ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Keyframes

November 12th, 2015

Rafael Radkowski

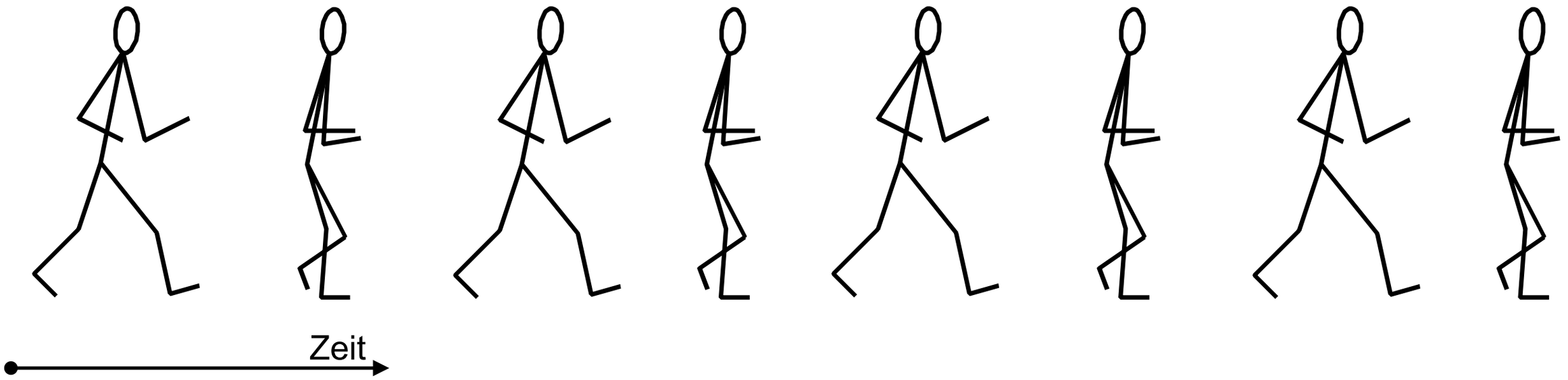
IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

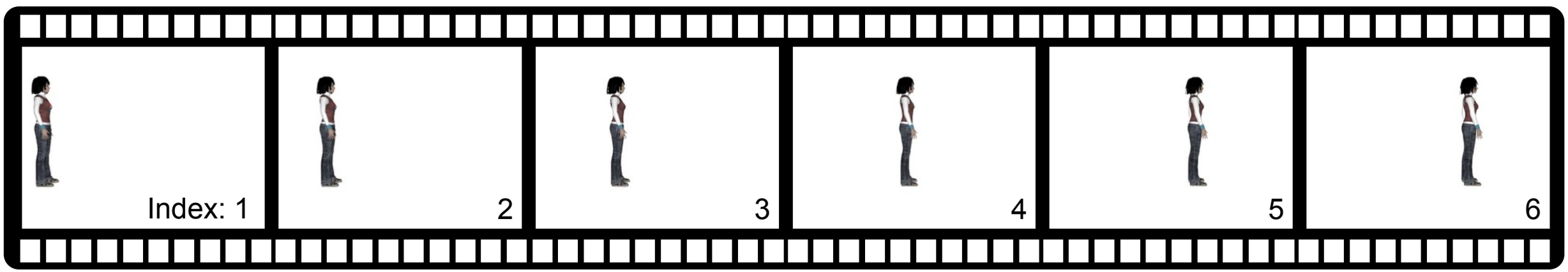
Content

- Computer Animation Introduction
- Multi-body Animation Introduction
- Animation in OpenGL

Video

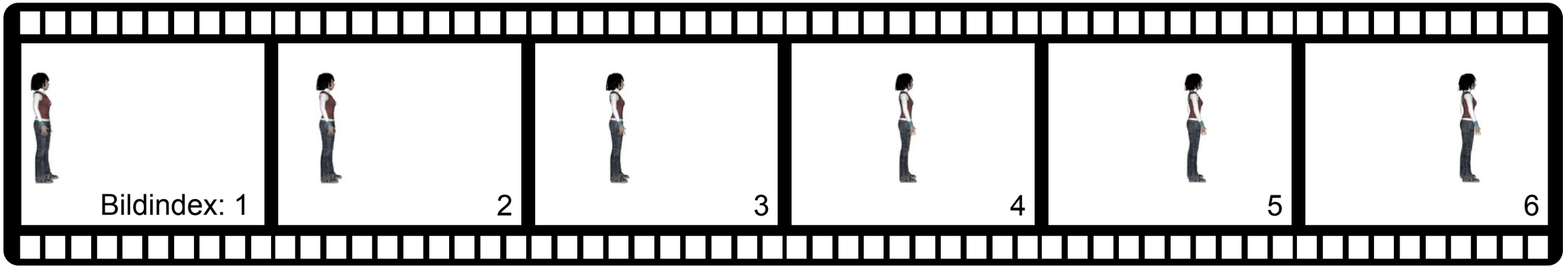
Animations





Animation is the process to generate moving objects on screen. The computer graphics rendering pipeline only generates static scenes. Animation techniques alter this scene from frame to frame. Thus, objects appear as moving objects.

Animation Methods



To generate an animation, the animated object need a new position and orientation in each frame.

Explicit Coding

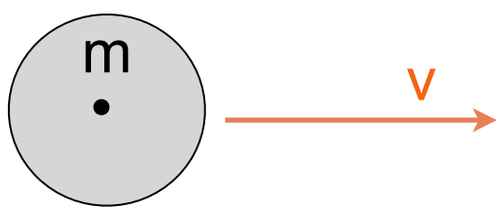
The position and orientation can be explicit coded for every frame. Frame number and translation are stored in a list an fetched when necessary.

Frame	Pos	Rot
1	P_1	R_1
2	P_2	R_2
3	P_3	R_3
4	P_4	R_4
....		
....		

Simulation

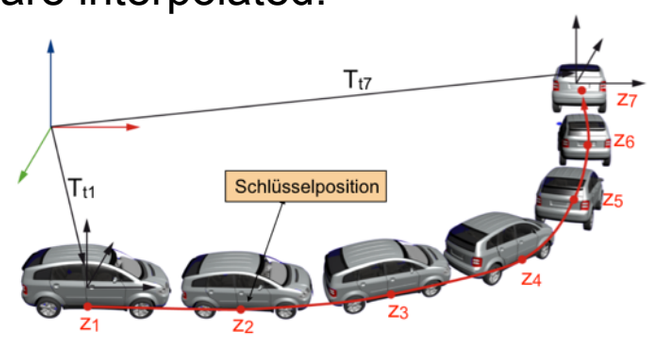
The motion of an rigid object is defined by a set of equations, e.g. Newton's law of dynamic. The position and orientation of each frame is calculated for each frame.

$$\ddot{x}m + \dot{x}d + xc = \sum F$$

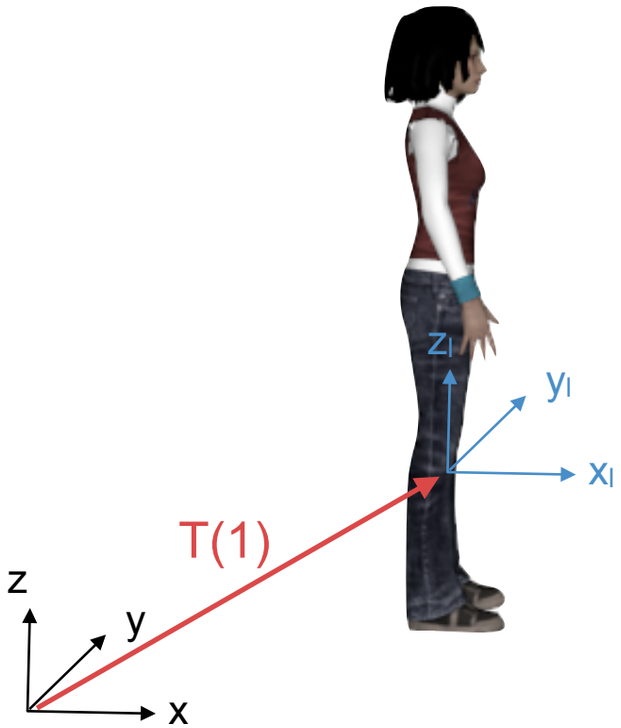


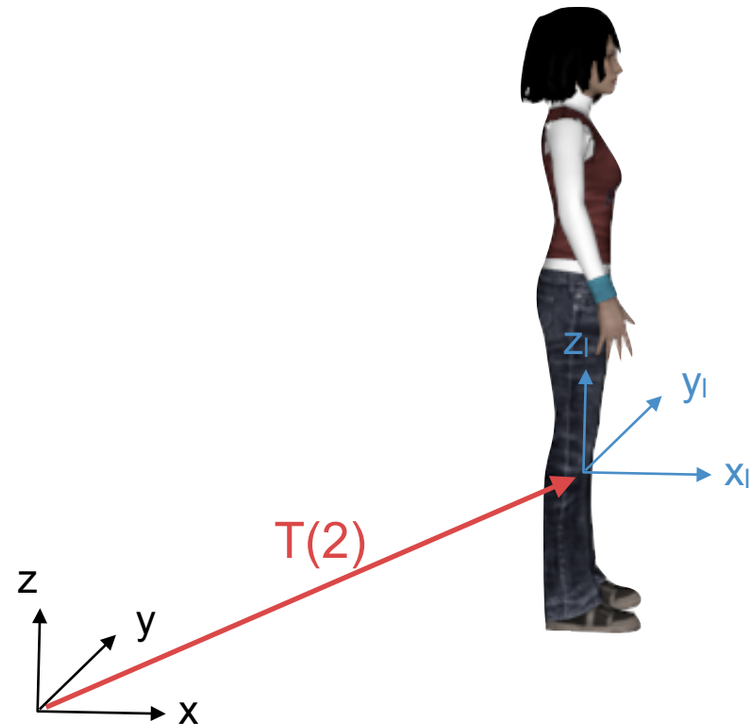
Keyframe Animation

The position and orientation of each object is set for selected key frames. All other positions / orientations between keyframes are interpolated.



Explicit Coding





A user perceives a smooth motion

$$T = \begin{bmatrix} r_0 & r_1 & r_2 & x \\ r_3 & r_4 & r_5 & y \\ r_6 & r_7 & r_8 & z \\ 0 & 0 & 0 & w \end{bmatrix}$$

T(1)

The position and orientation of each frame is described by a matrix. The parameters for each matrix are stored in table.

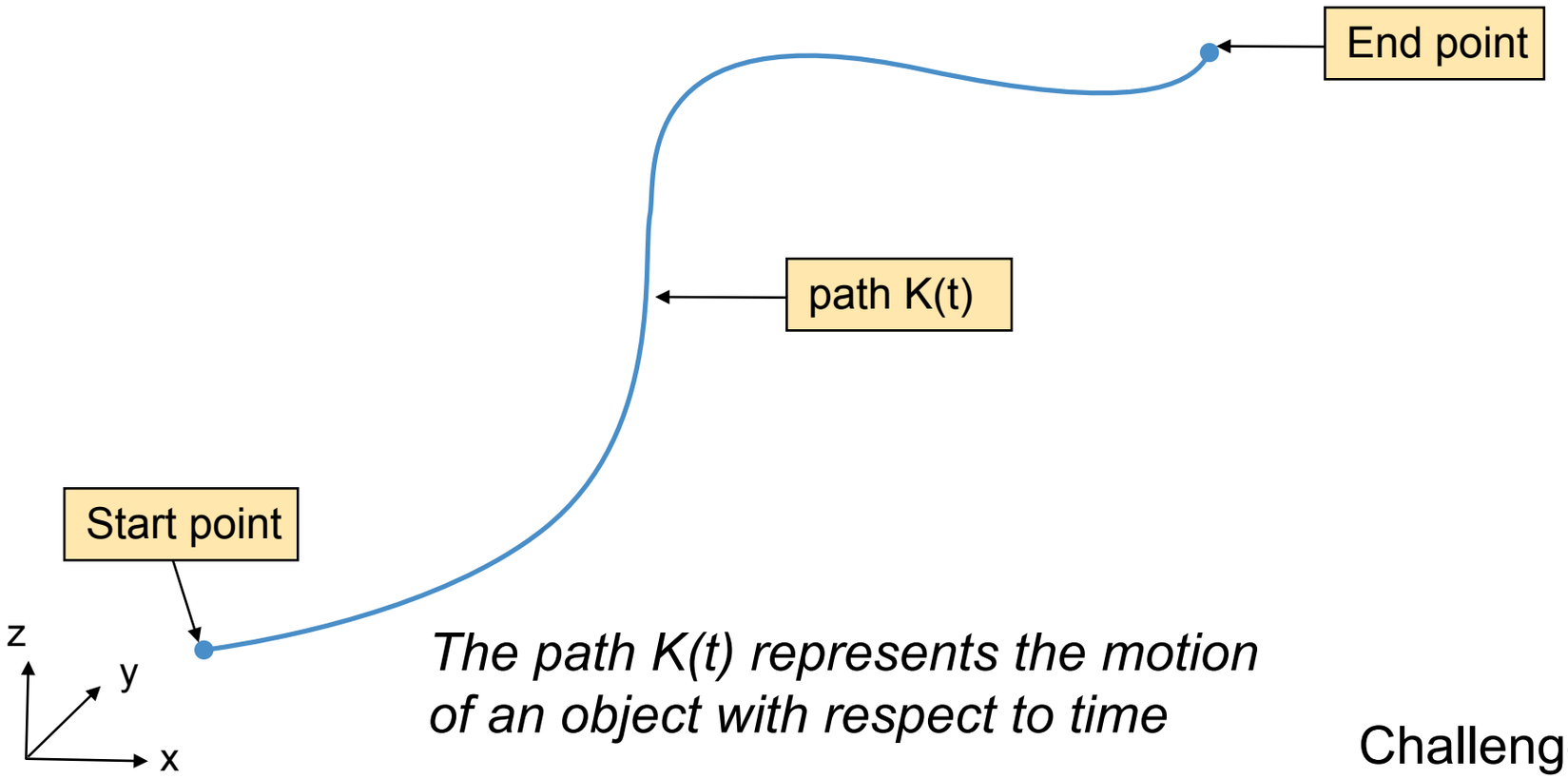
Frame	Pos	Rot
1	P_1	R_1
2	P_2	R_2
3	P_3	R_3
4	P_4	R_4
....		
....		

The motion data is stored in a table.

Areas of application:

- Measurement Engineering

Simulation



Path in parameter description

$$K = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

The path K(t) represents the motion of an object with respect to time

$$T = \begin{bmatrix} r_0 & r_1 & r_2 & x \\ r_3 & r_4 & r_5 & y \\ r_6 & r_7 & r_8 & z \\ 0 & 0 & 0 & w \end{bmatrix}$$

Task of a computer graphics specialist is to take this simulation data and represent it as matrix.

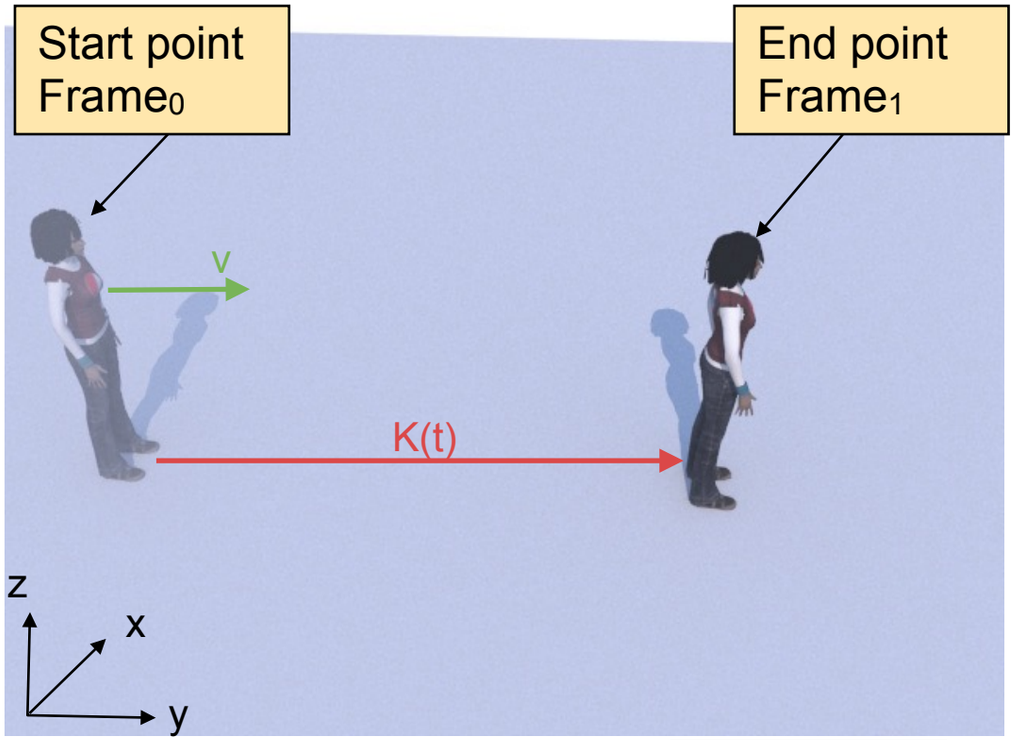
Challenges:

- Amount of data does not match to the frame rate
- No consistent interval between simulation points

Area of application:

- Rigid body simulation in engineering
- Flow simulation
- etc.

Simple Kinematic Simulation



Motion equation:

$$s = v \cdot t + s_0$$

Relative motion equation:

$$\Delta s = v \cdot \Delta t + s_0$$

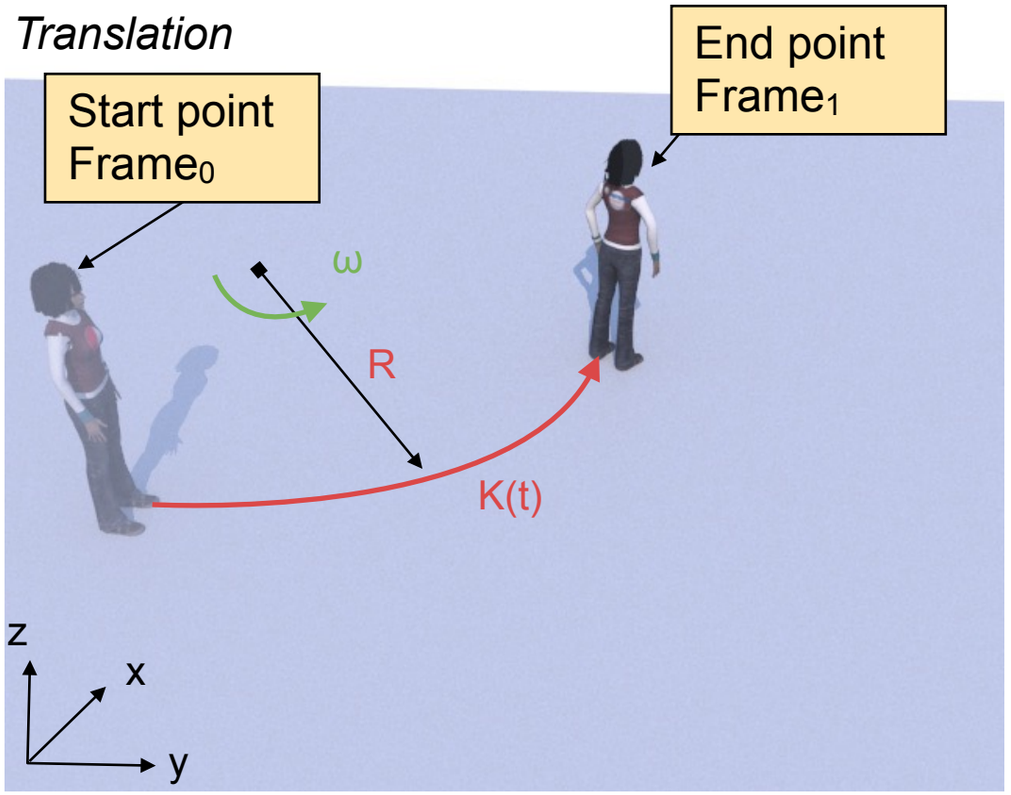
with:

$$\Delta t = t_{frame_1} - t_{frame_0}$$

Path

$$K(t) = \begin{pmatrix} \Delta s + s_{frame_0} \\ y \\ z \end{pmatrix}$$

Translation



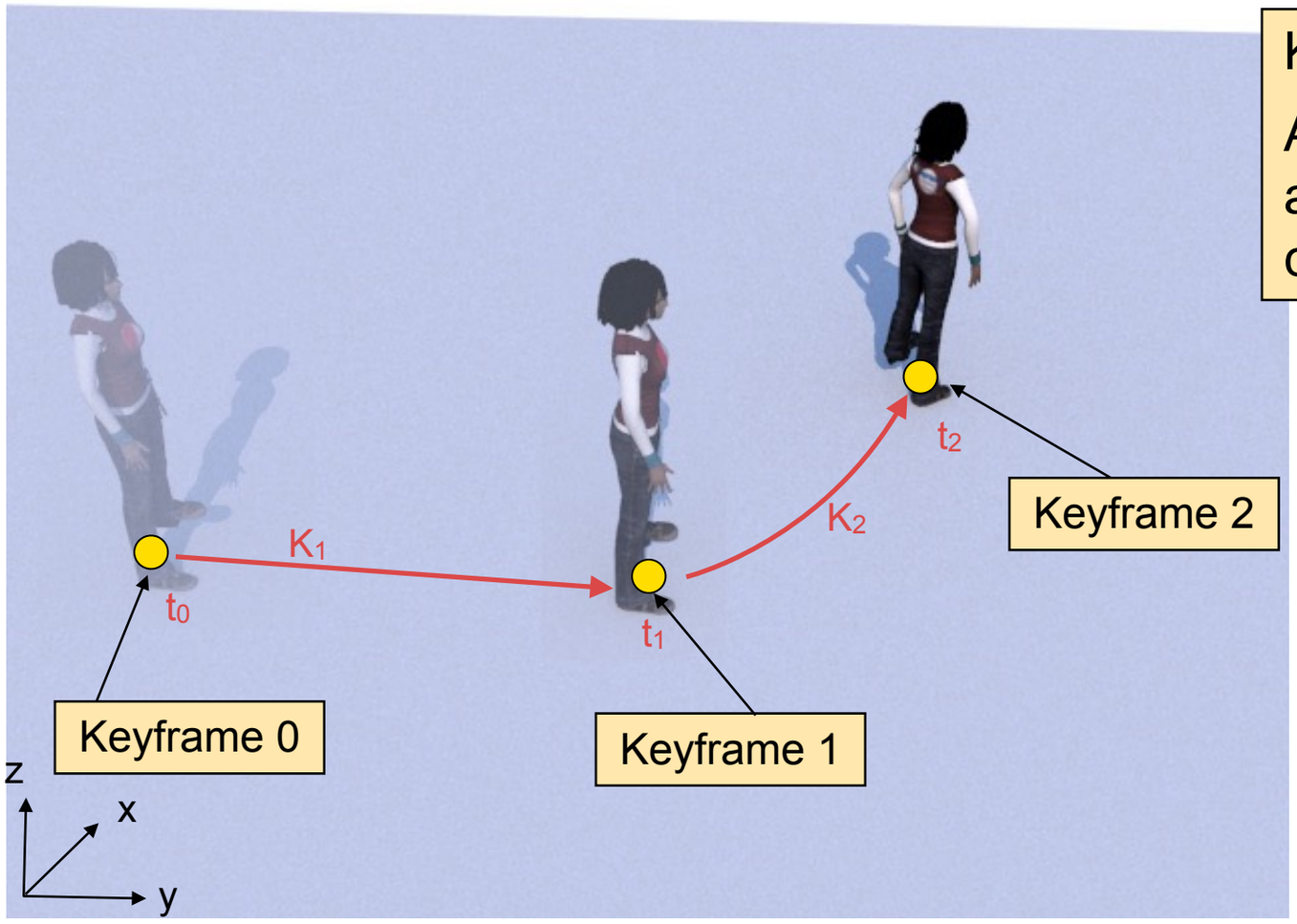
Rotation

Motion equation

$$K(t) = \begin{pmatrix} x(t) = R \cdot cos(\omega \cdot \Delta t) + x_0 \\ y(t) = R \cdot sin(\omega \cdot \Delta t) + y_0 \\ z = 0 \end{pmatrix}$$

s: position
v: velocity
s_0: start position
Δt: time between two frames
s_{frame0}: position in the last frame
ω: angular velocity
R: radius
x_0, y_0: Position in frame 0

Keyframe Animation



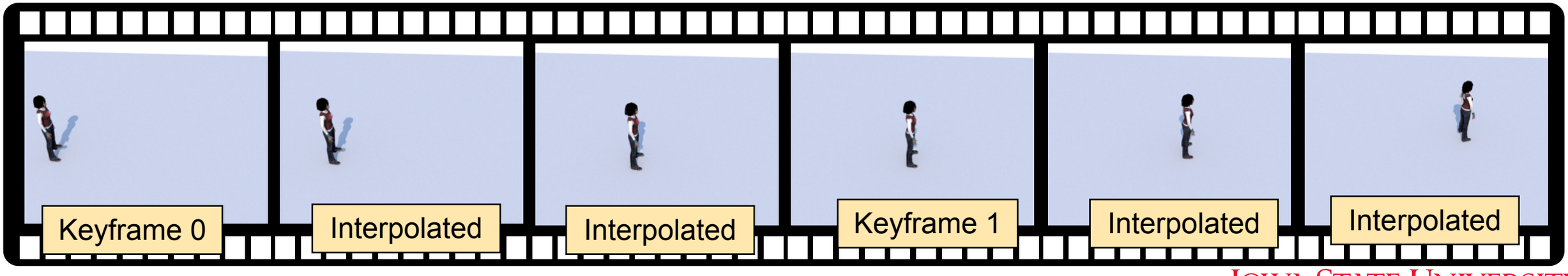
The path is represented by keyframe. The position between keyframes is interpolated

Keyframe:

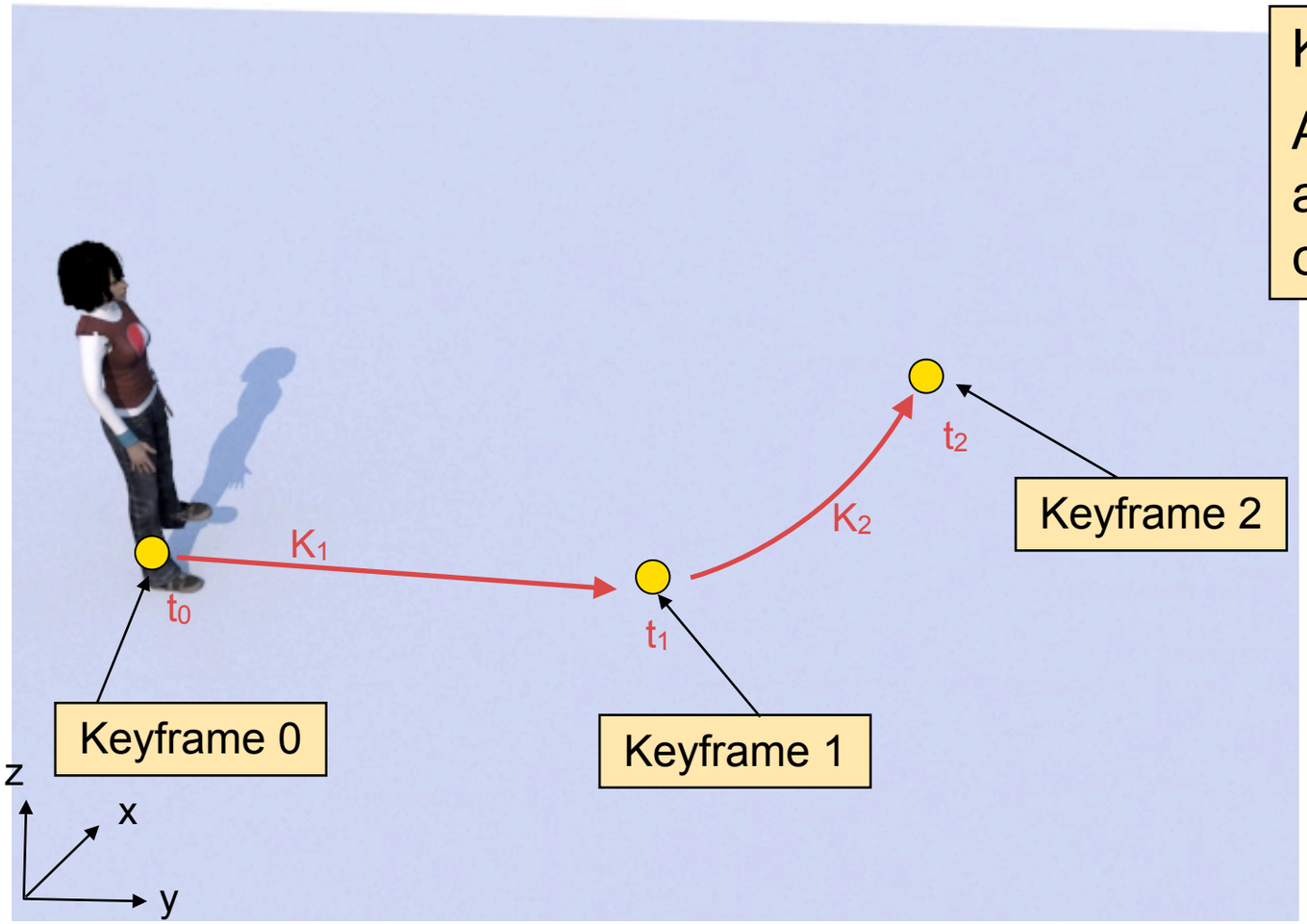
An image with an explicit defined attributes (position, orientation, color) for an object

Keyframe animations are a method that uses key situation which are associated to certain frames. An interpolation between keyframes is used to calculate the location of an object between keyframes.

K_1, K_2: Interpolation



Keyframe Animation



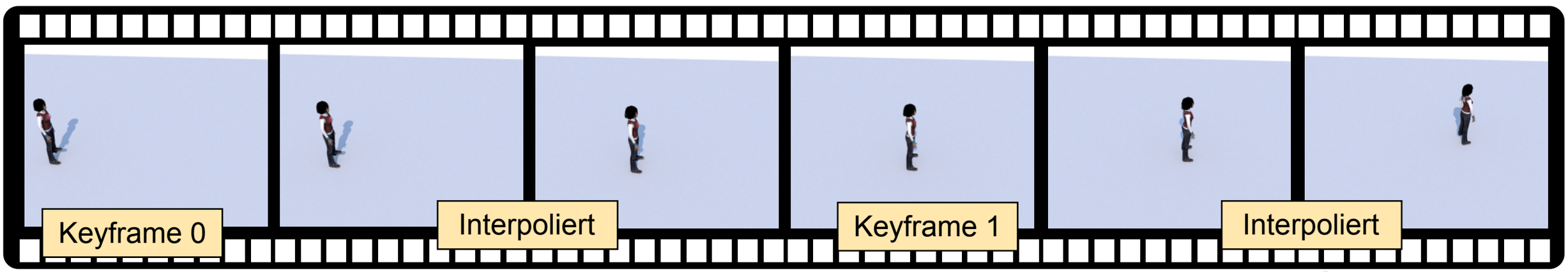
The path is represented by keyframe. The position between keyframes is interpolated

Keyframe:

An image with an explicit defined attributes (position, orientation, color) for an object

Keyframe animations are a method that uses key situation which are associated to certain frames. An interpolation between keyframes is used to calculate the location of an object between keyframes.

K_1, K_2: Interpolation



Linear Interpolation

VRAC | HCI

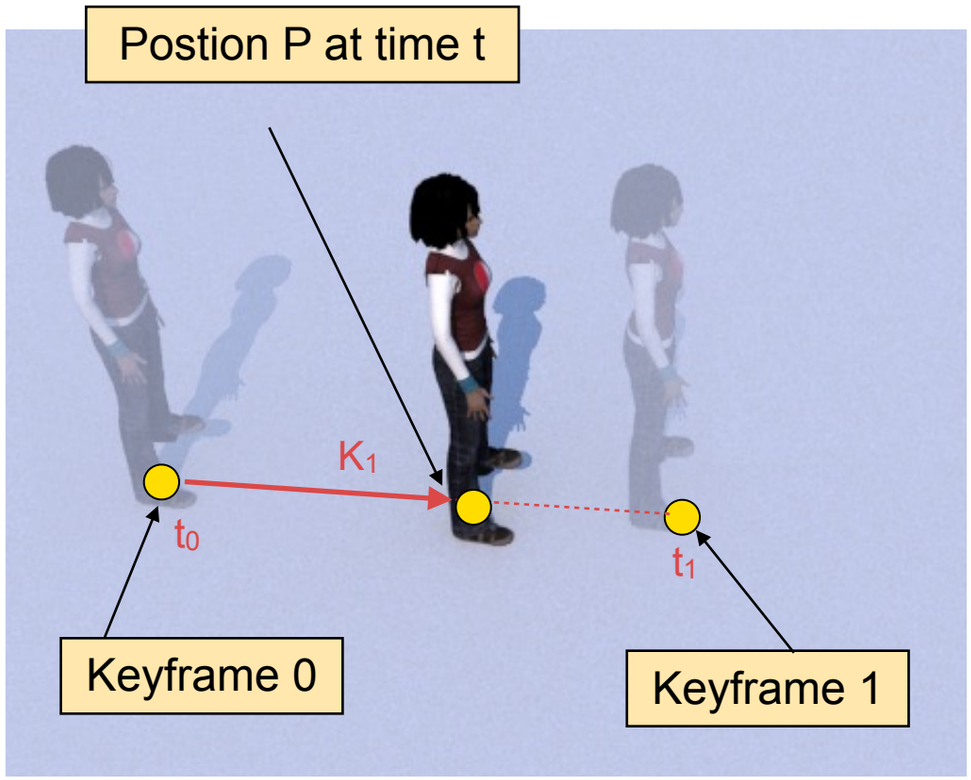
Keyframes

Index	Time	Pos	Rot
z_0	0.0 s	P_0	R_0
z_1	0.1 s	P_1	R_1
z_2	0.2 s	P_2	R_2
z_3	0.3 s	P_3	R_3
....			
....			

A keyframe keeps:
- time fraction of this keyframe
- position
- orientation
- color
- .....

Keep the time steps similar. This facilitates the creation of a uniform motion.

Linear interpolation of the position /orientation



Step 1: Seek the two keyframes in your table that encases the current point in time.

z_0: P_0, R_0

z_1: P_1, R_1

Step 2: Interpolate position P and rotation R

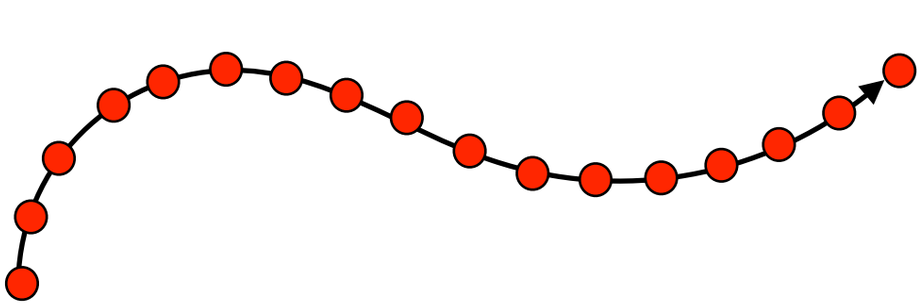
$$P(t) = P_0 + (P_1 - P_0) * \frac{t - t_0}{t_1 - t_0}$$

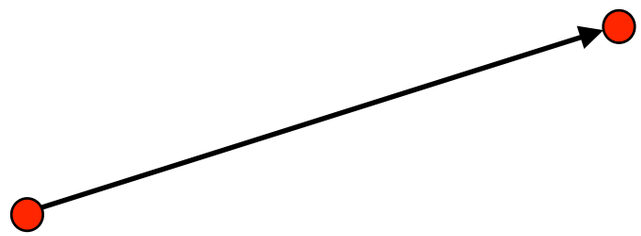
$$R(t) = R_0 + (R_1 - R_0) * \frac{t - t_0}{t_1 - t_0}$$

Step 3: Update the object's transformation matrix.

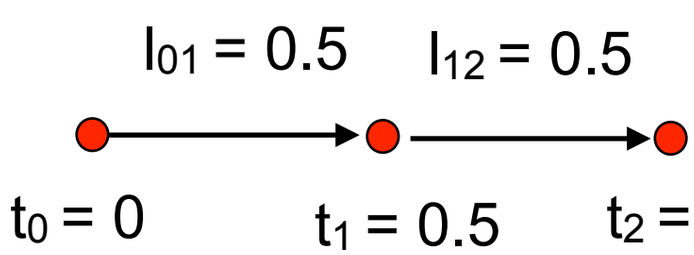
Keyframes Notes

- With a linear interpolation, the number of keyframes depend on the path that you like to describe. A curve needs more keyframes than a direct line.

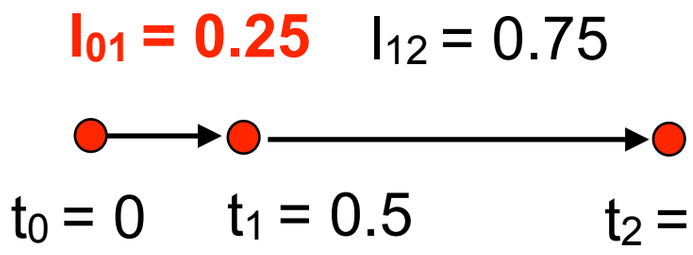




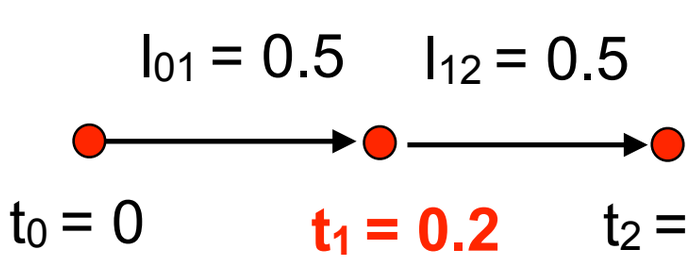
- To obtain a uniform velocity, you need equidistant keyframes OR



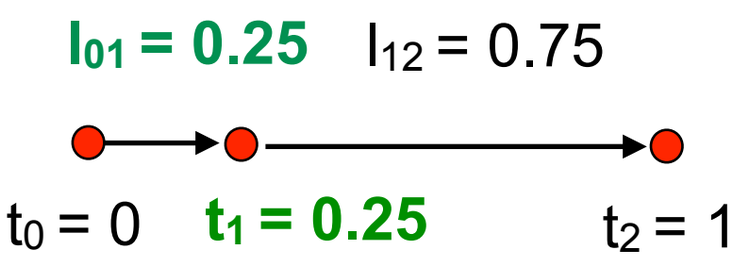
Uniform distribution



path too short



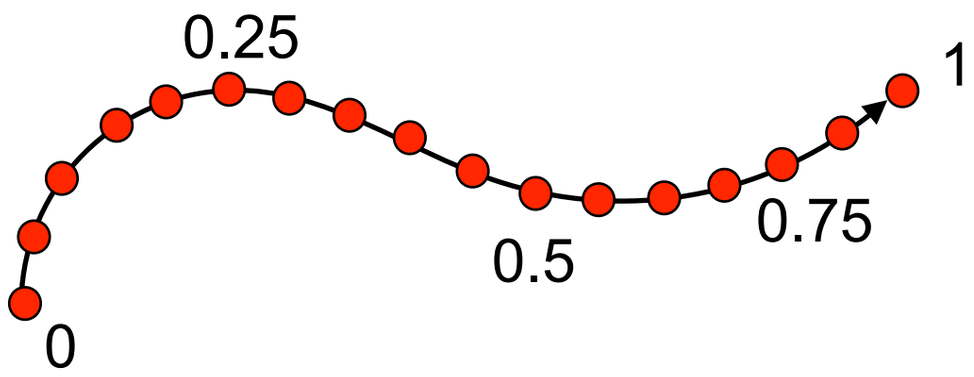
timing

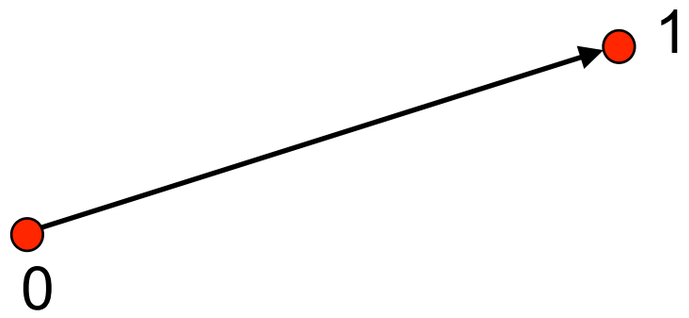


This will work

Keyframes Notes

- Instead of time values, represent the time as fraction in an interval between 0 and 1



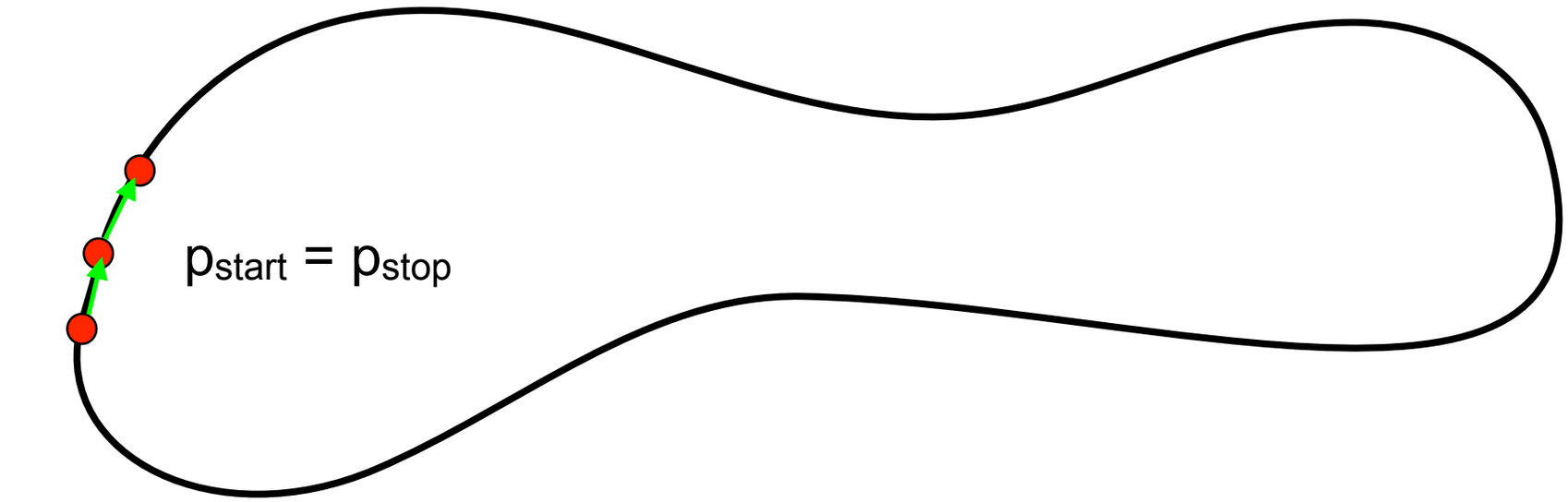


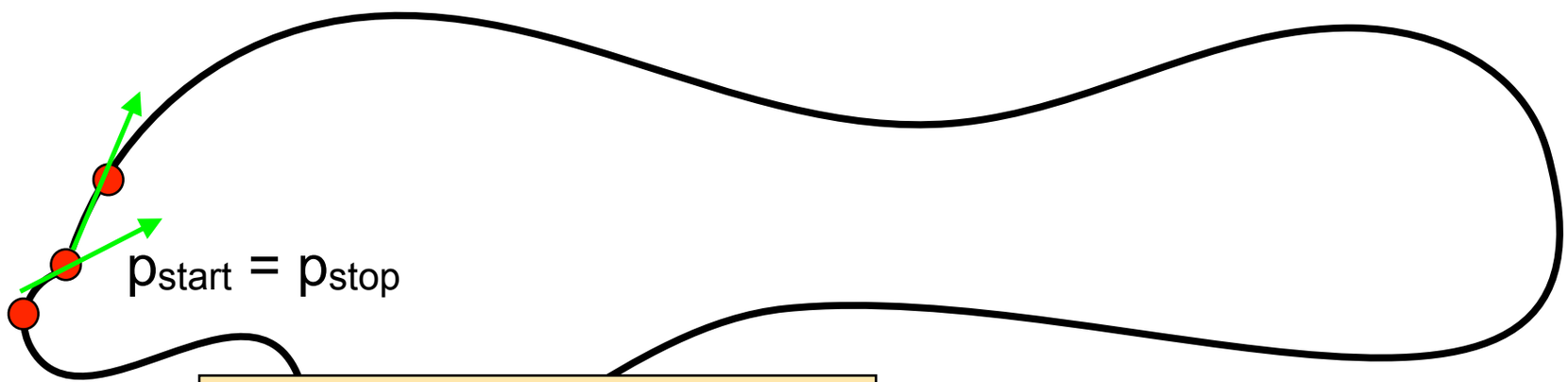
This allows you to change the duration of an animation by multiplying the timing fraction with a duration time.

Keyframes Notes

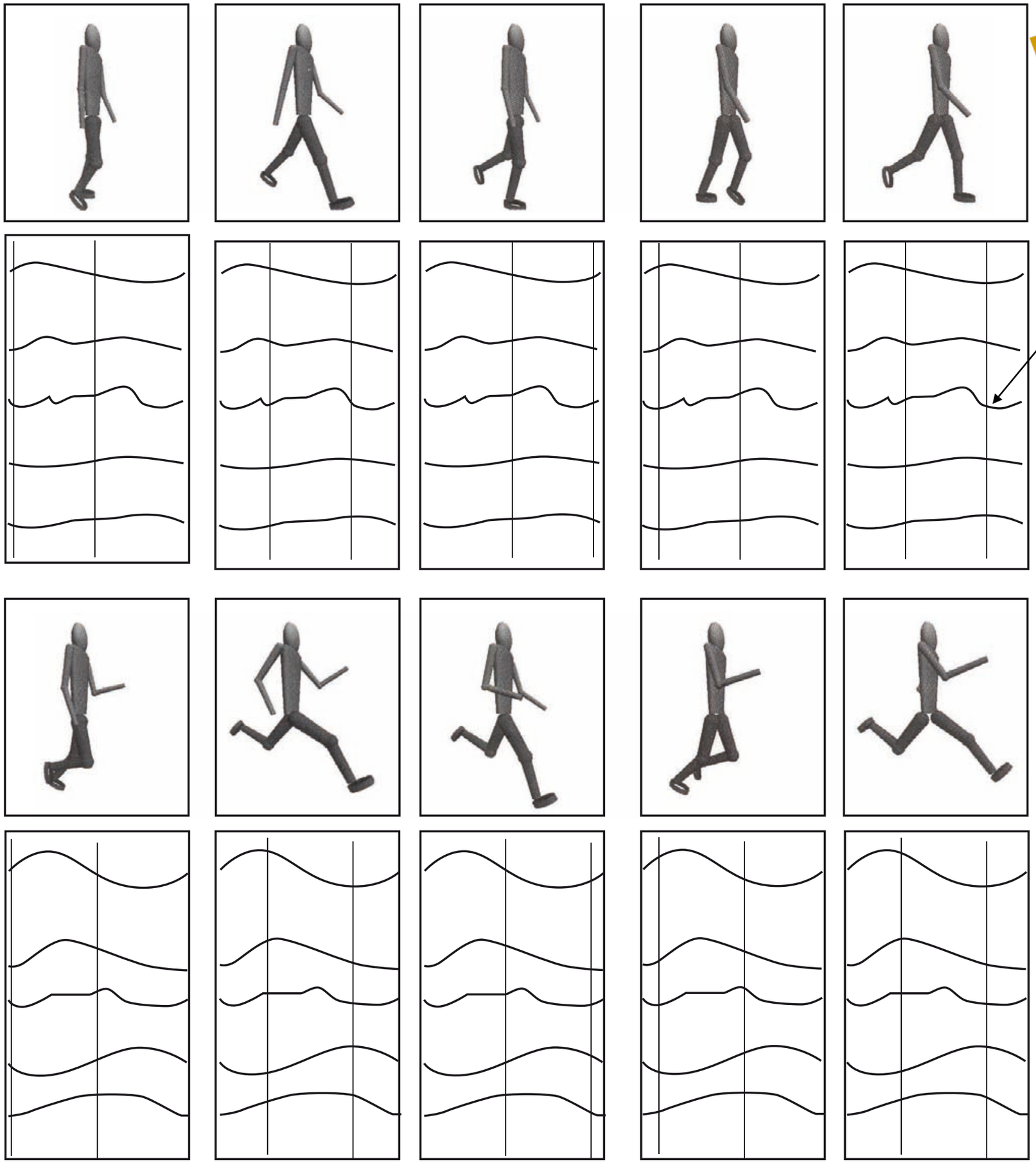
ARLAB

- For a closed loop, start and stop point should be equal (or you must change the interpolation code).
- Make also sure that the tangents gradients are equal





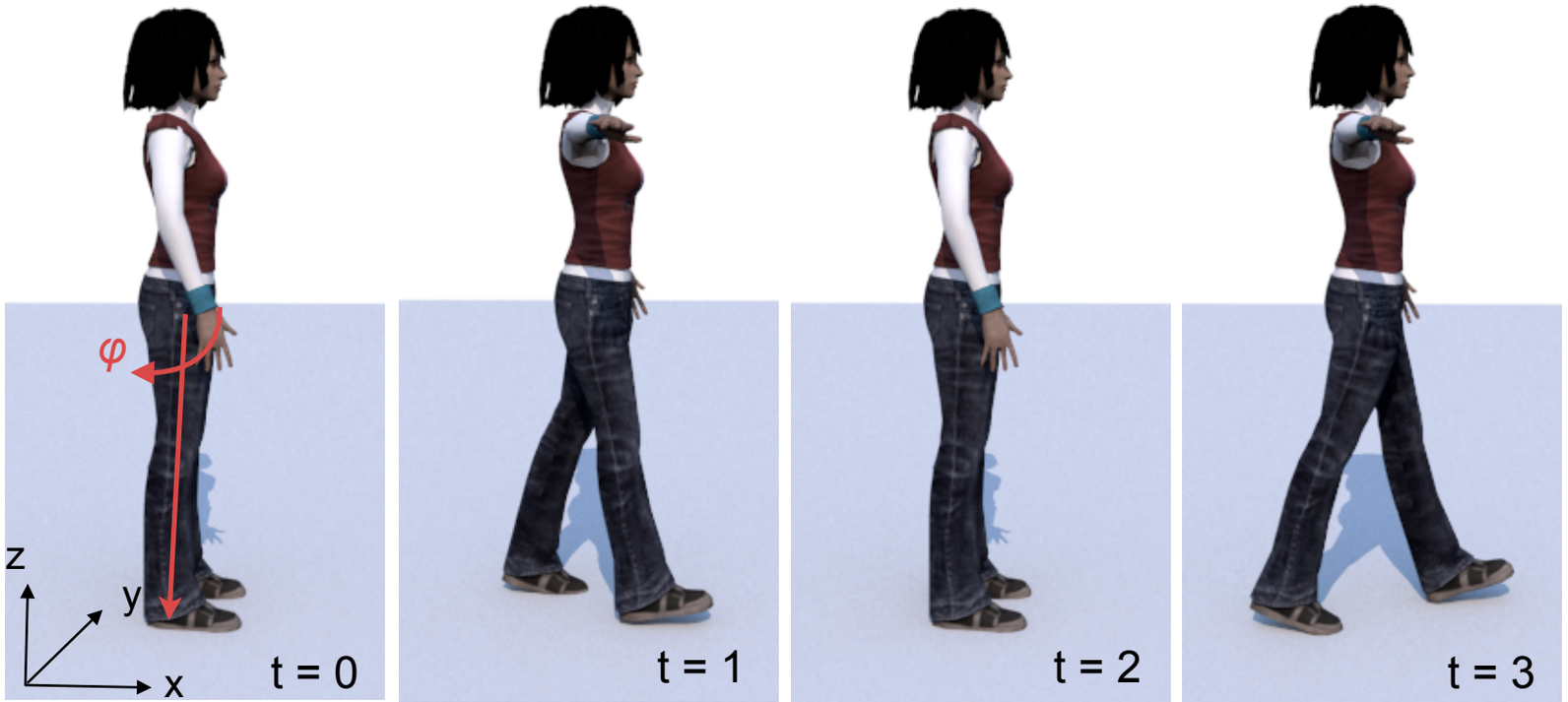
Steep gradients are often noticeable as a jump.

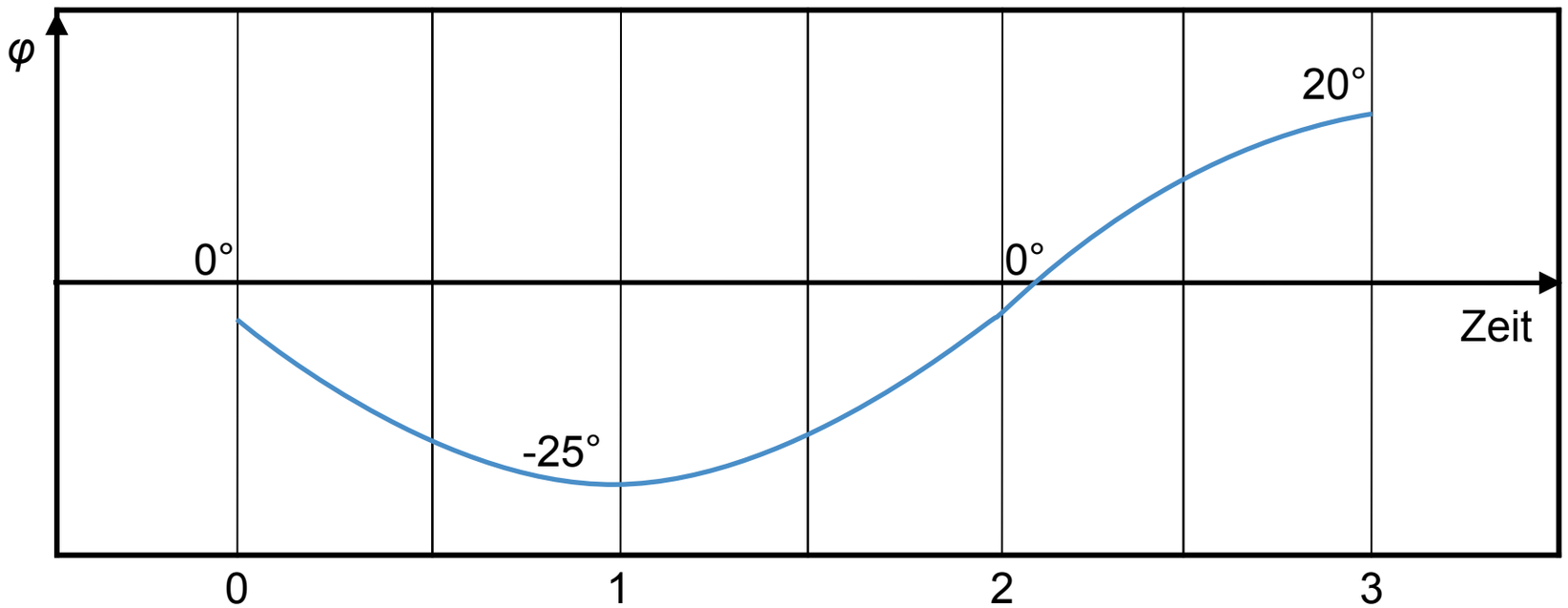


VRAC | HCI

Multi-body Animation

Example





Animation curve for the hip

$$T_H = R_y(\varphi(t)) \cdot P$$

with

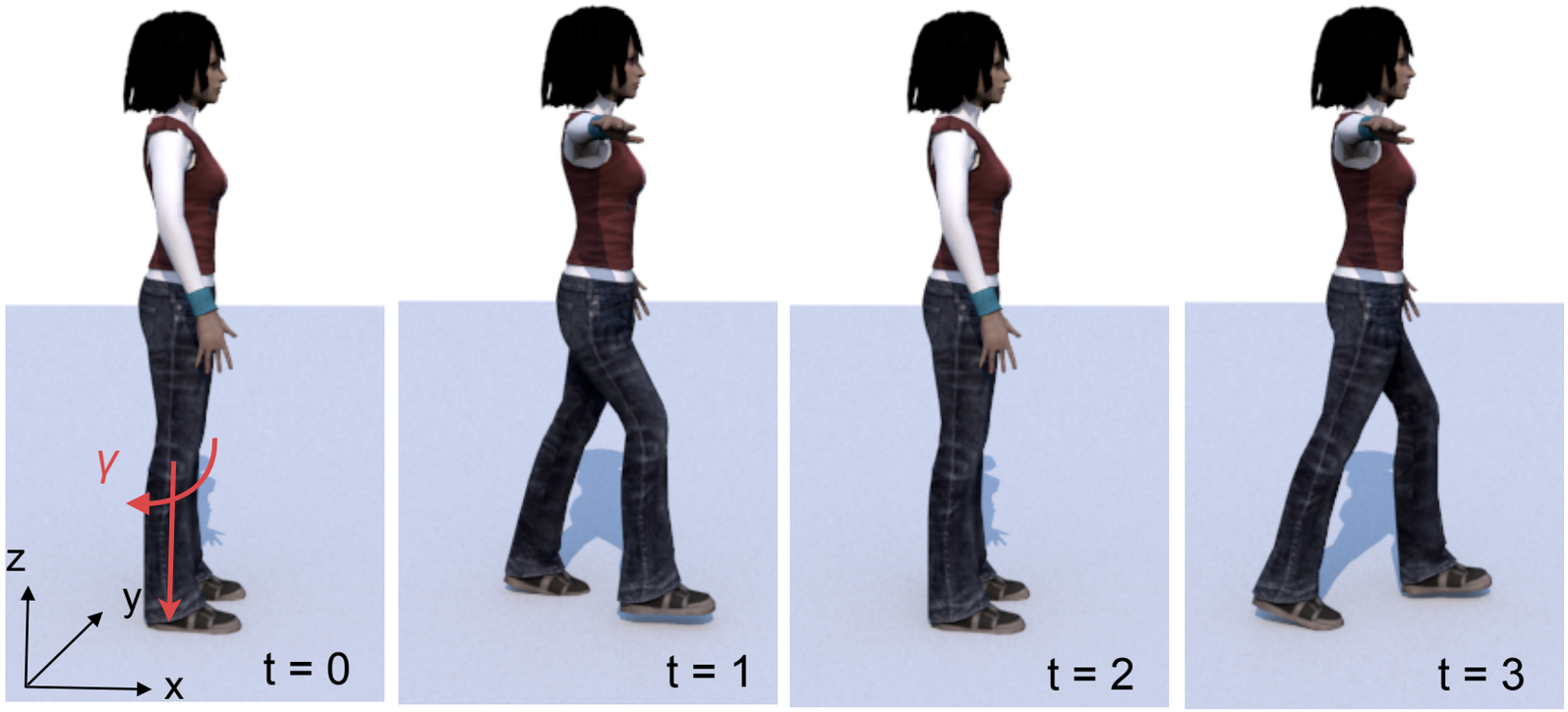
$$R_y(\varphi) = \begin{bmatrix} cos(\varphi(t)) & 0 & sin(\varphi(t)) & 0 \\ 1 & 0 & 0 & 0 \\ -sin(\varphi(t)) & 0 & cos(\varphi(t)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

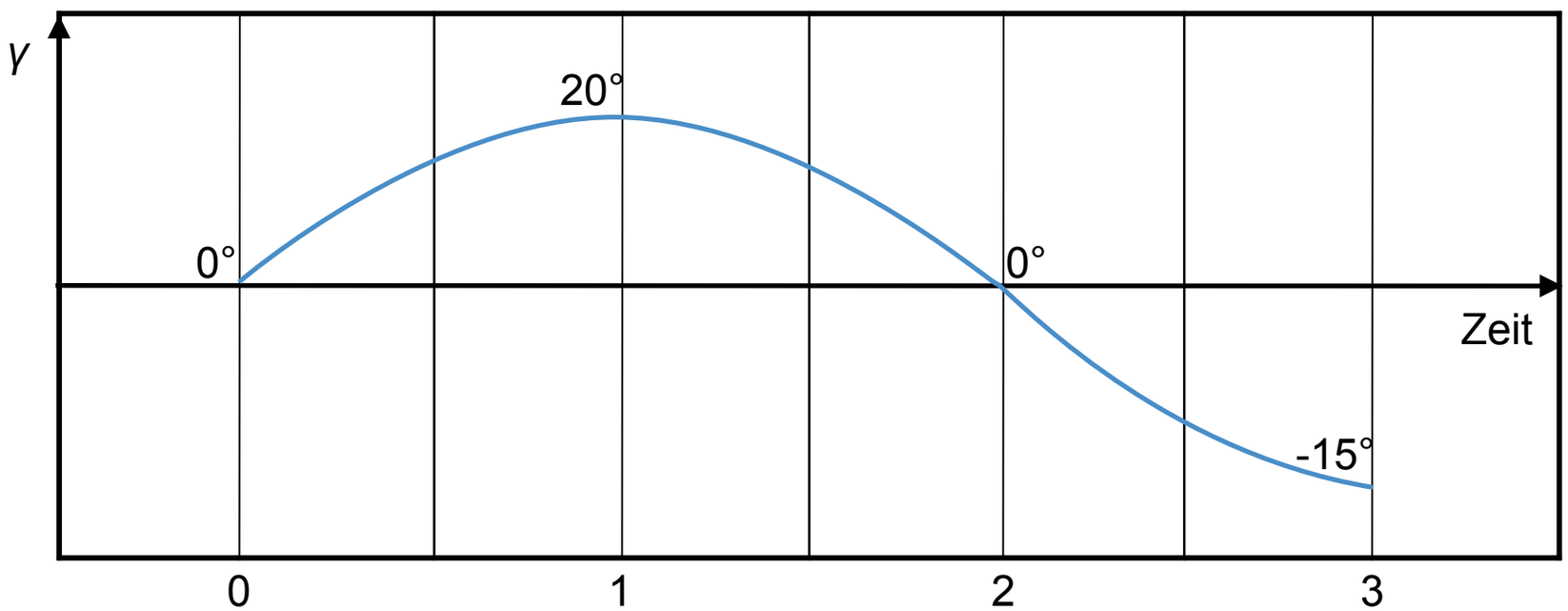
$$\varphi(t) = -sin(\omega \cdot t)$$

R: rotation matrix
P: coordinates of the 3D model
ω: angular velocity
t: time
φ: angle

For multi-body animations, the single animations for all bodies must be created independently an linked to each other

Example





Animation of the knee

$$T_K = R_y(\gamma(t)) \cdot P$$

with

$$R_y = \begin{bmatrix} cos(\gamma(t)) & 0 & sin(\gamma(t)) & 0 \\ 1 & 0 & 0 & 0 \\ -sin(\gamma(t)) & 0 & cos(\gamma(t)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\gamma(t) = sin(\omega \cdot t)$$

R: rotation matrix
P: coordinates of the 3D model
ω: angular velocity
t: time
φ: angle

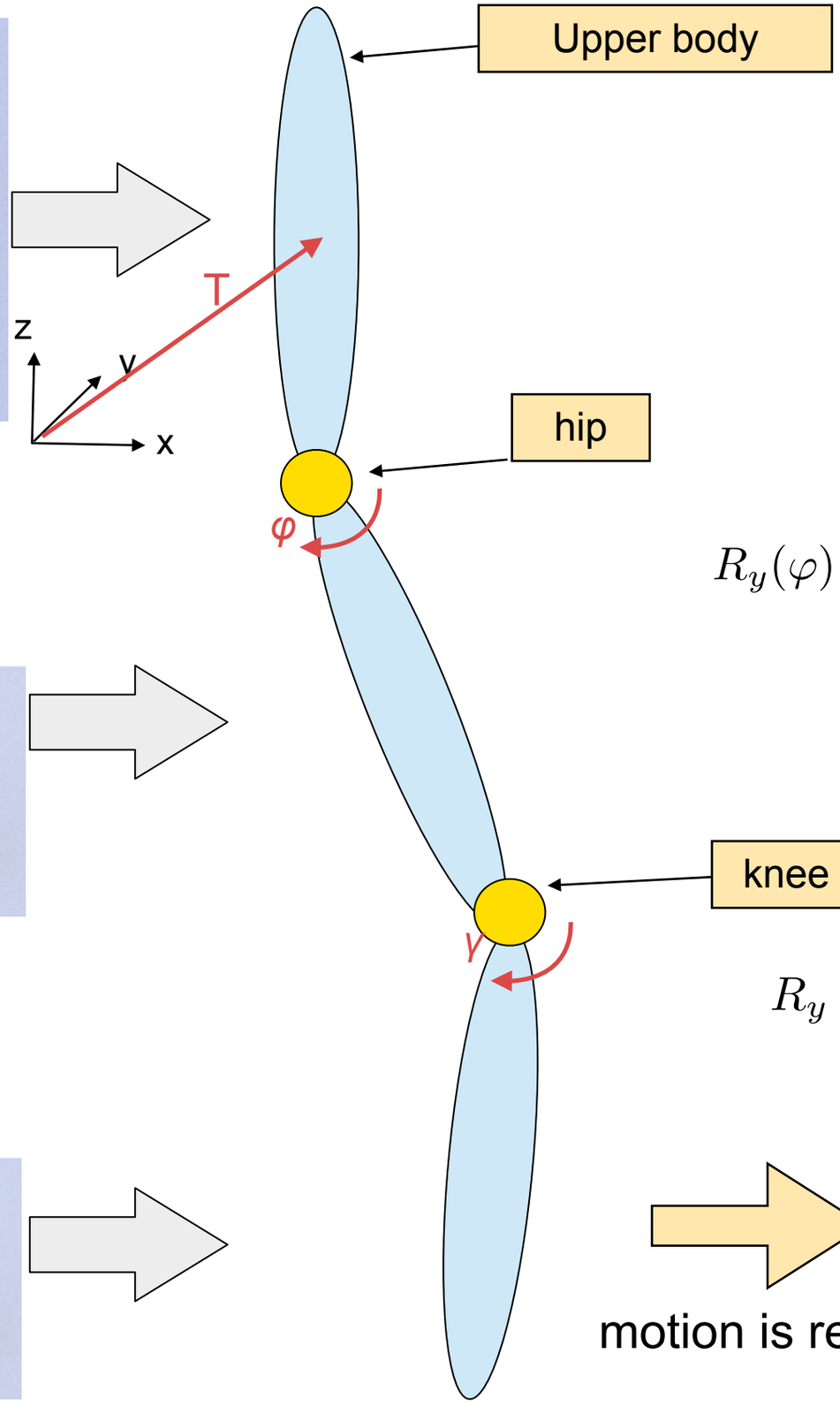
For multi-body animations, the single animations for all bodies must be created independently an linked to each other

Hierarchy of Animation

upper body

hip + thigh

knee + lower leg



$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\varphi) = \begin{bmatrix} cos(\varphi(t)) & 0 & sin(\varphi(t)) & 0 \\ 1 & 0 & 0 & 0 \\ -sin(\varphi(t)) & 0 & cos(\varphi(t)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} cos(\gamma(t)) & 0 & sin(\gamma(t)) & 0 \\ 1 & 0 & 0 & 0 \\ -sin(\gamma(t)) & 0 & cos(\gamma(t)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T \cdot R_x \cdot R_y$$

motion is related to each other

Rigid body

knee

Video

VRAC | HCI

VRAC | HCI

Animations in OpenGL

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

ToDo

Preparation:

- **Representing the keyframes**

During runtime:

- Calculate the current time fraction
- Find the two closest keyframes
- Interpolate the current position and orientation

Representing the keyframes

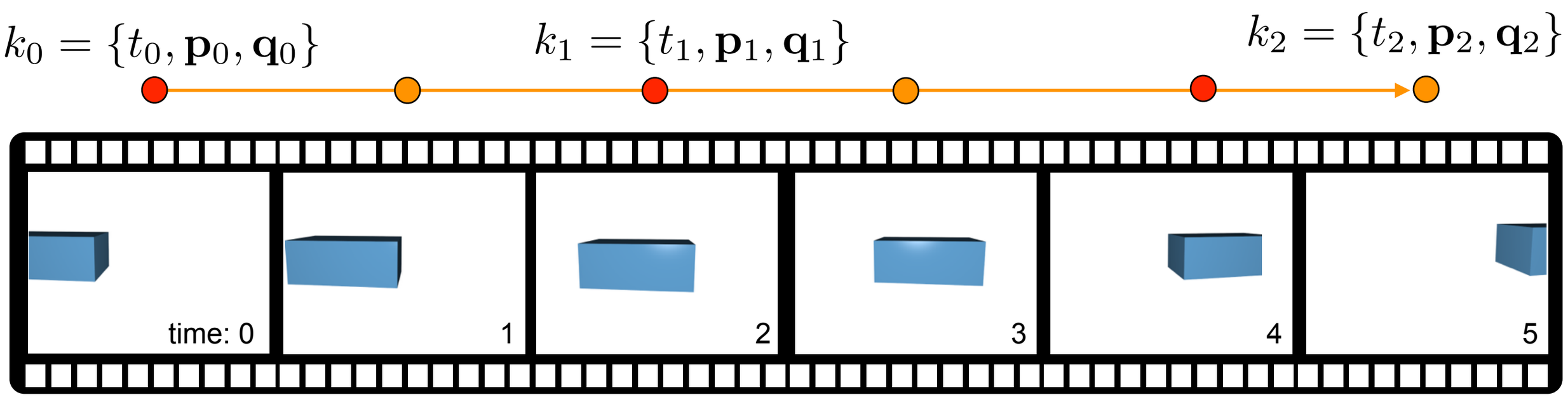
For every keyframe k_i we store

the time fraction in an interval between 0 and 1

the position as vector with {x, y, z} coordinates

the orientation of the object as quaternion q = {x, y, z, w}

$$k_i = \{t_i, \mathbf{p}_i, \mathbf{q}_i\}$$



C++ map

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

```
template < class Key, class T > class map
```

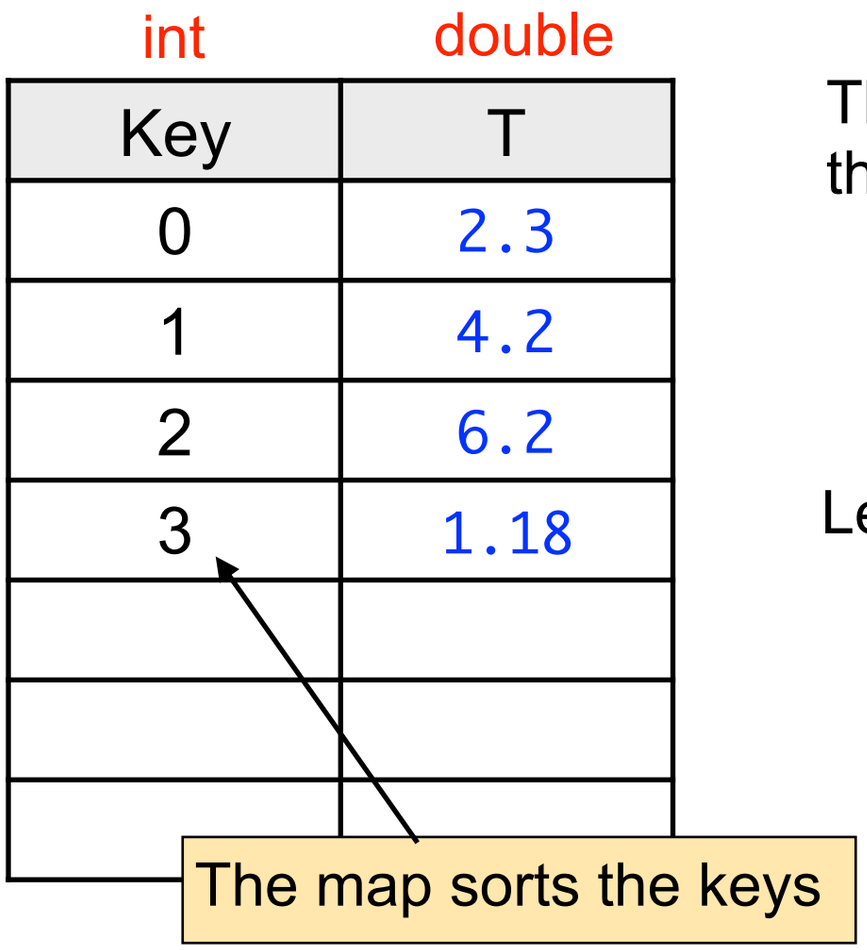
Parameter:

- Key: Type of the keys. Each element in a map is uniquely identified by its key value
- T: Type of the mapped value. Each element in a map stores some data as its mapped value.

We can request / search for data using an identifier.

Key	T
Identifier	Data

C++ map



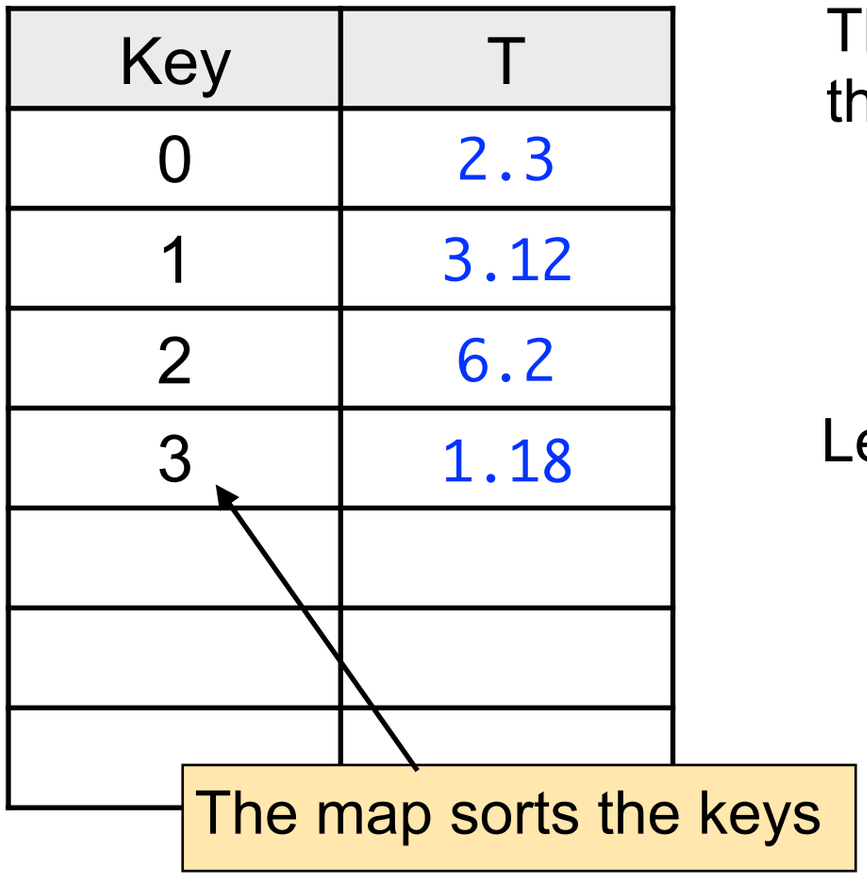
This defines a map where each key is an integer, the data is a double value:

```
map<int, double> my_map_01;
```

Let's add some data:

```
my_map_01[0] = 2.3;
my_map_01[2] = 6.2;
my_map_01[1] = 4.2;
my_map_01[3] = 1.18;
```

C++ map



This defines a map where each key is an integer, the data is a double value:

```
map<int, double> my_map_01;
```

Let's add some data:

```
my_map_01[0] = 2.3;
my_map_01[2] = 6.2;
my_map_01[1] = 4.2;
my_map_01[3] = 1.18;
```

Keys are unique. If we call them again, we overwrite the value

```
my_map_01[1] = 3.12;
```

C++ map

float	struct Keyframe
Key	T
0.0	2.3
0.1	4.2
0.2	6.2
0.3	1.18
...	
...	

This defines a map where each key is an integer, the data is a double value:

```
map<float, Keyframe> my_keyframes;
```

Let's add some data:

```
my_keyframes[0.0] = ...;
my_keyframes[0.1] = ...;
my_keyframes[0.2] = ...;
my_keyframes[0.3] = ...;
```

C++ - struct

A struct - a data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths.

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

Access to struct elements

writing:

```
object_names.member_name1 = value;
```

reading

```
type value = object_names.member_name1
```

Use the point operator (.) to read and write from and to struct objects.

```
/*!
 A struct to define keyframe
 */
typedef struct _keyframe
{
    float                   _t; // the time fraction
    glm::vec3               _p; // the position
    glm::quat               _q; // the orientation

    /*
     Constructor
     */
    _keyframe(float t, glm::vec3 p, glm::quat q)
    {
        _t = t;
        _p = p;
        _q = q;
    }

    /*
     Default constructor
     */
    _keyframe()
    {
        _t = -1.0;
        _p = glm::vec3(0.0,0.0,0.0);
        _q = glm::quat(0.0,0.0,0.0,0.0);
    }

}Keyframe;
```

Keyframe data

Example Keyframe struct

C++ - typedef

The operator ***typedef*** allows us to declare a type alias.

A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

```
typedef existing_type new_type_name ;
```

Example:

```
typedef double MyDouble;
MyDouble v = 5.0;
```

```
typedef struct _keyframe{
 ....
}Keyframe
```

Existing type

New type

Example Code

```
/*
 Type for the keyframe animation
 */
typedef map<double, Keyframe> KeyframeAnimation;

// Variable to store the keyframes
KeyframeAnimation myKeyframes;


/*!
 This initializes the keyframes.
 */
void initKeyframeAnimation(void)
{
    myKeyframes[0.0] = Keyframe(0.0, glm::vec3(0.0,0.0,0.0),
                                     angleAxis(0.0f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.5] = Keyframe(0.5, glm::vec3(0.5,0.0,0.0),
                                     angleAxis(0.57f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.7] = Keyframe(0.7, glm::vec3(0.7,0.5,0.0),
                                     angleAxis(1.28f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.8] = Keyframe(0.8, glm::vec3(0.35,0.7,0.0),
                                     angleAxis(1.53f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[1.0] = Keyframe(1.0, glm::vec3(0.2,0.9,0.0),
                                     angleAxis(1.98f, glm::vec3(0.0,0.1,1.0) ) );
}
```

Keyframe

Time

Storing Keyframes

ARLAB

```
void initKeyframeAnimation(void)
{
    myKeyframes[0.0] = Keyframe(0.0, glm::vec3(0.0,0.0,0.0),
                                     angleAxis(0.0f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.5] = Keyframe(0.5, glm::vec3(0.5,0.0,0.0),
                                     angleAxis(0.57f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.7] = Keyframe(0.7, glm::vec3(0.7,0.5,0.0),
                                     angleAxis(1.28f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[0.8] = Keyframe(0.8, glm::vec3(0.35,0.7,0.0),
                                     angleAxis(1.53f, glm::vec3(0.0,0.0,1.0) ) );
    myKeyframes[1.0] = Keyframe(1.0, glm::vec3(0.2,0.9,0.0),
                                     angleAxis(1.98f, glm::vec3(0.0,0.1,1.0) ) );
}
```

Keyframe

1

Time: must be equal

1. Use always keyframe time values in a range [0, 1].

   Multiply the keyframe value with the animation duration during runtime. This facilitates animation time changes.

```
// The animation time in seconds
GLfloat animation_duration = 8.5

GLfloat current_time_fraction =
  current_time /  animation_duration;
```

ToDo

Preparation:

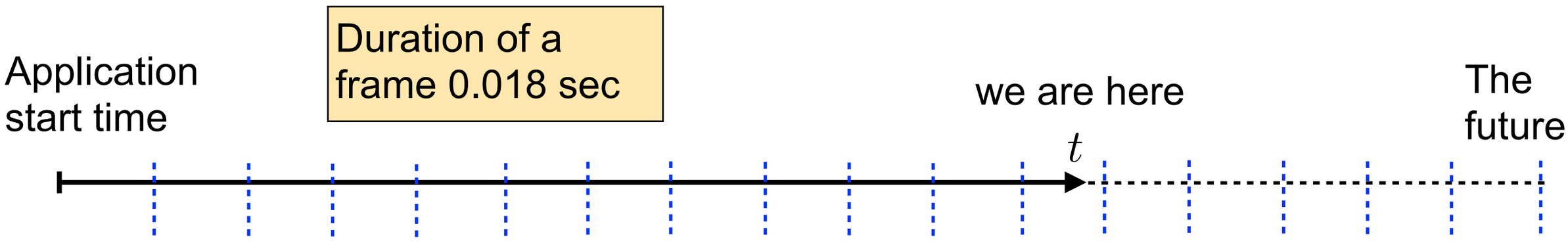
- Representing the keyframes

During runtime:

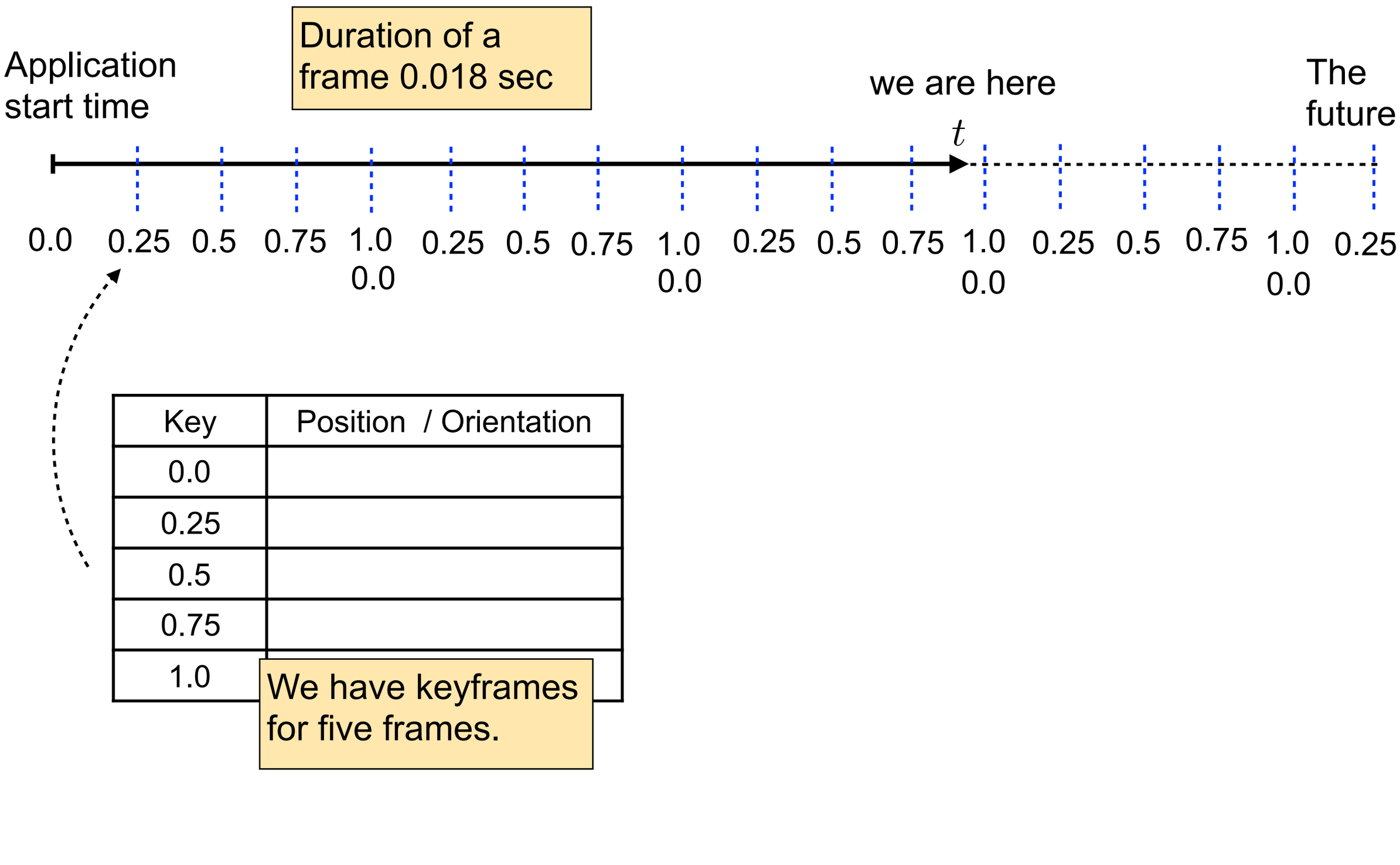
- **Calculate the current time fraction**
- Find the two closest keyframes
- Interpolate the current position and orientation

Timeline

Timeline


Application
start time
Duration of a
frame 0.018 sec
we are here
t
The
future

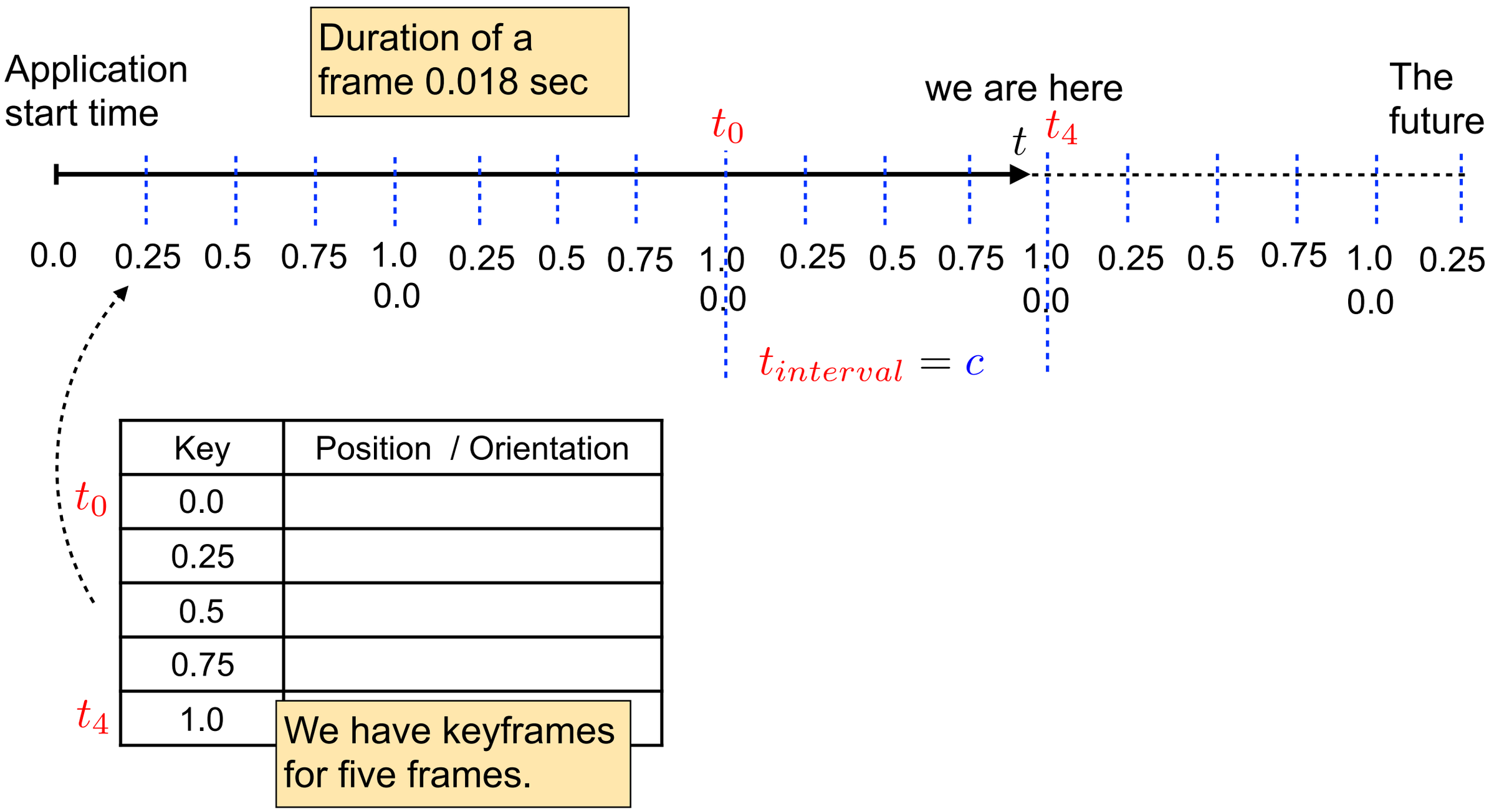
Timeline



Key	Position / Orientation
0.0	
0.25	
0.5	
0.75	
1.0	

We have keyframes for five frames.

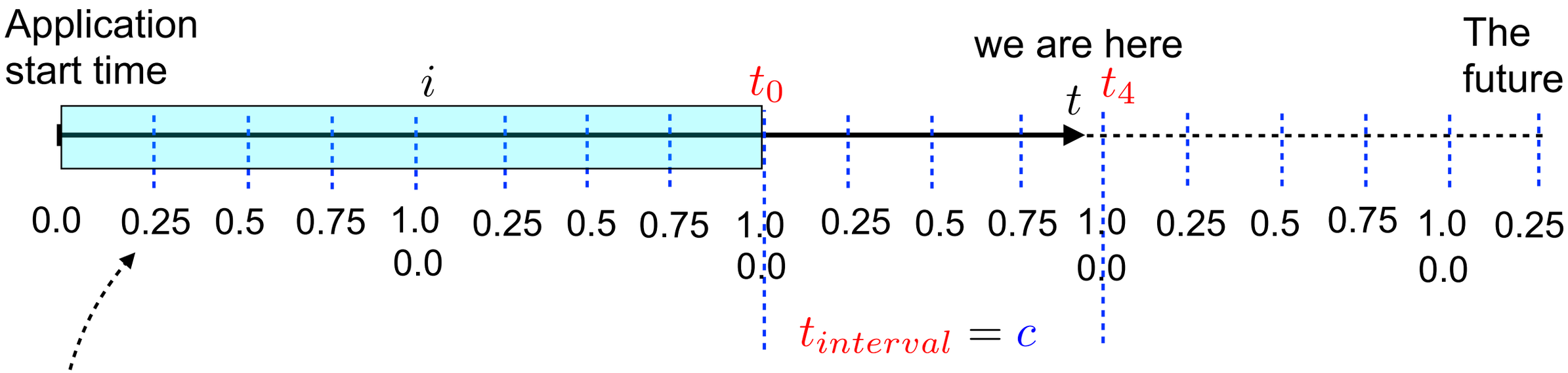
Timeline



	Key	Position / Orientation
t_0	0.0	
	0.25	
	0.5	
	0.75	
t_4	1.0	

We have keyframes for five frames.

Timeline



Key	Position / Orientation
0.0	
0.25	
0.5	
0.75	
1.0	

t_0

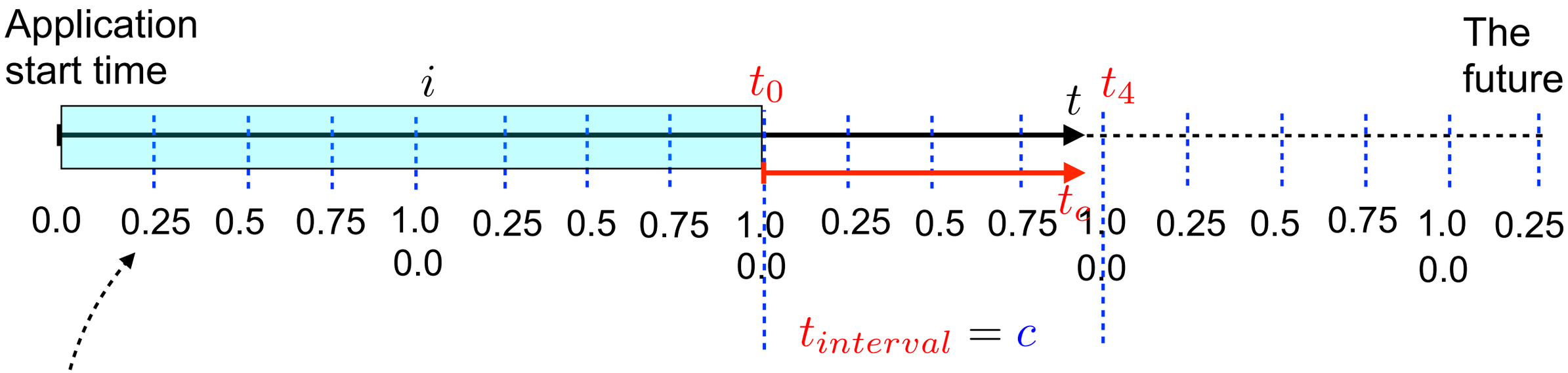
t_4

We have keyframes for five frames.

1. Calculate the number of past intervals *i*

$$i = \left\lfloor \frac{t}{t_{interval}} \right\rfloor$$

Timeline



Key	Position / Orientation
0.0	
0.25	
0.5	
0.75	
1.0	

t_0

t_4

We have keyframes for five frames.

1. Calculate the number of past intervals *i*

$$i = \left\lfloor \frac{t}{t_{interval}} \right\rfloor$$

2. Calculate the time in the current interval:

$$t_c = t - t_{interval}\ i$$

3. Return the fraction as values between [0, 1]

$$f = \frac{t_c}{t_{interval}}$$

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

Code Example

ARLAB

```
/*!
 @brief returns the time fraction for a given time and animation duration
 @param time - the current animation time, application runtime, etc. in seconds
 @param duration - the duration of the animation in seconds
 @return the time fraction in an interval between 0 and 1.
 */
float getTimeFraction(const float time, const float duration)
{
    // we cast to an int. this results in the number of
    float interval = floor(time/duration);

    // return the current interval time
    float current_interval = time - interval*duration;

    // return the fraction / position in our current timeline
    float fraction = current_interval / duration;

    return fraction;
}
```

Usage:

```
float f = getTimeFraction(time, 8.0); // we assume that the animation takes 8 seconds
```

time: current application runtime in seconds.

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

ToDo

Preparation:

- Representing the keyframes

During runtime:

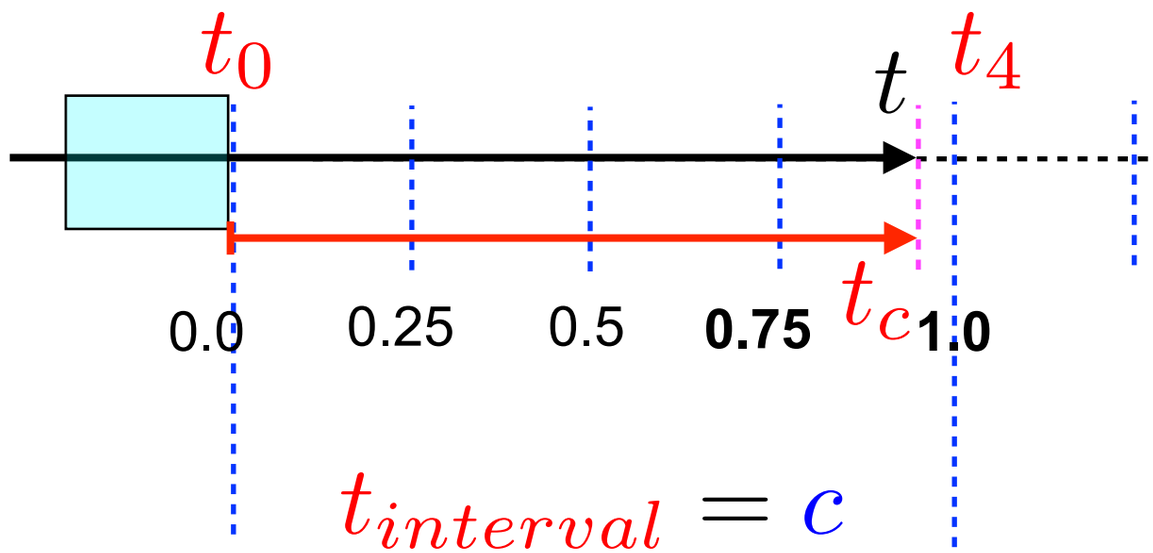
- Calculate the current time fraction
- **Find the two closest keyframes**
- Interpolate the current position and orientation

Find the two closest keyframes

ARLAB

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	



We already know:

- the current time fraction f
- all keyframes are stored in a map `myKeyframes`

We need:

- two keyframes that encase our f

We use a C++ map function: lower_bound to find the closest keyframe

lower_bound

```
iterator std::map::lower_bound(const key_type& k)
```

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	

Examples:

- for key 0.8: it returns the iterator that points to 1.0

lower_bound


```
iterator std::map::lower_bound(const key_type& k)
```

Returns an iterator pointing to the first element that is not less than the key *k*.

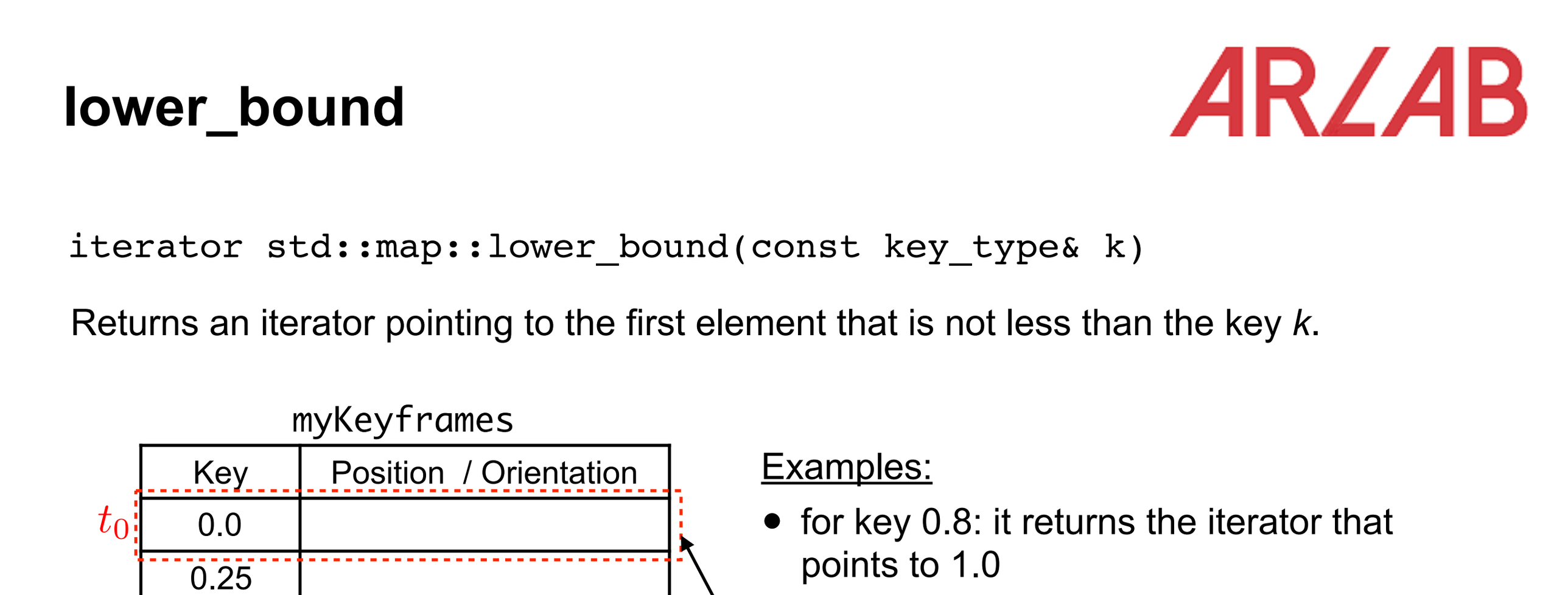
myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	

Examples:

- for key 0.8: it returns the iterator that points to 1.0
- 0.5111 returns 0.75

lower_bound




```
iterator std::map::lower_bound(const key_type& k)
```

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	

Examples:

- for key 0.8: it returns the iterator that points to 1.0
- 0.5111 returns 0.75
- 0.0 returns 0.0

lower_bound


```
iterator std::map::lower_bound(const key_type& k)
```

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	

Examples:

- for key 0.8: it returns the iterator that points to 1.0
- 0.5111 returns 0.75
- 0.0 returns 0.0
- 1.0 returns 1.0

lower_bound


```
iterator std::map::lower_bound(const key_type& k)
```

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
f	0.5	
	0.75	
t_4	1.0	

Examples:

- for key 0.8: it returns the iterator that points to 1.0
- 0.5111 returns 0.75
- 0.0 returns 0.0
- 1.0 returns 1.0
- 1.1 returns ?

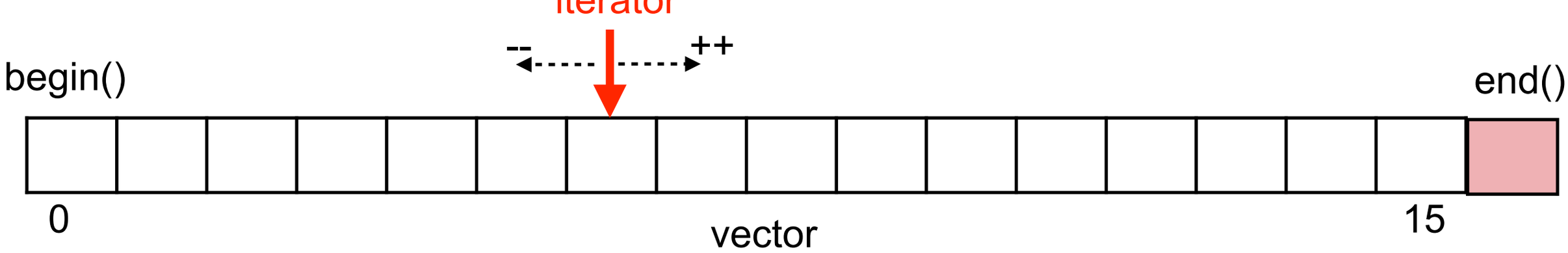
```
// get a keyframe iterator
KeyframeAnimation::iterator k_itr = keyframes.lower_bound(time);
```

C++ Container Iterators

C++ containers are all datatypes that store data:

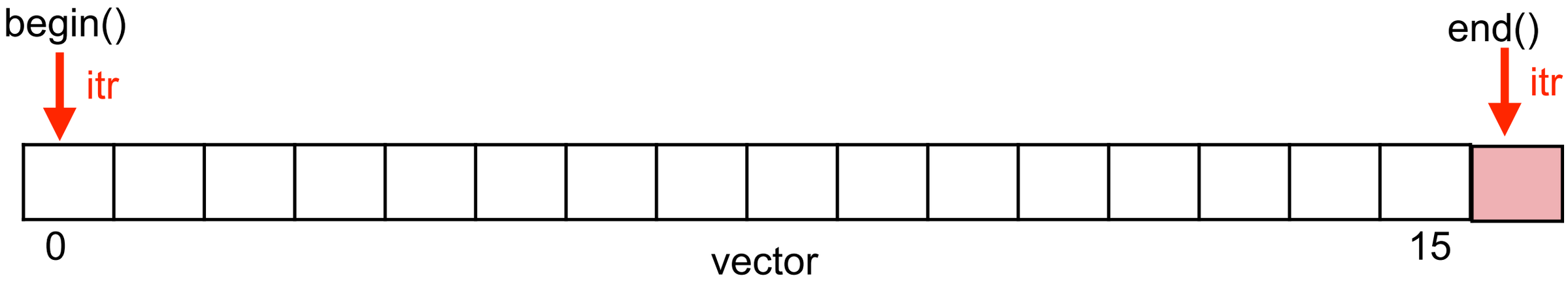
set, list, vector, map, multimap, stack, queue, etc.

An **Iterator** is a pointer which allows us to navigate / iterate through containers using operators such as ++ and --



Advantage: Iterators are faster than other access operators since you only move a pointer

C++ Container Iterators



begin()

The function begin returns an interator that points to the first element

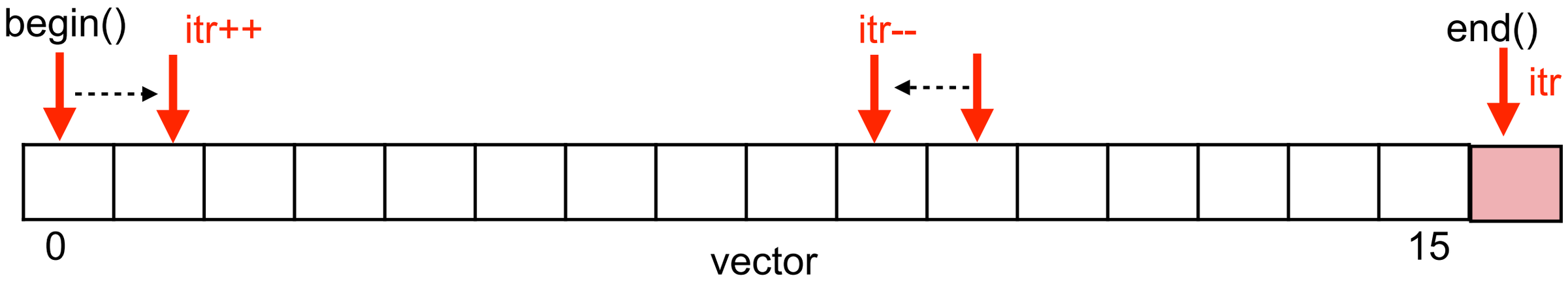
```
map<double, int> data;

map<double, int>::iterator itr = data.begin();
```

end()

The function end returns an interator that points beyond the last element.
It does not exist!! Do not read from this element. It is required to check whether you move within the container's bounds.

```
map<double, int>::iterator itr = data.end();

map<double, int> data;
```

C++ Container Iterators




```
map<double, int> data;

map<double, int>::iterator itr = data.begin();
```

operator ++

The operator ++ allows you to move one element forward per call.

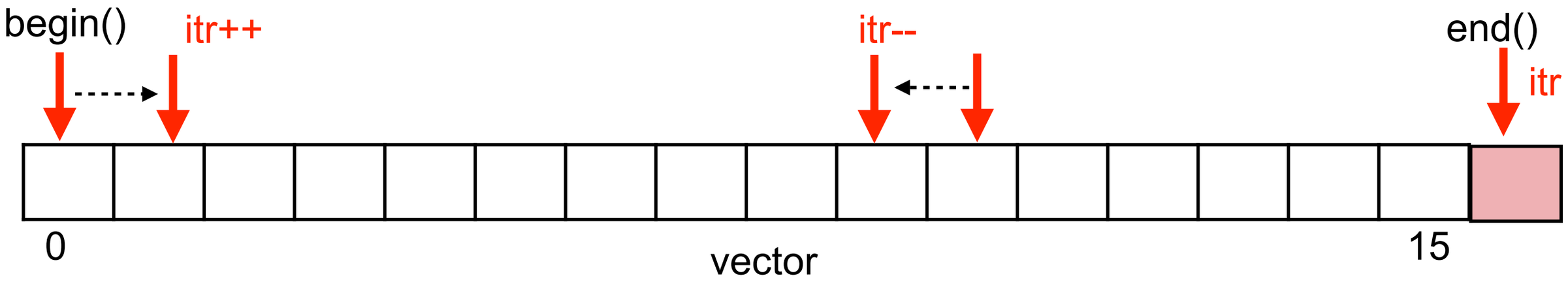
```
itr++;
```

operator --

The operator -- allows you to move one element backward per call.

```
itr--;
```

C++ Container Iterators




```
map<double, int> data;

map<double, int>::iterator itr = data.begin();
```

Access data

The operator is a pointer element. You must dereference it to access the data

```
double this_double = (*itr).first;

int this_int = (*itr).second;
```

The first / second operator is required for containers with more than one entry.

```
map<double, int> data;
```

Example

This is a typical algorithm that allows you to run through all elements.

```
map<double, int> data;

   [add some data to data]

map<double, int>::iterator itr = data.begin();
while(itr != data.end())
{

   double this_double = (*itr).first;
   int this_int = (*itr).second;

   itr++;
}
```

lower_bound

ARLAB

`iterator std::map::lower_bound(const key_type& k)`

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
	0.5	
	0.75	
t_4	1.0	k1

f

Examples:

- for a fraction *f* = 0.8

Lower bound returns the element that is larger than *f* or equal to *f*.

```
// get a keyframe iterator
KeyframeAnimation::iterator k_itr = keyframes.lower_bound(time);

// Obtain the first keyframe
k1 = (*k_itr).second; num_keyframes++;
```

lower_bound

ARLAB

`iterator std::map::lower_bound(const key_type& k)`

Returns an iterator pointing to the first element that is not less than the key *k*.

myKeyframes

	Key	Position / Orientation
t_0	0.0	
	0.25	
	0.5	
	0.75	k0
f t_4	1.0	k1

Examples:

- for a fraction *f* = 0.8

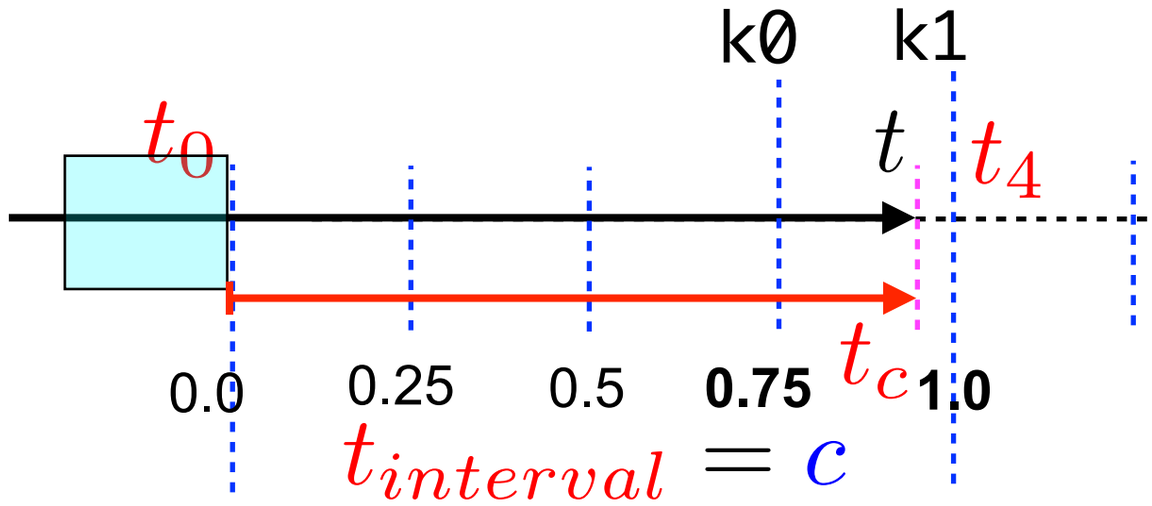
We take the element before k1 as the start point.

```
// Check whether we are not at the beginning of this map
if(k_itr != keyframes.begin())
{
    k_itr--;  // decrement
    k0 = (*k_itr).second; // obtain the second keyframe
    num_keyframes++;
}
```

```
int getKeyframes( KeyframeAnimation& keyframes,
const double time, Keyframe& k0, Keyframe& k1)
{
    int num_keyframes = 0;

    // get a keyframe iterator
    KeyframeAnimation::iterator k_itr = keyframes.lower_bound(time);

    Keyframe k0_temp, k1_temp;

    // Obtain the first keyframe
    k1 = (*k_itr).second; num_keyframes++;


    // Check whether we are not at the beginning of this map
    if(k_itr != keyframes.begin())
    {
        k_itr--;  // decrement
        k0 = (*k_itr).second; // obtain the second keyframe
        num_keyframes++;
    }

    // write the first keyframe into k0
     if we only have one
    if(num_keyframes == 1)
    {
        k0 = k1;
    }

    return num_keyframes;
```

ARLAB

Example



VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

ToDo

Preparation:

- Representing the keyframes

During runtime:

- Calculate the current time fraction
- Find the two closest keyframes
- **Interpolate the current position and orientation**

Representing the keyframes

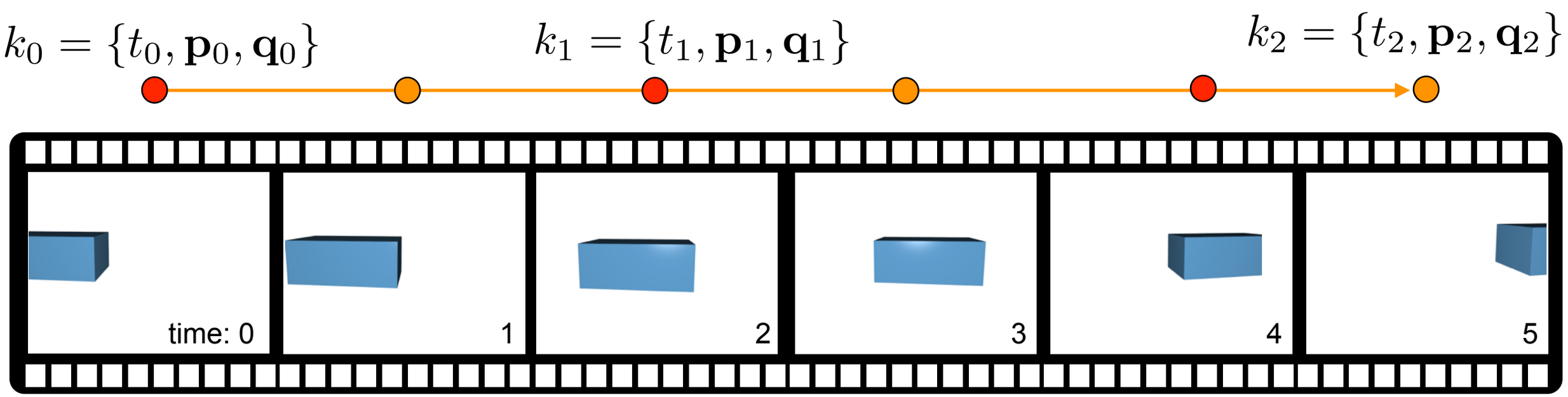
For every keyframe k_i we store

the time fraction in an interval between 0 and 1

the position as vector with {x, y, z} coordinates

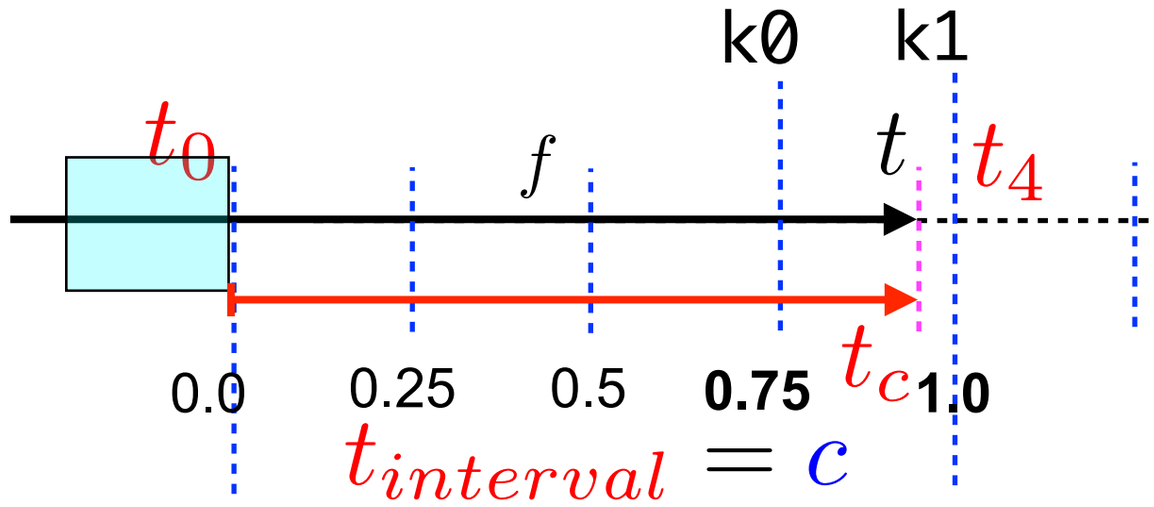
the orientation of the object as quaternion q = {x, y, z, w}

$$k_i = \{t_i, \mathbf{p}_i, \mathbf{q}_i\}$$



Linear Interpolation of the Position

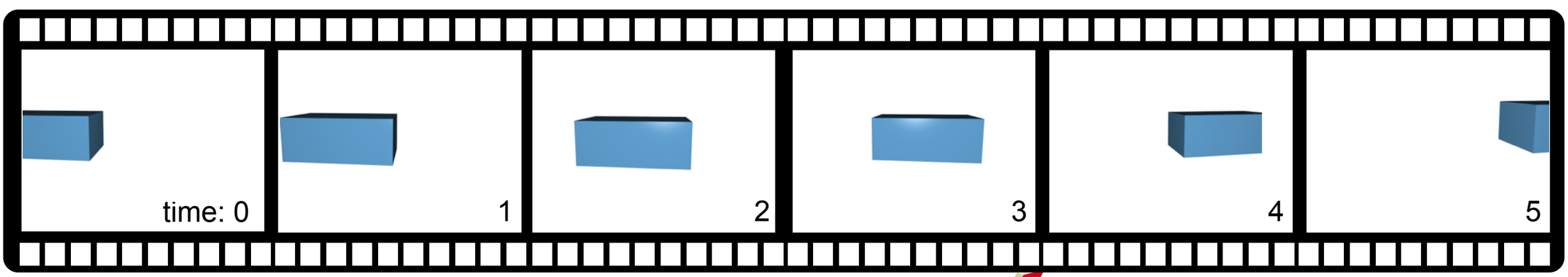
$$\mathbf{p}_{int} = \mathbf{p}_i + (\mathbf{p}_{i+1} - \mathbf{p}_{i+1}) \cdot \frac{f - t_i}{t_{i+1} - t_i}$$



with

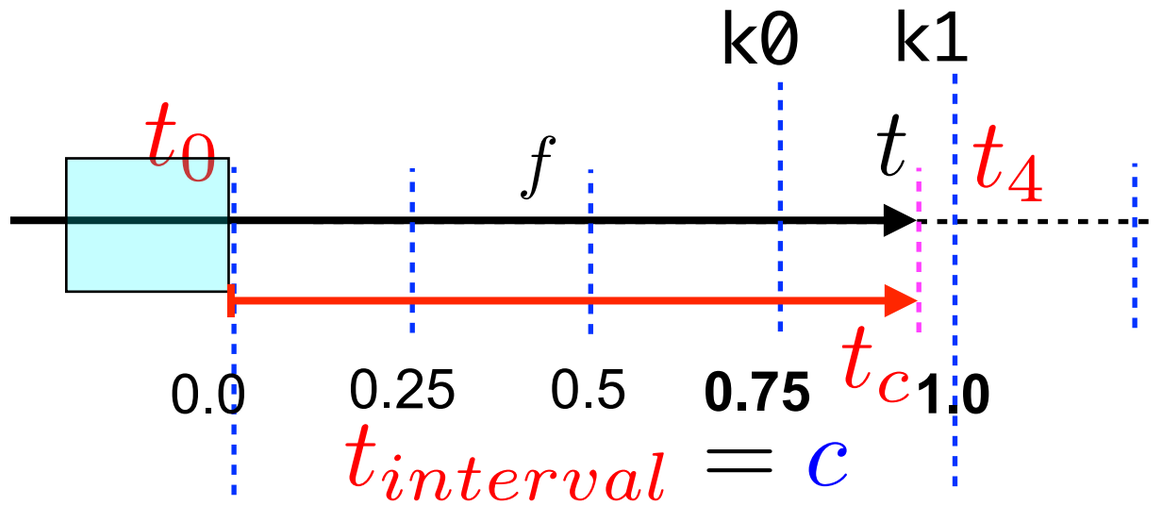
$\frac{f - t_0}{t_1 - t_i}$ to scale the value f into our current keypoint interval

$k_0 = \{t_0, \mathbf{p}_0, \mathbf{q}_0\}$ $k_1 = \{t_1, \mathbf{p}_1, \mathbf{q}_1\}$ $k_2 = \{t_2, \mathbf{p}_2, \mathbf{q}_2\}$



Linear Interpolation of the Rotation

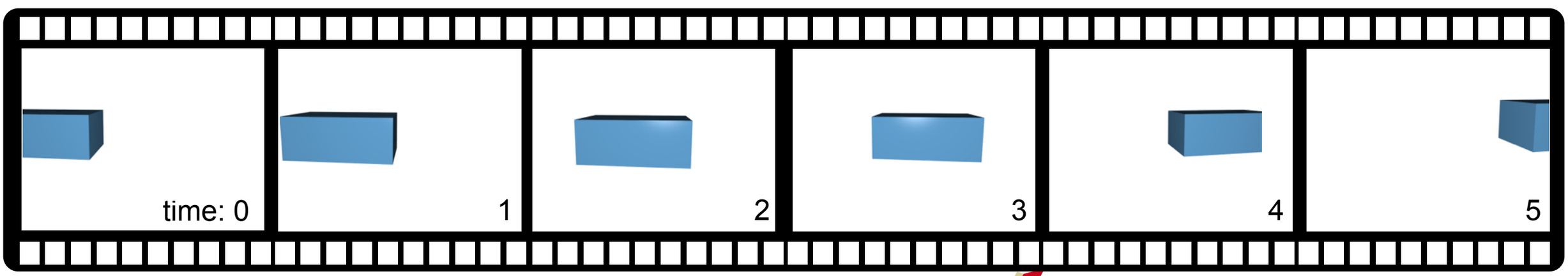
$$slerp(\mathbf{r}_0, \mathbf{r}_1, \frac{f - t_0}{t_1 - t_i})$$



with

$\frac{f - t_0}{t_1 - t_i}$ to scale the value f into our current keypoint interval

$k_0 = \{t_0, \mathbf{p}_0, \mathbf{q}_0\}$ $k_1 = \{t_1, \mathbf{p}_1, \mathbf{q}_1\}$ $k_2 = \{t_2, \mathbf{p}_2, \mathbf{q}_2\}$



Example

```cpp
bool interpolateKeyframe(const float fraction, const Keyframe& k0, const Keyframe& k1, Keyframe& res)
{
    ////////////////////////////////////////////////////////////////////////
    // 1. Check the time delta

    // delta time
    float delta_t = k1._t - k0._t;

    // Check whether we have a delta time. Otherwise, we are at the location of exactly one keyframe
    if(delta_t == 0.0f){
        res = k0;
        return true;
    }

    ////////////////////////////////////////////////////////////////////////
    // 2. Interpolat the position

    // get the delta
    glm::vec3 delta_p = k1._p - k0._p;

    // position interpolation
    glm::vec3 p_int =  k0._p + delta_p * (fraction- k0._t)/(delta_t);

    ////////////////////////////////////////////////////////////////////////
    // 3. Rotation interpolation

    // Calculate the distance between the target angle and the current angle.
    float delta_angle = sqrt((k1._q.x - k0._q.x)*(k1._q.x - k0._q.x)  +
                             (k1._q.y - k0._q.y)*(k1._q.y - k0._q.y)  +
                             (k1._q.z - k0._q.z)*(k1._q.z - k0._q.z)  +
                             (k1._q.w - k0._q.w)*(k1._q.w - k0._q.w));


    // Linear interpolation of the rotation using slerp
    glm::quat r_int = glm::slerp(k0._q, k1._q, (fraction- k0._t)/(delta_t));

    ////////////////////////////////////////////////////////////////////////
    // 4. Write the result
    res = Keyframe(fraction, p_int, r_int);

    return true;
}
```

Position interpolation

Rotation interpolation

Position interpolation

```
////////////////////////////////////////////////////////////////////////////
// 2. Interpolate the position

// get the delta
glm::vec3 delta_p = k1._p - k0._p;

// position interpolation
glm::vec3 p_int =  k0._p + delta_p * (fraction- k0._t)/(delta_t);
```

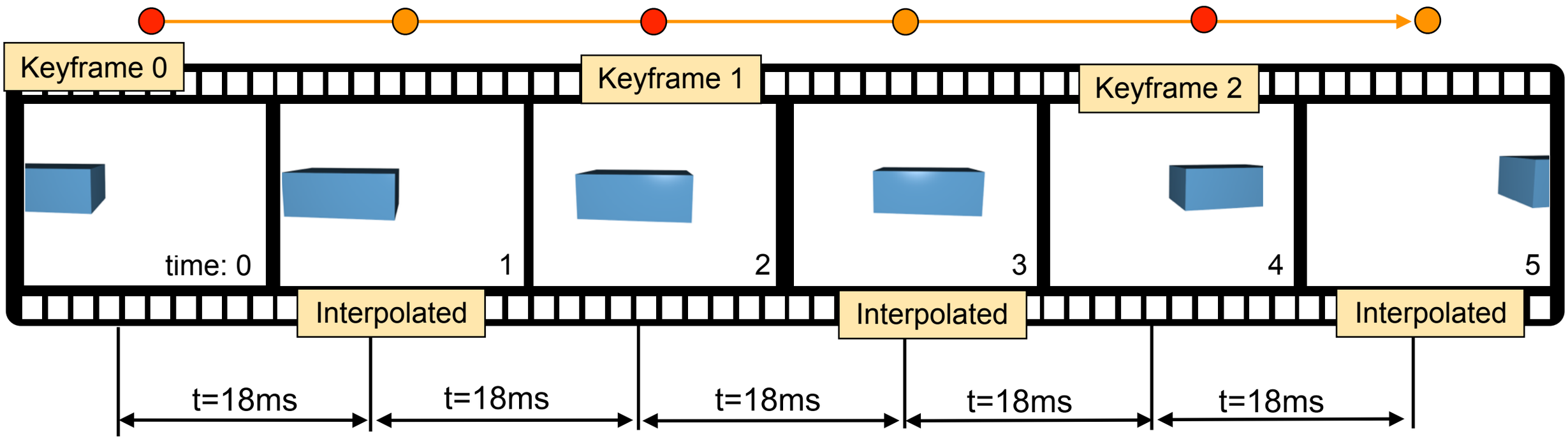
Rotation interpolation

```
////////////////////////////////////////////////////////////////////////////
  // 3. Rotation interpolation

  // Calculate the distance between the target angle and the current angle.
  float delta_angle = sqrt((k1._q.x - k0._q.x)*(k1._q.x - k0._q.x)  +
                           (k1._q.y - k0._q.y)*(k1._q.y - k0._q.y)  +
                           (k1._q.z - k0._q.z)*(k1._q.z - k0._q.z)  +
                           (k1._q.w - k0._q.w)*(k1._q.w - k0._q.w));
```

```
// Linear interpolation of the rotation using slerp
glm::quat r_int = glm::slerp(k0._q, k1._q, (fraction- k0._t)/(delta_t));
```

During runtime

Consistent Time Update



Animations depend on a timer function. GLFW returns the time in seconds with:

```
double time  glfwGetTime();
```

Runtime

```
int main(int argc, const char * argv[])
{
```

Program init

Prepare the keyframes

main loop

```
    while
    {
        // Interpolate between keyframes
        Keyframe k0, k1, k_res;
        float time = glfwGetTime();
        float f = getTimeFraction(time, 8.0);
        int num = getKeyframes(myKeyframes, f, k0, k1);
        bool ret = interpolateKeyframe(f, k0, k1, k_res);
```

Render the object

```
        draw object
    }
}
```

Video

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY