

Object-oriented Graphics Rendering Engine **Skeletal Animation** **&** **SceneNode Animation**

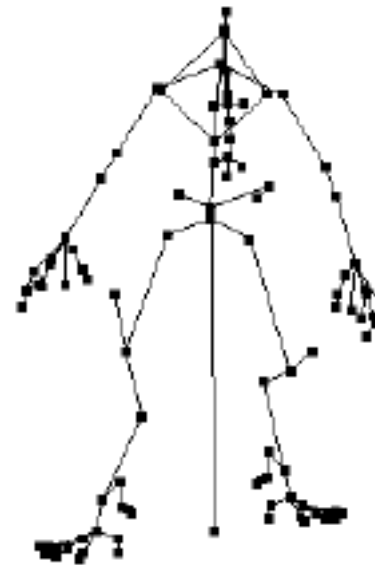
Material original: Ana Gil Luezas
Adaptación al curso 24/25: Alberto Núñez
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

- ❑ Modificar el valor de uno o más parámetros de un objeto a lo largo de un periodo de tiempo.
 - ❑ Algunos parámetros que se pueden modificar: posición, orientación, tamaño, color, coordenadas de textura.
- ❑ Para especificar cómo varían los valores con el tiempo se puede utilizar una función o un muestreo.
 - ❑ Muestreo. Secuencia (*track*) de instantáneas (*keyframes*): valores de los parámetros en distintos instantes de tiempo
$$Kf_0 = \langle t_0, v_0 \rangle, Kf_1 = \langle t_1, v_1 \rangle, \dots, Kf_n = \langle t_n, v_n \rangle$$
- ❑ La información necesaria para cada instantánea depende del tipo de animación:
 - ❑ Valor numérico
 - ❑ Transformación (posición, escala y orientación)
 - ❑ Malla

Los valores correspondientes a los puntos intermedios del tiempo se obtienen por interpolación de los valores vecinos

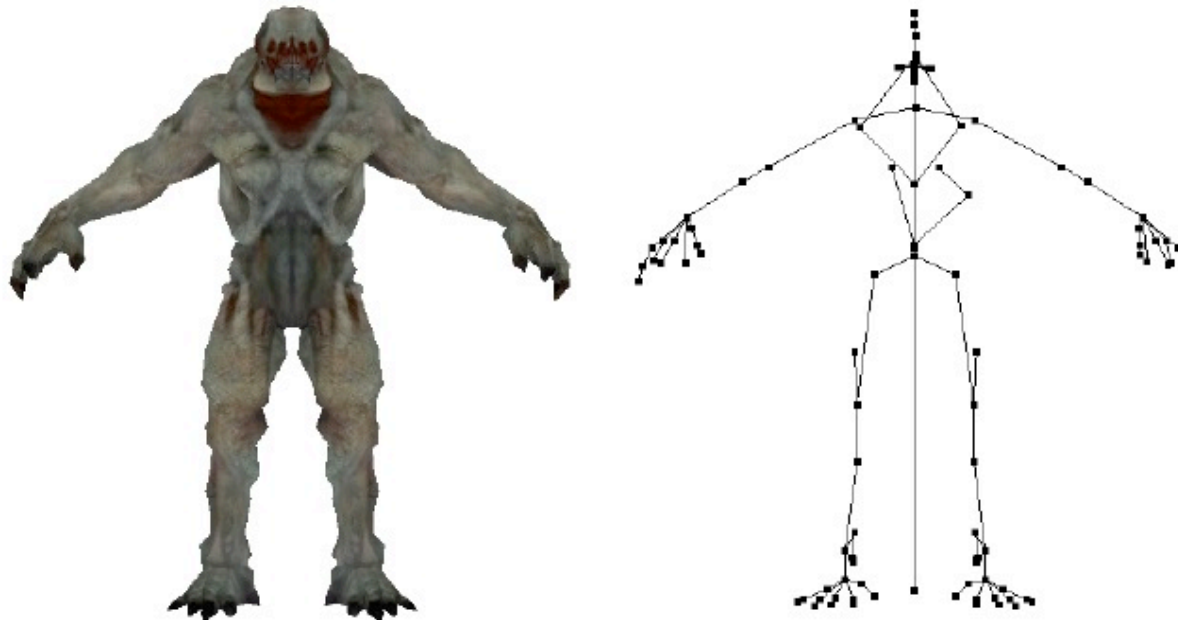
- ❑ En OGRE hay varias clases de animación:
 - ❑ **Numeric Value Animation.** Por ejemplo: intensidad de la luz
 - ❑ **SceneNode Animation:** modifica la posición, orientación y escala de los nodos de la escena. El valor es una transformación:
 - ❑ Transformación: Matriz4x4
 - ❑ Transformación:
 - ❑ Posición (vec3)
 - ❑ Orientación (quaternion -> vec4)
 - ❑ Escala (vec3 / float para escalas uniformes)
 - ❑ **Vertex Animation:** modifica los vértices de la malla. El valor es una malla.
 - ❑ Hay dos subtipos: *Morph* y *Pose* (gestos faciales)
 - ❑ **Skeletal Animation:** El valor es una transformación sobre una articulación de un esqueleto ligado a una malla

- ❑ **Skeletal trees:** Análogos al grafo de la escena.
 - ❑ Constan de una jerarquía de articulaciones (**joints / bones**) dadas por una posición, una rotación y una escala.
 - ❑ La transformación de una articulación actúa sobre algunos vértices de la malla (los enganchados a la articulación)



Puedes ver, por ejemplo, [robot.skeleton.xml](#): `<bones>` y `<bonehierarchy>`

- ❑ **Skeletal meshes:** Además contienen información sobre la asociación entre vértices y articulaciones.
- ❑ Al asociar una malla con su esqueleto (**rigging**) se elige una pose adecuada (**bind pose**).
- ❑ Las transformaciones asociadas son el punto de partida de las animaciones.



Puedes ver, por ejemplo, [robot.mesh.xml](#): `<boneassignments>`

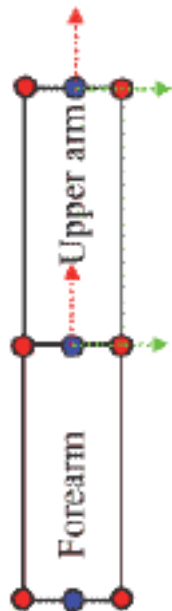
❑ Skeletal meshes: joints (bones) and weights

Grafo de la escena: Todos los vértices de la malla se adjuntan a un único nodo. Todos los vértices de la malla se ven afectados por igual por la transformación del nodo.

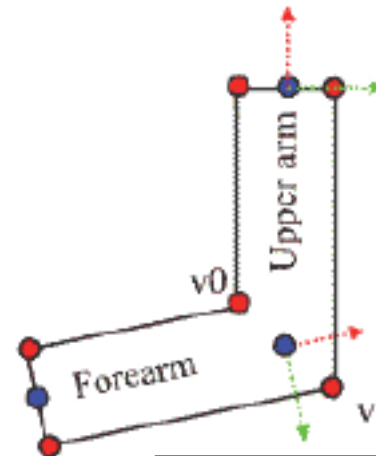
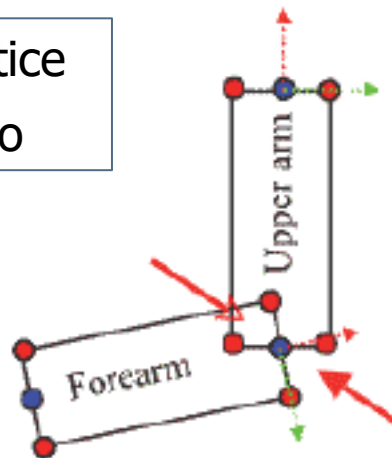
Esqueletos: Cada vértice de la malla se puede enganchar a varias articulaciones con distintos pesos.

Cada vértice de la malla se ve afectado por las transformaciones de las articulaciones a las que está enganchado.

Mayor coste en tiempo y espacio.



Rojo: vértice
Azul: nodo



Rojo: vértice
Azul: articulación

❑ Skeletal meshes: joints (bones) and weights

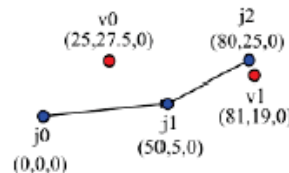
Position
Texture Coordinates
Normal
Bone ID0 Weight 0
Bone ID1 Weight 1
Bone ID2 Weight 2
Bone ID3 Weight 3

Información en la malla para cada vértice: posición, coordenadas de textura, normal, enganches a las articulaciones (articulación y peso).

$$\sum t(v).w$$

Coordenadas del vértice

Transformación y peso de la articulación



Vertex v0
Vertex pos: (0,25,0)
Vertex joint A: j0
Vertex weight A: 0.5
Vertex joint B: j1
Vertex weight B: 0.5

Vertex v1
Vertex pos: (10,0,0)
Vertex joint A: j1
Vertex weight A: 0.3
Vertex joint B: j2
Vertex weight B: 0.7

Rojo: vértice

Azul: articulación

$$v0: (0, 25, 0) + 0.5 (50, 5, 0) = (25, 27.5, 0)$$

$$v1: (10, 0, 0) + 0.3 j1 + 0.7 j2 = (81, 19, 0)$$

❑ Animating skeletal meshes (skinning)

- ❑ Mover el esqueleto: Definir las animaciones del esqueleto
- ❑ Cada animación consta de una secuencia de **key frames** con las transformaciones que se quieren aplicar al esqueleto

Puedes ver, por ejemplo, `robot.skeleton.xml`: `<animations>`

- ❑ Se pueden combinar varias animaciones para obtener animaciones compuestas:
 - ❑ Piernas corriendo + mover los brazos
 - ❑ Piernas corriendo + disparar

Para que se desplace hay que añadir una animación al nodo:
`SceneNode Animation`

- ❑ Una animación se define con un **AnimationState*** **animationState**

- ❑ Puede tomar valor de varias formas:

```
mSM->createAnimationState(name); //mSM gestor de la escena de la clase
entity->getAnimationState(name); //entity se construye sobre una malla
```

si el modelo es una **mesh** que tiene asociado un esqueleto con sus propias animaciones (**name**) predefinidas (**Sinbad.mesh**, por ejemplo)

- ❑ Las animaciones se activan y se repiten mediante:

```
animationState->setEnabled(true);
animationState->setLoop(true);
```

- ❑ Para que una animación avance es necesario indicarle al gestor de la animación el tiempo transcurrido.

- ❑ Para ello se puede usar el método **addTime()** de **AnimationState**

```
animationState-> addTime (evt.timeSinceLastFrame);
```

- ❑ Para reiniciar la animación:

- ❑ `animationState->setTimePosition(0.0);`

- ❑ Se puede averiguar el nombre (**name**) de todas las animaciones predefinidas de una malla (**mesh**) mediante el siguiente código:

```
AnimationStateSet * aux = ent->getAllAnimationStates();
auto it = aux->getAnimationStateIterator().begin();
while (it != aux->getAnimationStateIterator().end()) {
    auto s = it->first;
    ++it;
    cout << "Animation name: " << s << endl;
}
```

- ❑ Algunos nombres de las animaciones de **Sinbad**:
 - ❑ **Dance** (con el que Sinbad baila y saca la lengua)
 - ❑ **RunBase y RunTop** (con el que Sinbad mueve las piernas y corre, mientras la parte de arriba mueve los brazos acompasadamente), ...
- ❑ Para que, además de mover el esqueleto, la entidad se desplace, se reoriente, se escale
 - ❑ Hay que añadir animación al nodo con **SceneNode Animation**

Adjuntar entidades al esqueleto

- Podemos adjuntar una entidad “independiente” a una articulación de otra entidad con esqueleto mediante el siguiente comando

```
entity->attachObjectToBone("BoneName", MovableObject* );
```

Articulación del esqueleto

Entidad que se quiere enlazar

- Por ejemplo, para añadir espada al brazo derecho de Sinbad

```
entity->attachObjectToBone("Handle.R", sword);
```

- donde **sword** es una entidad construida con **Sword.mesh**

- Los elementos se “despegan” (detach) con

```
entity->detachObjectFromBone(MovableObject*);
```

- Para consultar el nombre de las articulaciones o huesos se utiliza

```
SkeletonInstance * skeleton = sinbadEnt->getSkeleton();  
  
int numBones = skeleton->getNumBones();  
  
for (int i=0; i<numBones; i++){  
    cout << skeleton->getBone(i)->getName() << endl;  
}
```

Animaciones y esqueleto de Sinbad

❑ Animaciones:

- ❑ Dance
- ❑ DrawSwords
- ❑ HandsClosed
- ❑ HandsRelaxed
- ❑ IdleBase
- ❑ IdleTop
- ❑ JumpEnd
- ❑ JumpLoop
- ❑ JumpStart
- ❑ RunBase
- ❑ RunTop
- ❑ SliceHorizontal
- ❑ SliceVertical

❑ Esqueleto

- ThumbMed.R
- IndexFingerMed.R
- Clavicle.R
- PinkyDist.L
- IndexFingerDist.R
- Cheek.L
- MiddleFingerMed.L
- Jaw
- TongueMid
- Ulna.L
- Handle.R
- Ulna.R
- Chest
- Foot.L
- Foot.R
- Hand.R
- IndexFingerDist.L
- Cheek.R
- PinkyDist.R
- IndexFingerProx.R
- Handle.L
- RingFingerProx.R
- LowerLip
- Neck
- TongueBase
- Head Sheath.R
- Stomach
- Toe.L
- MiddleFingerProx.L
- RingFingerMed.L
- PinkyMed.L
- MiddleFingerMed.R
- ThumbProx.L
- PinkyMed.R
- Clavicle.L
- MiddleFingerProx.R
- Toe.R
- Sheath.L
- TongueTip
- RingFingerProx.L
- Waist
- MiddleFingerDist.R
- Hand.L
- Humerus.R
- RingFingerDist.L
- Eye.L
- Humerus.L
- RingFingerDist.R
- MiddleFingerDist.L
- IndexFingerMed.L
- ThumbMed.L
- Root
- Thigh.L
- UpperLip
- RingFingerMed.R
- Eye.R
- Brow.L
- Brow.C
- Calf.L
- PinkyProx.L
- ThumbDist.L
- ThumbProx.R
- ThumbDist.R
- Calf.R
- PinkyProx.R
- IndexFingerProx.L
- Brow.R
- Thigh.R

❑ Aplicando una transformación al nodo en función del tiempo

- ❑ Por ejemplo, en el método `frameRendered()`

Ejemplo:

```
void Obj::frameRendered(const FrameEvent & evt) {  
    mNode->yaw(Ogre::Degree(10 * evt.timeSinceLastFrame));  
}
```

↑
velocidad

↑
Tiempo transcurrido (en segundos)

❑ Muestreo: Secuencia (*track*) de instantáneas (*keyframes*)

$K_f = \langle \text{time}, \text{value} \rangle$ // Los valores son transformaciones

$Kf_0 = \langle 0, v_0 \rangle$ $Kf_1 = \langle t_1, v_1 \rangle$ $Kf_2 = \langle t_2, v_2 \rangle$ $Kf_3 = \langle \text{duración_total}, v_3 \rangle$

Los valores correspondientes a los puntos intermedios del tiempo se obtienen por *interpolación de los valores vecinos*. El valor de cada *keyframe* se da con respecto al estado inicial

SceneNode Animation (NodeAnimationTrack)

- ❑ Usamos un objeto de la clase **Animation**
 - ❑ Creado mediante el método `createAnimation(name, duration)` de la clase `SceneManager`
 - ❑ Para especificar caminos/secuencias (`tracks`)
 - ❑ Mediante el método `createNodeTrack(short)` (short es el número de camino)
- ❑ Los caminos son de la clase **AnimationTrack**
 - ❑ Para animaciones de nodos usamos la subclase **NodeAnimationTrack**
- ❑ Un camino se define por una secuencia de puntos por los que pasa y el tiempo que se tarda en alcanzarlos
- ❑ En una animación se pueden definir varios caminos.
 - ❑ Todos tienen que tener la misma duración total, aunque no la misma longitud (número de `keyframes`)

SceneNode Animation (NodeAnimationTrack)

- ❑ Un camino es una secuencia de instantáneas **KeyFrames**
 - ❑ Objetos que guarda información de
(instante de tiempo, valor(=transformación) asociado al instante)
- ❑ Los **keyFrames** son objetos de la clase **TransformKeyFrame** que es subclase de **KeyFrame**, y se crean mediante

```
track->createNodeKeyFrame(time)
```

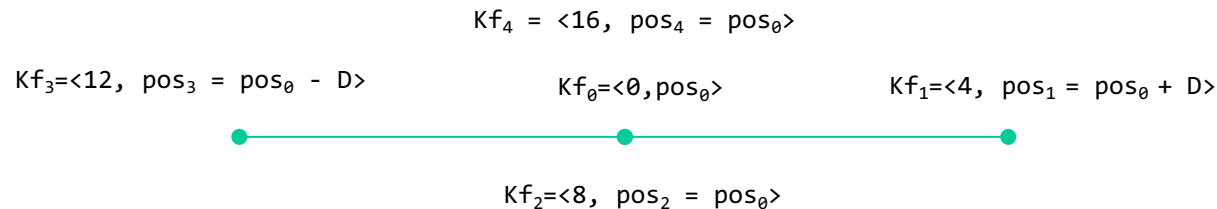
donde *time* es el momento en que se toma la instantánea

- ❑ Los valores de las instantáneas asociadas a un tiempo determinado son transformaciones que se obtienen mediante los métodos **setTranslate**, **setRotation** y **setScale**
- ❑ Para indicar el estado inicial del nodo:
 - ❑ `nodo->setInitialState()`
- ❑ Y para restablecer su estado inicial:
 - ❑ `nodo->resetToInitialState()`

Ejemplo: Desplazamiento lateral

```
// 5 keyFrames: origen (KF0), derecha (KF1), origen (KF2), izquierda (KF3), origen (KF4)  
// Duración total: 16 (totalDuration = 16)  
// Duración entre un KF y el siguiente: 4 (durStep = 4 = totalDuration / 4.)  
// Posición origen: pos0 = inicial + traslación  
// Longitud del desplazamiento: D (movementLength)
```

Los keyframes se deben crear en orden temporal



Las transformaciones hay que darlas a partir del estado inicial del nodo, que puede ser distinto de Kf_0 .

El estado inicial de un nodo se puede fijar con **setInitialState** (por defecto es la identidad)

Ejemplo: Desplazamiento lateral (II)

// Variables

```
int movementLength = 50;
Real duration = 16.0;
Vector3 keyframePos(0, 0, 0);
Real durStep = duration / 4.0;
```

Distancia del desplazamiento

Duración de la animación

Posición del keyframe. Inicialmente (0, 0, 0)

Duración de cada frame

// Create the animation and track

```
Animation * animation = mSM->createAnimation("sinbadWalking", duration);
animation->setInterpolationMode(Ogre::Animation::IM_SPLINE);
NodeAnimationTrack * track = animation->createNodeTrack(0);
track->setAssociatedNode(sinbadNode);
TransformKeyFrame * kf;
```

Nombre de la animación

Modo de interpolación

Track 0

Asociación con el nodo

Ejemplo: Desplazamiento lateral (III)

```
// Keyframe 0 (Init state)
```

```
kf = track->createNodeKeyFrame(durStep * 0);
```

```
kf->setTranslate(keyframePos);
```

```
// Keyframe 1: Go to the right
```

```
kf = track-> createNodeKeyFrame(durStep * 1);
```

```
keyframePos += Ogre::Vector3::UNIT_X * movementLength;
```

```
kf->setTranslate(keyframePos);
```

```
// Keyframe 2: Go to the initial position
```

```
kf = track-> createNodeKeyFrame(durStep * 2);
```

```
keyframePos += Ogre::Vector3::NEGATIVE_UNIT_X * movementLength;
```

```
kf->setTranslate(keyframePos);
```

```
// Keyframe 3: Go to the left
```

```
kf = track-> createNodeKeyFrame(durStep * 3);
```

```
keyframePos += Ogre::Vector3::NEGATIVE_UNIT_X * movementLength;
```

```
kf->setTranslate(keyframePos);
```

```
// Keyframe 4: Go to the right (initital position)
```

```
kf = track-> createNodeKeyFrame(durStep * 4);
```

```
keyframePos += Ogre::Vector3::UNIT_X * movementLength;
```

```
kf->setTranslate(keyframePos);
```

Ejemplo: Desplazamiento lateral (IV)

```
// Our defined animation  
animationState = mSM->createAnimationState("sinbadWalking");  
animationState->setLoop(true);  
animationState->setEnabled(true);
```

❑ En el método `frameRendered`, actualizamos el estado...

```
animationState->addTime(evt.timeSinceLastFrame);
```



- ❑ También es posible animar a Sinbad
- ❑ Las animaciones están previamente definidas en su malla ("Sinbad.mesh").
- ❑ Ver nombre de las animaciones en la diapositiva 11
- ❑ Para obtener la animación y "activarla"

```
AnimationState* animationState = sinbadEnt->getAnimationState("Dance");  
animationState->setLoop(true);  
animationState->setEnabled(true);
```

- ❑ Para "animar" la animación:

```
animationStateDance->addTime(evt.timeSinceLastFrame);
```

- ❑ Para gestionar una animación

- ❑ Hay que crear un `AnimationState`, con el método `createAnimationState()`

- ❑ Recuerda añadir el tiempo transcurrido al gestor de la animación

```
void ...::frameRendered(const Ogre::FrameEvent & evt) {  
    animationState->addTime(evt.timeSinceLastFrame);  
}
```

- ❑ Para configurar el tipo de interpolación entre keyframes (por defecto `IM_LINEAR`)

```
animation->setInterpolationMode(Ogre::Animation::IM_SPLINE);
```

- ❑ Para especificar el estado inicial del nodo a partir del cual se dan las transformaciones

```
mNode->setInitialState();
```

- ❑ Fija la transformación del nodo como estado inicial.

❑ Orientación de un objeto en 3D (se identifica con el eje Z)

Para las animaciones tenemos que usar **cuaterniones** para `setRotation`

❑ Ángulos de Euler. Toda rotación se puede establecer con los tres giros básicos (no de forma única). Problemas: Gimbal lock e interpolación

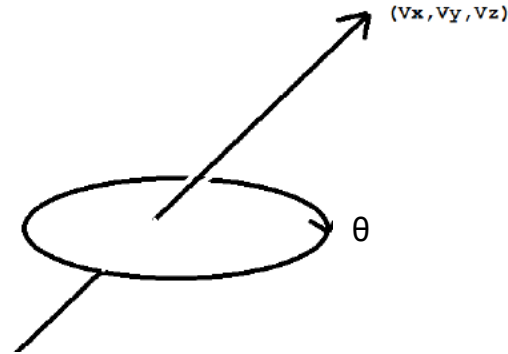
Yaw(β)
`glRotatef(β ,0,1,0)`

Pitch(α)
`glRotatef(α ,1,0,0)`

Roll(γ)
`glRotatef(γ ,0,0,1)`

❑ Rotación sobre un eje genérico:

Vector normalizado (V_x, V_y, V_z)
Ángulo de rotación (θ)



❑ Matriz 3x3 de rotación.

- ❑ `glRotatef(θ, V_x, V_y, V_z)`
- ❑ Problemas: interpolación

❑ Cuaterniones. La información se guarda en un `vec4`:

$$q = (w, x, y, z) = (\cos(\theta/2), V_x * \sin(\theta/2), V_y * \sin(\theta/2), V_z * \sin(\theta/2))$$

quaternion (4 valores) \leftrightarrow matriz de rotación 3x3 (9 valores)

Resuelve el problema de la interpolación de orientaciones

- ❑ Eje de rotación genérico normalizado: vector $V = (V_x, V_y, V_z)$
- ❑ Ángulo de rotación sobre el eje: θ
 - ❑ Esta información se guarda en la forma de **cuaternión unitario**:

$$(w, x, y, z) = (\cos(\theta/2), V_x * \sin(\theta/2), V_y * \sin(\theta/2), V_z * \sin(\theta/2))$$
$$= q = \cos(\theta/2) + V * \sin(\theta/2)$$

❑ **Composición de rotaciones:** se corresponde con el producto de cuaterniones.

❑ Análogo al producto de matrices de rotación: Asociativo y no conmutativo.

❑ Cuaterniones en OGRE

```
Ogre::Vector3 src(0, 1, 1);    Ogre::Vector3 dest(-1, 1, 0);
```

```
//quaternion para rotar de src a dest (ángulo menor)
```

```
Ogre::Quaternion quat = src.getRotationTo(dest);
```

```
//quaternion para pich(90) en forma de ángulo y eje de rotación
```

```
Quaternion q1 = Quaternion(Degree(90.0), Vector3(1, 0, 0));
```

```
//R5 = sqrt(0.5) para pich(90) en forma de cuaternión unitario
```

```
Quaternion q2 = Quaternion(R5, R5, 0, 0);
```

```
Quaternion qp = q1 * q2; //pich(180)
```

```
Quaternion qm = Quaternion(Matrix3);
```


❑ Ejemplos de cuaterniones: vector (V_x, V_y, V_z) y ángulo (θ)

❑ Se guarda en un Vec4 con la siguiente información:

$$(w, x, y, z) = (\cos(\theta/2), V_x * \sin(\theta/2), V_y * \sin(\theta/2), V_z * \sin(\theta/2))$$

$$\begin{aligned}\cos(0) &= 1 \\ \sin(0) &= 0\end{aligned}$$

$$\begin{aligned}\cos(90) &= 0 \\ \sin(90) &= 1\end{aligned}$$

$$\begin{aligned}\cos(45) &= \sqrt{0.5} \\ \sin(45) &= \sqrt{0.5}\end{aligned}$$

w	x	y	z	Description
1	0	0	0	Identity quaternion, no rotation
0	1	0	0	180° turn around X axis
0	0	1	0	180° turn around Y axis
0	0	0	1	180° turn around Z axis
sqrt(0.5)	sqrt(0.5)	0	0	90° rotation around X axis
sqrt(0.5)	0	sqrt(0.5)	0	90° rotation around Y axis
sqrt(0.5)	0	0	sqrt(0.5)	90° rotation around Z axis
sqrt(0.5)	-sqrt(0.5)	0	0	-90° rotation around X axis
sqrt(0.5)	0	-sqrt(0.5)	0	-90° rotation around Y axis
sqrt(0.5)	0	0	-sqrt(0.5)	-90° rotation around Z axis

setRotation(Quaternion)

- ❑ En Ogre podemos obtener el cuaternión necesario para rotar un vector (**src**) y llevarlo a un vector destino (**dest**) usando el método `getRotationTo()`

```
Ogre::Vector3 src(0, 1, 1);  
Ogre::Vector3 dest(-1, 1, 0);  
  
//quaternion para rotar de src a dest (por el ángulo menor)  
Ogre::Quaternion quat = src.getRotationTo(dest);  
keyFrame->setRotation(quat);
```

- ❑ Para configurar el tipo de interpolación entre keyframes (por defecto **RIM_LINEAR**)

```
animation->setRotationInterpolationMode(Ogre::Animation::RIM_SPHERICAL);
```

- ❑ También se pueden usar cuaterniones para las rotaciones de los nodos

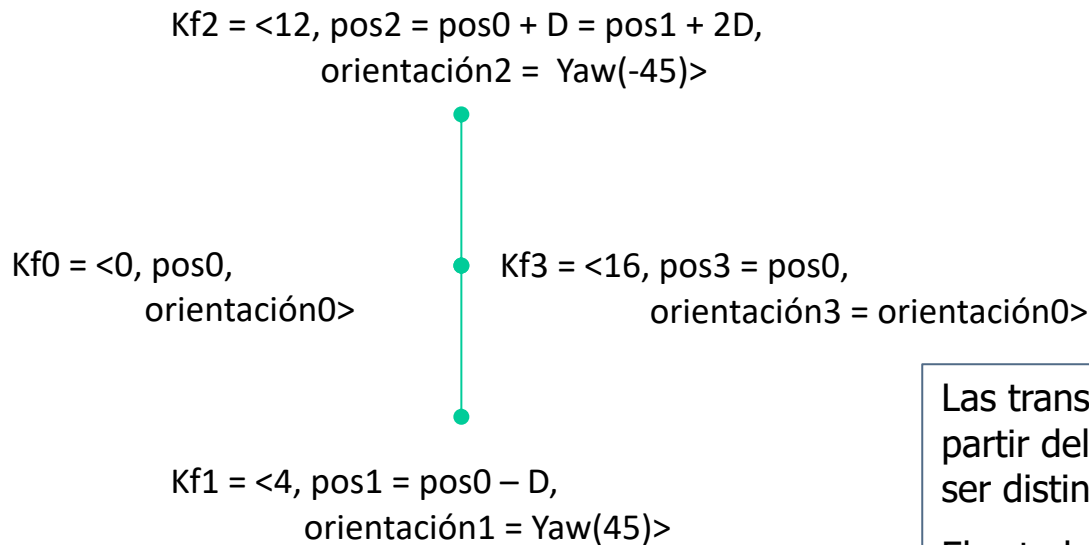
```
node->rotate(quat);  
node->setOrientation(quat);
```

Ejemplo: Desplazamiento vaivén arriba y abajo

❑ Ejemplo: Desplazamiento vaivén abajo y arriba, girando $\pm 45^\circ$ en el eje Y

```
// 4 keyFrames: origen (Kf0), abajo (Kf1), arriba (Kf2), origen (Kf3)
// Duración total: 16
// Duración entre un KF y el siguiente: no uniforme -> 0, 4, 12, 16 (durPaso =4)
// Posición y orientación iniciales:  $\text{pos}_0$ ,  $\text{orientación}_0$ 
// Longitud del desplazamiento:  $D$  (longDesplazamiento)
```

Los keyframes se deben crear en orden temporal



Las transformaciones hay que darlas a partir del estado inicial del nodo, que puede ser distinto de Kf_0 .

El estado inicial de un nodo se puede fijar con **setInitialState** (por defecto es la identidad)

Ejemplo: Desplazamiento vaivén arriba y abajo (II)

❑ Ejemplo: Hay que dar la transformación desde el estado inicial del nodo

```
Vector3 keyframePos(0.0); Vector3 src(0, 0, 1); // posición y orientación iniciales
TransformKeyFrame * kf; // 4 keyFrames: origen(0), abajo, arriba, origen(3)

kf = track->createNodeKeyFrame(durPaso * 0); // Keyframe 0: origen
kf = track->createNodeKeyFrame(durPaso * 1); // Keyframe 1: abajo

keyframePos += Ogre::Vector3::NEGATIVE_UNIT_Y * longDesplazamiento;
kf-> setTranslate(keyframePos); // Abajo
kf-> setRotation(src.getRotationTo(Vector3(1, 0, 1))); // Yaw(45)
kf = track->createNodeKeyFrame(durPaso * 3); // Keyframe 2: arriba

keyframePos += Ogre::Vector3::UNIT_Y * longDesplazamiento * 2;
kf-> setTranslate(keyframePos); // Arriba
kf-> setRotation(src.getRotationTo(Vector3(-1, 0, 1))); // Yaw(-45)

kf = track-> createNodeKeyFrame(durPaso * 4); // Keyframe 3: origen
```