**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA**
**DEPARTAMENTO DE ELECTRÓNICA**
**VALPARAÍSO - CHILE**




# Simulación de prendas en la dinámica corporal computada en la nube sobre videos monoculares RGB

**Italo Ariel Salgado Sepúlveda**

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO**

**PROFESOR GUÍA:** **Marcos Zúñiga Barraza.**

**PROFESOR CORREFERENTE:** **Gonzalo Carvajal Barrera.**

# *Agradecimientos*

En primer lugar, agradezco a mis padres: su determinación me ha mostrado la fuerza que una convicción resuelta y resiliente pueden lograr, bajo toda condición inicial. A todos mis amigos que se han hecho presente de manera constante pese a la distancia y las circunstancias: la sumatoria de sus perspectivas de vida me han ayudado a comprender las caras y fenómenos de lo que significa vivir en comunidad, además de otorgarme consejos y la confianza necesaria en los momentos más críticos. Agradezco también a mi profesor guía: su constante apoyo en el desarrollo de este trabajo ayudó a encausar y materializar este proyecto.

## *Resumen*

Las nuevas tecnologías de captura e interpretación de movimientos enfocadas en inteligencia artificial están abriendo espacios de aplicación de nuevas tecnologías y soluciones a través de servicios. La interpretación del movimiento de las personas en videos permite crear aplicaciones de consumo para tareas que anteriormente solo se creían posibles de manera presencial.

Este trabajo busca aproximarse a una propuesta que permita inferir el movimiento de una persona en un video monocular e imprimir una prenda sobre un motor de juego: Unreal Engine. El avance de las tecnologías de simulación permite mostrar movimiento y dinamismo realista en cualquier prenda diseñada.

Para lograr esto, se desarrolla un conjunto de soluciones basadas en una arquitectura por capas que permitirá procesar cada resultado por separado. La arquitectura de tres capas incluye la inferencia del movimiento del usuario, la extracción de un fondo repintado y la segmentación del usuario. Estos tres resultados debe ser importados a Unreal Engine: el fondo repintado y el usuario segmentado se importan como un asset, mientras que el movimiento del usuario y el modelo de malla debieron ser adaptados a través de Python utilizando la biblioteca de Blender como módulo, dado que las soluciones exploradas no integran la correspondiente transformación.

Finalmente se materializa la solución dentro de Unreal Engine: esto implica la creación de un level con todos los assets que corresponden, además de configurar correctamente el motor de simulación Chaos Cloth. Se evalúa la solución para cada tipo de proceso, concluyendo que las etapas de segmentación y repintado son certeras, pero que la importación de los resultados del módulo de captura de cuerpo requiere aún desarrollo vinculado a la interpretación de la disposición espacial del maniquí con respecto a la posición de la persona.

## *Abstract*

New motion capture and interpretation technologies focused on artificial intelligence are opening spaces for the application of new technologies and solutions through services. The interpretation of the movement of people in videos allows the creation of consumer applications for tasks that were previously thought to be possible only in person.

This work seeks to approach a proposal that allows to infer the movement of a person in a monocular video and print a garment on a game engine: Unreal Engine. The advance of simulation technologies allows to show realistic movement and dynamism in any designed garment.

To achieve this, a set of solutions is developed based on a layered architecture that will allow processing each result separately. The three-layer architecture includes the inference of the wearer's motion, the extraction of a repainted background and the segmentation of the wearer. These three results must be imported into Unreal Engine: the repainted background and the segmented user are imported as an asset, while the user movement and the mesh model had to be adapted through Python using the Blender library as a module, since the explored solutions do not integrate the corresponding transformation.

Finally, the solution is materialized within Unreal Engine: this implies the creation of a level with all the corresponding assets, as well as the correct configuration of the Chaos Cloth simulation engine. The solution is evaluated for each type of process, concluding that the segmentation and repainting stages are accurate, but that the import of the results of the body capture module requires further development linked to the interpretation of the spatial arrangement of the mannequin with respect to the position of the person.

# Índice general


*1.. Introducción* . . . . . 3

1.1. Contexto . . . . . 3

1.2. Planteamiento del problema . . . . . 3

1.2.1. Características del video a procesar y su salida . . . . . 4

1.2.2. Requisitos de arquitectura en el motor de juego . . . . . 4

1.3. Hipótesis . . . . . 7

1.4. Acercamiento a la solución . . . . . 7

1.5. Objetivos . . . . . 9

1.6. Estructura del Documento . . . . . 10

*2.. Estado del arte* . . . . . 12

2.1. Marco Teórico . . . . . 12

2.1.1. Body with Hand Tracking (BwHT) . . . . . 12

2.1.2. User Removal (URem) . . . . . 15

2.1.3. Video Inpaiting (VIn) . . . . . 18

2.1.4. Unreal Engine (UE) como motor de juego para realizar simulaciones . . . . . 19

2.1.5. UE4 en containers junto a Pixel Streaming . . . . . 22

2.2. Alternativas de solución . . . . . 25

2.2.1. Evaluación de soluciones presentes en el mercado . . . . . 25

2.2.2. Soluciones disponibles para Body with Hand Tracking (BwHT) 28

2.2.3. Soluciones disponibles para User Removal (URem) . . . . . 37

2.2.4. Soluciones disponibles para Video Inpainting (VIn) . . . . . 40

2.3. Resumen del EoA: Predominancia de un enfoque extensivo en datos complementado con una técnica extensiva en cómputo . . . . . 43

*3.. Diseño de la solución y evaluación de herramientas* . . . . . 46

3.1. Modelo general de la solución . . . . . 46

3.2. Pruebas preliminares: Etapas y Testing . . . . . 48

3.2.1. Pruebas de algoritmos de BwHT . . . . . 49

3.2.2. Pruebas de algoritmos de URem . . . . . 57

3.2.3. Pruebas de algoritmos de VIn . . . . . 59

3.2.4. Condiciones de ambiente en pruebas locales . . . . . 61

3.2.5. Selección de los algoritmos para cada problema . . . . . 62

3.3. Limitaciones, perspectivas de uso y compromisos entre tiempo de inferencia y precisión de los resultados . . . . . 63

*4.. Integración e implementación de la solución* . . . . . 65

4.1. Implementación de la solución en Unreal Engine . . . . . 65

4.1.1. Importar BwHT a UE . . . . . 66

4.1.2. Acondicionamiento del algoritmo de VIn . . . . . 66

4.1.3. Implementación del modelo de 3 capas en Unreal Engine . . . 68

4.2. Evaluación de la solución . . . . . 75

4.2.1. Herramientas de evaluación: Pruebas QA . . . . . 75

4.2.2. Pruebas a realizar . . . . . 76

4.2.3. Métricas . . . . . 78

4.2.4. Evaluación . . . . . 87

*5.. Conclusión y trabajo futuro* . . . . . 89

5.1. Conclusiones . . . . . 89

5.2. Trabajo Futuro . . . . . 91

# 1. INTRODUCCIÓN

## *1.1. Contexto*

La disponibilidad de servicios online ha explotado, y con ello se ha diversificado la matriz de servicios disponibles en diversas aristas. Los últimos años han demostrado que servicios ligados a la detección e interpretación de pose de las personas están en auge y han mejorado exponencialmente [22].

La proliferación de algoritmos de visión por computadora, el avance técnico del hardware para computarles y las facilidades que otorgan las herramientas en la nube, han democratizado las aplicaciones en todos los niveles: desarrolladores que se desempeñan en tópicos disjuntos buscan soluciones aplicadas que usen tecnologías de reconocimiento o seguimiento. Los dispositivos móviles y su alta disponibilidad generan oportunidades de integración de soluciones en diversos ámbitos del quehacer diario.

La mejora de estas tecnologías y la necesidad de proveer un servicio virtualizado de prueba de ropa a través de hardware disponible para cualquier usuario, convergen al inducir una solución de software que se presentará en esta memoria.

## *1.2. Planteamiento del problema*

Este documento presenta una de las aplicaciones que las tecnologías en visión pueden lograr: se desea realizar una alternativa para simular prendas en un sistema virtualizado de simulación física a través de cualquier sistema que disponga de un navegador.

El proyecto nace como una solución que desea entregar la empresa Nexar, entidad dedicada al marketing digital interactivo y soluciones inteligentes aplicadas a la industria. Con ello, los requisitos de implementación y el alcance de las solución mostrada en el documento están restringidas por los requisitos funcionales que esta empresa requiere como solución final, acorde a su proceso productivo.

### *1.2.1. Características del video a procesar y su salida*

El video a procesar será un video grabado de manera vertical, con una calidad FHD/HD (1920x1080/1280x720) a 30 fps. Estas son las características estándar que se espera al utilizar un *smartphone* para grabar un video a procesar. El video estará disponible antes de iniciar el procesamiento. Se espera que la mayoría del tiempo la persona se mantenga en el centro de la grabación y realice movimientos sobre su eje asociados al hecho de que se está probando una tela que desea mirar desde distintas perspectivas. La duración total del video será de aproximadamente 20 segundos.

El tiempo de procesamiento estará sujeto a un compromiso entre ejecución y la precisión que cada técnica ofrezca.

No se requerirá una interfaz de uso (i.e. que responda al teclado y al mouse) para visualizar el video resultante: a través de la herramienta Pixel Streaming se reproducirá un bucle del resultado, en una dirección web dada, accesible desde el navegador. La implementación de una interfaz escapa al desarrollo de esta memoria.

### *1.2.2. Requisitos de arquitectura en el motor de juego*

La solución dentro del motor de juego debe incluir una arquitectura en capas que permita realizar operaciones sobre cada capa de entrada de Unreal Engine por separado para mantener un resultado coherente en el video de salida. La Figura 1.1 muestra un diagrama inducido por los requisitos que se presentarán en breve. Las capas se verán sobrepuestas desde la perspectiva fija (la cámara en la Figura 1.1), de manera de ver la ropa simulada exactamente sobre el usuario segmentado, y esta capa a su

vez se verá exactamente sobre el fondo repintado. Esta arquitectura es útil ya que, por ejemplo, si se desea remover la ropa que actualmente está utilizando el usuario (como una chaqueta gruesa) para sobreponer un vestido emulado, se necesita realizar un repintando de fondo en las zonas donde ni la antigua ropa removida ni la simulada esté presente.

Los requisitos se describirán por capa. El diagrama representado en la Figura 1.1 muestra las 3 capas necesarias en la arquitectura del juego, mientras la Figura 1.2 muestra descripciones que las resumen. A grandes rasgos, podemos describir las capas de la siguiente manera:




*Fig. 1.1:* Diagrama de las capas requeridas en el simulador Unreal Engine.

- Escena fondo repintado: se procesa el video de entrada de manera de lograr extraer al usuario de todos los *frames* y luego repintar esas zonas de manera coherente al contexto de fondo. Al extraer alguna característica del usuario presente en la escena de usuario segmentado (la siguiente capa), como por ejemplo, eliminando una chaqueta gruesa que ya tenga puesta y luego solicitar probarse un vestido ligero dado por la simulación, las diferencias entre la chaqueta extraída

y el vestido ligero simulado formarán una zona donde no existiría información. El fondo repintado cumple la función de ayudar a aportar naturalidad en la solución final al mostrar en estas zonas un fondo coherente. Cualquier borrado sobre la capa de segmentación mostrará al fondo repintado con esta configuración.

- Escena usuario segmentado: Esta capa mostrará al usuario segmentado en todos los *frames* del video de entrada, con el resto de la información nula/transparente. Esta capa podrá ser procesada para segmentar ropa o partes que se deseen borrar de la persona y mostrar el Modelo 3D de la siguiente capa a describir en su lugar. En el contexto de esta memoria, basta con entregar el usuario segmentado sin realizar ninguna otra operación asociada a la segmentación.

- Escena Modelo 3D: esta capa contiene un avatar 3D que replica los movimientos del video del usuario. Sobrepuesto al avatar se encuentra una prenda de interés a ser simulada (i.e. está vestido con la prenda), con su correspondiente respuesta dinámica al movimiento. El avatar 3D será importado, provisto de manera externa y tendrá la forma del cuerpo del usuario, por lo que se asumirá disponible en todo momento. El avatar se configurará dentro de Unreal Engine para que sea transparente, y solo sea visible el movimiento de la prenda que se desea probar.

  Este modelo debe capturar tanto el cuerpo como las manos. La captura de manos es un requisito funcional de la empresa Nexar, para trabajos futuros.

- Perspectiva Fija: esta será la perspectiva dentro del motor de juego, donde las disposiciones de las 3 capas deben estar a una distancia y en una proporción correcta para lograr una vista que mantenga proporciones adecuadas, y observar la ropa simulada sobrepuesta sobre el video original.

El aporte principal de esta memoria es presentar una arquitectura de procesamiento externa al simulador, que entregue como resultado lo necesario para implementar la arquitectura de 3 capas requerida. Esta arquitectura de procesamiento presentará los

mejores algoritmos necesarios para entregar las entradas acondicionadas al motor de juego que permitan la simulación tal y como se describe.

## 1.3. Hipótesis

Con el contexto presentado, se induce el siguiente enunciado a verificar: *"Es posible a partir de un video monocular realizar un modelo simulado en un motor de videojuegos que permita una arquitectura de 3 capas donde se presente un fondo repintado sin el usuario, el usuario segmentado y un maniquí/avatar 3D que replique los movimientos del usuario"*.

- ¿Es viable estimar correctamente *joints* 3D desde un video monocular, sin disponer de un sensor de distancia?.
- ¿Que arquitectura de software es necesaria para transformar las salidas de un algoritmo de *tracking* para que puedan ser interpretados por el motor de videojuegos?.
- ¿Es lo suficientemente coherente el repintado efectuado sobre la capa de fondo para este contexto?.
- ¿Se puede procesar un video grabado de manera vertical de 20 segundos de duración, con una calidad FHD/HD (1920x1080/1280x720) a 30 fps en menos de 60 segundos para obtener la salida esperada sobre el motor de videojuegos?.

## 1.4. Acercamiento a la solución

El principal aporte de esta memoria es la selección y acondicionamiento de los algoritmos para materializar la arquitectura de 3 capas que se presentó como un requisito de implementación dentro del simulador. Esto incluye el tratamiento de las salidas de cada algoritmo para que puedan ser interpretadas por Unreal Engine, integrando las características necesarias para llevar la implementación a la nube, algo novedoso en el

contexto de Unreal Engine que el año 2021 ha disponibilizado las características que se requieren para esta implementación en su versión 4.27 (soporte de contenedores (*containers*), soporte de Pixel Streaming en *containers*, etc).

La Figura 1.2 es un diagrama simplificado que muestra la solución propuesta que le entregará a Unreal Engine todas las entradas necesarias.




*Fig. 1.2:* Diagrama simplificado de la solución final

Este proceso se inicia con un video grabado con un sistema monocular (e.g. smartphone), donde se desee probar la prenda emulada. Este video es procesado por los 3 algoritmos para lograr la arquitectura de 3 capas mostrada en 1.2. En primer lugar se ejecuta el algoritmo de estimación 3D FrankMocap, que incluye tanto las manos como el cuerpo. Se utiliza un *script* para lograr importa a Unreal Engine su salida, ya que este no integra esta característica. En segundo lugar utilizamos el algoritmo de segmentación de usuario, RobustVideoMatting (RVM), que entregará una máscara de

segmentación junto a un video del usuario ya segmentado. Finalmente, a través del algoritmo de repintado coherente DSTT, y la última máscara mencionada, podemos ejecutar un repintado para obtener un fondo sin el usuario. Estos resultados permitirán formar la arquitectura dentro de Unreal Engine buscada, posicionándolos correctamente.

La simulación de la prenda se realiza dentro de Unreal Engine, una vez exportada desde algún gestor de contenido 3D (como Blender). La calidad, peso y respuesta al movimiento se configura directamente en UE con la herramienta Chaos Cloth.

## *1.5. Objetivos*

Objetivo general:

- Desarrollar un sistema que a partir de un video monocular, genere un modelo simulado en un motor de videojuegos que permita una arquitectura de 3 capas donde se presente un fondo repintado sin el usuario, el usuario segmentado y un maniquí/avatar 3D que replique los movimientos del usuario.

Objetivos específicos:

- Inferir *joints* 3D que capturen los movimientos del usuario. Proponer una arquitectura que permita realizar todas las transformaciones necesarias para importarles a UE (Unreal Engine), e integrarlos en un avatar que replique esos movimientos.

- Lograr extraer el fondo del video original y repintar la zona donde antes estaba la persona.

- Integrar los *joints* capturados a un programa de renderización para lograr mostrar un avatar que replique los movimientos obtenidos.

- Implementar una arquitectura que permita realizar la integración de las salidas de cada algoritmo al motor de videojuegos.

- Realizar la segmentación del usuario a través de un algoritmo dado, que permita capturar de la manera más detallada la morfología del mismo.

Notar que el objetivo general es desarrollar un prototipo, estudiando la factibilidad del proyecto en conjunto con las herramientas disponibles en el estado del arte.

## *1.6. Estructura del Documento*

En primer lugar, se realiza una presentación a grandes rasgos de todos los contenidos. Luego, un estudio teórico del estado actual de los tópicos directamente relacionados para dar solución al problema (esto incluye el ambiente de simulación UE4), una presentación con pormenores del desarrollo y los encauses fructíferos (y no tanto) que dan solución al problema. Finalmente, se presentan conclusiones ligadas a los resultados esperados y los logrados. En particular:

- Capítulo 1, Introducción: se muestra el problema y sus restricciones, y se propone una solución. Se materializan objetivos que muestran y encausan los hitos esenciales de desarrollo.

- Capítulo 2, Estado del arte: se realiza un estudio de las técnicas actuales que dan solución al problema, mostrando un marco teórico y una etapa de discusión sobre la preselección de las mejores soluciones.

- Capítulo 3, Desarrollo de la solución: se muestra un bosquejo general de la solución utilizada, junto a sus etapas y decisiones de diseño para la solución final. Se muestran pruebas locales para cada algoritmo de interés. Basado en estas métricas se muestra una selección de los mejores algoritmos para cada tópico, que se integrarán al motor de videojuegos. Luego se realiza una etapa de evaluación de la solución para determinar si la hipótesis inicial fue resuelta, y una presentación cualitativa/cuantitativa que validen los algoritmos utilizados.

- Capítulo 4, Conclusiones: se expone un resumen generalizado que muestra resultados generales del desempeño de la solución, sus pormenores y el resultado final en términos de calidad. Un anexo de trabajo futuro muestra técnicas o soluciones adicionales que pueden mejorar la funcionalidad del producto desarrollado en esta memoria.

# 2. ESTADO DEL ARTE

En este capítulo, se presenta el estado actual de los tópicos que se relacionan directamente con el problema. En primer lugar, se muestra un marco teórico que define las principales técnicas de interés además de características del motor de juego elegido y sus funciones de simulación, para luego realizar comentarios que encausan la selección de algoritmos, la calidad esperable de los resultados y las opciones que pueden ser usadas en este contexto.

## *2.1. Marco Teórico*

### *2.1.1. Body with Hand Tracking (BwHT)*

#### *Contexto*

La *Monocular human pose estimation* es una tarea fundamental y desafiante para la comunidad que trabaja en visión por computador. Su objetivo es predecir la información de pose humana, como la localización espacial de los *body joints* y los parámetros asociados a la forma del cuerpo a través de un video grabado con un sistema monocular.

Una de las limitaciones trascendentales de este problema es, dado el enfoque *deep*[1] que entrega las mejores soluciones, encontrar *datasets* completos y no sesgados. Recientemente se han desarrollado muchos *datasets* que pueden ser utilizados para esta

[1] *Deep* es una contracción común para *Deep Learning*.

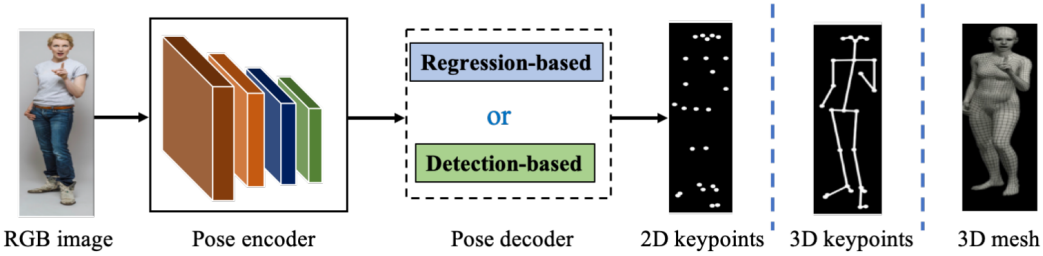



*Fig. 2.1:* Framework típico para la estimación de pose de una sola persona.

tarea, como COCO[2][19], MPII[3][1], Human3.6M[4][14] y 3DPW[5][31].

El cuerpo humano puede entenderse como un sistema flexible y poco rígido, con muchos grados de libertad, por lo cual predecir la pose real con una cámara monocular presenta muchos retos: posturas extrañas/complejas, interacción persona/objeto, oclusión, malas resoluciones, blur, etc. Para resolver estos problemas, la representación poderosa que ofrece el enfoque de deep learning es esencial para extraer la mayor cantidad de pistas para la estimación de la pose. Las redes en general tienen una arquitectura/configuración similar, prevalentemente enfocadas en una primera etapa *pose encoder* (un extractor de características generales) seguido de un *pose decoder* (caracterización de la salida de los *2D/3D-keypoints* o *3D-mesh*). En la Figura 2.1 se resume la arquitectura mencionada.

En los *pose decoders* (las salidas de la mayoría de los frameworks), tenemos métodos basados en extraer un mapa de características (o *heatmap*) o en directamente entregar a la salida los parámetros de interés (métodos regresivos). Las representaciones condicionan la arquitectura de entrada y salida de cada uno de los algoritmos, por lo que es importante distinguirles.

[2] Common Objects in Context
[3] Max-Planck-Institut für Informatik Dataset
[4] Human3.6m: Large scale datasets and predictive methods for 3D human sensing in natural environments
[5] 3D Poses in the Wild Dataset

*Representaciones*

Se usan diversas representaciones del cuerpo humano para describir poses complejas en diferentes situaciones. En la Figura 2.2 podemos ver las principales técnicas de representación de manera gráfica, que serán explicadas a continuación.

**Modelos Basados en Keypoints**

(a) Representaciones Basadas en coordenadas 2D/3D: representación de coordenadas 2D/3D de los *joints* de interés del modelo. Esta es la forma de modelar clásica e intuitiva.

(b) 2D/3D *heatmap*: para representar las coordenadas de manera apropiada para ser usada por una red convolucional. Por ejemplo, en (b) tenemos una representación Gaussiana a través de un *heatmap* de cada punto de interés (*joint*).

(c) *Orientation Maps*: métodos de representación con dirección/orientación. Pueden entenderse como modelos de *joints* unidos con vectores que asocian dos *keypoints*. OpenPose[3] es un buen ejemplo de esta técnica.

(d) Estructura de Huesos/vectores con jerarquía: una combinación de *joints* y huesos como vectores (puede ser 3D o 2D). Evoskeleton[17] es un buen ejemplo de esta técninca 3D. Cada vector-hueso se define desde un *keypoint* padre hacia un *keypoint* hijo, dando la noción de jerarquia.

**Modelos Basados en la representación**

(e) Modelo basados en representación de Partes Volumétricas: Se busca una representación de la estructura del cuerpo humano. Contiene más información que el modelo de *keypoints*. Por ejemplo, en Cheng[7], cada extremidad es representada por un cilindro. Cada cilindro se ubica alineando los centros de la superficie superior e inferior con puntos clave 3D de la extremidad. Otra representación a través de elipsoides es EllipBody[33] que se muestra en el modelo rosado.

(f) Modelos estadísticos 3D del cuerpo humano: similar al modelo anterior, pero describiendo una malla del cuerpo (un *body mesh*) que entrega más detalles al incluir la forma característica del cuerpo. Estos modelos pueden ser buscados como SMPL, *skinned multi-person linear model*. Disocia la forma y la pose del cuerpo humano, y codifica la 3D-mesh en otros parámetros. Este es un modelo de 23 *joints*, que se amplía dependiendo del contexto (e.g. SMLP+H incluye las manos).

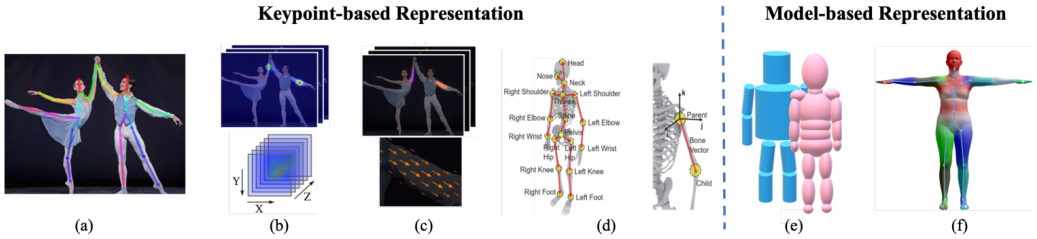



*Fig. 2.2:* Representaciones más usadas en BwHT.

Las representaciones y obtenciones de los *joints* 3D varían mucho dependiendo de la técnica. En general, predominan los algoritmos que, a través de los *joints* 2D obtenidos en un paso anterior, intentan extrapolar la información para lograr la estimación de los 3D. El problema de estimación de pose 2D está bien abordado y su predicción es lo suficientemente certera en la mayoría de los contextos. Con ello, estos algoritmos intentan aprovecharse de los avances 2D para lograr darle profundidad a estas formas de estimación. Otras aproximaciones incluyen estimar los *joints* 3D directamente a través de un *encoder*, un *regressor*, y un estimador de parámetros SMPL. Esto se abordará directamente cuando se analice particularmente cada algoritmo.

### *2.1.2. User Removal (URem)*

#### *Contexto*

El objetivo en este contexto es lograr separar una capa al usuario y otra el fondo. Con ello, es esencial la tarea de *video matting*. Esta es una técnica para separar el

video en dos (o más) capas, generalmente de primer plano (*foreground*, FG) y de fondo (*background*, BG), y generar *alpha mates* (máscara de primer plano) que determinan la combinación de las capas. Una de sus principales aplicaciones es la edición de video, ya que como se puede intuir, permite sustituir el fondo o procesar las capas individualmente. Así, esta sección se relaciona con dos tópicos centrales: el *Image Matting* y el *Video Matting*.

En general, se busca una técnica que ofrezca resultados más precisos de segmentación al estar enfocado en personas (que tienen características muy demandantes en términos de segmentación en torno al pelo, por ejemplo) que un código de inferencia de máscaras generalizado (como Mask R-CNN, técnica de segmentación poligonal multipropósito).

### *Image Matting y Video Matting*

*Image Matting* es el proceso de estimar con precisión el objeto en primer plano en imágenes y vídeos. Es una técnica muy importante en las aplicaciones de edición de imágenes y vídeos, especialmente en la producción de películas para crear efectos visuales.

El *Matting* normalmente produce una imagen matte[6] que se puede usar para separar el primer plano del fondo en una imagen determinada. El matte también se puede usar para combinar un primer plano dado con un fondo diferente para producir una nueva imagen plausible. El objetivo de este proceso es producir un *alpha matte* que se pueda usar para extraer el primer plano de una imagen. La tarea es desafiante porque en cada píxel necesitamos estimar el primer plano, el fondo y la opacidad del primer plano (matte).

Los métodos de *image matting* basados en el muestreo (*sampling-based image matting methods*) construyen las estimaciones a priori de *foreground* o *background* utilizando los píxeles muestreados para inferir los *alpha values*, mientras que los métodos basados en la afinidad propagan los valores alfa de los píxeles *foreground*

[6] Más detalles en https://es.wikipedia.org/wiki/Mate_(cinematograf%C3%ADa)

y *background* conocidos a los píxeles de la "región desconocida", basándose en la puntuación de afinidad.

Los métodos *matting deep-based* suelen entrenar una red neuronal convolucional *encoder-decoder* para predecir los *alpha values* o los colores de primer plano/fondo.

El problema central de el *video matting* es cómo obtener *alpha mattes* temporalmente coherentes (además de los problemas asociados a *image matting*) con una suficiente precisión como para denominar al resultado plausible. Ningún algoritmo puro (solo utilizando *optical flow*, solo con FCNN[7]) parece fructífero en resolver este problema, a menos que se provea al algoritmo de una imagen de *background* limpia sin una persona, para luego, una vez que el objeto de interés se presente separarle al *foreground* [18](BMv2).



*Fig. 2.3:* Resultados de algoritmo Video Matting de DVM[30].

La perspectiva que plantea la solución a este problema es usar *Attention Models*[8], enfocados en algoritmos de segmentación y matting. Estos proporcionan una forma eficaz para que las redes refuercen características correlacionadas y supriman el ruido, con lo cual el rendimiento de la segmentación se eleva mucho en comparación a métodos tradicionales [30].

[7] *Fully Convolutional Neural Network*: un tipo de red neuronal artificial utilizada en el reconocimiento y procesamiento de imágenes, como también en el procesamiento de lenguaje natural.

[8] En las redes neuronales, *attention* es una técnica que imita la atención cognitiva. El efecto mejora algunas partes de los datos de entrada mientras disminuye otras partes; la idea es que la red debería dedicar más atención a esa pequeña pero importante parte de los datos. Aprender qué parte de los datos es más importante que otras depende del contexto y debe ser entrenado.

### *2.1.3. Video Inpaiting (VIn)*

#### *Contexto*

Los algoritmos de *video inpainting* tienen por objetivo rellenar los espacios faltantes de una imagen con una apariencia espacio temporal factible. Los principales problemas a tener en cuenta en el enfoque *deep* de la solución son el alto costo de los algoritmos junto con las texturas poco realistas generadas, o porosas, que son antinaturales.

Ejemplos de implementación de algoritmos de inpaiting los tenemos en la Figura 2.4, que muestra un ejemplo de repintado y una comparación de características desde la perspectiva de DSTT [20], algoritmo comentado más adelante.

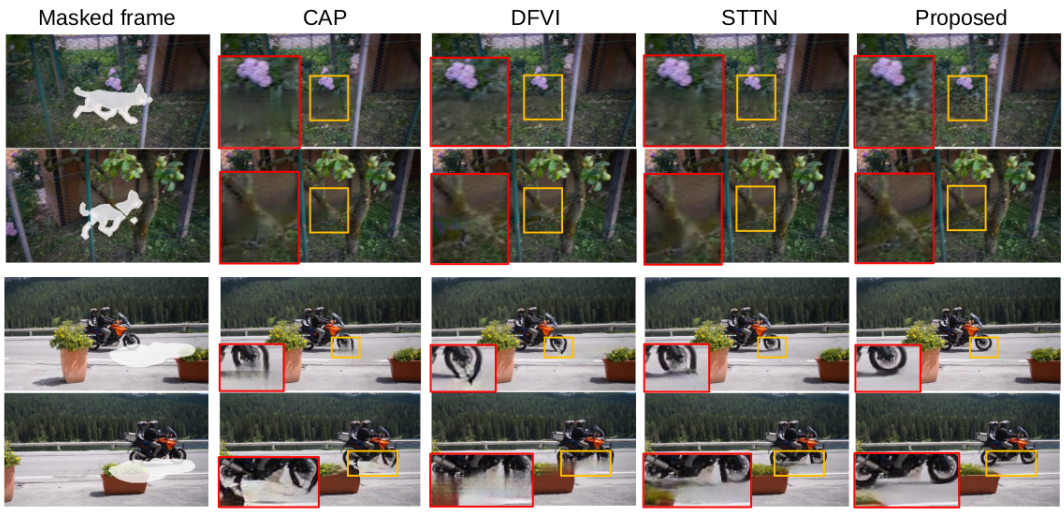



*Fig. 2.4:* 4 ejemplos de algoritmos de video inpainting.

#### *Deep Video Inpaiting*

Antes de la prevalencia del enfoque *deep*, era usualmente extendido el uso de métodos de repintado basado en parches. Estos son muy extensivos en cómputo, ya que computar pixel a pixel requiere un recorrido costoso al acceder tantas veces a todos los sectores de la imagen a repintar.

Los métodos *deep* han cambiado completamente el panorama y aumentaron el rendimiento y la eficiencia exponencialmente. En general, los métodos se dividen en los enfocados en propagación (utilizan el *field flow* para concluir características específicas) y métodos basados en la inferencia (i.e. *Fully Connected Networks*)[20].

Mientras el *flow field* extrae la información de movimiento, el enfoque de inferencia utiliza una integración de convolución 2D y 3D en una FCNN para inferir contenidos faltantes en secuencias de video.

Para lograr una mejor inferencia entre *frames* que tienen una correspondencia pero que están muy alejados en el tiempo, se presentan las redes *attention-based* que retienen características entre *frames*. Este es el enfoque de los principales algoritmos en el estado del arte y de nuestro interés [38] [20].

Hay dos variantes principales de este mecanismo. Una de ellas es el modelo de atención por canales (*channel-wise self-attention*). Dado un tensor de características de entrada, aprovecha la agrupación de promedios globales y la capa totalmente conectada para inferir un vector de pesos en función del canal para modular los *feature maps*.

Los otros enfoques se traducen en investigar de manera no-local. Se enfocan en la correlación espacio-temporal reforzando la consistencia de los *feature maps*. El enfoque de atención por canales es ampliamente adoptado en la segmentación de imágenes, y los algoritmos que se estudiarán se enfocan en esta técnica.

### *2.1.4. Unreal Engine (UE) como motor de juego para realizar simulaciones*

Unreal Engine es un *motor de juego* creado por la compañia Epic Games, utilizado para la reproducción y simulación de ambientes físicos con objetos que interactúan entre sí. Actualmente, la versión oficial y estable de UE es la 4.27. Esta escrito en C++, lo que implica que los *scripts* de manejo de entradas y salidas (I/O) también serán gestionadas con este lenguaje. Se encuentra bien documentado, tanto en videos y documentos propios de la compañía, como en soporte de la web en general. En lo sucesivo, se presentarán las principales características que se usarán, asociadas al

desarrollo en cuestión.

### *Animaciones*

El sistema de animación de Unreal Engine 4 (UE4) se compone de varias herramientas y editores de animación que mezclan la deformación de mallas (basada en el esqueleto) con la deformación de vértices (basada en la morfología), para permitir una animación compleja). Este sistema se puede utilizar para crear movimientos especiales personalizados como escalar salientes y paredes ( *Anim Montages*), aplicar efectos de daño o expresiones faciales (*Morph Targets*), controlar directamente las transformaciones de los huesos (*Skeletal Controls*), o crear máquinas de estado basadas en la lógica que determinan qué animación debe utilizar un avatar en una situación determinada (ésta es la perspectiva más útil en este contexto). Es posible complementar (o reemplazar) el uso de máquinas de estado con *scripts* directamente programados en C++.

En general, las animaciones y la malla asociada a esas animaciones se exportan en archivos **.fbx**. El siguiente diagrama muestra la relación entre un software que entregue los *joints* necesarios para ser interpretados por UE:

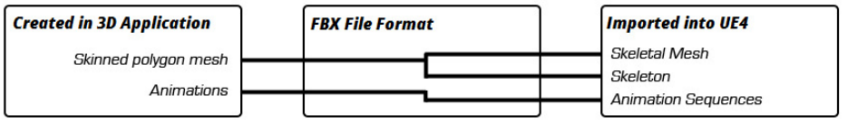



*Fig. 2.5:* Importar en UE.

La compatibilidad con la animación en el proceso de importación de un **.fbx** hace sencillo obtener *Skeletal Meshes* de aplicaciones 3D en Unreal Engine para su uso en simulaciones. Actualmente, sólo se puede exportar/importar una única animación para cada malla esquelética en un único archivo[9].

[9] Mayores detalles, en términos generales, los podemos encontrar en https://docs.unrealengine.com/4.26/en-US/AnimatingObjects/SkeletalMeshAnimation/Overview/.

*Colliders, Simulación de prendas en Unreal Engine 4*

Con la pretensión de dar naturalidad y percepción física a las prendas, es que UE otorga un motor de características que a través de una representación poligonal dinámica que interactúa con otros objetos, logra que se muestre, al observador, una respuesta dinámico-física plausible. El nivel de detalle de esta simulación es directamente proporcional a la calidad de los *colliders* y al nivel de detalle de la malla poligonal asociada a esa prenda.

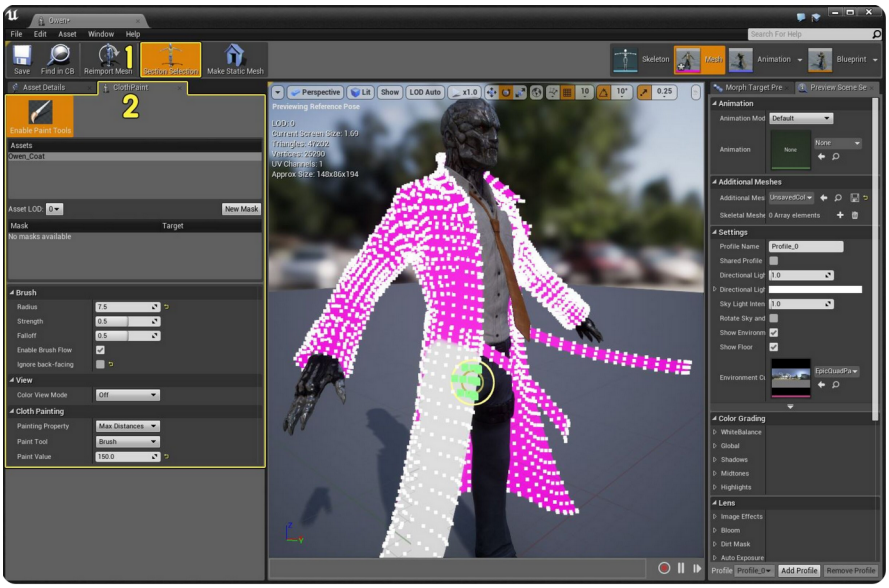


*Fig. 2.6:* Física poligonal de las prendas simuladas en UE4: el detalle poligonal dentro del motor de juego determina la capacidad de flexión de la vestimenta. Así, la naturalidad del movimiento aumenta en la medida que aumentan la cantidad de polígonos, y por consiguiente, aumenta la cantidad de cómputo requerido en su simulación.

Los *colliders* permiten todas las interacciones entre objetos, relacionadas con choques e interacciones de proximidad. Un objeto necesita *colliders* para responder correctamente al viento, al movimiento o a las caídas libres. Esto implica que todas las dinámicas físicas están asociadas a esta estructura y su nivel de detalle.

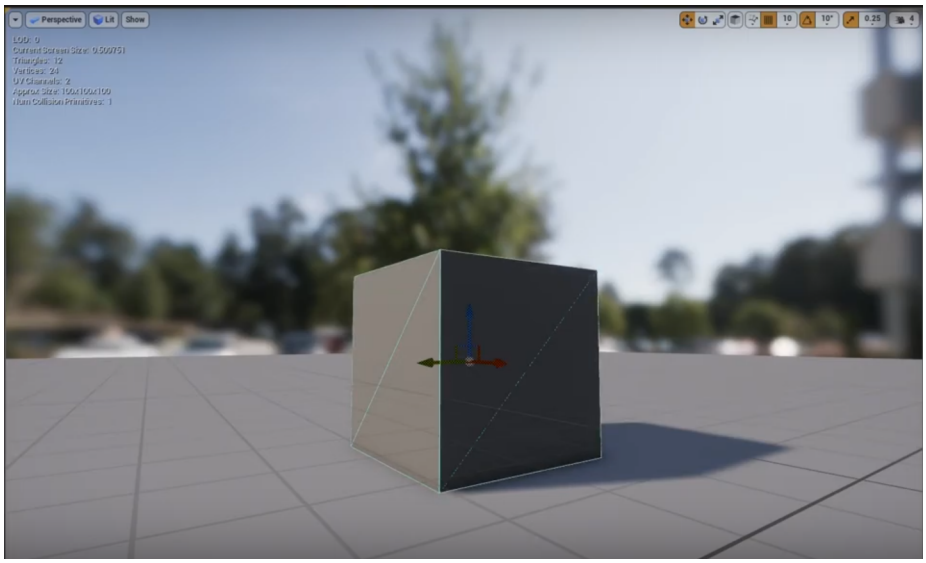


*Fig. 2.7:* Un *collider* simple para un cubo, compuesto de $4 * 2$ vértices. La aproximación a figuras más complejas (como la cabeza, los músculos o el torso) se componen de múltiples figuras volumétricas.

### *2.1.5. UE4 en containers junto a Pixel Streaming*

La perspectiva de interacción con el sistema (y su respectiva reproducción de video desde UE hacia otros dispositivos) es una ventaja reciente que se ofrece directamente como una opción en ambientes Windows, con un soporte parcial para Linux. Pixel Streaming permite mostrar en pantalla la simulación activa de UE en otro dispositivo, a través de una interfaz web.

Para Linux, tradicionalmente se ofrece una solución implementada por la comunidad, llamada *Pixel Streaming in Linux containers*[10] , un artículo que recopila como llevar la funcionalidad a este ambiente (ya que en general, el soporte es para Windows).

El año 2021, Epic ha liberado oficialmente su versión 4.27, donde se soportan los *containers* en Linux y lograr usar de forma simultánea Pixel Streaming. Esto facilita la implementación, la depuración de *bugs* y mejora la estabilidad de la herramienta

[10] https://adamrehn.com/articles/pixel-streaming-in-linux-containers/

en este entorno. La documentación de *containers* UE4[11] oficial aún se está construyendo, pero se basa fuertemente en proyectos muy reconocidos que ya tenían esta funcionalidad, como lo eran unrealContainers.com[12] y TensorWorks[13].

### *Pixel Streaming*

Con Pixel Streaming, se ejecuta una aplicación Unreal Engine empaquetada en un PC de escritorio o en la nube, junto con una pequeña pila de servicios web (véase WebRTC) que se incluyen con Unreal Engine. Los usuarios se conectan utilizando cualquier navegador web moderno en la plataforma que elijan (escritorio o smartphone) y transmiten los fotogramas renderizados y el audio de la aplicación Unreal Engine. No es necesario que los usuarios instalen o descarguen nada; es como transmitir un vídeo de YouTube o Netflix, salvo que los usuarios también pueden interactuar con la aplicación utilizando el teclado, el ratón, la entrada táctil e incluso la interfaz de usuario HTML5 personalizada que se crea en la página web del reproductor. Así, con la capacidad de servicios de streaming de video y sonido embebidas en UE, podremos lograr mostrar las imágenes sin grandes costos de implementación.

La página unrealcontainers.com muestra una forma de implementación clave en este contexto, sobre Linux. Aún no se ha actualizado, pero en breve, las instrucciones se orientarán a las nuevas funcionalidades de UE 4.27.

### *Containers*

Es importante tener en consideración la necesidad de exportar la arquitectura implementada en esta memoria hacia una solución escalable en la nube que permita ejecutarla desde cualquier dispositivo. Esto implica que se debe estudiar la principal forma de empaquetamiento (junto a sus limitaciones) utilizada en proyectos ejecutables en la nube: los contenedores.

[11] https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Containers/
[12] hrefhttps://unrealcontainers.com/
[13] https://tensorworks.com.au/

Los contenedores son una tecnología diseñada para empaquetar aplicaciones y sus dependencias de apoyo en una única unidad estándar que es portátil en toda la infraestructura informática, tanto en las instalaciones como en la nube.

De manera similar a las máquinas virtuales, los contenedores se almacenan en el disco como una imagen de contenedor, desde la cual se pueden ejecutar uno o más contenedores. A diferencia de las máquinas virtuales, los contenedores comparten un único núcleo de sistema operativo subyacente y pueden compartir datos comunes entre distintas imágenes de contenedor. Esto los hace computacionalmente ligeros y facilita un mayor número de despliegues simultáneos.

Hay dos tipos de imágenes de contenedores que se incluyen en Unreal Engine: *development images* (imágenes de desarrollo) y *runtime images* (imágenes de ejecución).

Las imágenes de desarrollo contienen el Editor de Unreal y las herramientas de construcción. Se utilizan para tareas que requieren el editor, como la construcción y el empaquetado de proyectos y plugins de Unreal Engine, la renderización de cinemáticas o la ejecución de *commandlets*.

Las imágenes en tiempo de ejecución sólo contienen las dependencias necesarias para ejecutar los proyectos empaquetados de Unreal Engine. Los desarrolladores amplían las *runtime images* añadiendo archivos para sus proyectos empaquetados de Unreal Engine y creando nuevos *containers* que pueden desplegarse en entornos de nube. Dado que las *runtime images* no contienen el editor ni las herramientas de compilación, su distribución está menos restringida que las *development images*.

A partir de Unreal Engine 4.27, se incluyen *development images* y *runtime images* oficiales con cada versión del motor. El código fuente de estos contedores se encuentra en el directorio Engine/Extras/Containers del código fuente de Unreal Engine, que puede descargarse de GitHub[14], o de Perforce[15] (aplicable solo a Windows). Las versiones preconstruidas de todos los contenedores se publican en el Registro de Conte-

[14] https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

[15] https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/SourceControl/Perforce/

nedores de GitHub y pueden descargarse sin necesidad de construirlas desde el código fuente.

Dentro de las *runtime images* disponibles, que son las útiles a la hora de implementar varias instancias del proyecto en la nube, tenemos 3 tipos: *runtime*, *runtime-pixel-streaming* y *runtime-windows*. Claramente, bajo nuestro contexto nos interesan los *containers* que presten el servicio de Pixel Streaming. La imagen *runtime-pixel-streaming* añade las dependencias necesarias para las aplicaciones de Pixel Streaming, incluyendo el soporte para la salida de audio. Como resultado, esta imagen es notablemente más grande que la *runtime* por defecto, aunque sigue siendo mucho más pequeña que la imagen de ejecución de Windows. Todas las imágenes presentadas anteriormente solo proveen funcionalidades para GPUs NVidia.

Otra tipo de *container* útil puede ser *pixel-streaming-signalling-server*, que se trata de una imagen contenedora de Linux que incluye la señalización **Cirrus** y el servidor web utilizados por el sistema Pixel Streaming.

## 2.2. Alternativas de solución

### 2.2.1. Evaluación de soluciones presentes en el mercado

#### Captura de Movimiento con otras herramientas

Se han probado herramientas relacionadas con la captura de movimientos, que ofrecen soluciones al principal problema en el contexto de esta memoria, la estimación de pose 3D. Estas herramientas son Mixamo, Kinetic y DeepMotion. Esta prueba se realizó sobre una persona bailando, que se muestra con su cuerpo entero, en una cámara centrada, sin oclusión y con una buena luminosidad. Su duración es de 30 segundos.

- Mixamo: página web soportada por Adobe, donde podemos descargar animaciones (un esqueleto y su respectivo movimiento) junto a un modelo de avatar

visual predeterminado, en formato FBX [16]. Es común que este archivo esté en binario, y no en formato ASCII, con lo cual, no podremos ver los *joints* asociados.

- DeepMotion: página web que captura los movimientos de un video entregado, y los representa sobre un avatar. El video a entregar debe cumplir con una cantidad bien definida de parámetros al ser grabado, como lo son: debe grabarse el cuerpo completo de la persona, luminosidad coherente, sin movimientos de la cámara, la persona debe posicionarse al centro, etc.

  El software funciona razonablemente bien en términos generales, pero la posición relativa (alineación entre el avatar y el suelo, siguiendo a la persona del video) del avatar creado se pierde transcurrido un tiempo. La posición de percepción de las extremidades y el cuerpo es acertada en la mayoría de los casos.

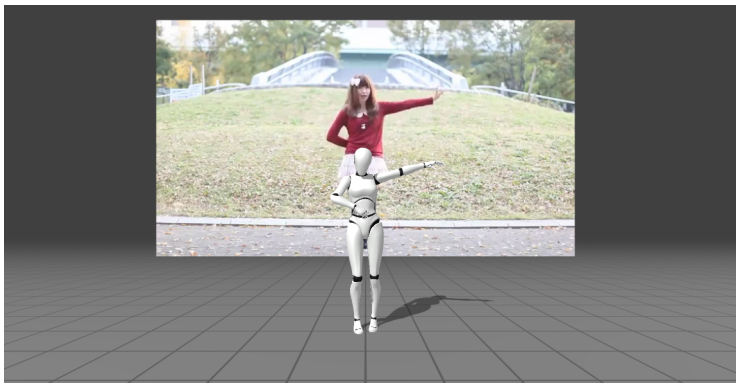


*Fig. 2.8:* Proyección del Avatar en DeepMotion 1: el avatar sigue inicialmente bien los movimientos y la posición relativa.

- Kinetix: página web que captura los movimientos en un video entregado, y los representa en un avatar. Se perciben fuertes problemas de posición relativa y de percepción de las extremidades y del cuerpo: existen errores de posición del cuerpo incluso de 180 grados (esto es, el avatar esta de espalda, cuando el video muestra a la persona de frente). No ofrece resultados razonables.

[16] *Filmbox*. Tipo de formato de archivos que contienen objetos de edición 3D y de contenido digital soportado por la mayoría de las plataformas de edición y de reproducción de movimiento.



*Fig. 2.9:* Proyección del Avatar en DeepMotion 2: mientras pasan los *frames*, decae la precisión de la posición relativa.

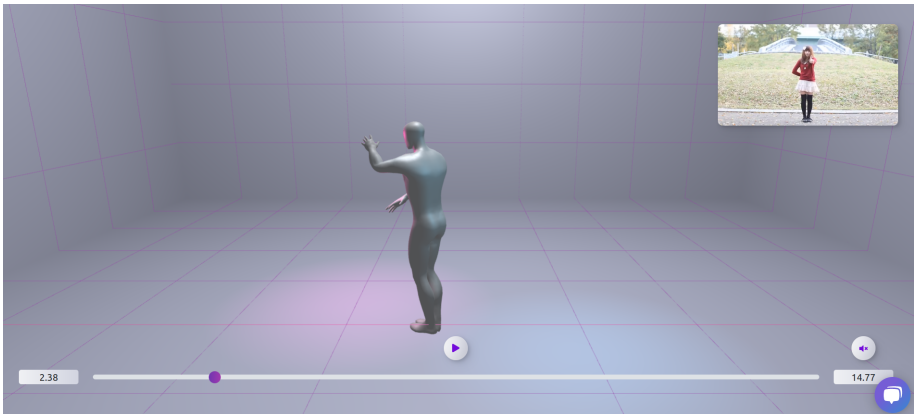



*Fig. 2.10:* Error en Kinetix: no se condice con la posición de rotación del cuerpo.

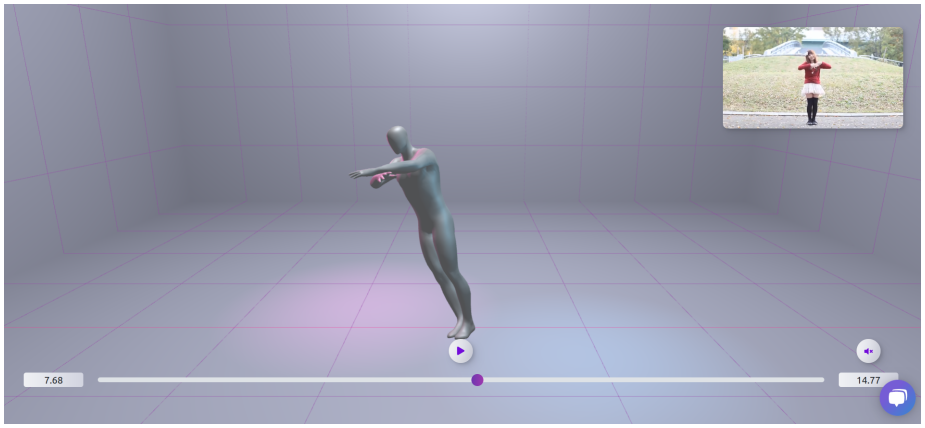



*Fig. 2.11:* Error en Kinetix: mala interpretación de la posición relativa del cuerpo.

### 2.2.2. Soluciones disponibles para Body with Hand Tracking (BwHT)

Las opciones exploradas abarcan perspectivas desde todos los ángulos posibles, buscando precisión, rapidez y buena capacidad de generalización en cada framework. Así, desde la perspectiva de mejor solución para BwHT, se realiza una evaluación tanto de variables cuantitativas como cualitativas. Estas últimas, incluyen los factores de interés en torno a la documentación, los comentarios de internet y los resultados observables a través de videos de muestra, junto con datos en los documentos relacionados (como los papers) que entreguen características de utilidad en nuestro contexto. Entre estas cualidades pueden incluirse la capacidad de entregar como output los *joints* 3D de manera sencilla, la entrega de un modelo SMPL por cada *frame*, las cualidades de implementación, etc.

Además, se buscan métodos de comparación cuantitativos. Estos incluyen la comparación de las métricas en los *datasets* de entrenamiento, la última actualización del git, el hardware necesario para su ejecución, la capacidad de ejecución en *frames*, y cualquier otra métrica que entregue una comparación mesurable.

Se presentan, junto a un resumen de características, tablas comparativas por algoritmo, que indican las principales características distintivas para cada uno y un resumen de sus métricas comunes a ser comparadas.

#### *Opciones BwHT*

**Blazepose** [2] es un algoritmo superligero enfocado en Smartphones. Intenta ser lo suficientemente ligero para ser implementado en tiempo real, a una buena cantidad de fps, usando como modelo de comparación OpenPose [3]. En general OpenPose encasilla mejor los objetos en contextos generalizados, mas, cuando se trata de posiciones de Yoga, o posiciones asociada a posturas de Yoga, Blazepose responde de manera superior. Esto se explica fácilmente dada la naturaleza del *dataset* usado: uno etiquetado manualmente de videos asociados al Yoga. Se presenta como una opción muy atractiva si la implementación se realiza de manera local, en hardware poco potente.

*Tab. 2.1:* Comparativa I de los algoritmos explorados en BwHT. Una cifra acompañada de L indica que la métrica es mejor mientras menor sea el valor, mientras que una H indica que a mayor valor la métrica es mejor.

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| BlazePose [2] | No posee. | Estimador de pose. Enfocado en ser una solución superligera. 33 body *keypoints*. Cuerpo entero, rostro, manos. 30 fps en Pixel2 (modelo superligero). Entrenado con *dataset* personalizado de yoga. Git: 22/dic/21. Video. |
| MotioNet [28] | [Human3.6M con MPJPE = 46.8], 10 % aproximado de sobreestimación de largo de huesos (poco en balance con los presentados). | Estimador de pose 3D de una sola persona. Especialmente diseñado para entregar 16 *joints* con solo error 10 % tamaño de huesos. No existe información de hardware y tiempo. Git: 14/ene/21. Video. |
| XNect [25] | [Human3.6m con MPJPE=63.6], [MPI-INF-3DHP con (3DPCK, AUC, MJPE)=(82.8H, 45.3H, 98.4L)], [3DPW con (MPJPE, PA-MPJPE)=(103.3L, 60.7L)] | Estimador de pose Multipersona. 3 Etapas: [Computa toda la imagen buscando sujetos en la escena, Entrega una pose 3D identificada, incluyendo *joints* ocluidos, Predictor del siguiente *frame*]. Enfoque ligero. 30 fps a 512x320 con GTx 1080Ti. Git: 26/jul/20. Video. Página. |
| MEVA [24] | [3DPW con (MPJPE, PA-MPJPE, ACC-ERR) = (86.9L, 54.7L, 11.6L)], [MPI-INF-3DHP con (MPJPE, PA-MPJPE, ACC-ERR) = (96.4, 65.4, 11.1)], [Human3.6m con (MPJPE, PA-MPJPE, ACC-ERR) = (76, 53.2, 15.3)] | Modelo de cuerpo 3D. Basado en VAE (*encoder-decoder*). Modelo de 23 *joints*, parametrizados en un modelo SMPL. Persigue un compromiso entre precisión y naturalidad. No hay información de hardware. Git: 29/may/21. Video. |

**MotioNet** [28] intenta reconstruir un esqueleto 3D de un video monocular centrándose en la coherencia: se centra en el problema de inferir en base a los datos, una rotación de los *joints* concreta que se condiga correctamente con lo que el video muestre. Pese a que su precisión no se destaca en comparación a algoritmos en el estado del arte, garantiza una media de sobreestimación máxima del tamaño de los huesos del 10 %,

muy bajo en comparación a los demás algoritmos presentados (no hay experiencia de métricas comparadas en el tamaño de los huesos con otro algoritmos de entre los presentados). Es una opción viable, pero no es la que presenta la mejor métrica.

**XNect** [28] es un algoritmo en tiempo real que busca realizar *Motion Capture* 3D Multipersona. Funcionará a 30 fps, responderá bien a oclusiones tanto de objetos como entre personas. Su metodología se basa en 3 etapas; la *StageI* usa una CNN para procesar un input *frame* de manera completa, buscando todos los sujetos presentes en la escena. En esta etapa se predicen *joints* 2D traducidos en un *heatmap* PAF (Part affinity Field) para asociar *joints* individuales en la escena, para luego aplicar un 3D Pose encoding por *joint* Detectado. En la *StageII*, tendremos una FCNN que realiza un decode de la salida de la *StageI* para entregar una 3D pose completa, donde se incluye la posición relativa de cada *joint* visible y también de los *joints* ocluidos, de manera individual. La *StageIII* es un ajuste secuencial de la salida anterior (obtener resultados temporalmente coherentes). Su enfoque multipersona no es el que buscamos directamente. Sus métricas son 8 % inferiores a FrankMocap, lo que es natural, ya que no está enfocado en la captura de una sola persona.

**MEVA** [24](Z.Luo. 2020) es un algoritmo de estimación de pose 3D que extrae secuencias cinemáticas aproximadas de una persona en un video, para luego refinar la secuencia produciendo una estimación 3D precisa. Descompone el proceso en dos etapas: la primera estimando una pose inicial a través de un VAE (que comprime-sintetiza la información y funciona como un agrupador de poses latentes). Luego, se refina la pose utilizando un decoder (*pose regressor*). Su salida es un modelo SMPL: el paper asume que la transformación de SMPL a 3D *joints* es sencilla, siempre que se necesite. Con ello, este modelo entrega los 23 *joints* que un modelo SMPL estándar utiliza. Sus métricas están en el estado del arte, pero no entregan información de hardware ni relacionadas con los cómputos.

**FrankMocap** [27] captura las manos de las personas buscando contexto al detectar también el MoCap (*Motion Capture*). Con ello, logra mejores capturas de que otros algoritmos que únicamente se enfocan en las manos, además de entregar un modelo

SMPL-X completo, tanto para el cuerpo como para las manos, con todos sus *joints* asociados a este modelo. Toma prestado el algoritmo de *Body Tracking* de SPIN [16] con modificaciones[17], que se traducen en una implementación propia con buenos rendimientos que ya está integrada en FrankMocap. Las métricas de SPIN se anexan a la tabla correspondiente. Su Git está activo y tiene activas respuestas en torno a su implementación, sus salidas y su desempeño, lo que da garantías de estabilidad, y soluciones en el caso de encontrar problemas. Su perspectiva es una solución completa e integrada en todo el framework, y no se requieren implementaciones mayores ni adecuaciones exageradas para lograr resultados viables (e.g. requerir *joints* 2D de un algoritmo externo). Los últimos dos factores lo convierten en una opción muy atractiva para lograr implementación.

**EvoSkeleton** [17] (Li 2020 CVPR) es un framework que estima la pose 3D aprovechándose fuertemente de una técnica denominada *data augmentation*, especialmente útil para disminuir el sesgo en los pocos *datasets* 3D existentes para entrenar algoritmos de *3D human pose estimation*. A través de una perspectiva de algoritmo evolutivo para la etapa de conversión 2D a 3D, explora muchas posibilidades de poses extravagantes que permiten extrapolar posiciones nunca vistas en la fase de entrenamiento de la red. Se usó en Human3.6m y MPI-INF-3DHP con resultados positivos, pero no destacan (6-9 % inferior a FrankMocap). Su enfoque no es en tiempo real, ni con el objetivo de entregar resultados en un tiempo crítico. Su principal virtud es la capacidad de estimar poses muy extravagantes de manera plausible donde otros algoritmos no entregarían ninguna.

**Epipolar Trans.**[13] es un algoritmo que se aprovecha de geometría epipolar (usa dos cámaras sincronizadas en tiempo y en espacio) para lograr estimaciones mucho más certeras que enfoques monoculares (lo que es completamente comprensible), Es usado como comparador referencial para visualizar la máxima precisión que el estado del arte podría lograr en un contexto de implementación libre. Vemos que en general este algoritmo se comporta el doble de mejor que con enfoques monoculares (con la

[17] https://github.com/nkolot/SPIN

*Tab. 2.2:* Comparativa II de los algoritmos explorados en BwHT. Una cifra acompañada de L indica que la métrica es mejor mientras menor sea el valor, mientras que una H indica que a mayor valor la métrica es mejor.

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| FrankMocap [27] | [Manos: (STB, RHD, MPII+NZSL) con AUC-2D/3D = (0.992H, 0.934H, 0.655H)], [Cuerpo: (MPI-INF-3DHP con (PCK, AUC, MPJPE) = (92.5H, 55.6H, 67.5L) ], [Cuerpo: 3DPW con (MPJPE, PA-MPJPE) = (96.9, 116.4) ] | Estimador de pose y de manos para una persona. Enfoque en capturar manos, buscando max integración con el cuerpo. *Joints* modelo: Manos: 20 art. Cuerpo: 21 art. Entrega modelo SMPL-X. Enfoque *encoder-decoder*. Hardware: 9.5 fps con RTX 2080. Git: 27/may/21. Video |
| EvoSkeleton [17] | [Human3.6m con MPJPE = 50.9], [MPI-INF-3DHP con (PCK, AUC, MPJPE) = (81.2H, 46.1H, 99.7L)], disponible desempeñó por actividad | Estimador de Pose 3D unipersonal que se aprovecha fuertemente de data augmentation para la extrapolación 2D-3D. Enfocado en mejorar detecciones de posiciones extravagantes. Usa 17 *joints* para el cuerpo. Git: 10/jul/21. Video. |
| Minimal-hand [41] | [(DO, ED, STB, RHD) con AUC-3D/2D = 0.948, 0.811, 0.898, 0.856] | Detector de manos. Superligero. Hardware: Nvidia 1080Ti. 100fps. Modelo de 21 *joints* y mesh MANO. Git: 24/mar/21. Página. Video. |
| Anatomy3D [4] | [MPI-INF-3DHP con (PCK, AUC, MPJPE) = (87.9, 54.0, 78.8)], [Human3.6m con MPJPE = 46.3] | Estimador de pose 3D. Perspectiva de predicción de dos tareas centrales: dirección y longitud de ”huesos”de manera jerárquica. Basado en VideoPose3D. Git: 19/mar/20 |
| OpenPose [3] | NA (Resultados de comparativas tipo COCO: mPA, AP, AR, etc). | Estimador de Pose 2D Multi-Persona. Contiene Modulo estimador de pose 3D con dos cámaras. Basado en PAFs. Modelo de 135: cuerpo, pies, manos y expresiones faciales. El framework mejor soportado y más usado por la comunidad. Git: 18/ab/21. |

métrica Human3.6m).

**Minimal-hand** [41] es un algoritmo ligero para el *Hand Tracking* 3D. Ligero,

robusto y en el estado del arte, forma una opción de bajo impacto en términos de cómputo que puede complementar a cualquier algoritmo de *Hand Tracking*. Se garantiza su funcionamiento en tiempo real, con un modelo de manos de 21 puntos, lo cual es de sobra para nuestra perspectiva de precisión.

**Anatomy3D** [4] (Chen 2021 AnatomyAware) es un framework estimador de pose 3D con un enfoque de estimación de huesos, con sus respectivos tamaños y orientaciones. Basa la etapa de transformación de *joints* 2D a 3D en el framework VideoPose3D [26]. Su perspectiva de funcionamiento, es utilizar ”huesos” y definir dos etapas de definición de características, donde se determinará la orientación y el largo, aprovechándose además de una perspectiva jerárquica que permite extrapolar cuando algún *joint* está ocluido.

**OpenPose** [3] es el framework de estimación de pose 2D y reconocimiento de personas más popular de entre los disponibles. Se basa en PAFs (Part Affinity Fields), un campo de vectores 2D que codifican orientación y localización de extremidades. El apartado 3D de estimación de pose, que actualmente esta en una fase beta, considera el tener dos cámaras disponibles, un requerimiento que se escapa a las limitaciones que se han impuesto en esta memoria (monocular). Aún así, es importante mencionarle por ser un referente en los algoritmos de dos pasos que se presentarán (Generar *joints* 2D es un paso inicial a través de este algoritmo).

**MonocularTotalCapture** [35] (Xiang 2018) es un framework de *Motion Capture* 3D que captura las dinámicas esenciales de expresión de una persona a través de una cámara monocular. Captura manos, expresiones faciales, cuerpo con pies y manos a través de una representación 2D que incluye una caracterización de dirección y sentido. La caracterización se logra con una técnica llamada POFs (Part Orientation Fields) que codifica de manera eficiente la orientación de cada parte del cuerpo en el espacio 2D. Consta de 3 etapas: la obtención del POF a través de una red FCNN, una segunda que crea una malla deformable adaptada del cuerpo, y una tercera que fuerza la consistencia temporal para evitar *jitter*. Utiliza el modelo de Adam para capturar todas las características del cuerpo humano (forma y pose), pero se garantiza poder

entregar otros modelos de salida no tan ricos, como SMPL. En su git garantizan la factibilidad de entrega del archivo **.bvh**[18] que captura la dinámica.

**3D-Multi-Person-Pose** [6] (Cheng 2021 CVPR) es un framework de estimación de pose 3D multipersona que usa un enfoque combinado *top-down*. La estimación de la pose humana 3D se realiza a través de 3 pasos: una red descendente para la estimación de la pose, una red ascendente para la estimación de la pose global, y una red de integración para integrar las estimaciones de las ramas descendente y ascendente con el discriminador de la pose entre personas. Además, se propone un proceso de entrenamiento semi-supervisado para mejorar la estimación de la pose 3D basándose en la consistencia de la reproyección. No presenta mejores métricas que algoritmos de estimación de pose unipersonales.

**Attention3DHumanPose** [21] (Liu 2020) presenta un enfoque *attention model* para la estimación de la postura 3D a partir de vídeos en 2D. Combina la dilatación multiescala con el módulo de *temporal attention*. Con esto, logran capturar las relaciones temporales de largo alcance a través de los *frames*, mejorando así significativamente la coherencia temporal. Se garantiza en su paper su funcionamiento en tiempo real. Su perspectiva, en general, se trata de predecir *joints* 2D para a partir de ellos recuperar los *joints* 3D usando su paradigma particular de modelos de atención.

**EasyMocap** [9] (Dong 2020) proponen capturar el movimiento humano, realizando una reconstrucción y entrega de un MoCap (*Motion Capture*) 3D, analizando vídeos de Internet (grabados monocularmente) que tienen muchas tomas de una misma persona en lugar de utilizar vídeos individuales. Esta nueva tarea plantea retos, como que los vídeos no están sincronizados, los puntos de vista de la cámara son desconocidos, las escenas de fondo son diferentes, y los movimientos humanos no son exactamente los mismos entre los vídeos. Además se aprovecha muy inteligentemente de espejos [11] y de numerosas cámaras con muchas personas (multipose) [8]. No cumple con la restricción de cámara monocular, pero muestra una perspectiva

[18] La extensión de archivo **.bvh** cumple una función parecida a la de **.fbx**, pero no captura los parámetros de forma, solo los de pose. Se estudiará su compatibilidad con UE en caso de ser necesario.

interesante, que además contempla transformaciones de su salida a formatos **.fbx**.

*Tab. 2.3:* Comparativa III de los algoritmos explorados en BwHT. Una cifra acompañada de L indica que la métrica es mejor mientras menor sea el valor, mientras que una H indica que a mayor valor la métrica es mejor.

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| EasyMocap [9] | No aplica | Algoritmo de Mocap 3D uni/multi-persona. Con videos de internet de la misma escena se estima de manera certera la pose (sin calibración). Se aprovecha de espejos, tomas de diferentes escenas, tomas de muchas escenas monoculares, etc. Git: 17/jul/21. |
| ThreeDPose UnityBarracuda [Source] | No tiene | *Full Body Motion Capture* con *3D pose*. Implementado para Unity. Un framework comercial *free-source* sin paper. Git: 19/ene/21. |
| Monocular Total Capture [35] | [Human3.6m con MPJPE = 58.3] | Estimador de pose 3D, de manos y cuerpo (gestos) usando POFs. POFs: codificar orientación 3D del cuerpo en espacio 2D. Utiliza el modelo de Adam. Utiliza GTX 1080Ti GPU, pero no indica tiempo de ejecución. Git: 26/abr/20. Video |
| 3D-Multi-Person-Pose [6] | [3DPW con PA-MPJPE = 75.6], [Human3.6m con (MPJPE, PA-MPJPE) = (40.7, 30.4)], [MuPoTS-3D con PCK = 88.9] | Estimador de pose 3D multipersona. Integra la perspectiva *Top-down* (Descendente para recopilar detalles en las personas detectadas) y la *Bottom-up* (ascendente, para recuperar características globales de pose, contexto). Sin información de Hardware. Git: 22/jun/21. |
| Attention3D HumanPose [21] | [Human3.6m con MPJPE = 45.1] | Estimador de pose 3D unipersonal. Probado en Nvidia Titan RTx, a 3000fps. Utiliza *attention models* para la estimación de la pose 3D, a través de los *joints* 2D predichos. Git: 20/nov/20. Video. Página. |

**ThreeDPoseUnityBarracuda** [Sitio] es un framework de predicción de pose 3D

implementado de manera comercial, pero comparte su código en un Git (no con las últimas mejoras). Está específicamente hecho para la herramienta Unity (motor de juego muy popular). Por ser implementado directamente para una plataforma, no esta implementado para Unreal Engine, lo que dificulta su uso. No presenta métricas, pero su evaluación en linea por usuarios en la web le otorgan una precisión suficiente para ejecutar sobre el un avatar, realizando acciones comunes.

**CanonPose** [32] es un enfoque auto-supervisado que aprende un estimador de pose 3D de una sola imagen a partir de datos con múltiples perspectivas no etiquetadas. Para ello, explotan las restricciones de consistencia de múltiples perspectivas para desentrañar la pose 2D observada en la pose 3D subyacente y la rotación de la cámara. A diferencia de la mayoría de los métodos existentes, no se requiere cámaras calibradas, y por tanto, se puede realizar el MoCap con cámaras en movimiento. Sin embargo, en el caso de una configuración de cámara estática, se presenta una extensión opcional para incluir rotaciones de cámara relativas constantes buscando obtener más precisión con respeto a otros algoritmos de cámaras fijas.

**ITES (Invariant Teacher and Equivariant Student)** [36](Xu 2021) es un algoritmo de estimación de pose 3D, que espera como entrada *keypoints* 2D para realizar su estimación. Dado que estimar la pose 3D de puntos 2D es un problema *ill-posed* (infinitos esqueletos 3D corresponden a los mismos *joints* 2D), se requiere una perspectiva *teacher-student* para manejar la regularización y la flexibilidad del modelo para delimitar las opciones que realmente sean plausibles. La pose 3D estimada se rota y se reproyecta para obtener una pose 2D, que se fuerza a intentar parecerse lo más posible a los puntos 2D originales. Esto garantiza limitar las posibilidades y dar buenas métricas. Su principal problema, es que se requieren los *joints* 2D de pose estimados desde algún software externo, e ITES no provee el cómo obtener esos *joints* de manera explicita (utilizan directamente un *dataset*, sin inferir en un video propio). Con ello, se debe implementar un estimador de pose 2D, para pasarle esta salida, como entrada a ITES. En su respectivo paper, se sugiere (no se implementa) utilizar para esta etapa a [5](Chen 2018), para luego estimar la pose 3D y medir la disparidad de

*Tab. 2.4:* Comparativa IV de los algoritmos explorados en BwHT. Una cifra acompañada de L indica que la métrica es mejor mientras menor sea el valor, mientras que una H indica que a mayor valor la métrica es mejor.

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| CanonPose [32] | [Human3.6 con (MPJPE, PMPJPE) = (74.3L, 53.0L)], [3DHP con (MPJPE, PMPJPE, PCK) = (104.0L, 70.3L, 77.0H)] | Algoritmo de estimación de pose 3D. Requiere 2 cámaras monoculares en diferentes perspectivas (pueden no estar calibradas). Muy buena precisión sobre Human3.6M e INF-3DHP cuando existe movimiento. Github: actualizado el 8/jul/21. Video. |
| ITES [36] | (*)[Human3.6m con (MPJPE, P-MPJPE) = (85.3, 59.8)] | Estimador de pose humana 3D. Al entregarle una pose 2D, utiliza un método *teacher-student learning* para obtener el 3D más plausible. Sin información hardware. Mantendrá un diccionario de poses, rotará la perspectiva capturada hasta hacerla coincidir con el input. Git: 6/jul/21. |

predicción con Human3.6m. Notar que sus métricas utilizan los *joints ground truth 2D* de los *datasets*, por lo que los resultados están sesgados[19].

### *2.2.3. Soluciones disponibles para User Removal (URem)*

En URem la variabilidad de algoritmos se reduce: el limite esencial que buscamos es una segmentación de la persona que no requiera una máscara de entrada.

#### *Opciones URem*

**MODNet** [15] es un algoritmo de *Portrait Matting* muy popular en la web, con muchos algoritmos divergentes en la comunidad. Es ligero, preciso y no requiere un

[19] Todos los algoritmos utilizan una etapa de inferencia para realizar la estimación 2D, para luego evaluar su calidad de inferencia 3D. Utilizar un *dataset* en vez de inferir la estimación 2D muestra un resultado parcial, ya que esa metodología no incluye el error propio de la inferencia 2D.

*Tab. 2.5:* Comparativa de los algoritmos explorados en URem. Todos los *datasets* de evaluación no son comunes.

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| MODNet [15] | [PPM-100 con (MSE, MAD) = (0.0046, 0.0097)] | Algo. de *image matting*. Tiene muchas ramificaciones de aplicación de la comunidad (Como esta). Tres etapas: *data flow* (estimación rápida), consistencia, promediar píxeles, compromiso entre las dos anteriores. 63 fps con GTx 1080Ti a 512x512. Git: 20/jul/21. Video. |
| BackgroundMatting V2 [18] | [AIM con (SAD, MSE) = (12.86, 12.01)], [PhotoMatte85 con (SAD, MSE) = (8.65, 9.57) ] | Algoritmo de *image matting*. Exige sacar una foto del fondo sin objeto de interés. Muy preciso. HD 60fps o 4k 30fps con Nvidia RTx 2080Ti. Git: 23/jun/21. Página. |
| TCVOM [39] | Sobre *dataset* personalizado (VideoMatting108) | Separacion de capas. Basado en CNN y *Attention models*. Requiere un *trimap* de entrada. Proveen el módulo generador de *trimap* de manera externa. Para objetos, no especializado en personas. Git: 5/jun/21. Video. |
| Omnimatte [23] | No tiene | Algo. de *image matting* que rescata las sombras en la segmentación, además entrega fondo repintado como segundo output. Requiere de entrada máscara sobre objeto interés Sin información de rendimiento en *datasets* ni hardware necesario. Git: 25/ago/21. Video. |

*treemap*[20] de entrada para realizar la segmentación de la persona. Su git e implementaciones están actualizados, por lo que resulta una atractiva solución que no compromete cómputo ni precisión. Divide el problema en 3 tareas específicas: la primera, una estimación de baja resolución de la segmentación de la persona. Luego, realiza una segunda estimación más agresiva basada en la anterior para finalmente, en la tercera, se realiza un suavizado de pixeles que compromete las dos anteriores predicciones,

[20] Un *mapa* que representa jerárquicamente alguna característica. Codificación en una imagen de algún tipo de información. En este contexto se refiere a algún tipo de información que explicite que zona debe extraerse.

entregando el *alpha matte* (la imagen donde solo está el usuario segmentado, con el resto de la imagen sin información) predicho.

**Background Matting V2** [18] es un algoritmo de *image matting* de muy alta resolución y de operación en tiempo real. Se basa en un primer módulo que utiliza una arquitectura *enconder-decoder* como base para realizar la segmentación, para luego utilizar un segundo módulo que opera sobre los parches donde se detecta un error de segmentación grande para repararlos, y un último módulo realiza un compromiso entre las dos etapas anteriores (ponderación). Como entrada se espera que se entregue la imagen a segmentar, y además el *background* sin el usuario a segmentar. Esto es una limitante muy importante, ya que presuponer que se dispone del *background* sin el objeto de interés limita la aplicabilidad en ciertos contextos donde, por ejemplo, se requiera un proceso automatizado de *matting* con solo una foto.

**TCVOM** [39] es un algoritmo de *video matting* que realiza una segmentación de un objeto dado configurado de antemano por la red, por lo que no está específicamente hecho para segmentar personas. Provee una red que al entregarle un *treemap* específico, realiza la segmentación del objeto de interés allí indicado. Es una opción viable, pero no específicamente hecha para la segmentación de personas. Sus métricas son sobre un *dataset* personalizado, con ello, la comparación es poco útil.

**Omnimatte**[23] es un algoritmo de separación por capas de objetos de interés y del fondo. Esto implica, obtener como salida el fondo repintado, y capas con los objetos de interés, cumpliendo funciones tanto de *user removal* como de *video inpainting*. Como entrada requiere indicar a través de una máscara de segmentación que objeto debe ser extraído del video original. Esto implica usar un algoritmo rápido de segmentación de personas que entregue máscaras, como Mask R-CNN, para entregar donde se encuentra la persona en cuestión. No presenta resultados cuantitativos en relación a la calidad de la segmentación final junto con la sombra: solo presenta resultados cualitativos en imágenes resultantes. Puede ser un algoritmo a considerar ser utilizado en la etapa de *user removal*, pero sin las métricas no tendremos la capacidad de comparar a priori.

Otros análisis importantes conforman a **SIM** [29] y también a **DVM** [30], ya que presentan métricas superiores. No se estudian dado que sus algoritmos no han sido liberados. Es importante remarcar que parte importante de la selección de un algoritmo idóneo es que esté disponible su código fuente.

### *2.2.4. Soluciones disponibles para Video Inpainting (VIn)*

Para *video inpainting* la variabilidad de algoritmos comienza a aumentar, dado que las técnicas predominantes divergen en una medida considerable, aunque no al nivel de complejidad de la estimación de pose. En general, existe un fuerte aprovechamiento de los algoritmos más exitosos en su lanzamiento (y que se han mantenido en general como los más exitosos): STTN y *Deep Flow Guided Video Inpainting*.

#### *Opciones*

**STTN** [38] es un algoritmo de repintado plausible que a través de *attention models* busca un repintado coherente alrededor de todos los *frames* de entrada. Así se logra obtener un repintado con contexto en la zona puntualizada a través de una máscara de segmentación. Utiliza un *encoder* que codifica los *frames*, un trasformador espacio temporal que rellena los huecos con contenidos coherentes, y un *decoder* de los *frames*.

**Deep Flow Guided Video Inpainting** [37](Xu 2019 CVPR) es un algoritmo de *video inpainting* basado fuertemente en el uso de *optical flow* para detectar movimiento, y allí operar más inteligentemente el repintado de la imagen, esto es, busca contexto directamente basado en movimientos rápidos. Dado que no reportan hardware usado ni tampoco métricas relacionadas con el tiempo, se sospecha que se aleja mucho de una ejecución en tiempo real.

**TSAM** [42](Zou 2020) es un algoritmo de repintado que se aprovecha de varias de las características de las técnicas usadas hasta ahora, que son 3: modelos completamente convolucionales, *optical flow* y modelos de atención.

*Tab. 2.6:* Comparativa de los algoritmos explorados en VIn. Aquí, FWE o $E_{warp}$ corresponde a *Flow Warping Error*. Notemos que la métrica SSIM (Structural SIMilarity (SSIM) a veces es entregado en % o como porcentaje en decimales. Se sobreentiende que la comparativa debe realizarse sobre decimales (0.9876) o sobre porcentajes (98.76).

| **Algoritmo** | **Datasets de Evaluación** | **Características** |
|---|---|---|
| STTN [38] | [Youtube-VOS con (PSNR, SSIM, FWE, VFID) = (32.34H, 96.55H, 0.1451L, 0.053L)], [DAVIS con (PSNR, SSIM, FWE, VFID) = (30.67H, 95.60H, 0.1779L, 0.149L) ] | Algoritmo de repintado plausible. Basado en perspectiva *encoder-decoder* con correspondencia de contexto. 24,3fps con Nvidia V100, o 10,43fps con Nvidia P100. Requiere máscara de entrada para encasillar el repintado. Git: 1/may/21. Video. |
| Deep-Flow-Guided Video-Inpainting [37] | [Youtube-VOS con (PSNR, SSIM) = (27.49, 0.41)], [DAVIS con (PSNR, SSIM) = (28.26, 0.48)] | Algoritmo de 3 pasos: estimación de *flow field*, propagación de secciones faltantes con el field calculado, y finalmente una red clásica de *inpainting*. Requiere entregarle una máscara de eliminación. Git: 10/dic/19. Sin información de hardware. Video. |
| TSAM [42] | [FVI con (PSNR, SSIM, VFID) = (35.48, 0.9160, 0.6129)], [DAVIS con (PSNR, SSIM, VFID) = (34.23, 0.8798, 0.3526)] | Algo. *video inpaiting* basado en *encoder-decoder* y un módulo complementario *flow-based*. Mejora sobre FGVC y STTN, los 2 *aproach* del estado del arte. No provee información clara de hardware, fps ni resolución. Git: 14/jun/21. |
| DSTT [20] | [Youtube-VOS con (PSNR, SSIM, VFID, $E_{warp}$) = (32.66H, 0.9655H, 0.052L, 0.1430L)], [DAVIS con (PSNR, SSIM, VFID, $E_{warp}$) = (31.75, 0.9650, 0.148, 0.1716)] | Algoritmo de *video inpaiting* basado en CNN *encoder-decoder* con un análisis espacio temporal separado en dos tareas. Mejora resultados de STTN en [FLOPs] fps (37.3 fps) y precisión, usando Nvidia 1080Ti. Git: 22/abr/21. |
| Omnimatte [23] | No tiene. | Alg. de *image matting* que rescata las sombras en la segmentación, además entrega fondo repintado como segundo output. Requiere de entrada máscara sobre objeto interés Sin información. Rendimiento en *datasets* ni hardware necesario. Git: 25/ago/21. Video. |

Proponen entonces una Red de Alineación de Características Temporales Progresivas"para combinar las ventajas y compensar las debilidades de los marcos de resolución temporal y los enfoques de deformación basados en el *optical flow*, utilizando un módulo de desplazamiento y alineación temporal (bautizado TSAM), en el que las características entre fotogramas vecinos se alinean (utilizando el *optical flow* aprovechándose del movimiento). Como nota, FVI es un subconjunto del *dataset* Youtube-VOS, por lo que sus métricas pueden ser comparadas, aunque con suspicacia. La falta de información clara de hardware y fps no nos dejan en claro su rendimiento en términos de tiempo.

**DSTT** [20] es un algoritmo de repintado fuertemente basado en STTN[38]. Tiene una estructura *encoder-transformer-decoder*, donde la tarea luego del decoder es desacoplar la tarea de realizar una extrapolación temporal plausible, y una tarea de extrapolación espacial (STTN usa solo una red para estas dos tareas). Con ello, el rendimiento final comparado directamente con STTN aumenta, además de generar inferencia más rápida. Con ello, su respuesta es en tiempo real: 30-40 fps de repintado con una tarjeta Nvidia 1080Ti. Sus métricas son superiores a STTN.

**Omnimatte**[23] es un algoritmo de separación por capas de objetos de interés y del fondo. Esto implica, obtener como salida el fondo repintado, y capas con los objetos de interés, cumpliendo funciones tanto de *user removal* como de *video inpainting*. La segmentación de objeto, además incluye la sobra del objeto seleccionado. Como entrada requiere indicar a través de una máscara de segmentación que objeto debe ser extraído del video original. Esto implica usar un algoritmo rápido de segmentación de personas que entregue máscaras, como Mask R-CNN, para entregar donde se encuentra la persona en cuestión. No presenta resultados cuantitativos en relación al repintado del fondo al extraer el objeto en cuestión de ese fondo. Solo presenta algunos resultados cualitativos sobre imágenes elegidas, comparados con algoritmos populares.

## 2.3. Resumen del EoA: Predominancia de un enfoque extensivo en datos complementado con una técnica extensiva en cómputo

El enfoque *deep* está presente en todos los algoritmos, siempre complementado con alguna técnica que ofrezca una respuesta más fiable o que ayude a generalizar más allá de lo que puede lograr este enfoque. Las tareas de reconocimiento de posición del *body tracking*, el problema más complejo y abierto de los 3, tiene resultados que ya son aceptables y pueden ser implementados en tiempo real a 5-10 fps [2] [25] en la teoría.

El contexto del problema exige la capacidad de lograr resultados coherentes en contextos dispares, ya que desde el lado del desarrollador no existe control del contexto de grabación de los videos (más allá de una sugerencia de uso): pueden ser grabados en espacios abiertos o cerrados, de noche o de día, e interactuando con distintos tipos de objetos. Es esta exigencia de mayor generalidad la que se traduce en la complementariedad de enfoques presentes en muchos algoritmos. Se debe tener presente para los enfoques *deep* los *datasets* de entrenamiento para los modelos, ya que el sesgo dado por la naturaleza de los datos (solo en interiores, por ejemplo) es un factor a considerar al elegir la opción correcta: limitarán la capacidad de respuesta en situaciones en las que no fueron entrenados. Se comentará en términos generales las soluciones posibles, en nuestro contexto, para cada uno de los 3 problemas presentados.

En general, los algoritmos de repintado de última generación (esto es, 2020 y 2021) tienden a compararse entre sí y declaran mejoras sustanciales sobre algoritmos anteriores a su propia fecha de publicación. En general integran las soluciones antiguas que aportan una perspectiva útil y con buenas métricas, y las combinan. Las 2 principales referencias son **FGVC** [12] y **STTN**[38], que ya muestran buenos resultados teóricos.

En general, se denota una tendencia a usar *optical flow* con una arquitectura *encoder-decoder*, junto a mejoras particulares. El algoritmo con mejor desempeño es **DSTT** [20], que declara en su paper una mejora sustancial en comparación al algoritmo en el

estado del arte del repintado, **STTN**. Con ello, es natural que sea una de las opciones más prometedoras a seleccionar.

La discusión trascendental para usar estos algoritmos se traduce en la técnica para obtener la máscara de repintado que requieren estos algoritmos para señalar dónde deber operar. Se espera lograr esta segmentación a través del algoritmo de URem, seleccionado en el paso anterior, que será más preciso que simplemente utilizar algún framework de segmentación multipropósito basado en Mask R-CNN, como Detectron2. [34].

Las implementaciones especializadas otorgan prestaciones en tiempo real y más control en la segmentación cuando son enfocadas en objetos particulares. Un ejemplo clave de esto es la segmentación del pelo, que en frameworks especializados en segmentación de personas presentan un especial interés, por ser el problema más importante en segmentación por su longitud y errática distribución. **ModNet** provee una solución en tiempo real, con buenos resultados cualitativos y fuertemente respaldada por la comunidad. Su git está muy activo, y se actualiza muy seguidamente. Se presenta como una opción muy cómoda para nuestro contexto que brindará todas las características que buscamos.

Es claro que los *datasets* usados para las métricas en los distintos algoritmos de URem son disjuntos, lo que dificulta fuertemente una comparativa cuantificable, más allá de evaluar los resultados de repintado de manera empírica. Se evaluará la generación de un *ground truth* construido manualmente.

Considerando ahora el problema de BwHT, recomendar un algoritmo implica elegir un parámetro común de comparación. Las tablas de rendimientos sobre los *datasets* mostradas en el punto anterior ayudan a establecer parámetros cuantitativos de comparación.

Utilizando las métricas basadas en Human3.6m (el *dataset* más popular entre todos), algoritmos como **Anatomy3D** y **Attention3DHumanPose** parecen los más atractivos (ambos basados en **VideoPose3D**, lo que significa que son redes que requieren una entrada de *joints* 2D para realizar la estimación de pose 3D). Desde la

perspectiva cualitativa, un Git estable, comentado y activo garantiza resultados y solución de problemas imprevistos: allí VideoPose3D se destaca. Aún cuando los 3 algoritmos son del 2020, sus rendimientos comparados con los presentados durante el 2021 (como **PoseFormer** [40]) no tienen mejores métricas en este *dataset*, por lo que no se justifica tener un mayor riesgo de implementación al estar menos documentados y testeados por la comunidad (mayor probabilidad de complicaciones) que podrían llegar a generar retrasos o complicaciones.

Por otro lado, utilizando como métrica los algoritmos que usan MPI-INF-3DHP, SPIN (algoritmo del cuerpo de **FrankMocap**) o **MEVA** tienen métricas superiores en comparación a Anatomy3D (y probablemente que Attention3DHumanPose, por su arquitectura similar).

Los principales modelos que extrapolan poses 3D utilizan VideoPose3D, que requerirá siempre una entrada de *joints* 2D. Esta puede obtenerse con Detectron2 con el *dataset* COCO, con una implementación no tan compleja[21]. En el caso de Anatomy3D, los ejemplos presentados utilizan AlphaPose[10], un estimador de pose multipersona 2D. En general, la mayoría de los algoritmos con un enfoque 2D a 3D requerirán una etapa previa de inferencia 2D: aunque su implementación no es un problema, tomarán tiempo y requieren sus propias dependencias y recursos para su ejecución.

Con ello, dado que registramos un empate en estos algoritmos, **FrankMocap** se destaca por sobre los otros por su integración del algoritmo de manos, además de no necesitar una implementación mayor para lograr la estimación de pose 3D final (no requiere entrada de *joints* 2D, solo el video). Esta sería la opción más viable considerando tiempo, cánones de implementación, y un muy ingente *pool* de preguntas-respuestas en su git. Para evaluar la calidad de este algoritmo se hará necesario realizar una comparativa empírica con MEVA, VideoPose3D, Attention3DHumanPose, y se evaluará la realización de más pruebas de algoritmos en la medida que se observen resultados deficientes entre todos ellos.

[21] Puede visitar la discusión en https://github.com/facebookresearch/VideoPose3D/issues/2

# 3. DISEÑO DE LA SOLUCIÓN Y EVALUACIÓN DE HERRAMIENTAS

Este capítulo está dedicado a dos tareas: en primer lugar, proponer una arquitectura de software con tareas bien definidas que permitan obtener los elementos necesarios para materializar la arquitectura de 3 capas, reconociendo requisitos de cada módulo y los pasos necesario para poder exportar sus salidas hacia el motor de videojuegos.

En segundo lugar, se evalúan las herramientas en el estado del arte. Los parámetros de interés son la precisión, el uso de recursos y la compatibilidad de su salida con el motor de videojuegos. Este análisis se realizará de manera particular para cada algoritmo de interés, en cada uno de los 3 tópicos estudiados. La selección final de un algoritmo para cada tópico se realiza basándose en estas evaluaciones.

## *3.1. Modelo general de la solución*

El modelo general de la solución es una arquitectura inducida por la necesidad de entregar el formato de 3 capas presente en UE (Unreal Engine). Podemos distinguir los 3 módulos de procesamiento de video para obtener la estimación de *joints*, el usuario segmentado y el fondo repintado.

La arquitectura de la solución propuesta se muestra, a grandes rasgos, en la Figura 3.1, donde se identifican entradas y salidas en modelos de bloque para cada sistema. Se explicitan los dos acondicionamientos más complicados y que se traducen en transformaciones que deben ser implementadas. En las próximas secciones se realizan pruebas de algoritmos de interés buscando un consumo de recursos acotado y precisión, en la medida de lo posible. Perfiladas las características y resultados de ca-

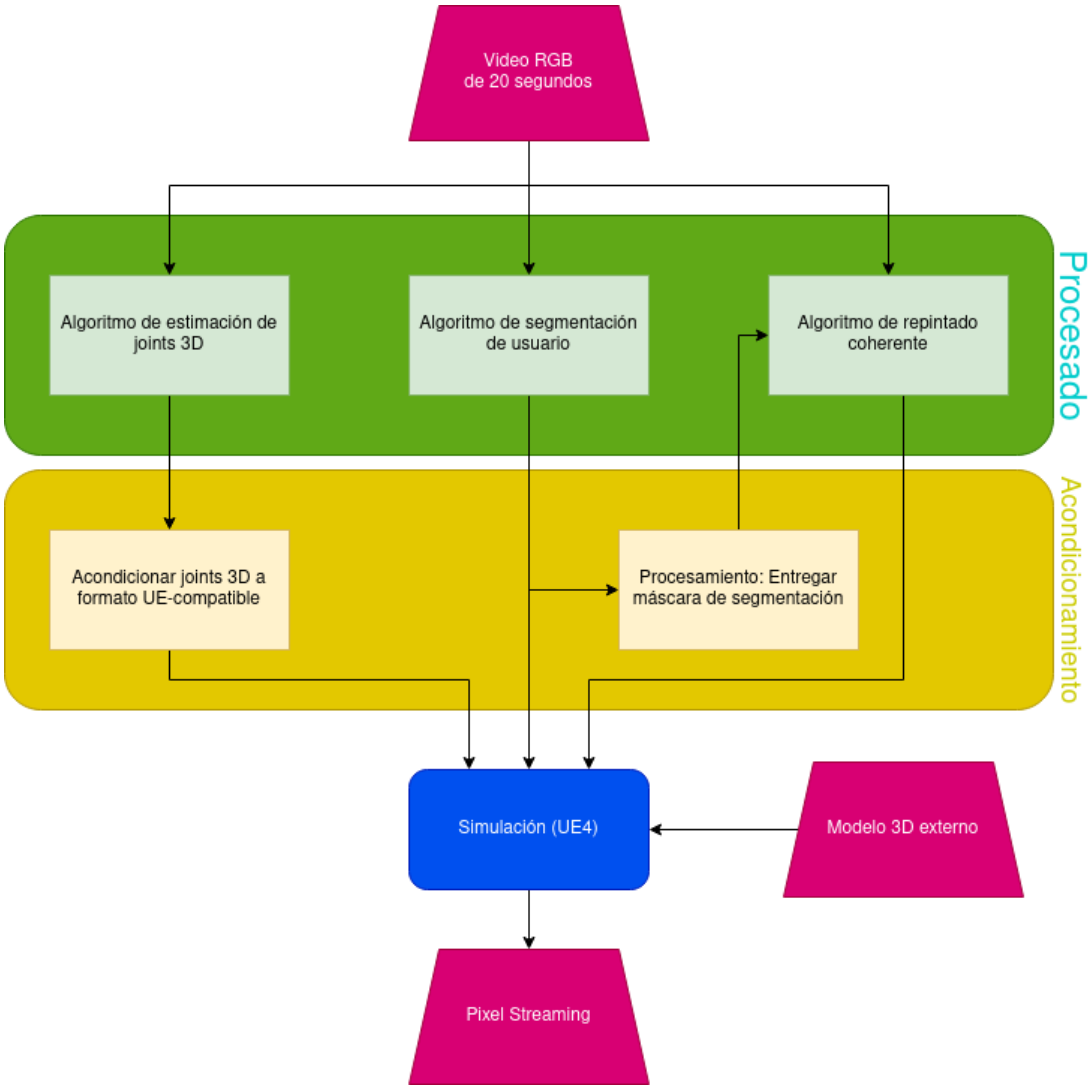



*Fig. 3.1:* Diagrama general de la solución.

da algoritmo, se decidirán los mejores para cada etapa de procesado y se realizará el acondicionamiento necesario para lograr integrar los respectivos resultados dentro del motor de videojuegos. Esto incluye transformaciones especialmente críticas para dos tareas específicas: por un lado, es necesario acondicionar la salida de cada algoritmo para ser interpretado por UE; en segundo lugar, es necesario crear las máscaras de segmentación a partir del algoritmo de segmentación de usuario, para explicitar las zonas donde el algoritmo de repintado coherente debe operar. Esto incluye filtros y thresholds.

Finalmente, con las transformaciones ya realizadas, y las salidas acondicionadas para ser leídas por UE, se realizará la integración automatizada con el motor de videojuegos, lo que incluye programación asociada al lenguaje que utiliza UE para realizar *scripting* con C++ o utilizar máquinas de estado con *blueprints*.

## *3.2. Pruebas preliminares: Etapas y Testing*

En la presente sección, se realiza una evaluación preliminar de las alternativas de solución mencionadas en el capítulo anterior, a través de pruebas de los distintos métodos en un ambiente local. Para esto, se utilizan implementaciones disponibles públicamente en un repositorio del autor del método, las que incluyen el código fuente y los pesos de la red neuronal ya entrenada.

Se mostrará una selección de algoritmos y su implementación en hardware local para evaluar su viabilidad como solución final. Esta arquitectura local deberá ser extendida a una implementación en la nube en un trabajo futuro. Con ello, se espera recopilar datos de interés en cada implementación que permitan una selección basada en rendimientos y en la precisión que cada algoritmo logra. Como se ve en el diagrama 3.1, debemos estudiar 3 tópicos esenciales, que serán presentados en las 3 subsecciones siguientes.

El ambiente de prueba local se ejecuta en el siguiente hardware/software:

- **CPU: AMD Ryzen 5900HX**. 8 x 3.3 - 4.6 GHz, 64.8 W PL2 / Short Burst, 34.4 W PL1 / Sustained, Cezanne H (Zen 3).
- **GPU: NVIDIA GeForce RTX 3070 Mobile**. 8192 MB, Núcleo: 1560 MHz, Memoría: 1750 MHz, 130 W TDP (including 15 W Dynamic Boost), GDDR6, Nvidia 461.40 DCH, Optimus.
- **RAM: 16[Gb]**. Dual-channel DDR4-3200.
- **Disco Nvme m.2 1[Tb]**; SK Hynix HFM001TD3JX013N, 1024 GB.

- **SO: Ubuntu 21.04**.

La implementación de cada algoritmo llegará hasta 2 estados: implementado con prueba sintética o implementación con prueba personalizada. La implementación con prueba sintética es lograr una prueba local con datos de prueba entregados por el autor. Una implementación con prueba personalizada es lograr procesar un video personal grabado con una cámara de teléfono, lo que implica realizar transformaciones que permitan al algoritmo la entrada a ese video. En general, las implementaciones que se prueban hasta realizar pruebas sintéticas y no avanzan a pruebas personalizadas se debe a que se encontraron dificultades en adaptar ese algoritmo a la solución de este contexto, en algún punto de las pruebas.

Para los algoritmos que pasen a una prueba personalizada, se registran parámetros de interés relacionados con el uso de recursos: CPU, RAM, disco, memoria GPU y tiempo. Estos son importantes en el contexto de uso de hardware, para determinar costos de implementación y del sistema que soportará la inferencia de los algoritmos (los costos de ejecutar nodos en nube con disponibilidad de GPU son altos). Estos registros se lograrán a través de sistemas de monitorización disponibles en Linux:

- *htop*: comando de la *shell* de Linux que registra dinámicamente el uso de RAM y CPU. Con los parámetros correctos se puede aislar el consumo específico de un programa determinado.

- *nvidia-smi*: comando que provee la API de Nvidia en sus controladores de Linux para registrar el uso de GPU, que registra todos los parámetros de interés desplegables por la terminal.

Estos datos se presentarán en las tablas respectivas por sección.

### *3.2.1. Pruebas de algoritmos de BwHT*

En este apartado se espera seleccionar un algoritmo local que pueda obtener los *joints* de la persona en el video, y lograr importar esas estimaciones al motor de videojuegos. Las etapas de procesamiento se describen en la Figura 3.2.

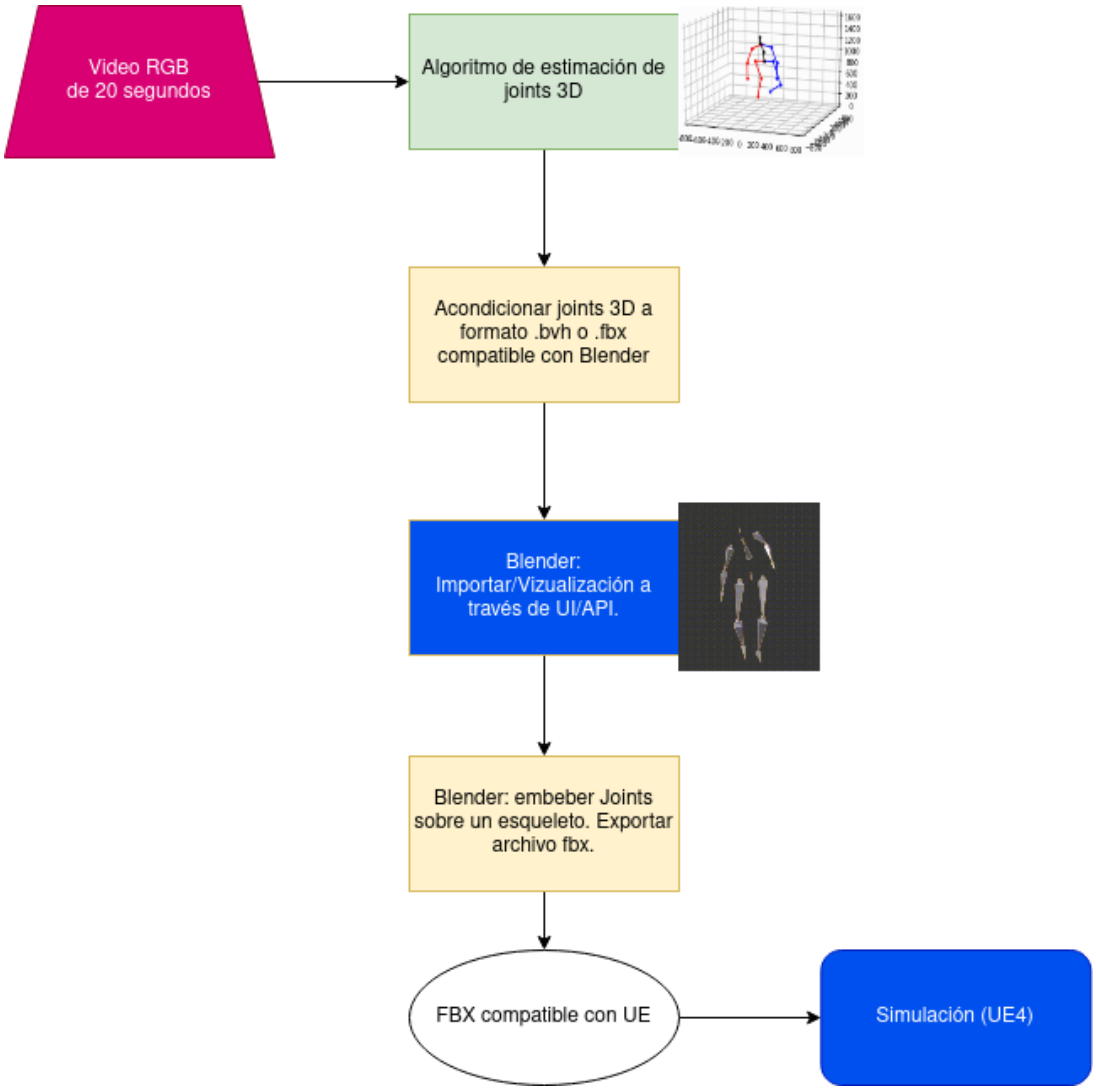



*Fig. 3.2:* Diagrama de la solución de BwHT.

El módulo de acondicionamiento de *joints* forma parte de un algoritmo necesario y dependiente del algoritmo de BwHT seleccionado, buscando obtener la mejor respuesta local de recuperación del movimiento del usuario. Este módulo de acondicionamiento debe realizar cambios al formato de salida del algoritmo de BwHT para que pueda ser interpretado por UE4. En general esto se traduce en alguna trasformación afín o una correcta selección de los *joints* entregados disponibles.

Una de las condiciones esenciales para la selección de uno de los siguientes algoritmos es que el movimiento pueda importarse a Unreal Engine 4. En general, esto se

traduce en que los *joints* tengan la relación geométrica que espera el software, además de un formato de archivo específico **.fbx** que tenga un esqueleto embebido en su configuración. Algunos algoritmos entregan como salida archivos en formato **bvh**[1], lo que requiere una transformación de bvh a fbx. Los algoritmos testeados se muestran en la Tabla 3.1.

La táctica de prueba inicial se basa en lograr importar los datos capturados a Blender, un programa dedicado especialmente al modelado, iluminación, renderizado, la animación y creación de gráficos tridimensionales. Existe una *Blender Python API* (Biblioteca **bpy**) que permite escribir *scripts* para procesar cambios de formato, integración de esqueletos, y otras funciones de post-procesado. Esto permitirá lograr la importación, lo que ayudará a visualizar el resultado de manera previa en Blender y funcionar como paso intermedio para la importación a Unreal Engine. Esta biblioteca no se puede instalar fácilmente a través de *pip*, el gestor de paquetes estándar de Python. Debe descargarse el código de Blender sin compilar, y hacer un *link* a un ambiente de Python creado previamente, de manera manual, a través de *cmake*. Con estos parámetros configurados, debe compilarse el programa Blender solo para ese ambiente. Para nuestro contexto es necesario el uso de un ambiente que permita la modularidad de bibliotecas dependiendo de las versiones de Python necesarias para cada algoritmo (e.g. Blender utiliza Python 3.7.8, mientras que FrankMocap utiliza 3.9.5). Puntualizar esta instalación es importante por el contexto de implementación: un *container* requerirá la instalación de **bpy** que no es posible si no es instalando en un ambiente.

Con las bibliotecas configuradas, se procede a probar los algoritmos de manera separada.

En primer lugar se prueba FrankMocap. Como entrega un modelo SMPL, se puede ver la malla correspondiente a la inferencia realizada por el algoritmo para la forma del cuerpo, superpuesta a la captura de movimiento (como puede verlo en la imagen 3.3). Sus resultados son buenos en general, pero se aprecia *jitter* de forma regular. Además,

[1] Este formato de salida solo recopila los parámetros de pose, y no los de forma.

*Tab. 3.1:* Resultado de pruebas locales para algoritmos BT. se uso [Th] para indicar *Thread*, uso por CPU.

| **Algoritmo** | **GPU** | **Time** | **CPU** | **RAM** | **Resultados** |
|---|---|---|---|---|---|
| FrankMocap | 3,02 [Gb] | 850 [s] | 15 [Th] al 5 %, 1 [Th] al 45 % | 4,30 [Gb] | **Implementación con prueba personalizada**. Sufre de *jitter*. Bien documentado. Solo logra 0.5 fps (a veces 1 fps) de procesamiento para el modelo de cuerpo y manos. |
| VideoPose3D | 2,35 [Gb] | 127[s] | 14 [Th] al 5 %, 2 [Th] al 65 % | 3,83 [Gb] | **Implementación con prueba personalizada.** Bien documentado. No tiene noción de espacio. Requiere preprocesado para llevar video a 50 fps. |
| Attention3D HumanPose | - | - | - | - | **Implementación con prueba sintética**. Falta documentación. Los principales algoritmos de segmentación 2D necesarios como entrada, no entregan los *joints* en el formato Human3.6m necesarios para este algoritmo. |
| MEVA | - | - | - | - | **Implementación con prueba personalizada**. La etapa de inferencia no entrega un archivo importable. |
| OpenPose | 2,22 [Gb] | 14,35[s] | 2 [Th] al 20 %, 1 [Th] al 80 %. | 3,67 [Gb] | **Implementación con prueba personalizada**. Problemas con cudNN y CUDA 10: esto trae consigo más consumo de memoria que el estándar. |
| MotioNet | 1,8 [Gb] | 25 [s] | 2 Th al 50 % | 9,07 [Gb] | **Implementación con prueba personalizada**. Depende de openpose. |
| Evoskeleton | - | - | - | - | **No se logra prueba sintética**. El algoritmo se ejecuta correctamente pero no entrega resultados. |

el tiempo de procesamiento es lento: procesar 600 *frames* toma 600 segundos. Para poder importar a Blender su resultado, se implementa un *script* que acomode los *joints* con ciertas rotaciones para que Blender pueda interpretarlo correctamente (se usa la Transformada de Rodrigues [2]).

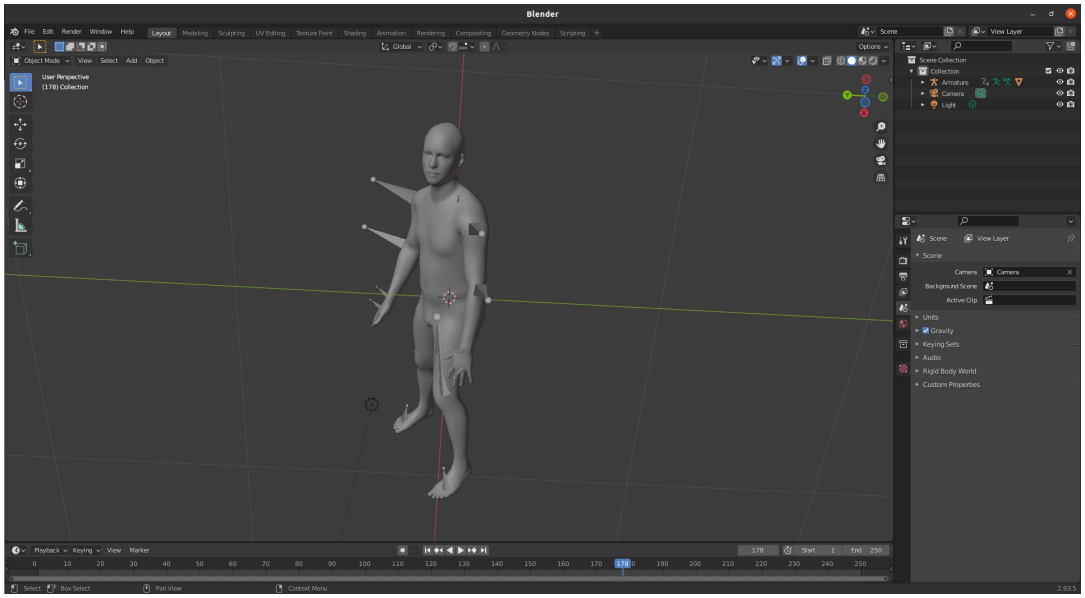



*Fig. 3.3:* FrankMocap en Blender. Tipo de archivo fbx (Contenido digital de mallas y *Motion Capture*).

En segundo lugar se prueba VideoPose3D. Se muestra la captura de movimiento en Blender. Sus resultados son buenos en general. Se aprecia la captura de movimiento sin la malla. El procesamiento de su salida se basa en leer un archivo donde guarda sus *joints* en un formato particular (un archivo de extensión **.npz** de Numpy), realizar un procesamiento que permite traducir su salida en formato Human3.6m[3] hacia un formato Blender compatible. Se usa la API de Blender para exportar estas posiciones a un archivo **bvh**[4].

Se probó Motionet, importando directamente a Blender su salida, ya que otorga la facilidad de exportar directamente un archivo **.bvh**, legible sin un post-procesamiento.

[2] Una descripción de esta función puede encontrarla aquí: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga61585db663d9da06b68e70cfbf6a1eac

[3] Este es uno de los *datasets* de entrenamiento/prueba de algoritmos de BwHT.

[4] Como se menciona en el marco teórico, **bvh** es un formato de salida para almacenar *joints*, **fbx** almacena cualquier tipo de información tridimensional, como *joints* y mallas.

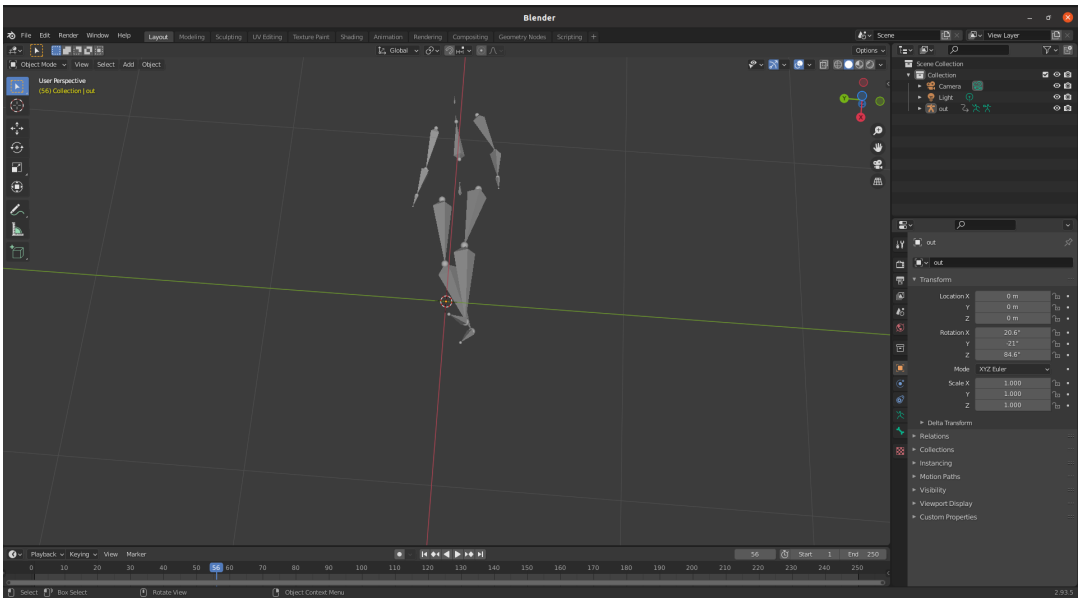



*Fig. 3.4:* VideoPose3D en Blender. Tipo archivo bvh (*Motion Capture*).

Sus resultados no son los esperados, ya que no detecta bien ciertos movimientos. Por ejemplo, las rotaciones del cuerpo de 60[°] se traducen en movimientos de 90[°]; giros cortos y rápidos se traducen en rotaciones de 360 [°] que nada tienen que ver con el movimiento que se muestra en el video original.

AttentionHumanPose no describía correctamente su etapa de inferencia, además de usar un formato de entrada poco común (formato de *joints* Human3.6m). Esto implicaba que la etapa de detección 2D requería un algoritmo que produjese una salida en ese formato (algo que nunca se usa; se buscan formatos estándar como COCO).

Con Evoskeleton no se obtuvieron resultados al ejecutar el algoritmo, y no desplegó ningún error. Buscando soluciones se logró encontrar comentarios del autor donde describe que sus funciones están implementadas para ser usadas en imágenes, no en videos.

Se han probado algoritmos de *tracking* de manos con distintas filosofías de funcionamiento, para comparar con la solución tentativa FrankMocap. Puede ver en la Tabla 3.2 para ver comparativas cuantitativas y cualitativas.

Minimal-Hand es el algoritmo de captura de manos más comentado y citado dentro

*Tab. 3.2:* Resultado de pruebas locales para algoritmos HT (Hand Tracking). se uso [Th] para indicar *Thread*: uso por CPU.

| **Algoritmo** | **GPU** | **Time** | **CPU** | **RAM** | **Resultados** |
|---|---|---|---|---|---|
| minimal hand | 1,10 [Gb] | Real Time 30 fps | 1 [Th] al 15 %, 2 [Th] al 10 %. | 813 [Mb] | **Implementación con pruebas sintéticas**. Rápido: es capaz a de inferir la posición en tiempo real de una mano. Requiere realizar proceso para detectar las dos manos. Requiere detectar con framwork de terceros la posición de las manos y entregar un recorte de 128x128 píxeles. |
| FrankMocap (solo manos) | 2,74 [Gb] | 790 [s] | 2 [Th] al 35 %, 2 [Th] al 10 %. | 4,10 [Gb] | **Implementación con pruebas personalizadas**. Integrado con FrankMocap. Fácil integración con módulo de cuerpo. Integra solución smplx. |
| InterHand2.6M | - | - | - | - | **Implementación con pruebas sintética**. Solo procesa por imagen y no tiene un visor claro de los resultados. |
| HandTailor | 1,75 [Gb] | 61 [s] | 1 [Th] al 35 %, 2 [Th] al 10 %. | 1201 [Gb] | **Implementación con pruebas personalizada**. Solo procesa por imagen. No existe orden en la salida de las imágenes resultantes, y por consiguiente, en la salida capturada en archivo *.pkl*. |
| EventHands | - | - | - | - | **Sin prueba satisfactoria**. El modelo entrenado no se entrega de manera libre. |
| body2hands | - | - | - | - | **Sin prueba satisfactoria**. El modelo no se libera aún. |

del *hand tracking*. En general se ha adaptado a varios contextos donde se necesita una captura rápida y eficaz de manos (e.g. captura de manos importadas a Unity). Trabajar con algoritmos específicamente diseñados solo para inferir posición 3D implica, en

general, realizar *crops* de las dos manos de baja resolución y a partir de allí inferir resultados. Esto implica utilizar una herramienta de detección de manos de terceros que retorne la detección delimitada por una *bounding box* y que cree el recorte de esa detección.

FrankMocap tiene integrada en su solución la capacidad de inferir la forma y posición de las manos en 3D, teniendo el video de cuerpo entero. Esto permite que no sea necesario llevar a cabo la implementación de la detección de las manos con un algoritmo de terceros: embebido en su etapa de inferencia realiza esta tarea. Se puede solicitar dentro de esa etapa que se desea la posición de las manos y del cuerpo, entregando segmentaciones en formato COCO en un archivo *.json* por *frames*. El hecho de que la herramienta integre la detección y ubicación de las manos le otorga una versatilidad que otros proyectos no tienen.

Proyectos como InterHand2.6M y HandTailor fueron probados y siguen la misma filosofía de Minimal-hand, pero las API de inferencia que estos algoritmos integran son poco prácticas para procesar videos. Por ejemplo, para realizar inferencias sobre un video cualquiera, debemos transformarle de video a *frames*, y modificar los códigos para que al leer y guardar sus resultados mantengan un orden acorde al video. Además, se requiere realizar una transformación desde sus salidas (e.g. desde el modelo MANO[5] hacia una perspectiva de movimiento que Blender pueda interpretar). Como Minimal-hand, estos proyectos también requieren un preprocesado, donde se detecten y recorten las manos *frame* a *frame*, para luego realizar la inferencia de la posición en cada uno de ellos.

Body2hands es un algoritmo para mejorar sustancialmente la capacidad de inferencia de otros algoritmos que capturan el cuerpo completo. Como entrada, recibiendo la posición estimada del cuerpo por algún algoritmo de estimación de pose 3D realiza una predicción con contexto, o sea, se beneficia de la posición de las manos (la posición del cuerpo influye directamente en la posición de la inferencia de las manos), lo que es muy superior a la metodología de otros algoritmos. Su problema central es que

[5] https://mano.is.tue.mpg.de/

aún no se libera el modelo de inferencia. Entrenarlo tomaría tiempo y recursos de los cuales no se dispone con las limitaciones actuales de tiempo de desarrollo. Garantizan compatibilidad con FrankMocap.

### *3.2.2. Pruebas de algoritmos de URem*

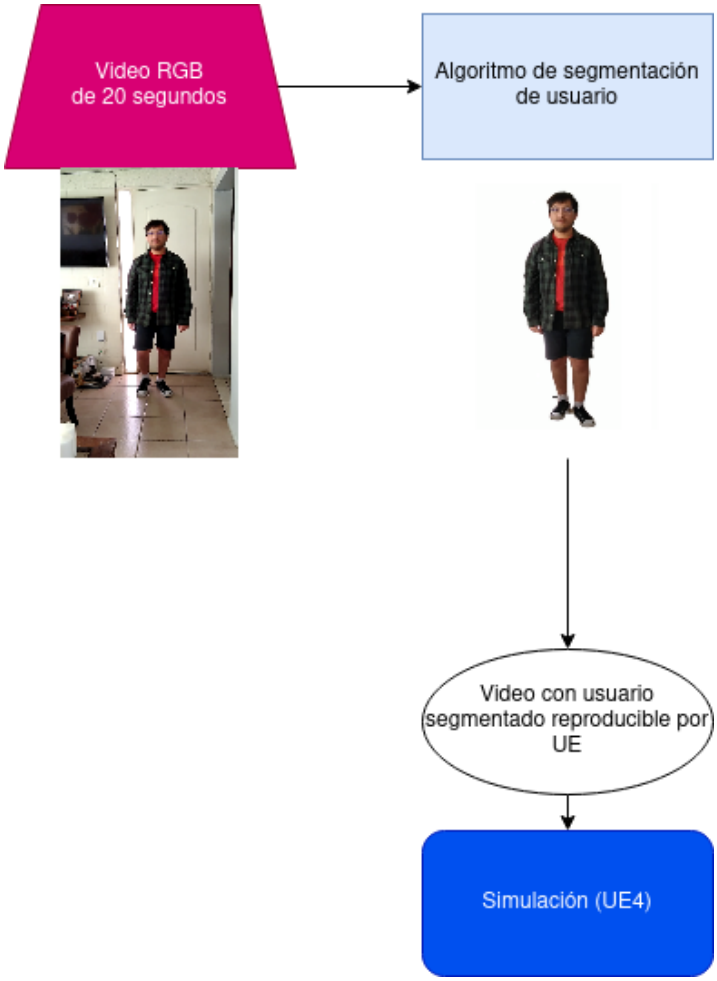



*Fig. 3.5:* Diagrama de la solución de URem.

En general, los algoritmos de segmentación de personas están bien documentados y son de fácil implementación, en comparación con los algoritmos de repintado o de *tracking* de personas. En el caso de la generación de máscaras para los algoritmos de repintado, deben ser obtenidos en esta etapa: el algoritmo los provee o debe generarse

a partir de su salida a través de filtros. En la Figura 3.5 se presentan los pasos asociados a este proceso.

La Tabla 3.3 nos muestra las soluciones implementadas. Las opciones presentadas muestran una buena respuesta a las necesidades de tiempo y calidad necesaria para este contexto.

*Tab. 3.3:* Resultado de pruebas locales para algoritmos URem.

| **Algoritmo** | **GPU** | **Time** | **CPU** | **RAM** | **Resultados** |
|---|---|---|---|---|---|
| MODNet | 1857 Mb | 46 [s] | 16 [Th], al 15 % c/u | 3.7 [Gb] | **Implementación con prueba personalizada.** Bien documentado. Para obtener la máscara de segmentación debemos preprocesar el resultado. |
| RVM | 5475 Mb | 96 [s] | 16 [Th], al 15 % c/u | 3.6 [Gb] | **Implementación con prueba personalizada**. Buenos resultados. Entrega máscara de segmentación como salida, no es necesario realizar el preprocesamiento. |

Para realizar una estimación de segmentación precisa, es necesario que los colores que componen al usuario (e.g. el pelo, la ropa) sean claramente diferentes del fondo, ya que la única fuente de información son los pixeles y su información RGB, sin información de profundidad para realizar algún otro tipo de identificación. Con ello, se debe tener en cuenta que televisores negros en pelo oscuros, o ropa con un color muy parecido al fondo causarían problemas en esta etapa.

En general, MODNet y RVM funcionan muy bien en ambientes disjuntos siempre y cuando se respeten las condiciones anteriormente mencionadas. MODNet requiere procesar su salida a través de *thresholding* y filtros para obtener máscaras de segmentación, mientras que RVM los entrega automáticamente.

### 3.2.3. *Pruebas de algoritmos de VIn*

La arquitectura de la solución del algoritmo de repintado coherente se muestra en la Figura 3.6. Como se muestra, se requiere el video original y una máscara de segmentación proveniente de el algoritmo de segmentación de usuario.

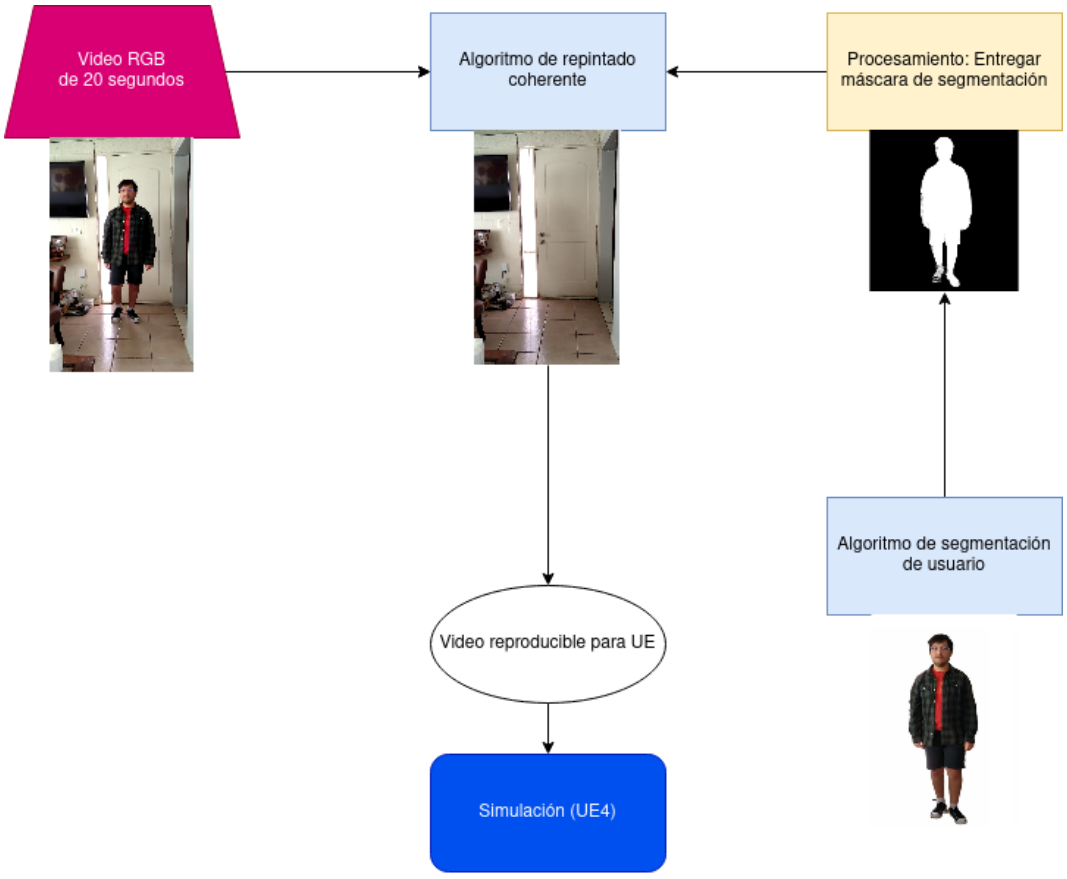



*Fig. 3.6:* Diagrama de la solución de VIn.

El rendimiento en general de los algoritmos de repintado es dispar, y su consumo de recursos es muy alto. En términos de GPU, el consumo de memoria es muy alto si se ejecuta en un video de 20 segundos (16 Gbs de memoria como mínimo). Procesar 1000 *frames* en un algoritmo de repintado se vuelve imposible, al menos en una tarjeta dedicada con 8 GBs de memoria. La solución es ejecutar el algoritmo interactivamente cada 100 *frames*. Esto implica ejecutar un preprocesado donde el video sea descompuesto en *frames*, reordenar en carpetas de 100 *frames* y ejecutar sobre cada

carpeta, de manera iterativa, el algoritmo a utilizar.

La Tabla 3.4 muestra una comparativa de los algoritmos de repintado. Todos los algoritmos requieren una máscara de segmentación, que explicita qué sector del video debe ser repintado. Este debe provenir de la salida del algoritmo de Urem, acondicionada para obtener una máscara de segmentación, si fuese necesario. El control del repintado y su agresividad se traduce en las zonas cubiertas por la máscara, con lo cual es trascendental ensancharle, para que en el caso de lograr zonas puntuales donde no se segmente absolutamente a toda la persona, se realiza el repintado igualmente.

*Tab. 3.4:* Resultado de pruebas locales para algoritmos VIn, donde los videos se han segmentado en carpetas de 100 *frames*.

| **Algoritmo** | **GPU** | **Time** | **CPU** | **RAM** | **Resultados** |
|---|---|---|---|---|---|
| STTN | 6.5 [Gb] | 184 [s] | 1 [Th] al 95 %, 1 [Th] al 15 % | 4,52 [Gb] | **Implementación con prueba personalizada.** Malos resultados. Bien documentado. |
| DSTT | 5.5 [Gb] | 144 [s] | 1 [Th] al 95 %, 1 [Th] al 15 % | 3,52 [Gb] | **Implementación con prueba personalizada**. Documentación símil a STTN. Bien Documentado. |
| TSAM | - | - | - | - | **Implementación con prueba sintética**. Falta documentación. No esta desarrollado para realizar pruebas con videos personalizados, la etapa de inferencia no está documentada. |
| IVII | 6,98 [Gb] | 1800[s] | 1 [Th] al 10 %, 1 [Th] al 20 % | 3,32 [Gb] | **Implementación con prueba sintética**. Con reducción de *batch size*, iteraciones y calidad de resultados se logra un tiempo de inferencia de 30 minutos (desde las 4 horas originales), excesivo en este contexto. |

DSTT fue el algoritmo que cumplió de mejor manera los parámetros buscados.

Su consumo de memoria es alto (al igual que todos los otros algoritmos probados, en torno a los 7 Gbs en un tramo de 100 frames), pero usando una máscara de segmentación como entrada que abarque todo el cuerpo, es capaz de repintar coherentemente.

### *3.2.4. Condiciones de ambiente en pruebas locales*

Las principales dificultades en torno a la instalación de las herramientas radican en un soporte poco transferible de los entornos Conda instanciados en otros computadores (i.e. en otros proyectos), donde los ambientes creados dependen de bibliotecas que no están presentes y causan problemas de ejecución. En general estos errores vienen sucedidos por una mala interconexión de Conda con los drivers de Nvidia y las capacidades de soporte del ambiente a características de CUDA, cuDNN y de dependencias asociadas que Conda no soporta de manera natural. Para ilustrar estas incapacidades, una tarjeta que utiliza la arquitectura RTX30 requiere la versión de CUDA más nueva posible para aprovechar sus características correctamente, junto a las versiones de Pytorch 1.9. Actualmente Conda solo soporta la 11.3 y la 11.0, lo cual impedía instalar la versión más nueva o una puntual. El nexo entre el compilador de Nvidia que se instala con sus *toolkits* y los *paths* dentro del ambiente Conda también son deficientes.

Se recomienda utilizar una versión de Ubuntu LTS. En específico, la versión 20.04 es muy recomendada en muchos repositorios. Para probar los algoritmos se utiliza *virtualenv* siempre que sea posible, para usar los controladores de la tarjeta de video que utilice Ubuntu (CUDA, controladores, cudNN, Nvidia Toolkit, etc), y dejar las dependencias asociadas a bibliotecas Python al ambiente *virtualenv*. En todo otro caso, será necesario configurar un ambiente Conda, que tiende a ser más pesado, más complicado de configurar y no tiene el mismo soporte en *containers* que el estándar *virtualenv*.

### 3.2.5. Selección de los algoritmos para cada problema

La selección de los algoritmos en cada una de las 3 etapas está condicionada por la precisión, el tiempo de inferencia y la factibilidad de aplicabilidad en este contexto.

Una alta precisión y un bajo tiempo de inferencia es deseable. Con ello, se premia que los tiempos de inferencia estén limitados a algunos minutos, y que la precisión refleje los movimientos propios del usuario: rotaciones, cambios de posición, movimientos de brazos y otros tipos de desplazamientos que ayuden a validar la efectividad de la solución.

El último punto implica tanto la capacidad de lograr transformar su salida a un **fbx** (o **bvh**; algún tipo de archivo soportado por Blender y UE) junto a las adaptaciones necesarias para lograr ser leídas e interpretadas en algún tipo de formato estándar. Esto implica la capacidad de mostrar las rotaciones y traslaciones respectivas según lo inferido.

Se elige **FrankMocap** como algoritmo de captura de cuerpo y manos (BwHT). La elección se induce por las características de inferencia (suficientemente precisa, pero sufre de *jitter*) como también la existencia de documentación de transformación de la salida a formatos compatibles con Blender que potencialmente son compatibles con UE. Su salida es un archivo SMPL-X con un formato no estándar de extensión **.pkl**, lo que implica que debe realizarse la transformación necesaria hacia un archivo **.fbx** que sea compatible con UE. La extensa documentación ayudará a materializar la trasformación. Además, como su solución de manos y de cuerpo están integradas, solo requerirá trasformar su salida para dar solución al problema de captura de manos y de cuerpo.

Se elige **DSTT** como algoritmo de repintado coherente, para dar solución al problema de la capa de fondo. Se elige por su capacidad de entregar un resultado de repintado coherente al contexto, siempre que se le entregue una segmentación de usuario lo suficientemente precisa. La luminosidad y la correcta elección del fondo es trascendental a la hora de obtener buenos resultados. Estas sugerencias se anexarán al

apartado de uso general derivado de las pruebas realizadas en este punto (en conjunto las pruebas se realizan en la evaluación de la solución).

Se elige **RVM** como algoritmo de segmentación de usuario, por tener mejor rendimiento teórico y presentarse en sus documentos como una mejor solución a MODNet. En general, los algoritmos analizados enfocados en personas se comportan bastante bien siempre y cuando el contraste entre el usuario a segmentar y el fondo sea razonable. Ejemplos donde el contraste puede dar problemas son situaciones donde las tonalidades del pelo sean las mismas que el fondo (e.g. pelo negro, fondo negro), ropa de color similar a fondos (e.g. polera negra junto a un televisor apagado donde se ve la pantalla negra) o zapatos de un solo color que se confundan con el piso. Estos factores deben ser explicitados en un apartado que funcione como instructivo para quien utilice la solución. Se debe hacer hincapié en que RVM utiliza más del doble de recursos que MODNet. Esto puede verse en la Tabla 3.3.

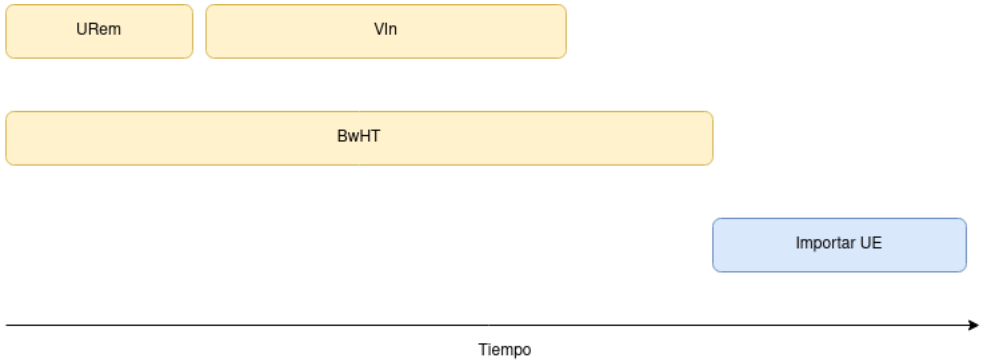



*Fig. 3.7:* Diagrama de ejecución de la solución general en el tiempo.

## *3.3. Limitaciones, perspectivas de uso y compromisos entre tiempo de inferencia y precisión de los resultados*

Existen limitaciones con respecto a la capacidad de inferir de manera coherente en todos los tipos de algoritmos. Naturalmente, como la información disponible se limita al video grabado, y no tenemos una noción de profundidad dada por algún sensor,

las operaciones se ven fuertemente influenciadas por el contraste entre la persona y el fondo. Las siguientes sugerencias se inducen por la necesidad de eventualmente ayudar a que exista una diferenciación pixel a pixel entre los objetos de interés:

- La precisión aumenta en la medida que el fondo sea limpio y homogéneo. Al realizar pruebas con objetos pequeños que compartan algún color con la ropa o el pelo, estos tienden a confundir al algoritmo de segmentación de usuario y son capturados en esa capa. Esto aumenta las zonas de repintado y el tiempo de ejecución del algoritmo.

- Los colores del fondo y del usuario a segmentar deben tener un contraste lo suficientemente alto. No debe usarse ropa o pelo del mismo color que el fondo. Esto incluye televisores de pantalla plana y cuadros que puedan tener parecidos con el pelo, el objeto más crítico a la hora de realizar la segmentación.

- La calidad del video debe ser por lo menos HD, aunque se recomienda que sea FHD, a 30fps.

- El video debe ser grabado verticalmente: no existe la opción de grabarlo horizontalmente, según requerimientos de la empresa Nexar.

# 4. INTEGRACIÓN E IMPLEMENTACIÓN DE LA SOLUCIÓN

Este capítulo esta dedicado a la construcción del modelo de 3 capas en Unreal Engine, el motor de videojuegos. En primer lugar es necesario implementar las tareas necesarias para lograr que cada salida de los algoritmos seleccionados sean interpretables por el motor.

En segundo lugar, se utilizan las herramientas que provee el motor de videojuegos para lograr la construcción de un escenario que materialice la arquitectura de 3 capas propuesta, junto con el acondicionamiento de la prenda a simular para así obtener la respuesta dinámica buscada.

## *4.1. Implementación de la solución en Unreal Engine*

La integración de la solución con UE implica los siguientes pasos:

- Acondicionamiento de salidas desde el algoritmo de BwHT para lograr importar la *animation* asociada a esa captura hacia UE.
- Preconfigurar *assets* y crear la escena: *materials*, *textures*, *bones* y un *mesh* deben estar en una escena (*level* instanciados dentro de UE, para solo importar la animación proveniente de el algoritmo de BwHT: todo lo demás ya está predefinido).
- Implementación de la estructura de 3 capas: 2 pantallas (2 actores) que reproduzcan un video adquirido de las etapas de URem y VIn. Luego, ubicar estratégicamente el avatar que reproduce los movimientos capturados (que será la tercera capa).

A continuación se explicará la implementación de lo antes mencionado.

### *4.1.1. Importar BwHT a UE*

Para importar el resultado de FrankMocap a UE, necesitamos usar un algoritmo que permita acondicionar su salida. Ya se ha implementado una solución que permite llevar a Blender (y también a UE, por consiguiente) la salida de FrankMocap a través de un **.fbx**, pero sin considerar los dedos. La fuente de esta transformación para el cuerpo es una sección de código de un proyecto de BT *(Body Tracking)* llamado VIBE[1]. La implementación de los dedos requiere desarrollo directo ya que no hay una solución existente que de soporte a esta característica.

Existe un proyecto de referencia que rescata también los dedos, utilizando como entrada Easymocap, un algoritmo de BwHT que usa varias cámaras en distintas perspectivas para inferir la posición 3D de cuerpo y manos de manera robusta. Una implementación en Python para el programa Blender transforma la salida de este algoritmo para que pueda ser interpretada correctamente, incluyendo tanto la posición del cuerpo como de las manos. Se espera realizar un reacondicionamiento de este *add-on* de Blender, que permita interpretar los dedos provenientes de FrankMocap.

### *4.1.2. Acondicionamiento del algoritmo de VIn*

A diferencia del algoritmo de URem, el uso de máscaras y acondicionamiento del algoritmo de VIn (DSTT) requiere mayor procesado, como ya se describió en la sección 3.2.3. Es necesario entonces, implementar un *script* que permita realizar las transformaciones para obtener una salida útil, de manera automática.

Se tiene, en un inicio, el video grabado por el usuario. En primer lugar, este tiene que transformarse de video a *frames*, dada la forma en la que funciona DSTT. Esta operación debe también realizarse para las máscaras de segmentación, ya que el algoritmo de URem entrega un video en formato **.mp4**. Una vez realizada esta operación,

[1] https://github.com/mkocabas/VIBE

se requiere separar en carpetas los *frames*, por la limitación de memoria antes discutida. Esta operación de separación se ejecuta tanto para las máscaras como para los *frames* del video original.

Se implementa una API en Python que permite explicitar cuántos *frames* se desean por carpeta, para lograr más control sobre la ejecución de todo el algoritmo. El algoritmo DSTT, entonces, se ejecutará tantas veces como carpetas se creasen en este proceso.

Antes de realizar la ejecución iterativa de DSTT, se sugiere utilizar los filtros morfológicos probados en la sección anterior, para mejorar el resultado de las máscaras. Un ejemplo del efecto que estos filtros tienen puede visualizarse en la Figura 4.1.

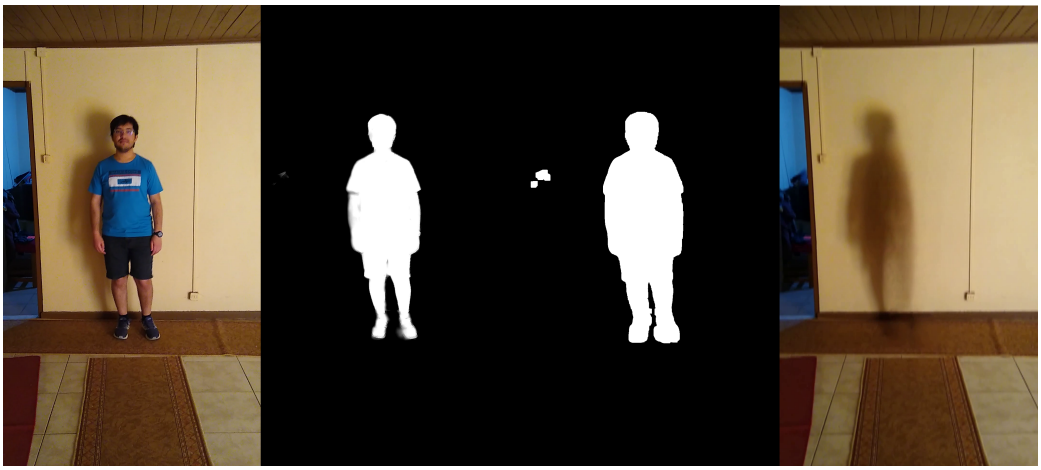


*Fig. 4.1:* De izquierda a derecha: 1) *Frame* del video original. 2) *Frame* del algoritmo de URem que genera máscara de segmentación. 3) Máscara de segmentación procesada por filtros morfológicos. 4) Resultado del repintado de DSTT.

Si requiere rotar los *frames* generados a partir de las máscaras de segmentación de salida de URem (si se entrega un video de salida que no coincide con la posición del video original 1920x1080) se genera un *script* que realiza esta tarea. Así, los *frames* obtenidos se rotarían en 90 grados (posición vertical, 1080x1920). Esto se presupone útil dado que es posible que el video grabado no tenga los metadatos correctos con respecto a su formato y tamaño, algo que ocurrió en las pruebas realizadas con un teléfono Android. El hecho de no contar con estos datos genera errores en la salida de

URem y se obtendrá una salida correcta pero rotada, que este algoritmo soluciona.

La salida del algoritmo de repintado, DSTT, son videos. Esto implica que tendremos un video por cada vez que se ejecutó el algoritmo. Se implementa un código en Python que une los videos obtenidos en uno solo. Este video entonces, recopila todos los *frames* que fueron repintados, unidos en el orden correcto.

### *4.1.3. Implementación del modelo de 3 capas en Unreal Engine*

La implementación en Unreal Engine que da solución al problema se basa en 2 *actors* principales en el *level*: uno representa una pantalla donde se reproduce el video, mientras otro es el avatar que se mueve en la escena. La estructura del avatar estará predefinida dentro de UE, mientras que el movimiento (la animación) se rescatará de FrankMocap. Esto puede apreciarse de manera visual en la Figura 4.2.

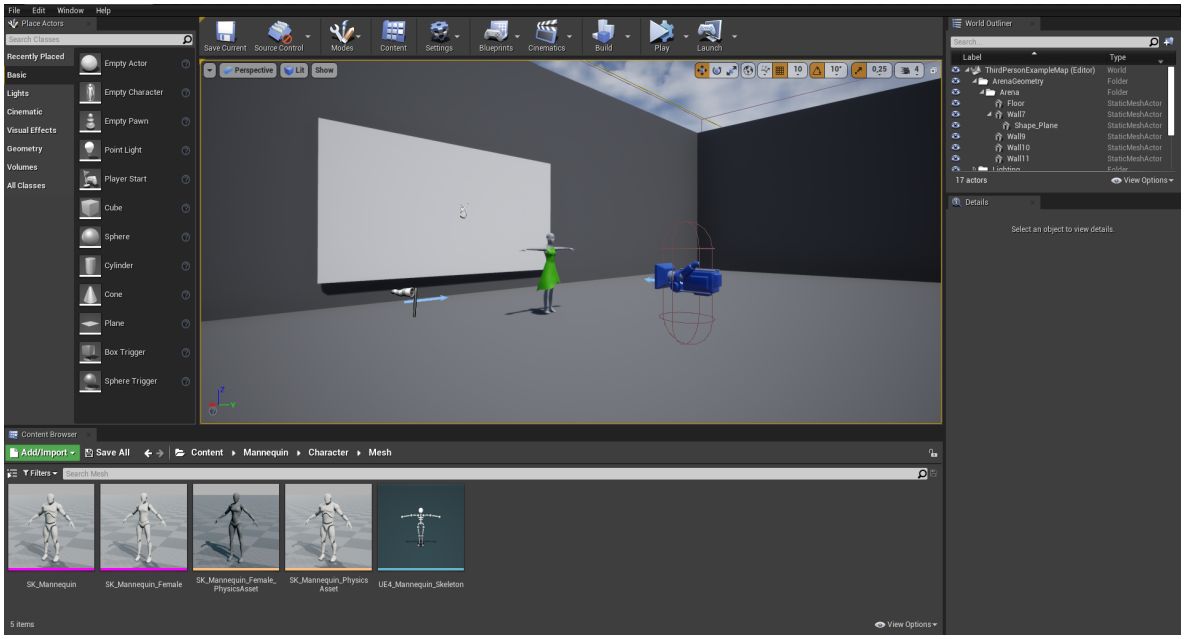



*Fig. 4.2:* Perspectiva con una cámara, un maniquí, y una capa de fondo que reproduce un video en UE.

La configuración del *mesh*, junto a la dinámica de una ropa específica unida a ese *mesh*, son algunas de las etapas complejas a desarrollar. Para realizar operaciones sobre los modelos a importar a UE, utilizamos Blender. Para nuestro caso, usaremos

una malla de una falda como prenda de prueba a anexar al modelo de cuerpo, como se observa en la Figura 4.3.

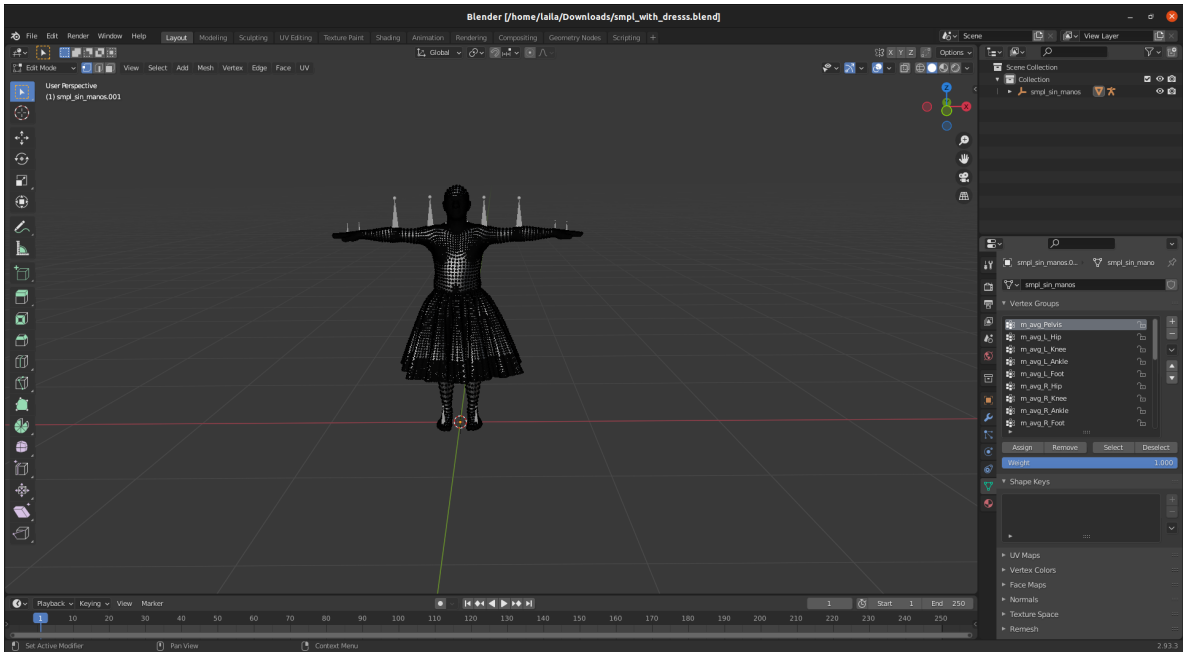



*Fig. 4.3:* Composición dentro de Blender y manejo de componentes del cuerpo capturado.

En la imagen puede distinguirse la prenda, el *mesh* del cuerpo y el *skeleton* (la estructura de *joints* de movimiento representada por los triángulos en la imagen 4.3). Una vez unidos los *mesh* de la falda y el cuerpo, estos compartirán los vértices que se aprecian en la imagen. Las pruebas para obtener la dinámica buscada comprenden un manejo de varias opciones de implementación de esta estructura en Blender que luego fueron probadas en UE, como se ve en las Figuras 4.4 y 4.5.

Las posibles combinaciones que se prueban son las siguientes:

1. Anexar al *mesh* de la falda al *mesh* del cuerpo.

2. Unir con un *blueprint* dentro de UE el *mesh* de la falda y el cuerpo, para establecer una dinámica interna en este archivo conjunto dentro de UE.

3. Anexar el *mesh* de la falda al *skeleton* del cuerpo, para luego unir ambos modelos dentro de UE a través de un *blueprint*.




*Fig. 4.4:* Implementación en UE de varias opciones de *mesh*.




*Fig. 4.5:* Implementación en UE de opciones de *mesh*, con dinámica asociada a las prendas.

4. Anexar el *mesh* de la falda al *skeleton* del cuerpo y los *colliders* del cuerpo capturado. Simular el movimiento de la falda interactuando con estos *colliders*. Anexar el *mesh* visible del cuerpo dentro de UE a través de *blueprints*.

5. Anexar el *mesh* de la falda al *skeleton* del cuerpo, los *colliders* y el *mesh* del cuerpo capturado.

La opción que entrega mejores resultados es la opción número 4. Otras opciones no entregan dinámicas fiables, o simplemente no se desplazan (1 y 5). Otros casos

no usan el *mesh* base como se espera: sus *colliders* no funcionan y no desprenden una dinámica acorde a la presencia del cuerpo (2). En general la única manera de lograr una implementación según los criterios buscados es probar iterativamente las configuraciones: la información en la internet es acotada. Con las opciones propuestas implementadas, se logra una modularidad (el *mesh* de la falda es intercambiable) y una interacción correcta con los *colliders* del cuerpo al configurar el modelo explicitando *mesh*, *colliders* para el cuerpo y *colliders* para la falda.

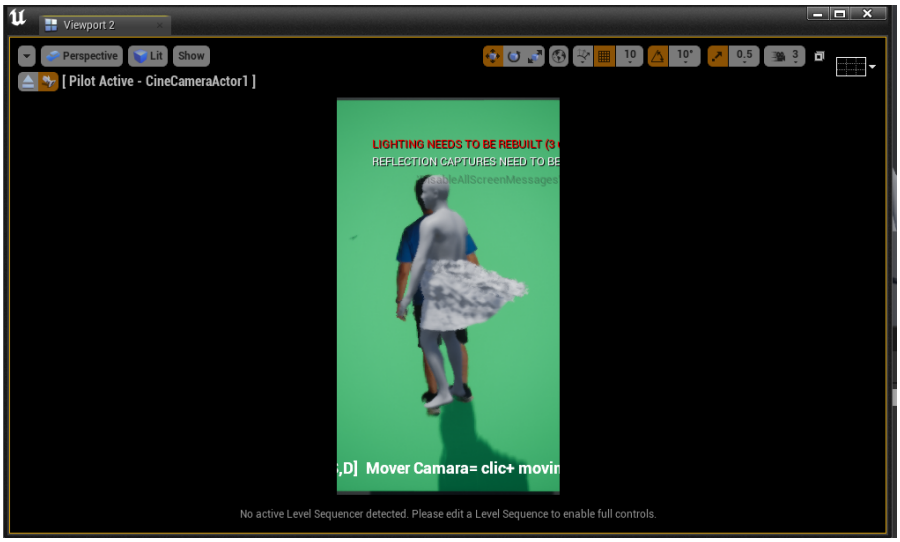



*Fig. 4.6:* Cámara específica capturando la dinámica de la ropa dentro de UE.

La aplicación de una respuesta dinámica de la tela se realiza completamente dentro de UE, a través de Chaos Cloth. La implementación consta de una serie de pasos bien determinados: se explicita a través de una interfaz qué sectores de un *mesh* dado se comportarán como una tela, y se le asignan pesos y estructuras que darán lugar a un comportamiento específico. Habrá interacción con todos los *colliders* de la escena.

Con esta prueba, se identifican los componentes necesarios para implementar la solución final dentro de UE: las dos pantallas junto a una tela con una dinámica que responda en tiempo real a los movimientos capturados. Los pasos posteriores buscan encaminar la implementación de una configuración de dos capas.

Los videos mostrados en Unreal Engine requieren que se posicione sobre una car-

peta todos los **.png** correspondientes a ese video, y no un **.mp4**. Esto es una limitación propia del motor.

Luego de tener la perspectiva buscada y las pruebas de movilidad de la tela probadas, se procede a generar la primera capa con el usuario segmentado. Esto se logra reutilizando la máscara entregada por el algoritmo de URem (previamente utilizado por módulo VIn). Aunque se puede procesar directamente el video con el fondo verde mostrado en la Figura 4.6, utilizar las máscaras de segmentación junto al video original permite un manejo más preciso sobre lo que se desea mostrar y lo que no.

Luego de procesar, se requiere que Unreal Engine extraiga el canal *alpha* de cada imagen a mostrar para que interprete el canal correctamente.

Así, obtenemos la estructura de 3 capas buscada, que cumple con mostrar solo al usuario en una de las capas y el fondo detectado en la segunda. Esta estructura puede visualizarse en la Figura 4.7. La distancia que se ve entre el usuario capturado, y el fondo repintado, muestra la separación de ambas capas.

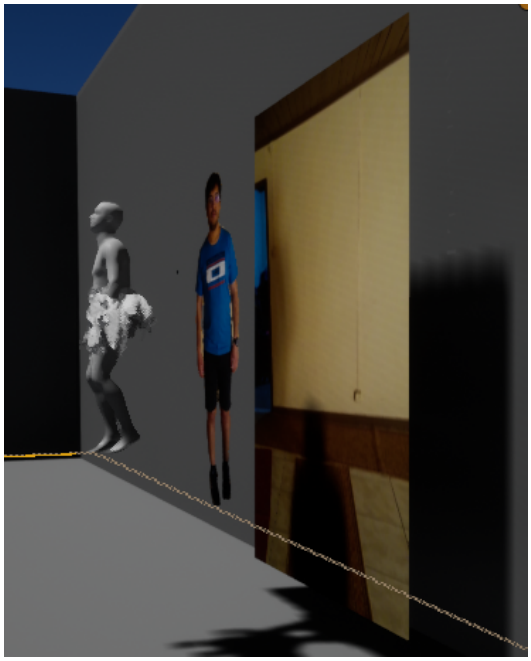


*Fig. 4.7:* Estructura de 3 capas dentro de UE.

Con las capas implementadas dentro de Unreal Engine, se descubren problemas

de implementación relacionados con los fps del video utilizado al ser leidos por el módulo RVM.

Al parecer las cámaras de los teléfonos Android (un teléfono Motorola G7 Power para este caso) no graban siempre con el *framerate* declarado, si no más bien con uno muy cercano. En este caso, con los 29,24 fps, una biblioteca de RVM aproximará a 30 fps las salidas, y los *frames* de este video obtenido no estarán perfectamente alineados temporalmente. Esto causa evidentes problemas visuales cuando se repinta, ya que las máscaras obtenidas con RVM, con estas diferencias, causarán un repintado en secciones erróneas, al utilizar como base el video original transformado a *frames*. Este error es percibido en la imagen 4.8. Allí, se muestran los *frames* del video original y las máscaras generadas de esos mismos *frames*. Las máscaras y el video generan las mismas cantidades de *frames*, por lo que el error es difícilmente identificable utilizando el conteo de *frames* para asegurar la sincronización.

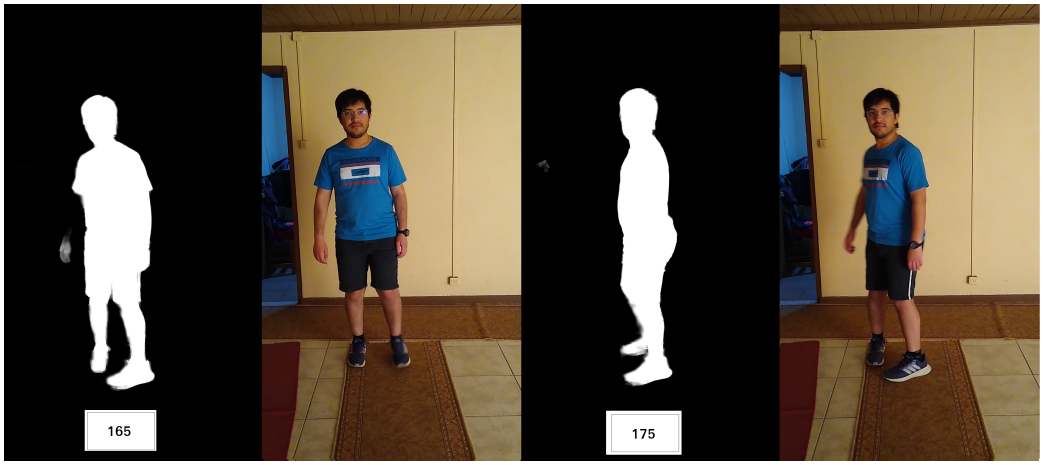



*Fig. 4.8:* Error percibido en la salida de RVM en los *frames* 165 y 175 de un video de 618.

Este problema se soluciona preprocesando el video original, y fijando entonces el *framerate* para garantizar que sean 30 fps. Además, se configuran parámetros de resolución que, aunque sean respetados por el video, no son identificables en sus características por el sistema operativo. Al parecer la grabación de video en Android tampoco configura *labels* correctamente en el video. En el caso que no se configure la resolución, la salida de RVM estará rotada y deberá girarse manualmente (con un

*script*), como hasta ahora se había realizado. Estas configuraciones pueden apreciarse en la Figura 4.9.

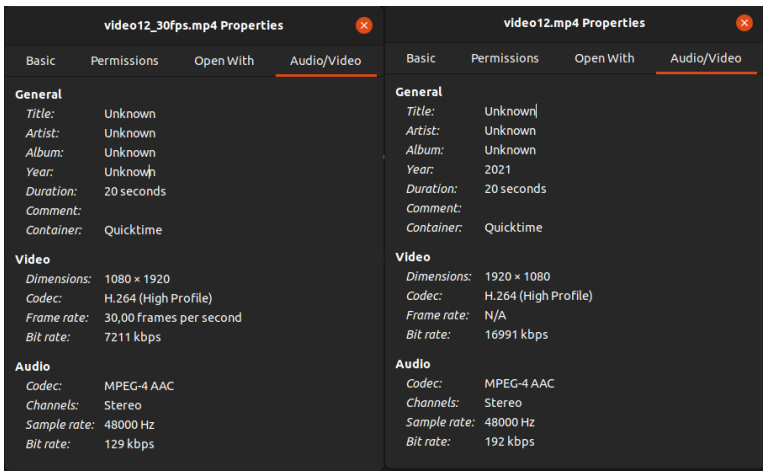



*Fig. 4.9:* Cambios de configuración sobre el video original.

## 4.2. Evaluación de la solución

### 4.2.1. Herramientas de evaluación: Pruebas QA

En esta sección, se proponen pruebas a ejecutar sobre el proyecto final desarrollado en virtud de evaluar la flexibilidad, rendimiento y operabilidad de la solución desarrollada.

La prueba QA es el proceso de encontrar errores en el producto desarrollado. También verifica si los resultados reales pueden coincidir con los resultados esperados, así como ayuda en la identificación de defectos o requisitos que faltan para la solución. Incluye examen, análisis, observación y evaluación de diferentes aspectos de un producto.

Las diversas pruebas a registrar pueden incluir tanto pruebas manuales como automatizadas. Esencialmente se busca someter a la solución a dos tipos de pruebas:

- Pruebas funcionales: verificar el funcionamiento de cada aplicación relacionada directamente con el servicio. Aquí el código fuente no juega un rol predominante. Algunas pruebas incluyen:
  - *Unit testing*: probar los componentes individuales de la solución.
  - *integration testing*: probar los módulos trabajando en su conjunto.
  - *interface testing*: pruebas de interconexión entre salida/entrada de dos módulos.
  - Entre otros.
- Pruebas no funcionales: en estas pruebas se consideran parámetros como flexibilidad, confiabilidad, usabilidad y rendimiento. Algunas pruebas incluyen:
  - *Performance testing*: se busca registrar velocidad y tiempo de la solución desarrollada.
  - *Stress testing*: pruebas que extralimitan los parámetros predefinidos de una operación regular.

- *Compatibility testing*: pruebas de compatibilidad en arquitecturas disjuntas.
- *Usability testing*: pruebas relacionadas con la interfaz de usuario y la capacidad de interactuar este y presentar errores.
- Entre otros.

En esta sección nos enfocaremos en las pruebas que tengan sentido en este contexto. Existen más tipos de pruebas dependiendo de la solución, que en este documento no se abordarán[2].

### *4.2.2. Pruebas a realizar*

Se proponen las siguientes pruebas categorizadas por tópico:

- Pruebas funcionales:
  - Pruebas por etapa. Principalmente podemos diferenciar tres etapas esenciales en la solución: el procesado del video a través de los tres algoritmos, el acondicionamiento de la solución para que UE pueda interpretar las salidas, para finalmente, como última etapa, importar las soluciones ya acondicionadas a UE (se pueden diferenciar las etapas en el diagrama 3.1). Debemos probar el proceso de importación y exportación de resultados automatizados en estos tres módulos para garantizar su funcionamiento por separado. Ya se han probado las etapas de procesado en la etapa de selección de los algoritmos, y con ello, la tarea debe extenderse a los dos módulos faltantes.
  - Prueba manual generalizada con entradas variables, respetando las condiciones de entrada predefinidas: Esta prueba se compone de varios *tests* generalizados con varios videos de entrada con distinta ropa, que cumplan los criterios de color y fondo descritos, además de respetar la duración y

[2] Para mayor detalle visite: https://www.testim.io/blog/software-testing-basics/

la calidad FHD a 30fps. Se calcula que 4 pruebas son lo suficientemente representativas.

- Pruebas no funcionales

    - Pruebas de estrés de calidad de video. Estas pruebas rompen con las suposiciones de calidad del video en torno a su calidad de iluminación, de ambiente (de objetos molestos, con colores que confundan al algoritmo) y de color de fondo. Puede esperarse una prueba que contenga mala iluminación, y una prueba que contenga color y objetos de interés no deseados.
    - Pruebas de estrés de calidad de video enfocadas en los algoritmos de URem y de VIn. Estos *tests* se plantean como una forma de aislar los problemas asociados al procesamiento del video en torno a su calidad de iluminación, de ambiente (de objetos molestos, con colores que confundan al algoritmo) y de color de fondo. Estas pruebas son necesarias dado que conocemos de antemano que estos factores influyen notablemente, más en estos dos algoritmos que en el BwHT.
    - Pruebas de estrés relacionadas con la calidad de grabación del usuario. Estas pruebas se relacionan directamente con la forma de grabación del video, esto es, la disposición espacial de la cámara usada. No se alineará al usuario con la cámara y se reproducirá un ángulo anormal. Una segunda prueba es realizar la grabación en horizontal.
    - Pruebas de rendimiento en el uso de recursos. Se hará, junto a las pruebas iterativas del módulo completo, un registro del uso de recursos asociados a la ejecución normal de la solución (i.e. con los videos cumpliendo las restricciones).

### *4.2.3. Métricas*

Para llevar a cabo una comparación cuantitativa y cualitativa, realizando una evaluación de rendimiento por etapas y descubrir el alcance de la solución en general, es necesario registrar el tiempo de ejecución de cada etapa de procesamiento, además de mostrar varios resultados visuales. Así, se logrará diagnosticar cuáles son las etapas más lentas, y en general, descubrir cuál módulo requiere retoques y desarrollos futuros. Los módulos de ejecución son:

- P1: el primer módulo de ejecución asociado a la etapa de repintado (VIn) y de segmentación del usuario (URem).
- P2: el módulo de ejecución que corresponde a la detección de los movimientos del usuario a través del algoritmo de BwHT.
- P3: el último módulo de ejecución que corresponde a la transformación de la salida de la etapa anterior (P2) hacia un *.fbx* compatible con UE.

Esta separación se induce por las dependencias que cada módulo requiere para su ejecución. P1 requiere todas las dependencias del algoritmo de URem y VIn. Dada la estrecha relación de estos dos algoritmos, y el hecho de que VIn solo puede comenzar a ejecutarse cuando se procese la salida de URem, es natural que se encuentren embebidos en un mismo módulo lógico que comparta las dependencias de Python en un ambiente virtual. P2 es el ambiente que el algoritmo de FrankMocap requiere para su ejecución, mientras que P3 requiere la biblioteca de Blender como módulo (**bpy**) para ejecutarse correctamente. Dada la sensibilidad de esta instalación, se decide aislar estas dependencias en un ambiente Python separado de FrankMocap.

Cada módulo de ejecución consta de pequeñas etapas que tienen tareas bien definidas. Las pruebas registrarán resultados particulares de cada una de estas etapas. En primer lugar, se describirá cada etapa para comentar la relación entre el tiempo de ejecución y la tarea puntual que debió realizar.

Para P1 tenemos los siguientes resultados:

E1 **Borrar Basura:** esta etapa borrará toda la información de ejecuciones pasadas (i.e. basura). Su tiempo ejecución es marginal, pero es necesario que exista para evitar errores.

E2 **Algoritmo Rvm:** en esta etapa se ejecuta el algoritmo de URem sobre el video de interés. Se entregarán las máscaras en un video **.mp4** como salida.

E3 **Video a Frame:** esta etapa transforma el video de entrada y las máscaras de segmentación a *frames*, para ser utilizado por el algoritmo de repintado.

E4 **Rotar Imagen:** esta etapa se encarga de la rotación de las imágenes en 90 grados, en el caso que la entrada del video sea de 1920x1080, en vez de los 1080x1920 solicitados como entrada estándar (grabación horizontal y no vertical). Puede ignorarse este tiempo de procesado si se asume que siempre se grabará el video en vertical.

E5 **Filtro Foreground** esta etapa está encargada de operar sobre las máscaras de segmentación, *frame* a *frame*. Aplicará filtros morfológicos sobre ellas para controlar el repintado.

E6 **Procesar Carpetas:** esta etapa separará los *frames* en carpetas tanto del video original como de las máscaras, para ejecutar iterativamente el algoritmo DSTT. De manera predeterminada, la cantidad de *frames* que tiene una carpeta son 100.

E7 **Algoritmo DSTT:** este algoritmo ejecuta DSTT tantas veces como carpetas se hubiesen creado en la etapa E6.

E8 **Video a Frame DSTT** este algoritmo transforma la salida final de DSTT hacia *frames*, para que UE pueda interpretarlos correctamente.

E9 **Procesar Fondo Verde:** esta etapa permite generar el canal *alpha* a partir de los *frames* del video original y las máscaras inferidas originalmente por el algoritmo de URem.

Para el módulo P2, tenemos una sola etapa:

E1 **Algoritmo FrankMocap:** esta etapa ejecuta FrankMocap para capturar el movimiento del usuario en el video original subido.

Para el módulo P3, tenemos una sola etapa:

E1 **FrankMocap FBX:** esta etapa transforma la salida de FrankMocap hacia un **fbx** interpretable por Unreal Engine (UE).

Los videos seleccionados son demostrativos con respecto a lo que se desea medir en las pruebas anteriormente mencionadas. Entre ellos encontramos videos que presentan rotaciones, distintos fondos, videos con el mismo fondo pero ropas distintas, etc. Las características de los videos utilizados para las pruebas son:

- Video30: video con iluminación irregular, que no siempre captura el cuerpo completo en el video. Video de muy mala calidad.
- Video29: video que cumple parámetros de iluminación y captura de cuerpo completo en todo el video, con fondo irregular.
- Video28: video que cumple parámetros de iluminación, pero no captura el cuerpo completo en todo el video, con fondo regular.
- Video26: video que cumple parámetros de iluminación y captura de cuerpo completo en todo el video, con fondo irregular.
- Video23: video que cumple parámetros de iluminación y captura de cuerpo completo en todo el video, con fondo regular. Mismo fondo que el video 22, pero con una prenda distinta.
- Video22: video que cumple parámetros de iluminación y captura de cuerpo completo en todo el video, con fondo regular. Mismo fondo que el video 23, pero con una prenda distinta.

- Video21: video que cumple parámetros de iluminación y captura de cuerpo completo en todo el video, con fondo levemente irregular. Grabado horizontalmente.

- Video20: video que cumple parámetros de iluminación, pero presenta una rotación irregular de aproximadamente 30 grados, con fondo levemente irregular.

Los resultados procesados para cada uno de estos videos dentro de UE se muestran en la Tabla 4.1.

*Tab. 4.1:* Tabla de videos procesados y ejecutados en UE. Los videos 5 y 6 son resultados representativos de videos bien grabados, que entregan los resultados esperados.

| | | |
|---|---|---|
| 1 - Video30 | 2 - Video29 | 3 - Video28 |
| 4 - Video26 | 5 - Video23 | 6 - Video22 |
| 7 - Video21 | 8 - Video20 | |

En estos videos se demuestra la capacidad de los módulos P2 y P3 de capturar el movimiento de la persona, y llevar ese resultado a UE. Como se aprecia en los videos, el hecho de no contar en la etapa de transformación a **fbx** con una relación de posición en el espacio del modelo SMPL, causa una desalineación entre la persona del video y el maniquí. Esto es apreciable al no superponerse perfectamente el maniqui a el usuario en UE. A pesar de aquello, es posible distinguir la captura del movimiento, junto a la dinámica de la ropa simulada sobre Unreal Engine.

Así, con los videos ya procesados, nos encontramos con los siguientes tiempos de ejecución mostrados en las Tablas 4.2, 4.3 y 4.4. En ellas se desglosa el rendimiento por etapa, además de mostrar cuántos *frames* tenía cada video de prueba.

De los resultados se puede concluir que las etapas que definen el tiempo de demora clave en cada módulo son:

- La etapa E9 presente en el módulo P1 requiere un tiempo de procesamiento excesivo, ya que solo genera los canales *alpha*. Esto se produce por alguna función no optimizada al recorrer pixel a pixel una imagen. Se menciona en

*Tab. 4.2:* Tabla de resultados I: Se muestran la etapa E1, E2, E3, E4 y E5 del módulo P1. Los valores están en segundos.

| | P1 | | | | |
|---|---|---|---|---|---|
| | E1 | E2 | E3 | E4 | E5 |
| 1 - Video30 | 0,19 | 94,00 | 35,96 | 42,24 | 14,65 |
| 2 - Video29 | 0,38 | 116,00 | 53,41 | 44,29 | 16,39 |
| 3 - Video28 | 0,33 | 72.00 | 39,25 | 31,45 | 12,71 |
| 4 - Video26 | 0,16 | 93,00 | 39,62 | 40,16 | 15,26 |
| 5 - Video23 | 0,39 | 90,00 | 49,54 | 37,28 | 14,85 |
| 6 - Video22 | 0,22 | 105,00 | 50,78 | 42,17 | 16,54 |
| 7 - Video21 | 0,32 | 89,00 | 41,74 | 36,73 | 15,25 |
| 8 - Video20 | 0,34 | 86,00 | 39,89 | 36,92 | 14,55 |

*Tab. 4.3:* Tabla de resultados II: Se muestran la etapa E6, E7, E8 y E9 del módulo P1. Los valores están en segundos.

| | P1 | | | |
|---|---|---|---|---|
| | E6 | E7 | E8 | E9 |
| 1 - Video30 | 0,76 | 148,34 | 28,45 | 264,92 |
| 2 - Video29 | 0,67 | 147,45 | 32,19 | 250,52 |
| 3 - Video28 | 0,86 | 132,78 | 24,73 | 213,21 |
| 4 - Video26 | 0,76 | 153,32 | 23,13 | 231,40 |
| 5 - Video23 | 1,08 | 154,39 | 27,14 | 228,71 |
| 6 - Video22 | 1,09 | 170,49 | 27,98 | 249,49 |
| 7 - Video21 | 0,73 | 148,90 | 23,47 | 240,57 |
| 8 - Video20 | 0,67 | 144,42 | 24,29 | 228,32 |

el trabajo futuro optimizar esta sección para lograr tiempos mucho menores (paralelizable a través de procesos GPU y optimización con Cython).

- La ejecución de los algoritmos de URem y de VIn en el módulo P1 requieren un tiempo considerable. Estos tiempos son irreductibles, dado que son módulos predefinidos por la implementación de sus autores, y existe pocas formas de optimización de tiempo de ejecución.

- La etapa P1-E4 es solo referencial, en el caso de necesitar rotaciones en el video de entrada. No es necesario contabilizar este tiempo como parte de la ejecución

*Tab. 4.4:* Tabla de resultados III: Se muestran la etapa E1 del módulo P2. Se muestra la etapa E1 del módulo P3. Se explicita la cantidad de *frames* por video, como una forma de relacionar los tiempos de funcionamiento con la carga de trabajo. Los valores están en segundos.

| | P2 | P3 | |
|---|---|---|---|
| | E1 | E1 | Frames |
| 1 - Video30 | 881,66 | 29,38 | 601 |
| 2 - Video29 | 1140,93 | 35,00 | 613 |
| 3 - Video28 | 734,14 | 27,83 | 468 |
| 4 - Video26 | 1070,48 | 30,63 | 626 |
| 5 - Video23 | 907,75 | 28,86 | 617 |
| 6 - Video22 | 885,41 | 29,49 | 609 |
| 7 - Video21 | 778,14 | 23,18 | 620 |
| 8 - Video20 | 914,72 | 23,18 | 610 |

estándar, pero era necesario cuantificar el tiempo necesario en el caso que el video de entrada no cumpliera con el requisito de grabación vertical.

- Los tiempos de ejecución de los módulos P2 y P3 son los esperados, ya que se habían estudiado previamente sus tiempos de ejecución. Con ello, es poco probable lograr mayores optimizaciones para estos algoritmos.

Se pueden estudiar los resultados del algoritmo de repintado buscando las zonas pintadas no coherentes. Si existe presencia de algún elemento que se espera que no esté presente, se considerará que el repintado no fue lo suficientemente preciso: un ejemplo puede verse en los *frames* presentes en la imagen 4.10. Elementos que pueden considerarse errores en el repintado que pueden estar presentes son: dedos, brazos, puntas de zapatillas, etc.

Los resultados se expresan en la Tabla 4.5. Allí, se muestra errores de distinta índole, tanto en pixeles como en porcentajes. En un video correctamente grabado (como mínimo con una correcta iluminación), el error está asociado al calzado: cuando este comparte colores con algún elemento del fondo o de los objetos de ese fondo, el algoritmo de segmentación de usuario URem no captura correctamente la zona, y no se repinta. La iluminación errática se muestra como una gran fuente de error: aumenta



*Fig. 4.10:* Imágenes que muestran lo que se considera un error de repintado: existe presencia de algún objeto asociado al usuario que no debiese estar presente.

el porcentaje de éste hasta un 10,29 %, lo que no es aceptable a la hora de entregar resultados. Si omitimos esta muestra, el error promedio entre los 7 otros videos que cumplen una calidad mínima de iluminación es de 0,25 %, un valor acotado que mantiene un resultado coherente en las imágenes.

Con los resultados expuestos, se denota un repintado coherente siempre y cuando el video a procesar presente buena iluminación: los rangos de error con esta restricción están, a lo sumo, en torno al 0,4 %. Las zonas donde se repinta el usuario tienden a

*Tab. 4.5:* Tabla de resultados del algoritmo de repintado del módulo P1 (que a su vez usa el módulo de segmentación de usuario). FE/FT: *frames* definidos con error sobre los *frames* totales. TFE: tramo de *frame* con error. EP: cantidad estimada de error, reportado en pixeles por *frames* en el tramo TFE. EP %: porcentaje de error basado en EP dividido en el número de pixeles que el formato de grabación FHD.

| FT | FE/TF | TFE | EP | EP % | Comentario |
|---|---|---|---|---|---|
| 1 - Video30 | 334/601 | [1, 335] | 78480 | 3,78 % | Usuario con muy poca iluminación |
| | 66/601 | [336, 402] | 213312 | 10,29 % | Toma con muy poca luminosidad. Repintado no coherente |
| | 198/601 | [403, 601] | 81700 | 3,94 % | Usuario con muy poca iluminación |
| 2 - Video29 | 0/613 | - | - | - | - |
| 3 - Video28 | 132/468 | [54, 186] | 6417 | 0,31 % | Color común de zapatilla con objeto de fondo |
| 4 - Video26 | 0/626 | - | - | | - |
| 5 - Video23 | 79/617 | [305, 384] | 7371 | 0,36 % | Color común de zapatilla con fondo (lado derecho) |
| | 68/617 | [452, 520] | 7139 | 0,35 % | Color común de zapatilla con fondo (lado izquierdo) |
| 6 - Video22 | 69/609 | [299, 368] | 7081 | 0,34 % | Color común de zapatilla con fondo. |
| 7 - Video21 | 0/620 | - | - | - | - |
| 8 - Video20 | 83/610 | [316,399] | 6160 | 0,30 % | Color común de zapatilla con fondo |

ser más oscuras que el fondo original, como se ve en la Figura 4.10. Esto no influye en el resultado, ya que estas zonas tienen siempre sobrepuesto al usuario segmentado correspondiente a la segunda capa. Por ello, esto no se considera un error.

Es importante destacar que todos los errores presentes en estas pruebas con respecto al repintado se deben a que el algoritmo de segmentación confunde el calzado con algún objeto de fondo. Es crítica esta consideración: implica que mantener al usuario con un fondo regular en esta zona debiese ser prioridad.

### *4.2.4. Evaluación*

Al analizar el desempeño de la aplicación según los resultados de las pruebas y las métricas pertinentes, es posible validar la solución implementada dando respuesta a las preguntas de trabajo mencionadas en la sección 1.3, las cuales se listan a continuación:

- ¿Es viable estimar correctamente *joints* 3D desde un video monocular, sin disponer de un sensor de distancia?.

  Como se demuestra en las pruebas del capítulo 3.2 (Pruebas de algoritmos de BwHT), es viable efectuar la inferencia de la posición en 3D utilizando solo una cámara monocular. La inferencia con algoritmos como FrankMocap o VideoPose3D son lo suficientemente certeras como para tener una base en la cual operar para estos fines. Filtros internos disponibles en Unreal Engine permiten rápidamente suavizar el resultado, pero es evidente que es necesario mejorar el resultado a través de un algoritmo específicamente diseñado para cada método. En nuestro caso, para FrankMocap, es necesario suavizar el efecto de *jitter* y así lograr un resultado más realista.

- ¿Que arquitectura de software es necesaria para transformar las salidas de un algoritmo de *tracking* para que puedan ser interpretados por el motor de videojuegos?.

  No existe un algoritmo de captura de movimiento 3D de cuerpo y manos con la restricción monocular, con salida **fbx** interpretable por un motor de juego. La solución fue adaptar un proyecto de otro contexto para lograr una interpretación parcial de FrankMocap, algoritmo que integra tanto la inferencia de manos como de cuerpo. Esta solución parcial puede potencialmente extrapolarse, dado que en la inferencia de FrankMocap ya se tienen las manos.

  La arquitectura necesaria, ya implementada, implica la ejecución del algoritmo de BwHT (FrankMocap) para luego utilizar un *script* externo basado en la biblioteca de Blender como módulo (**bpy** en Python) e interpretar los huesos, la

malla y la posición de los *joints* de tal manera que UE pueda importarlos sin errores.

- ¿Es lo suficientemente coherente el repintado efectuado sobre la capa de fondo para este contexto?.

  Sí, es lo suficientemente coherente. En la sección 3.5.3 (Métricas) se explicitó un conjunto de métricas que permiten, de manera cuantitativa, determinar las limitaciones del repintado. La conclusión en esta sección es clara: el resultado es certero (solo tendrá un error promedio a nivel de pixeles del 0,25 %) siempre y cuando el fondo no sea demasiado irregular y la iluminación sea correcta.

  Se aprecia en las pruebas una ligera tendencia a segmentar de mala manera las zonas del calzado, lo que causa que no sean repintadas. Aún con ello, no es un error crítico (0,36 %) y los resultados son satisfactorios. En el motor de juego, las zonas repintadas de baja calidad quedan siempre ocultas por la segunda capa. Esto evita que el resultado se vea poco natural.

- ¿Se puede procesar un video grabado de manera vertical de 20 segundos de duración, con una calidad FHD/HD (1920x1080/1280x720) a 30 fps en menos de 60 segundos para obtener la salida esperada sobre el motor de videojuegos?

  Como se muestra en la sección 3.5.3 (Métricas), los resultados de ejecutar toda la etapa de inferencia para el modelo de 3 capas dentro del motor de juego excede con creces los 60 segundos: no sería posible llegar a este tiempo utilizando mejor hardware, si no más bien optimizando e iterando sobre los modelos ya construidos.

  El amplio número de operaciones necesarias para acondicionar las salidas implica un encarecimiento en el tiempo necesario para obtener un resultado legible por el motor de juego. Además, las dependencias de algunas etapas (URem y VIn) no permiten una ejecución en varios hilos de aquellas secciones de la solución, dado que es necesario que sean procesados secuencialmente.

# 5. CONCLUSIÓN Y TRABAJO FUTURO

## *5.1. Conclusiones*

En esta memoria se presentó el desarrollo iterativo sobre la solución de un probador virtual implementado en Unreal Engine 4, compuesto de una estructura de 3 capas: en la primera se ubica un modelo 3D que captura los movimientos en el espacio del usuario inferido a través del algoritmo de BwHT. La segunda capa muestra la captura del algoritmo de URem procesada para obtener un canal *alpha* que permita, en UE, mostrar solo al usuario sin el fondo. La última capa, la tercera, mostrará el fondo repintado obtenido del algoritmo de VIn.

Para lograr este modelo, se siguió una estructura de investigación, selección y desarrollo iterativo.

En primer lugar, la investigación, se basó en una validación preliminar del estado del arte. Esto implica una exploración sobre los 3 tópicos de procesamiento del video: URem, VIn y BwHT. Se encausó la búsqueda hacia algoritmos que permitieran dar respuesta a cada problemática y se compararon a nivel teórico. De esta manera, se logró establecer el conjunto de herramientas que permitieron implementar la solución al problema. Además, se exploraron herramientas y conceptos útiles dentro de Unreal Engine, el motor de juego donde se implementaron todas las soluciones seleccionadas.

A continuación, se realizó una validación preliminar de los métodos del estado del arte. Luego, se escogió el mejor algoritmo en cada tópico que resolviera de mejor manera el problema bajo requisitos de eficacia y eficiencia:

- Para BwHT se eligió FrankMocap. Su buena documentación permitió obtener un primer avance en la exportación de la captura de los movimientos del usuario

hacia Unreal Engine: esto es, transformar su salida en un archivo interpretable en formato **.fbx**. Este desarrollo resolvió el problema esencial a la hora de seleccionar un algoritmo: obtener una salida legible por un motor de juego o un modelador de gráficos tridimensionales (Blender). Otras opciones de inferencia de la pose no incluyen la inferencia de las manos, mientras que FrankMocap la integra.

- Para URem se elige RVM. Es rápido, actual (2021) y certero. Permitió obtener la silueta del usuario correctamente, además de entregar un video de máscaras que ahorra cómputos posteriores. Mostró mayor rendimiento teórico que el resto de los algoritmos.

- Para VIn se eligió DSTT. Su buena documentación además de su API bien implementada permitieron obtener salidas coherentes de manera más rápida que otros algoritmos. Se identificó un problema compartido por todos los algoritmos de VIn: un uso exagerado de memoria de GPU. Se resolvió este problema ejecutando iterativamente el algoritmo sobre una sección de los *frames*, lo que permitió utilizar una parte marginal de la memoria en comparación a realizarlo sobre todos los *frames* al mismo tiempo.

Posteriormente, se implementó la solución basada en las herramientas seleccionadas y métodos escogidos, desarrollada sobre el ecosistema de Unreal Engine. Se adecuaron las salidas de cada algoritmo para que fueran interpretables por este. Esas implementaciones fueron:

- Para el algoritmo de BwHT se requiere la transformación desde su salida en varios archivos **.pkl** hacia un solo archivo **.fbx**.

- Para el algoritmo de URem se requirió transformar el video de salida del usuario segmentado a *frames*, para luego incorporar el canal *alpha* activo, con el fin de que puedan ser visualizados en Unreal Engine.

- Para el algoritmo de VIn se implementó un *script* que trabajase iterativamente sobre las máscaras con filtros aplicados, entregadas por el algoritmo de VIn.

Finalmente, se evaluó el desempeño de la aplicación a través de pruebas basadas en movimientos y situaciones dispares, definiendo pruebas estandarizadas que ayudaran a definir mejoras futuras necesarias, además de evaluar el alcance de la implementación. En general la etapa de repintado y de segmentación de usuario funcionan correctamente: solo existen errores en la segmentación de las extremidades cuando se comparte un color con el fondo. La captura de movimiento es la esperada, y se puede asociar la dinámica de movimiento del usuario a la prenda sobrepuesta al maniquí SMPL.

## 5.2. Trabajo Futuro

Los trabajos futuros contemplan tanto detalles de implementación necesarios para perfeccionar los resultados observados, como también el añadir funcionalidades atractivas relacionadas directamente con el contexto del proyecto. Las propuestas son:

- Implementación de códigos de Python y trasladar a Numpy/OpenCV: existen varias implementaciones en el procesamiento de imágenes, filtros y transformación de videos a *frames* que no están correctamente optimizados. Si se desea llevar al menor tiempo posible el tiempo de inferencia de toda la solución, deben optimizarse las operaciones sobre matrices, siempre que se pueda, utilizando Numpy en toda operación que incluya interacción sobre matrices. Para las operaciones sobre pixeles, por ejemplo, recorrer iterativamente cada pixel con las funciones clásicas de Python cambiando valores es una operación costosa por la naturaleza de Python (e.g. por la sobrecarga de los tipos, corrobora el tipo de dato en todas las comparaciones, algo que no es necesario en este contexto). Usar Numpy para recorrer los pixeles como valores o utilizar funciones específicas optimizadas de OpenCV puede disminuir notablemente el tiempo de demora en el recorrido, comparación y escritura.

- Filtros sobre *joints* en la detección de FrankMocap: mejorar la inferencia en bruto de FrankMocap es esencial para lograr una respuesta visual más suave, que muestre mayor naturalidad en el motor de juego, y así se exhiban movimientos sin *jitter* en la ropa simulada. La implementación de un filtro debe ser estudiada con detenimiento, ya que la representación de los *joints* de FrankMocap no es estándar al formato SMPL.

- Mejora del algoritmo de VIn: el repintado actualmente se está implementado en toda la silueta del usuario. Esto implica que el cómputo se lleva a cabo en zonas que nunca se verán. Para mejorar el uso de recursos y el tiempo de respuesta, es necesario realizar una máscara con menor área, que no incluya las zonas que nunca serán visibles en el video: esto es, las zonas del fondo cubiertas por el usuario, que no puedan ser vistas.

- Mejora de funcionamiento en DSTT: al entregar una ventana de *frames* más amplia: el algoritmo de repintado se basa, esencialmente, en la información entregada para realizar el repintado en las zonas indicadas. Esto implica que, si tenemos 100 *frames* a repintar, los extremos de este *pool* de imágenes tendrán menor información para que el repintado sea coherente (el *frame* 1 y 100 en este ejemplo). Para mejorar la calidad del repintado en ambos extremos, debe ampliarse la ventana donde se realiza el repintado siempre que sea posible.

# Bibliografía


[1] M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele. 2d human pose estimation: New benchmark and state of the art analysis. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3686–3693, 2014.

[2] Valentin Bazarevsky, Ivan Grishchenko, Karthik Raveendran, Tyler Zhu, Fan Zhang, and Matthias Grundmann. Blazepose: On-device real-time body pose tracking, 2020.

[3] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields, 2019.

[4] Tianlang Chen, Chen Fang, Xiaohui Shen, Yiheng Zhu, Zhili Chen, and Jiebo Luo. Anatomy-aware 3d human pose estimation with bone-based pose decomposition, 2021.

[5] Yilun Chen, Zhicheng Wang, Yuxiang Peng, Zhiqiang Zhang, Gang Yu, and Jian Sun. Cascaded pyramid network for multi-person pose estimation, 2018.

[6] Yu Cheng, Bo Wang, Bo Yang, and Robby T. Tan. Monocular 3d multi-person pose estimation by integrating top-down and bottom-up networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7649–7659, June 2021.

[7] Yu Cheng, Bo Yang, B. Wang, Yan Wending, and R. Tan. Occlusion-aware networks for 3d human pose estimation in video. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 723–732, 2019.

[8] Junting Dong, Qi Fang, Wen Jiang, Yurou Yang, Hujun Bao, and Xiaowei Zhou. Fast and robust multi-person 3d pose estimation and tracking from multiple views. In *T-PAMI*, 2021.

[9] Junting Dong, Qing Shuai, Yuanqing Zhang, Xian Liu, Xiaowei Zhou, and Hujun Bao. Motion capture from internet videos, 2020.

[10] Hao-Shu Fang, Shuqin Xie, Yu-Wing Tai, and Cewu Lu. RMPE: Regional multi-person pose estimation. In *ICCV*, 2017.

[11] Qi Fang, Qing Shuai, Junting Dong, Hujun Bao, and Xiaowei Zhou. Reconstructing 3d human pose by watching humans in the mirror. In *CVPR*, 2021.

[12] Chen Gao, Ayush Saraf, Jia-Bin Huang, and Johannes Kopf. Flow-edge guided video completion. In *European Conference on Computer Vision*, 2020.

[13] Yihui He, Rui Yan, Katerina Fragkiadaki, and Shoou-I Yu. Epipolar transformers, 2020.

[14] Catalin Ionescu, Dragos Papava, Vlad Olaru, and C. Sminchisescu. Human3.6m: Large scale datasets and predictive methods for 3d human sensing in natural environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36:1325–1339, 2014.

[15] Zhanghan Ke, Kaican Li, Yurou Zhou, Qiuhua Wu, Xiangyu Mao, Qiong Yan, and Rynson W.H. Lau. Is a green screen really necessary for real-time portrait matting? *ArXiv*, abs/2011.11961, 2020.

[16] Nikos Kolotouros, Georgios Pavlakos, Michael J Black, and Kostas Daniilidis. Learning to reconstruct 3d human pose and shape via model-fitting in the loop. In *ICCV*, 2019.

[17] Shichao Li, Lei Ke, Kevin Pratama, Yu-Wing Tai, Chi-Keung Tang, and Kwang-Ting Cheng. Cascaded deep monocular 3d human pose estimation with evolutio-

nary training data. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2020.

[18] Shanchuan Lin, Andrey Ryabtsev, Soumyadip Sengupta, Brian Curless, Steve Seitz, and Ira Kemelmacher-Shlizerman. Real-time high-resolution background matting, 2020.

[19] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[20] Rui Liu, Hanming Deng, Yangyi Huang, Xiaoyu Shi, Lewei Lu, Wenxiu Sun, Xiaogang Wang, and Li Hongsheng. Decoupled spatial-temporal transformer for video inpainting. *arXiv preprint arXiv:2104.06637*, 2021.

[21] Ruixu Liu, Ju Shen, He Wang, Chen Chen, Sen-ching Cheung, and Vijayan Asari. Attention mechanism exploits temporal contexts: Real-time 3d human pose reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5064–5073, 2020.

[22] Wu Liu, Qian Bao, Yu Sun, and Tao Mei. Recent advances in monocular 2d and 3d human pose estimation: A deep learning perspective, 2021.

[23] Erika Lu, Forrester Cole, Tali Dekel, Andrew Zisserman, William T Freeman, and Michael Rubinstein. Omnimatte: Associating objects and their effects in video. In *CVPR*, 2021.

[24] Zhengyi Luo, S. Alireza Golestaneh, and Kris M. Kitani. 3d human motion estimation via motion compression and refinement, 2020.

[25] Dushyant Mehta, Oleksandr Sotnychenko, Franziska Mueller, Weipeng Xu, Mohamed Elgharib, Pascal Fua, Hans-Peter Seidel, Helge Rhodin, Gerard Pons-Moll, and Christian Theobalt. XNect: Real-time multi-person 3D motion capture with a single RGB camera. volume 39, 2020.

[26] Dario Pavllo, Christoph Feichtenhofer, David Grangier, and Michael Auli. 3d human pose estimation in video with temporal convolutions and semi-supervised training. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[27] Yu Rong, Takaaki Shiratori, and Hanbyul Joo. Frankmocap: Fast monocular 3d hand and body motion capture by regression and integration, 2020.

[28] Mingyi Shi, Kfir Aberman, Andreas Aristidou, Taku Komura, Dani Lischinski, Daniel Cohen-Or, and Baoquan Chen. Motionet: 3d human motion reconstruction from monocular video with skeleton consistency, 2020.

[29] Yanan Sun, Chi-Keung Tang, and Yu-Wing Tai. Semantic image matting, 2021.

[30] Yanan Sun, Guanzhi Wang, Qiao Gu, Chi-Keung Tang, and Yu-Wing Tai. Deep video matting via spatio-temporal alignment and aggregation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2021.

[31] Timo von Marcard, Roberto Henschel, Michael J. Black, Bodo Rosenhahn, and Gerard Pons-Moll. Recovering accurate 3D human pose in the wild using IMUs and a moving camera. In *European Conference on Computer Vision (ECCV)*, volume Lecture Notes in Computer Science, vol 11214, pages 614–631. Springer, Cham, September 2018.

[32] Bastian Wandt, Marco Rudolph, Petrissa Zell, Helge Rhodin, and Bodo Rosenhahn. Canonpose: Self-supervised monocular 3d human pose estimation in the wild. In *Computer Vision and Pattern Recognition (CVPR)*, June 2021.

[33] Min Wang, Feng Qiu, Wentao Liu, Chen Qian, Xiaowei Zhou, and Lizhuang Ma. Ellipbody: A light-weight and part-based representation for human pose and shape recovery. *ArXiv*, abs/2003.10873, 2020.

[34] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[35] Donglai Xiang, Hanbyul Joo, and Yaser Sheikh. Monocular total capture: Posing face, body, and hands in the wild, 2018.

[36] Chenxin Xu, Siheng Chen, Maosen Li, and Ya Zhang. Invariant teacher and equivariant student for unsupervised 3d human pose estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3013–3021, 2021.

[37] Rui Xu, Xiaoxiao Li, Bolei Zhou, and Chen Change Loy. Deep flow-guided video inpainting. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[38] Yanhong Zeng, Jianlong Fu, and Hongyang Chao. Learning joint spatial-temporal transformations for video inpainting. In *The Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[39] Yunke Zhang, Chi Wang, Miaomiao Cui, Peiran Ren, Xuansong Xie, Xian sheng Hua, Hujun Bao, Qixing Huang, and Weiwei Xu. Attention-guided temporal coherent video object matting, 2021.

[40] Ce Zheng, Sijie Zhu, Matias Mendieta, Taojiannan Yang, Chen Chen, and Zhengming Ding. 3d human pose estimation with spatial and temporal transformers. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2021.

[41] Yuxiao Zhou, Marc Habermann, Weipeng Xu, Ikhsanul Habibie, Christian Theobalt, and Feng Xu. Monocular real-time hand shape and motion capture using multi-modal data. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[42] Xueyan Zou, Linjie Yang, Ding Liu, and Yong Jae Lee. Progressive temporal feature alignment network for video inpainting. In *CVPR*, 2021.