesidad de trajes o marcadores especiales ©Organic Motion, Inc.

1.9. Reorientación de animaciones (*Animation retargeting*)

Las animaciones producidas generalmente solo son compatibles con el único esqueleto para el que fueron hechas. Aunque, se puede hacer una excepción a esta regla para los esqueletos que están estrechamente relacionados. Por ejemplo, si un grupo de esqueletos son idénticos, excepto por una serie de articulaciones hojas (*leaf joints*) opcionales que no afectan a la jerarquía fundamental, una animación creada para uno de estos esqueletos debería funcionar en cualquiera de ellos. El único requisito es que el software utilizado sea capaz de ignorar los canales de animación para las articulaciones que no se pueden encontrar en el esqueleto que se está animando (Figura 1.16).

Figura 1.16: Ilustración de una variedad de personajes animados con movimientos redirigidos. El esqueleto original usado como referencia para la reorientación de las animaciones se encuentra en el centro [11].

1.10. Skinning

Dentro del *pipeline* de creación y animación de personajes, skinning es el proceso mediante el cual se define como los vértices de una superficie geométrica (skin/mesh) de un modelo se deben asociar a las articulaciones de un esqueleto. Con el objetivo de detallar como la malla 3D debería deformarse en cada fotograma de una animación de acuerdo a una función de poses esqueléticas (conjunto estados actuales de las articulaciones) [12].

Las mallas se asocian a los esqueletos por medio de sus vértices y cada vértice puede estar asociado a una o más articulaciones (joints). Si un vértice se asocia a una sola articulación, este sigue el movimiento de dicha articulación exactamente. Si se asocia a dos o más articulaciones, la posición del vértice se convierte en un promedio ponderado de las posiciones que habría asumido si se hubiese unido a cada articulación de forma independiente (Figura 1.17). Es por esto que para asociar una skin a un esqueleto es necesario especificar para cada vértice la siguiente información adicional: el índice o los índices de la(s) articulación(es) a la que está vinculado, y para cada articulación un factor de ponderación (*weighting factor*) que describa cuánta influencia debe tener esa articulación en particular sobre la posición final del vértice (se supone que los factores de ponderación se suman a uno, como es habitual al calcular cualquier promedio ponderado).

Figura 1.17: Visualización del área de influencia que cada hueso de un esqueleto tiene sobre los vértices de una malla. De izquierda a derecha: modelo estático, el modelo con un esqueleto, y el área de influencia para cada hueso. ©Wolfire Games

1.11. Rigging

Toda geometría 3D que se va animar necesita de algún tipo de sistema que proporcione a los animadores el control y la flexibilidad necesaria para mover ese objeto de alguna manera, este sistema de control es conocido como rig (Figura 1.18). De esta forma, *rigging* se define como el proceso mediante el cual se integra la malla/skin con varios mecanismos de animación, envueltos en controladores intuitivos para proveerle a los animadores las herramientas necesarias para poder mover los modelos [5].

El rig de un personaje puede variar desde simple y elegante hasta increíblemente complejo. Una configuración sencilla que permita seleccionar, trasladar y rotar los joints de un esqueleto para crear poses simples se puede llegar a construir en unas horas; mientras que un rig como los utilizados por animadores en películas animadas de alta calidad, podría llegar a requerir días y hasta semanas de trabajo. Esto debido a que no solo ofrecen manipular las extremidades de un esqueleto, sino que además incluyen: la posibilidad de editar pequeños detalles como expresiones faciales y de manos, herramientas para cambiar la forma de las superficies geométricas (deformadores), controles para definir como las mallas se deben conectar al esqueleto (skinning), y hasta mecanismos para agregar/limitar efectos dinámicos adicionales durante secuencias de animación.

En la práctica, este proceso conocido como *rigging* resulta ser costoso y puede llegar a consumir bastante tiempo, debido a que la mayoría de los pasos involucrados se realizan de forma manual o semiautomática. Y estos suelen repetirse varias veces hasta refinar la configuración de los modelos y obtener mejores resultados.

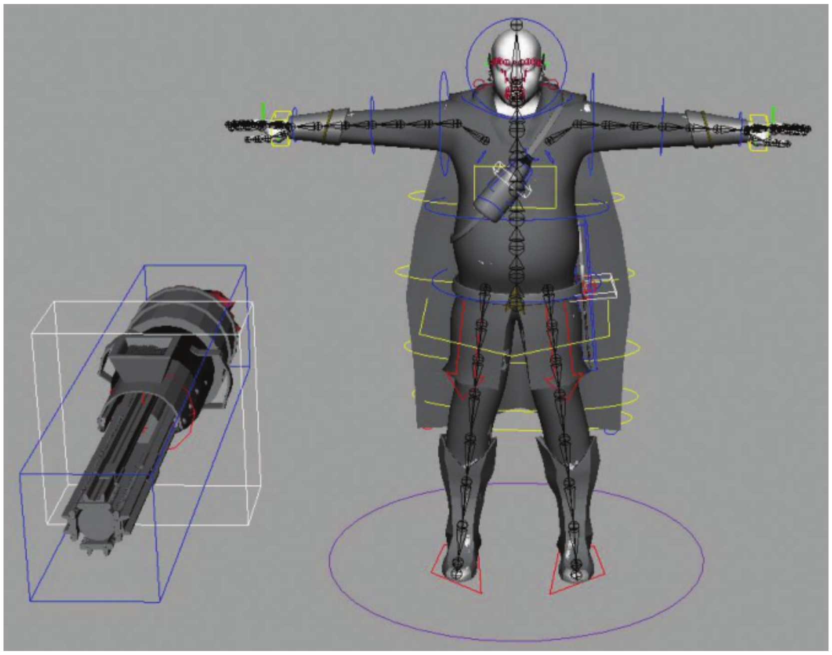


Figura 1.18: El rig para un arma (izquierda) y el rig para un modelo humano (derecha) en Autodesk Maya [5].

1.12. Incrustación de esqueleto (*skeleton embedding*)

Es una técnica que busca automatizar el *rigging* de personajes al crear, cambiar de tamaño y posicionar una estructura esquelética dentro de la superficie geométrica de un modelo, con el objetivo de facilitar su animación mediante el uso de animaciones predefinidas (Figura 1.19).

La incrustación del esqueleto (*skeleton embedding*) puede ser formulada como un problema de optimización: "calcular las posiciones de las articulaciones de tal manera que el esqueleto resultante encaje dentro del personaje lo mejor posible y se parezca tanto como sea posible al esqueleto original dado" [3].

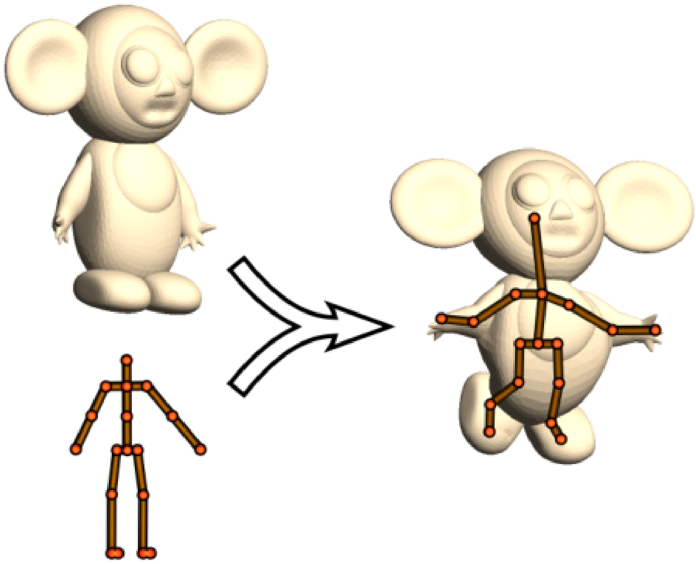


Figura 1.19: Ilustración del método de rigging automático conocido como Incrustación de esqueleto (*skeleton embedding*) utilizado para facilitar animación de personajes [3].

Capítulo 2

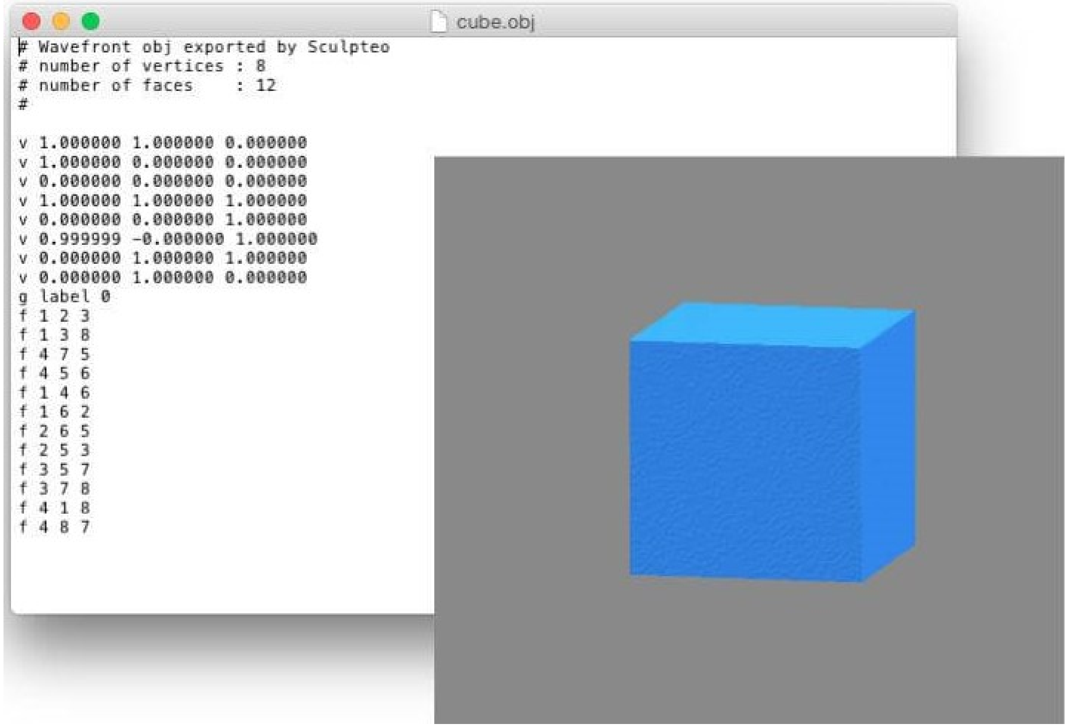
Herramientas y Tecnologías utilizadas

La aplicación desarrollada para el presente trabajo especial de grado fue creada usando ***WebGL y Three.js***. Los modelos/personajes usados dentro de la misma provienen de ***archivos .obj***. Se lleva a cabo un proceso de incrustación de esqueleto dentro de los modelos cargados empleando la ***API de Pinocchio***, compilada usando la cadena de herramientas provista por ***Emscripten***. Y la animación de estos modelos es impulsada por datos de captura de movimiento cargados a partir de ***archivos .bvh***. Este capítulo describe brevemente cada una de los componentes, herramientas y tecnologías previamente mencionadas que fueron fundamentales para la implementación de la aplicación y son necesarias para el funcionamiento de la misma.

2.1. Archivo OBJ

Wavefront OBJect (.obj) es un formato de archivo de definición de geometría de código abierto desarrollado por primera vez por Wavefront Technologies para su paquete de software de gráficos 3D The Advanced Visualizer (TAV), con el objetivo de almacenar objetos geométricos compuestos por líneas, polígonos y curvas/superficies de forma libre.

En pocas palabras, el formato de archivo OBJ almacena información sobre modelos 3D, puede codificar la geometría de la superficie de un modelo 3D y también puede almacenar información de referencia de color y textura. Un archivo OBJ habitualmente es generado mediante un software de Diseño asistido por computadora (*Computer Aided Design* – CAD) como producto final del proceso de modelado 3D y su extensión de archivo correspondiente es simplemente ".obj" (Figura 2.1).



```
# Wavefront obj exported by Sculpteo
# number of vertices : 8
# number of faces    : 12
#

v 1.000000 1.000000 0.000000
v 1.000000 0.000000 0.000000
v 0.000000 0.000000 0.000000
v 1.000000 1.000000 1.000000
v 0.000000 0.000000 1.000000
v 0.999999 -0.000000 1.000000
v 0.000000 1.000000 1.000000
v 0.000000 1.000000 0.000000
g label 0
f 1 2 3
f 1 3 8
f 4 7 5
f 4 5 6
f 1 4 6
f 1 6 2
f 2 6 5
f 2 5 3
f 3 5 7
f 3 7 8
f 4 1 8
f 4 8 7
```

Figura 2.1: Ejemplo de un archivo .obj representando un cubo con un lado de 1 unidad [24].

2.1.1. Especificación del formato de archivo OBJ

Los archivos .obj se codifican en ASCII (formato más común para archivos de texto) y en su forma más simple le permiten a los usuarios definir objetos poligonales (triángulos y cuadriláteros) mediante puntos, líneas y caras; los cuales en conjunto conforman la superficie de los distintos modelos 3D.

La especificación original no especifica cuál debería ser el carácter de fin de línea, por lo que algunos programas de software usan *carriage-returns* (CR – '\r') y algunos usan *linefeeds* (LF – '\n'). Además, el primer carácter de cada línea es muy importante, ya que este especifica el tipo de comando (o palabra clave/keyword) que describe el tipo de datos presentado. Cada línea sigue una estructura básica: el primer carácter es siempre un tipo de comando, seguido por sus argumentos. A continuación, se listan los comandos más utilizados (Todo lo que se muestra entre corchetes es opcional). –La descripción completa del formato de archivo .obj se puede encontrar en el manual del programa Advanced Visualizer de Wavefront [29].

- **El comando de comentario '#':**

  # una línea de comentarios

  El carácter # indica que la línea es un comentario y debe ignorarse. La primera línea suele ser siempre un comentario que especifica qué programa produjo el archivo.

- **El comando de vértice 'v':**

  v x y z

  El comando v define un vértice por sus tres coordenadas cartesianas (x, y, z), y es usado para especificar los vértices de los polígonos que conforman la superficie geométrica de los modelos. Cada vértice tiene un nombre asignado automáticamente dependiendo del orden en que aparezca dentro del archivo; el primer vértice en el archivo recibe el nombre '1', el segundo se llama '2', el tercero '3' y así sucesivamente.

- **El comando de un vector normal 'vn':**

  vn x y z

  Un vector normal es un vector direccional cuyos componentes (x, y, z) son usados generalmente para delimitar una orientación sobre una superficie en el espacio; inicialmente dicho vector no está asociado a ningún punto vértice.

- **El comando de vértice de textura 'vt':**

  vt u v [w]

  Este comando se usa para representar coordenadas de vértices de textura; donde vt es el comando, u es el valor de dirección horizontal y v es el valor de dirección vertical (w sería la dirección perpendicular al plano uv al usar un mapa tridimensional). El orden de los vértices de textura se basa en sus posiciones en el archivo .obj, comenzando con 1.

- **El comando de cara 'f':**

  f v1[/vt1][/vn1] v2[/vt2][/vn2] v3[/vt3][/vn3] ...

  El comando de cara es probablemente el comando más importante, ya que este especifica una cara poligonal hecha de los vértices que vinieron antes de esta línea. Para hacer referencia a un vértice, simplemente se sigue el sistema de numeración implícito de los vértices, por ejemplo, 'f 18 19 21 23' significa una cara poligonal construida a partir de los vértices 18, 19, 21, 23 en orden.

Para cada vértice, se puede asociar un comando vn, que luego asocia ese vector normal al vértice correspondiente; de la misma manera se puede asociar un comando vt a un vértice, que determinará la asignación de texturas a usar en ese punto. Si se especifican comandos vt o vn para un vértice, se deben especificar para todos.

- **El comando de grupo 'g':**

  g nombre

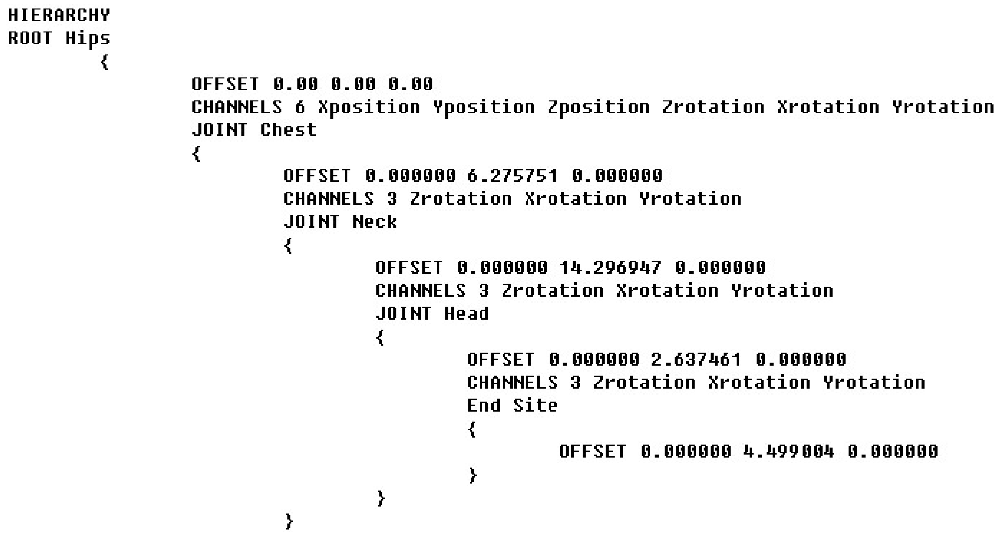
  El comando g se usa para definir una agrupación de sub-objetos; todos los comandos f (de caras) que le siguen estarán en el mismo grupo. Esto es útil si se quiere reutilizar alguna información en particular, como el tipo de material empleado (propiedades de color y sombreado de una superficie).

2.2. Archivo BVH

Los datos de captura de movimiento son una representación bidimensional de movimientos que tienen lugar en una mundo tridimensional durante un período de tiempo dado, y una de las formas más populares para almacenar dichos datos es mediante el formato de archivo Biovision Hierarchy (BVH). Desarrollado originalmente por Biovision, una compañía de servicios de captura de movimiento, como una forma de proporcionar a sus clientes información organizada sobre estructuras esqueléticas junto a datos de movimiento.

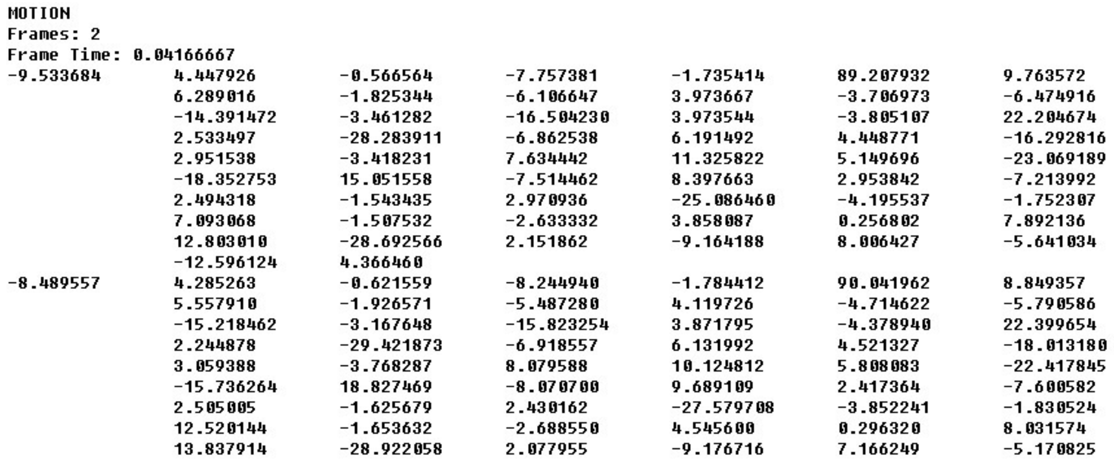
2.2.1. Especificación del formato de archivo BVH

El formato BVH es un archivo de texto ASCII simple con extensión ".bvh" que consta de dos partes: una sección de encabezado donde se describe la jerarquía de articulaciones de un esqueleto y se proporcionan las especificaciones necesarias para establecer una postura inicial del mismo (Figura 2.2); y una sección de datos que contiene una secuencia de periodos de tiempo detallando diferentes especificaciones para poses posteriores del esqueleto (Figura 2.3).



```
HIERARCHY
ROOT Hips
        {
                OFFSET 0.00 0.00 0.00
                CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
                JOINT Chest
                {
                        OFFSET 0.000000 6.275751 0.000000
                        CHANNELS 3 Zrotation Xrotation Yrotation
                        JOINT Neck
                        {
                                OFFSET 0.000000 14.296947 0.000000
                                CHANNELS 3 Zrotation Xrotation Yrotation
                                JOINT Head
                                {
                                        OFFSET 0.000000 2.637461 0.000000
                                        CHANNELS 3 Zrotation Xrotation Yrotation
                                        End Site
                                        {
                                                OFFSET 0.000000 4.499004 0.000000
                                        }
                                }
                        }
```

Figura 2.2: Ejemplo de la primera parte de un archivo .bvh detallando la jerarquía y pose inicial de un esqueleto (sección de encabezado) [19]. La posición de cada articulación viene dada por tres valores flotantes que especifican su ubicación relativa en relación a la articulación padre de cada una de estas.



```
MOTION
Frames: 2
Frame Time: 0.04166667
-9.533684     4.447926      -0.566564     -7.757381     -1.735414     89.207932     9.763572
              6.289016      -1.825344     -6.106647     3.973667      -3.706973     -6.474916
              -14.391472    -3.461282     -16.504230    3.973544      -3.805107     22.204674
              2.533497      -28.283911    -6.862538     6.191492      4.448771      -16.292816
              2.951538      -3.418231     7.634442      11.325822     5.149696      -23.069189
              -18.352753    15.051558     -7.514462     8.397663      2.953842      -7.213992
              2.494318      -1.543435     2.970936      -25.086460    -4.195537     -1.752307
              7.093068      -1.507532     -2.633332     3.858087      0.256802      7.892136
              12.803010     -28.692566    2.151862      -9.164188     8.006427      -5.641034
              -12.596124    4.366460
-8.489557     4.285263      -0.621559     -8.244940     -1.784412     90.041962     8.849357
              5.557910      -1.926571     -5.487280     4.119726      -4.714622     -5.790586
              -15.218462    -3.167648     -15.823254    3.871795      -4.378940     22.399654
              2.244878      -29.421873    -6.918557     6.131992      4.521327      -18.013180
              3.059388      -3.768287     8.079588      10.124812     5.808083      -22.417845
              -15.736264    18.827469     -8.070700     9.689109      2.417364      -7.600582
              2.505005      -1.625679     2.430162      -27.579708    -3.852241     -1.830524
              12.520144     -1.653632     -2.688550     4.545600      0.296320      8.031574
              13.837914     -28.922058    2.077955      -9.176716     7.166249      -5.170825
```

Figura 2.3: Ejemplo de la segunda parte de un archivo .bvh describiendo secuencias de poses de un esqueleto (sección de datos de movimiento) [19]. Primero se especifica el numero de fotogramas (FRAMES); segundo la tasa de muestreo por segundo (FRAME TIME); y tercero, para cada fotograma, una línea de valores flotantes que proporcionan las coordenadas de traslación y/o rotación que se procesarán de acuerdo con las especificaciones de los canales/grados de libertad (CHANNELS) definidos en la sección de encabezado.

- **Sección de encabezado:** (Figura 2.2)

  La sección jerárquica del archivo comienza con la palabra clave “HIERARCHY”, seguida en la línea siguiente con la palabra clave “ROOT” y el nombre de la articulación que es la raíz de la jerarquía del esqueleto. La palabra clave “ROOT” indica el inicio de una nueva estructura esquelética jerárquica y aunque un archivo .bvh es capaz de contener muchos esqueletos, lo normal es tener solo un esqueleto definido por archivo.

  El resto de la estructura del esqueleto se define de manera recursiva donde la definición de cada articulación/hueso, incluyendo las articulaciones hijos, se encapsula en llaves {} delimitadas en la línea anterior con la palabra clave “JOINT” (o ROOT en el caso de la articulación raíz) seguida por el nombre de la articulación/hueso. Con la introducción de una llave izquierda { , es una buena practica indentar el contenido de la articulación a definir y alinear la llave de cierre } con el de apertura correspondiente; a pesar de que las indentaciones en distintos niveles de la jerarquía no son absolutamente necesarios, si ayudan a que el archivo sea más legible para los humanos.

  Dentro de la definición de cada articulación, la primera línea, delimitada por la palabra clave “OFFSET”, detalla la traslación del origen de la articulación con respecto al origen de su articulación padre (o globalmente en el caso de la articulación raíz) a lo largo de los ejes xyz respectivamente. Este desplazamiento además define implícitamente la longitud y dirección de los huesos dentro del esqueleto.

  La segunda línea de la definición de una articulación está prefijado con la palabra clave “CHANNELS” seguido de un número que define los canales/grados de libertad (*degrees of freedom* – DOF) para la articulación actual y luego una lista de esa cantidad de etiquetas que indican el tipo de cada canal. La importancia del orden en que se presentan los canales es relevante de dos maneras distintas: En primer lugar, el orden en que se ve cada canal en la sección de jerarquía del archivo coincide exactamente con el orden de los datos en la sección de movimiento del archivo; Y segundo, cada canal representa una coordenada de un ángulo de Euler (conjunto de tres coordenadas angulares que sirven para especificar la orientación de un sistema de referencia de ejes ortogonales) por lo que su orden de concatenación es fundamental al momento de crear las matrices de rotación para cada articulación y poder producir animaciones visualmente correctas.

  Después de las líneas OFFSET y CHANNEL, las siguientes líneas no anidadas en la definición de los huesos, se usan para definir articulaciones hijos, comenzando con la palabra clave “JOINT”, sin embargo, en el caso de las articulaciones hojas (*leaf joints*) de la jerarquía, se usa una etiqueta especial, “End Site”, que encapsula otro trío de números prefijados por la palabra clave “OFFSET” usados para inferir la longitud y orientación del último hueso en una rama de la jerarquía.

- **Sección de datos de movimiento:** (Figura 2.3)

  Una vez que se ha definido la jerarquía esquelética, la segunda sección de un archivo BVH, que se denota con la palabra clave “MOTION”, contiene el número de fotogramas en la animación, la velocidad de fotogramas y los datos para cada uno de los canales (grados de libertad) de las articulaciones en el esqueleto.

  La línea que contiene el número de fotogramas (o muestras de movimiento) comienza con la palabra clave “Frames:” seguida de un entero positivo que es el número de fotogramas. La velocidad de fotogramas (o tasa de muestreo de los datos) está en la siguiente línea que comienza con la frase clave “Frame Time:” seguido de un decimal positivo que representa la duración de un solo fotograma; para obtener el numero de fotogramas por segundo, solo se necesita dividir 1 entre la velocidad de fotogramas. Después de que se han definido el número de fotogramas y su respectivo tiempo, el resto del archivo contiene los datos de los canales para cada articulación en el orden en que fueron especificados al momento de definir la jerarquía esquelética, donde cada línea de valores decimales representa un fotograma de animación.

2.3. WebGL

Como describe Parisi [22], la Web Graphics Library (WebGL) es una interfaz de programación de aplicaciones (API) multiplataforma, libre de regalías (royalty-free) para renderizar gráficos 2D/3D, basada en OpenGL ES [14] que se ejecuta en la web como un contexto de dibujo dentro del elemento <canvas>de HTML5 (HyperText Markup Language) con integración completa con las interfaces de modelo de objetos del documento (DOM) de bajo nivel. WebGL es también una API que utiliza el lenguaje de sombreado de OpenGL, GLSL ES, que permite que el código se ejecute en la GPU de una computadora. Además, WebGL como un estándar web puede ser combinado sin problemas con otro contenido web que se encuentre superpuesto en la parte superior o inferior del contenido gráfico y está integrado en los principales navegadores web como Safari, Google Chrome, Mozilla Firefox; es ideal para la creación de aplicaciones web dinámicas con gráficos 2D/3D escritas en el lenguaje de programación JavaScript. Lanzado inicialmente el 3 de marzo de 2011, fue diseñado y es mantenido por el consorcio industrial sin fines de lucro Khronos Group.

Esta definición comprende varias ideas centrales:

- **WebGL es una API:** WebGL se accede exclusivamente a través de un conjunto de interfaces de programación en JavaScript; ; no hay etiquetas acompañantes como las hay en HTML. El renderizado de gráficos 3D en WebGL es análogo a los dibujos 2D utilizando el elemento <canvas>, en ambas todo se hace a través de llamadas a la API usando JavaScript. De hecho, el acceso a WebGL se proporciona a través del elemento Canvas existente y a través de un contexto de dibujo especial específico para WebGL.
- **WebGL está basado en OpenGL ES 2.0:** OpenGL ES es una adaptación de las especificaciones estándar OpenGL para el renderizado de gráficos 2D y 3D establecidas desde hace mucho tiempo. El ES significa "embedded systems" (sistemas embebidos). lo que significa que ha sido diseñado para su uso en pequeños dispositivos informáticos, especialmente teléfonos y tabletas. Los diseñadores de WebGL consideraron que basar la API en la pequeña huella de OpenGL ES facilitaría la entrega de una API 3D consistente y multiplataforma para la Web.
- **WebGL se combina con otro contenido web:** WebGL se superpone encima o debajo del otro contenido de una página web. El elemento <canvas>con el contenido gráfico 2D/3D puede ocupar solo una parte de la página o toda la página, y puede residir dentro de las etiquetas <div>que están ordenas por sus coordenadas de profundidad (z-buffer). Esto significa que los gráficos son desarrollados usando WebGL, pero todos los otros elementos de una página son construidos con HTML básico. Además, el navegador compone/combina todos los gráficos de la página para que el usuario tenga una experiencia fluida.
- **WebGL está diseñada para aplicaciones web dinámicas:** WebGL fue creada con su entrega y presentación en la web como un aspecto principal. WebGL comienza con OpenGL ES, pero ha sido adaptada con características específicas que se integran bien con los navegadores web, trabajan con el lenguaje JavaScript y son amigables para la entrega web.
- **WebGL es multiplataforma:** WebGL es capaz de ejecutarse en cualquier sistema operativo, en dispositivos que van desde teléfonos y tabletas hasta computadoras de escritorio.

- **WebGL es libre de regalías:** Como todas las especificaciones web abiertas, WebGL es de uso libre y gratuito.

2.3.1. Estructura de las aplicaciones WebGL

Páginas web dinámicas se pueden crear mediante el uso de una combinación de HTML/CSS y JavaScript. Con la introducción de WebGL, el lenguaje de sombreado GLSL ES debe agregarse a la mezcla, lo que significa que las páginas web que utilizan WebGL se crean utilizando tres idiomas: HTML5 (lenguaje de marcado de hipertexto), JavaScript y GLSL ES. En la Figura 2.4 se muestra la arquitectura de software de las páginas web dinámicas tradicionales (lado izquierdo) y las páginas web que utilizan WebGL (lado derecho).

Sin embargo, dado que GLSL ES generalmente está escrito en JavaScript, solo los archivos HTML y JavaScript son realmente necesarios para las aplicaciones WebGL. Por lo tanto, aunque WebGL agrega complejidad al JavaScript, este conserva la misma estructura que las páginas web dinámicas estándar, solo con archivos HTML y JavaScript. En el caso de las hojas de estilo en cascada – CSS (*Cascading Style Sheets*), cuando el objetivo es desarrollar una aplicación que ocupe toda la página web, usualmente no es necesario hacer nada sofisticado con respecto al diseño de la misma; basta con asegurar que el elemento HTML <body>ocupe toda la pantalla y que el elemento <canvas>dentro de este llene todo el espacio de su cuerpo (<canvas>es el contenedor que permite la generación de gráficos dinámicamente por medio de scripting JavaScript).

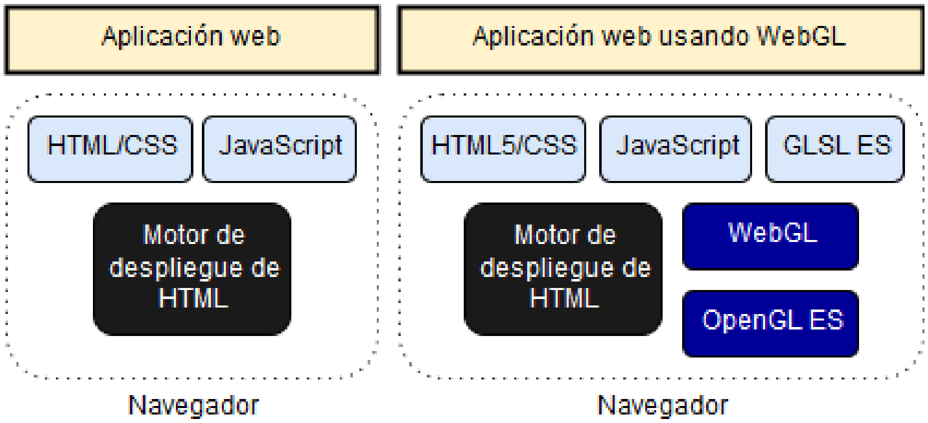


Figura 2.4: Diferencia entre la arquitectura de software de páginas web dinámicas tradicionales (izquierda) y páginas web usando WebGL (derecha) [18].

Para renderizar WebGL en una página, una aplicación debe, como mínimo, realizar los siguientes pasos:

- Crear un elemento canvas.
- Obtener un contexto de dibujo para el canvas.
- Inicializar la ventana gráfica, los límites rectangulares de dónde dibujar (viewport).
- Crear uno o más buffers que contengan los datos que se van a renderizar (típicamente vértices).
- Crear una o más matrices para definir las transformaciones necesarias para llevar los vértices almacenados en los buffers al espacio de coordenadas de la ventana.
- Crear uno o más sombreadores (shaders) para implementar el algoritmo de dibujo.
- Inicializar los sombreadores (shaders) con sus respectivos parámetros.
- Dibujar.

WebGL es una API de dibujo de bajo nivel: no hay representación DOM de la escena 3D, no hay formatos de archivo 3D compatibles de forma nativa para cargar geometría y animaciones, y con la excepción de algunos eventos de sistema de bajo nivel, no hay un modelo de eventos incorporado para informar de los acontecimientos que ocurren dentro del canvas-lienzo de dibujo 2D/3D (ejemplo: no hay eventos de click-de-mouse que indiquen en qué objeto se hizo click). Esta falta de construcciones de alto nivel causa que las aplicaciones requieran una gran cantidad de código para su creación y funcionamiento; a pesar de ofrecerle a los programadores un alto control sobre el rendimiento y el conjunto de características de sus aplicaciones, que les permiten experimentar y desarrollar gráficos complejos. Es por esto que el uso de herramientas construidas sobre WebGL que brindan acceso de alto nivel a la API ahorran mucho tiempo de trabajo.

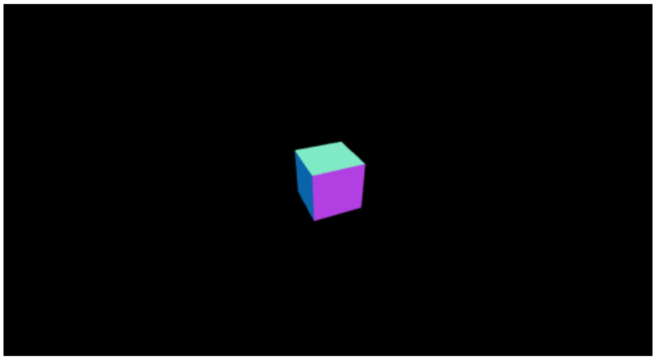
2.4. Three.js

Three.js es una biblioteca (library) de código abierto escrita en JavaScript que permite a sus usuarios crear y presentar escenas de gráficos y animaciones 3D directamente dentro de un navegador web [9]. Three.js proporciona una extensa API fácil de usar con un amplio conjunto de funciones que le permiten a los desarrolladores abstraerse de las llamadas de API de bajo nivel requeridas al desarrollar aplicaciones WebGL; facilitando y haciendo más productivos los procesos de creación-manipulación de objetos 3D y escenas de gráficos sin tener que saber demasiado sobre WebGL.

La Figura 2.5 sirve como ejemplo de la alta funcionalidad ofrecida por Three.js. El código mostrado inicializa una escena gráfica completa en three.js con su respectiva cámara y renderizador WebGL; a su vez, crea un cubo geométrico de color, lo coloca dentro de la escena y lo hace girar en cada llamada de renderizado. Todo esto dentro de unas pocas líneas de código relativamente sencillas; el mismo ejemplo de haberse realizado directamente con la API WebGL hubiese requerido mucho más esfuerzo y literalmente cientas de líneas de código.

```
var camera, scene, renderer;
var geometry, material, mesh;

init();
animate();

function init() {

        camera = new THREE.PerspectiveCamera( 70, window.innerWidth / window.innerHeight, 0.01, 10 );
        camera.position.z = 1;

        scene = new THREE.Scene();

        geometry = new THREE.BoxGeometry( 0.2, 0.2, 0.2 );
        material = new THREE.MeshNormalMaterial();

        mesh = new THREE.Mesh( geometry, material );
        scene.add( mesh );

        renderer = new THREE.WebGLRenderer( { antialias: true } );
        renderer.setSize( window.innerWidth, window.innerHeight );
        document.body.appendChild( renderer.domElement );

}

function animate() {

        requestAnimationFrame( animate );

        mesh.rotation.x += 0.01;
        mesh.rotation.y += 0.02;

        renderer.render( scene, camera );

}
```

(a) Código three.js que inicializa una escena gráfica completa con su respectiva cámara y renderizador WebGL. Además crea un cubo geométrico, lo coloca dentro de la escena y lo hace girar.



(b) Cubo girando dentro de la escena gráfica creada

Figura 2.5: Código para crear una escena básica en three.js: los componentes básicos –la escena, –la cámara y –el renderizador son el núcleo de cualquier aplicación three.js. Se necesita una función de renderizado que invoque repetidamente *renderer.render(scene,camera)* para hacer que aparezca cualquier objeto agregado a la escena, en este caso, un cubo simple. El método *requestAnimationFrame(animate)* llama de manera continua a la función de renderizado para mantener el ciclo de dibujo. El código es tomado directamente de la página oficial de github de three.js [20].

2.4.1. Descripción General de Three.js

Three.js fue creado por Ricardo Cabello (conocido por su nombre de usuario Mr.doob) y la primera versión fue lanzada en 2010. Es de código abierto, con su código fuente alojado en GitHub (`https://github.com/mrdoob/three.js/`), con varios autores que contribuyen al proyecto; se incluye dentro de una página web al enlazar una copia local o remota. Three.js ha crecido en potencia, y se ha convertido en la opción más popular para crear aplicaciones 3D en WebGL, por ser la biblioteca WebGL más completa desde el punto de vista de características ofrecidas (la mayor parte del gran contenido WebGL que puede verse en línea ha sido creado utilizando three.js)[22]. Mucha información sobre Three.js puede encontrarse en línea en su sitio web principal `https://threejs.org/`, aquí hay un resumen de lo que tiene para ofrecer:

- **Three.js oculta los detalles de bajo nivel del proceso de renderizado de WebGL:** Three.js abstrae los detalles de la API de WebGL, representando la escena 3D mediante el uso de mallas, materiales y luces (es decir, los tipos de objetos con los que normalmente trabajan los programadores gráficos).
- **Three.js es potente:** Three.js es más que una simple interfaz extra (*wrapper*) alrededor de WebGL; ya que además de incluir las capacidades del paquete principal, contiene muchos objetos precompilados de gran utilidad para el desarrollo de juegos, animaciones, presentaciones, visualización de datos, aplicaciones de modelado y efectos especiales de postprocesamiento.
- **Three.js es fácil de usar:** La API de Three.js fue diseñada para ser amigable y de fácil aprendizaje; la biblioteca viene con muchos ejemplos que pueden ser utilizados como punto de partida.
- **Three.js es rápido:** Three.js emplea las mejores prácticas de gráficos 3D para mantener un alto rendimiento sin sacrificar la usabilidad.
- **Three.js es robusto** Existen extensas comprobaciones de errores, excepciones y advertencias de consola para mantener al desarrollador informado y fuera de problemas.
- **Three.js apoya la interacción:** WebGL no proporciona soporte nativo para la selección, es decir, saber cuándo el puntero del mouse se encuentra sobre un objeto. Three.js ayuda con la selección, lo que facilita agregar interactividad a sus aplicaciones.
- **Three.js facilita los cálculos matemáticos:** Three.js tiene objetos potentes y fáciles de usar para matemáticas 3D, como matrices, proyecciones y vectores.

- **Three.js incorpora soporte para distintos tipos de formato de archivo:** Permite la carga de archivos en formatos de texto exportados por los populares paquetes de modelado 3D; también ofrece formatos JSON y binarios específicos a Three.js.
- **Three.js está orientado a objetos:** Los programadores trabajan con objetos JavaScript de primera clase en lugar de simplemente hacer llamadas a funciones de JavaScript.
- **Three.js es extensible:** Es bastante fácil agregar funciones y personalizar Three.js. Si el programador no ve el tipo de datos que necesita, este puede diseñarlo, crearlo y conectarlo.
- **Three.js también se renderiza en 2D-Canvas, SVG y CSS:** A pesar de lo popular que se ha vuelto WebGL, todavía no se utiliza en todas partes, o puede que no sea la mejor opción para algunas aplicaciones. Es por esto que Three.js puede renderizar la mayoría del contenido gráfico en un elemento canvas-2D o un elemento SVG (Gráficos Vectoriales Escalables, es un formato de imagen vectorial basado en XML). Esto puede ser particularmente útil si el contexto del elemento canvas-3D no estuviese disponible, permitiendo que el código tenga opciones alternativas para su ejecución. Three.js también se puede usar para renderizar y transformar elementos CSS (*Cascading Style Sheets*).

2.5. Emscripten

Emscripten [30] es un compilador de código abierto que toma bitcode LLVM (*Low Level Virtual Machine*) y lo convierte en JavaScript, que puede ejecutarse en la web (o en cualquier otro lugar que pueda ejecutar JavaScript). -Página principal del proyecto: `http://emscripten.org/`

El proyecto de infraestructura de compiladores LLVM es una colección de herramientas y tecnologías de compilación modulares y reutilizables [13] [16] enfocadas principalmente en los lenguajes de programación C, C++ y Objective-C. Dichos lenguajes son compilados a través de una interfaz (*front-end*) de compilador (los principales son Clang y LLVM-GCC) en la representación intermedia (*intermediate representation* - IR) de LLVM, la cual puede ser código de bits (*bitcode*) para máquinas o un lenguaje ensamblador legible por humanos; y luego los pasa a través del motor (*back-end*) del compilador encargado de generar el código de máquina real para una arquitectura particular. Aquí es donde Emscripten desempeña el papel de motor del compilador (*back-end*) y convierte el bitcode de representación intermedia en un código JavaScript de bajo nivel.

Al utilizar Emscripten, código escrito en otros lenguajes de programación distintos de JavaScript pueden ser ejecutados en la web usando uno de los siguientes métodos [30]:

1. Compilando el código directamente en la representación intermedia de LLVM, para luego compilarlo en JavaScript usando Emscripten. Este enfoque funciona para los lenguajes reconocidos por alguna de las interfaces (front-end del compilador) LLVM existentes, como es el caso para C/C++.

2. Compilando en LLVM toda la biblioteca de tiempo de ejecución (*runtime library*) utilizada para analizar y ejecutar código de un lenguaje de programación particular, para después compilarla en JavaScript usando Emscripten, y finalmente utilizar la biblioteca de tiempo de ejecución (*runtime*) compilada para ejecutar el código escrito en ese lenguaje en la web. Este es un enfoque útil si el runtime de un lenguaje de programación está escrito en otro lenguaje para el cual existe una interfaz de LLVM, pero el lenguaje en sí no tiene esa interfaz. Por ejemplo, dado el caso del lenguaje de programación Python, es posible compilar CPython (la implementación estándar de Python escrita en C) en JavaScript y luego ejecutar código Python dentro del runtime compilado de CPython en la web.

2.5.1. Cadena de herramientas (*Toolchain*) de Emscripten

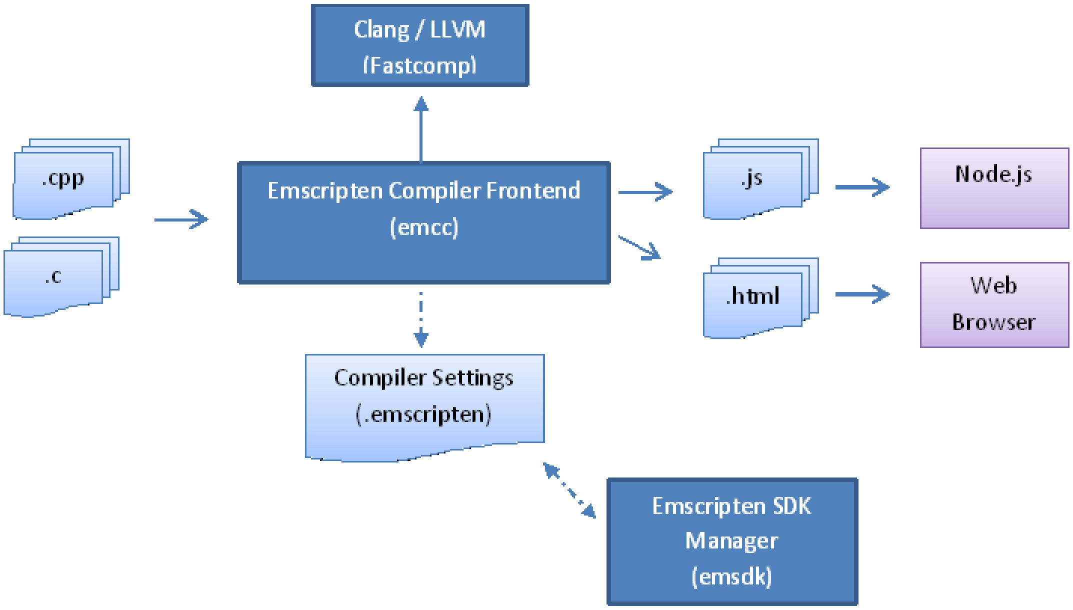


Figura 2.6: Visualización de alto nivel de la cadena de herramientas utilizadas por Emscripten para generar código Javascript a partir de archivos C/C ++. Fuente: `https://kripken.github.io/emscripten-site/docs/introducing_emscripten/about_emscripten.html`

Una vista de alto nivel de la cadena de herramientas presentes en Emscripten puede ser visualizada en la Figura 2.6. Donde la herramienta principal es la interfaz del compilador de Emscripten (emcc - *Emscripten Compiler Frontend*), el cual es un reemplazo directo para un compilador estándar como GCC (*GNU Compiler Collection*).

Emcc usa el compilador Clang para convertir archivos C/C++ a código de bits LLVM (bitcode), y emplea Fastcomp (Núcleo del compilador de Emscripten - LLVM backend) para compilar el código de bits a JavaScript. Los archivos Javascript (.js) de salida pueden ejecutarse desde HTML dentro de un navegador web, o mediante cualquier otro entorno de ejecución (run-time environment) que ejecute JavaScript, por ejemplo Node.js.

El Kit de desarrollo de software (SDK - *software development kit*) de Emscripten (emsdk), se usa para administrar múltiples SDK y herramientas, y para especificar el SDK/conjunto de herramientas particulares que se van a utilizar en un momento dado para compilar código (Active Tool/SDK). Además permite la "instalación" (descargar y construir/*build*) de la última cadena de herramientas (*toolchain*) disponible en su repositorio Github: `https://github.com/kripken/emscripten`.

Emsdk escribe la configuración "activa" en el archivo de configuración del compilador Emscripten (.emscripten). Mientras que Emcc utiliza dicho archivo para obtener/verificar la cadena de herramientas actual correcta antes de llevar a cabo sus operaciones.

Una serie de otras herramientas no se muestran — por ejemplo, Embind la cual es parte del toolchain de Emscripten y es usada para para enlazar (*bind*) funciones y clases de C++ con JavaScript, de modo que el código compilado se puede utilizar de forma "natural" mediante el uso de Invocación de funciones (*calls*) y objetos (*Objects*) estándar dentro del código Javascript.

Toda la cadena de herramientas (*toolchain*) se entrega en el SDK de Emscripten y se puede utilizar en Windows, Linux o MacOS.

2.6. Pinocchio API

Hay dos formas de generar un esqueleto para un modelo 3D de manera automática: una se llama extracción de esqueleto (*skeleton extraction*) y la otra incrustación de esqueleto (*skeleton embedding*); a pesar de que ambas buscan construir el esqueleto basándose en la información provista por el propio modelo, cuando el propósito de su uso es la animación automática de personajes Baran y Popović [3] señalan que la incrustación de esqueleto es mucho más adecuada que la extracción. La principal ventaja de la incrustación (*embedding*) sobre la extracción es que el esqueleto dado proporciona información sobre la estructura esperada del personaje, que puede ser difícil de obtener a partir de sólo la superficie geométrica (malla). Por ejemplo, el usuario puede tener datos de movimiento para un esqueleto cuadrúpedo, pero para un personaje cuadrúpedo complicado, es probable que el esqueleto extraído resultante tenga una topología diferente.

Ilya Baran y Jovan Popović [3] presentaron un método para facilitar los procesos de rigging y animación automática de personajes. Dada una malla 3D única y un esqueleto de entrada, su método adapta dicho esqueleto dentro de la superficie geométrica (incrustación de esqueleto) y después de asociarlos permite que el uso de datos de movimiento esqueléticos animen el personaje. Su implementación de este proceso fue presentada en un sistema llamado Pinocchio, cuyo trabajo de investigación (*paper*) y video de presentación pueden ser vistos y descargados en: `http://www.mit.edu/~ibaran/autorig/`. A su vez, el código fuente de la biblioteca de Pinocchio (*Pinocchio library*) para el rigging automático de personajes puede ser descargado en: `http://www.mit.edu/~ibaran/autorig/pinocchio.html`.

2.6.1. Proceso de incrustación de esqueleto - Pinocchio API

El método de incrustación de esqueleto cambia el tamaño y posición de un esqueleto dado de tal manera que este quepa dentro de la superficie geométrica de un personaje. Esto se puede formular como un problema de optimización: "calcular las posiciones de las articulaciones de tal manera que el esqueleto resultante encaje dentro del personaje lo mejor posible y se parezca tanto como sea posible al esqueleto original dado" [3]. Para un esqueleto con s articulaciones (siendo s un número natural positivo, representando la cantidad de vértices en la estructura esquelética de árbol - incluyendo las hojas), este es un problema tridimensional con una función objetivo complicada. Resolver este problema directamente usando optimización continua no es factible.

Por lo tanto, Pinocho discretiza el problema mediante la construcción de un grafo cuyos vértices representan las posibles posiciones de las articulaciones (*joints*) y cuyas aristas (*edges*) son segmentos de huesos potenciales. Esto es un desafío porque el grafo debe tener pocos vértices y aristas, y sin embargo, debe capturar todos los posibles enlaces para los huesos dentro del personaje. El grafo se construye empaquetando esferas centradas en la superficie media aproximada [1] dentro del personaje y conectando los centros de estas esferas con las aristas del grafo. Luego, Pinocchio encuentra la incrustación óptima del esqueleto en este grafo con respecto a una función de penalización discreta [4]; y utiliza la solución discreta como punto de partida para la optimización continua.

Pinocchio recibe dos parámetros de entrada: Una malla 3D y un esqueleto; y mediante el proceso de incrustación calcula las posiciones de las articulaciones del esqueleto dentro del personaje. Una visualización de este proceso y los módulos de la API pueden ser observados en la Figura 2.7. A continuación una explicación más detallada de los pasos

---

[1] El eje medio (medial axis) de un objeto 2D se define como el conjunto de todos los puntos que tienen más de un punto más cercano al borde del objeto, en otras palabras, se define como el lugar geométrico dado por los centros de todos los discos internos de un objeto que tocan el borde de dicho objeto en dos o más puntos [27]; también referido como el esqueleto topológico de una figura (Ver Figura 2.10). Para un objeto 3D, se usan esferas en lugar de discos, y el conjunto de puntos - centros de estas esferas resulta en la superficie media (*medial surface*) del objeto. Para figuras discretizadas, el eje/superficie media se aproxima como un subconjunto de píxeles/voxels, y esta aproximación se conoce como el eje/superficie media discreta.

involucrados [3]:

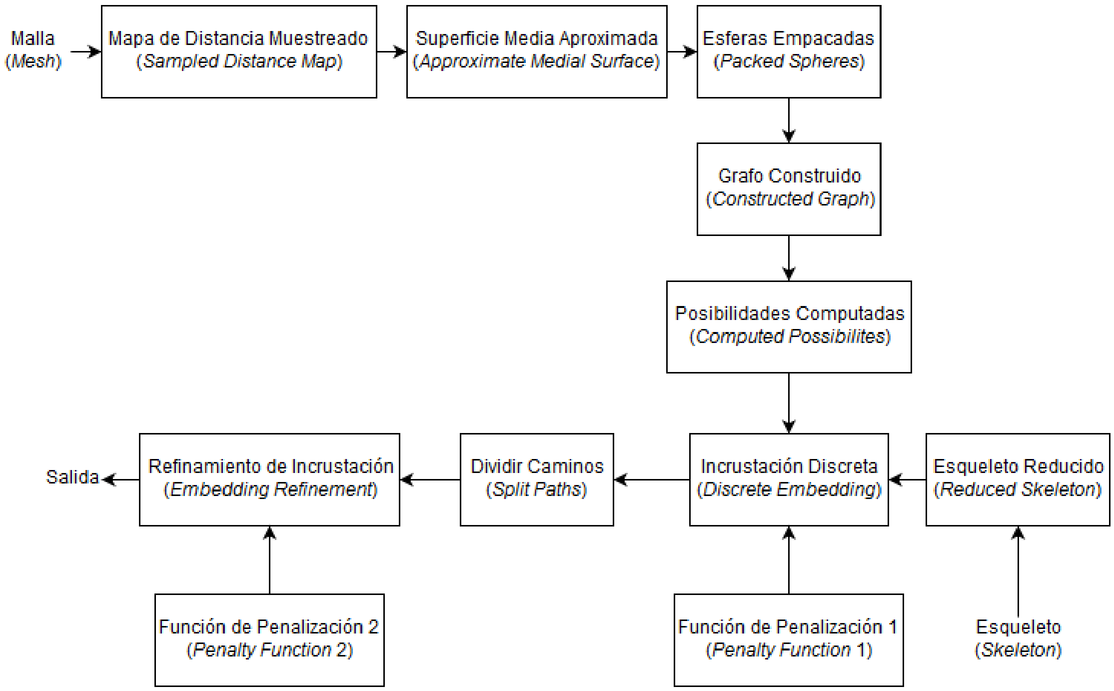


Figura 2.7: Visualización gráfica de los módulos involucrados dentro de la API Pinocchio para el proceso de incrustación de esqueleto. Parámetros de entrada: malla 3D y un esqueleto. Salida: posiciones de las articulaciones del esqueleto dentro del personaje. [28].

2.6.1.1. Discretización

Antes de realizar cualquier otro cálculo, Pinocchio re-escala la malla 3D de tal manera que esta quepa dentro de un cubo unitario alineado con los ejes de coordenadas del espacio (*axis-aligned*); luego construye un árbol-kd (kd-tree [2]) sobre la malla para evaluar la distancia exacta con signo a la superficie desde un punto arbitrario. De esta forma, todos los cálculos realizados y las tolerancias son relativas al tamaño del personaje.

- **Campo de Distancia** (*Distance Field*) Para aproximar la superficie media y facilitar otros cálculos, Pinocchio calcula un campo de distancia con signo muestreado adaptativamente con interpolación trilinear en un octree (*trilinearly interpolated adaptively sampled signed distance field on an octree*) [10].

---

[2] En ciencias de la computación, un Árbol kd (abreviatura de árbol k-dimensional) es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de k dimensiones. Los árboles kd son un caso especial de los árboles BSP (*Binary space partitioning* / Partición Binaria del Espacio).

Un campo/mapa de distancia es un campo escalar que especifica la distancia mínima a una forma, donde la distancia puede ser positiva o negativa (con signo/signed) para distinguir entre el interior y el exterior de la figura. En otras palabras, un campo/mapa de distancia es la salida de una cuadrícula de puntos discretos con cada punto marcado como un punto de característica (*feature point* - celda que delimita el borde de una figura, en el caso 3D la superficie de una malla) o un punto de fondo (*background point*); se llevan a cabo cálculos de manera iterativa a partir de los puntos de fondo más cercanos a los puntos característica y se continua hasta que se han calculado todos los puntos de fondo (Figura 2.8), los cálculos realizados varían dependiendo de la distinción que se quiere establecer entre los puntos (la distancia euclidiana es utilizada cuando se desea representar la distancia entre dos vértices en el espacio).

El campo de distancia es una representación efectiva para formas. Sin embargo, los campos de distancia muestreados regularmente tienen inconvenientes debido a su tamaño y resolución limitada. Debido a que los detalles finos requieren un muestreo denso, se necesitan inmensos volúmenes para representar con precisión los campos de distancia clásicos con un muestreo regular cuando se presenta cualquier detalle fino, incluso cuando los detalles finos ocupan solo una pequeña fracción del volumen. Para superar esta limitación se emplean los mapas/campos de distancia adaptativa (*Adaptive Distance Field* - ADF), los cuales usan muestreo adaptativo, dirigido a detalles, con altas tasas de muestreo en regiones donde el campo de distancia contiene detalles finos y bajas tasas de muestreo donde el campo varía suavemente. El muestreo adaptativo permite una precisión arbitraria en el campo reconstruido junto con un uso eficiente de memoria (ver comparación en la Figura 2.9). Para procesar los datos muestreados adaptativamente de manera más eficiente, los ADF almacenan los datos muestreados en una jerarquía (árbol octal/*octree*) para una rápida localización. La combinación de muestreo dirigido al detalle y el uso de una jerarquía espacial para el almacenamiento de datos junto a un método para reconstruir el campo de distancia subyacente (Interpolación trilineal para reconstrucción y estimación de gradientes) a partir de los valores muestreados.

Con estos principios Pinocchio utiliza un enfoque de arriba-abajo (*Top-down approach*) para dividir iterativamente cada celda del octree hasta que caen dentro de la tolerancia de la distancia ($\theta = 0.003$), y debido a que solo las distancias negativas (es decir, desde los puntos dentro del personaje) son importantes, Pinocho no divide las celdas que tienen la garantía de no intersecar el interior del personaje.

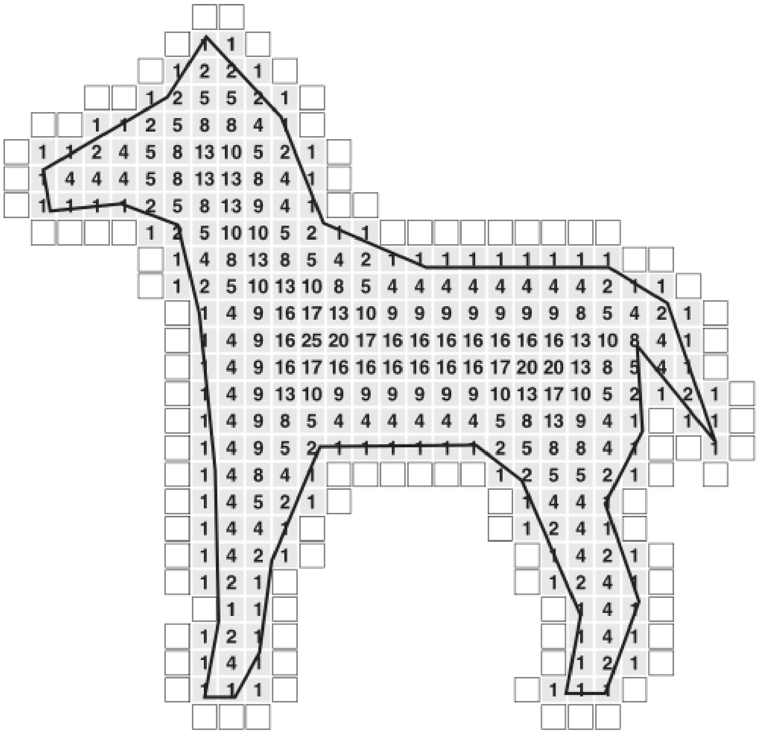


Figura 2.8: El campo/mapa de distancia euclidiana 2D (*Euclidean distance map* - EDM) para un polígono discreto con forma de animal. Las cuadrículas blancas representan puntos de características, que son puntos de margen en este caso. Y las cuadrículas con dígitos representan los puntos de fondo, el valor de cada celda es el cuadrado de la distancia euclidiana a la celda exterior más cercana [27].

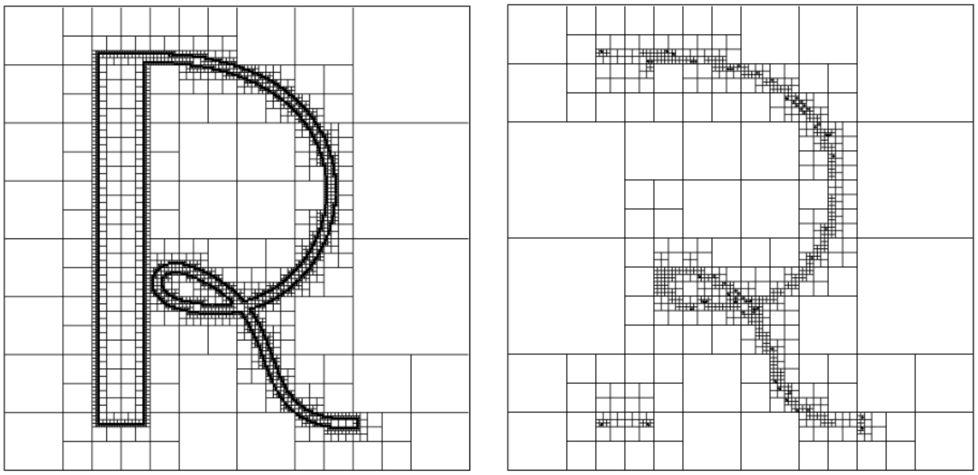


Figura 2.9: Comparación entre la cantidad de celdas requeridas para almacenar una figura "R" en quadtrees (árbol cuaternario) empleando campos/mapas de distancia con muestreos regular y adaptativos. Con muestreo regular el quadtree contiene 23573 celdas mientras que al emplear un campo de distancia adaptativa solo se emplean 1713 [10]

- **Superficie Media Aproximada** (*Approximate Medial Surface*) Pinocho utiliza el campo de distancia adaptativa para calcular una muestra de puntos aproximadamente en la superficie media (Figura 2.12). La superficie media es el conjunto de discontinuidades C1[3] del campo de distancia. Dentro de una sola celda del octree, el campo de distancia interpolada está garantizado ser C1, por lo que es necesario mirar solo los limites de las celdas (*cell boundaries*) [3]. Por lo tanto, Pinocho atraviesa el octree y, para cada celda, observa una cuadrícula (de espaciado θ) de puntos en cada cara de la celda. Luego calcula los vectores gradiente para las celdas adyacentes a cada punto de la cuadrícula -si el ángulo entre dos de ellos es 120º o mayor, agrega el punto a la muestra de la superficie media. La condición de 120º se impone para evitar las partes "ruidosas" de la superficie media- se quieren los puntos donde es probable que se encuentren las articulaciones esqueléticas. Por la misma razón, Pinocchio filtra los puntos muestreados que están demasiado cerca de la superficie del personaje (dentro de 2θ).

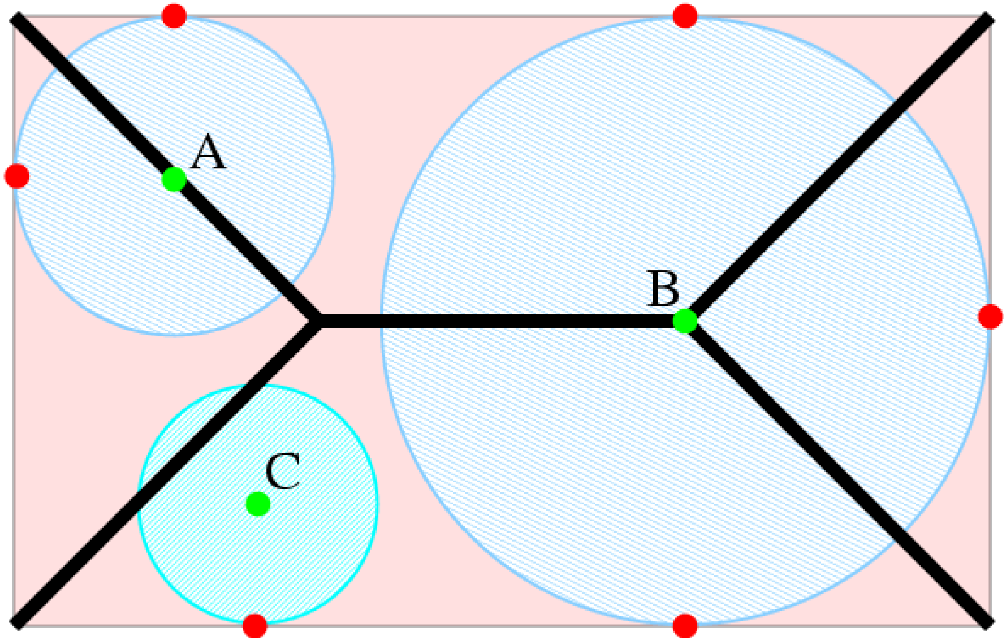


Figura 2.10: Esqueleto Topológico de un rectángulo. Los puntos A y B pertenecen al eje medio (medial axis) del objeto, mientras que el punto C no pertenece al esqueleto (El esqueleto está marcado por segmentos de líneas negras gruesas.) . Fuente: `http://www.inf.u-szeged.hu/~palagyi/skel/skel.html`

---

[3] Una curva con continuidad es una curva suave y sin pliegues ni roturas. Una curva con discontinuidad es una curva que tiene dobleces y se rompe. Una discontinuidad C0 es una ruptura real en la curva, y una discontinuidad C1 es un cambio en la tangente (una torcedura en la curva); ver ejemplos en la Figura 2.11

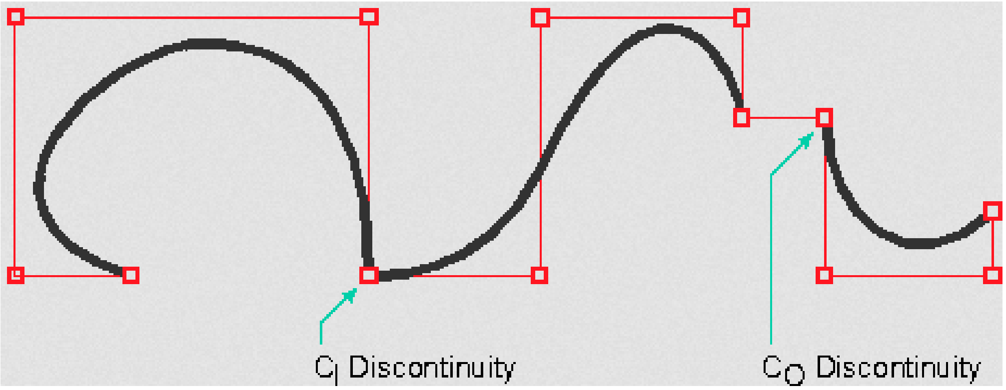


Figura 2.11: Una curva con discontinuidad es una curva que tiene dobleces y se rompe. Una discontinuidad C0 es una ruptura real en la curva, y una discontinuidad C1 es un cambio en la tangente (una torcedura en la curva) . Fuente: `http://www.inf.u-szeged.hu/~palagyi/skel/skel.html`

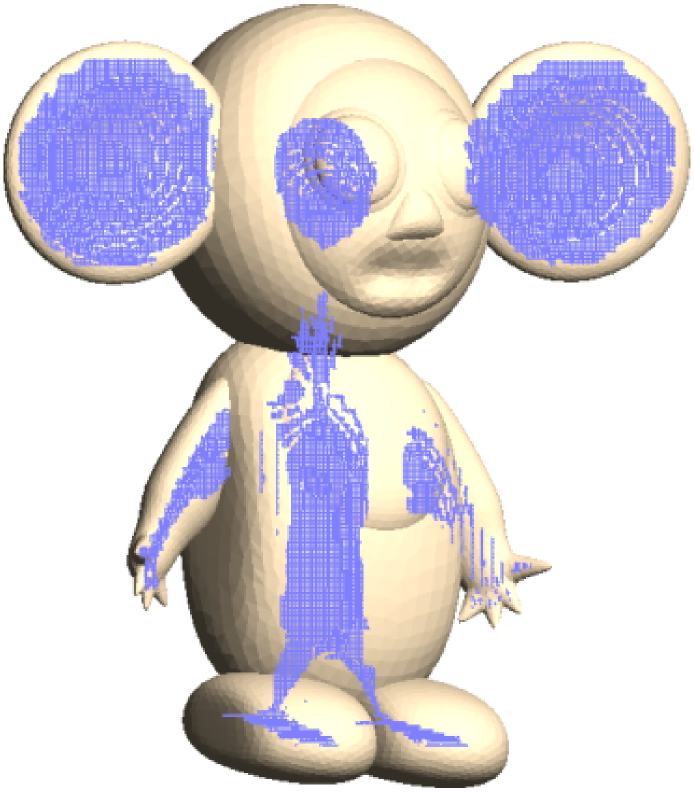


Figura 2.12: Visualización de la superficie media aproximada (*Approximate Medial Surface*) calculada por Pinocchio para una malla 3D [3].

- **Empaquetado de Esferas** (*Sphere Packing*) Para seleccionar los vértices del grafo de la superficie media, Pinocho empaqueta esferas dentro del personaje de la siguiente manera:

ordena los puntos de la superficie media por su distancia a la superficie de la malla (los que están más alejados de la superficie son los primeros). Luego procesa estos puntos en orden y si un punto está fuera de todas las esferas agregadas previamente, agrega la esfera centrada en ese punto cuyo radio es la distancia a la superficie. En otras palabras, las esferas más grandes se agregan primero, y ninguna esfera contiene el centro de otra esfera (Figura 2.13). Aunque el procedimiento descrito anteriormente toma $\mathcal{O}(nb)$ tiempo en el peor de los casos (donde n es el número de puntos, y b es el número final de esferas insertadas), el comportamiento del peor de los casos rara vez se ve porque la mayoría de los puntos se procesan mientras hay una pequeño número de grandes esferas.

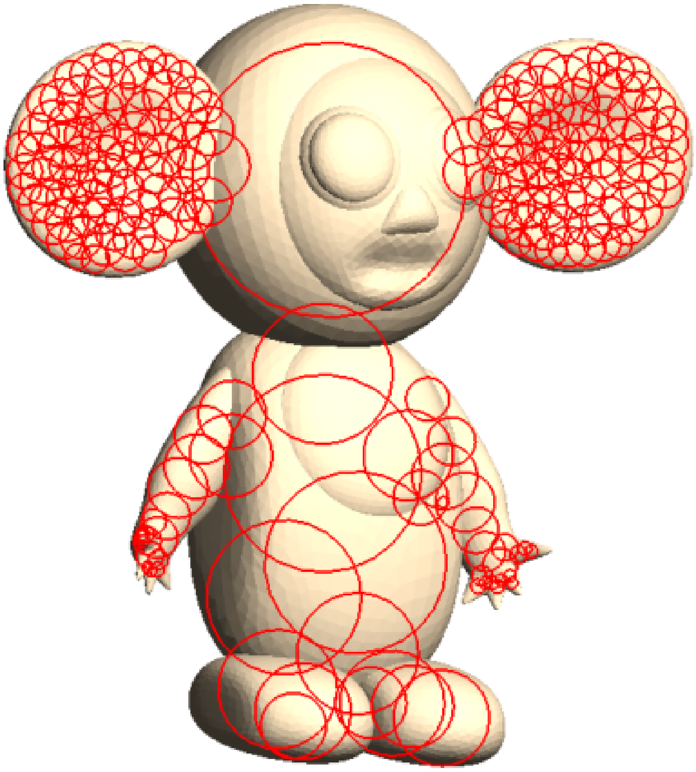


Figura 2.13: Visualización de las esferas empaquetadas (*Packed Spheres*) por Pinocchio dentro una superficie geométrica (malla) [3].

- **Construcción del grafo** (*Graph Construction*) El último paso del proceso de discretización construye los bordes (aristas) del grafo al conectar algunos pares de centros de esferas (Figura 2.14). Pinocchio agrega un borde entre dos centros de esfera si las esferas se intersecan; también agrega bordes entre esferas que no se intersecan si esa arista está bien dentro de la superficie y si resulta ser "esencial" para la interconexión del esqueleto. Por ejemplo, las esferas del cuello y del hombro izquierdo del personaje de la Figura 2.13 son disjuntas, pero aún así debe haber un borde entre ellas. La condición precisa que utiliza Pinocchio para construir un borde es que la distancia desde cualquier punto de dicho borde a la superficie debe ser al menos la mitad del radio de la esfera más pequeña, y la esfera más cercana que se centra al punto medio del borde debe corresponder a los puntos finales del mismo. Además, es en este paso de discretización que Pinocchio pre-calcula los caminos más cortos entre todos los pares de vértices en este grafo para acelerar la evaluación de la función de penalización durante el proceso de incrustación.

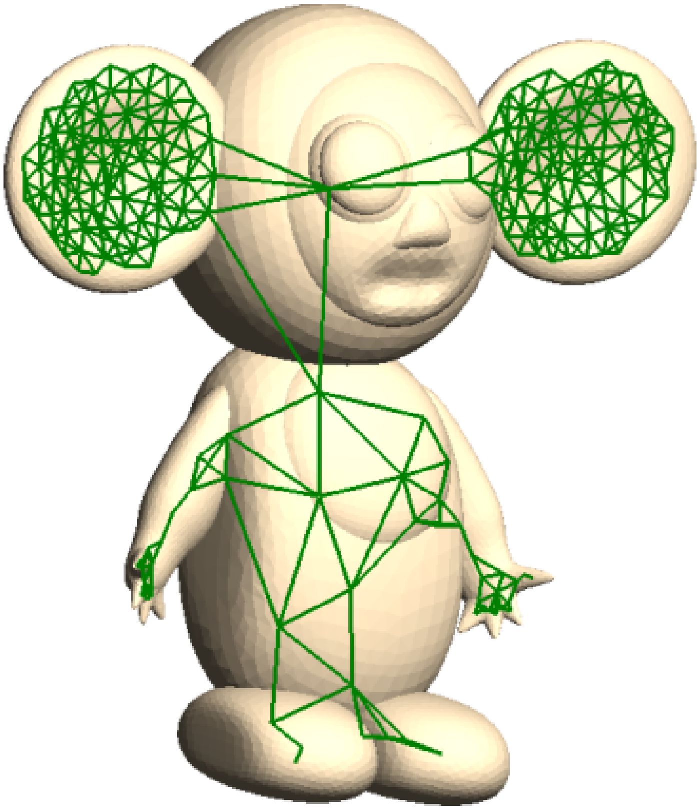


Figura 2.14: Visualización del grafo construido por Pinocchio dentro de una malla durante el proceso de discretización. [3].

2.6.1.2. Esqueleto reducido

La etapa de discretización construye un grafo geométrico $G = (V, E)$, donde el grafo G es un par ordenado que comprende un conjunto V de vértices, nodos o puntos junto con un conjunto E de bordes, aristas o líneas, que son subconjuntos de 2 elementos de V; dentro de dicho grafo es donde Pinocchio necesita incrustar el esqueleto dado de una manera óptima. El esqueleto se da como un árbol con raíz (*rooted tree* [4]) en s articulaciones. Para reducir los grados de libertad, para la incrustación discreta, Pinocchio trabaja con un esqueleto reducido, en el que todas las cadenas de huesos se han combinado (todas las articulaciones de grado dos, como las rodillas, eliminadas), como se muestra en la Figura 2.15. El esqueleto reducido tiene solo r articulaciones. Esto funciona porque una vez Pinocchio sabe dónde están los puntos finales de una cadena de huesos en V, puede calcular las articulaciones intermedias tomando el camino más corto entre los puntos finales y dividiéndolo de acuerdo con las proporciones del esqueleto no reducido. Para un esqueleto humanoide se usan $s = 18$, pero $r = 7$ (Ver Figura 2.16); sin un esqueleto reducido, el problema de optimización normalmente sería intratable.

Por lo tanto, el problema de incrustación de esqueleto discreto es encontrar la incrustación del esqueleto reducido en G, representado por una r-tupla $v = (v_1, \ldots, v_r)$ de vértices en V, la que minimiza una función de penalización $f(v)$ que está diseñada para penalizar las diferencias en el esqueleto incrustado del esqueleto original dado.

---

[4] Árbol con raíz (o enraizado): Un árbol con raíz es un árbol en el cual un vértice ha sido designado como la raíz y cada arista es dirigida desde la raíz.

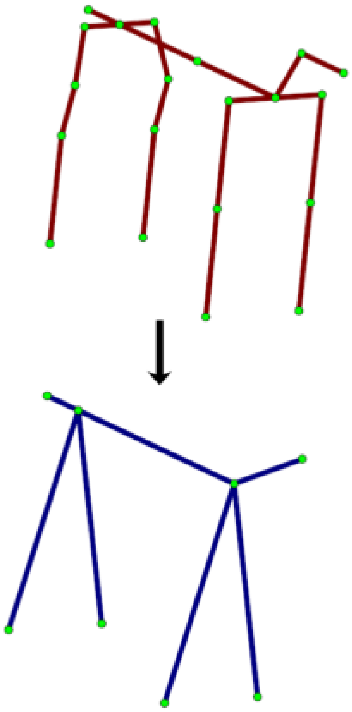


Figura 2.15: Ejemplo gráfico de un esqueleto cuadrúpedo reducido en comparación a su esqueleto original. [3].

Articulación del Esqueleto	#Original	#Reducido
shoulders	0	0
back	1	
hips	2	1
head	3	2
Lthigh	4	
Lknee	5	
Lankle	6	
Lfoot	7	3
Rthigh	8	
Rknee	9	
Rankle	10	
Rfoot	11	4
Lshoulder	12	
Lelbow	13	
Lhand	14	5
Rshoulder	15	
Relbow	16	
Rhand	17	6

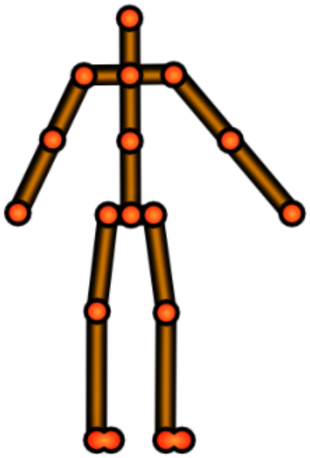


Figura 2.16: Nombres y números generados en Pinocchio para las articulaciones del esqueleto original y el esqueleto reducido; junto a una visualización gráfica del esqueleto original y sus articulaciones.

2.6.1.3. Función de penalización discreta

La contribución más importante que Baran y Popović hicieron a Pinocchio es la función de penalización discreta, la cual tiene un gran impacto en la generalidad y la calidad de los resultados de incrustación. Una correcta incrustación de esqueleto debe tener las proporciones, orientación de los huesos y tamaño similar al esqueleto original dado. Para facilitar esto, las articulaciones del esqueleto se suministran con alguno de los siguientes atributos (sin sacrificar demasiada generalidad) para mejorar la calidad y rendimiento de los cálculos involucrados para la incrustación:

- Una articulación puede estar marcada como “ symmetric” (simétrica) con respecto a otra articulación (articulaciones de los brazos y piernas). Esto da como resultado penalizaciones de simetría si la distancia entre la articulación y su padre difiere de la distancia entre la articulación simétrica y su padre.
- Una articulación puede estar marcada como “foot” (pie). Esto resulta en una penalización si la articulación no está en la posición más baja del personaje.
- Una articulación se puede marcar como “fat” (gorda). Esto restringe la posible colocación de la articulación en el centro de las esferas más grandes; en un esqueleto bípedo estas articulaciones vendrían siendo las ubicadas en la cabeza, las caderas y entre los hombros.

La función de penalización discreta mide la calidad de un esqueleto reducido incrustado dentro del volumen de modelos discretizados. La función de penalización f se representa como una combinación lineal de k funciones de penalización de “base”, las cuales se combinan de acuerdo a un conjunto de pesos de influencia específicos para equilibrarlas entre sí. Baran y Popović estudiaron esta función de penalización a través de más de 400 incrustaciones para garantizar la generalidad y correctitud de los coeficientes que aparecen en estas funciones [3] [4].

Pinocchio usa $k = 9$ funciones de penalización básicas que son:

- Penaliza los huesos cortos. Penaliza un hueso en el esqueleto reducido si la longitud es demasiado corta en comparación con la del esqueleto no reducido.
- Penaliza la orientación entre las articulaciones si la dirección entre las dos articulaciones es diferente de la del esqueleto original.
- Penaliza las diferencias en longitud entre los huesos que están marcados como simétricos en el esqueleto.
- Penaliza las cadenas de huesos que comparten vértices. Si dos o más cadenas de huesos comparten un vértice con una distancia a la superficie de la malla inferior a 0.2 Ô una cadena de huesos más corta se superpone con una más larga, entonces se aplicará una penalización a estas articulaciones.
- Penaliza las articulaciones que están marcadas como pies si no están en la posición

más baja posible.

- Penaliza las cadenas de huesos de longitud cero. Penalización para cuando una articulación y su padre están incrustados en el mismo vértice.
- Penaliza los segmentos de huesos que están orientados incorrectamente en relación con los huesos dados.
- Penaliza las articulaciones de grado uno que deberían estar más alejadas de sus articulaciones padres pero que no lo están.
- Penaliza las articulaciones que están incrustadas una cerca de la otra en el grafo, pero están muy lejos una de la otra a lo largo de los caminos de los huesos.

2.6.1.4. Incrustación discreta

El cálculo de una incrustación discreta que minimice una función de penalización general es intratable porque hay exponencialmente muchas incrustaciones. Sin embargo, si es fácil estimar un buen límite inferior en la función de penalización f a partir de una incrustación parcial (de las primeras articulaciones), entonces es posible aplicar un método de ramificación y acotación (*branch-and-bound method*) para resolver este problema.

- **Método de Ramificación y Acotación** (también llamado Ramificación y poda) es un paradigma de diseño de algoritmos para problemas de optimización discreta y combinatoria, así como optimización matemática. Este método consiste en una enumeración sistemática de soluciones candidatas por medio de la búsqueda en el espacio de estados: se considera que el conjunto de soluciones candidatas forma un árbol con el conjunto completo en la raíz. El algoritmo explora las ramas de este árbol, que representan subconjuntos del conjunto de soluciones. Antes de enumerar las soluciones candidatas de una rama, la rama se compara con los límites estimados superior e inferior de la solución óptima, y se descarta si no puede producir una solución mejor que la mejor encontrada hasta el momento por el algoritmo. El algoritmo depende de la estimación eficiente de los límites inferior y superior de las regiones/ramas del espacio de búsqueda. Si no hay límites disponibles, el algoritmo degenera en una búsqueda exhaustiva.
- **Algoritmo de búsqueda A*** (A estrella) : es un algoritmo de computadora usado ampliamente en la búsqueda de caminos/rutas (*path*) y recorrido de grafos, los cuales son procesos de encontrar un camino entre múltiples puntos (nodos). Ampliamente usado debido a su

rendimiento y precisión. A* es un algoritmo de búsqueda informado (*best-first search*), lo que significa que está formulado en términos de grafos ponderados: a partir de un nodo de inicio específico de un grafo, tiene como objetivo encontrar un camino hacia el nodo objetivo dado que tenga el menor costo (menor distancia recorrida, menor tiempo, etc.). Para ello, mantiene un árbol de rutas que se originan en el nodo de inicio y extiende dichas rutas una arista (*edge*) a la vez hasta que se cumpla su criterio de terminación. En cada iteración de su bucle principal, A* necesita determinar cuál de sus caminos se extenderá. Lo que hace basándose en el costo del camino y en una estimación del costo requerido para extender el camino hasta la meta.

Pinocchio basado en estas ideas: mantiene una cola de prioridad de incrustaciones parciales ordenadas por sus estimaciones de límite inferior. En cada paso, toma la mejor inserción parcial de la cola, la extiende de todas las formas posibles con la siguiente articulación y empuja los resultados a la cola; de esta manera la primera incrustación completa extraída de la cola está garantizada ser la óptima. Este proceso es esencialmente el algoritmo A* en el árbol de posibles incrustaciones. Para acelerar el proceso y conservar memoria, si una incrustación parcial tiene un límite inferior muy alto, se rechaza inmediatamente y no se inserta en la cola.

2.6.1.5. Refinamiento de incrustación

Pinocchio toma la incrustación óptima del esqueleto reducido encontrado por la optimización discreta y reinserta las articulaciones de grado dos al dividir las rutas más cortas en el grafo G en proporción al esqueleto dado. La incrustación de esqueleto resultante debe tener la forma general buscada, pero por lo general, no encajará bien dentro del personaje. Además, es probable que los huesos más pequeños tengan una orientación incorrecta porque no fueron lo suficientemente importantes como para influir en la optimización discreta. El refinamiento de incrustación corrige estos problemas al minimizar una nueva función de penalización continua (Figura 2.17).

Esta función de penalización continua g que Pinocchio trata de minimizar es la suma de cuatro funciones de penalización aplicadas sobre los huesos del esqueleto [4]:

- Penaliza los huesos que no están cerca del centro del objeto.
- Penaliza los huesos que son demasiado cortos cuando se proyectan sobre sus contrapartes en el esqueleto dado.

- Penaliza los huesos orientados incorrectamente.
- Penaliza la asimetría en los huesos que están marcados como simétricos.

Cabe destacar que algunas de estas penalizaciones son similares a las utilizadas en la función de penalización discreta, pero cada vez se utilizan diferentes medidas y factores.

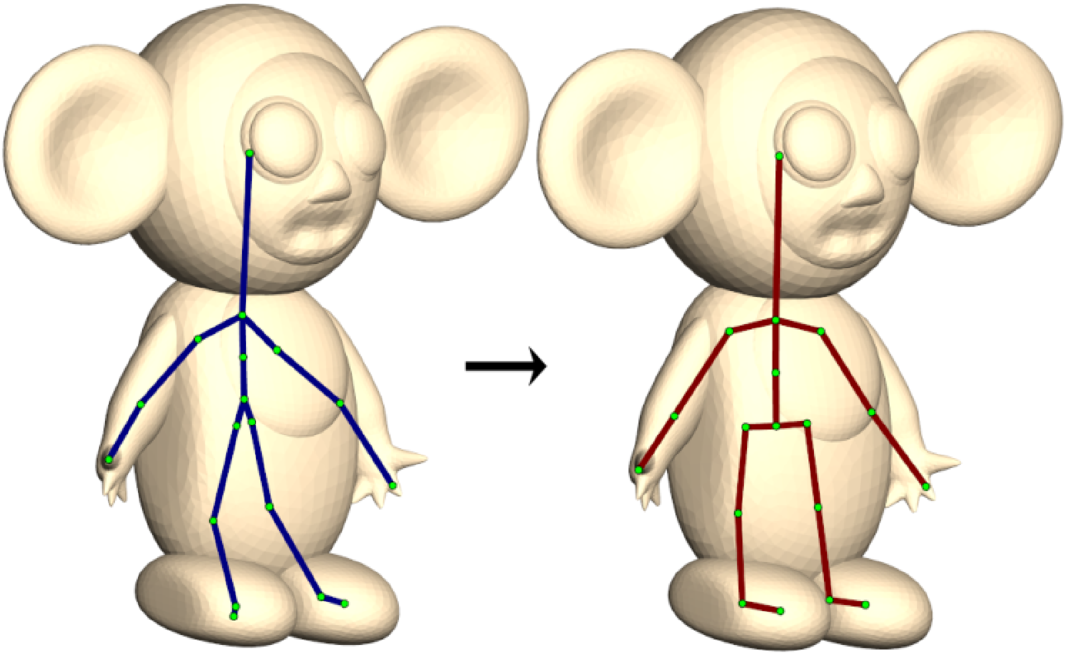


Figura 2.17: Visualización del esqueleto incrustado por Pinocchio después del proceso de Incrustación Discreta (azul) y del esqueleto resultante después del Refinamiento de Incrustación (rojo) [3].

Capítulo 3

Implementación de la Aplicación

La aplicación desarrollada para el presente trabajo especial de grado adapta un esqueleto genérico dentro de una malla estática de un modelo dado y permite visualizar animaciones sencillas de dicho modelo a partir de datos de captura de movimiento. La aplicación funciona bajo el entorno web provisto por Three.js y corre directamente en cualquier navegador web que soporte el motor de rendering gráfico WebGL. Este capítulo presenta una visión general del funcionamiento de la misma y trata de describir los componentes y métodos involucrados en su creación y ejecución.

3.1. Solución Desarrollada

Siendo el objetivo principal de este trabajo: "desarrollar una aplicación utilizando three.js para la animación de personajes a partir de datos de captura de movimiento", es posible subdividir dicha meta en una serie de tareas especificas que contribuyan al resultado final, faciliten su desarrollo y guíen el diseño de la misma. De esta manera, la serie de pasos y tareas que determinaron el flujo de trabajo para la implementación de la aplicación, son las mismas que describen su flujo de ejecución y funciones básicas:

1. Inicializar three.js
2. Crear la escena
3. Establecer la interfaz de usuario

4. Cargar los modelos en formato .obj
5. Aplicar incrustación de esqueletos
6. Asociar las mallas 3D con sus respectivos esqueletos (skinning)
7. Cargar las animaciones en formato .bvh
8. Reproducir las animaciones

3.2. Inicializar three.js

Al ser una aplicación web el primer archivo esencial para su creación y ejecución es el archivo *index.html* (Código 3.1), donde se define la estructura básica de la página HTML junto al código CSS mínimo necesario para asegurar que la aplicación ocupe toda la página web. A continuación, es necesario cargar todas las bibliotecas (archivos .js) indispensables para crear la escena de la aplicación y agregar las respectivas funcionalidades de la misma; cuando el archivo *three.min.js* es cargado Three.js puede ser utilizado dentro de la aplicación web.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width,␣user-scalable=no,␣minimum
            -scale=1.0,␣maximum-scale=1.0" />

        <title>JaneDoe Title</title>
        <link rel="shortcut␣icon" href="favicon.ico">

        <style>
            * {
                margin: 0;
                padding: 0;
                border: 0;
                −webkit−user−select: none;
                −khtml−user−select: none;
                −moz−user−select: −moz−none;
                −o−user−select: none;
                user−select: none;
            }

            body {
                color:#fff;
                font−family: Consolas, Monaco, monospace;
                font−size: 11px;
                background−color: #353839;
                overflow: hidden;
            }

            #container {
                position: absolute;
                width: 100 %;
                height: 100 %;
            }
        </style>
    </head>
    <body>
        <div id="container"> </div>

        <script type="text/javascript" src="js/three.min.js"></script>
        <script type="text/javascript" src="js/OrbitControls.js"></script>
        <script type="text/javascript" src="js/uil.min.js"></script>
        <script type="text/javascript" src="js/OBJLoader.js"></script>
        <script type="text/javascript" src="js/BVHLoader.js"></script>
        <script type="text/javascript" src="js/pinocchioApi.js"></script>

        <script type="text/javascript" src="js/main.js"></script>
    </body>
</html>
```

Código 3.1: Página HTML simple con código básico de hojas de estilo en cascada (CSS).

3.3. Escena

Three.js utiliza el concepto de *escena* para definir el espacio 3D donde se pueden colocar los objetos cargados, las cámaras, las luces para la iluminación, etc. Es por esto que los primeros elementos de configuración necesarios en la aplicación son: una escena, una cámara y un renderizador, para poder renderizar la escena con la cámara. La cámara (en perspectiva) simula el comportamiento de una cámara de película en la vida real, su posición y la dirección hacia la que se enfrenta determinan las partes de la escena que se renderizan en la pantalla. La iluminación de la escena (esencial para poder "ver" los objetos cargados) viene dada por dos tipos de luces: 1. la luz ambiental, tinte de color suave que se aplica a todos los objetos de la escena globalmente; y 2. la luz direccional, gran fuente de luz distante que brilla desde una dirección.

La configuración inicial utilizada para los elementos previamente mencionados puede ser observada en los códigos de las Figuras 3.2 y 3.3. Además en la Figura 3.1 se puede ver la escena inicial de la aplicación renderizada con la cual el Usuario interactuaría; destacando la inclusión de la geometría de un plano que actúa como un "piso" en el espacio 3D.

Figura 3.1: Escena básica inicial de la aplicación, se puede observar la geometría de un plano (usada como piso/floor) y el menú de opciones ubicado en la esquina superior derecha.

```
container = document.getElementById('container');

// Scene
scene = new THREE.Scene();
scene.add(new THREE.AmbientLight(0xffffff));

// Renderer
renderer = new THREE.WebGLRenderer({antialias: true, alpha: true});
renderer.setClearColor(0x000000, 0);
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.autoClear = true;

// Tone
renderer.gammaInput = true;
renderer.gammaOutput = true;
renderer.toneMapping = THREE.Uncharted2ToneMapping;
renderer.toneMappingExposure = 3.0;
renderer.toneMappingWhitePoint = 5.0;

// Shadow
renderer.shadowMap.enabled = true;
renderer.shadowMap.soft = true;
renderer.shadowMap.type = THREE.PCFSoftShadowMap;

container.appendChild(renderer.domElement);

// Camera
var aspect = window.innerWidth / window.innerHeight;
var radius = 40;

camera = new THREE.PerspectiveCamera(50, aspect, 0.1, 10000);
camera.position.set(0.0, radius, radius*3.5);

// Camera - Controls
controls = new THREE.OrbitControls(camera, renderer.domElement);
controls.target.set(0, radius, 0);
controls.update();
```

Figura 3.2: Código de configuración inicial usado en la aplicación desarrollada para la escena, cámara y renderizador.

```
// Light
var ambient = new THREE.AmbientLight(0x282824, 1);
var light = new THREE.DirectionalLight(0xffffff, 1);
light.position.set(10, 150, 60);
light.lookAt(new THREE.Vector3(0,0,0));

var d = 100;
light.shadow.camera.left = -d;
light.shadow.camera.right = d;
light.shadow.camera.top = d * 1.5;
light.shadow.camera.bottom = -d;
light.castShadow = true;
light.shadow.camera.far = 300;
light.shadow.mapSize.width = 1024;
light.shadow.mapSize.height = 1024;
light.shadow.bias =  -0.005;

scene.add(ambient);
scene.add(light);

// planeMat
var planeMaterial = new THREE.MeshPhongMaterial({
    color: '#707070',
    side: THREE.FrontSide
});
plane = new THREE.Mesh(new THREE.PlaneBufferGeometry(200, 200, 1, 1), planeMaterial);
plane.geometry.applyMatrix(new THREE.Matrix4().makeRotationX(-Math.PI*0.5));
plane.castShadow = false;
plane.receiveShadow = true;

scene.add(plane);
```

Figura 3.3: Código de configuración inicial usado para las luces y el piso en la escena de la aplicación desarrollada.

3.4. Interfaz de Usuario

Para cumplir con el objetivo principal de la aplicación de animar personajes, es necesario primero determinar que modelos de personajes animar y cuales clips de animación optar por reproducir. Dentro del flujo de ejecución de la aplicación desarrollada el Usuario participa en cuatro tareas específicas:

1. Carga de archivos .obj: Mediante el uso del explorador de archivos, el Usuario escoge un archivo .obj para iniciar el proceso de incrustación de esqueleto dentro del modelo especificado.
2. Carga de archivos .bvh: Mediante el uso del explorador de archivos, el Usuario escoge un archivo .bvh para iniciar el proceso de lectura y carga del clip de animación especificado.

3. Selección de modelo de personaje a visualizar: Cuando dos o más modelos de personajes (archivos .obj) han sido cargados en la aplicación, el Usuario tiene la opción de optar por cual de estos será visualizado.

4. Selección de clip de animación a reproducir: Cuando se ha cargado algún modelo de personaje (.obj), el Usuario opta por cual clip de animación reproducir.

Adicionalmente al Usuario se le ofrecen ciertas opciones para que este pueda cambiar y manipular la visualización general de la escena:

- Controles de cámara para cambiar directamente la vista de la escena 3D.
- Opción para mostrar/ocultar el esqueleto de los modelos cargados.
- Opción para mostrar/ocultar la malla de los modelos cargados.
- Control deslizante para desplazar verticalmente la geometría del plano usado como piso.
- Opción para restringir el movimiento de los personajes durante sus animaciones (bloquear en un lugar fijo la articulación de las caderas/hip de los personajes).

El menú (Figura 3.4) y los controles de cámara son los medios mediante los cuales el usuario interactúa con la aplicación para controlar las operaciones a realizar y manipular la visualización general de la escena. En la Figura 3.5 puede ser observado el código utilizado para construir el menú (interfaz gráfica), mientras que el código que habilita los controles de cámara puede ser visto en la Figura 3.2.

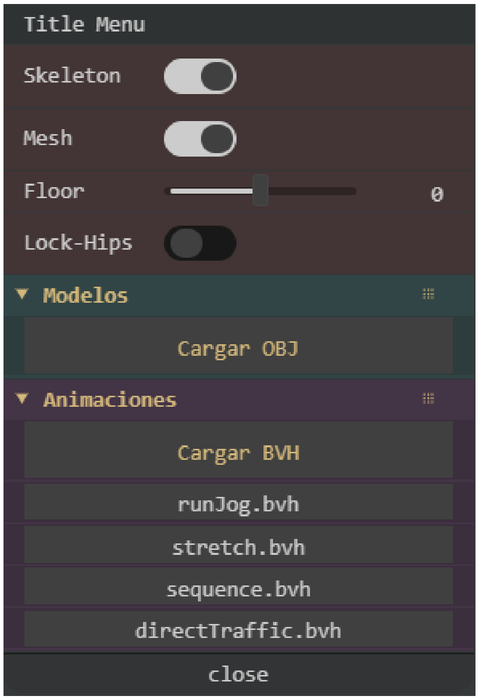
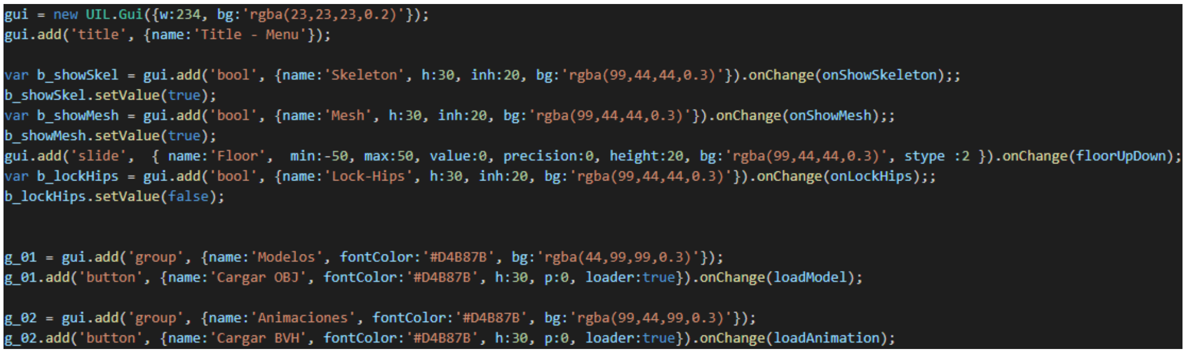
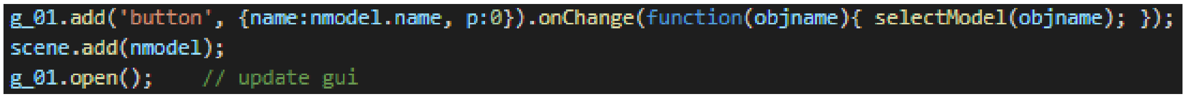


Figura 3.4: Menú de opciones de la aplicación (Interfaz Gráfica). Opciones para cambiar visualizaciones dentro de la escena y botones para cargar/seleccionar modelos y animaciones.



```
gui = new UIL.Gui({w:234, bg:'rgba(23,23,23,0.2)'});
gui.add('title', {name:'Title - Menu'});

var b_showSkel = gui.add('bool', {name:'Skeleton', h:30, inh:20, bg:'rgba(99,44,44,0.3)'}).onChange(onShowSkeleton);;
b_showSkel.setValue(true);
var b_showMesh = gui.add('bool', {name:'Mesh', h:30, inh:20, bg:'rgba(99,44,44,0.3)'}).onChange(onShowMesh);;
b_showMesh.setValue(true);
gui.add('slide',  { name:'Floor',  min:-50, max:50, value:0, precision:0, height:20, bg:'rgba(99,44,44,0.3)', stype :2 }).onChange(floorUpDown);
var b_lockHips = gui.add('bool', {name:'Lock-Hips', h:30, inh:20, bg:'rgba(99,44,44,0.3)'}).onChange(onLockHips);;
b_lockHips.setValue(false);


g_01 = gui.add('group', {name:'Modelos', fontColor:'#D4B87B', bg:'rgba(44,99,99,0.3)'});
g_01.add('button', {name:'Cargar OBJ', fontColor:'#D4B87B', h:30, p:0, loader:true}).onChange(loadModel);

g_02 = gui.add('group', {name:'Animaciones', fontColor:'#D4B87B', bg:'rgba(99,44,99,0.3)'});
g_02.add('button', {name:'Cargar BVH', fontColor:'#D4B87B', h:30, p:0, loader:true}).onChange(loadAnimation);
```

(a) Código inical del menú



```
g_01.add('button', {name:nmodel.name, p:0}).onChange(function(objname){ selectModel(objname); });
scene.add(nmodel);
g_01.open();    // update gui
```

(b) Cada vez que un modelo .obj es cargado en la aplicación, se agrega este código al menú para habilitar la opción de su selección.

Figura 3.5: Visualización del Código utilizado para la configuración del menú (interfaz gráfica) de la aplicación desarrollada.

- **La Interfaz Gráfica** fue realizada mediante el uso de la herramienta *uil.js* [1], una interfaz de usuario simple y ligera para Javascript. Las opciones y botones observables en el menú (Figura 3.4) de la aplicación corresponden a las tareas requeridas por parte del Usuario para el flujo de ejecución del programa, en adición a las alternativas de visualización previamente mencionadas:
    1. Skeleton: Botón interruptor para mostrar/ocultar el esqueleto de los modelos cargados.
    2. Mesh: Botón interruptor para mostrar/ocultar la malla (superficie geométrica) de los modelos cargados.
    3. Floor: Control deslizante (slider) para desplazar verticalmente el plano utilizado como piso en la escena.
    4. Lock-Hips: Botón interruptor para bloquear/desbloquear los desplazamientos de los modelos cargados durante la reproducción de animaciones (in-place animations).
    5. Cargar OBJ: Botón para cargar modelos en formato .obj dentro de la escena. Una vez el modelo ha sido cargado se inicia el proceso de incrustación de esqueleto automáticamente.
    6. [nombre del archivo].obj: Botón (listado de botones) para seleccionar (visualizar de manera exclusiva) el modelo de nombre específico.
    7. Cargar BVH: Botón para cargar animaciones en formato .bvh; después de su lectura son agregadas al listado existente para su posible reproducción.
    8. [nombre del archivo].bvh: Botón (listado de botones) para reproducir la animación de nombre específico.
- **Los Controles de Cámara** fueron agregados con el uso del archivo *OrbitControls.js* proveniente de los ejemplos oficiales de Three.js [2]. Este conjunto de controles permite realizar movimientos de órbita (orbiting - mover la cámara alrededor de un objetivo), panorámica (panning - movimiento sobre el eje vertical u horizontal) y zooming (alejar y acercar la cámara).

  Dichos movimientos de cámara pueden ser ejecutados mediante el uso de un ratón

---

[1] uil.js - interfaz de usuario simple y ligera para Javascript. Fuente: `https://github.com/lo-th/uil`

[2] OrbitControls.js - fuente: `https://github.com/mrdoob/three.js/blob/master/examples/js/controls/OrbitControls.js`

(mouse) o un panel táctil (trackpad):

1. Órbita: Botón izquierdo del mouse.
2. Panorámica: Botón derecho del mouse Ó teclas arriba-abajo-izquierda-derecha de un teclado.
3. Zoom: rueda del ratón (mousewheel).

3.5. Carga de Modelos en formato .obj

El proceso de carga de archivos de objeto (.obj) se realiza de manera sencilla con el uso de uno de los cargadores (loader) de Three.js para cargar recursos .obj, *OBJLoader.js* [3].

Una vez se seleccione la opción de "Cargar OBJ" en la interfaz de usuario y se haya especificado el archivo .obj deseado, la aplicación automáticamente inicia el proceso de incrustación de esqueleto y en caso de que este sea exitoso será posible visualizar la malla dentro de la escena (Figura 3.6). Para que un modelo en formato .obj sea procesado exitosamente y pueda ser visualizado correctamente, este debe cumplir ciertos requisitos (Alcance) que evitan inconvenientes y resultados no deseados:

- **La malla debe ser única y cerrada:** la superficie geométrica en el archivo debe ser un solo componente conectado. Evitar modelos de personajes con múltiples partes que no crean un volumen compacto.
- **La malla debe tener su volumen bien definido:** la superficie geométrica debe cumplir con las siguientes propiedades: 1. Cada arista pertenece a dos caras. 2. Cada vértice está rodeado por una secuencia de aristas y caras. 3. Las caras solo se intersecan entre sí en bordes y vértices comunes. Otra forma de expresarlo, el modelo debe tener una forma que pueda manufacturarse en la vida real y pueda desplegarse en una pieza plana (el modelo debe ser *manifold*).
- **La malla debe estar orientada correctamente:** el modelo de personaje debe estar erguido y frente al usuario.

---

[3] OBJLoader.js – fuente: `https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/OBJLoader.js`

- **El modelo debe ser proporcionalmente similar al esqueleto inicial:** el modelo de personaje debe poseer una anatomía aproximadamente antropomórfica.

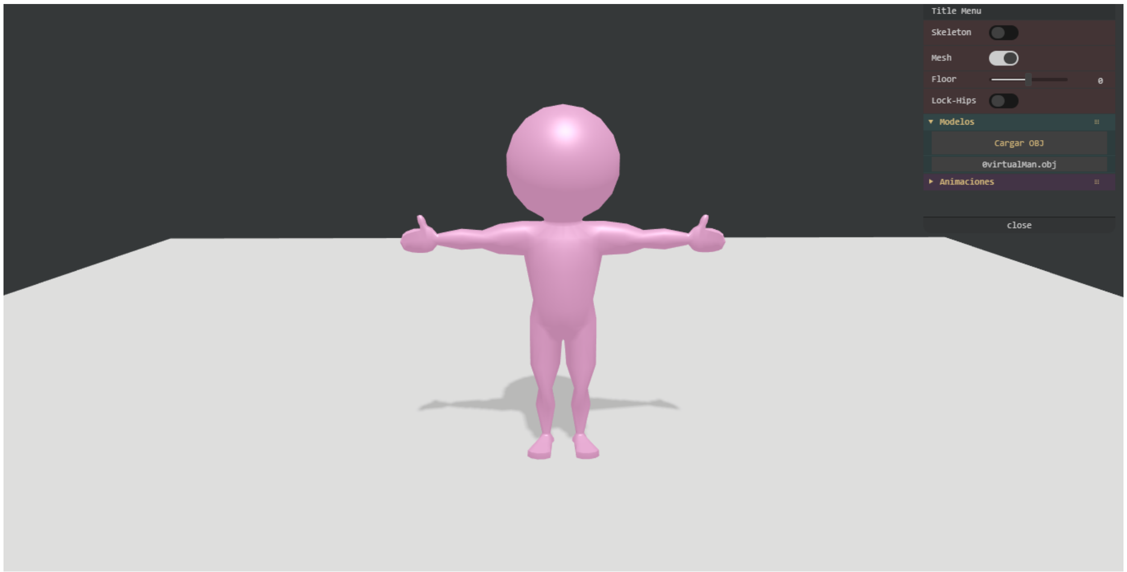


Figura 3.6: Visualización de la malla de un personaje en la escena de la aplicación, después de haber sido cargado a partir de archivo .obj.

3.6. Incrustación del esqueleto

Si la lectura del archivo .obj fue realizada exitosamente, entonces el proceso de incrustación se inicia automáticamente y el primer paso para esto es llamar al modulo de la API de Pinocchio en el archivo *pinocchioApi.js*, con la malla como parámetro de entrada. La API de Pinocchio calcula la incrustación del esqueleto inicial dentro del personaje y retorna la jerarquía esquelética (información de las articulaciones/joints) dentro de un arreglo (array). Dicho array contendría 72 valores numéricos correspondientes a las 18 articulaciones del esqueleto incrustado: tres coordenadas (x, y, z) especificando la ubicación la articulación y un número de índice indicando su articulación padre. (Código Figura 3.7)

El proceso de cómo la API de Pinocchio lleva a cabo la incrustación de esqueletos fue detallado en el *Capítulo 2 - Sección Pinocchio API*. Mientras que el resultado de una incrustación exitosa (modelo cargado dentro de la escena de la aplicación) puede ser visualizado en la Figura 3.8.

```
var objLoader = new THREE.OBJLoader(manager);
var object = objLoader.parse(data);

// detect multi-meshes
if (object.children.length == 1) {
    object.traverse(function(child) {
        if (child instanceof THREE.Mesh) {
            // pinocchio embedding
            console.log("skeleton_embedding: " + name + " ... Waiting");

            var resultPino = Module.get_embedding(data);
            var rPino_checkSz = 72;

            if (resultPino.size() >= rPino_checkSz) {
                console.log("skeleton_embedding: " + name + " ... Done");
                /* resultPinocchio:_18 joints_ (x,y,z) coordinates && previous joint index
```

Figura 3.7: Visualización del código de la aplicación desarrollada donde se realiza la lectura de los modelos (.obj) y se obtiene la información de las articulaciones después del proceso de incrustación de esqueleto "var resultPino = Module.get_embedding(data);".

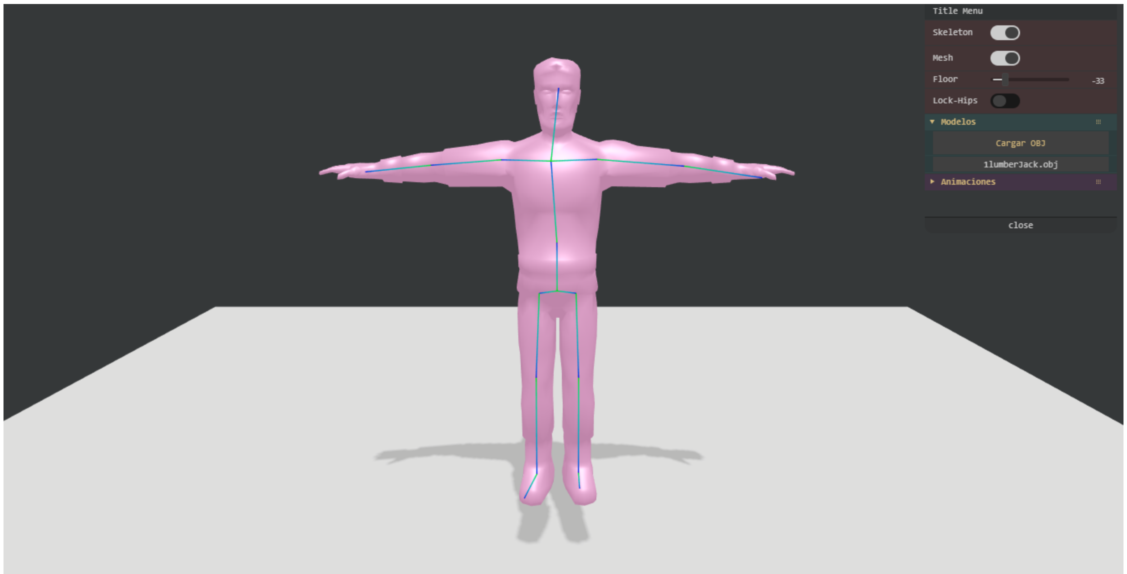


Figura 3.8: Visualización del modelo de un personaje y el esqueleto incrustado resultante después de haber cargado exitosamente un archivo .obj en la aplicación.

3.6.1. Enlace entre la aplicación y Pinocchio

La biblioteca de Pinocchio para el rigging automático de personajes se encuentra escrito en el lenguaje de programación C++, mientras que la aplicación realizada se implementó usando *WebGL y Three.js*, las cuales son herramientas que trabajan con el lenguaje JavaScript. Por lo que era necesario encontrar la forma de conectar el código de una e integrarlo en la otra. Para lograr esto se utilizaron un conjunto de programas dentro de la cadena de herramientas de *Emscripten*.

El primer paso fue la compilación del código de Pinocchio en bitcode (.bc). El segundo, fue el de crear un archivo de "enlace" (bind) para definir la función que se utiliza dentro del código Javascript (mediante la cual se obtiene la jerarquía esquelética incrustada). Y finalmente, la compilación del archivo de enlace junto al código de Pinocchio en bitcode resulta en el archivo **pinocchioApi.js**, la API de Pinochio en lenguaje Javascript que se incluye en la aplicación.

Embind es la herramienta de Emscripten que permite enlazar (bind) funciones y clases en C++ con JavaScript, el archivo de enlace creado para la aplicación fue bindingPinocchio.cpp, cuyo código puede ser visto en el Código 3.2. Dicho archivo define la función *get_embedding(mesh_data)*, la cual recibe como parámetro de entrada los datos de la malla obtenidos durante la lectura del archivo .obj y ejecuta las siguientes instrucciones:

1. Crea un objeto "Mesh" con los datos de la malla recibidos.
2. Inicializa un objeto "Skeleton" que representa el esqueleto inicial humanoide que sirve como base para el proceso de incrustación.
3. Invoca la función *autorig(skeleton, mesh)* de la API de Pinocchio con el esqueleto y la malla con los cuales se debe de llevar a cabo el proceso de incrustación de esqueleto.
4. Verifica que el proceso de incrustación haya sido exitoso.
5. Inicializa el vector donde se retornarán los datos del esqueleto incrustado.
6. Iterativamente reajusta las posiciones de las articulaciones del esqueleto para que sean proporcionales y correspondan con el tamaño y escala de la malla original.
7. Para cada articulación de la jerarquía esquelética almacena sus coordenadas xyz y el índice de su articulación padre dentro del vector previamentente inicializado.
8. Retorna los datos del esqueleto.

```
#include "pinocchioApi.h"
#include <emscripten/bind.h>
#include <iostream>

using namespace emscripten;

std::vector<float> get_embedding(std::string filedata) {

    Mesh m(filedata);

    Skeleton skeleton = HumanSkeleton();
    skeleton.scale(1. * 0.7);

    PinocchioOutput o;
    o = autorig(skeleton, m);

    if (o.embedding.size() == 0) {
        std::vector<float> jointsInfo;

        // Error embedding _ size 0
        return jointsInfo;

    } else {
        std::vector<float> jointsInfo;
        int i, k=0;

        m.computeVertexNormals();
        m.normalizeBoundingBox();

        for (i = 0; i < (int)o.embedding.size(); i++)
            o.embedding[i] = (o.embedding[i] - m.toAdd) / m.scale;

        // joints: x y z prevJointIdx
        for (i = 0; i < (int)o.embedding.size(); i++) {
            jointsInfo.push_back(o.embedding[i][0]);
            jointsInfo.push_back(o.embedding[i][1]);
            jointsInfo.push_back(o.embedding[i][2]);
            jointsInfo.push_back(skeleton.fPrev()[i]);

            k+=4;
        }

        return jointsInfo;
    }
}

EMSCRIPTEN_BINDINGS(my_module) {
    register_vector<float>("joinstInfo");
    emscripten::function("get_embedding", &get_embedding);
}
```

Código 3.2: Definición de la función *get_embedding(mesh_data)*, mediante la cual la aplicación recibe los datos del esqueleto incrustado una vez este ha sido calculado por la API de Pinocchio.

3.6.2. Esqueleto inicial

El esqueleto utilizado para el proceso de incrustación corresponde a una jerarquía de articulaciones para modelos bípedos con una anatomía antropomórfica. Una visualización aproximada del mismo puede ser observada en la Figura 3.9. Dicho esqueleto posee 18 articulaciones y las posiciones xyz iniciales para cada una de ellas junto a sus respectivas articulaciones padre pueden ser observadas en la Tabla 3.1.

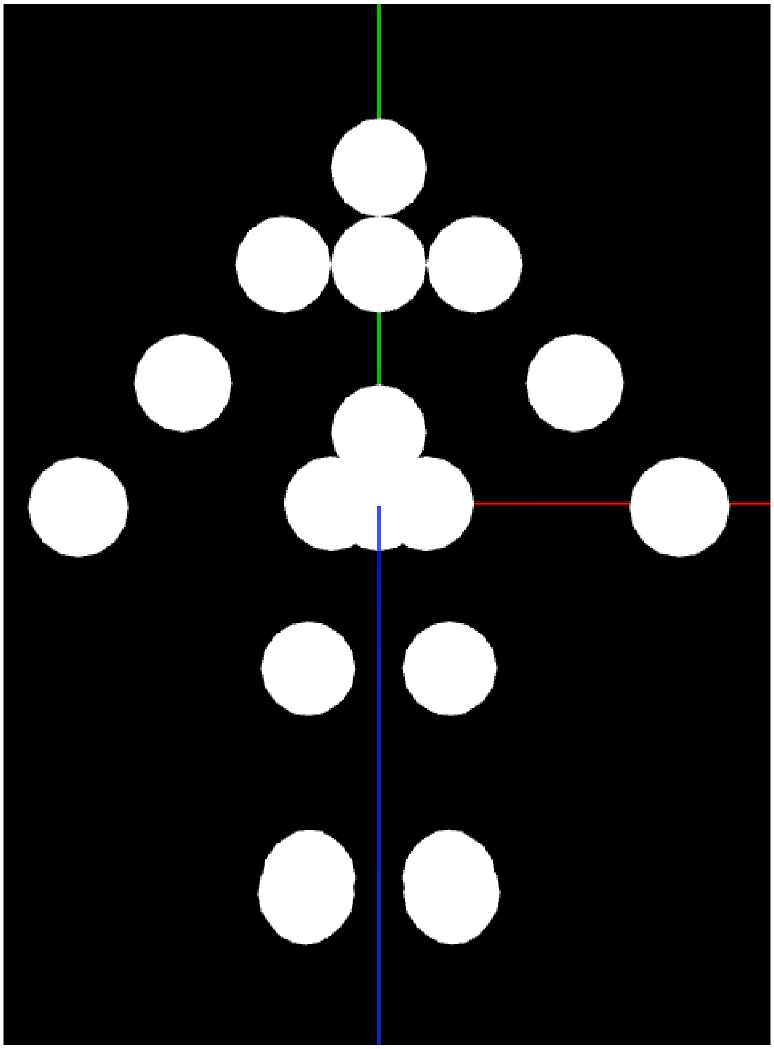


Figura 3.9: Visualización de la jerarquía de articulaciones genérica inicial utilizado por Pinocchio para realizar el proceso de incrustación de esqueleto.

Articulación del Esqueleto	X	Y	Z	Articulación padre
shoulders (raíz)	0	0.5	0	–
back	0	0.15	0	shoulders
hips	0	0	0	back
head	0	0.7	0	shoulders
rthigh	-0.1	0	0	hips
rknee	-0.15	-0.35	0	rthigh
rankle	-0.15	-0.8	0	rknee
rfoot	-0.15	-0.8	0.1	rankle
lthigh	0.1	0	0	hips
lknee	0.15	-0.35	0	lthigh
lankle	0.15	-0.8	0	lknee
lfoot	0.15	-0.8	0.1	lankle
rshoulder	-0.2	0.5	0	shoulders
relbow	-0.4	0.25	0.075	rshoulder
rhand	-0.6	0	0.15	relbow
lshoulder	0.2	0.5	0	shoulders
lelbow	0.4	0.25	0.075	lshoulder
lhand	0.6	0	0.15	lelbow

Tabla 3.1: Jerarquía de articulaciones utilizada por Pinocchio como esqueleto inicial antes de realizar el proceso de incrustación. Nombre de la articulación, posiciones xyz y articulación padre para cada una de las 18 articulaciones de la jerarquía esquelética.

3.7. Skinning de los Modelos

Una vez los datos de la malla (skin) del modelo han sido leídos y el proceso de incrustación de esqueleto dentro de la misma ha sido exitoso, el siguiente paso antes de poder visualizar animaciones de los personajes cargados es definir el como la skin de estos se debe deformar en cada fotograma de una animación de acuerdo a una función de poses esqueléticas (conjunto de estados actuales de las articulaciones). Para esto la malla/skin se "vincula" al esqueleto obtenido por medio de sus vértices. Cada vértice puede estar vinculado a una o más articulaciones. Si está vinculado a una sola articulación, el vértice sigue exactamente el movimiento de esa articulación. Si se vincula a dos o más articulaciones, la posición del vértice se convierte en un promedio ponderado de las posiciones que habría asumido si se hubiese vinculado a cada articulación de manera independiente.

Para que este proceso de skinning sea posible, se debe proporcionar la siguiente información adicional en cada vértice:

- el índice o índices de la(s) articulación(es) a los que está vinculado, y
- para cada articulación, un factor de ponderación que describe cuánta influencia debe tener esa articulación en la posición final del vértice.

En la aplicación, para cada vértice de la malla del modelo cargado, iterativamente se compararon las distancias entre dicho vértice a cada una de las articulaciones del esqueleto y se tomó la menor de estas para determinar a cual articulación el vértice en particular debía vincularse. De esta manera, cada vértice de la superficie geométrica queda asociado a la articulación más cercana del esqueleto incrustado y la influencia que cada articulación tiene sobre todos los vértices vinculados a ella es máxima; individualmente cada vértice se encuentra influenciado por un factor de ponderación igual a 1. Dicho proceso puede ser observado en el código de la Figura 3.10.

```
// skinning
var aux_kk = 0;
var minDistPP = -9000;
var bonIdx = -1;
for (var i = 0; i < ggeometry.vertices.length; i++) {
    var vtxP = new THREE.Vector3(ggeometry.vertices[i].x, ggeometry.vertices[i].y, ggeometry.vertices[i].z);
    bonIdx = -1;
    minDistPP = 999999999;
    aux_kk = 0;

    for (var j = 0; j < skeJoints.length; j++) {
        var bonP = new THREE.Vector3(auxJointsPts[aux_kk], auxJointsPts[aux_kk+1], auxJointsPts[aux_kk+2]);
        var dist_PP = vtxP.distanceTo(bonP);

        if (dist_PP < minDistPP) {
            minDistPP = dist_PP;
            bonIdx = j;
        }
        aux_kk += 3;

    }
    ggeometry.skinIndices.push(new THREE.Vector4(bonIdx, 0, 0, 0));
    ggeometry.skinWeights.push(new THREE.Vector4(1, 0, 0, 0));
}
```

Figura 3.10: Sección de Código de la aplicación desarrollada donde se realiza el proceso skinning en las mallas de los modelos cargados.

3.8. Carga de Animaciones en formato .bvh

El proceso de carga de los archivos .bvh con los datos de captura de movimiento para la animación de los personajes en la aplicación se realiza con el uso del cargador (loader) de Three.js - *BVHLoader.js* [4].

Una vez seleccionada la opción “Cargar BVH” en la interfaz de usuario y se haya especificado el archivo .bvh que se desea leer, la aplicación carga los datos de captura de movimiento y almacena el respectivo clip de animación para su futuro uso en la reproducción de los mismos. Para que se logren reproducir las animaciones cargadas, las articulaciones/-joints en el archivo deben de tener un nombre específico. Es por esto que todos los archivos utilizados en la aplicación fueron obtenidos del mismo sitio: `https://sites.google.com/a/cgspeed.com/cgspeed/motion-capture/cmu-bvh-conversion`. Este es un gran conjunto de movimientos humanos capturados profesionalmente de una amplia variedad de tipos, adecuados para su uso en software de animación; que resultan ser una conversión BVH mejorada del conjunto original de movimientos humanos provenientes de la base de datos de captura de movimiento de la universidad Carnegie Mellon (CMU Graphics Lab Motion Capture Database [5]).

Las razones (ventajas) por las cuales se utiliza el conjunto de archivos especificados sobre los datos originales de Carnegie-Mellon son:

- **Formato BVH:** esta versión tiene los movimientos en el formato BVH más comúnmente usado en lugar del formato ASF/AMC original.
- **Poses-T:** cada archivo BVH tiene una pose-T añadida como su nuevo primer fotograma. La pose-T se posiciona frente al eje-Z positivo y, por lo tanto, es compatible con varios software de animación.
- **Cambio de nombre de las articulaciones (joints):** se cambiaron los nombres de las articulaciones para que fuesen compatibles con las convenciones de nombres de varios software de animación y facilitar su uso (la Tabla 3.2 muestra los cambios realizados).
- **Índice / archivos de información:** esta versión incluye índices consolidados que enumeran los nombres de los archivos de movimiento y sus descripciones. Casi todos los archivos .bvh tienen una descripción de texto, y esas descripciones están todas en un solo archivo de referencia.

---

[4] BVHLoader.js - fuente: `https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/BVHLoader.js`

[5] CMU Graphics Lab Motion Capture Database - fuente: http://mocap.cs.cmu.edu/

Nombre Anterior (CMU)	Nuevo Nombre (.bvh)
hip	Hips
lhipjoint	LHipJoint
lfemur	LeftUpLeg
ltibia	LeftLeg
lfoot	LeftFoot
ltoes	LeftToeBase
rhipjoint	RHipJoint
rfemur	RightUpLeg
rtibia	RightLeg
rfoot	RightFoot
rtoes	RightToeBase
lowerback	LowerBack
upperback	Spine
thorax	Spine1
lowerneck	Neck
upperneck	Neck1
head	Head
lclavicle	LeftShoulder
lhumerus	LeftArm
lradius	LeftForeArm
lwrist	LeftHand
lhand	LeftFingerBase
lfingers	LFingers
lthumb	LThumb
rclavicle	RightShoulder
rhumerus	RightArm
rradius	RightForeArm
rwrist	RightHand
rhand	RightFingerBase
rfingers	RFingers
rthumb	RThumb

Tabla 3.2: Cambios de nombres para cada una de las articulaciones dentro del conjunto de archivos .bvh obtenidos.

3.9. Reproducción de Animaciones

Dentro del sistema de animación three.js se pueden animar varias propiedades de los modelos: los huesos de un modelo con una skin y un rig asignado, objetivos de morph, diferentes propiedades de materiales (colores, opacidad, booleanos), visibilidad y transformaciones. Las propiedades animadas pueden ser distorsionadas (warped) y a su vez se les pueden aplicar distintos efectos de desvanecimiento (fade-in, fade-out, cross-fade). Las escalas de peso y tiempo de diferentes animaciones simultáneas en el mismo objeto así como en distintos objetos se pueden cambiar de forma independiente. Varias animaciones en el mismo y en diferentes objetos se pueden sincronizar.

Los componentes principales del sistema de animación three.js que trabajan juntos para lograr todo esto son:

- **Los clips de animación (Animation Clips):** Cuando los objetos 3D animados son importados exitosamente, uno de sus atributos debe ser un arreglo llamado "animations" que contiene los clips de animación (*AnimationClips*) para este modelo. Cada *AnimationClip* es un conjunto reutilizable de pistas de fotogramas clave, las cuales generalmente contiene los datos de una determinada actividad del objeto. Si la malla es un personaje, por ejemplo, puede haber un *AnimationClip* para una secuencia de caminata (walkcycle), un segundo clip para un salto, un tercer clip para bailar, y así sucesivamente.

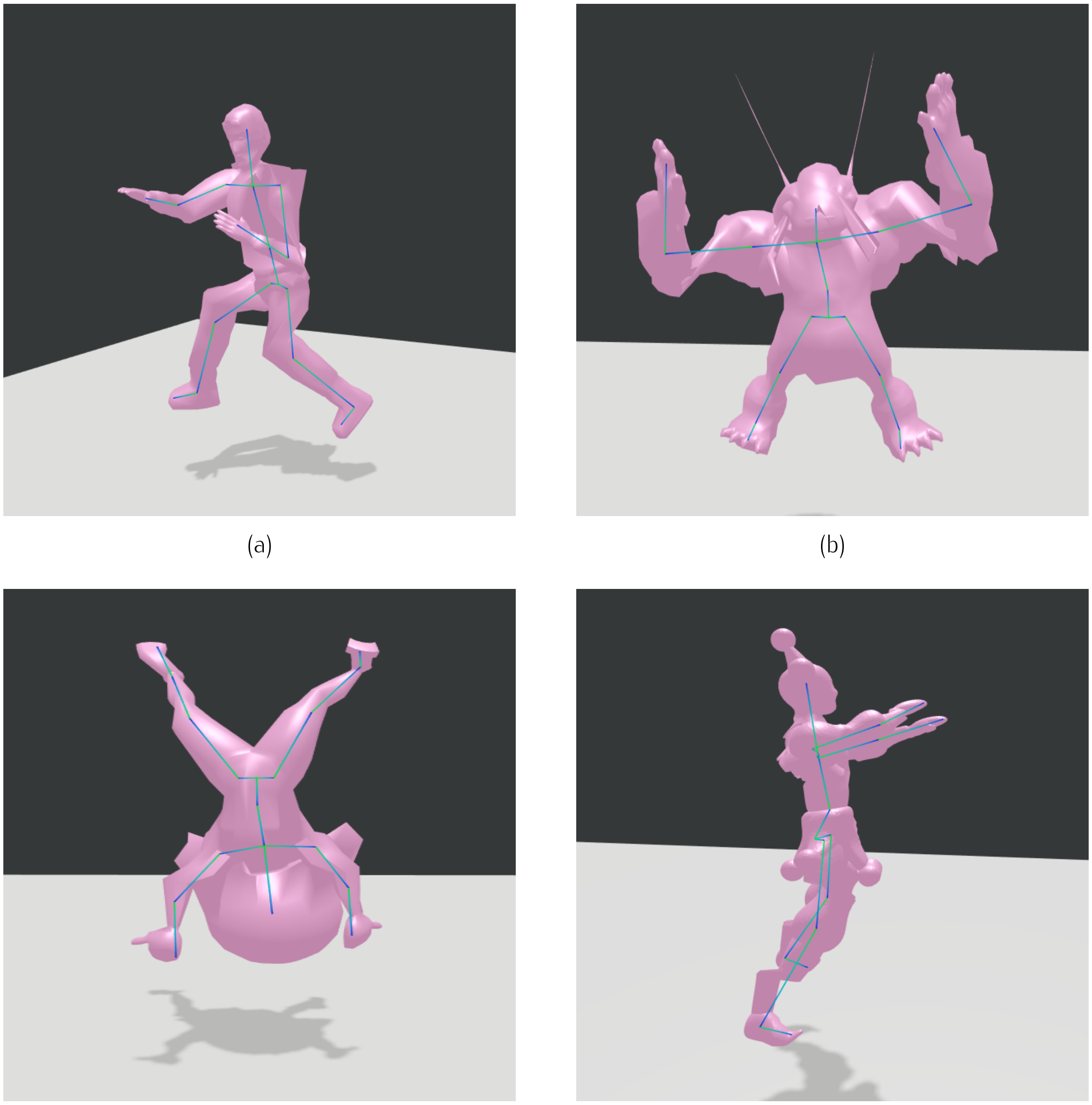
  Hay que tener en cuenta que no todos los formatos de modelo incluyen animaciones, el formato de archivo .obj utilizado en la aplicación en particular no soporta animaciones. Es por esto que para animar los modelos (personajes) cargados se tuvo que aplicar un método de reorientación de animaciones, donde se transfiere la información de las articulaciones (posición y rotación) de un esqueleto fuente (source skeleton) a un esqueleto objetivo (target skeleton). El esqueleto fuente vendría siendo la jerarquía de articulaciones definida en cada uno de los archivos .bvh, el esqueleto objetivo seria la jerarquía esquelética previamente incrustada y asociada a los modelos cargados, y la información a transferir son los datos de captura de movimiento (clips de animación) almacenados en los archivos .bvh.

- **Las pistas de fotogramas clave (Keyframe Tracks):** Dentro de un dicho *AnimationClip*, los datos de cada propiedad animada se almacenan en un *KeyframeTrack* (pista de fotograma clave), la cual es una secuencia cronometrada de fotogramas clave compuesta de listas de tiempos y valores relacionados, que se utilizan para animar una propiedad específica de un objeto. Suponiendo un objeto-personaje cargado junto a su esqueleto, una pista de fotograma clave podría almacenar los datos de los cambios de posición de la articulación de la mano a través del tiempo, una pista distinta contiene los datos de los cambios de rotación de la misma articulación, una tercera pista podría almacenar los cambios de posición/rotación de alguna otra articulación, y así sucesivamente. Debe quedar claro que un clip de animación puede estar compuesto de muchas de estas pistas.

El conjunto de pistas de fotogramas clave - *KeyframeTracks* que se encuentran en cada *AnimationClip*, contienen la información de las articulaciones necesaria para animar los esqueletos incrustados de los modelos cargados. El proceso de transferencia/reorientación de animaciones consiste en copiar pista a pista cada una de las listas de tiempos y valores relacionados de las posiciones y rotaciones para cada articulación del esqueleto fuente que tenga el mismo nombre en el esqueleto objetivo.

- **El mezclador de animaciones (Animation Mixer):** Los datos almacenados forman solo la base de las animaciones; la reproducción real está controlada por el *AnimationMixer*. Este objeto no solo actúa como un reproductor de animaciones, sino además simula el hardware de una consola mezcladora real, que puede controlar varias animaciones simultáneamente, combinándolas (blending) y mezclándolas (merging).
- **Las acciones de animación(Animation Actions):** El *AnimationMixer* en sí tiene muy pocas propiedades y métodos (generales), ya que puede ser controlado por *AnimationActions*. Al configurar una *AnimationAction* se puede determinar cuándo se debe reproducir, pausar o detener un determinado *AnimationClip* en uno de los mezcladores, si es necesario repetir el clip, y con qué frecuencia, si se debe reproducir con alguna escala de tiempo, y algunas otras cosas adicionales, como crossfading o sincronización.

En la Figura 3.11 se pueden observar varios modelos en distintas poses durante la reproducción de ciertas animaciones en la aplicación.



(a)

(b)

(c)

(d)

Figura 3.11: Visualización de personajes cargados en la aplicación en distintas poses durante la reproducción de ciertas animaciones.

Capítulo 4

Resultados

Los tres aspectos fundamentales para llevar a cabo las animaciones de los modelos cargados dentro de la aplicación desarrollada para este trabajo son: 1. la incrustación del esqueleto (el rig mediante el cual se controla la malla), 2. el método de skinning aplicado (definición del como los vértices de la malla se deben asociar a las articulaciones del esqueleto), y 3. el método de animación usado (reorientación de animaciones a partir de datos de captura de movimiento). Es por esto que al momento de observar y evaluar los resultados obtenidos, estos son los factores a tener en cuenta y que se deben analizar para medir como influyen las animaciones producidas.

4.1. Incrustación de esqueletos

Baran y Popović [3] evaluaron los resultados obtenidos de su sistema Pinocchio con respecto a tres criterios: generalidad, calidad y rendimiento. Los modelos que utilizaron para probar su aplicación fueron 16 mallas construidas por un artista usando Cosmic Blobs® software (desarrollado por Dassault Systemes SolidWorks Corp). La Figura 4.1 muestra los 16 personajes de prueba y los esqueletos incrustados por Pinocchio en cada uno de ellos.

La biblioteca de Pinocchio fue desarrollada originalmente en el lenguaje de programación C++; la aplicación desarrollada para este trabajo especial de grado se implementó usando WebGL y Three.js, herramientas que trabajan con el lenguaje JavaScript. La inclusión de Pinocchio al código de dicha aplicación fue posible gracias al uso del conjunto de programas dentro de la cadena de herramientas de Emscripten, mediante los cuales se compiló el código fuente de Pinocchio y se convirtió a Javascript.

Al utilizar la API de Pinocchio en la aplicación desarrollada –si se aplica el método de incrustación de esqueleto sobre los mismos 16 personajes de prueba, es de esperar que los resultados obtenidos sean idénticos. En la Figura 4.2 se puede observar como los esqueletos incrustados resultantes son iguales a los presentados por Baran y Popović en su trabajo [3] (Figura 4.1), y estos no se vieron afectados ni alterados de ninguna manera al portar el código fuente C++ de la biblioteca de Pinocchio a Javascript.

Dados los 16 resultados de las pruebas, los esqueletos incrustados fueron correctos en 13 personajes e incorrectos en 3 de ellos (Modelos 7, 10 y 13). Estos resultados demuestran el rango de proporciones que la técnica de incrustación de esqueleto de Pinocchio puede tolerar: modelos humanos bien proporcionados, brazos grandes y piernas pequeñas (6; en 10, esto causa problemas), y piernas grandes y brazos pequeños (15; En 13, las brazos cortos causan problemas.). En general, los esqueletos casi siempre fueron incrustados correctamente en personajes bien proporcionados cuya postura coincidía con la del esqueleto original dado.

Los modelos 7, 10 y 13 (donde los esqueletos fueron incorrectamente incrustados) sirven de prueba para ilustrar el problemas con el método de incrustación realizado por Pinocchio [28]. El personaje 7 es una mujer en vestido y Pinocchio no tiene la capacidad de detectar la forma del modelo en los primeros pasos del proceso. Por lo general, cuando el personaje es bípedo, el resultado del empaquetado de esferas naturalmente retorna una forma bípeda porque el límite de la malla restringe el radio de las esferas. Sin embargo, en tal caso, el espacio mucho más amplio en el área de las piernas obliga al sistema a elegir aproximadamente el centro del vestido para empacar su primera esfera (la más grande). Aunque Pinocchio impone penalizaciones sobre las articulaciones simétricas de las piernas, estas no salvan el caso tan extremo presentado, y esto es debido a que la configuración de los parámetros en la función de penalización son para personajes bípedos "normales". Los esqueletos incrustados en los modelos 10 y 13 ilustran el mismo problema de detección de forma, y la consecuencia de que Pinocchio no tiene la capacidad de eliminar/descartar articulaciones en posiciones imposibles, como el guante en el personaje 10 y el cabello del personaje 13.

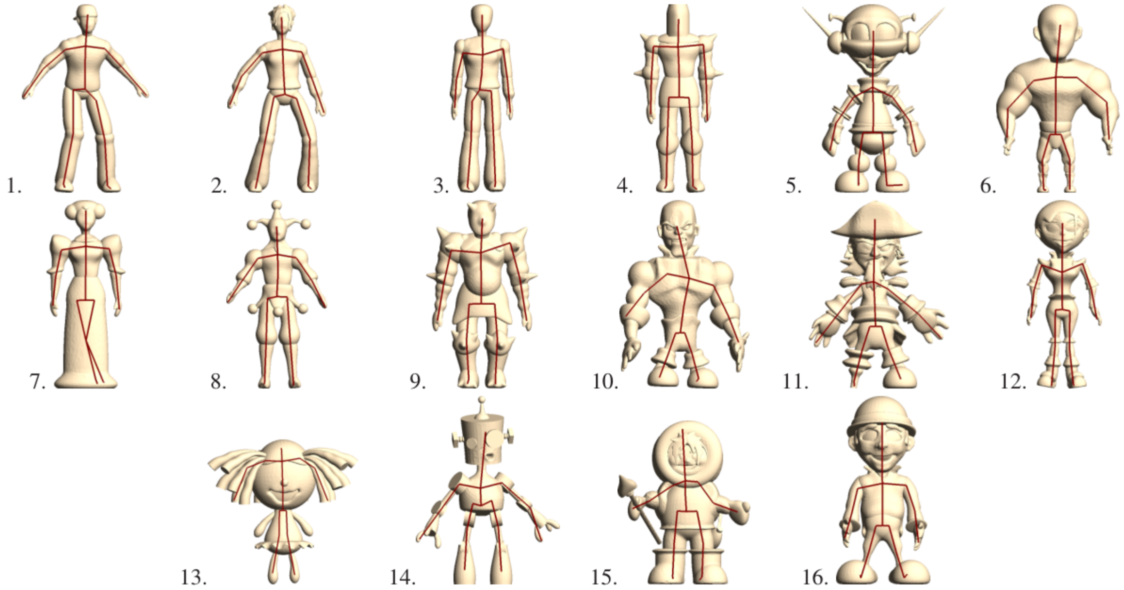
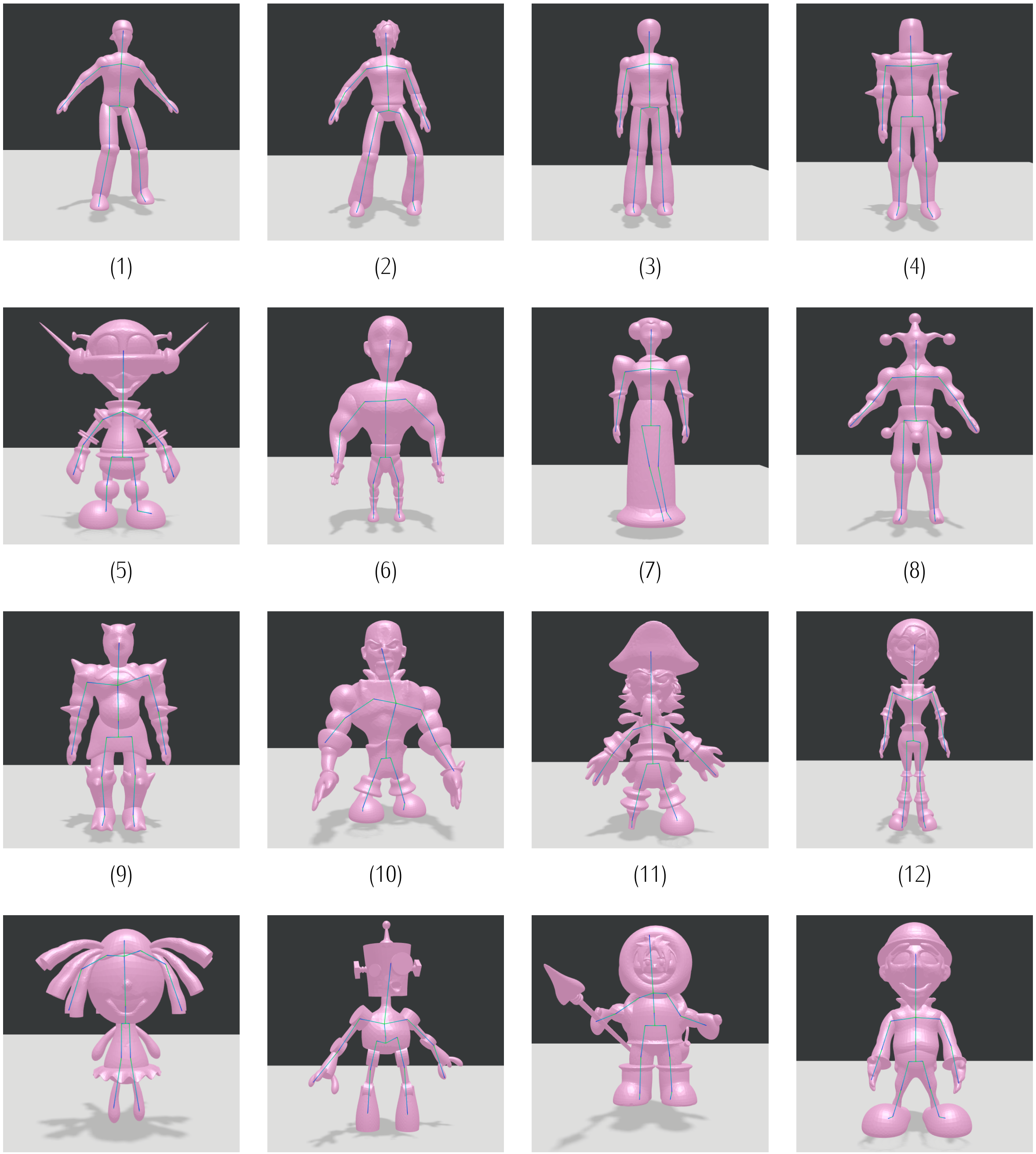


Figura 4.1: Resultados de incrustación de un esqueleto bípedo genérico en 16 modelos distintos realizado por Pinocchio en el trabajo de Baran y Popović [3].



(1) (2) (3) (4)

(5) (6) (7) (8)

(9) (10) (11) (12)

(13) (14) (15) (16)

Figura 4.2: Visualizaciones de los 16 modelos de prueba y sus esqueletos incrustados resultantes.

4.2. Skinning

El método de skinning aplicado para vincular los vértices de las mallas de los modelos a las articulaciones de los esqueletos incrustados, fue extremadamente básico y debido a la sencillez del mismo, las deformaciones resultantes de la malla/skin durante las animaciones no mantienen un volumen constante, y causan autointersecciones locales, fallando así en producir deformaciones naturales convincentes. La cantidad de imperfecciones observadas en la superficie geométrica (skin) de los modelos durante las animaciones reproducidas, varían dependiendo de la pose original en la que se encuentren los modelos y la forma/proporciones de las distintas partes del cuerpo de los personajes (cabeza, brazos y piernas).

Debido a que cada vértice de una malla es vinculado exclusivamente a una articulación de la jerarquía esquelética incrustada (la más cercana), el nivel de influencia que esta ejerce sobre todos los vértices asociados a ella resulta ser máxima (factor de ponderación igual a 1). Para modelos humanoides bien proporcionados cuya postura coincide con la del esqueleto original dado, este método de skinning produce animaciones parcialmente convincentes. Mientras que para modelos con ciertas desproporciones en sus distintas extremidades o cuyas poses originales dificultan la asignación de los vértices a las articulaciones correctas, dicha influencia puede llegar a ser muy extrema y causa deformaciones erróneas (no deseadas) en los grupos de vértices que son asignados a las articulaciones equivocadas.

1. En la Figura 4.3 se puede observar un modelo humano bien proporcionado cuya pose original facilita la asignación de los vértices a las correctas articulaciones y las deformaciones de la malla resultan adecuadas.
2. En la Figura 4.4 se observa un modelo humano bien proporcionado cuya pose original causa que un conjunto de vértices en la malla del personaje sean asignados a una articulación equivocada debido a su proximidad, resultando así en deformaciones no deseadas durante la reproducción de animaciones.
3. La Figura 4.5 muestra un modelo no humano relativamente desproporcionado (brazos largos y área el torso amplio) cuya pose inicial facilita la asignación de los vértices a las articulaciones correctas, resultando en animaciones parcialmente adecuadas.
4. La Figura 4.6 presenta un modelo no-humano desproporcionado (brazos y piernas cortas, junto a una cabeza y orejas grandes); cuyas características resultan en deformaciones no deseadas y hasta rupturas de la malla durante las animaciones reproducidas.

La mayoría de estos problemas son característicos de la técnica de skinning lineal conocida como *linear blend skinning (LBS)*, frecuentemente usada por su eficiencia computacional y facilidad de implementación directa en la GPU. Las limitaciones de LBS han sido estudiadas extensamente, y se han propuesto muchas técnicas para evitar sus defectos visuales (autointersecciones y pérdida de volumen) [23]. Una posibilidad es enriquecer el espacio de los pesos involucrados al momento de hacer el skinning, lo que lleva a métodos que siguen siendo lineales pero que ofrecen una gama más amplia de deformaciones. Estos métodos se denominan técnicas de skinning multilineal, donde la selección de buenos pesos para el proceso de skinning es fundamental para evitar los mismos defectos visuales y generar deformaciones de mallas más naturales.

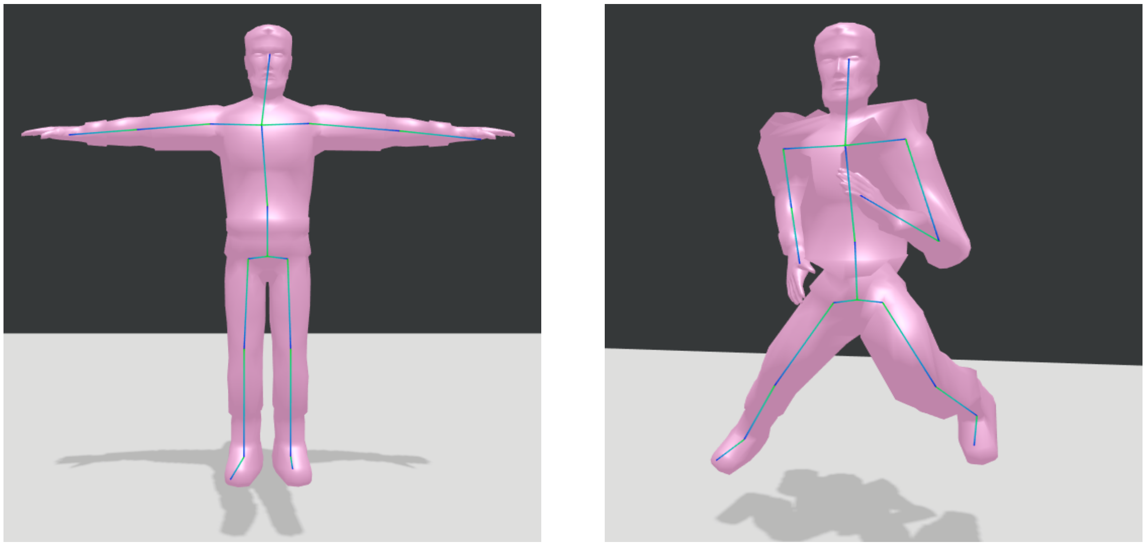
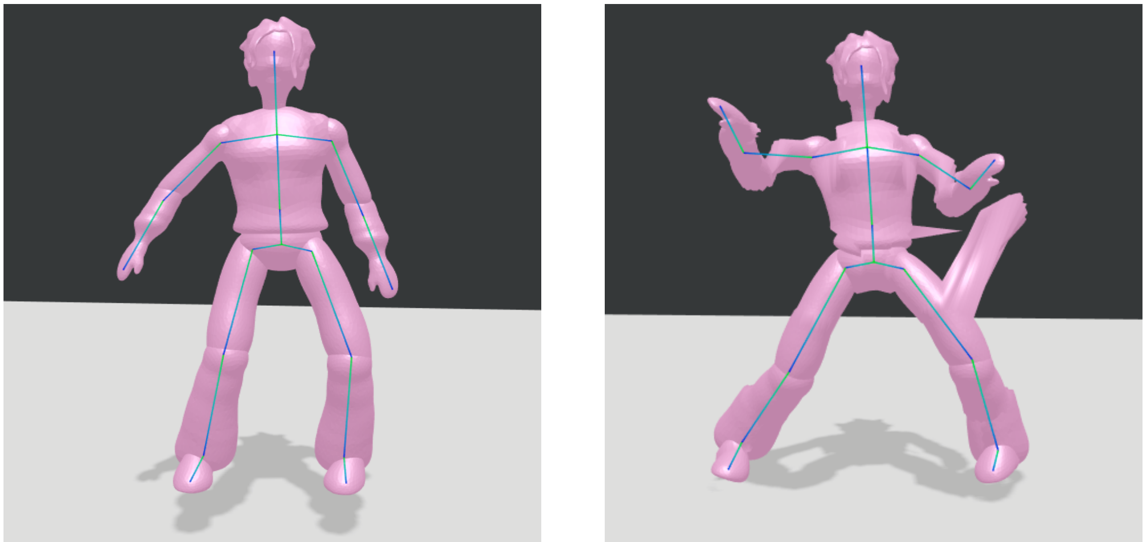
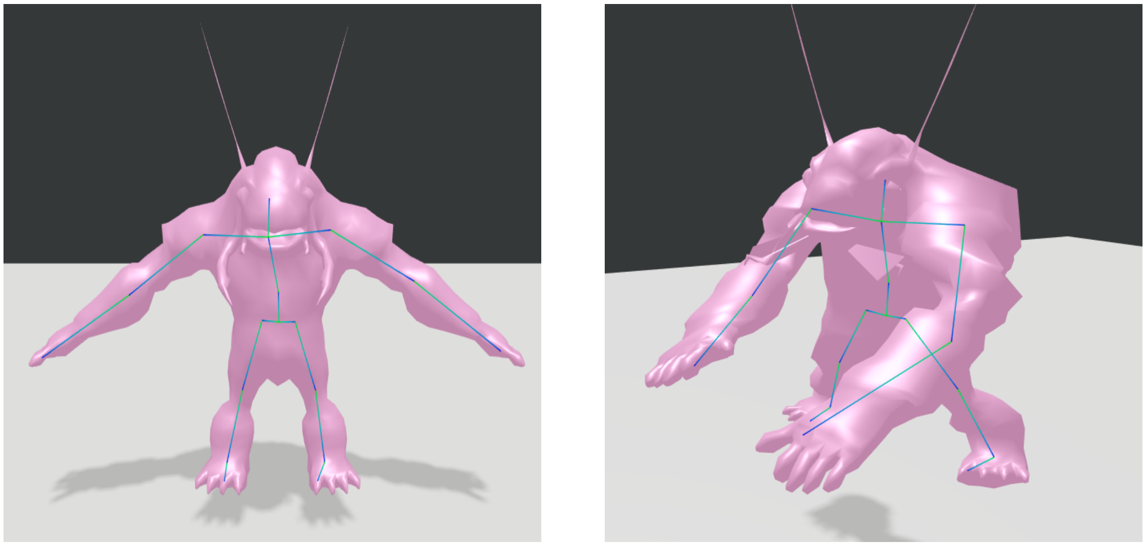

(a) (b)

Figura 4.3: Visualización de personaje humano bien proporcionado en Pose-T y el resultado de una animación ejemplo.



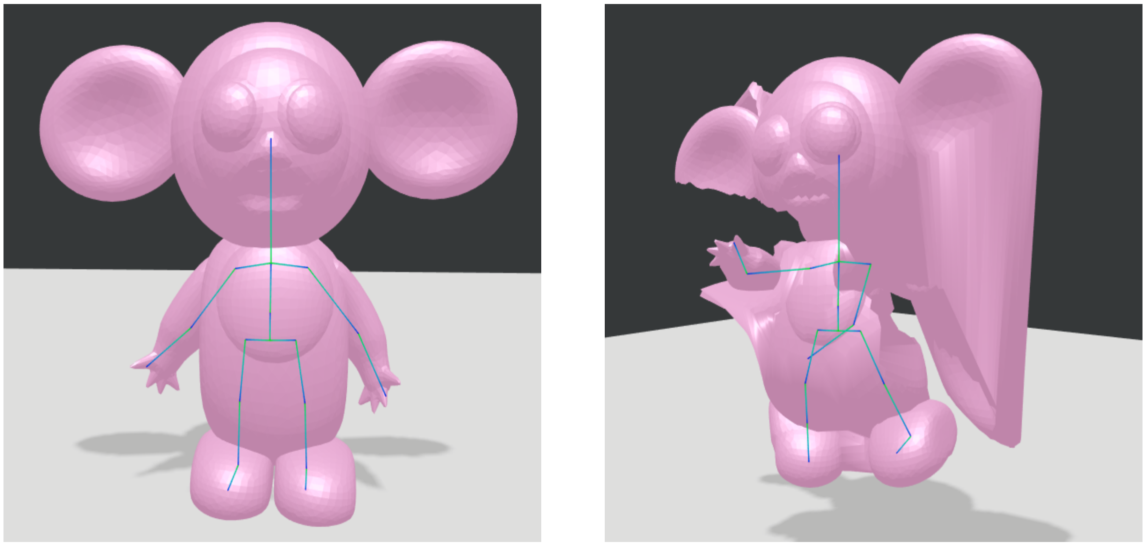
(a) (b)

Figura 4.4: Visualización de personaje humano bien proporcionado cuya pose inicial causa que un grupo de vértices en la pierna sea asignado erróneamente a la articulación de la mano debido a su proximidad; y la deformación incorrecta de la malla durante una animación ejemplo.



(a) (b)

Figura 4.5: Ejemplo visual de un modelo humanoide con extremidades largas (brazos) y área del torso amplio, cuya pose original facilita la asignación de los vértices, y resulta en una animación parcialmente correcta.



(a) (b)

Figura 4.6: Ejemplo visual de un modelo no-humano desproporcionado: cabeza amplia, orejas grandes, brazos y piernas cortas. Resultando en animaciones con deformaciones de malla totalmente incorrectas.

4.3. Animaciones

El método de animación aplicado para mover los esqueletos incrustados y lograr animar los modelos cargados dentro de la aplicación, fue el de reorientación de animaciones (animation retargeting) a partir de datos de captura de movimiento. Los esqueletos fuente junto a los clips de animación (mocap data) a transferir son definidos y detallados dentro de cada uno de los archivos .bvh cargados, mientras que los esqueletos objetivo, donde se copia la información de las articulaciones, son los esqueletos incrustados dentro de los personajes.

El proceso de transferencia/reorientación de animaciones fue simple y consistió en iterar las articulaciones de un esqueleto fuente dado, comparar los nombres de las articulaciones con las del esqueleto objetivo, y para todas aquellas articulaciones que compartiesen el mismo nombre, copiar cada una de las pistas de fotogramas claves correspondientes.

Debido a la simplicidad del proceso previamente descrito, visualmente los resultados obtenidos fueron mixtos. Donde el rango de movimiento de los personajes animados se veían influenciados por las proporciones de sus extremidades y el tamaño original del esqueleto incrustado en comparación a la jerarquía de articulaciones (esqueleto fuente) descrita en los archivos .bvh. Como puede ser visto en la Figura 4.7, dependiendo de las dimensiones del esqueleto/modelo a animar, un mismo clip de animación podía visualizarse de distintas maneras y las diferencias en el rango de los movimientos involucrados resultan notables.

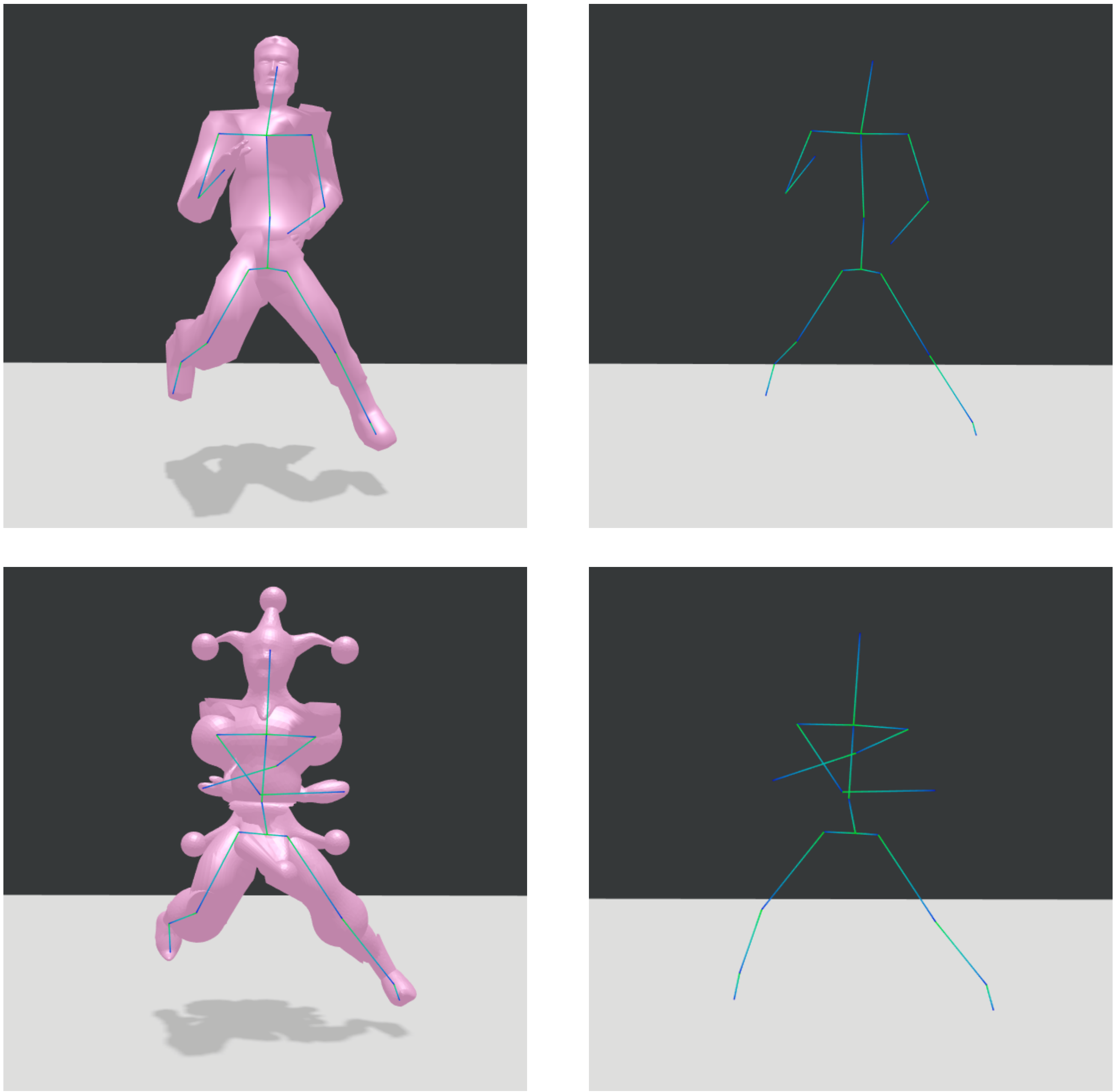


Figura 4.7: Visualización de ejemplo donde el mismo clip de Animación -Correr/Run- es aplicado en dos personajes distintos y se puede observar como el rango de movimientos de los brazos es diferente para cada uno de ellos.

Conclusiones

La animación esquelética es una técnica de animación que se ha convertido en una herramienta estándar para crear animaciones 3D, introducida en 1988 por Magnenat-Thalmann, Laperrière y Thalmann [17]. Esta técnica se utiliza en prácticamente todos los sistemas de animación actuales y mediante el uso de interfaces de usuario simplificadas permiten a los animadores controlar algoritmos complejos y grandes cantidades de modelos geométricos. Al ser una herramienta principal para la animación en numerosas industrias, incluyendo películas y videojuegos, esta es a menudo usada junto con otras técnicas de animación y procesos matemáticos con el objetivo de controlar las deformaciones de los datos de malla.

Al concluir la realización de este Trabajo Especial de Grado, se presenta la construcción del prototipo de una aplicación web la cual permita visualizar animaciones sencillas de modelos 3D de personajes a partir de datos de captura de movimiento. La cual aplicando técnicas y métodos existentes de incrustación de esqueleto, skinning y reorientación de animaciones esqueléticas ofrece la oportunidad a usuarios principiantes de pasar de una malla estática de un modelo a un personaje animado de forma rápida y sin esfuerzo. Se muestra que utilizando el método de incrustación de esqueleto presentado por Baran y Popović [3] (biblioteca Pinocchio) es posible automatizar el proceso de rigging de personajes en un entorno web, y junto a una colección amplia de movimientos esqueléticos (datos de captura de movimiento) se puede establecer un sistema de animación simple y fácil de usar.

Las prioridades al momento de desarrollar la aplicación para este trabajo fueron *simplicidad y completitud*; las áreas de estudio que abarcan animación, skinning y rigging automático de personajes pueden llegar a ser bastante amplias y complejas en sus aspectos más avanzados y detallados. Evitando que el proyecto fuese abrumador, se implementaron técnicas y procesos sencillos que a pesar de obtener los resultados buscados, dejan aspectos notables que pueden mejorarse. Los cuales a su vez delimitan los ámbitos de posibles trabajos futuros que extiendan el alcance de la aplicación y ofrezcan oportunidades de aprendizaje.

Trabajos Futuros

Basado en los factores descritos en el *Capítulo 4: Resultados*, los tres aspectos fundamentales donde la aplicación desarrollada se beneficiaria de varias mejoras y vale la pena realizar futuros trabajos/estudios son:

1. **El método de incrustación de esqueleto utilizado:** no todos los esqueletos incrustados por Pinocchio fueron correctos, la API utilizada fallaba en detectar y descartar aquellas partes del cuerpo de los modelos que tenían formas similares a sus extremidades (caso de ejemplo: modelo cuyo cabello fue confundido por los brazos), áreas problemáticas donde la incrustación de esqueletos era indebida. Además es necesario establecer un método que recorra la jerarquía de articulaciones, verifique y ajuste/corrija todas aquellas mal colocadas (caso de ejemplo: modelo con vestido donde las articulaciones de los pies no eran simétricas). Para mejorar estos problemas se podría implementar la técnica descrita en el trabajo presentado por *Haolei Wang, 2012* [28], donde se muestra que al aplicar un algoritmo de agrupamiento basado en densidad DBSCAN es posible filtrar algunos vértices imposibles (localizaciones candidatas donde ubicar alguna articulación) para corregir errores en las extremidades de los personajes (cabello, manos y pies).
2. **El método de skinning aplicado:** la calidad de las deformaciones en las mallas de los modelos durante la reproducción de animaciones es uno de los primeros aspectos que se debería buscar mejorar en la aplicación. Esto no resulta del todo difícil especialmente considerando que el método de skinning usado fue bastante básico. La aplicación de alguna técnica de skinning multilineal, la cual amplie el espacio de los pesos involucrados sería una manera de mejorar las imperfecciones visuales en las mallas (autoinserccciones, pérdida de volumen) y evitaría el problema principal que tiene la aplicación con las deformaciones indeseadas cuando algún grupo de vértices es asignado a la articulación errónea. Las técnicas presentadas por *Dionne y de Lasa, 2013* [8] podría ser considerada, la cual a partir de un proceso de voxelización inicial de la malla, calcula los pesos de influencia basado en la distancia geodésica entre cada vóxel que se encuentre sobre un "hueso" del esqueleto y todos los vóxeles no exteriores. Por supuesto sin importar que modelo de skinning se decida aplicar, se debe tener la consideración de que three.js solo soporta un máximo de 4 pesos de influencia por vértice (4 skinning weights per vertex).

3. **El método de animación usado:** la reorientación de animaciones a partir de los datos de captura de movimiento en los archivos .bvh fue aplicado de una manera bastante simple, copiando las pistas de fotogramas claves para cada articulación y creando un clip de animación que luego era reproducido por el objeto AnimationMixer provisto por Three.js. El principal problema de este enfoque es que las proporciones de los modelos no son tomadas en cuentas; y cuando los clips de animación son aplicados en esqueletos incrustados cuyas dimensiones difieren al esqueleto original descrito en los archivos .bvh, las animaciones producidas son inconsistentes. Es por esto que la implementación de un mejor esquema de reorientación es necesario, las animaciones deben ser más convincentes y se deben tratar de reducir las autointersecciones globales.

Bibliografía

[1] Wasim Abbas. *Step by Step Skeletal Animation in C++ and OpenGL, Using COLLADA*. `http://www.wazim.com/Collada_Tutorial_1.htm`. Accessed: 2018-08-11.

[2] Wasim Abbas. *Step by Step Skeletal Animation in C++ and OpenGL, Using COLLADA*. `http://www.wazim.com/Collada_Tutorial_2.htm`. Accessed: 2018-08-11.

[3] Ilya Baran y Jovan Popović. "Automatic Rigging and Animation of 3D Characters". En: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH '07. San Diego, California: ACM, 2007. DOI: `10.1145/1275808.1276467`. URL: `http://doi.acm.org/10.1145/1275808.1276467`.

[4] Ilya Baran y Jovan Popović. *Penalty Functions for Automatic Rigging and Animation of 3D Characters*. 2007. URL: `http://people.csail.mit.edu/ibaran/papers/2007-SIGGRAPH-Pinocchio-Penalty.pdf`.

[5] Andy Beane. *3D Animation Essentials*. 1st. Alameda, CA, USA: SYBEX Inc., 2012. ISBN: 1118147480, 9781118147481.

[6] Samya Chattopadhyay. *Charlie Chaplin, Frame Rates & Time Lapse*. `http://www.linkedin.com/pulse/charlie-chaplin-frame-rates-time-lapse-samya-chattopadhyay`. Accessed: 2018-08-15.

[7] Stephanie Cheng. "Human Skeleton System Animation". Tesis de mtría. University of Zagreb, Faculty of Electrical Engineering y Computing, 2017.

[8] Olivier Dionne y Martin de Lasa. "Geodesic Voxel Binding for Production Character Meshes". En: *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '13. Anaheim, California: ACM, 2013, págs. 173-180. ISBN: 978-1-4503-2132-7. DOI: `10.1145/2485895.2485919`. URL: `http://doi.acm.org/10.1145/2485895.2485919`.

[9] J. Dirksen. *Three.Js Essentials*. Community experience distilled. Packt Publishing, 2014. ISBN: 9781783980864.

[10] Sarah F. Frisken y col. “Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics”. En: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, págs. 249-254. ISBN: 1-58113-208-5. DOI: `10.1145/344779.344899`. URL: `http://dx.doi.org/10.1145/344779.344899`.

[11] Michael Gleicher. “Comparing Constraint-Based Motion Editing Methods”. En: *Graph. Models* 63.2 (mar. de 2001), págs. 107-134. ISSN: 1524-0703. DOI: `10.1006/gmod.2001.0549`. URL: `http://dx.doi.org/10.1006/gmod.2001.0549`.

[12] Jason Gregory. *Game Engine Architecture, Second Edition*. 2nd. Natick, MA, USA: A. K. Peters, Ltd., 2014. ISBN: 1466560010, 9781466560017.

[13] LLVM Developer Group. *The LLVM Compiler Infrastructure*. URL: `https://llvm.org/`.

[14] Inc Khronos Group. *OpenGL ES Overview*. URL: `https://www.khronos.org/opengles/`.

[15] Vladislav Kraevoy y Alla Sheffer. “Mean-Value Geometry Encoding”. En: 12 (jun. de 2006), págs. 29-46.

[16] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. *See* `https://llvm.org/`. Tesis de mtría. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, dic. de 2002.

[17] N. Magnenat-Thalmann, R. Laperrière y D. Thalmann. “Joint-dependent Local Deformations for Hand Animation and Object Grasping”. En: *Proceedings on Graphics Interface ’88*. Edmonton, Alberta, Canada: Canadian Information Processing Society, 1988, págs. 26-33. URL: `http://dl.acm.org/citation.cfm?id=102313.102317`.

[18] K. Matsuda y R. Lea. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*. Addison-Wesley, 2013. ISBN: 9780321902924.

[19] Michael Meredith y Steve Maddock. *Motion Capture File Formats Explained*. Inf. téc. University of Sheffield, Department of Computer Science, ene. de 2001.

[20] Mr.doob. *Three.js JavaScript 3D library*. URL: `https://github.com/mrdoob/three.js/`.

[21] Ramakrishnan Mukundan. *Advanced Methods in Computer Graphics: With examples in OpenGL*. Springer, 2012. ISBN: 1447123395, 9781447123392.

[22] Tony Parisi. *Programming 3D Applications with HTML5 and WebGL: 3D Animation and Visualization for Web Pages*. 1st. O’Reilly Media, Inc., 2014. ISBN: 1449362966, 9781449362966.

[23] Nadine Abu Rumman y Marco Fratarcangeli. “State of the Art in Skinning Techniques for Articulated Deformable Characters”. En: *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications: Volume 1:*

GRAPP. GRAPP 2016. Rome, Italy: SCITEPRESS - Science y Technology Publications, Lda, 2016, págs. 200-212. ISBN: 978-989-758-175-5. DOI: `10.5220/0005720101980210`. URL: `https://doi.org/10.5220/0005720101980210`.

[24] Sculpteo. *OBJ File : Color 3D Printing File Format*. `https://www.sculpteo.com/en/glossary/obj-file-3d-printing-file-format/`. Accessed: 2018-09-06.

[25] Peter Shirley y Steve Marschner. *Fundamentals of Computer Graphics*. 3rd. Natick, MA, USA: A. K. Peters, Ltd., 2009. ISBN: 1568814690, 9781568814698.

[26] James M. Van Verth y Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications, Second Edition: A Programmer's Guide*. 2.ª ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123742978, 9780123742971.

[27] Lawson Wade y Richard E. Parent. "Automated Generation of Control Skeletons for Use in Animation". En: *Vis. Comput.* 18.2 (abr. de 2002), págs. 97-110. ISSN: 0178-2789. DOI: `10.1007/s003710100139`. URL: `http://dx.doi.org/10.1007/s003710100139`.

[28] Haolei Wang. "Using density-based clustering to improve skeleton embedding in the pinocchio automatic rigging system". Tesis de mtría. Manhattan, Kansas: Department of Computing and Information Sciences College of Engineering., Kansas State University, 2012.

[29] Inc. Wavefront Technologies. *The Advanced Visualizer User's Guide. Apendix B1*. 1991. URL: `https://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf`.

[30] Alon Zakai. "Emscripten: An LLVM-to-JavaScript Compiler". En: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, págs. 301-312. ISBN: 978-1-4503-0942-4. DOI: `10.1145/2048147.2048224`. URL: `http://doi.acm.org/10.1145/2048147.2048224`.