Computational Graphics: Lecture 10

Alberto Paoluzzi

Thu, Mar 31, 2015

Outline: Hierarchical structures

1. Introduction
2. Affine transformations
3. Traversal algorithm
4. LAR-CC implementation
5. Examples
6. 2D robot arm

Introduction

Geometric models

A geometric model is a pair (geometry, topology) in a given coordinate system,

Geometric models

A geometric model is a pair (geometry, topology) in a given coordinate system,

topology is the LAR specification of highest dimensional cells of a cellular decomposition of the model space,

geometry is specified by the coordinates of vertices, the spatial embedding of 0-cells of the cellular decomposition of space.

Geometric models

A geometric model is a pair (geometry, topology) in a given coordinate system,

topology is the LAR specification of highest dimensional cells of a cellular decomposition of the model space,

geometry is specified by the coordinates of vertices, the spatial embedding of 0-cells of the cellular decomposition of space.

- A model is either an instance of the `Model` class, or simply a pair (`vertices`, `cells`), where

Geometric models

A geometric model is a pair (geometry, topology) in a given coordinate system,

topology is the LAR specification of highest dimensional cells of a cellular decomposition of the model space,

geometry is specified by the coordinates of vertices, the spatial embedding of 0-cells of the cellular decomposition of space.

- A model is either an instance of the `Model` class, or simply a pair (`vertices`, `cells`), where
- `vertices` is a two-dimensional array of floats arranged by rows

Geometric models

A geometric model is a pair (geometry, topology) in a given coordinate system,

topology is the LAR specification of highest dimensional cells of a cellular decomposition of the model space,

geometry is specified by the coordinates of vertices, the spatial embedding of 0-cells of the cellular decomposition of space.

- A model is either an instance of the `Model` class, or simply a pair (`vertices`, `cells`), where
- `vertices` is a two-dimensional array of floats arranged by rows
- `cells` is a list of lists of vertex indices

Structures

A structure is the LAR representation of a hierarchical organisation of spaces into substructures, where each part may be specified in a local coordinate system.

A structure is given as an (ordered) list of substructures and transformations of coordinates, that apply to all the substructures following in the same list.

Structures

A structure is the LAR representation of a hierarchical organisation of spaces into substructures, where each part may be specified in a local coordinate system.

A structure is given as an (ordered) list of substructures and transformations of coordinates, that apply to all the substructures following in the same list.

- A structure gives a graph of the scene, since a substructure may be given a name, and referenced within other structures.

Structures

A structure is the LAR representation of a hierarchical organisation of spaces into substructures, where each part may be specified in a local coordinate system.

A structure is given as an (ordered) list of substructures and transformations of coordinates, that apply to all the substructures following in the same list.

- A structure gives a graph of the scene, since a substructure may be given a name, and referenced within other structures.
- The structure network, including references, can be seen as an acyclic directed multigraph

Structures

A structure is the LAR representation of a hierarchical organisation of spaces into substructures, where each part may be specified in a local coordinate system.

A structure is given as an (ordered) list of substructures and transformations of coordinates, that apply to all the substructures following in the same list.

- A structure gives a graph of the scene, since a substructure may be given a name, and referenced within other structures.
- The structure network, including references, can be seen as an acyclic directed multigraph
- Struct class, whose parameter is a list of either other structures, or models, or transformations of coordinates, or references to structures or models.

Assemblies

An assembly is an (unordered) list of models all embedded in the same coordinate space, i.e. all using the same coordinate system (the world coordinate system, WCS)

- An assembly may be either defined by the user as a list of models, or automatically generated by the traversal of a structure network.

Assemblies

An assembly is an (unordered) list of models all embedded in the same coordinate space, i.e. all using the same coordinate system (the world coordinate system, WCS)

- An assembly may be either defined by the user as a list of models, or automatically generated by the traversal of a structure network.
- At traversal time, all the structures and models are transformed from local coordinate systems to the world coordinates, that correspond to the coordinate frame of the root of the traversed network.

Assemblies

An assembly is an (unordered) list of models all embedded in the same coordinate space, i.e. all using the same coordinate system (the world coordinate system, WCS)

- An assembly may be either defined by the user as a list of models, or automatically generated by the traversal of a structure network.
- At traversal time, all the structures and models are transformed from local coordinate systems to the world coordinates, that correspond to the coordinate frame of the root of the traversed network.
- An assembly is the linearised version of the traversed structure network, where all the models are using the world coordinate system.

Affine transformations

Design decision

- assume the scipy `ndarray` as the type of vertices, stored in row-major order;

Design decision

- assume the scipy `ndarray` as the type of vertices, stored in row-major order;
- use the last coordinate as the homogeneous coordinate of vertices, but do not store it explicitly;

Design decision

- assume the scipy `ndarray` as the type of vertices, stored in row-major order;
- use the last coordinate as the homogeneous coordinate of vertices, but do not store it explicitly;
- store explicitly the homogeneous coordinate of transformation matrices.

Design decision

- assume the scipy `ndarray` as the type of vertices, stored in row-major order;
- use the last coordinate as the homogeneous coordinate of vertices, but do not store it explicitly;
- store explicitly the homogeneous coordinate of transformation matrices.
- use labels '`verts`' and '`mat`' to distinguish between vertices and transformation matrices.

Design decision

- assume the scipy `ndarray` as the type of vertices, stored in row-major order;
- use the last coordinate as the homogeneous coordinate of vertices, but do not store it explicitly;
- store explicitly the homogeneous coordinate of transformation matrices.
- use labels '`verts`' and '`mat`' to distinguish between vertices and transformation matrices.
- transformation matrices are dimension-independent, and their dimension is computed as the length of the parameter vector passed to the generating function.

Elementary transformations: Translation matrices

```
def t(*args):
    d = len(args)
    mat = scipy.identity(d+1)
    for k in range(d):
        mat[k,d] = args[k]
    return mat.view(Mat)
```

Elementary transformations: Scaling matrices

```
def s(*args):
    d = len(args)
    mat = scipy.identity(d+1)
    for k in range(d):
        mat[k,k] = args[k]
    return mat.view(Mat)
```

Elementary transformations: Rotation matrices

```
def r(*args):
    args = list(args)
    n = len(args)
    @< plane rotation (in 2D) @>
    @< space rotation (in 3D) @>
    return mat.view(Mat)
```

plane rotation (in 2D)

```
if n == 1: # rotation in 2D
    angle = args[0]; cos = COS(angle); sin = SIN(angle)
    mat = scipy.identity(3)
    mat[0,0] = cos;    mat[0,1] = -sin;
    mat[1,0] = sin;    mat[1,1] = cos;
```

Elementary transformations: rotation matrices

space rotation (in 3D)

```
if n == 3: # rotation in 3D
    mat = scipy.identity(4)
    angle = VECTNORM(args); axis = UNITVECT(args)
    cos = COS(angle); sin = SIN(angle)
    @< elementary rotations (in 3D) @>
    @< general rotations (in 3D) @>
```

elementary rotations (in 3D)

```
if axis[1]==axis[2]==0.0:     # rotation about x
    mat[1,1] = cos;     mat[1,2] = -sin;
    mat[2,1] = sin;     mat[2,2] = cos;
elif axis[0]==axis[2]==0.0:     # rotation about y
    mat[0,0] = cos;     mat[0,2] = sin;
    mat[2,0] = -sin;     mat[2,2] = cos;
elif axis[0]==axis[1]==0.0:     # rotation about z
    mat[0,0] = cos;     mat[0,1] = -sin;
    mat[1,0] = sin;     mat[1,1] = cos;
```

Elementary transformations: rotation matrices

general rotations (in 3D)

```
else:  # general 3D rotation (Rodrigues' rotation formula)
    I = scipy.identity(3)
    u = axis

    Ux = scipy.array([
        [0,          -u[2],        u[1]],
        [u[2],          0,       -u[0]],
        [-u[1],     u[0],           0]])
    UU = scipy.array([
        [u[0]*u[0],    u[0]*u[1],    u[0]*u[2]],
        [u[1]*u[0],    u[1]*u[1],    u[1]*u[2]],
        [u[2]*u[0],    u[2]*u[1],    u[2]*u[2]]])
    mat[:3,:3] = cos*I + sin*Ux + (1.0-cos)*UU
```

Hierarchical complexes

Hierarchical models of assemblies are generated by an aggregation of subassemblies

each one defined in a local coordinate system, and relocated by affine transformations of coordinates

- each elementary part and each assembly, at every hierarchical level, are defined independently from each other, using a local coordinate frame, suitably chosen to make its definition easier

Hierarchical complexes

Hierarchical models of assemblies are generated by an aggregation of subassemblies

each one defined in a local coordinate system, and relocated by affine transformations of coordinates

- each elementary part and each assembly, at every hierarchical level, are defined independently from each other, using a local coordinate frame, suitably chosen to make its definition easier
- only one copy of each component is stored in the memory, and may be instanced in different locations and orientations how many times it is needed.

Traversal algorithm

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)
- strings (think of transformation matrices)

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)
- strings (think of transformation matrices)

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)
- strings (think of transformation matrices)

any structure list may contain:

- any combination of numbers, strings, and structure lists (either explicit, or via python references to structure lists, i.e. through names of structure variables)

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)
- strings (think of transformation matrices)

any structure list may contain:

- any combination of numbers, strings, and structure lists (either explicit, or via python references to structure lists, i.e. through names of structure variables)

Emulation of scene multigraph traversal

use two types of nodes:

- numbers (think of vertices)
- strings (think of transformation matrices)

any structure list may contain:

- any combination of numbers, strings, and structure lists (either explicit, or via python references to structure lists, i.e. through names of structure variables)

Design goal

All components of any structure are (recursively) transformed to the coordinate frame of the first element of the structure

Emulation of scene multigraph traversal

```
from pyplasm import *
def __traverse(CTM, stack, o):
    for i in range(len(o)):
        if ISNUM(o[i]): print CTM, o[i]
        elif ISSTRING(o[i]):
            CTM.append(o[i])
        elif ISSEQ(o[i]):
            stack.append(o[i])               # push the stack
            __traverse(CTM, stack, o[i])
            CTM = CTM[:-len(stack)]          # pop the stack

def algorithm(data):
    CTM,stack = ["I"],[]
    __traverse(CTM, stack, data)
```

Examples of multigraph traversal

```
data = [1,"A", 2, 3, "B", [4, "C", 5], [6,"D", "E", 7, 8], 9]
print algorithm(data)
>>> ['I'] 1
    ['I', 'A'] 2
    ['I', 'A'] 3
    ['I', 'A', 'B'] 4
    ['I', 'A', 'B', 'C'] 5
    ['I', 'A', 'B'] 6
    ['I', 'A', 'B', 'D', 'E'] 7
    ['I', 'A', 'B', 'D', 'E'] 8
    ['I', 'A', 'B'] 9

data = [1,"A", [2, 3, "B", 4, "C", 5, 6,"D"], "E", 7, 8, 9]
print algorithm(data)
>>> ['I'] 1
    ['I', 'A'] 2
    ['I', 'A'] 3
    ['I', 'A', 'B'] 4
    ['I', 'A', 'B', 'C'] 5
    ['I', 'A', 'B', 'C'] 6
    ['I', 'A', 'B', 'C', 'E'] 7
    ['I', 'A', 'B', 'C', 'E'] 8
    ['I', 'A', 'B', 'C', 'E'] 9
```

Examples of multigraph traversal

```
dat = [2, 3, "B", 4, "C", 5, 6,"D"]
print algorithm(dat)
>>> ['I'] 2
    ['I'] 3
    ['I', 'B'] 4
    ['I', 'B', 'C'] 5
    ['I', 'B', 'C'] 6

data = [1,"A", dat, "E", 7, 8, 9]
print algorithm(data)
>>> ['I'] 1
    ['I', 'A'] 2
    ['I', 'A'] 3
    ['I', 'A', 'B'] 4
    ['I', 'A', 'B', 'C'] 5
    ['I', 'A', 'B', 'C'] 6
    ['I', 'A', 'B', 'C', 'E'] 7
    ['I', 'A', 'B', 'C', 'E'] 8
    ['I', 'A', 'B', 'C', 'E'] 9
```

Algorithm: geometric structure traversal

Script 8.3.1 (Traversal of a multigraph)

algorithm TRAVERSAL $((N, A, f) : multigraph)$ {
    CTM := identity matrix;
    TraverseNode ($root$)
}

proc TRAVERSENODE ($n : node$) {
    foreach $a \in A$ outgoing from n **do** TraverseArc (a);
    ProcessNode (n)
}

proc TRAVERSEARC ($a = (n, m) : arc$) {
    Stack.push (CTM);
    CTM := CTM * a.mat;
    TraverseNode (m);
    CTM := Stack.pop()
}

proc PROCESSNODE ($n : node$) {
    foreach object $\in n$ **do** Process(CTM * object)
}

LAR-CC implementation

Algorithm: geometric structure traversal

decides between different cases, depending on the type of the current object

```
def traversal(CTM, stack, obj, scene=[]):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
            scene += [larApply(CTM)(obj[i])]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2 or len(obj[i])==3):
            scene += [larApply(CTM)(obj[i])]
        elif isinstance(obj[i],Mat):
            CTM = scipy.dot(CTM, obj[i])
        elif isinstance(obj[i],Struct):
            stack.append(CTM)
            traversal(CTM, stack, obj[i], scene)
            CTM = stack.pop()
    return scene
```

- If the object is a `Model` instance, then applies to it the `CTM` matrix;

Algorithm: geometric structure traversal

decides between different cases, depending on the type of the current object

```
def traversal(CTM, stack, obj, scene=[]):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
            scene += [larApply(CTM)(obj[i])]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2 or len(obj[i])==3):
            scene += [larApply(CTM)(obj[i])]
        elif isinstance(obj[i],Mat):
            CTM = scipy.dot(CTM, obj[i])
        elif isinstance(obj[i],Struct):
            stack.append(CTM)
            traversal(CTM, stack, obj[i], scene)
            CTM = stack.pop()
    return scene
```

- If the object is a `Model` instance, then applies to it the `CTM` matrix;
- else if the object is a `Mat` instance, then the `CTM` matrix is updated by (right) product with it;

Algorithm: geometric structure traversal

decides between different cases, depending on the type of the current object

```
def traversal(CTM, stack, obj, scene=[]):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
            scene += [larApply(CTM)(obj[i])]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2 or len(obj[i])==3):
            scene += [larApply(CTM)(obj[i])]
        elif isinstance(obj[i],Mat):
            CTM = scipy.dot(CTM, obj[i])
        elif isinstance(obj[i],Struct):
            stack.append(CTM)
            traversal(CTM, stack, obj[i], scene)
            CTM = stack.pop()
    return scene
```

- If the object is a `Model` instance, then applies to it the `CTM` matrix;
- else if the object is a `Mat` instance, then the `CTM` matrix is updated by (right) product with it;
- else if the object is a `Struct` instance, then the `CTM` is pushed on the stack, initially empty,

Algorithm: geometric structure traversal

decides between different cases, depending on the type of the current object

```
def traversal(CTM, stack, obj, scene=[]):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
            scene += [larApply(CTM)(obj[i])]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2 or len(obj[i])==3):
            scene += [larApply(CTM)(obj[i])]
        elif isinstance(obj[i],Mat):
            CTM = scipy.dot(CTM, obj[i])
        elif isinstance(obj[i],Struct):
            stack.append(CTM)
            traversal(CTM, stack, obj[i], scene)
            CTM = stack.pop()
    return scene
```

- If the object is a `Model` instance, then applies to it the `CTM` matrix;
- else if the object is a `Mat` instance, then the `CTM` matrix is updated by (right) product with it;
- else if the object is a `Struct` instance, then the `CTM` is pushed on the stack, initially empty,
- then the `traversal` is called (recursion),

Algorithm: geometric structure traversal

decides between different cases, depending on the type of the current object

```
def traversal(CTM, stack, obj, scene=[]):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
            scene += [larApply(CTM)(obj[i])]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2 or len(obj[i])==3):
            scene += [larApply(CTM)(obj[i])]
        elif isinstance(obj[i],Mat):
            CTM = scipy.dot(CTM, obj[i])
        elif isinstance(obj[i],Struct):
            stack.append(CTM)
            traversal(CTM, stack, obj[i], scene)
            CTM = stack.pop()
    return scene
```

- If the object is a `Model` instance, then applies to it the `CTM` matrix;
- else if the object is a `Mat` instance, then the `CTM` matrix is updated by (right) product with it;
- else if the object is a `Struct` instance, then the `CTM` is pushed on the stack, initially empty,
- then the `traversal` is called (recursion),
- and finally, at (each) return from recursion, the `CTM` is recovered by popping the stack.

Examples

We start with a simple 2D example of a non-nested list of translated 2D object instances and rotation about the origin.

```
""" Example of non-nested structure with translation and rotations """
import sys; sys.path.insert(0, 'lib/py/')
from larlib import *

square = larCuboids([1,1])
table = larApply( t(-.5,-.5) )(square)
chair = larApply( s(.35,.35) )(table)
chair1 = larApply( t(.75, 0) )(chair)
chair2 = larApply( r(PI/2) )(chair1)
chair3 = larApply( r(PI/2) )(chair2)
chair4 = larApply( r(PI/2) )(chair3)
scene = Struct([table,chair1,chair2,chair3,chair4])
VIEW(SKEL_1(STRUCT(MKPOLS(struct2lar(scene)))))
```

Example: Table and chairs

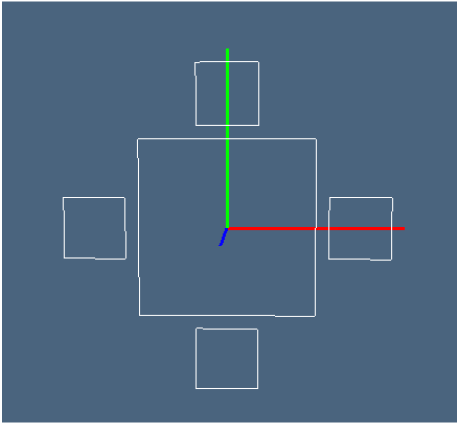

Figure 2:Table and chairs: non-nested list

A different composition of transformations, from local to global coordinate frames, is used in the following example.

```
""" Example of non-nested structure with translation and rotations """
import sys; sys.path.insert(0, 'lib/py/')
from larlib import *

square = larCuboids([1,1])
table = larApply( t(-.5,-.5) )(square)
chair = larApply( s(.35,.35) )(table)
chair = larApply( t(.75, 0) )(chair)
struct = Struct([table] + 4*[chair, r(PI/2)])
scene = evalStruct(struct)
VIEW(SKEL_1(STRUCT(CAT(AA(MKPOLS)(scene)))))
```

Finally, a similar 2D example is given, by nesting one (or more) structures via separate definition and call by reference from the interior.

```
""" Example of nested structures with translation and rotations """
import sys; sys.path.insert(0, 'lib/py/')
from larlib import *

square = larCuboids([1,1])
table = larApply( t(-.5,-.5) )(square)
chair = Struct([ t(.75, 0), s(.35,.35), table ])
struct = Struct( [t(2,1)] + [table] + 4*[r(PI/2), chair])
struct = Struct(10*[struct,t(0,2.5)])
scene = Struct(10*[struct,t(3,0)])
VIEW(SKEL_1(STRUCT(MKPOLS(struct2lar(scene)))))
```

Example: Table and chairs

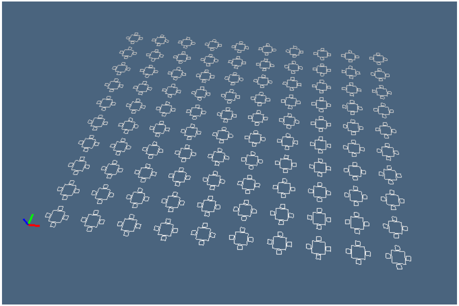

Figure 3:Table and chairs: nesting one (or more) structures

2D robot arm

Example: 2D robot arm (lar-cc package)

```
from larlib import *

link = Struct([t(-1,-19),s(2,20),larCuboids([1,1])])
def joint(a): return [t(0,-18),r(a*PI/180)]
def arm(a1,a2,a3):
    return Struct([s(.1,.1)] + [link] + joint(a1) + [link] + joint(a2)
                        + [link] + joint(a3) + [link])

hpcs = MKPOLS(struct2lar(arm(30,60,90)))
VIEW(STRUCT(hpcs))
```

Example: 2D robot arm (`lar-cc` package)

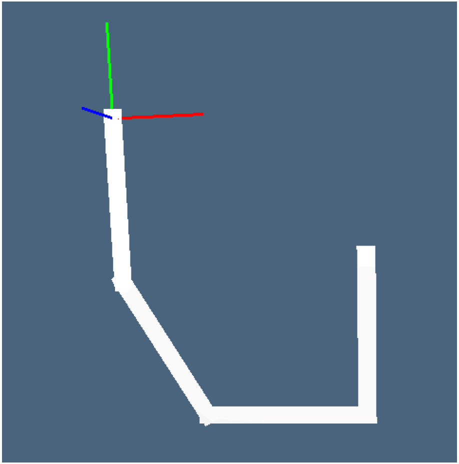

Figure 4:2D robot arm (`lar-cc` package)

Example: 2D robot arm (pyplasm package)

```
from pyplasm import *

link = T([1,2])([-1,-19])(CUBOID([2,20]))

def joint(a):
    return COMP([T(2)(-18), R([1,2])(a*PI/180)])

def arm(a1,a2,a3):
    return STRUCT([ S([1,2])([.1,.1]), link, joint(a1), COLOR(RED)(link),
                joint(a2), COLOR(GREEN)(link),
                joint(a3), COLOR(BLUE)(link) ])

VIEW(arm(30,60,90))
```

Example: 2D robot arm (`pyplasm` package)

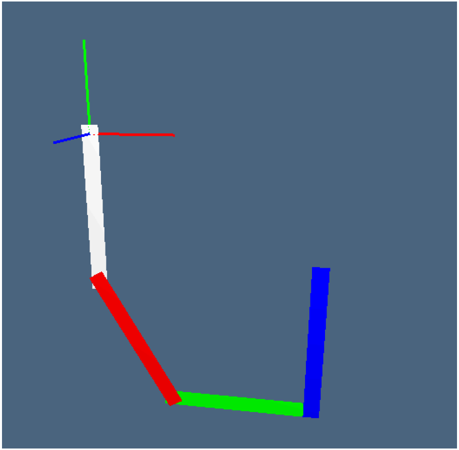

Figure 5:2D robot arm (`pyplasm` package)