Hierarchical Graphics and Animation

Angel Ch.8,
Watt and Watt Ch.16

Hierarchical Models

Hierarchical models used to represent complex objects
- explicit dependency between sub-parts of an object
- object-oriented approach to implementation
- eg Articulated objects (robot arm)

Scene hierarchical uses to represent all objects in as a hierachy
- shapes/lights/viewpoints/transforms/attributes
- 'Scene Graph'

Scenes can be represented non-hierarchically
- leads to difficulties in scaling to large scale complex scenes
- all functions explicit in *display()* function
- inflexible

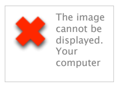
Design of graphics systems with multiple objects
- hierarchical models
- object-oriented design
- scene graphs

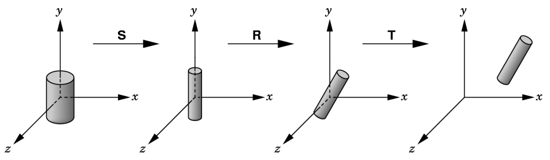
Non-Hierarchical Modelling

- Treat object independently
- reference object by a unique symbol ie a,b,c….

Object initially defined in local object coordinates

Transform each object instance from local to world coordinates:





OpenGL display function:

```
display(){
    .....
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(...);
    glRotatef(...);
    glScalef(...);
    draw_object();
    .....
};
```

- All objects are treated independently
- display() function transforms/draws each object explicitly
- No interrelations between objects

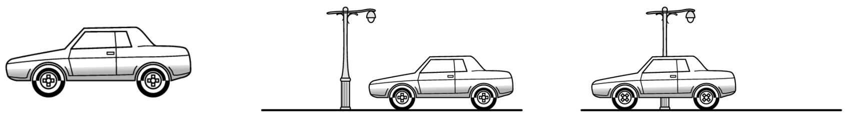
Can represent objects by a table structure:

- each object has a symbol
- each object has corresponding translation/rotation/scale
- each object has set of attributes colour/material properties etc.
- render object by calling drawing each symbol in turn with specified transformation/attributes

Symbol	Scale	Rotate	Translate
1	s_x, s_y, s_z	u_x, u_y, u_z	d_x, d_y, d_z
2			
3			
1			
1			
.			
.			

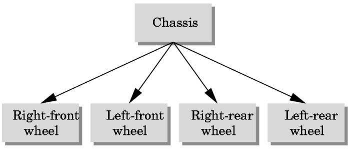
Hierarchical Models

Consider a more complex model composed of several sub-objects
car = chassis + 4 wheels

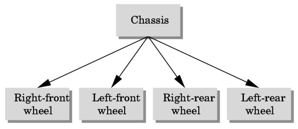


Representation 1: Treat all parts independently (non-hierarchical)
- apply transformation to each part independently
  - *chassis: translate, draw chassis*
  - *wheel 1: rotate, translate, draw wheel 1*
  - *wheel 2: rotate, translate, draw wheel 2*
  - *....*
- redundant, repeated computation of translate
- no explicit representation of dependence between chasis and wheels

Representation 2: Group parts hierarchically
- exploit relation between parts
- exploit similarity
  ie wheels are identical (just translated)



Graph Structures



Graph Representation
- **nodes:** objects + attributes? + transforms?
- **edges:** dependency between objects
  parent-child relation between nodes

'Directed-Graph' edges have a direction associated with them

Tree - directed graph with no closed-loops
ie cannot return to the same point in the graph
- **'root node'** : no entering edges
- Intermediate nodes have one parent and one or more children
- **'leaf node'** : no children

Parameters such as location & attributes may be stored either in nodes or edges

Example: Robot Arm

Represented by a tree with a single chain

Explicit hierarchical implementation

(i) Base: Rotate about base $R(\theta_1)$

$$M_1 = R(\theta_1)$$

(ii) Upper-arm: translate & rotate

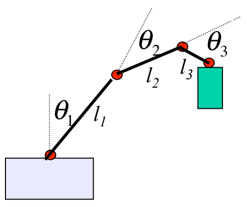
$$M_2 = M_1 T(l_2) R(\theta_2)$$

(iii) lower-arm: translate & rotate

$$M_3 = M_2 T(l_2) R(\theta_3)$$

(iv) end-effector: translate & rotate

$$M_4 = M_3 T(l_3) R(\theta_4)$$



$$\begin{tikzcd}
\boxed{\text{Base}} \arrow[d] \\
\boxed{\text{upper-arm}} \arrow[d] \\
\boxed{\text{lower-arm}} \arrow[d] \\
\boxed{\text{end-effector}}
\end{tikzcd}$$

OpenGL:
```
display(){
    draw_base()
    glRotatef(θ1,0,0,1);
    draw_upperarm();
    glTranslatef(0,l1,0);
    glRotatef(θ2,0,0,1);
    draw_lowerarm();
     .....
}
```

This example demonstrates an explicit hierarchy
- hard-coded in display function
- hierarchy cannot be changed (inflexible)

Object-oriented hierarchical tree data structure

Each node ‘object’ store

(1) Transformation of object M

(2) Pointer to function to draw object

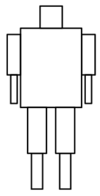
(3) Pointers to children

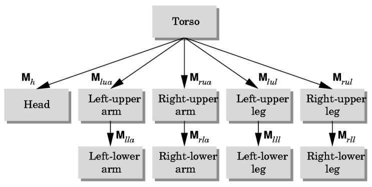
OpenGL psuedo code for single chain tree:
```
display(){
    draw_arm(root);          /* single call to recursive function */
}

draw_arm(node){
    glTransform(node.M);     /* apply model transform */
    node.draw();             /* draw this part */
    draw_arm(node.child);    /* recursive call to children */
}
```

Example: Skeleton

Skeleton is a tree with multiple branches





Represent transformation matricies between each parent and child
- each matrix is the transformation of the object in local coordinates into the parents coordinates

How do we traverse the tree to draw the figure?
- Any order ie depth-first, breadth-first

2 methods to implement traversal:
(1) Stack based - use matrix stack to store required matrices
(2) Recursive - store matrix within nodes of data structure

(1) Stack-based tree traversal
- use matrix stack to store intermediate matrices
- current ModelView matrix M determines position of figure in scene

```
draw_figure(){
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();         /* torso transform */
    draw_torso();
    glTranslatef(...);      /* transform of head relative to torso */
    glRotatef(...);
    draw_head();
    glPopMatrix();          /* restore torso transform */
    glPushMatrix();
    glTranslate();          /* left_arm */
    glRotate();
    draw_upperarm();
    glTranslate();
    glRotate();
    draw_lowerarm();
    glPopMatrix();          /* restore torso transform */
    glPushMatrix();
    glTranslate();          /* right arm */
    ......
}
```

Can also use Push/Pop values from attribute stack ie colour etc.

glPushAttrib();
glPopAttrib();

Limitation of stack-based approach:

- explicit representation of tree in single function
- relies on application programmer to push/pop matrices
- hard-coded/inflexible
  source code must be changed for different hierarchical structure
- no clear distinction between building a model and rendering it

(2) Recursive tree data-structures

- each node is a recursive structure with pointers to children
- use a standard tree structure to represent hierarchy
- render via tree traversal algorithm (independent of model)

C Implementation:

```
typedef struct treenode {
    Glfloat m[16];
    void (*draw)();
    int nchild;
    struct treenode *children;
} treenode;

void draw_tree(treenode *node){
    glPushMatrix();  /* save transform*/
    glMultMatrixf(node->m);
    node->draw();
    for (i=0;i<node->nchild;i++)
        draw_tree(node->children[i]);
    glPopMatrix();  /* restore transform */
}
```

C++ Implementation:

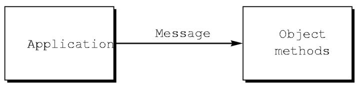
```
class treenode{
   public:
      void draw();
   private:
      Glfloat m[16];
      int nchild;
      treenode *children;
};

void treenode::draw(){
      glPushMatrix();
               .....

      glPopMatrix(); ..
}
```

Graphical Objects and Hierarchies

Represent all objects of a scene in a single hierarchy
- Shape (geometric objects points/lines/polygons…)
- Lights
- Viewer
- Material Properties (attributes)

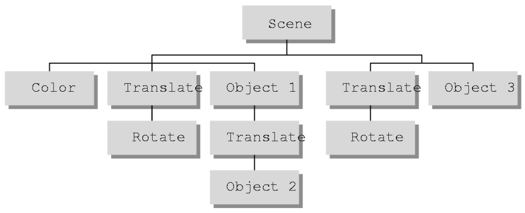


'Object-Oriented' approach
- each object is self-contained module
- Application programmer does NOT have to know internal representation
- Data encapsulation (no external use of pointers to member data)
- interface to access object via methods
- reuse code

Tree-structure to represent complex objects
- reuse primitive object in multiple instances
- represent hierarchical relation (parent-child) between objects
- Use inheritance (C++) to derive complex objects from simple primitives: Object B 'is a' instance of object A
- Examples: Car, skeleton

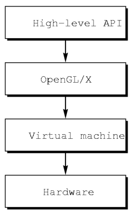
Scene Graphs

Represent all objects in a hierarchy:
- Shape/Lights/Cameras



Scene graph represents explicitly the relationship between objects
- render by traversing the graph
- state attributes/matrices are restored for each branch in graph (Push/Pop)

Object-Oriented Graphics API
- layer on top of OpenGL or other graphics API
- represent scene with a 'scene-graph'
- render the scene graph by tree traveral using OpenGL
- SGI Open Inventor/VRML/ DirectX/Java-3D
- OpenSceneGraph, OpenSG

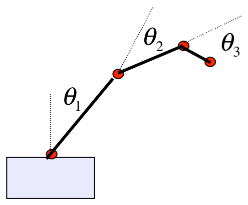


Animation

Articulated Model - Kinematic chain of Rigid Parts
- Control by a small set of parameters (joint angles)

Forward Kinematics
- give a set of joint angle parameters $\vec{\phi}$

$$x_e = f(\vec{\phi}) \\ = M(\theta_1)M(\theta_2)M(\theta_3)M(\theta_4)x$$



Forward kinematic model propagates joint angles information to evaluate the transformation of the end-effector
- single solution for a given set of angles
- no dynamics (forces, mass, inertia)

Widely used to control characters
- joint angles generated manually from **key-frames** interpolation used to fill in intermediate frames
- captured from markers on a real-subject

Avatartool

Inverse Kinematics

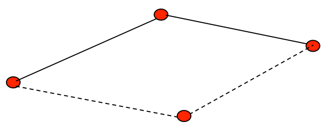
Given a desired end-effector position x_e
what combination of joint angles will produce this position

$$\vec{\phi} = f^{-1}(x_e)$$

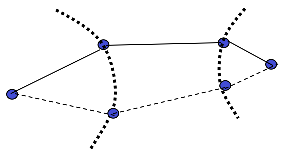
Used for interactive character positioning
ie moving end-effector changes arm joint angles

Problem: Multiple solutions for a given end-effector position
- in general there is no unique inverse

2-Link chain
2 solutions



3-Link Chain
infinite solutions



Solution of Inverse Kinematics Problems for Animation

Consider the forward kinematics equation:

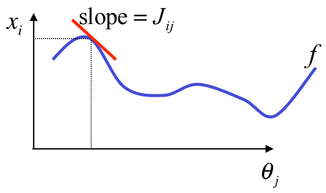
$x_e = f(\vec{\phi})$

x_e – n dimensional vector position of end effector

$\vec{\phi}$ – m dimensional vector of joint angles

Jacobian matrix J is the matrix of partial derivatives relating an infinitesimal change in each of the parameters to the change in end-effector position

$$\Delta x_e = J(\vec{\phi})\Delta\vec{\phi}$$



J is an nxm matrix of partial derivatives

$$J_{ij} = \frac{\partial x_i}{\partial \theta_j} = \frac{\partial f_i(\vec{\phi})}{\partial \theta_j}$$

J_{ij} is the partial derivative of end effector position x_i with respect to angle θ_j

The Jacobian is a local **linear** (first-order) approximation of the highly non-linear function f at a particular set of parameters $\vec{\phi}$

Solution of Inverse Kinematics using the Inverse Jacobian

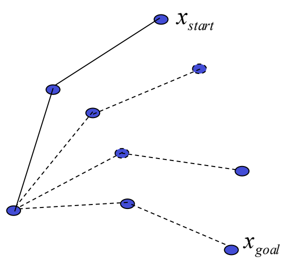
Jacobian provides a local linear approximation of the rate-of-change of end-effector position x with respect to parameters $\vec{\phi}$

Inverse Jacobian is a local approximation of the rate of change of parameters $\vec{\phi}$ with respect to the end effector position x

Use this to interactively move the end-effector position x towards the desired position:

$$x_{new} = x_{current} + \Delta x_e = x_{current} + \Delta(x_{goal} - x_{current})$$

Δ – step length



The corresponding change in step length is given by:

$$\vec{\phi} = f^{-1}(x_e)$$

$$\Delta\vec{\phi} = J^{-1}(\vec{\phi})\Delta x_e$$

$J^{-1}(\vec{\phi})$ is the inverse on an nxm matrix (not square)
- requires **psuedo-inverse** computation

$$\vec{\phi}_{new} = \vec{\phi}_{curr} + \Delta\vec{\phi} = \vec{\phi}_{curr} + J^{-1}(\vec{\phi}_{curr})\Delta x$$

Approximation is only valid locally at $\vec{\phi}$ therefore must take small steps to solution

Example Constructing the Jacobian Matrix J for a 2-link Chain in 2D

Forward kinematic equation:

$$x_e = f(\vec{\phi})$$

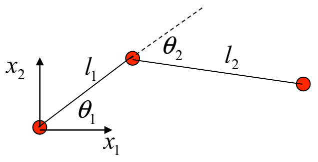
For a 2-link chain in 2 dimensions:

$$x_e = f(\theta_1, \theta_2) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$x_1 = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2)$$

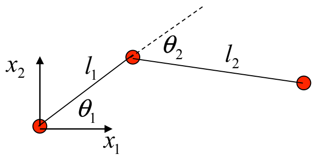
$$x_2 = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2)$$

$$x_e = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2) \end{bmatrix}$$



Example Constructing the Jacobian Matrix J for a 2-link Chain in 2D

$$x_e = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2) \end{bmatrix}$$



Jacobian *J* relating a change in joint angle to a change in end effector position:

$$\Delta x_e = J(\vec{\phi})\Delta\vec{\phi}$$

$$\Delta x_e = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}, \Delta\vec{\phi} = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix}$$

Partial derivative:

$$J_{ij} = \frac{\partial x_i}{\partial \theta_j}$$

$$J_{11} = \frac{\partial x_1}{d\theta_1} = -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 - \theta_2)$$

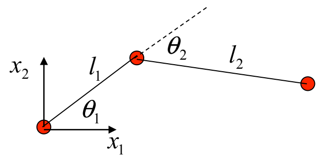
$$J_{12} = \frac{\partial x_1}{d\theta_2} = l_2 \sin(\theta_1 - \theta_2)$$

$$J_{21} = \frac{\partial x_2}{d\theta_1} = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2)$$

$$J_{22} = \frac{\partial x_2}{d\theta_2} = -l_2 \cos(\theta_1 - \theta_2)$$

Example Constructing the Jacobian Matrix J for a 2-link Chain in 2D

$$x_e = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} l_1\cos(\theta_1) + l_2\cos(\theta_1 - \theta_2) \\ l_1\sin(\theta_1) + l_2\sin(\theta_1 - \theta_2) \end{bmatrix}$$



Jacobian J relating a change in joint angle to a change in end effector position:

$$\Delta x_e = J(\vec{\phi})\Delta\vec{\phi}$$

$$\Delta x_e = \begin{bmatrix} \Delta x_{e1} \\ \Delta x_{e2} \end{bmatrix}, \Delta\vec{\phi} = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix}$$

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} = \begin{bmatrix} -l_1\sin\theta_1 - l_2\sin(\theta_1 - \theta_2) & l_2\sin(\theta_1 - \theta_2) \\ l_1\cos\theta_1 + l_2\cos(\theta_1 - \theta_2) & -l_2\cos(\theta_1 - \theta_2) \end{bmatrix}$$

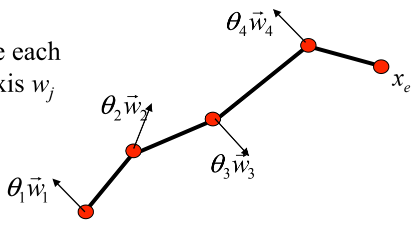
$$\Delta x_e = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} -l_1\sin\theta_1 - l_2\sin(\theta_1 - \theta_2) & l_2\sin(\theta_1 - \theta_2) \\ l_1\cos\theta_1 + l_2\cos(\theta_1 - \theta_2) & -l_2\cos(\theta_1 - \theta_2) \end{bmatrix}\begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix}$$

Geometric Evaluation of Partial Derivatives

Constructing Jacobians algebraically is tedious for complex kinematic chains and trees - more direct geometric approach

Consider a general kinematic chain where each link has a rotation θ_j about a unit length axis w_j

$\theta_j\vec{w}_j$ angle-axis representation of an arbitrary rotation R_j



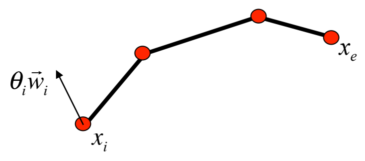
What is the partial derivative:

$$x_e = f(\vec{\varphi})$$
$$J_{ij} = \frac{\partial x_i}{\partial \theta_j}$$

$\frac{\partial x_i}{\partial \theta_j}$ rate-of-change of i^{th} end-effector position coordinate with respect to change in j^{th} joint parameter θ_j

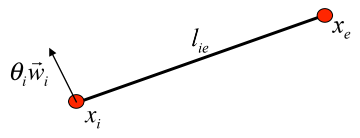
Geometric computation of the Jacobian

Rate-of-change end-effector position wrt parameter θ_i
- depends only on section of chain from joint *i* to the end-effector



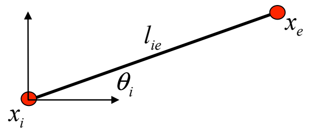
- rigid wrt θ_i (all other degrees of freedom are constant)

Equivalent to having a single rigid link from the i^{th} joint to the end-effector:



Example: 2D Rotation in the plane

Consider the single link in a plane orthogonal to the rotation axis: $\vec{w}.l_{ie} = 0$



Can compute partial derivative for rate-of-change in end effector position wrt the i^{th} joint without considering intermediate joints which are rigid (constant)

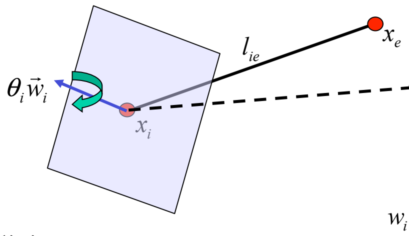
$$x_e = l_{ie}(\cos\theta_i, \sin\theta_i)$$

Note: l_{ie} is constant w.r.t θ_i

$$\frac{\partial x_e}{\partial \theta_i} = l_{ie}(-\sin\theta_i, \cos\theta_i)$$

Geometric computation of Jacobian for general 3D rotation

A general 3D rotation axis *w* is not orthogonal to the link axis



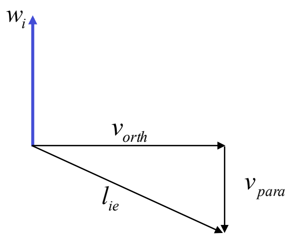
Vector l_{ie} can be split into two components:

component parallel to w_i

$v_{para} = (w_i \cdot l_{ie})w_i$

component orthogonal to w_i

$v_{orth} = l_{ie} - v_{para} = l_{ie} - (w_i \cdot l_{ie})w_i$



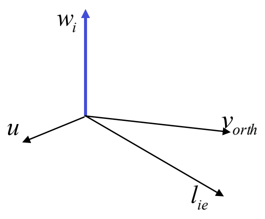
v_{orth} is rotated about w_i by θ_i degrees

$v_{orth\_rot} = R(w_i, \theta_i)v_{orth}$

Note : v_{para} is not changed by rotation about w_i

Therefore :

$Rl_{ie} = Rv_{para} + Rv_{orth} = v_{para} + v_{orth\_rot}$

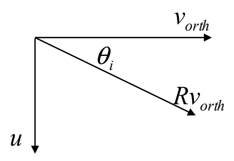


Now consider the vector u orthogonal to w_i and v_{orth} :

$u = w_i \times \mathrm{v}_{\mathrm{orth}} = w_i \times l_{ie}$

Rotation of v_{orth} in the plane orthogonal to w_i *is*

$R(\theta_i, w_i)v_{orth} = v_{orth}\cos(\theta_i) + u\sin(\theta_i)$



Rotation of line l_{ie} :

$$
\begin{aligned}
R(\theta_i, w_i)l_{ie} &= v_{para} + v_{orth\_rot} \\
&= v_{para} + v_{orth}\cos\theta_i + u\sin\theta_i \\
&= (w_i \cdot l_{ie})w_i + (l_{ie} - (w_i \cdot l_{ie})w_i)\cos\theta_i + (w_i \times l_{ie})\sin\theta_i \\
&= l_{ie}\cos\theta_i + (1-\cos\theta_i)(w_i \cdot l_{ie})w_i + (w_i \times l_{ie})\sin\theta_i
\end{aligned}
$$

This is the general expression for the rotation of a vector l_{ie} about an arbitrary 3D axis w_i through angle θ_i

Use this expression to compute the partial derivative of the end-effector postion with respect to the rotation of a specific joint

Note: This expression allows the Jacobian matrix to be computed directly from geometric operations on vectors.

Geometric computation of Jacobian for a kinematic chain

Given the expression for the 3D rotation about an axis w :

$$R(\theta_i, w_i)l_{ie} = l_{ie}\cos\theta_i + (1-\cos\theta_i)(w_i \cdot l_{ie})w_i + (w_i \times l_{ie})\sin\theta_i$$

Can derive the rate of change in end - effector position wrt θ_i

For small θ_i we can make the approximation as $\theta_i \longrightarrow 0 \quad \cos\theta_i \longrightarrow 1 \quad \sin\theta_i \longrightarrow \theta_i$

This gives the approximation :

$$R(\theta_i, w_i)l_{ie} \approx l_{ie} + \theta_i(w_i \times l_{ie})$$

$l_{ie} = (x_e - x_i)$

For incremental changes $\Delta\theta_i = \theta_i w_i$

$$l_{ie\_rot} = l_{ie} + \theta_i(w_i \times l_{ie})$$

$$\Delta x_{ie} = \Delta\theta_i \times l_{ie}$$

This is known as the 'moving axis formula'

- relates an incremental change in the angle to a corresponding change in the end - effector position
- can be used to approximate a column in the Jacobian J_i

Geometric computation of Jacobian for a kinematic chain

Now consider the effect of all joints on the end effector

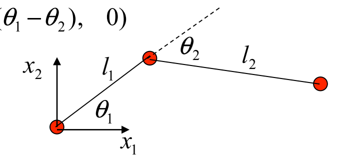
$$\Delta x_{ie} = \sum_{i=0}^{n} \Delta x_{ei} = \sum_{i=0}^{n} \Delta\theta_i \times l_{ie}$$

rate of change of position of the end-effector is the sum over all intermediate frames of the cross-product of the angular rate of change $\Delta\theta_i$ with the vector from the joint centre to the end effector

This is equivalent to the sum of the rate of change in end-effector wrt each joint

Example: of simple 2-link chain in 2D
(see previous Example of analytic computation)

$$l_{1e} = (l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2),\quad l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2),\quad 0)$$
$$l_{2e} = (l_2 \cos(\theta_1 - \theta_2),\quad l_2 \sin(\theta_1 - \theta_2),\quad 0)$$
$$w_i = (0,0,1)$$



$$\Delta\theta_1 = d\theta_1(0,0,1)$$
$$\Delta\theta_2 = d\theta_2(0,0,1)$$

$$\begin{aligned}\Delta x_e &= \Delta\theta_1 \times l_{1e} + \Delta\theta_2 \times l_{2e} \\ &= d\theta_1(0,0,1)\times l_{1e} + d\theta_2(0,0,1)\times l_{2e} \\ &= d\theta_1((-l_1 \sin(\theta_1) - l_2 \sin(\theta_1 - \theta_2),\quad l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2),\quad 0)) \\ &\quad + d\theta_2(l_2 \sin(\theta_1 - \theta_2), -l_2 \cos(\theta_1 - \theta_2), 0)\end{aligned}$$

$$\Delta x_e = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} -l_1 \sin\theta_1 - l_2 \sin(\theta_1 - \theta_2) & l_2 \sin(\theta_1 - \theta_2) \\ l_1 \cos\theta_1 + l_2 \cos(\theta_1 - \theta_2) & -l_2 \cos(\theta_1 - \theta_2) \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix}$$

Sanity check - this is the same as we obtained for the Jacobian by direct differentiation

Interactive Animation

Inverse kinematics using the Inverse Jacobian allows interactive position of kinematic structures

- used for character animation
- posing of character in key-frames

$$\Delta\vec{\phi} = J^{-1}(\vec{\phi})\Delta x_e$$

Use an iterative solution

$$\vec{\phi}_{new} = \vec{\phi}_{curr} + \Delta\vec{\phi} = \vec{\phi}_{curr} + J^{-1}(\vec{\phi}_{curr})\Delta x$$

This solution converges to an approximation of the required end effector position

- error depends on step-size

$$\varepsilon = \left\| J(\vec{\phi})\Delta\theta - \Delta x_e \right\|$$

Solution requires a psuedo-inverse of the Jacobian

Problems: - Multiple Solution
- Singularities
- Ill contitioning

Problems in Inverse Kinematic Solution

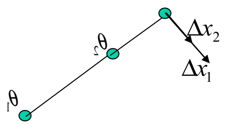
(1) Multiple Solutions

The iterative solution relies on a local linear approximation of the forward kinematic function *f* and only converges to a local minima via 'gradient descent'

- the solution obtained is the nearest local minima
- arbitrary may violate physical constraints

(2) Singularities in the Inverse Jacobian

- Rank of matrix J is the number of independent columns of the matrix
- During iteration rank may change to <n ie 2 columns are linearly dependent
  This occurs when axis of the kinematic chain align 'gymbal-lock'
  the angles become linearly dependent
- both angle parameters produce changes in
  end-effector position in exactly the same direction



(3) Ill-conditoning

- In the region close to a singularity the solution may oscillate about the local minima
- add damping to error to limit rate of change in angles

$$\varepsilon = \left\| J(\vec{\phi})\Delta\theta - \Delta x_e \right\|^2 + \lambda \left\| \Delta\theta \right\|^2$$

Summary

Hierarchical data structures
- tree traversal
- recursive function calls
- use matrix stack to combine matricies
- Object-Oriented design

Animation
- Forward Kinematics: position end-effector for given angles
- Inverse Kinematics: compute angles for given end-effector
    - Iterative solution via inverse Jacobian
    - Jacobian computed geometrically for arbitrary chain
        - ‘moving axis formula’
    - Used for interactive character animation