Escuela
Politécnica
Superior

Animación Procedimental: Dando vida a través del código.

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Laura Hernández Rielo
Tutor:
Faraón Llorens Largo

Septiembre 2019

Universitat d'Alacant
Universidad de Alicante

Animación Procedimental

Dando vida a través del código

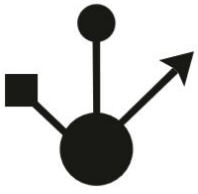
Autora

Laura Hernández Rielo

Tutor/es

Faraón Llorens Largo

Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Septiembre 2019

Resumen

Este documento, titulado "*Animación procedimental: Dando vida a través de código"* corresponde al Trabajo de Fin de Grado de Laura Hernández Rielo en el grado de Ingeniería Multimedia, realizado durante el curso 2018-2019 y tutorizado por Faraón Llorens Largo.

En este trabajo se introduce el concepto de animación procedimental y su uso en videojuegos, además de explicarse las distintas técnicas de desarrollo y herramientas que permiten ponerlas en práctica. Por otro lado, se documenta el desarrollo de un sistema básico de animación procedimental usando la técnica de la Cinemática Inversa para su uso en el motor gráfico *Unity*, además de justificar el uso de dicho motor y técnica. Esto último pretende ser un método de aprendizaje tanto de los principios de la animación procedimental, de la técnica de cinemática inversa y del desarrollo de sistemas para *Unity*.

A mi familia, por enseñarme el camino del arte y la ciencia,
por apoyar mis locuras y por acompañarme en el camino.
A mis amigos, que han creído en mí y de los que he aprendido casi todo lo que sé.
A mis compañeros y familia de Wasted Horchata,
por enseñarme y tirar de mí en los peores momentos,
A Elahi, por estar siempre ahí.

"We understand so much,

but the sky behind those lights ─mostly void, partially stars─,

that sky reminds us we don't understand even more."

Welcome To Night Vale

"The world always seems brighter
when you've just made something that wasn't there before."
Neil Gaiman

Índice

1. Introducción

Actualmente estamos viviendo una edad dorada en la animación. Las técnicas están avanzando cada año a pasos agigantados y nos ofrecen un amplio abanico de obras que disfrutar, cada cual contando con un despliegue técnico más increíble que la anterior. Dentro de esta evolución constante destacan dos corrientes: la búsqueda del máximo realismo, de replicar el mundo tal cual lo vemos de la manera más fiel posible, y el camino de la originalidad, cuyo objetivo es crear lenguajes nuevos e innovar a nivel artístico.

Y es esta evolución constante a nivel técnico-gráfico la que está creando un problema cada vez más grande en la industria del videojuego y su avance se está viendo ralentizado respecto al de la industria cinematográfica. El avance gráfico también supone una mayor exigencia de recursos y, dada la naturaleza interactiva de los videojuegos, los números se disparan y se crea un cuello de botella en el desarrollo. Además, por esta misma naturaleza interactiva, es más complicado conseguir realismo y comportamientos naturales en las animaciones. Lo que en la industria cinematográfica es un proceso exacto y cerrado, con un número limitado de recursos que realizar y unas animaciones que siempre se comportarán igual, pues siempre veremos las mismas escenas de la manera en la que el director así desea, en un videojuego parte de la experiencia está en lo impredecible que puede llegar a ser o en la gran variedad de acciones e interacciones que se producen. De esta manera, nos encontramos con que cada vez se necesita dedicar más tiempo a crear una cantidad cada vez más grande de animaciones y además se busca que estas sean naturales y realistas.

Es en este punto donde entra en juego la animación procedimental. ¿Y si pudiéramos recortar el tiempo de producción de animaciones de manera más óptima y además permitir un gran número de variedad y combinación entre estas? La animación procedimental se lleva utilizando durante años en videojuegos, pero especialmente centralizada en un número reducido de elementos. Son escasos los videojuegos que hayan explorado el uso de esta técnica en personajes y la gran mayoría de este reducido número son apuestas de estudios independientes.

En este trabajo, por tanto, se pretende hablar sobre las distintas técnicas de animación procedimental y sus utilidades para ayudar en el desarrollo de videojuegos, industria todavía dominada por el uso de las técnicas tradicionales.

2. Objetivos

El objetivo principal de este proyecto, más allá de concluir con el desarrollo de una herramienta, es entender las bases de la animación procedimental y su uso en la industria. Además de no tener un uso especialmente extendido, esta técnica es muy poco conocida fuera del sector tanto de los videojuegos como cinematográfico, así que el primer paso será definir el concepto de Animación Procedimental y las diferentes técnicas que lo componen. Para esto habrá que explicar los conceptos básicos de la animación, el uso de animación en videojuegos, los estándares y los problemas que existen actualmente.

Entrando en un marco práctico, el objetivo del desarrollo es entender los conceptos y principios básicos de la cinemática inversa. Ésta es una de las técnicas de animación procedimental más destacables a la hora de trabajar con personajes y criaturas, elementos en los videojuegos que menos utilizan dicho tipo de animación y que más se podrían beneficiar de sus ventajas. Dado que el escaso uso de la animación procedimental en personajes y criaturas está generalizado en la industria y supondría un gran cambio, va a ser el foco del desarrollo de este proyecto. Para ello se busca elaborar un sistema de cinemática inversa básico para el motor gráfico *Unity* enfocado a la animación procedimental de enemigos.

Este punto anterior va de la mano del tercer objetivo, que es profundizar en el conocimiento sobre el sistema de componentes y elaboración de *scripts* en *C#* del motor gráfico *Unity*, a la vez que sobre el concepto de *prefabs,* un tipo de recurso característico de este motor gráfico que permite almacenar una serie de recursos con todas sus propiedades en el interior, actuando como una plantilla para crear nuevas instancias del mismo objeto en una escena, con el objetivo de realizar un sistema reutilizable en distintos proyectos.

En conclusión, los objetivos se resumen en la siguiente lista de puntos:

1. Definir y entender los conceptos de la animación procedimental y sus técnicas.

2. Entender los principios básicos de la cinemática inversa.
3. Aprender sobre el sistema de *prefabs*, *scripts* en *C#* y componentes de *Unity*.
4. Desarrollar un sistema de *Unity* de cinemática inversa básica para conseguir la animación de enemigos en videojuegos.

3. Animación Procedimental

3.1. Animación

3.1.1 Definición

A pesar de que la espina dorsal de este estudio sea el uso de animación en videojuegos, cabe añadir previamente una breve introducción al mundo de la animación, remontándonos a su origen. En pocas palabras, la definición más precisa de animación es *"*imágenes en movimiento*"*. Si nos remontamos al origen del término, procedente del latín, *animatio* significa revivir o "*otorgar alma"*, así que podríamos categorizar este arte como aquel que da vida a las cosas sin vida, como son las imágenes. Este arte lleva entre nosotros desde décadas antes que el invento de la fotografía y el cine, y ha sido este último, hasta años recientes, el campo que más ha bebido de la animación, hasta tal punto que en ambientes generalizados, el término animación se usa como sinónimo de "cine de animación".

Ilustración 1: Blancanieves y los Siete Enanitos (David Hand, 1937)

Pero la animación, a día de hoy, va mucho más allá de los dibujos en movimiento que iniciaron este arte (ilustración 1) y se han formado tres ramas diferenciadas. El método más simple y antiguo es la *hand-drawn animation (*también conocida como animación clásica), que consiste en dibujar a mano todos los fotogramas de la secuencia. Los principios de este método siguen siendo los mismos, pero las herramientas han cambiado gracias a las nuevas tecnologías de dibujo digital. El segundo método es casi tan antiguo como el primero y originalmente fue utilizado para la creación de los primeros efectos especiales del cine. Éste es el *stop-motion*, técnica similar a la anterior (ilustración 2), ya que se trabaja fotograma a fotograma, pero capturando éstos en fotografía y manipulando los objetos fotografiados, en vez de usar dibujos a mano.

Ilustración 2: Vincent (Tim Burton, 1982)

Por último, la animación generada por computador, es la técnica que recientemente está avanzando a mayor rapidez y se ha convertido en la más popular y usada en la industria cinematográfica hoy en día. Gracias a las nuevas tecnologías, se consigue crear imágenes que ninguna de las dos técnicas anteriores podría, de manera más eficaz. Su propio nombre la define y, a pesar de que tanto imágenes 2D como 3D pueden ser generadas por computador, ese término se utiliza en su totalidad a la animación de imágenes en 3 dimensiones (ilustración 3, ilustración 4).

Ilustración 3: Toy Story (John Lasseter, 1995)

Pero, a pesar de considerarse el arte de la animación un término perteneciente a la industria cinematográfica, su avance ha ido prácticamente mano a mano con la evolución de los videojuegos ya que, a menudo obviado, la animación es un pilar fundamental de éstos.

Ilustración 4: Kubo y Las Dos Cuerdas Mágicas (Travis King, 2016)

3.1.2. Animación en videojuegos

A pesar de que, a simple vista, animación en cine y videojuegos se perciba igual, distan más de lo que parece una de la otra. En la primera solamente se ha de trabajar en la escena que se ve dentro del plano, permitiendo tomar atajos y "hacer trampas" para ahorrar trabajo, ya que la imagen en pantalla solamente se ve desde un punto de vista inamovible y la audiencia no tiene el control de lo que ocurre. Por el contrario, los videojuegos están hechos para ser interactivos, y esto significa un amplio abanico de puntos que tener en cuenta en el trabajo de un animador. Ya no solo en cuestión de la cantidad de puntos de vista desde los que se va a ver esa animación en pantalla, sino también en cuestión de las interacciones que ocurrirán debido a las diversas acciones que pueda llevar a cabo el jugador.

3.1.2.1. Evolución

En los videojuegos, técnica y arte una vez más han ido de la mano avanzando a lo largo de los años. A pesar de que la industria cinematográfica llevaba décadas trabajando en animación en los años 70, en los videojuegos solamente se podían ver animaciones muy primitivas (ilustración 5).

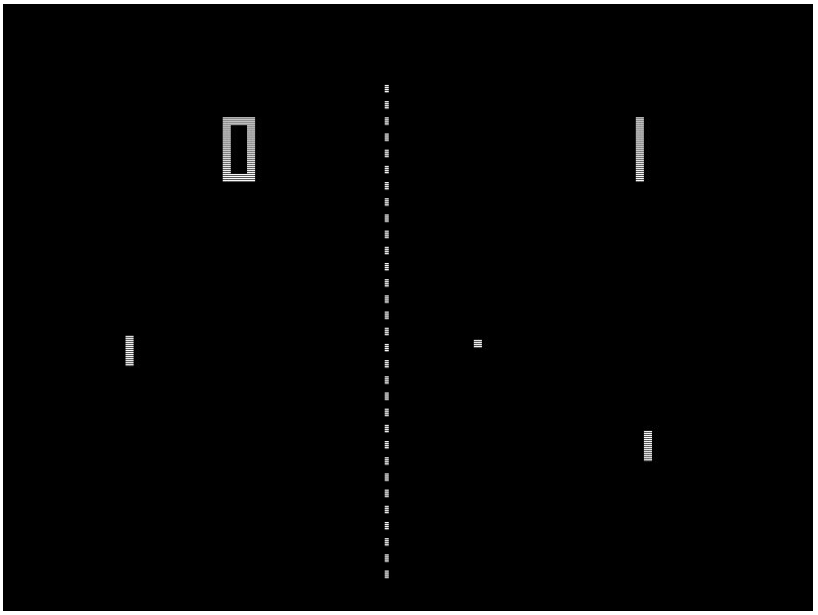


Ilustración 5: Pong (1972)

A medida que los gráficos avanzan en cada nueva generación de videojuegos, también deben hacerlo las técnicas de animación usadas en estos. Hoy en día la animación en videojuegos han alcanzado a la usada en industria cinematográfica, convirtiendo los primeros en películas interactivas, pero en sus inicios la distancia entre ambas disciplinas empezó siendo mucho mayor.

Durante las dos primeras décadas de vida de los videojuegos como los conocemos hoy en día únicamente fueron usados gráficos en dos dimensiones, avanzando en complejidad a medida que el hardware evolucionaba. En los 70 las animaciones consistían simplemente en objetos estáticos desplazándose por la pantalla, desde un puñado de píxeles blancos en un fondo negro, como podemos ver en el *Pong* de 1972. Pero el empujón a nivel gráfico empezó a partir de los años 80, con la salida del *Nintendo Entertaining System*, que permitió al fin que el desarrollo de los juegos pudiese adoptar una identidad propia a nivel estético. Así comenzó la década de los arcades y de un estándar de animación más cercano a las técnicas cinematográficas.

El estándar de integración de gráficos en videojuegos fue a través de *sprites*, mapas de bits dibujados en pantalla, de manera que los artistas creaban imágenes 2D que representaban objetos, personajes y entorno. De esta manera se conseguían introducir animaciones de éstos mismos de la misma manera que se animaba en cine: fotograma a fotograma, usando un dibujo para cada paso del movimiento o, en este caso, sprite a sprite. Estas animaciones consistían en un escaso número de poses generalmente dedicadas al movimiento del personaje. Fue a partir de 1987, con la salida de *The Legend of Zelda,* que contaba también con novedosas animaciones de ataque, que el avance en complejidad a nivel de animaciones a través de *sprites* creció exponencialmente. Hoy en día esta técnica ya no es tan popular, si bien es muy común el uso de *sprites* para introducir otro tipo de elementos gráficos como botones o mensajes en las interfaces. También empezaron a aparecer los casos en los que se introducían fotogramas digitalizados de los actores para realizar las animaciones, como se puede ver en la ilustración 6.

Ilustración 6: Mortal Kombat (1992)

A partir de los años 90, el *boom* de los gráficos en tres dimensiones comenzó y esto significó que se debía hacer a su vez un gran avance en las técnicas de animación (ilustración 7). Ya no bastaba con usar una pose estática para simular un salto, los personajes ahora daban volteretas, movían sus extremidades de distinta manera en el aire, seguían las leyes de la física en sus movimientos. A medida que los videojuegos se expandían y permitían a los jugadores realizar cada vez más acciones, una animación de caminar y otra de correr ya no eran suficientes. Desde entonces, el 3D se ha convertido en el formato gráfico más famoso a la hora de desarrollar videojuegos.

Hoy en día, las animaciones cada vez nos recuerdan más a la vida real, haciendo que los personajes y entornos parezcan vivos y permitiendo una mayor inmersión en los mundos creados. Especialmente, la animación de personajes se ha convertido en uno de los componentes vitales de los videojuegos. La tendencia que está siguiendo la mayor parte de la industria es asemejarse al máximo al mundo que vemos a nuestro alrededor, a la imagen real y, a pesar de que hay otras corrientes menores que se centran en explorar distintas expresiones artísticas, esta primera ha ido de la mano de técnicas de animación cada vez más modernas, adelantándose en ocasiones a las técnicas de animación cinematográfica por primera vez.

Ilustración 7: Super Mario 64 (Nintendo, 1996), izquierda. Super Mario Odyssey (Nintendo, 2017), derecha.

3.1.2.2. Estándares

En los últimos años de desarrollo de la industria de los videojuegos, si bien los gráficos y las técnicas no han dejado de avanzar, destacan unos estándares básicos a la hora de animar. Antes de nada, cabe distinguir la diferencia entre animación cinemática y animación *in game*, consistiendo la primera en animación de secuencias en formato cinematográfico (escenas desde un punto de vista que no cambia, sin interacción del jugador) y usada con fines únicamente narrativos, y la segunda siendo todas las animaciones de personajes y entorno que se dan al producirse interacciones. En estas últimas existen más restricciones, ya que quien controlará las animaciones será el propio jugador, así que se suma la dificultad de que las transiciones entre ellas sean lo más fluidas y creíbles posibles.

Si hablamos de animaciones cinemáticas, en los últimos años destaca especialmente la *mocap* o captura de movimiento a la hora de crear dichas secuencias. En cine y videojuegos, este proceso consiste en grabar el movimiento de actores y trasladarlo a los modelos creados digitalmente, pero en otras industrias la información capturada se utiliza para diferentes tipos de análisis y recopilación de datos. Las técnicas de captura de movimiento varían según la industria pero la usada en videojuegos y cine es la más

popular: una serie de marcadores son colocados en cuerpo y rostro de los actores, cerca de las distintas articulaciones y, una vez filmadas las secuencias, los sistemas de captura interpretan el movimiento según el ángulo y distancia de dichos marcadores para recrear las acciones de los actores. En la ilustración 8 podemos ver al futbolista estadounidense Tim Howard participó en la captura de movimiento de los jugadores de la famosa franquicia de videojuegos de fútbol *Fifa*.

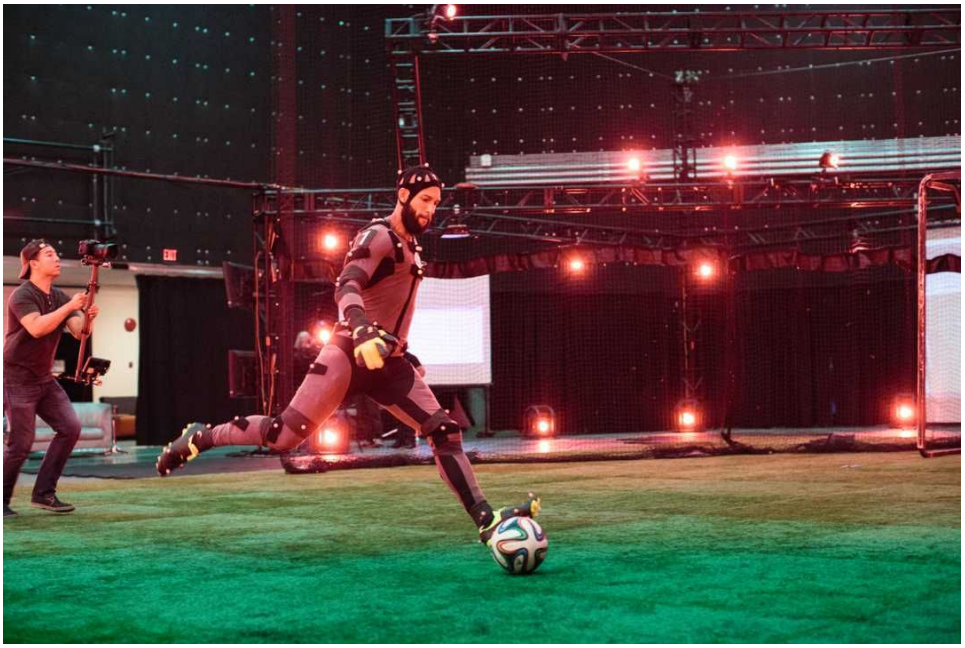


Ilustración 8: Fifa 15 (Electronic Arts, 2014)

La captura de movimientos faciales se suele separar de la *mocap* general, ya que en un principio solamente se buscaba una manera de generar movimientos humanos realistas, pero poco a poco se le ha ido dando más importancia a la interpretación y se han contratado actores profesionales que no solamente prestan sus movimientos, sino también sus caras a los personajes a los que dan vida.

Ilustración 9: Face Motion Capture for Last of Us 2

La animación facial es mucho más costosa debido a la mayor resolución necesaria para capturar hasta las expresiones más sutiles que pueden ser cambios milimétricos. En la ilustración 9 se puede ver a la actriz Ashley Johnson da vida al personaje de Ellie en *The Last Of Us 2* prestando su voz y sus movimientos, pero no sus rasgos, que provienen de un modelo previamente diseñado. El software permite un nivel de detalle muy amplio que captura hasta el movimiento ocular.

Pero la captura de movimiento no es de exclusivo uso en la animación de cinemáticas, ya que para animaciones dentro del juego también es muy utilizada, si bien no es el estándar ya que es un proceso caro (en costes de software, hardware y el personal necesario) y los movimientos que no siguen las leyes de la física no pueden ser simulados. También, debido a los costes, no es la única técnica usada en animaciones cinemáticas y muchos estudios siguen confiando en la animación tradicional para ello.

En cuanto a animaciones *in game*, o aquellas que suceden en el entorno de interacción del videojuego y son causadas por las distintas acciones que el jugador pueda realizar con el personaje o por las interacciones de los elementos y seres vivos entre ellos y con el entorno. Todo esto sucede en tiempo de juego y no ha sido producido previamente, sino que las animaciones “detonan” en el momento. Las técnicas son múltiples para cada

tipo de animación: personajes, entorno, objetos… Pero los estándares, a pesar de haber mejorado con el tiempo, no han variado mucho en los últimos años. Esto quiere decir que, en general, se sigue animando de la misma manera, aunque con tecnologías cada vez mejores.

Si nos centramos en la sección más amplia de la animación 3D en videojuegos, que es la animación de personajes, se conoce como "animación tradicional" el proceso en el que, un artista animador, trabaja sobre un modelo en 3 dimensiones para colocarlo en una serie de poses que formarán la animación completa. Los modelos 3D de los personajes llegan a los animadores una vez se ha hecho el *rigging*, es decir, una vez se coloca un esqueleto móvil al modelo. De esta manera, los animadores pueden mover y posar el modelo sin peligro de que este se deforme, y dicho esqueleto interior puede tener varios niveles de detalle. En general, los artistas animadores crean a mano prácticamente todas las animaciones de los personajes (salvo excepciones como las animaciones de muertes, que cuentan con su propio estándar), moviendo los modelos 3D gracias a los esqueletos. Hace años el proceso era más similar al de animación 2D: colocar cada pose para cada fotograma hasta crear la animación deseada. Pero hoy en día, los distintos software con los que se anima permiten simplificar la tarea gracias a la interpolación. En pocas palabras, la interpolación dentro del contexto de la animación 3D es el proceso del cálculo de un número de fotogramas comprendido entre dos fotogramas clave. Así, los artistas solamente colocan el personaje en las poses clave para el movimiento, llamadas *keyframes (*de aquí el término *keyframming* con el que se conoce también a la animación tradicional), y el programa calcula el movimiento entre pose y pose, creando nuevos fotogramas que sigan ese movimiento. De esta manera, el animador tiene control directo sobre las posiciones, formas y movimientos de los modelos en todo momento de la animación con mayor facilidad.

A pesar de haber multitud de programas específicos de modelado y animación 3D (ilustración 10*)*, además de motores gráficos con sistemas de animación incorporados como *Unity* o *Unreal Engine,* el software estandarizado en la industria es *Autodesk Maya.* Como la mayoría de programas más utilizados, sus herramientas están enfocadas tanto

como para el modelado y texturizado como para el *rigging* y animación de estos, permitiendo así el proceso entero de creación y animación de cualquier personaje sin tener que cambiar de programas para cada fase. Hoy en día, *Autodesk Maya* es el número uno en la industria no por sus herramientas de modelado, sino por ser considerado el mejor a la hora de animar tanto de manera manual como de aplicar la captura de movimiento, vista anteriormente, gracias a su proceso intuitivo y su sistema de capas de modelado.

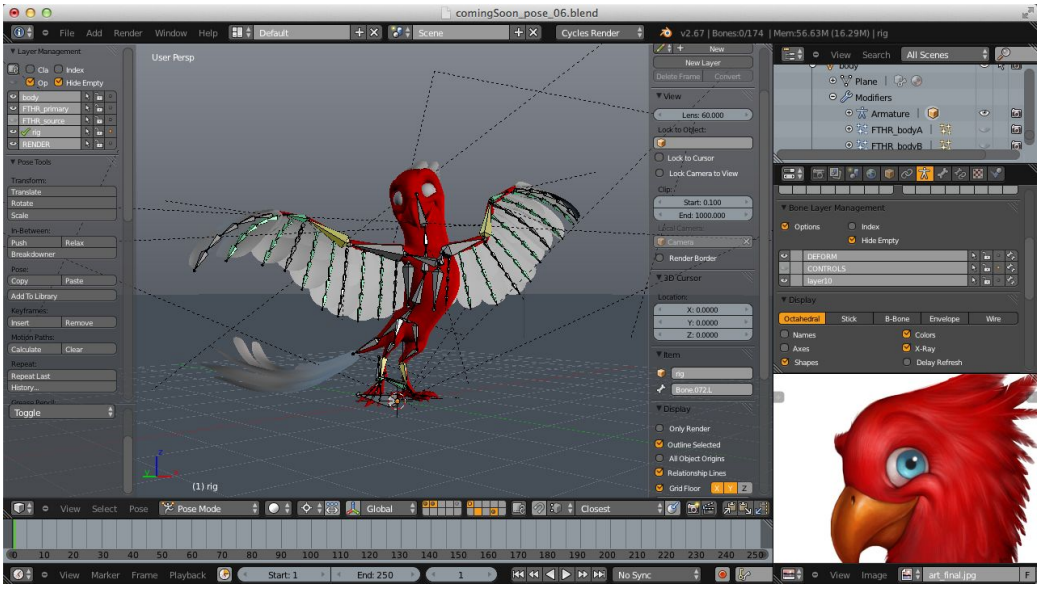


Ilustración 10: Animación en Blender (Ken Tammell)

Al contrario que en la industria cinematográfica, en vez de crear la animación entera de una escena, los artistas separan cada animación de los distintos movimientos, haciendo hincapié especialmente entre que el principio y final de cada animación se pueda encadenar con el resto de estas para crear un movimiento de personaje lo más fluido y realista posible, a pesar de que las acciones de éste estén siendo controladas por un jugador.

Hemos hablado casi en su totalidad de las técnicas de animación estándar empleadas en videojuegos únicamente 3D, ya que en las dos dimensiones los avances no han sido tan grandes, si bien no ha sido necesario. Tanto en animación de cinemáticas como en animaciones *in game*, se sigue utilizando la animación fotograma a fotograma y

es el actual estándar, aunque los nuevos motores gráficos también han supuesto un avance al permitir la interpolación de imágenes 2D, que hacen más sencilla y rápida la tarea.

Con esto vemos que, a pesar de existir técnicas muy diversas de animación tanto 3D como 2D en la industria del videojuego, en general se anima todo "a mano", a través de artistas que se toman su tiempo en mover aquello que, valga la redundancia, tendrá movimiento.

3.1.2.3. Problemas

Pero, vistas estas técnicas, nos encontramos una serie de problemas que entorpecen el desarrollo, sea tanto a nivel económico como a nivel de tiempo de producción. Y es que, dado que a medida que los gráficos avanzan, los videojuegos demandan cada vez una mayor perfección en el campo de la animación (más realismo, más fluidez…), pero las técnicas estándar hoy en día están generando un "cuello de botella" a la hora de mejorar el movimiento de los personajes.

Como ya hemos visto, la captura de movimiento es la técnica más usada a la hora de animar cinemáticas, así como es utilizada también en las animaciones *in game.* El problema más destacable de dicha técnica es, sin duda, el alto presupuesto que requiere (no solo en personal o hardware, sino también en instalaciones, algo que se suele obviar al hablar de este tema), lejos de ser viable para estudios independientes. Pero el principal problema del *mocap* son sus limitaciones: por efectivo que sea a la hora de animar humanoides y otorgar vida y personalidad a las animaciones de infinidad de personajes, es poco útil en cualquier otro tipo de animaciones y está sujeto a las leyes de la física humana. Aquí se plantea la cuestión de si merece la pena usar esta técnica en proyectos cuya mayor carga de animación no sean personajes humanos realizando acciones realistas.

A pesar de estos inconvenientes, la roca en el camino que supone trabajar con *mocap* es el coste a nivel de tiempo que supone, no solamente a nivel de desarrollo de proyecto, sino a nivel de tiempo de juego. Las técnicas usadas para transicionar entre animaciones dentro del mismo juego requieren una máquina de estados de animaciones

dentro del juego y, cuando se trata de proyectos muy grandes, se convierte en un proceso muy lento y complicado. Actualmente, en el campo de la captura de movimiento, se está trabajando en la optimización y solución de estos problemas. Además, el procesado y limpiado de datos es lento, pues siempre se necesita un postprocesado tras las capturas, no consiste simplemente en que un actor se ponga el traje y se empiece a grabar, sino que también lleva un pulido que supone un trabajo de días.

De esta manera, a pesar de dar grandes resultados en cinemáticas, a la hora de su uso regular en animaciones dentro del juego supone demasiadas desventajas.

En cuanto a la animación tradicional o, como hemos visto, *keyframming*, hemos visto que los artistas animadores conforman cada pose de manera manual, pero cuando los objetos a animar empiezan a ser significativamente más complejos, el uso de dichas técnicas se vuelve lento y complicado. Al ser la técnica más empleada, sí es cierto que ha sido perfeccionada a lo largo de los años y lleva una gran evolución, pero sigue conllevando un gran problema que no ha cambiado: el sacrificio de la naturalidad en favor del tiempo ahorrado. Los estudios han de elegir entre gastar más tiempo a la hora de enfrentarse a las animaciones en pos de conseguir mayor realismo y naturalidad en los movimientos, o elegir optimizar tiempo de desarrollo y mano de obra para llegar a un resultado menos pulido y creíble. Obviamente, según los principios y objetivos de los directores, el ratio de cada una de las elecciones varía y se dedica un tiempo u otro según si es considerado o no, pero eso no quita que el problema esté ahí y que, en general, se decida dejar de lado el conseguir una animación fluida y creíble a favor de terminar cuanto antes el proceso. Esto no quita que estudios independientes hayan empezado a experimentar y aplicar otras técnicas, pero en la parte más comercial de la industria, el objetivo es vender y acabar cuanto antes, por lo que se establece otra prioridad antes del mejor resultado posible de las animaciones.

Pero esto no significa que no haya más técnicas que no estén siendo usadas o sin descubrir, ya que estamos hablando de una mayoría generalizada pero no total. Esto nos lleva al siguiente grupo de técnicas de animación menos usadas que podrían suponer una

solución a dichos problemas comentados que están provocando un estancamiento del desarrollo de la animación en videojuegos y de sus resultados.

3.2. Animación Procedimental

3.1.1. Definición

Ya hemos visto en qué consiste el término "animación" y cómo se aplica a la industria del videojuego, además de exponer las técnicas estándar utilizadas y los problemas que presentan, pero es importante añadir unas breves definiciones antes de dar el salto al núcleo de este estudio.

3.2.1.1. Generación procedimental de contenido

En el campo del desarrollo de videojuegos, cuando hablamos de generación procedimental de contenido, o PGC (*procedural content generation)*, nos referimos a la generación de contenido mediante algoritmos con participación humana nula o limitada. Este contenido son mapas, niveles, texturas, personajes, dinámicas de juego, animaciones… Pero no el propio motor del juego ni el comportamiento de personajes no jugables.

Ilustración 11: Elite: Dangerous (2014)

El objetivo principal del uso de PGC es proporcionar variedad en el contenido, reducir el tiempo de desarrollo y los costes del mismo, además de otras ventajas como ahorrar espacio en el disco o permitiendo la adaptabilidad en los juegos. En la ilustración 11 se puede ver uno de los mayores ejemplos de contenido generado procedimentalmente, con la representación de cientos de sistemas estelares creados para la exploración de los jugadores.

Actualmente, los estudios sobre el contenido generado procedimentalmente se inclinan más hacia el proceso de creación de niveles, mapas y mecánicas. Esto se debe a la búsqueda de una serie de objetivos como es el crear un juego enteramente a partir de algoritmos o llevar el *game design* hacia un camino en el que se guíe completamente por este tipo de contenido. El objetivo es dejar de usar este contenido creado mediante algoritmos para juegos ya existentes, donde el corazón del juego podría sobrevivir perfectamente sin él, y empezar a enfocar el diseño de videojuegos hacia el completo uso de algoritmos para prácticamente todos sus componentes.

Por otro lado, uno de los grandes problemas del uso del PGC es que, normalmente, el contenido parece genérico y poco original. En cualquier videojuego *roguelike* donde se utilice este contenido, como la saga *Diablo*, cuyas mazmorras se generan procedimentalmente, no parecen más que un puñado de bloques colocados de manera aleatoria y sin mucha personalidad. Más allá de la disposición y aspecto que puedan tener, normalmente los niveles generados procedimentalmente carecen de una sensación de progresión y objetivo para los jugadores. De esta manera, el objetivo de los estudios acerca de este contenido también intenta abarcar este problema para que el resultado del contenido parezca diseñado por un, valga la redundancia, diseñador humano. Este reto está abierto a diversas interpretaciones sobre lo que sería contenido coherente, original y útil tanto para jugadores como para desarrolladores.

Pero, a pesar de ser un campo ardiente en su desarrollo hoy en día, en el lado más artístico de los videojuegos (música, modelado, animación, etc.) todavía se lleva cautela en su uso, precisamente porque los problemas mencionados anteriormente son más notorios

cuando se trata del aspecto de un personaje o la canción de un nivel. Si se abusa de su uso sin solucionar los problemas mencionados, se consiguen videojuegos sin elementos destacables y atractivos para los jugadores, nada más allá que productos para pasar el tiempo sin adentrarse en otros temas; es decir, productos contrarios a la tendencia que se sigue hoy en día en el desarrollo de videojuegos: productos que despierten emociones y sean rememorados. Por ejemplo, *The Legend of Zelda: Twilight Princess* sigue siendo un claro ejemplo a seguir en cuanto a diseño de niveles, con sus magníficas 9 mazmorras originales (ilustración 12).

Ilustración 12: The Legend of Zelda: Twilight Princess (2006)

3.2.1.2. Animación generada procedimentalmente

Una vez hemos visto en qué consiste tanto la animación en videojuegos como el contenido generado procedimental, es hora de adentrarse al fin en el tópico de la animación procedimental. Hay diferencia de opiniones en cuanto a incluirla bajo el término de PGC, si bien la definición de animación procedimental, en pocas palabras, consiste en el uso de algoritmos para generar animaciones en tiempo real.

En el mundo de los videojuegos, esta técnica es utilizada para recrear movimientos y situaciones físicas que serían difícil de recrear con animación tradicional, o una inversión de tiempo absurda. Destaca su uso como técnica más común mayoritariamente en la

animación de sistemas de partículas (impensable recrearlas tradicionalmente) y diferentes elementos como fuego, agua, ropa o pelo. Para personajes o criaturas, en general, su uso es menos usual, prácticamente restringido a unos escasos movimientos estandarizados, normalmente.

Como ya hemos visto, generalmente los movimientos de personajes en videojuegos son estáticos, es decir, cada movimiento es creado por un artista (sea a mano o a través de *mocap*), consiguiendo así una librería de movimientos del personaje, llamados *assets*. Este es el acercamiento estándar de la industria: la animación tradicional domina en el movimiento de personajes. Si bien hay ciertos movimientos en los que el uso de la animación procedimental se ha estandarizado, son casos muy específicos o pequeños detalles, no ciclos completos ni comportamientos de personajes, por lo menos en el lado más comercial de la industria.

Anteriormente hablábamos de los problemas a sortear en la generación procedimental de contenido, y en el caso de la animación se añade uno más: el gran reto de esta técnica es conseguir llegar a las altas expectativas de los observadores humanos sin tener que usar la estilización de un personaje (convertirlo a un estilo visual diferente al realismo, donde los movimientos no necesitan parecer totalmente sacados de la vida real) como solución.

Como dijo David Rosen en su charla *Animation Bootcamp: And indie approach to Procedural Animation* en la *Games Developers Conference* de 2014, la tendencia que se debía tomar para hacer evolucionar de manera eficiente la animación en el mundo de los videojuegos (y que estaba empezando a expandir entre los desarrolladores independientes) constaba de tres puntos fundamentales: la búsqueda de naturalidad antepuesta al realismo; la premisa de dirigir en vez de animar, es decir, decirle al personaje lo que tiene que hacer según unos parámetros en vez de animar a mano su completo movimiento; y, por último, la más importante y que comprende las dos anteriores, y es integrar la animación en el código, alejarse del temor a juntar "arte y código", pues se conseguirá el mejor resultado una vez se rompa la barrera entre ambos.

La animación procedural está recibiendo un enfoque moderno impulsado por el constante desarrollo en la industria del videojuego, especialmente gracias al auge del desarrollo independiente, pues en el ámbito más comercial todavía encontramos esa falta de integración de animación y código.

Dentro de esta manera de animar, la técnica más común es la simulación de físicas para recrear los movimientos, ya sea humanos o de otros elementos, de manera que se comporten como lo harían de estar sujetos a las leyes físicas del mundo real. Por otro lado, en los últimos años ha crecido la investigación y desarrollo de algoritmos basados en el uso de inteligencia artificial para crear las animaciones, un reto diferente a las técnicas procedimentales de animación más tradicionales. Y si bien no encajan al completo dentro de la definición de este tipo de animación, también destaca el uso de una fusión entre algoritmia y *keyframming* en varios de los últimos proyectos publicados recientemente, como veremos a continuación.

3.2.2. Técnicas usadas en videojuegos

3.2.2.1. Elementos: Sistemas de partículas, dinámicas de fluidos, ropa y pelo

La mayor parte de la animación procedural se basa en simulaciones físicas de menor o mayor complejidad, ya que la gran mayoría de veces se intentan representar situaciones inspiradas en el mundo real. Es más, en cuanto se habla de animación basada en físicas no tardan en venirnos a la mente los sistemas de partículas, las simulaciones de agua o el movimiento del pelo de un personaje. Durante años se han usado el código para animar elementos que, de otra manera, conllevarían una gran complejidad y una considerable cantidad de tiempo invertido.

A pesar de que este estudio se centre en la animación de personajes y criaturas, cabe incluir este apartado para hablar de una parte de gran importancia en los videojuegos.

Dentro de las animaciones de elementos podemos diferenciar entre los sistemas de partículas, los fluidos, el pelo y la ropa. Debido a que la animación de estos elementos recae estrictamente sobre simulaciones físicas y entra en su campo, veremos por encima los diferentes métodos sin entrar en la deconstrucción física de éstos.

Empezando por los sistemas de partículas, éstos consisten la técnica más popular a la hora de recrear efectos visuales de diversos tipos gracias a su sencilla implementación, eficiencia y control artístico, así como su versatilidad a la hora de aplicarse a distintas situaciones (fuego, humo, explosiones, nieve, polvo, efectos puramente fantásticos como hechizos, etc.). Además, más allá de su animación, a los sistemas de partículas se les pueden otorgar propiedades físicas para su participación en el *gameplay* y crear interacciones con los jugadores como repulsión o atracción. Éstos pueden ser, además, bidimensionales o tridimensionales y hoy en día todos los motores gráficos populares usados en el mercado cuentan con herramientas para su fácil implementación.

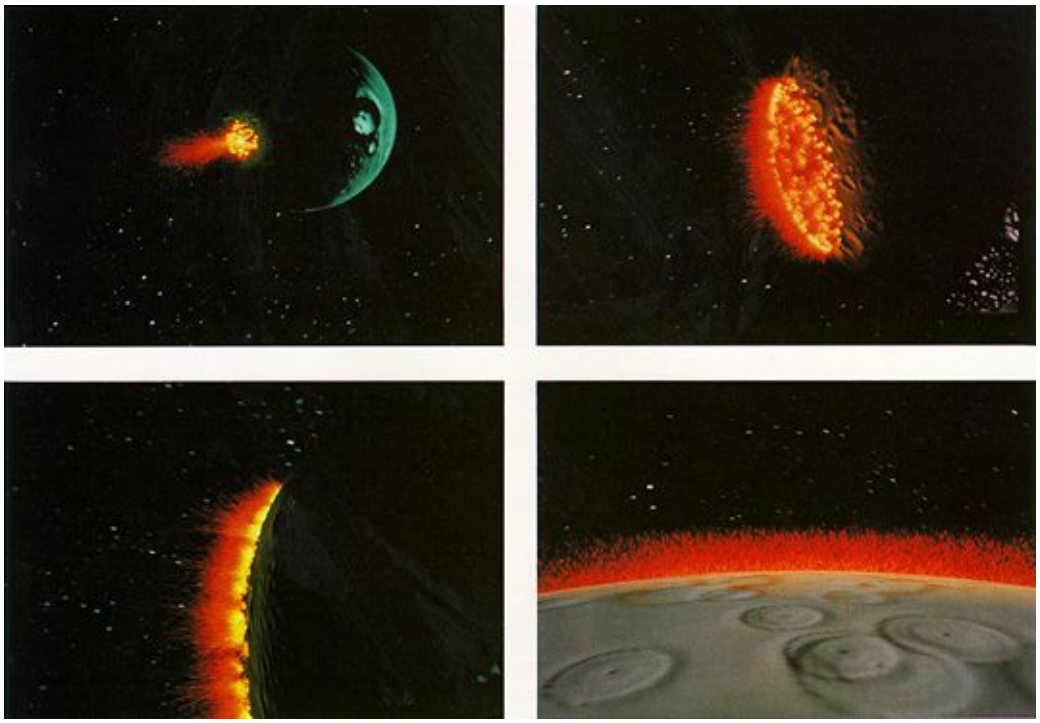


Ilustración 13: Star Trek II: La Ira de Khan (Nicholas Meyer, 1982)

En la secuencia conocida como Génesis de la segunda entrega cinematográfica de Star Trek nació el término *sistema de partículas*, siendo esta la primera vez que se usaba la técnica en el cine, en este caso para recrear un efecto de fuego sobre un planeta (ilustración

13). El comportamiento de los sistemas de partículas normalmente se compone de dos fases, a su vez separadas en varios pasos: la fase de actualización de parámetros, donde se realizan los pasos de generación, simulación y extinción, y la fase de renderizado. Inicialmente se establece un emisor, que ejerce como el origen de las partículas y cuenta con una serie de parámetros del comportamiento de las partículas, entre los que se incluyen el número de partículas generadas por unidad de tiempo, la dirección en las que son emitidas al originarse, el tiempo de vida de éstas, una serie de especificaciones sobre su aspecto visual y demás parámetros convenientes para su comportamiento. Normalmente, estos parámetros son difusos y el animador establece un rango de aleatoriedad según la situación para que el comportamiento de los sistemas sea más realista. A continuación, durante el proceso de simulación, las partículas son continuamente generadas y avanzan en una simulación física según sus características especificadas en los parámetros. A cada actualización del proceso, se comprueba si han excedido su tiempo de vida y, de ser así, son descartadas de la simulación. Las partículas pueden actuar bajo distintas fuerzas (gravedad, fricción, etc.) que cambiará su comportamiento de una manera u otra. La fase en que las partículas "mueren" y son eliminadas de la simulación consiste en el proceso de extinción.

Normalmente las partículas son elementos lógicos a los que se le aplican propiedades gráficas para que sean visibles, como la adición de material. De esta manera, cuando la fase de actualización es completada, cada partícula es renderizada, normalmente en forma de un cuadrilátero texturizado que siempre da la cara hacia el observador o en forma de un único pixel. También se utilizan pequeños modelos 3D para asegurar fidelidad incluso en altas resoluciones, sin consumir demasiado tiempo de renderizado. Además, la vida de una partícula puede renderizarse a lo largo del tiempo o de una sentada, creando así el efecto de muchas partículas moviéndose en distintos puntos del espacio o el efecto de curvas o filamentos de material que muestran la trayectoria completa de las partículas.

Este último punto nos lleva a hablar de la animación de pelo. Usando el renderizado de la trayectoria completa en una sola vez de las partículas que, como ya hemos visto, crean filamentos de material, se pueden crear simulaciones de pelo. El tema de la

animación y renderizado de pelo realista sigue llevando de cabeza a los programadores gráficos en la actualidad, debido a los múltiples problemas que supone este elemento. En el cine o en las mismas cinemáticas de videojuegos se suele añadir mucho más detalle que en el pelo en tiempo de juego, ya que el renderizado (ni la animación) no tiene que ser en tiempo real. De todas maneras, hay múltiples modos de construir el pelo de los personajes. Ya hemos visto uno usando los sistemas de partículas, pero el método elegido suele depender en gran parte del estilo visual y el nivel de estilización que tenga el proyecto. De esta manera, un método muy común es el uso de de polígonos y la consecuente aplicación de texturas de pelo que se renderizan con cierta transparencia (ilustración 14).

Ilustración 14: Ejemplos de pelo modelado a través de polígonos (Chuan Koon Koh)

De todas maneras, el modelado del pelo en videojuegos no es el tema de este estudio y no ahondaremos en detalle acerca de éste. En cuanto a la animación del pelo, que es lo que nos acontece, se suele trabajar con dinámicas de sólidos deformables (o *soft-body dynamics),* donde la distancia relativa de los puntos del objeto no es fija pero el cuerpo mantiene su forma hasta cierto grado (a diferencia de los fluidos). Generalmente estos métodos solamente proporcionan una simulación plausible a nivel visual y carecen de realismo científico, pero en el campo de los videojuegos solamente se intenta llegar a un acabado realista estéticamente, no a nivel de interacción. De esta manera, el resultado final suele ser un punto medio entre algo visualmente agradable y el coste computacional, pues la corrección física no es una prioridad en este caso. Los motores de físicas más usados en la industria (*Bullet, CryEngine 3, Bounce,* entre otros) cuentan con software que permite la

simulación de dinámicas de cuerpos deformables que pueden ser usados tanto con pelo modelado de manera poligonal (en la mayoría de casos cuenta con su propio *rigging*) como con pelo a base de filamentos.

Otro elemento de gran importancia que se anima gracias a las *soft-body dynamics* es la ropa y diferentes tejidos. Se conoce como simulación de tejidos a la simulación de cuerpos deformables en forma de dos membranas elásticas continuas (representan cada uno de los lados del tejido), ignorando la estructura actual de un tejido formado por hilos entrelazados (no es necesario llegar a ese nivel de detalle y la complejidad se dispararía). A través del renderizado se pueden producir emulaciones realistas de diferentes tejidos y ropa y conseguir un resultado más rápido y creíble que animando éstos de manera tradicional. La simulación de tejidos se puede dividir en dinámicas basadas en fuerzas o en posición. Las primeras toman las mallas poligonales que forman el tejido y determinan las fuerzas de resorte que actúan en cada instante (en combinación con fuerzas externas como la gravedad) y cuyas ecuaciones vienen dadas por la segunda ley de Newton; por otro lado, la mayoría de softwares de simulación de tejidos usan las segundas y el sistema de fuerzas se transforma en un sistema de restricciones en la que la distancia de los nodos que forman las mallas poligonales debe ser igual al principio y final de la simulación, evitando deformaciones permanentes del tejido y permitiendo diferentes grados de dureza del mismo, así como pudiendo añadirse otras restricciones que permitan que la ropa de los personajes esté siempre en su lugar.

Por último, a la hora de hablar de las animaciones de elementos basadas en físicas, es inevitable hablar de las animaciones de fluidos, otro de los grandes retos a representar en los videojuegos. Los fluidos tienen gran libertad de movimiento, el cual es no-lineal y cuya forma y topología puede variar.

Ilustración 15: The Witcher 3: Wild Hunt (CDProjekt Red, 2015)

El mayor bache en el camino a la hora de representar agua en videojuegos (ilustración 15) es el gran coste computacional que supone, pues no solo se tiene en cuenta su animación, sino su renderizado para alcanzar un resultado realista, teniendo en cuenta sus propiedades específicas de reflexión y refracción (por ello es que se está optando por lenguajes visuales más estilizados para esto, antes del fotorrealismo), por ello es mucho más sencillo conseguir simulaciones realistas *off-line* cuando uno se puede permitir amplios tiempos de renderizado. Así que al igual que con la ropa o el pelo, generalmente las simulaciones de fluidos en tiempo de juego se enfocan en la calidad visual del comportamiento de los fluidos, y no en resultados físicos rigurosos. Esto se consigue, entre otros, usando los efectos de los *shaders* sobre físicas de baja resolución, simulando solamente las zonas activas y visibles, reduciendo las dimensiones y trabajando bidimensionalmente en muchas ocasiones y, desde luego, trabajando el *level of detail* (disminuir la complejidad de los modelos 3D a medida que se alejan del observador o pierden importancia).

Destacan tres enfoques diferentes a la hora de animar agua en videojuegos, según su finalidad. Por un lado tenemos el uso de sistema de partículas, destinado a cuerpos de agua en movimiento como pueden ser salpicadura, chorros o incluso sangre cayendo. Es una técnica rápida y bastante sencilla de recrear con cualquier motor de físicas, que

representa el fluido como un conjunto de partículas, pero no es óptima para trabajar sobre superficies vastas de fluido. A la hora de representar superficies sin límites, como grandes océanos, se emplea la técnica estrictamente conocida como *agua procedimental*. La premisa es la simulación del efecto, no de la causa, lo que permite gran nivel de control e infinidad de movimientos en la superficie (dado que es lo único con lo que se trabaja y se trata como una membrana a que se le aplican funciones para recrear el movimiento, ya sean de distorsión de la malla o de las propias texturas), pero es un resultado menos realista y con el que supone una gran dificultad crear interacciones con la escena. Por último, una de las técnicas más usadas, especialmente a la hora de recrear fluidos con los que se vaya a interactuar de manera continuada, es la *height field water* o campos de altura. Usada mayormente para la simulación de lagos, estanques u otras superficies de agua limitadas, representa la superficie del fluido como una función 2D *u(x,y),* que se traduce en un array 2D u[i, j], reduciendo así las dimensiones. De esta manera, se crean columnas que, aplicadas las ecuaciones de olas basadas en las leyes Newtonianas, variarán en posición (la altura de la columna) y velocidad (el cambio de altura por unidad de tiempo), recreando así olas y diferentes movimientos de la superficie. No es un método físicamente correcto pero sí consigue recrear el efecto de manera eficaz ante el ojo humano.

Ilustración 16: Assassins Creed IV: Black Flag (Ubisoft, 2013)

En la cuarta entrega de la famosa saga *Assassins Creed* (ilustración 16) de *Ubisoft* podemos ver una combinación de las técnicas vistas anteriormente, como suele ser usual en juegos con gran interacción con fluidos. La base de la superficie se recrea con *height fields*, mientras que los efectos de salpicaduras son sistemas de partículas y el agua lejana con la que no se interactúa son mallas animadas procedimentalmente.

Vistas las diferentes técnicas procedimentales que se usan para la animación de diferentes tipos de elementos en videojuegos, podemos llegar a la conclusión que el objetivo es conseguir un resultado estéticamente agradable antes de una precisión física a la hora de acercarse al realismo.

3.2.2.2. Físicas *Ragdoll*

Se conocen como físicas *ragdoll (*término inglés que significa "muñeco de trapo") al proceso de animación usado específicamente para las muertes de personajes en videojuegos. La idea detrás de esta técnica es que el movimiento de los personajes puede ser simulado y se puede obtener un comportamiento realista si se replican sobre los personajes las restricciones y normas de un cuerpo humano. De esta manera, para crear este "muñeco de trapo", se crea un esqueleto donde cada extremidad se una por una articulación con el mismo grado de rotación y libertad que tendría en la vida real (al contrario que en el proceso de *rigging,* que vimos anteriormente, donde no se le suele prestar atención a dichas restricciones). Si a dicho esqueleto se le aplican las leyes físicas del mundo real, se puede simular de manera sencilla cómo una persona o criatura inconsciente caería. Por esto mismo se ha popularizado su uso desde hace años como "animación de muerte", pues permite tener personajes que caen de manera realista y cuyos cuerpos interactúan con el entorno independientemente de cuál sea éste, tarea casi imposible de conseguir con animación tradicional.

La mayor desventaja del uso de *ragdolls* es que son impredecibles y pueden llegar a darse situaciones ridículas con su uso (ilustración 17). Pero, aún así, los *ragdolls* se han

convertido en un estándar imprescindible en la industria para su uso específico en animaciones de muerte.

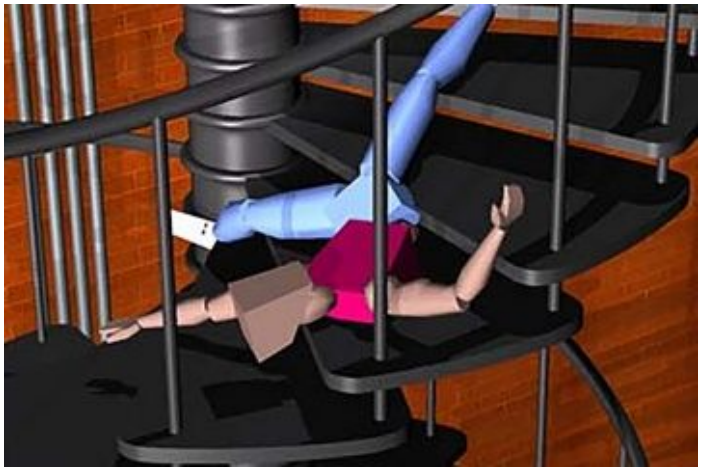


Ilustración 17: John Nagle Simulation

El primer juego en integrar estas técnicas fue el *Jurassic Park: Trespasser* (*DreamWorks Interactive)* en 1998. A pesar de no ser un éxito de mercado y pasar casi desapercibido en su época y sus primitivos *ragdolls* ser erráticos, supuso un importante hito en la animación de videojuegos y a partir de entonces se han usado físicas *ragdoll* en prácticamente todos los títulos publicados hasta la fecha.

Dentro de las físicas *ragdoll* encontramos diferentes aproximaciones y enfoques. El enfoque básico de "cuerpo rígido constreñido" aplica el algoritmo de Featherstone. La función de éste es estimar los efectos de las fuerzas aplicadas a estructuras rígidas unidas por juntas o enganches, usando una representación de coordenadas reducida. El resto de enfoques se consideran pseudo-*ragdoll*, a pesar de que se incluyen bajo el mismo término.

Una de las técnicas más comunes en videojuegos triple A y que cada vez se está usando más es la *blended ragdoll (*que se traduciría como *ragdoll* batida o muñeco de trapo batido), es básicamente la mezcla de *ragdoll* y animaciones. Consiste en utilizar una animación prefabricada y, una vez el ciclo termina, se aplican las físicas *ragdoll.* De esta manera, se evita la sensación de personajes que se quedan lánguidos de repente sin una transición desde su anterior movimiento y ofreciendo la correcta interacción con el entorno

al caer. Este método requiere tanto procesado de animaciones como de físicas, haciendo que sea más lento que el uso de *ragdolls* tradicional, pero ofreciendo un resultado más realista. La primera entrega de la serie *Uncharted* (ilustración 18) es uno de los ejemplos que usan la técnica de *blended ragdolls.*

Ilustración 18: Uncharted: Drake's Fortune (Naughty Dog, 2007)

Por otro lado tenemos la integración de Verlet, un método popularizado por Thomas Jakobsen. Es una técnica mucho más simple y rápida a la hora de resolver las simulaciones en sistemas rígidos completamente modelados, resultado así en un menor consumo de la CPU por parte de los personajes. La integración de Verlet es un método numérico usado para integrar las ecuaciones de movimiento de Newton y suponen una manera simple de resolver sistemas de múltiples partículas con restricciones. En el caso de aplicarlo a los *ragdolls*, cada hueso del esqueleto del modelo es considerado como una partícula o punto, conectado a un número arbitrario de otras partículas a través de restricciones sencillas. Un ejemplo que usa este método es el videojuego *Hitman: Codename 47 (IO Interactive,* 2000).

Por último tenemos el uso de post-procesado a través de cinemática inversa (técnica de la que hablaremos en páginas futuras), y que se basa en utilizar una animación de muerte prefabricada y, a través de las técnicas de cinemática inversa, forzar al esqueleto del personaje a adoptar una posición posible después de que dicha animación se haya completado, de manera que al terminar la animación completa de muerte, todos sus huesos

quedasen colocados en un espacio válido. No es un método popular a la hora de aplicarse junto a las físicas *ragdoll*, si bien la cinemática inversa es una técnica muy popular en el desarrollo de animación procedural. El famoso *Half-Life (Valve,* 1998) hace uso de esta técnica para sus personajes.

Partiendo de las técnicas base de las físicas *ragdoll*, siempre se puede añadir complejidad y detalle a los detalles y desarrollar sistemas de simulación del movimiento basado en parámetros no solo de grados de rotación, sino también de modelos físicos de huesos, músculos y nervios.

La principal ventaja del uso del *ragdoll* ante las técnicas tradicionales de animación es el realismo que ofrece a la hora de interactuar con el entorno, evitando tener que animar a mano todas las posibles muertes y caídas en multitud de diversas circunstancias. De esta manera obtenemos gran variedad, un grado de realismo muy alto y un ahorro de tiempo a nivel de desarrollo. Aún así, la mayoría de juegos se quedan con las aproximaciones simples que ofrecen las físicas *ragdoll*, pues no se considera que merezca la pena ahondar en detalles en este tipo de animaciones.

Actualmente, motores gráficos populares como *Unity* o *Unreal Engine* cuentan con herramientas de uso muy simple que permiten convertir modelos humanoides a *ragdoll* de manera rápida y sencilla.

3.2.2.3. Simulaciones de cuerpos rígidos

El problema de los *ragdolls* es que carecen de cualquier tipo de control motor y realmente conectar extremidades con articulaciones no permite que hagan nada más que caer en diferentes contextos, lo cual no sirve para representar el movimiento de personajes interactuando en un videojuego. Pero a través de las simulaciones de cuerpos rígidos y las bases de las físicas *ragdoll* se consiguen llegar a una aproximación útil que ha llevado a interesantes resultados poco vistos en los títulos más grandes de la industria.

Antes de introducirnos en cómo se usan las simulaciones de cuerpos rígidos en los videojuegos, cabe hablar sobre las dinámicas de sólidos rígidos. Si entendemos un cuerpo rígido como un sólido inflexible, en pocas palabras dichas dinámicas estudian el movimiento y equilibrio de sistemas de sólidos interconectados, ignorando sus deformaciones, de manera que se simplifica el análisis. Las dinámicas de un sistema de sólidos rígidos son descritas por las leyes de la cinemática y la aplicación de la segunda ley de Newton (la fuerza es directamente proporcional a la masa y a la aceleración de un cuerpo). De esta manera, la solución a estas ecuaciones produce una descripción de la posición, movimiento y aceleración de los componentes individuales del sistema y del sistema en su totalidad, en función del tiempo. Distinguimos las simulaciones de partículas de las de sólidos rígidos principalmente porque estos últimos tienen volumen y forma, de manera que incluyen el aspecto de la orientación, es decir, incluyen información rotacional e información que representa el cambio de orientación a través del tiempo.

Cuando se trata de videojuegos, se usan las físicas de sólidos rígidos de manera simplificada, a pesar de que los objetos en la vida real están muy alejados de esta idea de sólidos que no se deforman, ya que es fácil y barato a la hora de implementarse y, por tanto, la técnica más usada en videojuegos. Los movimientos de los cuerpos rígidos pueden simularse usando mecánicas newtonianas, resultando en una recreación lo suficientemente creíble y, por tanto, inmersiva, para el jugador.

El mayor coste computacional en las simulaciones de dinámicas de cuerpos sólidos viene dado por la detección de colisiones y la consecuente interpretación de dichas colisiones. En este contexto, una colisión ocurre cuando las formas de dos cuerpos rígidos intersectan. Como ya hemos dicho, la detección de colisiones es un proceso caro computacionalmente, así que se suele optimizar dividiendo dicho proceso en dos fases: la *broad phase* (fase ancha) y la *narrow phase* (fase estrecha). La primera es responsable de encontrar pares de cuerpos rígidos que vayan potencialmente a colisionar, excluyendo aquellos que seguro que no lo harán. La segunda, por tanto, opera sobre aquellos pares encontrados anteriormente que puede que colisionen, siendo así un sistema de refinamiento donde se determinan los cuerpos que realmente colisionarán (ilustración 19). Para cada

colisión encontrada, se computan los puntos de contacto. Una vez detectadas las colisiones, se deben resolver, es decir, es necesario determinar la consecuencia de dicha colisión.

Ilustración 19: Nextgen Sandbox (Proud Arts, 2017)

Con estos conceptos sobre la mesa, retomamos el problema de las físicas *ragdoll* que habíamos planteado anteriormente. Los *ragdoll* carecen de cualquier control de movimiento, solamente pueden caer. Pero combinando las dinámicas de dichos modelos unidos por juntas con el comportamiento de los cuerpos rígidos podemos conseguir movimientos realistas e interesantes, además de controlables. Por ejemplo, si tenemos un modelo formado por múltiples cuerpos rígidos que pueden colisionar, podemos hacer que varios de estos sean controlados vía código mientras que el resto se comporten como huesos conectados de un *ragdoll*, consiguiendo animaciones fluidas que de otra manera sería costoso de conseguir.

Podemos ver claramente esta técnica en el videojuego 2D *Rain World* (Videocult, 2017), que destaca por su multitud de criaturas y el realismo de las animaciones de cada una. Las extremidades que permiten el movimiento de dichas criaturas están controladas completamente por código y su comportamiento se rige bajo las dinámicas de cuerpos sólidos, mientras que el resto de sus cuerpos son una serie de huesos conectados por juntas de bisagra que se adaptan a todos los movimientos guiados por las extremidades.

Otro juego que destaca por el uso de esta técnica hasta el punto de que es su principal característica, es el famoso *Grow Home* (Ubisoft Reflections, 2015) (ilustración 20). Cuando el jugador mueve al protagonista B.U.D., un simpático robot, la posición de sus pies es forzada a través del código, comportándose como dos cuerpos rígidos e interactuando de tal forma con el entorno. Pero el resto de su cuerpo está sujeto a las mismas constricciones que un *ragdoll*, completando las animaciones de una manera plausible. Lo que al principio empezó como un intento fallido de desarrollar una herramienta para el rápido prototipado de personajes, se acabó convirtiendo en la principal característica y hasta un rasgo de la personalidad del protagonista de este juego.

Ilustración 20: Grow Home (Ubisoft Reflections, 2015)

Este concepto mixto se explota al máximo en el videojuego *Gang Beasts* (Boneloafs, 2017). Se abraza al completo el comportamiento torpe y flexible de los *ragdolls,* consiguiendo un resultado atractivo para los jugadores, que disfrutan del descontrol de las animaciones de estos muñecos de trapo. Pero lo cierto es que dichos *ragdoll* cuentan con una esfera invisible que actúa como soporte y como cuerpo rígido sujeto a sus propias dinámicas, además de los sólidos controlados vía código que se encuentran en las extremidades del modelo. El resultado visual se resume en personajes tontos que se mueven de manera aparentemente impredecible, pero sin margen de error.

3.2.2.4. Cinemática inversa

Las simulaciones de sólidos rígidos permite crear animaciones sencillas y las instrucciones son simples, pues los motores de físicas hacen casi todo el trabajo. Esta técnica funciona con personajes sencillos, pero la gran mayoría de veces todavía falla a la hora de mostrar realismo y complejidad, pues solamente se tienen en cuenta parámetros físicos como la masa o la gravedad, pero carecen del conocimiento del contexto en el que se darán las animaciones. En el amplio mundo que crean los videojuegos, llenos de fantasía y ciencia ficción, no siempre es conveniente seguir las restricciones impuestas por la física.

Una técnica impulsada en los últimos años y considerada por muchos animadores como el paso óptimo es la cinemática inversa. Usando cualquier tipo de *ragdoll*, esta técnica es capaz de encontrar la manera de moverlo al punto exacto deseado, cumpliendo la premisa vista anteriormente de *dirigir en vez de animar.* De esta manera, conseguimos un movimiento dirigido y pensado, evitando la aleatoriedad e imprevisibilidad que causan otras técnicas, pero con las ventajas de evitar el costoso trabajo de la animación tradicional. Mientras que en el *Rain World* o el *Grow Home* se deja a las físicas determinar cómo las juntas se moverán cuando son aplicadas una serie de fuerzas externas, la cinemática inversa las obliga a adoptar una postura deseada.

Si bien la cinemática inversa es útil en multitud de áreas, uno de sus usos más comunes es uno de los más sutiles: mantener los pies de los personajes en el suelo en todo momento. Es decir, es posible hacer caminar a un personaje por terrenos irregulares sin que sus pies floten o se hundan en éstos en diferentes momentos y no causen poses antinaturales. La cinemática inversa calcula la forma más natural de colocar el resto de huesos del cuerpo del personaje teniendo en cuenta en que sus pies deben estar en contacto con la superficie del suelo en todo momento (cuando se trate de caminar o correr, no en saltos u otros movimientos en el aire, obviamente). Bajo esta premisa, también se pueden conseguir otras animaciones que parecen triviales pero le dan un nivel de realismo y calidad al producto, como la animación de personajes humanoides cogiendo diferentes tipos de objetos, de manera que no hace falta animar cada situación tradicionalmente. Con

solamente indicar el objetivo, la cinemática inversa encuentra la manera más natural de mover el brazo del personaje. Si esto se hiciese mediante simulaciones de sólidos rígidos, se asemejaría más al movimiento de una marioneta cuyas extremidades son arrastradas por los hilos enganchados a sus articulaciones.

Así que, en resumidas cuentas, la técnica de cinemática inversa consiste en simular el movimiento de una cadena de huesos sin tener que controlar manualmente cada uno de ellos, permitiendo fijar poses determinadas y calculando el movimiento más natural para llegar a ellas. Originalmente se aplicaba para controlar brazos robóticos, si bien pronto se incorporó al campo de los videojuegos. En ejemplos como el de la ilustración 21, se muestra un gran uso de la cinemática inversa tanto en el movimiento sobre el terreno de los personajes y los animales, como a la hora de escalar y colgarse de los grandes colosos a las que se enfrenta.

Ilustración 21: Shadow of the Colossus (Team Ico, 2005)

Acerca de esta técnica también existen diferentes enfoques.

3.2.2.4.1. Raw IK (cinemática inversa básica)

El método estándar que viene de la mano de la definición de esta técnica. Se trabaja con un modelo con cuerpo articulado y se le indica de entrada una posición que debe alcanzar a través de los algoritmos de cinemática inversa, de manera que los datos con los que se trabajan son las posiciones a cada momento de cada uno de los huesos del cuerpo articulado en vez de con una pose completa (ilustración 22). Al contrario que la cinemática directa, que calcula la posición de las partes de la estructura a partir de la posición y rotación de la parte "padre" de la cadena, la cinemática inversa calcula dicha posición a partir de la posición final del objetivo.

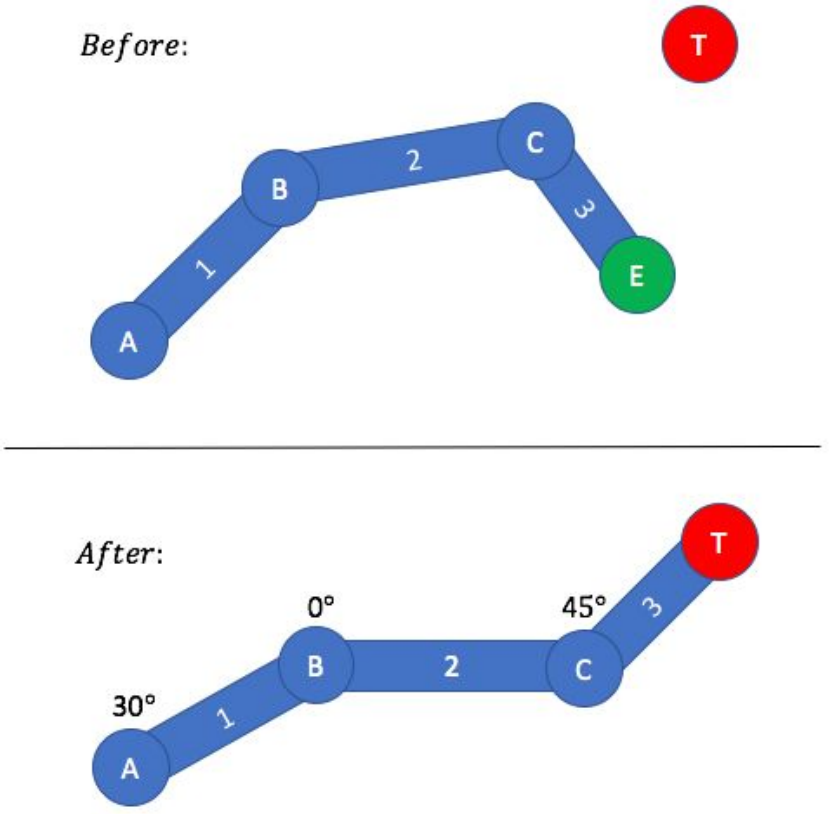


Ilustración 22: Overview of Inverse Kinematics (Luis Bermudez, 2017)

3.2.2.4.2. Semi IK (semi cinemática inversa)

Es muy habitual adoptar un enfoque mixto y usar la cinemática inversa solamente en pequeños detalles de las animaciones. De esta manera, toda la animación se realiza de la

manera tradicional, pero cierto elemento del modelo está siendo animado a base de cinemática inversa y, por tanto, puede resultar en un movimiento diferente a cada momento, al contrario que los movimientos pre-animados. Se utiliza principalmente para que personajes miren en diversas direcciones, ya sea en el caso de un enemigo manteniendo la mirada en sus objetivos o de un personaje mirando a otro mientras le habla. Puede ser aplicado en menor (ojos) o mayor (torso) medida, de múltiples formas según el tipo de personaje. Basta con fijar un objetivo en el hueso de la cabeza de manera que éste siga al jugador o a lo que sea que tiene que seguir. El movimiento resultante es realista, pero controlado gracias a la animación prefabricada y se puede aplicar a multitud de contextos, no solamente en los descritos.

3.2.2.4.3. Interpolación de *keyframes* en tiempo de juego

Anteriormente ya se ha hablado de la técnica de *keyframming*, usada actualmente en la animación tradicional para simplificar el proceso de animar a mano. En ese caso, el animador posa el modelo en diversas poses clave o *keyframes* del movimiento completo y el software de animación calcula los *frames* restantes entre *keyframe* y *keyframe*, resultando en una animación con un número variable de fotogramas. El proceso de calcular el movimiento entre poses clave se conoce como interpolación y es una gran ayuda a la hora de animar tradicionalmente, pues se ahorra mucho trabajo. Pero, aún así, de esta manera se siguen generando animaciones prefabricadas que se insertarán en el juego, con todos sus fotogramas (el estándar es 24 fotogramas por segundo, a pesar de que en el cine a veces se sube a 48).

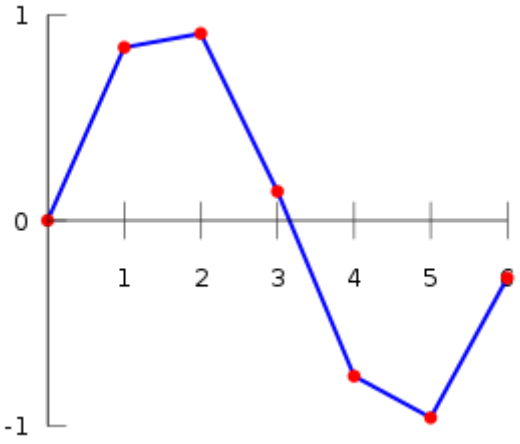
Cuando hablamos de interpolación, nos referimos a construir nuevos valores de datos en el rango de un conjunto discreto de datos ya conocidos. La interpolación de *keyframes* en tiempo de juego es una técnica de animación procedimental que se centra en el ahorro de un gran número de frames, bajando así la carga de datos en memoria en gran medida. El principio básico es usar el mínimo número de fotogramas clave y realizar la interpolación de ambos en tiempo de juego basándose en el *input* del jugador y el contexto del personaje (terreno, velocidad, movimiento previo, fuerzas externas, etc.). Así se

consiguen no solamente animaciones naturales, sino transiciones entre animaciones fluidas. Podríamos tomar esta técnica como un método mixto entre animación tradicional y procedimental, dado que utiliza fotogramas y código para crear la animación al momento. Este método resume al completo la premisa vista anteriormente de unir código y arte de manera que se pueden conseguir animaciones de gran calidad y fluidez, en menor tiempo y con mayor ahorro de memoria.

Las bases de esta técnica son sencillas: posar el personaje para crear dos fotogramas clave y aplicar el algoritmo de interpolación entre ellos. Lo más importante a la hora de realizar el primer paso es seguir la máxima de que los fotogramas clave son aquellos que describen el movimiento a realizar de la mejor manera (no el del principio y el del final de la animación, error común), de esta manera se consigue mantener la personalidad e integridad del movimiento. Por ejemplo, al realizar una animación de salto, los *keyframes* imprescindibles serían aquellos que muestran la pose durante la subida y la pose durante la bajada, ambas en el aire, pues indican la manera de saltar. El diseñador y animador David Rosen dio una conferencia en la *Games Developers Conference* de 2014 sobre esta técnica que revolucionó el panorama de la animación de videojuegos *indie*. En el momento de la charla, el proyecto *Overgrowth* (ilustración 23) en el que trabajaba, un videojuego de peleas protagonizado por animales humanoides, contaba solamente con dos trabajadores.

Ilustración 23: Overgrowth (Wolfire Games, 2017)

A continuación, la manera más rápida de obtener un resultado es aplicar una interpolación lineal. La interpolación lineal es un método de ajuste de curvas (encontrar la curva más aproximada que contenga una serie de curvas y posiblemente cumpla una serie de restricciones adicionales) usando polinomios lineales para aproximar un valor de la función *f(x)* en un valor desconocido de *x* (gráfica 1).



Gráfica 1: Interpolación lineal de la función f(x).

De esta manera, se interpola de un ángulo que conforma cada una de las uniones de los huesos del *rigging* al siguiente calculando una aproximación entre los dos frames. Esto ayuda a preservar la continuidad espacial, pero no tiene en cuenta otros factores como la velocidad o la aceleración, así que los resultados no son especialmente realistas para animaciones orgánicas. Sin embargo, si se usa la interpolación bicúbica, que es una extensión de la interpolación cúbica (puede conseguirse a través de algoritmos de convolución cúbica) para interpolar datos en una red bidimensional, se consiguen resultados mucho más suavizados y por ello es el estándar a la hora de simular comportamientos realistas. De esta manera, además de preservar la continuidad espacial, también se preserva la continuidad de la velocidad.

A pesar de que la interpolación bicúbica sea el estándar a la hora de trabajar con keyframes, es frecuente el uso de distintos algoritmos para hallar las curvas deseadas según los parámetros de distintas funciones físicas, como puede ser para crear animaciones de amortiguación basándonos en el modelo físico de masa-resorte-amortiguador, o teniendo

en cuenta anticipación y momento angular en saltos. Una de las grandes ventajas de este método es que se puede aplicar cualquier tipo de curva de interpolación a casi cualquier tipo de animación que se quiera realizar, permitiendo una gran versatilidad y la transición fluida entre todo tipo de animaciones. Además, partiendo del reducido número de fotogramas con los que se trabajan, añadir variaciones a los movimientos, como puede ser un personaje sujetando un arco a dos manos o una espada a una mano, se convierte en una tarea muy sencilla sin tener que recurrir a superposición de poses o recreación de cientos de fotogramas para múltiples animaciones que contengan dicha variación.

Al usarse algoritmos de interpolación y no estrictamente de cinemática inversa, a veces se suele distinguir como una técnica a parte, pero sus principios siguen siendo los mismos. Esta técnica es usual verla de la mano de otros tipos de animación procedimental como ayuda para refinar las animaciones, como puede ser con el uso estándar de cinemática inversa visto anteriormente o físicas secundarias en ciertos elementos (capas que se mueven, orejas de animal que botan) para añadir todavía más naturalidad.

Es importante recordar que la interpolación de *keyframes* en tiempo de juego crea la necesidad de centrarse en los momentos importantes de la ejecución del movimiento, recogiendo solamente las poses importantes y evitando las repeticiones y las transiciones innecesarias.

A pesar de la utilidad y los increíbles resultados de la cinemática inversa, sigue siendo una técnica que en la gran mayoría de los casos no se utiliza para realizar una animación final compleja, sino como una herramienta para el pulido de dicha animación, por lo que es muy normal encontrarse el enfoque de la *semi IK* visto anteriormente en la gran mayoría de videojuegos. La principal razón es porque la animación tradicional permite “mayor control y precisión” pero, como hemos podido ver, el descontrol no es un problema real en esta técnica.

3.2.2.5. Inteligencia Artificial

Una de las técnicas más innovadoras y recientes en la animación procedimental, que se ha estado investigando e incorporando en los últimos años, es el uso de inteligencia artificial para animar personajes. No hay que confundir el uso de IA para animar con la simulación de comportamiento (como las técnicas que recrear el comportamiento de las bandadas de pájaros o las que recrean batallas a tiempo real), error habitual al ver personajes moverse. En estas últimas se combina la simulación del comportamiento con inteligencia artificial con animaciones prefabricadas, cosa que dista del tema principal de este estudio, que es la animación procedimental. De esta manera, cuando hablamos de animación procedimental a través de IA nos referimos al desarrollo de sistemas que generen animaciones al momento dependiendo del *input* del jugador, en vez de almacenar todos los datos de las animaciones e ir seleccionando la que más convenga en el momento. Así se pueden dar infinidad de combinaciones de movimiento basadas en el terreno, la posición del personaje o el *input* del jugador, entre otros, para generar complejas interacciones con el terreno y, lo más importante, de manera que el sistema aprenda cuales son las que más le conviene usar en cada momento.

Uno de los estudios que destacan en este campo es el basado en el uso de redes neuronales con función de fase para el control de personajes, llevado a cabo por un equipo de la universidad de Edimburgo y *Method Studios* (ilustración 24). Este sistema empieza por la creación de una gran base de datos de, valga la redundancia, datos de animaciones. A continuación, se usa *machine learning* (rama de la inteligencia artificial basada en la idea de que los sistemas pueden aprender de datos) para producir las animaciones basándose en el *input* del jugador. Para la demo que presentaron en 2017, 1.5GB de datos de *mocap* fueron capturados y pasados a una red que pasó 30 horas estudiándolos. De esta manera, la red aprendió a combinar esas animaciones para crear completamente nuevas combinaciones según las necesidades del juego, creando así un rango mucho más amplio de animaciones que las que fueron capturadas en un inicio.

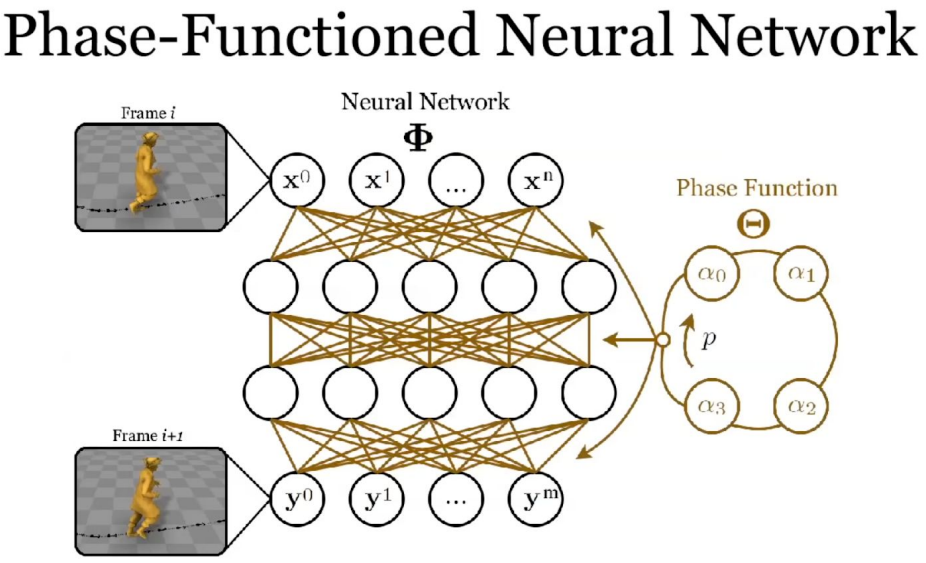


Ilustración 24: Phase-Funcitoned Neural Networks for Character Control (Holden, Komura, Saito, 2017)

"Esperamos que esta técnica permita a los diseñadores a ser más atrevidos y creativos con el tipo de entornos que crean" - Daniel Holden, 2017

El mayor problema de este sistema es la necesidad de adquirir los datos y saber cuáles son necesarios para que el resultado de las animaciones se pueda adaptar a la situación deseada, además del tiempo que lleva que la red aprenda de los datos y, lo más importante, del tiempo real de proceso para generar dichas animaciones.

El estudio de la Universidad de Edimburgo no es el único en la línea del uso de *machine learning*. En la *keynote* dada por Unity en Berlin en 2018, fue anunciado (y a día de hoy todavía no ha sido publicado) un paquete desarrollado por Unity Labs llamado Kinematica. Kinematica consiste en un sistema que cuenta con una única librería previamente formada por las animaciones creadas por los artistas y decide (automáticamente y en tiempo real) combinar pequeños fragmentos de dicha librería en una secuencia que coincida con el *input* del controlador, el contexto del escenario y cualquier requisito del *gameplay*. A su vez, el desarrollador decide (basado en su propio diseño de juego) cuáles serán las animaciones base para sus personajes e incluso podrá definir la forma en la que estas interactúan con el escenario. El objetivo de este sistema,

por tanto, no es crear animaciones desde 0, sino optimizar el proceso de transición entre animaciones prediseñadas y utilizar estas últimas para crear variedad. Según anunciaron en la conferencia de presentación, el mayor beneficio de este sistema es obtener un resultado de alta calidad y versatilidad gracias al gran número de variaciones que pueden obtenerse del mismo set de datos, además de no tener que preocuparse por la síntesis de las animaciones a través de estructuras de grafos o árboles de animaciones, permitiendo a los animadores iterar más rápido y centrarse en el apartado artístico. No tenemos que confundir esta técnica usando inteligencia artificial con el *motion matching*, otra técnica bastante reciente que está siendo implementada en varios motores gráficos y de animación, pero que no implica el uso de inteligencia artificial.

Por otro lado, enfocado al completo a la animación 2D, en 2017 se presentó Midas Creature, un software de Midas Touch Interactive (cuyo technical director es un veterano del famoso estudio Pixar). Dicha herramienta permite a los animadores dar una serie de instrucciones y datos sobre un personaje 2D a un motor de inteligencia artificial, y el software interpretará el personaje y decidirá de qué manera tiene que moverse. Adicionalmente, este programa también permite traducir datos de captura de movimiento de 3D a 2D, permitiendo usarlos en las animaciones.

A menor escala podemos encontrar ya algunos ejemplos del uso de la inteligencia artificial en animación en videojuegos populares. Es el caso del *Grand Theft Auto V,* cuyas animaciones de *ragdolls* son supervisadas por una inteligencia artificial para evitar poses imposibles para el cuerpo humano en el mundo real. Pero, además de escasos ejemplos similares a este último, realmente es una técnica todavía en desarrollo prematuro y no destaca por su implementación en el mercado actual. A pesar de ello, es un campo de investigación está siendo sujeto de gran interés durante los últimos años y sin duda tiene mucho que ofrecer en el futuro de la industria del videojuego.

La cuestión principal que se interpone en el uso de la animación a través de inteligencia artificial tiene respuestas contrarias según distintos desarrolladores. Usando inteligencias artificiales para animar al completo personajes de un videojuego, ¿hará que

carezcan de personalidad o del "estilo de animación" que se les pueden otorgar incluso con las técnicas vistas anteriormente?

3.2.3. Herramientas

La base y corazón de la animación procedimental, al fin y al cabo, es el código. A través de fórmulas matemáticas y leyes físicas en forma de código se puede dar vida a personajes, objetos y mundos en general. Pero, a pesar de ello, el uso de herramientas específicas dedicadas a esta técnica de animación siempre son una gran ayuda y optimizan el proceso de trabajo. Es por eso que en los últimos años, con el creciente desarrollo de este tipo de animación, han surgido cada vez más herramientas dedicadas especialmente a ella. Especialmente en estudios de videojuegos del panorama *AAA*, la tendencia cada vez se acerca más al propio desarrollo de sus herramientas de animación. A pesar de la variedad de opciones que se presenta hoy en día, destacan un pequeño número en cuanto a uso.

Por un lado tenemos la incorporación más o menos reciente de *add ons* o paquetes externos añadidos a softwares 3D. No es inusual encontrar animaciones procedimentales desarrolladas en estos programas a través de *scripts* desarrollados por animadores independientes, pero los programas de 3D más usados hoy en día ya empiezan a contar con distintas herramientas que facilitan el proceso.

Ilustración 25: ANIMAX, Procedural Animation System for Blender (2017)

Blender, el software de modelado 3D gratuito más usado, como ya hemos visto anteriormente, cuenta con ANIMAX (ilustración 25), un sistema para conseguir animar múltiples objetos de manera que sería complicado con las herramientas de animación originales de *Blender.* Este *plugin* está especializado en animaciones de partículas y simulaciones de objetos separándose en distintas piezas o rompiéndose y permite trabajar a través de una serie de parámetros ajustables según el resultado al que quiera llegar el usuario. Su objetivo es agilizar el proceso de trabajo y por ello cuenta con una librería de efectos establecida que pueden customizarse o crear desde 0 y, a pesar de ser una herramienta procedural, los parámetros pueden ser adaptados al proceso de *keyframming*. Las principales limitaciones de esta herramienta son su uso específico para un abanico muy pequeño de efectos y el límite de objetos a animar. Además, solamente permite controlar la localización, rotación y escala de los objetos con los que trabaja.

Por otro lado, *3DSMax*, uno de los estándares de la industria, como también se vio, cuenta con una serie de modificadores de animación procedimental que, cuando se superponen y mezclan, pueden crear complejos resultados a la hora de deformar modelos y también pueden ser controlados a través de los fotogramas clave. Una vez más, esta herramienta está especialmente enfocada a la animación de partículas y elementos similares.

Por otro lado, los motores gráficos más usados en la industria del videojuego no han tardado en añadir sus propias funcionalidades en el campo de la animación procedimental. Por un lado *Unity*, uno de los motores gráficos más usados en la industria, además de haber presentado el software Kinematica del que se habló con anterioridad, cuenta con *Mecanim*, una tecnología de animación que ha estado en desarrollo durante años. Trabaja con cinemática inversa para conseguir movimiento de personajes y objetos de resultado fluído y natural, con una interfaz eficiente. Esta herramienta, además, incluye facilidades para la creación de máquinas de estados y árboles de mezcla desde el propio editor de *Unity,* además de una librería de animaciones prefabricadas disponible en su propia tienda de recursos, y permite a desarrolladores ofrecer sus propias animaciones en dicha tienda para su uso con *Mecanim.*

En la misma línea de cinemática inversa, técnica que parece que está despuntando a la hora de animar personajes, por encima de las otras vistas, a finales del 2018 fue publicada una herramienta experimental y todavía en desarrollo para el famoso motor gráfico *Unreal Engine* de *Epic Games*. *Unreal* ya contaba anteriormente con *FABRIK,* una herramienta para resolución de cinemática inversa que trabajaba con cadenas de huesos de distinto largo. El *CCDIK (Cyclic Coordinate Descent Inverse Kinematics),* por tanto, es una herramienta similar a la anterior, pero con la gran diferencia de permitir definir constricciones angulares, gran utilidad a la hora de limitar rotaciones de huesos a la hora de obtener los resultados de la cinemática inversa, consiguiendo movimientos más realistas todavía. A pesar de ser presentada como una herramienta ajena a *FABRIK*, se entiende como una mejora de ésta y espera sustituirla cuando su desarrollo esté terminado.

Gracias a estas incorporaciones en los motores *Unity* y *Unreal Engine*, se ha puesto al alcance la animación procedural de personajes a los desarrolladores usuarios de estos motores, permitiendo trabajar en los propios editores de manera efectiva sin tener que recurrir a programas ajenos o a complejos algoritmos.

Pero quizá la herramienta más destacable en los últimos años especializada en este tipo de animación sea *Euphoria*, desarrollada por *Natural Motion*. No es un paquete de un motor gráfico ni un *add on* de un programa de 3D, sino que consiste en un motor de animación especialmente dedicado a la animación procedimental. Su pilar principal es que se basa en la Síntesis de Movimiento Dinámico, tecnología patentada por *Natural Motion* que consiste en la simulación completa de un personaje 3D, incluyendo cuerpo, músculos y sistema nervioso. Las acciones y reacciones de dichos personajes son sintetizadas en tiempo real y son diferentes cada vez. La Síntesis de Movimiento Dinámico funciona bajo el principio de usar el poder de procesado de la CPU para crear los movimientos de los personajes que, citando a *Natural Motion*, "prácticamente se animan por sí solos". Los impulsos nerviosos que son simulados mueven las fibras musculares que, en consecuencia, mueven el esqueleto completo, es decir, las simulaciones se basan en su totalidad en la biología. Similar a los principios de la cinemática inversa, *Euphoria* trabaja con "poses

activas" y el sistema determina si el personaje tiene el poder muscular necesario para alcanzarlas, y de qué manera lo haría. Con esto los desarrolladores querían poner fin al "cuello de botella" que se estaba formando en el desarrollo de animaciones: cada vez era requerida una mayor cantidad y realismo, dado que los juegos cada vez eran más grandes, y los métodos tradicionales de animación empezaban a no alcanzar a los requerimientos de las nuevas generaciones de videojuegos, además de que los *assets* creado no eran verdaderamente interactivos.

Euphoria es compatible con todos los motores de físicas comerciales y ya ha sido usado en títulos *AAA* y en juegos de su propia cosecha como el *Backbreaker* o el *Clumsy Ninja.* Tanto el juego *Star Wars: The Force Unleashed II* (ilustración 26) como su precuela fueron de los primeros videojuegos en trabajar con el software *Euphoria*, originalmente desarrollado para una entrega cancelada del mismo estudio, *LucasArts*.

Ilustración 26: Star Wars: The Force Unleashed II (LucasArts, 2010)

En 2009 fue publicado *Grand Theft Auto IV,* el que sería la primera entrega del estudio *Rockstar Games* trabajando con *Euphoria.* Hoy en día, este estudio no solamente ha seguido trabajando con el software en sus siguientes proyectos publicados, sino que ha integrado el motor de animaciones en su propio motor de juegos *Rockstar Advanced Game Engine (RAGE).*

A pesar de haber desarrollos independientes de herramientas similares y otros softwares todavía en desarrollo, estas son las mayormente utilizadas hoy en día en la industria. A pesar de ello, todavía no se puede hablar de un estándar en la industria a la hora de herramientas, pero se prevé que se llegue a esto en un futuro cercano.

3.3. Esquema de las técnicas de animación

Técnicas de animación tradicional usadas en videojuegos	
Keyframming	Técnica en la que el animador trabaja sobre un modelo 3D para colocarlo en una serie de poses que formarán una animación completa.
Captura de movimiento	Técnica en la que se graba el movimiento de un actor y se traslada a los modelos creados digitalmente.

Tabla 1: Resumen de técnicas de animación tradicional usadas en videojuegos.

Técnicas de animación procedimental usadas en videojuegos	
Sistemas de partículas	Técnica con la que se recrean distintos tipos de efectos. Se basa en la aplicación de leyes y propiedades físicas a un número variable de partículas.
Dinámicas de fluidos	Conjunto de técnicas basadas en leyes físicas para la animación de superficies de agua. Suelen ser costosas y tener gran coste computacional.
Simulaciones de cuerpos deformables	Técnica basada en las dinámicas de sólidos deformables que normalmente sólo proporciona una simulación plausible a nivel visual y no práctico. Usadas para animar pelo y ropa.
Físicas Ragdoll	Técnica basada en física Newtoniana usada para animar cuerpos sin vida o inconscientes.
Simulaciones de cuerpos rígidos	Técnica basada en las dinámicas de sólidos no deformables que las usa de manera simplificada, la técnica más usada en videojuegos.
Cinemática inversa	Técnica que calcula el movimiento que debe seguir una cadena de huesos para alcanzar la pose deseada. Especialmente útil en la animación de personajes.
Inteligencia artificial	Técnica relativamente reciente que se basa en el desarrollo de sistemas de inteligencia artificial que generan animaciones en tiempo de juego.

Tabla 2: Resumen de técnicas de animación procedimental usadas en videojuegos

4. Metodología

Para la primera parte del proyecto, en la que se exponen los conceptos básicos sobre la animación y se define la animación procedimental, acompañado todo de una enumeración y explicación sobre cada una de las técnicas que se utilizan, se ha realizado una recopilación y analizado de información de distintas fuentes. Dichas fuentes se componen en su mayoría por conferencias de profesionales tanto de la industria de los videojuegos (en mayor medida) como de físicos y animadores (en menor medida), y estudios publicados por universidades de todo el mundo. Cabe destacar la utilidad de las conferencias de la *Game Developers Conference*, el evento anual más grande de desarrolladores de videojuegos que tiene como objetivos el aprendizaje, la inspiración y el *networking* entre profesionales. Las conferencias realizadas en dicho evento son eventualmente subidas a la plataforma *YouTube*, al alcance de cualquier interesado, y son presentadas por todo tipo de profesionales de la industria, desde independientes hasta trabajadores de grandes estudios.

De esta manera, una vez recopilada y analizada la información, se ha buscado resumirla en explicaciones sencillas al alcance de un nivel intermedio-básico de conocimiento del tema, intentando definir los puntos sin entrar en detalles técnicos y utilizando el apoyo principal de ejemplos sacados de productos publicados que, en este caso son, en su gran mayoría, videojuegos.

A la hora de elegir el sujeto de desarrollo, se ha decidido trabajar específicamente sobre la animación de enemigos, ya que las criaturas y los personajes son los elementos que mayormente se animan tradicionalmente en videojuegos. Después de barajar la animación de un personaje principal se ha preferido aplicar un enfoque más general y trabajar sobre la idea de enemigos no humanoides cuyos movimientos sean el desplazamiento en función de la posición del jugador y su reacción a dicho jugador. En este caso no se ha entrado en el campo del diseño de mecánicas de enemigos, es decir, de ataques y similares, pues no es el punto del proyecto, pero sí podría ser un derivado a implementar si se diese una extensión de este desarrollo. Por esto mismo, después de

exponer y contrastar las distintas técnicas de animación procedimental, se ha elegido implementar un sistema de cinemática inversa para crear el sistema de animación procedimental. La cinemática inversa es una técnica de animación procedimental que permite determinar el movimiento de una cadena de articulaciones para lograr que se adopte una posición concreta.

En cuanto al entorno de desarrollo, se ha elegido trabajar con un motor gráfico, tanto para facilitar la implementación y la obtención de resultados gráficos inmediatos, como para aprovechar su uso para extender el aprendizaje sobre éste. Las dos opciones planteadas han sido *Unity* (ilustración 28) y *Unreal Engine* (ilustración 27), dos de los motores gráficos más usados en el sector de los videojuegos en la actualidad.

Ilustración 27: Logotipo del motor Unreal Engine 4

Unreal Engine es un motor gráfico de alta potencia creado por *Epic Games* que destaca por su sistema de implementación basado en *Blueprints*, es decir, un sistema visual de *scripting* que permite un desarrollo más rápido y centrado en el diseño y lógica del juego sin tener que usar *C++*, el lenguaje en el que se desarrolla en este motor.

Ilustración 28: Logotipo del motor Unity

Unity, por otro lado, es un motor gráfico de *Unity Technologies* usado especialmente en proyectos que requieren baja potencia, como los videojuegos móviles. Utiliza el lenguaje *C#* y se trabaja más activamente con *scripts* que en *Unreal.* Por eso ha

sido el motor gráfico usado en este proyecto. Además del mayor conocimiento de la interfaz de *Unity* y la opción de ampliar el conocimiento en su uso, dado que el objetivo era desarrollar un sistema basado puramente en programación, el uso de un motor más sencillo para programar *scripts* de animación ha sido la decisión final. De todas maneras, no se descarta explorar en un futuro el desarrollo de un sistema de cinemática inversa para *Unreal Engine*, o la adaptación del propio.

Por último, al tratarse de un proyecto de desarrollo centrado solamente en un sistema con un número pequeño de funcionalidades y llevado a cabo por una sola, no ha sido imperativo el uso de una herramienta de planificación de tareas. De todas maneras se ha decidido usarla para tener un seguimiento más claro y agilizar el flujo de trabajo. Para ello se ha elegido usar *Trello* que, a diferencia de otras herramientas similares como *JIRA* (usada con más frecuencia en el sector profesional), cuenta con un plan gratuito con todas las funcionalidades necesarias para llevar el control de las tareas del proyecto. De esta manera, se han podido listar las diferentes tareas (ilustración 29) y clasificar en distintos estados de procesado.

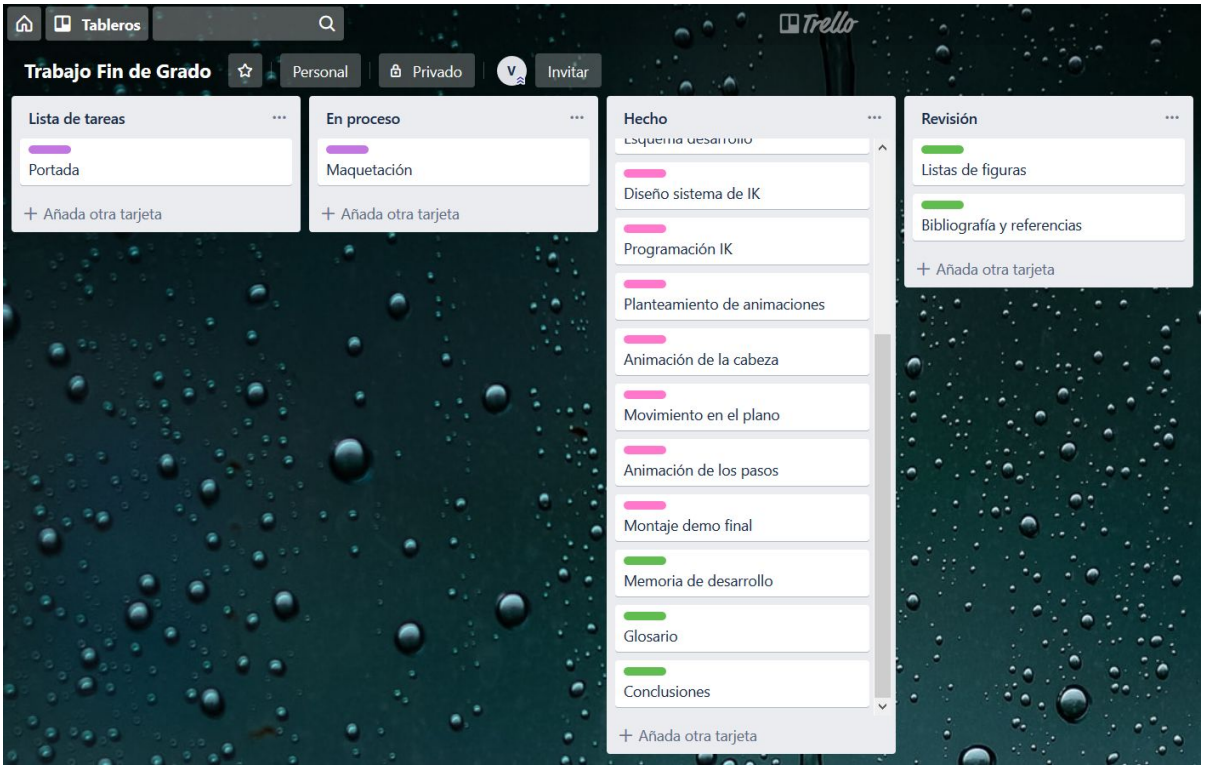


Ilustración 29: Captura de tareas del tablero de Trello

5. Desarrollo

5.1. Conceptos Iniciales

En esta segunda parte del proyecto se detalla el proceso de planteamiento del sistema y su posterior desarrollo. Para facilitar la implementación de los objetivos y la explicación del proceso, antes cabe definir una serie de conceptos básicos.

5.1.1 Componentes

Antes de hablar sobre elementos específicos del motor gráfico *Unity* relevantes para el proyecto, es importante indicar que éste está basado en una arquitectura basada en componentes. El flujo de trabajo de *Unity* se construye alrededor de esta estructura, alejándose del enfoque clásico de la programación orientada a objetos.

En programación, un componente es una pequeña pieza de una máquina más grande con una función o funciones específicas que, en el mejor de los casos, puede realizar esta tarea sin la ayuda de fuentes externas. Además, no pertenecen solamente a una máquina, sino que pueden combinarse con varios sistemas para conseguir tareas específicas. Los principios detrás de éstos son que sean reusables, diseñados sin un contexto específico, extensibles, encapsulados e independientes.

En *Unity,* esto se refleja de manera visual e intuitiva en la parte de la interfaz llamada Inspector, un panel que muestra todas las propiedades de un objeto. Si un enemigo, por ejemplo, tiene tres componentes, cada uno estará listado en este panel, junto a sus variables públicas y demás parámetros.

En un ejemplo sencillo sobre el sistema de componentes, se puede imaginar el personaje de un videojuego. Si este tiene la habilidad de volar, se le asignará un componente de vuelo. Si puede ser herido y recuperar vida, tal vez se le asigne un

componente de salud. Si puede morir, en cambio, será uno de muerte. Cada una de las características y acciones que podrá hacer nuestro personaje se definirá en un componente que se le será asignado. Un enemigo y un jugador (ilustración 30) serían dos instancias de la misma clase, pero serían sus componentes diferentes los que harían que fuesen entidades distintas.

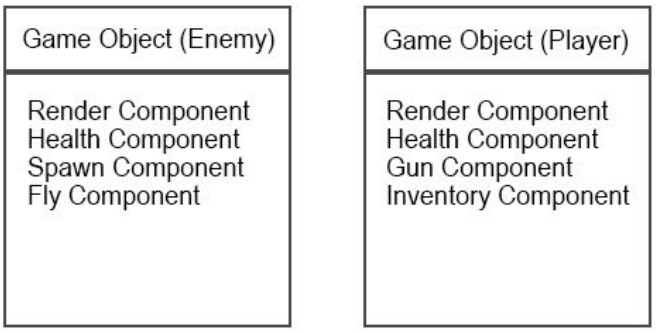


Ilustración 30: Ejemplo de estructura de componentes (Elaboración propia)

Lo más difícil de trabajar con componentes es aprender a estructurar los proyectos para adaptarlos a su uso. Las clases grandes se dividen en muchas piezas pequeñas y la comunicación entre ellas puede ser un quebradero de cabeza. Por suerte, trabajando con *Unity* existen una serie de funciones del propio motor gráfico que nos permiten conseguirlo de manera relativamente sencilla, consiguiendo referencias a componentes específicos, comprobando todos los objetos en búsqueda de aquellos que contengan un componente en concreto, etc.

5.1.2. Unity Prefab System

En el ámbito de *Unity*, se denomina *prefab* (término proveniente de la palabra inglesa *prefabricated,* es decir, prefabricado) a un tipo de archivo que te permite almacenar un objeto con todas sus propiedades dentro, actuando como una plantilla que puede usarse para crear nuevas instancias del mismo objeto en cualquier parte del proyecto. Esto permite un control mucho mayor sobre los *assets* o recursos utilizados en el proyecto, pudiendo duplicarlos y editarlos independientemente de manera rápida. Además, a pesar de ser copias independientes, cualquier edición que se haga dentro del *prefab* se verá reflejada en

todas las instancias de éste, permitiendo hacer cambios rápidos en muchos objetos esparcidos por todo el proyecto al mismo tiempo. De esta manera, por ejemplo, se pueden crear personajes con las mismas características pero que varíen entre ellos en según qué parámetros, todo de manera rápida y ágil. Además, los *prefabs* permiten establecer jerarquías, almacenar otros *prefabs* en su interior y sus instancias pueden ser creadas a en tiempo de ejecución desde un *script*.

Los *prefabs* pueden ser fácilmente creados seleccionando *Asset* > *Create* > *Prefab* en el menú principal de *Unity*. Esta acción abrirá un nuevo *prefab* vacío y tan solo basta con arrastrar los objetos deseados en su interior. Si a continuación se arrastra dicho *prefab* a la escena en la que se esté trabajando, se habrá creado una instancia. A partir de ahí, el objeto de la instancia puede ser editado libremente y, de querer editar el *prefab*, solo hace falta seleccionarlo en el panel del proyecto y ajustar las propiedades en el inspector.

De esta manera, los *prefabs* otorgan la flexibilidad de poder hacer cambios a gran escala que se reflejen en el resto de objetos o cambios individuales y concretos según sea necesario.

5.1.3. *Scripts*

Como ya hemos visto, en *Unity* el comportamiento de los objetos (o *GameObjects*) es controlado por los componentes que los forman. *Unity* permite crear nuevos componentes a través de los *scripts*, y con estos se pueden activar o desactivar eventos de juego, modificar propiedades de otros componentes y responder al *input* de usuario de cualquier manera deseada. Para ello, *Unity* utiliza el lenguaje de programación C#. El *scripting* o programación de *scripts* es el ingrediente principal en todos los proyectos desarrollados en *Unity*, pues son la base para determinar cualquier aspecto del proyecto.

Unity permite crear *scripts* directamente dentro de la aplicación con un simple clic a través de *Assets* > *Create* > *C# Script* (ilustración 31) en el menú principal, y automáticamente aparecerá el archivo .cs en la carpeta del proyecto que tengas

seleccionada. A partir de ahí se puede editar en un editor de texto externo y contendrá una clase base llamada *MonoBehaviour*, de la que todo script en C# debe derivar. También habrá dos funciones definidas en la clase: la función *Update()*, donde se colocará el código que gestionará la actualización de fotogramas para el objeto (movimiento, respuestas al *input* del usuario, activación de acciones…); y la función *Start()*, que será llamada por *Unity* antes de que la ejecución del juego comience.

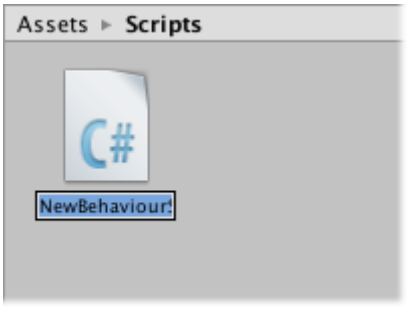


Ilustración 31: Creación de un script de Unity (Elaboración propia)

Un *script* actúa como "plantilla" de un componente, así que no funcionará hasta que no esté asociado a un objeto. Para llevar a cabo esta acción solamente es necesario arrastrar el *script* deseado desde la carpeta contenedora hasta el objeto objetivo en el panel de jerarquía de la interfaz. Su integración será representada como un componente más en el panel Inspector (ilustración 32) y, una vez realizado esto, empezará a trabajar cada vez que se inicie la ejecución.

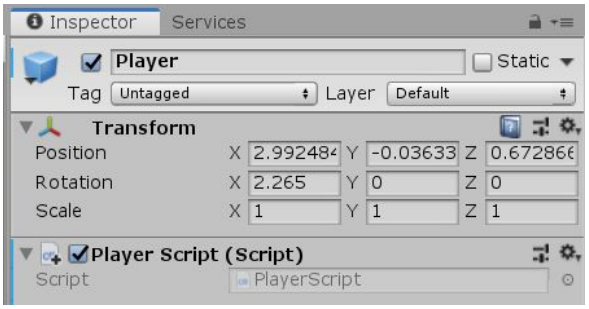


Ilustración 32: Inspector de Unity (Elaboración propia)

De esta manera podemos definir el comportamiento de cualquier objeto a través de código e integrarlo en el objeto al que queramos aplicarle estas acciones de forma rápida y

concreta, ahorrando documentos de código de centenares de líneas y permitiendo su reutilización en cualquier momento. En este proyecto, el funcionamiento del sistema desarrollado se llevará a cabo íntegramente a través de *scripts* en C#.

5.1.4. Cinemática directa

Como deja ver su nombre, la cinemática directa es la técnica contraria a la cinemática inversa para calcular la posición final de una estructura articulada. La cinemática directa usa la rotación del padre de una jerarquía para determinar la posición y orientación de sus hijos así que, en una cadena de huesos, todos los huesos influirán a aquellos que se sitúan por debajo en la jerarquía. Es el método básico que usan los softwares de animación tradicional para permitir a los animadores colocar los personajes en la pose deseada hueso a hueso (ilustración 33). En una cadena secuencial, la solución es simple: solamente existirá un único resultado para cada conjunto de ángulos.

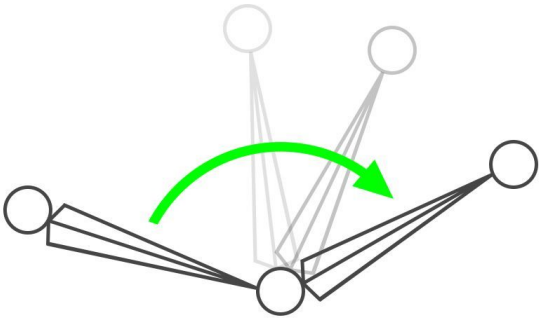


Ilustración 33: Movimiento de cinemática directa (Unreal Engine Documentation)

A partir de una rotación, la cinemática directa te indica la posición final a la que ha llegado. En cinemática inversa, al contrario, se establece el objetivo y ésta calcula el movimiento para alcanzarlo.. Gracias al sistema de *parenting* de *Unity*, que permite establecer jerarquías de padres e hijos de manera sencilla (cuando se asigna un hijo a otro objeto, automáticamente hereda la posición, rotación y escala del padre), resolver el problema de la cinemática directa en este caso no requiere más que un simple gesto. De esta manera podemos construir un esqueleto a base de mallas 3D que actúe como un esqueleto con huesos funcionales que se use en el *rigging* de un personaje.

5.1.5. Cuaterniones

En matemáticas, los cuaterniones son una extensión tetradimensional de los números complejos generados de manera análoga añadiendo las unidades imaginarias *i, j* y *k* a los números reales (ilustración 34), siendo su ecuación:

$$i^2 = j^2 = k^2 = ijk = -1$$

Se utilizan normalmente para describir rotaciones en 3 dimensiones y por ello son tan recurrentes en los gráficos por computador.

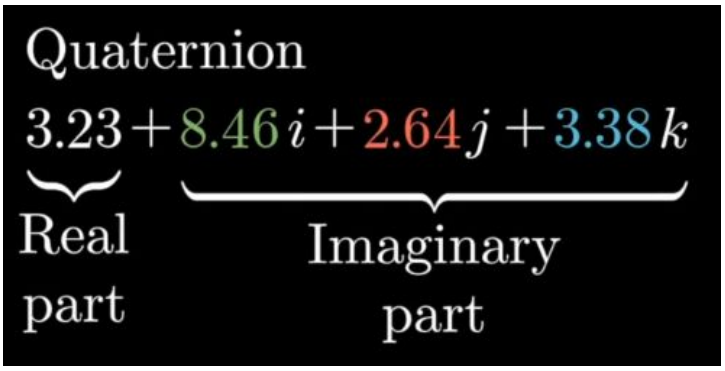


Ilustración 34: Representación de un cuaternión (3Blue1Brown, YouTube)

En *Unity*, donde como se ha visto ya se trabaja con C#, un *quaternion* es una estructura similar al *Vector3* que incluye un ángulo de rotación alrededor del vector. Los elementos individuales son interdependientes y no se pueden modificar individualmente. Esta estructura cuenta con funciones que permiten coger las rotaciones existentes y usarlas para construir nuevas rotaciones a partir de interpolaciones entre ellas.

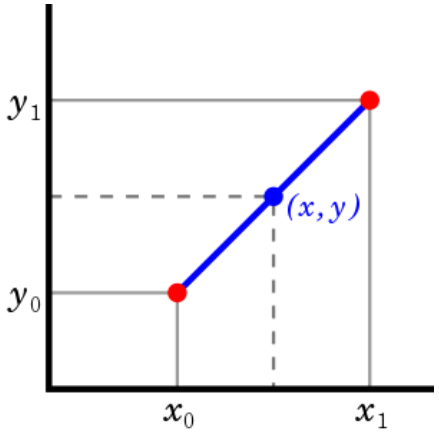
La propiedad más relevante de los *quaternions* es la *Quaternion.identity*, que corresponde a un cuaternión "sin rotación", es decir, el objeto está perfectamente alineado con el mundo o los ejes principales. Entre sus métodos, por otro lado, destaca *Quaternion.Euler*, que devuelve la solución de rotar z grados sobre el eje Z, x grados sobre el eje X e y grados sobre el eje Y.

5.1.6. Interpolaciones

Para resolver animaciones de rotaciones en 3 dimensiones se utiliza la interpolación de cuaterniones. Los *quaternions* de *Unity* cuentan con dos métodos de interpolación diferentes, el *Quaternion.Lerp* y el *Quaternion.Slerp.*

El LERP, conocido también como *Linear Interpolation* (gráfica 2), es un método lineal para determinar el recorrido entre dos puntos conocidos. Dicho recorrido será la línea recta entre ambos puntos, de ahí el nombre del método. El método de *Unity Quaternion.Lerp* interpola entre el cuaternión *a* y el *b* en función del tiempo y normaliza el resultado al final. El parámetro *t* está en el rango entre 0 y 1.

$$\mathrm{Lerp}(p_0, p_1; t) = (1-t)p_0 + tp_1$$



Gráfica 2: Interpolación lineal en función del tiempo.

El SLERP o *Spherical Linear Interpolation*, por otro lado, es un método de interpolación lineal que sigue la superficie de una esfera entre los dos puntos a interpolar, en vez de cortar una línea recta entre ellos. Esto permite una interpolación más suavizada y consistente. Como en el método de *Unity* anterior, el *Quaternion.SLERP* también interpola entre el cuaternión *a* y el *b* en función del tiempo y normaliza el resultado al final.

$$\mathrm{Slerp}(p_0, p_1; t) = \frac{\sin\left[(1-t)\Omega\right]}{\sin\Omega} p_0 + \frac{\sin[t\Omega]}{\sin\Omega} p_1$$

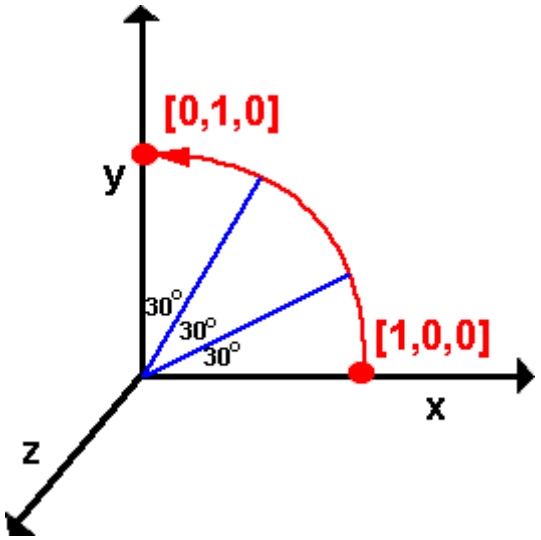


Ilustración 35: Interpolación esférica lineal (Wikipedia)

Por tanto es conveniente trabajar con cuaterniones para resolver sistemas de cinemática inversa en *Unity*, ya que nos permiten representar las rotaciones y traslaciones de los huesos del sistema en el espacio tridimensional de una forma unificada y compacta mientras que los métodos de interpolación de la propia escultura *Quaternion* agilizan el proceso de resolución.

5.2. Diseño del sistema

Una vez definidos los conceptos importantes que aparecerán de manera recurrente durante el desarrollo, es necesario plantear el sistema en papel y qué se requiere que haga. El objetivo es conseguir un sistema básico de cinemática inversa que se pueda utilizar para la animación procedimental de enemigos sencillos. De esta manera, se diferencian dos partes claras: la elaboración del sistema de cinemática inversa (*IK*) y la programación de la animación utilizando dicho sistema.

Para plantear los requisitos que debe cumplir el sistema de *IK* hay que pensar en el tipo de criaturas que animará y sus estructuras esqueléticas. Dado que el objetivo principal del proyecto es el aprendizaje de los principios básicos de la cinemática inversa y la animación procedimental, las animaciones deben ser simples y empezar desde lo más básico, así que significa plantear un enemigo genérico y de movimiento sencillo. Dicho

enemigo, por tanto, será una criatura no humanoide de 2 o más patas formadas por cadenas de 3 huesos (ilustración 36).

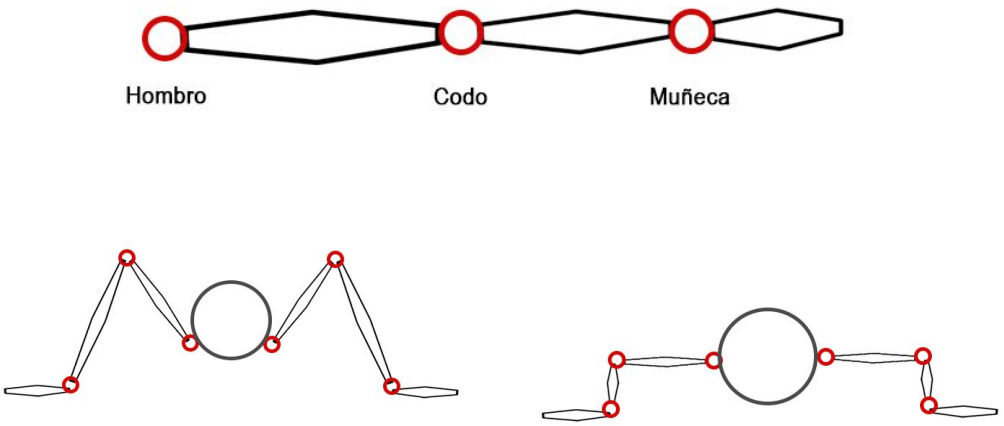


Ilustración 36: Esquema de la cadena de 3 huesos que puede formar distintos tipos de patas (Elaboración propia)

La animación de este enemigo se centrará en su movimiento por el entorno y respecto al jugador, por lo que se descarta la implementación de animaciones de ataque o de interacciones similares. Así que los requisitos imprescindibles son la animación de los pasos de una manera realista y el seguimiento de un objetivo o *target* por el mapa, pudiendo ser un añadido extra la animación del giro del cuello para seguir al *target* tan utilizada en la animación con *IK*.

Así que, en resumen, los objetivos a cumplir a la hora del desarrollo son: para el sistema de cinemática inversa, la resolución del movimiento de cadenas de 3 huesos, y para la animación en sí, el movimiento de los pasos y el movimiento del cuerpo entero por el entorno. Además, se requerirá para la demo final también la implementación de una cámara que permita visualizar el resultado y un elemento objetivo que actúe como el jugador y pueda ser manipulado por el usuario.

5.3. Sistema de Cinemática Inversa Básica

5.3.1. Preparación inicial

5.3.1.1. Entorno

Dado que se va a desarrollar a través de *Unity*, el primer paso es preparar el proyecto para poder trabajar sobre él. A la hora de crear un nuevo proyecto, el propio motor gráfico ofrece una serie de plantillas para establecer una base. En este caso se utiliza una plantilla de proyecto 3D que cuenta de primeras con una escena sin título (el contenedor de los objetos del juego, cada escena representa una fase o nivel) completamente vacía a excepción de una cámara llamada *Main Camera* y una luz llamada *Directional Light* (ilustración 37).

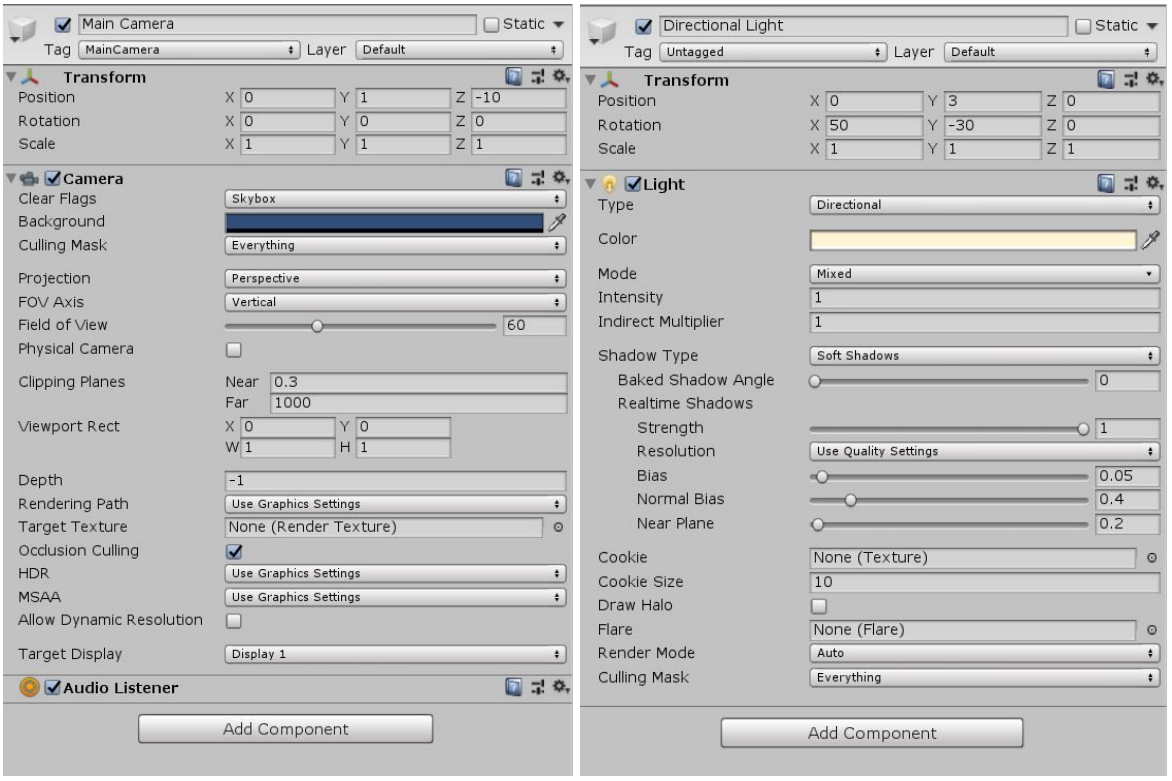


Ilustración 37: Propiedades de la Main Camera y la Directional Light de Unity (Elaboración propia)

También será conveniente colocar un suelo para la posterior implementación de la animación. Basta con usar un objeto 3D como un cubo con una altura pequeña y ya se tendría un suelo funcional.

5.3.1.2. Esqueleto

A partir de aquí ya se puede empezar a trabajar. Para la implementación del sistema de cinemática inversa lo más importante es tener un esqueleto sobre el que se aplique el susodicho. Como se ha comentado con anterioridad, a pesar de que *Unity* permite la importación de modelos con *rigging* y trabajar con ellos, su sistema de jerarquías y el hecho de que no distinga entre huesos de un esqueleto 3D y formas básicas hace que sea muy sencilla la creación de un esqueleto de prueba con el que trabajar.

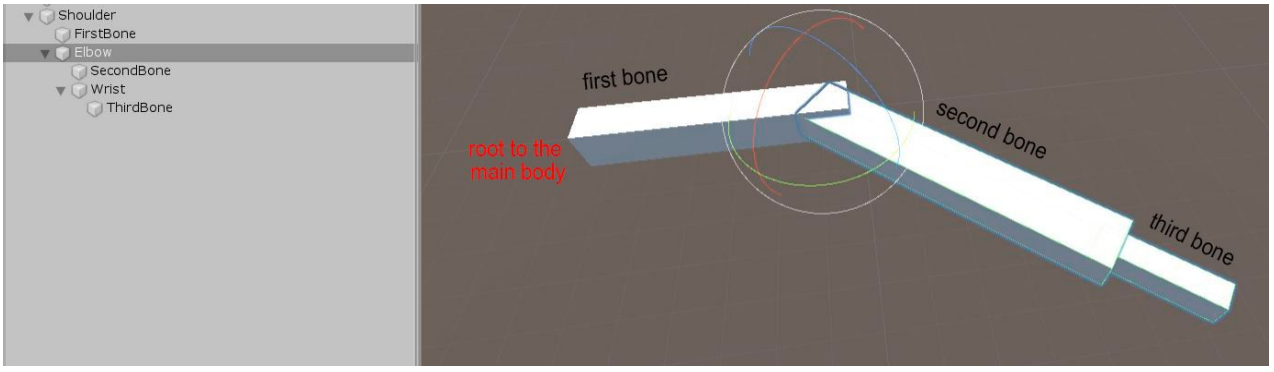


Ilustración 38: Estructura de 3 huesos creada a base de cubos en Unity (Elaboración propia)

Una vez creado el esqueleto de prueba con sus tres huesos (ilustración 38), hay que establecer un objetivo o *target*, que será el punto hacia el que la cadena intentará llegar y dicta cómo los huesos deben rotar. Dado que lo importante es la posición 3D del *target*, basta con crear una esfera o cualquier otra figura en tres dimensiones que sirva como tal.

5.3.2. Implementación

De esta manera, con los elementos necesarios en escena, es hora de crear el primer *script*, que corresponderá al sistema de cinemática inversa. Como se ha visto también con anterioridad, una vez creado, el *script* cuenta con una clase *MonoBehaviour* que contendrá

el código *C#.* El primer paso es establecer las variables necesarias (código 1) y un punto importante es el uso de *SerializeField*, un atributo que permite exponer una variable en el inspector de *Unity* manteniéndola privada. Esto servirá para poder usar el *script* de manera sencilla una vez implementado.

```
//---USER SETS THESE PARAMETERS FROM THE UNITY EDITOR--//
//target the chain should bent to
[SerializeField] Transform target;
[SerializeField] Transform pole;

//the legs are formed by a chain of 3 bones
[SerializeField] Transform firstBone;
[SerializeField] Transform secondBone;
[SerializeField] Transform thirdBone;
```

Código 1: Variables principales del sistema.

Como se puede ver en la ilustración 35, las variables tipo *Transform* (componente de todo *GameObject*) se usan para manipular la posición, rotación y escala de un objeto. Gracias al *SerializeField* se podrá asignar el componente *Transform* de cada elemento desde el editor cada vez que se quiera usar el sistema, haciendo posible su uso en distintas ocasiones y escenas.

A continuación, el sistema contará con tres funciones. La plantilla de *script* que viene implícita al crear el susodicho incluye las funciones *Update()* y *Start()*. En este caso, la función *Start()* no es necesaria, por lo que será sustituida por la función *OnEnable().* El único uso de esta sencilla función es la comprobación de que todos los parámetros necesarios para el sistema hayan sido introducidos. De no haber sido así, se mostrará el siguiente mensaje de error (código 2).

```
Debug.LogError("Bones not initialized. Please, link the bones.", this);
```

Código 2: Error que salta cuando los huesos no están bien encadenados (elaboración propia)

Esta función propia de la clase *MonoBehaviuor* es llamada cuando el objeto al que están adjuntos pasa a estar activo. A diferencia que la función *Start(),* que se llama una sola

vez por objeto y se usa especialmente para la inicialización de estos, la función *OnEnable()* se llamará cada vez que el *script* se active y, por tanto, es más útil en este caso.

En cuanto a la función *Update(),* esta será sustituida por *LateUpdate()*, ya que esto permitirá a otros sistemas actualizar el entorno antes, permitiendo a la función de cinemática inversa a adaptarse antes del frame. Esta función es llamada una vez por frame.

La tercera y última función contendrá las operaciones necesarias para la resolución del movimiento de los huesos a través de *IK*. Existen múltiples enfoques para resolver sistemas de cinemática inversa. Uno de los más conocidos y usados es el algoritmo CCD o *Cyclic Coordinate Descent*, utilizado especialmente en brazos robóticos, cuyo principio básico es aplicar las rotaciones desde el hueso más cercano al objetivo hasta el hueso de la base. Es fácil de implementar pero requiere una serie de iteraciones antes de llegar a una solución y puede causar rotaciones erróneas de las articulaciones. En este caso particular no se profundizará en aquel algoritmo ya que se ha optado por trabajar con algo más sencillo debido a las dimensiones del sistema. A través de la ayuda de las funciones de la clase *Quaternion* de *Unity* resulta sencillo implementar un algoritmo de triangulación. Entre los diferentes algoritmos normalmente utilizados, la triangulación de ángulos es un método sencillo que ofrece soluciones sin grandes rotaciones y es rápido y útil en el ámbito de la animación. Se utilizan las propiedades de los triángulos para calcular con certeza los ángulos requeridos para mover los huesos de la cadena cinemática hacia el objetivo. Los huesos mayores y más cercanos a la base rotan primero, seguidos por el resto de la cadena hasta llegar al último, de manera que se genera un movimiento más fluido y natural.

En este caso, dado que se va a utilizar para la interpolación del movimiento de patas, a pesar de contar con una cadena de 3 huesos, realmente el sistema solo tiene que resolver los movimientos de 2 de ellos, dado que el último hueso corresponde a la pata que irá apoyada en el suelo. De esta manera, basta con un simple algoritmo de trigonometría basado en la Ley de los Cosenos para resolver los ángulos (ilustración 39) de rotación de los dos huesos correspondientes a lo que sería la pierna.

$$c^2 = a^2 + b^2 - 2ab\cos\gamma$$

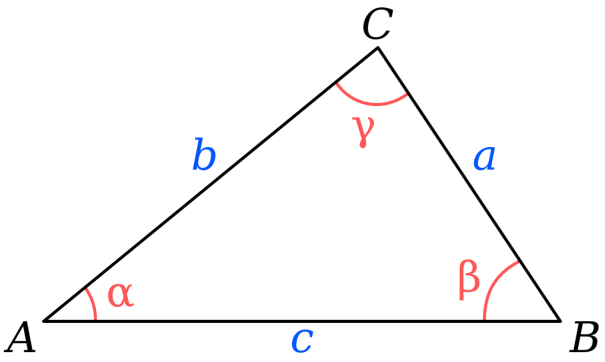


Ilustración 39: Triángulo con sus ángulos correspondientes (Wikipedia)

Antes de empezar se necesitan una serie de datos para la aplicación de la triangulación. Primero se añaden a la lista de variables una serie de campos serializados para la introducción de los ángulos de Euler de cada hueso. Los ángulos de Euler son 3 coordenadas angulares que sirven para especificar la orientación de un sistema de referencia respecto a otro de ejes fijos, en este caso para especificar la orientación de cada hueso de la cadena pues, como se ve en la ilustración 32, con una cadena de 3 huesos se pueden conformar piernas o patas de distintas formas con sus huesos en distintas orientaciones (ilustración 40). De esta manera tendremos la orientación local de cada hueso.

Una vez establecidas esas variables con el atributo *SerializeField*, es necesario identificar los vectores de movimiento hacia el *target* y hacia el *pole.* El *target* es el punto al que la cadena está intentando llegar, mientras que el *pole* es el punto sobre el que la cadena se dobla para llegar al *target*. Se utiliza como punto de control para la dirección y rotación del codo. También se ha de calcular los vectores del hueso principal o *FirstBone* hacia el *pole*, el *target* y el hueso adyacente o *SecondBone.* La operación es simple y consiste en una sencilla resta de los vectores posición de los distintos *GameObjects* (los huesos, el *target* y el *pole).*

Una vez hecho esto, con la función *Distance(Vector3 a, Vector3 b)* se calcula la longitud tanto de los huesos *FirstBone* y *SecondBone* y de la cadena completa.

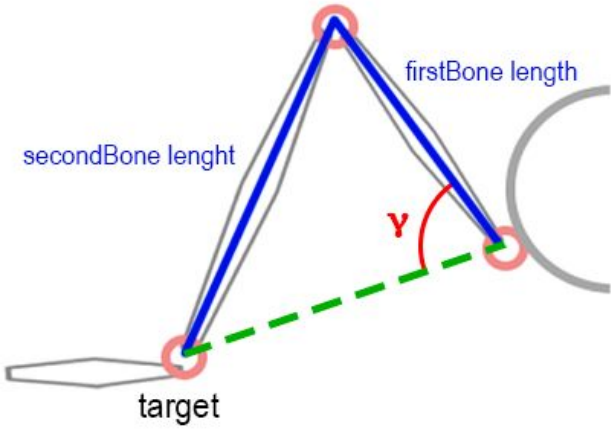


Ilustración 40: Triangulación de los huesos de la pata (Elaboración propia).

Con todos los datos necesarios calculados para empezar, el primer paso es alinear el hueso de la base o *FirstBone* con el *target,* calculando su rotación en el mundo (*world rotation*) y rotación local (relativa a la *world rotation* del padre). Ya que se trabaja con los *Quaternion* de *Unity*, es tan simple como usar las funciones de esta clase. Para calcular la rotación global del hueso alineado, se usa la función *LookRotation(Vector3 f, Vector3 u)* que crea una rotación con las direcciones especificadas en sus parámetros. El primer vector es la dirección hacia la que mirar, en este caso el vector desde el *FirstBone* hacia el *target*, al que se ha llamado *towardTarget*. El segundo vector es aquel que define en que dirección global se orienta, en este caso *towardPole*. El eje Z se alineará con el vector *towardTarget*, mientras que el eje X se alineará con el producto vectorial entre los vectores *towardTarget* y *towardPole* (**código 3**); por último, el eje Y se alineará con el producto vectorial de los ejes Z y X.

```
firstBone.rotation = Quaternion.LookRotation(towardTarget, towardPole);
firstBone.localRotation *= Quaternion.Euler(firstBoneEulerAngle);
```

Código 3: Código de alineación del primer hueso de la cadena hacia el target (Elaboración propia)

Para calcular la rotación local, por otra parte, se usa otra función de la clase *Quaternion, Quaternion.Euler(Vector3 euler).* Esta función rota *z* grados el eje Z, x grados el eje X e y grados el eje Y, siendo *Vector3 euler(x, y, z).* De esta manera, la rotación local

del hueso alineado resultará en la multiplicación de su rotación local inicial por el *firstBoneEulerAngle* establecido desde el editor.

Antes de calcular el ángulo de rotación del primer hueso es necesario limitar la distancia entre éste y la posición del *target* (vector *targetDistance)* a un número menor que la distancia total de la cadena de huesos para evitar que se formen triángulos inválidos. Esto se consigue con la función del cálculo del valor mínimo *Mathf.Min(float a, float b)* entre *targetDistance* y *boneChainLength. Mathf* es una colección de funciones matemáticas comunes.

Es hora de resolver la ecuación de la Ley de los Cosenos (código 4) de manera que se obtenga el ángulo de rotación de *firstBone*. De esta manera, se debe resolver la siguiente ecuación:

$$cos\gamma = \frac{a^2 + b^2 - c^2}{2ab}$$

$$\gamma = arccos(cos\gamma)$$

```
var adj = ((firstBoneLength * firstBoneLength) + (targetDistance * targetDistance)
    - (secondBoneLength * secondBoneLength)) / (2 * firstBoneLength * targetDistance);
var angle = Mathf.Acos(adj) * Mathf.Rad2Deg;
```

Código 4: Código de la ecuación de la Ley de los Cosenos (Elaboración propia)

Antes de aplicar la rotación final sobre el hueso, hay que tener en cuenta el *pole* establecido, que ayuda a controlar la rotación del codo (la articulación que une el *firstBone* y el *secondBone*), por lo que se ejerce una rotación sobre el vector ortogonal entre el vector hacia el *pole* (*towardsPole*) y hacia el segundo hueso (*towardsSecondBone*). Para ello se calcula el producto vectorial (ilustración 41) entre ambos gracias a la función *Vector3.Cross(Vector3 a, Vector3 b).*

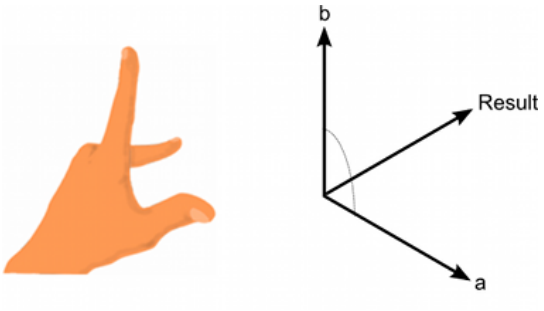


Ilustración 41: Producto vectorial (Unity Documentation)

Una vez se ha tenido en cuenta la dirección del *pole*, se aplica la rotación con la función *Transform.RotateAround(Vector3 point, Vector3 axis, float angle)*, que gira el *transform* del hueso sobre el punto que indican las coordenadas del propio hueso alrededor de la solución del producto vectorial hecho anteriormente, que actúa como eje, con el ángulo calculado con anterioridad.

Una vez rotado el primer hueso, dado que es una cadena, el resto se moverán con éste. De esta manera, lo único que quedaría sería orientar el segundo hueso *secondBone* hacia el *target*, como se hizo con el primero anteriormente (código 5).

```
var secondBoneTargetRot = Quaternion.LookRotation(target.position - secondBone.position, cross);
secondBoneTargetRot *= Quaternion.Euler(secondBoneEulerAngle);
secondBone.rotation = secondBoneTargetRot;
```

Código 5: Código de la alineación del secondBone con el target (Elaboración propia)

Por último, solamente falta aplicar la rotación final del último hueso, aquel que toca el suelo, el pie o *thirdBone*. Esto se consigue de manera rápida, dado que la rotación con respecto al mundo del *thirdBone* será igual a la del *target,* y la rotación local de éste se calcula fácilmente con la ecuación vista anteriormente *Quaternion.Euler(Vector3 euler)*.

De esta manera se obtiene un sistema simple de cinemática inversa para la resolución de movimientos de dos huesos, aplicable a las patas de una criatura o a las piernas de un humano. A continuación, se verá cómo se implementa la animación de movimiento que usa dicho sistema de *IK*.

5.4. Animación

Una vez implementado el sistema de cinemática inversa que resuelve el movimiento de cadenas de 3 huesos, corresponde aplicarlo a una animación, pues no sirve de mucho por sí solo. Como ya se ha planteado, el objetivo es implementar una animación de movimiento para una criatura enemiga que sea capaz de seguir al jugador. De esta manera, se dividirá la implementación en tres etapas: el seguimiento del objetivo, es decir, la implementación del movimiento del cuerpo de la criatura por el mapa siguiendo al *target*; la animación de los pasos, donde actuará el sistema de *IK*; y la animación para que un hueso, como puede ser la cabeza, mire en la dirección del *target*.

Antes que nada, se necesita un esqueleto completo para poder implementar las animaciones. Como se ha hecho antes, se puede realizar a partir de jerarquías de objetos 3D e incluso se le podría asignar a una malla 3D importada en un futuro.

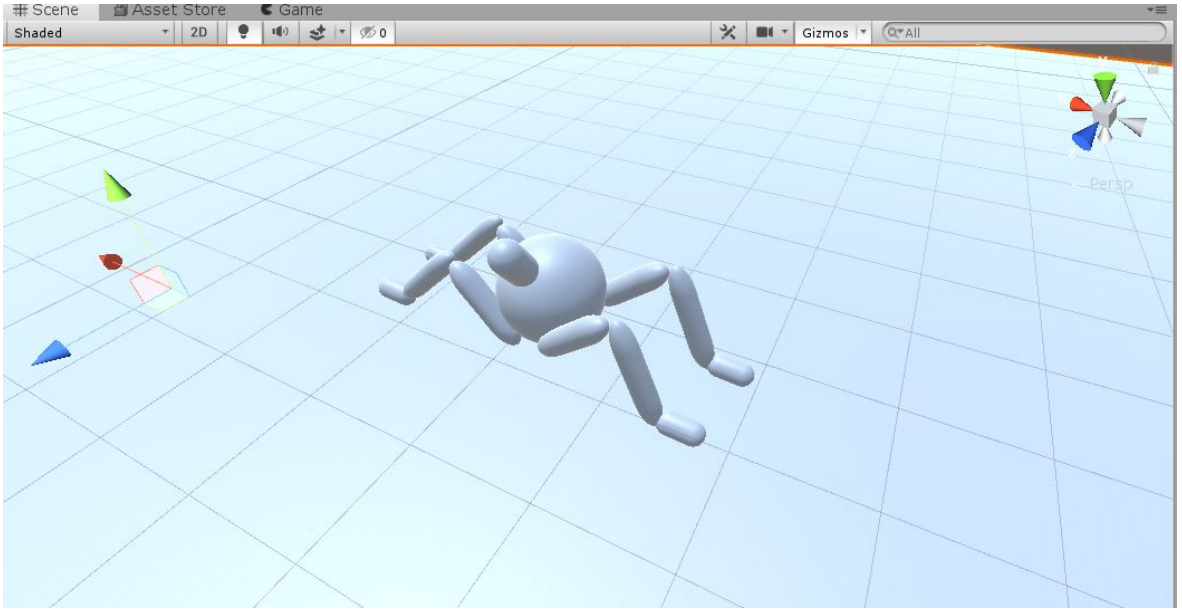


Ilustración 42: Esqueleto de prueba generado a base de formas 3D (Elaboración propia)

A continuación es conveniente crear un nuevo *script* correspondiente al control de las animaciones de movimiento, que se llamará *AnimationController*. Se genera de la misma manera que se ha visto con anterioridad. Una vez listo el esqueleto y creado el *script,* falta añadir un objetivo o *main target* que la criatura vaya a seguir. De nuevo, basta

con crear una forma 3D como un cubo, esfera o cápsula, que sirva como objetivo (que más adelante el usuario podrá manipular para probar el funcionamiento del sistema).

5.4.1. Seguimiento de un objetivo

El principio detrás de la implementación de este movimiento es tratar la criatura como un solo objeto sin preocuparse por los distintos huesos del esqueleto o elementos adicionales. Esto se puede hacer de manera sencilla, una vez más, gracias al sistema de *parenting* de *Unity*.

El primer paso es definir la velocidad del movimiento, tanto para actualizar la posición (*movementSpeed*) como la rotación (*rotationSpeed*), ya que la criatura podrá girar. Para mayor customización y versatilidad del sistema, estas velocidades podrán ser modificadas desde el editor por el usuario, por lo que se usan de nuevo variables con el atributo *SerializeField* (código 6). Para añadir un componente de realismo y naturalidad, se usarán también variables de aceleración. Por último, es importante establecer una distancia máxima y mínima del *main target* que determinen el rango de reacción de la criatura, así como un ángulo máximo de giro respecto al *main target* para asegurar que no avance cuando no pueda verlo.

```
//---USER SETS THESE PARAMETERS FROM THE UNITY EDITOR--//
//target that the creature will follow
[SerializeField] Transform mainTarget; //we will call it mainTarget to separate from the target of the IK system
//Movement and turn velocity of the creature
[SerializeField] float rotationSpeed;
[SerializeField] float movementSpeed;
[SerializeField] float rotationAccel;
[SerializeField] float movementAccel;

//range of distance and angle from the target to not to lose it
//these could be totally private but let's keep them serialized for customization
[SerializeField] float minDistance = 5f;
[SerializeField] float maxDistance = 7.5f;
[SerializeField] float maxAngle = 25f; //if the creature is above this angle from the target, it starts turning

Vector3 currentVel;
float currentAngularVel;
```

Código 6: Variables necesarias para la implementación del movimiento de la criatura por el plano XZ.

Una vez establecidas las variables, se crea la función *RootMovement()* que contendrá todas las operaciones necesarias para realizar el movimiento de la criatura. Primero se realiza la rotación del cuerpo y para ello es necesario obtener primero el vector de movimiento entre la criatura y el *main target*. Una vez hecho eso, dado que solamente es necesario trabajar en 2D porque la criatura se va a desplazar por una superficie plana, se calcula la proyección de dicho vector sobre el plano XZ a través de la función *Vector3.ProjectOnPlane(Vector3 a, Vector3 planeNormal),* que proyecta un vector en un plano definido por una normal ortogonal al plano. Obtenido el vector en el plano XZ, calculamos el ángulo entre la posición de la criatura y la posición del *main target* con la función *Vector3.SignedAngle (Vector3 a, Vector3 b, Vector3 axis),* que devuelve el ángulo en grados entre los dos primeros vectores alrededor del tercer vector, que actúa como eje (en este caso, será el eje Y) (código 7).

```
//vector of movement toward the main target from the creature's position
Vector3 towardMainTarget = mainTarget.position - transform.position;
Vector3 towardTargetProj = Vector3.ProjectOnPlane(towardMainTarget, transform.up);
//we get the angle from the criature's position to the main target direction
float angleMainTarget = Vector3.SignedAngle(transform.forward, towardTargetProj, transform.up);
```

Código 7: Código del cálculo de vectores (Elaboración propia)

Hecho esto, solamente falta comprobar la dirección hacia la que tendrá que girar la criatura para orientarse antes de aplicar la rotación. Dado que *Unity* por defecto funciona en el sentido de las agujas del reloj, si el ángulo con el *main target* es positivo, significa que debe girar a la derecha; si es negativo, lo hace hacia la izquierda. A la hora de rotar, usando una interpolación *LERP*, cuyo funcionamiento se ha visto con anterioridad, se consigue suavizar el cambio de velocidad de manera que el movimiento sea más natural (código 8). *Unity* cuenta con su propia función para la interpolación lineal, *Mathf.Lerp(float a, float b, float t).* Por último, basta con aplicar la rotación entorno al eje Y.

```
//using lerp interpolation to smooth the velocity change
currentAngularVel = Mathf.Lerp(currentAngularVel, targetAngularVel, 1-Mathf.Exp(-rotationAccel * Time.deltaTime));
//rotate around Y axis
transform.Rotate(0, Time.deltaTime * currentAngularVel, 0, Space.World);
```

Código 8: Código de animación de la rotación del cuerpo a partir de una interpolación lineal (Elaboración propia)

Una vez solucionado el giro respecto al *main target*, solo queda implementar el desplazamiento hacia éste. Con la distancia máxima y mínima a la que puede estar la criatura del objetivo establecida en las variables, solo hace falta implementar que la criatura se acerque cuando esté demasiado lejos y se aleje cuando esté demasiado cerca. Una vez más, se usa la interpolación lineal (código 9) para conseguir un movimiento natural y se aplica el movimiento al eje XZ en función del *deltaTime.*

```
//using lerp interpolation to smooth the velocity change
currentVel = Vector3.Lerp(currentVel, mainTargetVel, 1 - Mathf.Exp(-movementAccel * Time.deltaTime));
//apply the movement to XZ axis
transform.position += currentVel * Time.deltaTime;
```

Código 9: Código de animación del desplazamiento del cuerpo a partir de una interpolación lineal (Elaboración propia)

5.4.2. Animación de giro de un hueso

La animación de un hueso para que siga el movimiento del *main target* y parezca que mira hacia éste sigue unos principios similares al de la animación del desplazamiento del cuerpo completo. Se seguirá trabajando en el mismo *script*, solo que en una función nueva, *HeadMovement()*, que será llamada desde la función de *LateUpdate()*. De nuevo, el primer paso es definir las variables necesarias (código 10), en este caso es necesario añadir el hueso que se desea mover, la velocidad a la que se moverá y, una variable importante, el ángulo de rotación máxima del hueso. Esta última variable sirve de restricción de movimiento para impedir (o permitir) que la cabeza de la criatura gire 360º, cosa que será poco realista.

```
//HEAD MOVEMENT
[SerializeField] Transform headBone;
[SerializeField] float headMaxAngle;
[SerializeField] float headSpeed;
```

Código 10: Variables necesarias para la implementación del seguimiento de un hueso (Elaboración propia)

Una vez establecidas las variables, en la función *HeadMovement()*, es necesario guardar antes que nada la rotación actual del hueso, pues se sobreescribirá cada vez que se llame a la función. Una vez guardada ésta en la variable *headRotation,* se resetea la rotación local del hueso para aplicarle más tarde la rotación final. Esto se hace usando *Quaternion.identity*, que es igual a 0. A continuación se calcula el vector dirección que va desde el hueso que se moverá hacia el *main target* al que está mirando. Este resultado será un vector dirección global, es decir, un vector en el sistema de coordenadas de la escena completo, de manera que es necesario transformarlo a un vector dirección local, es decir, un vector relativo al padre padre en la jerarquía de objetos. Esta acción se realiza fácilmente con la función *Vector3.InverseTransform Direction(Vector3 direction).*

Una vez se tiene la dirección local del *main target*, para limitar el ángulo de giro, ya que normalmente una articulación no gira 360º y se busca crear un movimiento realista, se usa la función *Vector3.RotateTowards(Vector3 current, Vector3 target, float maxRadiansDelta, float maxMagnitudeDelta)*, donde el último parámetro representa una longitud que no se tendrá en cuenta en este caso. La rotación que genera se utilizará para obtener la rotación que debe realizar el hueso para mirar hacia el *main target* con la función *Quaternion.LookRotation(Vector3 forwards, Vector3 upwards),* ya utilizada anteriormente.

Por último, de nuevo se aplicará una interpolación para realizar un movimiento suavizado, este caso será una interpolación esférica o *SLERP* (código 11), más consistente que la lineal y que devuelve un resultado más natural en este caso.

```
//using slerp interpolation to smooth the rotation
headBone.localRotation = Quaternion.LookRotation(targetLocalDirection, Vector3.up);
headBone.rotation = Quaternion.Slerp(headBone.rotation, targetLocalRotation,
    1 - Mathf.Exp(-headSpeed * Time.deltaTime));
```

Código 11: Código de animación de la rotación del hueso a partir de una interpolación esférica (Elaboración propia)

5.4.3. Animación de pasos

El paso restante para terminar el movimiento es animar las patas de la criatura. Como se ha visto a la hora de implementar el sistema de cinemática inversa, la estructura de las patas es muy sencilla (ilustración 37). El pie o tercer hueso va alineado al plano XZ, ya que está apoyado en el suelo. Así que, en este caso, remitiendo a los elementos *target* y *pole* vistos en el sistema de *IK*, el *target* estará en la misma posición que el pie, mientras que el *pole* controlará la dirección correspondiente a la rodilla de la pata (o al codo de la cadena) para una dirección consistente. El primero será independiente del esqueleto de la criatura y su posición y rotación será global, pero el segundo formará parte de la jerarquía y rotará con el cuerpo, así como se mantendrá en la posición relativa a la cadena de huesos.

Al ser un sistema simple de cinemática inversa, el ciclo de caminar también será simple y se basará en un movimiento sencillo desde A hasta B. Para ello, primero es importante, una vez más, establecer las variables con las que se va a trabajar. El principio de esta animación recae en el uso de una posición origen para cada pata. Esta es una posición en el centro del rango al alcance de la cadena de huesos de la pata y la posición origen de cada una de ellas. Es importante introducir dicha posición origen en la jerarquía, para que en todo momento su rotación sea la misma que la del cuerpo y se moverá y rotará sobre el plano con éste. Con esta posición se decidirá cuándo y dónde mover el *target*, pues si éste queda fuera de rango, se llamará a la función que resuelva el paso para devolver el *target* a la posición inicial, generando así una animación natural de las patas.

Para mantener el orden y la estructura de componentes, ya que el *script* a implementar solamente corresponde a las patas, se crea un componente nuevo llamado *WalkController* que formará parte del *target* de cada pata. Las variables necesarias (código 12) serán, primero, la susodicha posición de origen de la pata, así como la distancia máxima que puede alejarse la pata del origen (o, en otras palabras, la longitud máxima del paso) y la duración del paso en tiempo. Se añade también una última variable cuyo rango se permite variar entre 1 y 0 gracias al atributo *Range[x, y]* que establece un porcentaje de extensión desde la posición origen para crear un rango en el que se pueda extender más el

movimiento. Estableciendo el atributo *Range* se permite también que esta extensión sea nula. Por último, se establece una variable que indique si la pata se está moviendo o no en ese momento.

```
//---USER SETS THESE PARAMETERS FROM THE UNITY EDITOR--//
//Constraints for the leg movement
[SerializeField] Transform legOrigin;
//if we exceed this distance, next move will try to succeed
[SerializeField] float maxStepDistance;
[SerializeField] float stepDuration; //in time
//fraction for how far of the origin the leg can move
[SerializeField, Range(0, 1)] float stepRangeFraction;

public bool isMoving; //if the leg is moving
```

Código 12: Variables necesarias para la animación de los pasos de la criatura (Elaboración propia)

En la función *Update()* creada de manera predeterminada, antes que nada, se comprueba si la pata está en movimiento con la última variable *bool* que se ha establecido con anterioridad, *isMoving* y, de ser cierto, no empieza otro movimiento. De no ser así, comprueba si la pata está demasiado lejos en posición o rotación, primero calculando la distancia entre la posición de la pata y la posición del origen con la función *Vector3.Distance(Vector3 a, Vector3 b)* vista anteriormente, y después comprobando si es mayor que la longitud máxima del paso. De cumplirse esta última condición, se iniciará el movimiento de la pata.

A continuación es necesario implementar el movimiento del *target* respecto al origen. Esto se hace mediante una corrutina, pues los pasos de cada pata no serán instantáneos, si no, no quedaría un movimiento realista. Una corrutina es una función que tiene la habilidad de pausar la ejecución y devolver el control a *Unity*, pero permitiendo continuar desde el punto que se había dejado la ejecución en el siguiente fotograma. Dicha función es declarada con un retorno de tipo *IEnumerator* y con la declaración *yield return null* incluída en alguna parte del cuerpo de la función, indicando el punto en el que la ejecución se pausará y será retomada en el siguiente fotograma. Las corrutinas se llaman a partir de la función *StartCoroutine(IEnumeratior routine)*.

Una vez declarada la función, el primer paso dentro de ésta es indicar que la pata se está moviendo a través de la variable *isMoving*. A continuación se guarda tanto la posición como la rotación en ese fotograma de la pata, al igual que la rotación final, que será la del punto de origen.

Después, con ayuda del porcentaje seleccionado a través de la variable *stepRangeFraction* desde el editor de *Unity*, se calcula el rango en el que puede estar el *target* alrededor del origen, primero multiplicando dicho porcentaje por la distancia máxima del paso y luego aplicándolo al vector de posición que va desde el origen hacia la pata (código 13). Dicho rango solamente se aplicará en el eje XZ, así que se proyecta dicho vector sobre dicho plano correspondiente con el suelo con la función *Vector3.ProjectOnPlane(Vector3 a, Vector3 planeNormal),* también vista anteriormente. Una vez hecho esto, se guarda posición final de la pata.

```
float stepRange = maxStepDistance * stepRangeFraction;
Vector3 stepRangeVector = towardOrigin * stepRange;
//we restrict the vector to be leveled with the ground projecting it on the XZ plane
stepRangeVector = Vector3.ProjectOnPlane(stepRangeVector, Vector3.up);
//apply the total distance
Vector3 endPosition = legOrigin.position + stepRangeVector;
```

Código 13: Código del cálculo de la posición final de la pata (Elaboración propia)

Pero el movimiento de un paso no solamente ocurre en el plano XZ, sino que también en el eje Y, pues en un movimiento real de caminar, la pata se levanta. Este levantamiento se calcula, primero, calculando un vector punto medio entre la posición inicial y la posición final calculada anteriormente, donde el pie estará levantado. A dicho vector se le añade entonces en el eje Y la altura del paso, que en este caso es la mitad de la distancia del paso dado.

Por último, para completar el movimiento, solo hay que interpolar entre posición y rotación iniciales y finales con las funciones de interpolación vistas con anterioridad (código 14). En el caso de la posición, se ha optado por una interpolación basada en una

Curva de Bézier o, lo que es lo mismo, tres interpolaciones lineales anidadas, pues genera un movimiento mucho más fluído en este caso. Para la rotación, basta con un *SLERP*.

```
do
{
    timePassed += Time.deltaTime;
    float normalTime = timePassed / stepDuration;

    //Solving the position and the rotation while interpolating for smooth movement
    //Quadratic Bezier curve or anidated Lerps
    transform.position = Vector3.Lerp(Vector3.Lerp(startPosition, centerPosition, normalTime),
        Vector3.Lerp(centerPosition, endPosition, normalTime), normalTime);
    //single SLERP
    transform.rotation = Quaternion.Slerp(startRotation, endRotation, normalTime);

    yield return null;
}
while (timePassed < stepDuration);
```

Código 14: Código de la animación de la pata a través de interpolaciones (Elaboración propia)

Una vez terminado este paso, lo único que queda es implementar un objetivo para la criatura para que pueda interactuar con éste y que permita al usuario probar la animación controlando dicho objetivo.

5.5. Input y montaje del sistema

Una vez más, la implementación del *input* o entrada de usuario en *Unity* es relativamente simple. Basta con crear un objeto 3D de la forma deseada y un *script* para el componente *PlayerInput.* El objetivo es permitir al usuario mover el objeto que servirá de objetivo para la criatura por la escena con el clic del ratón.

Las variables necesarias que deberá introducir el usuario para utilizar un objeto 3D como objetivo serán el *Transform* del mismo objeto y una velocidad a la que desee que se mueva dicho objetivo. Además, son necesarios para la implementación un plano XZ y la variable de la posición del objetivo, que será actualizada a cada frame desde el *Update()*.

Es en la función *Update()*, de hecho, en la que ocurrirá todo (código 15). Primero hay que determinar el destino del objetivo a través de la posición del ratón en el plano XZ.

Se recoge el *click* del ratón a través de la función *Input.GetMouseButton(0)* y se establece un *ray*, es decir, una línea infinita que va en una dirección, desde la cámara hasta el punto de la pantalla indicado con el ratón gracias al método *Camera.ScreenPointToRay(Vector3 a)*. Una vez hecho esto, se calcula la intersección del *ray* con el plano para obtener la posición donde tiene que moverse el objetivo. Por último, con una interpolación lineal, se mueve el objetivo a dicho punto en función de la velocidad elegida anteriormente.

```
void Update()
{
    //Set the target destination to mouse position on XZ plane
    if (Input.GetMouseButton(0))
    {
        var ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if(floor.Raycast(ray, out float hitDistance))
        {
            playerTargetPosition = ray.origin + ray.direction * hitDistance;
        }
    }
    //Move target to mouse click position
    playerTarget.position = Vector3.Lerp(playerTarget.position,
        playerTargetPosition, 1 - Mathf.Exp(-speed * Time.deltaTime));

}
```

Código 15: Código del movimiento del objeto 3D objetivo al lugar donde ha llegado el input del ratón (Elaboración propia)

Con esto, el sistema de animación ya está listo para ser usado en *Unity*. Lo único que falta es realizar los *links* entre *scripts* y sus elementos correspondientes que se han visto anteriormente, aplicar los parámetros deseados y la criatura ya estaría animada.

5.6. Demo final

Como se puede ver en las ilustraciones 43, 44, 45 y 46, el sistema es aplicable a criaturas con distintos números de patas y es posible elegir cuál es el hueso que se desea que mire hacia el objetivo, como una cabeza (ilustración 43) o una cola (ilustración 46).

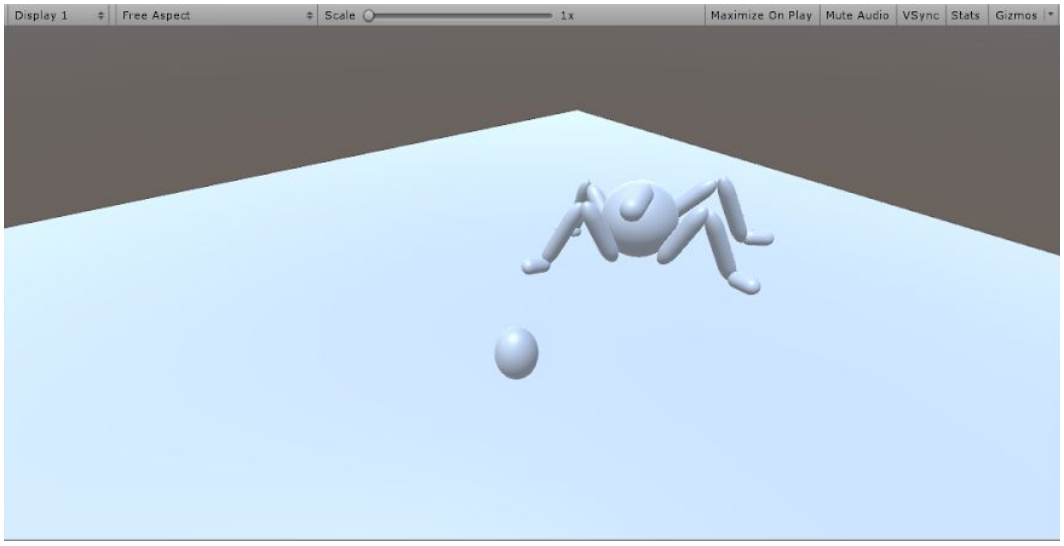


Ilustración 43: Criatura cuadrúpeda de la primera prueba caminando hacia el objetivo (Elaboración propia)

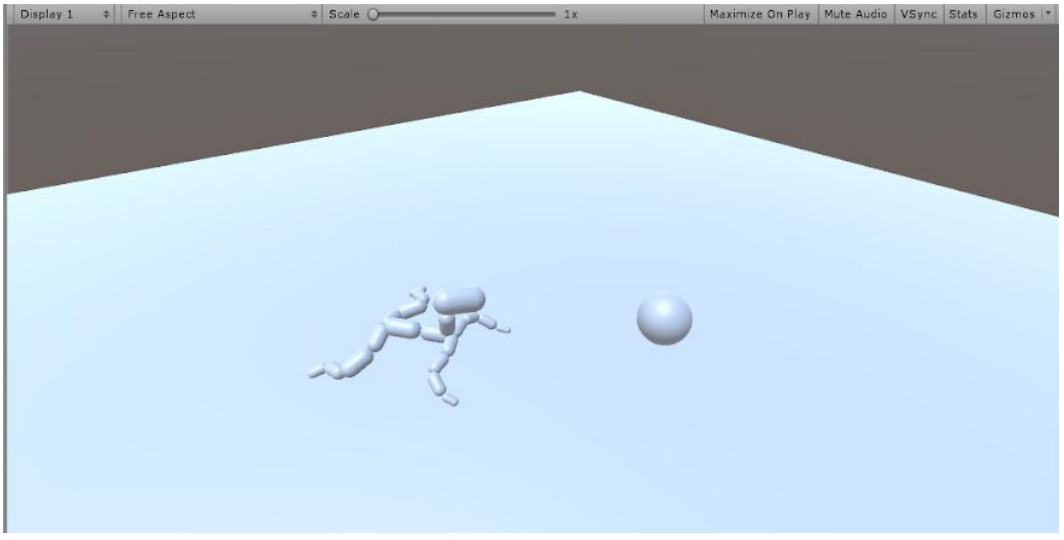


Ilustración 44: Criatura cuadrúpeda de segunda prueba caminando hacia el objetivo (Elaboración propia)

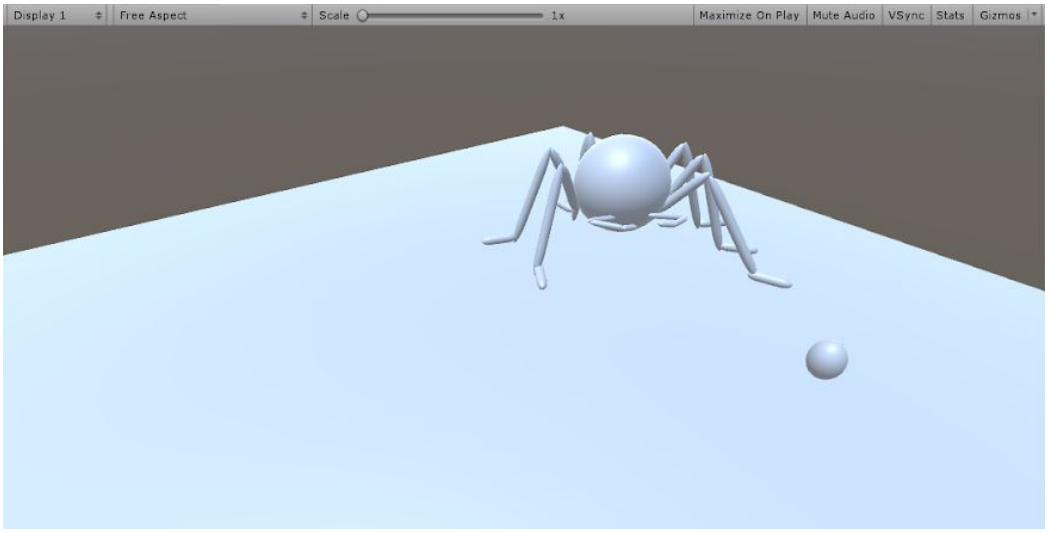


Ilustración 45: Criatura con cuerpo arácnido caminando hacia el objetivo (Elaboración propia)

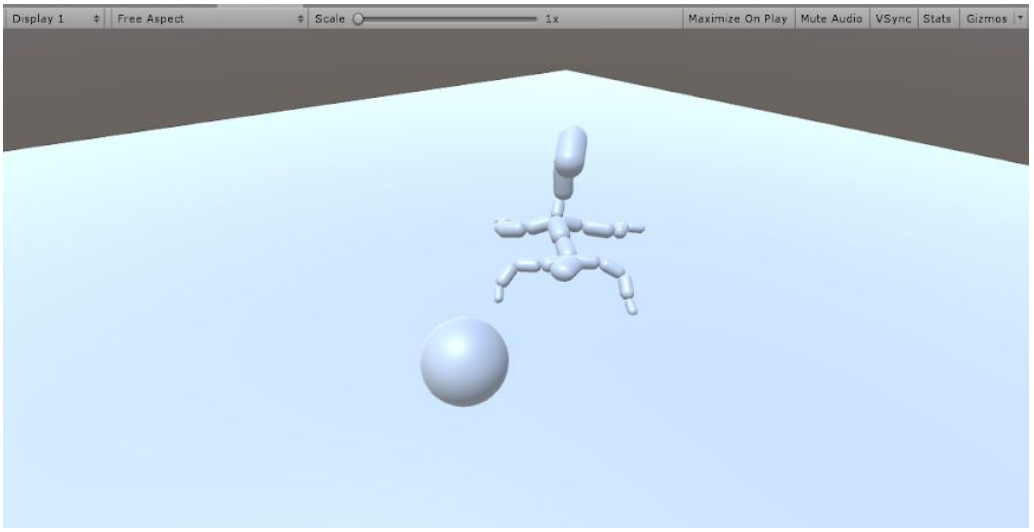


Ilustración 46: Criatura con cuerpo de escorpión caminando hacia el objetivo (Elaboración propia)

6. Conclusiones

6.1. Animación Tradicional VS Animación Procedimental

Conociendo ahora todos los detalles importantes sobre las técnicas de animación, tanto tradicionales como procedimentales, y las herramientas más utilizadas, se plantean una serie de cuestiones que tanto animadores como desarrolladores no dejan de hacerse: ¿Qué método es mejor? ¿No se pierde la personalidad de los personajes al controlar por código sus movimientos? ¿Llegará la animación procedimental a sustituir por completo a la tradicional?

Antes de nada, para poder responder a esas preguntas, primero hay que darse cuenta de que el objetivo de la animación procedimental está muy lejos de una serie de concepciones erróneas: ni se intenta sustituir la figura del animador ni se busca abaratar gastos en el proceso de desarrollo.

La animación procedimental pretende que los animadores puedan centrarse en la actuación de los elementos que animan, agilizando su proceso de trabajo y quitando las partes lentas y repetitivas, y ofreciendo mejores resultados. Ofrece un método de trabajo más lineal, es más sencillo de organizar y controlar y reduce la carga de trabajo. Pero también ofrece muchas ventajas que la animación tradicional no puede alcanzar y eso es un hecho. En resumen, la animación procedimental busca la generación de movimiento creíble, la facilidad de movimientos personalizables y la actuación interactiva en tiempo real.

Pero, después de conocer todas estas técnicas vistas, el resumen concluye en que la animación procedimental busca el contacto y la interacción de los elementos, especialmente de los personajes, con el entorno, y esa es la dirección adecuada a seguir en el futuro de animaciones en videojuegos. A fin de cuentas, como ya sabemos, los videojuegos son puramente interactivos y con su rápida evolución, requieren un número de reacciones que un animador trabajando de manera tradicional no puede predecir. La línea

por la que se quiere llevar la animación en esta industria es una en la que nos permita ver a personajes interactuar con el entorno y los elementos independientemente de la escena y del mundo. Ahí es cuando se consigue la verdadera inmersión del jugador. La cuestión no es hacer algo más barato y rápido, es conseguir mejores animaciones que respondan a fuerzas desconocidas que están fuera del control de los animadores.

En cuanto a la cuestión de la pérdida de "personalidad" o "creatividad" en el proceso procedural, existen posturas muy distintas. Pero la realidad es que en ningún momento se está eliminando la figura creativa y la idea de que animaciones por código dan lugar a resultados robóticos es errónea e ignorante. Solo hay que ver los ejemplos comentados con anterioridad, como el *Grow Home,* videojuego en el que precisamente se ha usado el movimiento por código para crear la personalidad del personaje protagonista, o como en el *Overgrowth* crean un reducido número de poses con la personalidad que quieren darle al movimiento y generan la animación a través de cinemática inversa. Y todavía se puede llegar más allá, o bien incorporando parámetros sobre el tipo de lenguaje corporal o a través de distintos métodos de interacción con el entorno.

A fin de cuentas, todavía es un campo en desarrollo con un pie dentro y otro fuera de la industria, pues todavía cuesta ver proyectos publicados que usen la animación procedimental para algo más que para los estándares establecidos y, sin duda, hay una carencia muy grande de personajes animados procedimentalmente, pero la semilla está plantada y lleva creciendo muchos años. Es incierto saber si la animación procedimental llegará a sustituir la tradicional, a pesar de que parece algo viable en la industria de los videojuegos (mucho menos en la industria cinematográfica, que carece del componente interactivo). Por ahora es necesario seguir desarrollando y publicando ejemplos reales para que se pierda ese rechazo a esta serie de técnicas que han demostrado increíbles resultados.

El camino no es decantarse entre arte o código, el camino es crear una sinergia entre ambos para obtener, no solo animaciones hiperrealistas e inmersivas, sino resultados que exploten la creatividad y las reglas establecidas.

6.2. Conclusiones del desarrollo

En cuanto al trabajo realizado durante el proyecto, se puede afirmar que los objetivos correspondientes al desarrollo establecidos al inicio en el apartado 2. Objetivos han sido cumplidos. A través de la programación de un sistema de cinemática inversa y de un sistema de animación simple, se han aprendido los principios básicos de la primera. La mayor ayuda para esta etapa tanto de desarrollo como de aprendizaje han sido la colección de artículos y conferencias de profesionales recogidas en la bibliografía. No solo se ha aprendido una manera simple y efectiva de resolver el sistema, mediante el cálculo de ángulos a través de la triangulación, sino que se ha descubierto un amplio abanico de soluciones para distintas propuestas usando la misma técnica, puerta que se queda abierta a una futura continuación de la investigación. El proceso de animación, a pesar de desarrollarse de manera escueta para una animación muy sencilla, ha sentado las bases del conocimiento sobre cómo enfocar futuros proyectos de animación procedimental, pues lo más complicado de abordar ha sido el enfoque que usar a la hora de programar. La mayor dificultad al principio, que era la programación, ha resultado al final pasar a segundo plano, pues realmente lo más difícil del proyecto ha sido diseñar el sistema de animación y sus funcionalidades para que hiciese lo que se había planteado. Con todo esto, además de una *demo* funcional con un sistema escalable, se ha obtenido también un manejo más en profundidad del motor gráfico *Unity* y varias de sus características más destacables a la hora de programar para dicho motor. Dicho motor gráfico es una herramienta versátil y sencilla de usar, su sistema de componentes, aunque quizá complicado de entender al principio, resulta de gran utilidad y crea una manera de pensar y programar totalmente nueva para aquellos que vienen de la programación orientada a objetos. Su potencia y accesibilidad lo convierten en la herramienta perfecta, no solamente para proyectos profesionales de mediana medida, sino también para abordar este tipo de trabajos de desarrollo, pues el entorno de trabajo es cómodo y permite hacer pruebas de manera rápida con resultados visuales inmediatos. Sin duda, un campo de juego para cualquier desarrollador interesado en trabajar con #C.

Pero, más allá de lo aprendido y desarrollado, el objetivo *extraoficial* que se ha conseguido es el planteamiento de mejoras para el sistema y el interés por alcanzar nuevos objetivos que, si bien no han tenido cabida en este proyecto, se quedan archivados para un futuro del proyecto a partir del trabajo realizado aquí. A partir de estos cimientos, las posibilidades son múltiples y no dejan de crecer: desde la combinación con el uso de *keyframes* para animaciones complejas, juegos con aleatoriedad de parámetros para la creación de nuevas mecánicas o la animación completa de un videojuego a partir de dicho sistema. Sin duda, es un estudio con un interés creciente y con posible futuro fuera de este trabajo que ha sentado sus bases.

6.2. Opinión final del trabajo

En último lugar, me gustaría añadir un apartado final más cercano y poner un trocito de corazón en este tema en el que he estado trabajando meses. El interés por la animación procedimental surgió de mi indecisión a la hora de elegir entre programación y arte, una duda que hasta los últimos años me había perseguido cuando me planteaba mi futuro profesional. En este tema he encontrado un reflejo y una corriente de pensamiento con la que no sabía que podía llegar a sentirme identificada. En cuanto empecé a investigar sobre este tipo de animación, comencé también a plantear mi objetivo: realizar un videojuego completamente animado de esta manera. Pero no conté con la primera fase, lo más importante, el aprendizaje. A medida que consumía más información, más conferencias veía, más interés le ponía no solo al tema, sino a la propia escena de desarrolladores que impulsaban esta técnica e ideología creativa de unir técnica y arte para crear. Así que, finalmente, después de caer y caer en ese pozo de información cada vez más profundo (que en un principio me había parecido escaso y escueto), me fui dando cuenta de que debía cambiar mi objetivo y utilizar este trabajo como una manera de aprender.

Y es que no solo se aprenden las técnicas, los algoritmos y los programas, sino que se aprenden las visiones a la hora de trabajar, las maneras de desarrollar y las intenciones de los desarrolladores. Después de todo este tiempo tan apegada a un tema que sigue resultándome fascinante, mi opinión no podría estar más consolidada: hay que romper de

una vez las barreras que separan el código del arte, más allá de lo poco que se está consiguiendo hoy en día, por completo, uniendo ambas disciplinas que al final acaban destacando por la necesidad de creatividad. Ni el código es una amenaza para los artistas ni el arte debe ser un mundo ajeno para los programadores: trabajar juntos es la manera de avanzar y conseguir, no solamente juegos más realistas o potentes, sino verdaderas innovaciones y revoluciones en la industria. Una vez pasada esta primera etapa de aprendizaje sobre el tema, no se me ocurre otra cosa que seguir aprendiendo. Y, quién sabe, quizá sea hora de dar el siguiente paso y volver a pensar en el futuro desarrollo de proyectos que usen estas técnicas, o incluso desarrollar nuevas opciones.

Después de 4 años en este grado y meses de trabajo final, creo que estudios enfocados de manera que ataquen a ambas disciplinas a la vez son importantes, nunca dejando de lado la base técnica ni la artística, para formar una nueva generación de artistas apoyados en la ciencia y viceversa.

Si algo nos han demostrado años de videojuegos y avances tecnológicos, es que la vida también se encuentra en el código.

7. Bibliografía

Aristidou, A., Chrysanthou, Y., Lasenby, J. (2016) *Extending FABRIK with model constraints*. Computer Animation and Virtual Worlds, vol. 27, nº 1, 35-37.

Baraff, D., Witkin, A. (1997) Physically Based Modeling: Principles and Practice. Recuperado de: http://www.cs.cmu.edu/~baraff/sigcourse/

Burderlin, A. (1995) *Procedural motion control techniques for interactive animation of human figures.*
Recuperado de: https://core.ac.uk/download/pdf/56370858.pdf

DitzelGames (2019) *C# Inverse Kinematics in Unity.*
Recuperado de: https://www.youtube.com/watch?v=qqOAzn05fvk

Glimberg, S., Engel, M. (2007) *Comparison of ragdolls methods - Physics-based animation.*
Recuperado de: http://image.diku.dk/projects/media/glimberg.engel.07.pdf

Halén, H., Wester, M. (2007) *Real-Time Hair Simulation and Visualization for Games.*
Recuperado de:
http://www.dice.se/wp-content/uploads/2015/01/RealTimeHairSimAndVis.pdf

Holden, D., Komura, T., Saito, J. (2017) *Phase-Functioned Neural Networks for Character Control.*
Recuperado de: http://theorangeduck.com/media/uploads/other_stuff/phasefunction.pdf

Jakobsen, T. (2008) *Advanced Character Physics.*
Recuperado de:
https://web.archive.org/web/20080410171619/http://www.teknikus.dk/tj/gdc2001.htm

Martínez Vilar, R. (2015) *Videojuego basado en la generación procedimental de mundos o niveles.*
Recuperado de: http://rua.ua.es/dspace/handle/10045/48231

Müller-Cajar, R., Mukundan, R. (2007) *Trangulation: A new algorithm for Inverse Kinematics.*
Recuperado de:
https://pdfs.semanticscholar.org/e83d/1438d2910e0bb7bac7d9942b926263b46d7f.pdf

Müller-Fischer, M. (2008) *Fast Water Simulation for Games Using Height Fields.*
Recuperado de:
https://www.researchgate.net/publication/237757773_Fast_Water_Simulation_for_Games_Using_Height_Fields

Natural Motion (2008), *Euphoria brings Dynamic Motion Synthesis live onto PlayStation 3, Xbox 360 and PC.*
Recuperado de:
https://web.archive.org/web/20080924232333/http://www.naturalmotion.com/euphoria.htm

Peng, X. B., Abbeel, P., Levine, S., Van de Panne, M. (2018) *Example-guided deep reinforcement learning of physics-based character skills.*

Rajagopalan, A. (2013) *Real Time Simulation of Game Character Clothing.*
Recuperado de:
https://pdfs.semanticscholar.org/55a1/3a6d330b1352c95a91afa0e414fda4ef20e8.pdf

Song, W., Guang, H. (2011) *A Fast Inverse Kinematics Algorithm for Joint Animation.*
Recuperado de: https://core.ac.uk/download/pdf/82473611.pdf

Togelius, J., Champandard, A. J., Luca Lanzi, P., Mateas, M., Paiva, A., Preuss, M., Stanley, K. O. (1998) *Procedural Content Generation: Goals, Challenges and Actionable Steps.*
Recuperado de: http://drops.dagstuhl.de/opus/volltexte/2013/4336/pdf/7.pdf

Unity Documentation, v. 2019.2. (2019)
Recuperado de: https://docs.unity3d.com/

Van den Bergen, G. (2015) *Math for Game Programmers: Inverse Kinematics Revisited.*
Recuperado de: http://www.dtecta.com/files/GDC15_VanDenBergen_Gino_Math_Tut.pdf

Zucconi, A. (2017) *An Introduction to Procedural Animations.*
Recuperado de: https://www.alanzucconi.com/2017/04/17/procedural-animations/

7.1. Videojuegos referenciados en el análisis

Assassins Creed IV: Black Flag (2013) Ubisoft.
Backbreaker (2010) NaturalMotion.
Clumsy Ninja (2012) NaturalMotion.
Elite: Dangerous (2014) Frontier.
Fifa 15 (2014) EA Sports.
Gang Beasts (2017) Boneloaf.
Grand Theft Auto V (2013) Rockstar North.
Grow Home (2015) Ubisoft Reflections.
Half-Life (1998) Valve Corporation.
Hitman: Codename 47 (2000) IO Interactive.
Jurassic Park: Trespasser (1998) LucasArts.
Last Of Us 2 (En desarrollo) Naughty Dog.
Mortal Kombat (1992) Midway.
Overgrowth (2017) Wolfire Games.
Pong (1972) Atari.
Rain World (2017) Videocult.
Saga Diablo (1996-2014) Blizzard Entertainment.
Shadow of the Colossus (2005) Team IC0.
Star Wars: The Force Unleashed II (2010) LucasArts.
Super Mario 64 (1996) Nintendo.
The Legend of Zelda (1987) Nintendo.
The Legend of Zelda: Twilight Princess (2006) Nintendo.
The Witcher 3: Wild Hunt (2015) CD Projekt Red.
Uncharted: Drake's Fortune (2007) Naughty Dog.

7.2. Conferencias de la *Games Developers Conference*

Rosen, D. (2014) *An Indie Approach to Procedural Animation.*
Disponible en: https://www.youtube.com/watch?v=LNidsMesxSE

Rhodes, G. (2015) *Math for Game Programmers: Mixing Geodetic, Hand-crafted and Procedural Geometry.*
Disponible en: https://www.youtube.com/watch?v=0A4P6MaQxTs

Jakobsson, J. Therrien, J. (2016) *The Rain World Animation Process.*
Disponible en: https://www.youtube.com/watch?v=sVntwsrjNe4

Bereznyak, A. (2016) *IK Rig: Procedural Pose Animation.*
Disponible en: https://www.youtube.com/watch?v=KLjTU0yKS00

Zadziuk, K. (2016) *Motion Matching, The Future of Games Animation… Today.*
Disponible en: https://www.youtube.com/watch?v=KSTn3ePDt50&t=419s

Mach, M. (2017) *Physics Animation in Uncharted 4: A Thief's End.*
Disponible en: https://www.youtube.com/watch?v=7S-_vuoKgR4

Compton, K. (2017) *Practical Procedural Generation for Everyone.*
Disponible en: https://www.youtube.com/watch?v=WumyfLEa6bU

Lista de figuras

Lista de códigos

Lista de tablas

Lista de gráficas

Glosario de términos

AAA: Triple A, clasificación de videojuegos producidos y distribuidos por un editor o una desarrolladora importante.

Add on: Extensión de un programa.

Asset: Recurso para el desarrollo de videojuegos (modelos 3D, texturas, sonidos, botones…)

Frame: Fotograma.

Game design: Diseño del juego.

Gameplay: Forma en la que un jugador interactúa con un videojuego.

GDC (Games Developers Conference): Famoso evento sobre desarrollo de videojuegos para profesionales del sector.

Height field: Imagen cuyos píxeles almacenan datos de alturas.

IK (Inverse kinematics): Cinemática inversa, técnica de animación procedimental.

In game: Dentro del juego, en tiempo de juego.

Input: Datos de entrada.

Keyframe: Fotograma clave. En animación, es aquel que define los puntos de inicio y final, y cualquier pose importante de una transición.

Keyframming: Nombre con el que se conoce también a la animación tradicional.

Keynote: Conferencia importante de una empresa o estudio.

LERP (Lineal Interpolation): Interpolación lineal.

Level of detail: Técnica de optimización que consiste en la disminución de la complejidad de los modelos 3D a medida que se alejan del observador.

Machine learning: Rama de la inteligencia artificial basada en la idea de que los sistemas pueden aprender de datos.

Mocap: Técnica de animación que consiste en capturar los movimientos de un actor.

Networking: Técnica de marketing para establecer una red de contactos con empresas y profesionales.

Off-line: No conectado a la red.

Parenting: Sistema de herencia y jerarquías.

PCG (Procedural Content Generation): Generación procedimental de contenido.

Ragdoll: Modelo 3D al que se le han aplicado leyes físicas del mundo real sin darle más interacciones.

Rigging: Colocación de un esqueleto móvil a un modelo 3D para su posterior posado y animación.

Roguelike: Género de videojuegos basados en exploración de mazmorras.

SLERP (Spherical Linear Interpolation): Interpolación lineal esférica.

Soft-body dynamics: Dinámicas de sólidos deformables.

Target: Objetivo.

Unity: Motor gráfico de videojuegos.

Unreal Engine: Motor gráfico de videojuegos.