TIEA311
Tietokonegrafiikan perusteet
kevät 2019

("Principles of Computer Graphics" – Spring 2019)

TIEA311 Tietokonegrafiikan perusteet – kevät 2019 (“Principles of Computer Graphics” – Spring 2019)

Adapted from: *Wojciech Matusik*, and *Frédo Durand*: 6.837 Computer Graphics. Fall 2012. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu/.

Frontpage of the local course version, held during Spring 2019 at the Faculty of Information technology, University of Jyväskylä:
`http://users.jyu.fi/~nieminen/tgp19/`

TIEA311 - Today in Jyväskylä

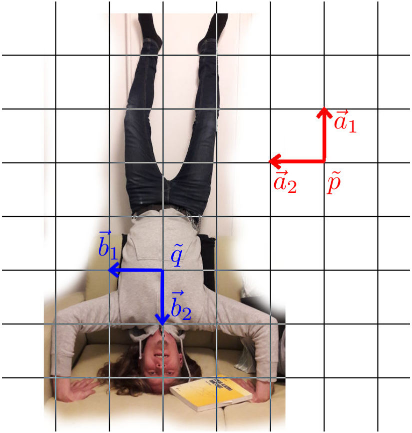
Plan for today:

- Usual warm-up.
- Continue from yesterday
- Go through theory
- Remember to have a break!
- The teacher will try to remember and make use of the fact that we have groups of 2-3 students with pen and paper.

TIEA311 - Local plan for today

- Maybe some things I forgot to mention yesterday?
- Very brief recap of what went on previously.
- Then forward, with full speed!

TIEA311



"Midterm" revisited

Live exercise time!

$\rightarrow$ See lecture video.

This is important.

Do this!

(non-Finnish ones need to cope with English slides from MIT that will summarize this later; make sure you go through the frame switches using pen and paper, not only looking at them!)

Linear algebra is friendly and simple **after the initial pain** of learning it. (this slide is transcripted from MIT OCW originals; I think the matrices got inversed A vs A^{-1} w.r.t. our lecture example in Finnish. But that is the point: we learn how to re-learn this any time we need to!)

Some transformation is specified by a matrix S in "car" frame $\vec{\mathbf{f}}$ as $\vec{\mathbf{f}}^t\mathbf{c} \rightarrow \vec{\mathbf{f}}^t S\mathbf{c}$.
How is the world frame $\vec{\mathbf{a}}$ affected by this?

- Frame can be interchanged with matrix and inverse: $\vec{\mathbf{f}}^t = \vec{\mathbf{a}}^t A^{-1}$ and $\vec{\mathbf{a}}^t = \vec{\mathbf{f}}^t A$.
- Coordinates transform too: $\vec{\mathbf{a}}^t\mathbf{d} = (\vec{\mathbf{f}}^t A)\mathbf{d} = \vec{\mathbf{f}}^t(A\mathbf{d})$ and $\vec{\mathbf{f}}^t\mathbf{c} = (\vec{\mathbf{a}}^t A^{-1})\mathbf{c} = \vec{\mathbf{a}}^t(A^{-1}\mathbf{c})$.
- So, start from transformation given in $\vec{\mathbf{f}}$:
  $\vec{\mathbf{f}}^t\mathbf{c} \rightarrow \vec{\mathbf{f}}^t S\mathbf{c}$
- Plug in the above expressions. Transformation then reads: $(\vec{\mathbf{a}}^t A^{-1})(A\mathbf{d}) \rightarrow (\vec{\mathbf{a}}^t A^{-1})S(A\mathbf{d})$
- Rearrange parentheses: $\vec{\mathbf{a}}^t(A^{-1}A)\mathbf{d} \rightarrow \vec{\mathbf{a}}^t(A^{-1}SA)\mathbf{d}$
- Rid of identity matrix: $\vec{\mathbf{a}}^t\mathbf{d} \rightarrow \vec{\mathbf{a}}^t(A^{-1}SA)\mathbf{d}$. Done!

Linear algebra is friendly and simple **after the initial pain** of learning it. (this slide **uses the notations we created together** during the Finnish lecture example! And this, if anything, proves the main point: we **have learned how to re-learn** and **verify** this any time we need to!)

Some transformation is specified by a matrix R in "dude" frame $\vec{\mathbf{b}}$ as $\vec{\mathbf{b}}^t\mathbf{c} \rightarrow \vec{\mathbf{b}}^t R\mathbf{c}$.
How is the world frame $\vec{\mathbf{a}}$ affected by this?

- Frame can be interchanged with matrix and inverse: $\vec{\mathbf{b}}^t = \vec{\mathbf{a}}^t A$ and $\vec{\mathbf{a}}^t = \vec{\mathbf{b}}^t A^{-1}$.
- Coordinates transform too: $\vec{\mathbf{a}}^t\mathbf{d} = (\vec{\mathbf{b}}^t A^{-1})\mathbf{d} = \vec{\mathbf{b}}^t(A^{-1}\mathbf{d})$ and $\vec{\mathbf{f}}^t\mathbf{c} = (\vec{\mathbf{a}}^t A)\mathbf{c} = \vec{\mathbf{a}}^t(A\mathbf{c})$.
- So, start from transformation given in $\vec{\mathbf{f}}$:
  $\vec{\mathbf{f}}^t\mathbf{c} \rightarrow \vec{\mathbf{f}}^t R\mathbf{c}$
- Plug in the above expressions. Transformation then reads: $(\vec{\mathbf{a}}^t A)(A^{-1}\mathbf{d}) \rightarrow (\vec{\mathbf{a}}^t A)R(A^{-1}\mathbf{d})$
- Rearrange parentheses: $\vec{\mathbf{a}}^t(AA^{-1})\mathbf{d} \rightarrow \vec{\mathbf{a}}^t(ARA^{-1})\mathbf{d}$
- Rid of identity matrix: $\vec{\mathbf{a}}^t\mathbf{d} \rightarrow \vec{\mathbf{a}}^t(ARA^{-1})\mathbf{d}$. Done!

Those who saw the lecture 13 of TIEA311 Spring 2019 either live or on video witnessed the following:

- Insecure teacher, in panic, trying to figure out if he got it right this time (after two consecutive years of failing the first attempt at explaining this bit) or not.
- The **effect** of panic and extreme **deadline pressure** on somebody who thinks he can do this thing any time, and (seemingly) can, too:). Circumstances matter.
- In the end, there was ultimately no mistake, but uncertainty was acknowledged. And **that** is **the main ingredient**, folks!!

Learnings to take home:

- This stuff is easy. . . but only **after getting it right** and being **sure** it was right.
- It is **necessary to doubt everything**, starting especially from yourself. The same in all math **and programming**!
- Finally: math (and software!) does not lie. It works or it doesn't, and there is a reason. It **can be verified/tested**.

Further necessary exercises:

Compute the thing with actual matrices, using the power tools of pen and paper, and verify it works for a simple transform, like the 90 degree rotation we did together on Finnish lectures.

Celebrate the “magic” of mathematics that **you can now perform**: the **algebraic equation** we sort of **found** is valid for **any** affine transform and **any** two frames!

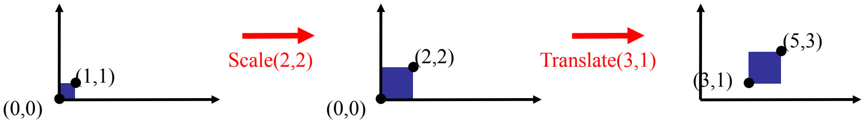
Think about how you can follow either the transformations of the frame (multiply frame from right), or transformations of the coordinates (multiply coordinates from left) one-by-one and end up with the **same result**. Real-world objects may help the brain.

Observe that after computing either way, **there is finally only one** matrix $M = ARA^{-1}$ that performs the whole transform.

This is the same for **any number of combined transforms**!

How are transforms combined?

Scale then Translate



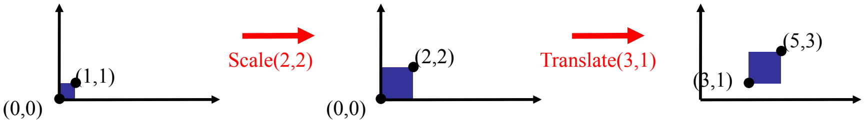
Use matrix multiplication: p' = T (S p) = TS p

$$TS = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

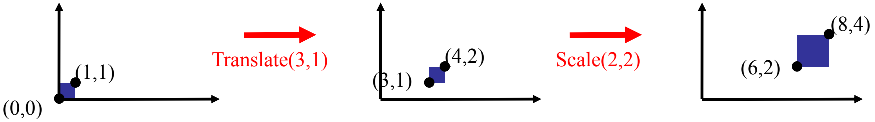
Caution: matrix multiplication is NOT commutative!

Non-commutative Composition

Scale then Translate: $p' = T (S p) = TS p$



Translate then Scale: $p' = S (T p) = ST p$



Non-commutative Composition

Scale then Translate: p' = T (S p) = TS p

$$TS = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Translate then Scale: p' = S (T p) = ST p

$$ST = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 6 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

6.837 Computer Graphics
Hierarchical Modeling

Wojciech Matusik, MIT EECS

Some slides from BarbCutler &
Jaakko Lehtinen

1

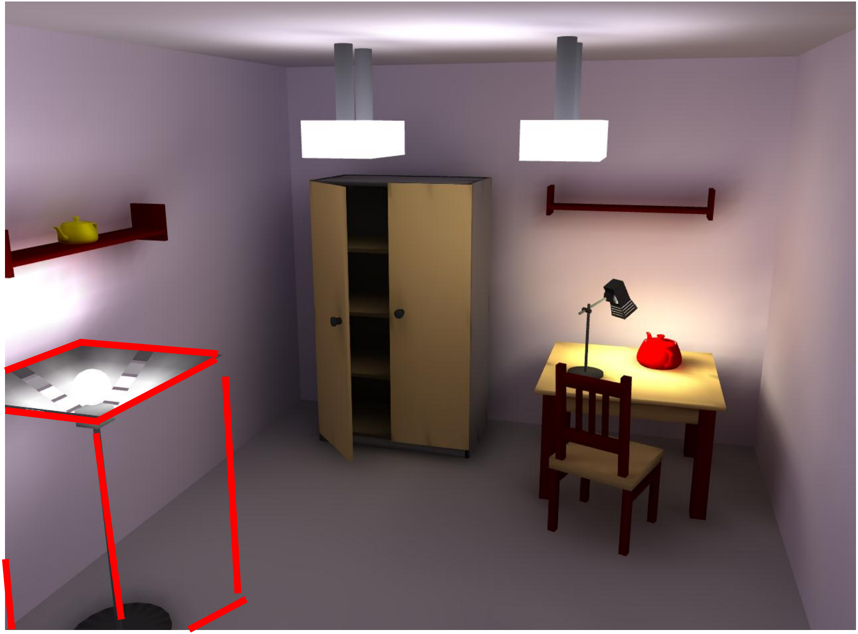
Hierarchical Modeling

- Triangles, parametric curves and surfaces are the building blocks from which more complex real-world objects are modeled.

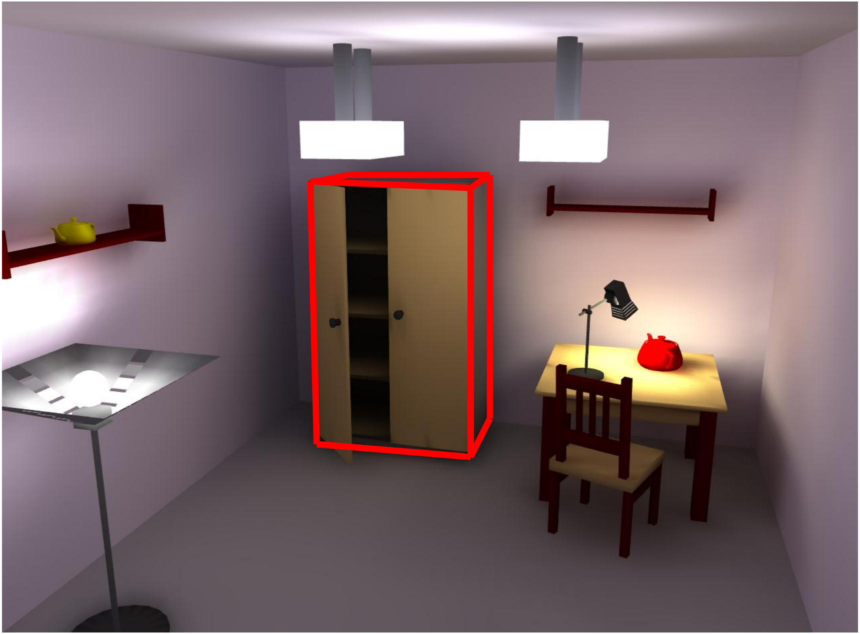
- Hierarchical modeling creates complex real-world objects by combining simple primitive shapes into more complex aggregate objects.

Hierarchical models

Hierarchical models



Hierarchical models



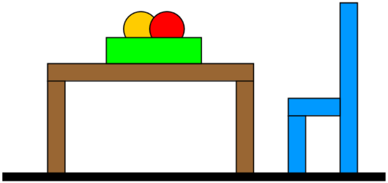
Hierarchical models

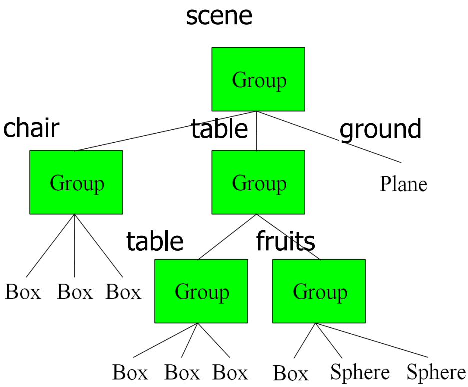
Hierarchical models

Hierarchical models

Hierarchical Grouping of Objects

- The “scene graph” represents the logical organization of scene

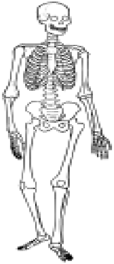




Scene Graph

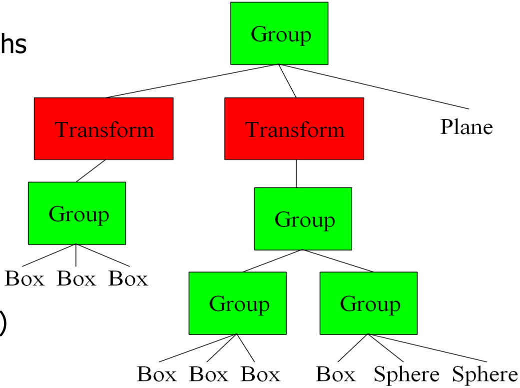
Image courtesy of David Bařina, Kamil Dudka, Jakub Filák, Lukáš Hefka on Wikimedia Commons.

- Convenient Data structure for scene representation
  - Geometry (meshes, etc.)
  - Transformations
  - Materials, color
  - Multiple instances
- Basic idea: Hierarchical Tree
- Useful for manipulation/animation
  - Also for articulated figures
- Useful for rendering, too
  - Ray tracing acceleration, occlusion culling
  - But note that two things that are close to each other in the tree are NOT necessarily spatially near each other



Scene Graph Representation

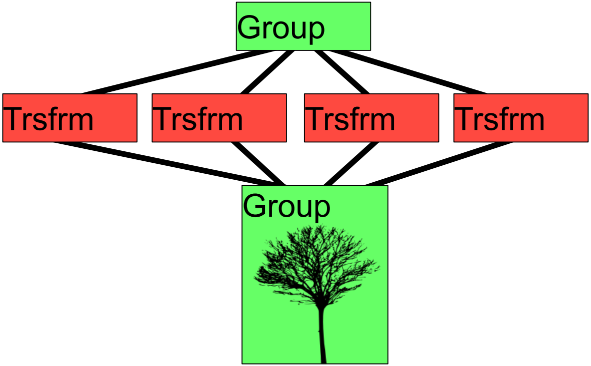
- Basic idea: Tree
- Comprised of several node types
  - Shape: 3D geometric objects
  - Transform: Affect current transformation
  - Property: Color, texture
  - Group: Collection of subgraphs



- C++ implementation
  - base class Object
    - children, parent
  - derived classes for each node type (group, transform)

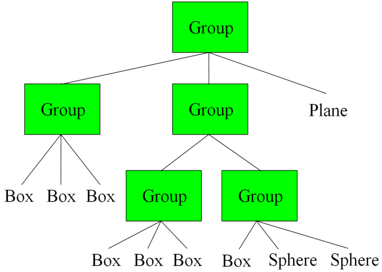
Scene Graph Representation

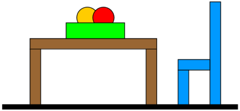
- In fact, generalization of a tree: Directed Acyclic Graph (DAG)
  - Means a node can have multiple parents, but cycles are not allowed
- Why? Allows multiple instantiations
  - Reuse complex hierarchies many times in the scene using different transformations (example: a tree)
    - Of course, if you only want to reuse meshes, just load the mesh once and make several geometry nodes point to the same data



Simple Example with Groups

```
Group {
     numObjects 3
     Group {
          numObjects 3
          Box { <BOX PARAMS> }
          Box { <BOX PARAMS> }
          Box { <BOX PARAMS> } }
     Group {
          numObjects 2
          Group {
               Box { <BOX PARAMS> }
               Box { <BOX PARAMS> }
               Box { <BOX PARAMS> } }
          Group {
               Box { <BOX PARAMS> }
               Sphere { <SPHERE PARAMS> }
               Sphere { <SPHERE PARAMS> } } }
     Plane { <PLANE PARAMS> } }
```

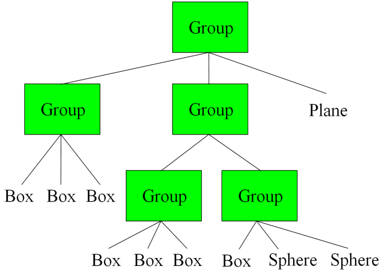


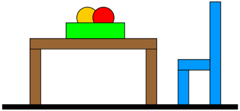


Text format is fictitious, better to use XML in real applications

Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```



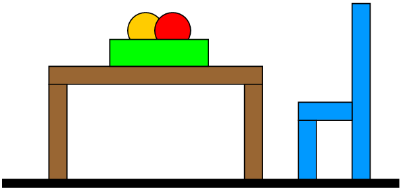


Here we have only simple shapes, but easy to add a “Mesh” node whose parameters specify an .OBJ to load (say)

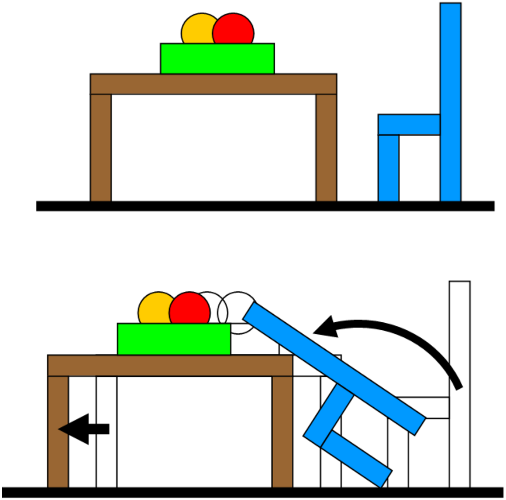
Adding Attributes (Material, etc.)

```
Group {
    numObjects 3
    Material { <BLUE> }
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Material { <BROWN> }
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Material { <GREEN> }
            Box { <BOX PARAMS> }
            Material { <RED> }
            Sphere { <SPHERE PARAMS> }
            Material { <ORANGE> }
            Sphere { <SPHERE PARAMS> } } }

    Plane { <PLANE PARAMS> } }
```



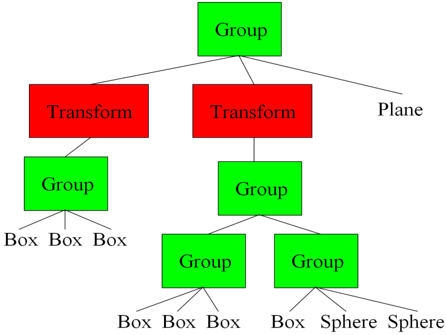
Adding Transformations



Questions?

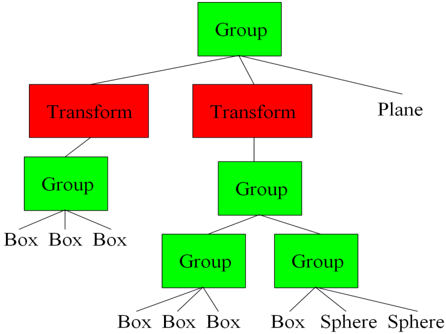
Scene Graph Traversal

- Depth first recursion
  - Visit node, then visit subtrees (top to bottom, left to right)
  - When visiting a geometry node: Draw it!
- How to handle transformations?
  - Remember, transformations are always specified in coordinate system of the parent

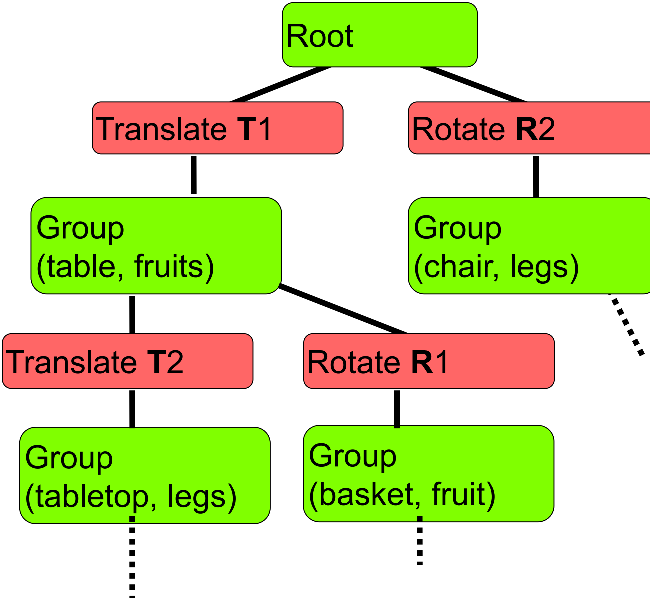


Scene Graph Traversal

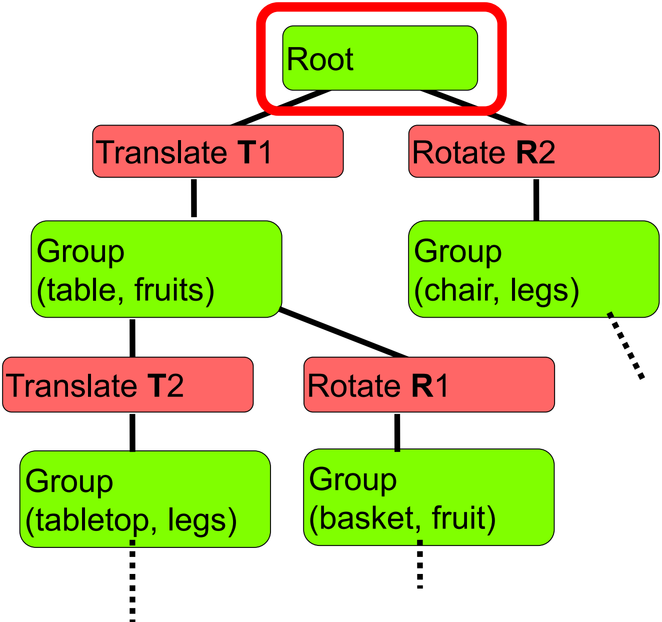
- How to handle transformations?
  - Traversal algorithm keeps a **transformation state S** (a 4x4 matrix)
    - from world coordinates
    - Initialized to identity in the beginning
  - Geometry nodes always drawn using current **S**
  - When visiting a transformation node **T**: multiply current state **S** with **T**, then visit child nodes
    - Has the effect that nodes below will have new transformation
  - When all children have been visited, **undo the effect of T!**



Traversal Example

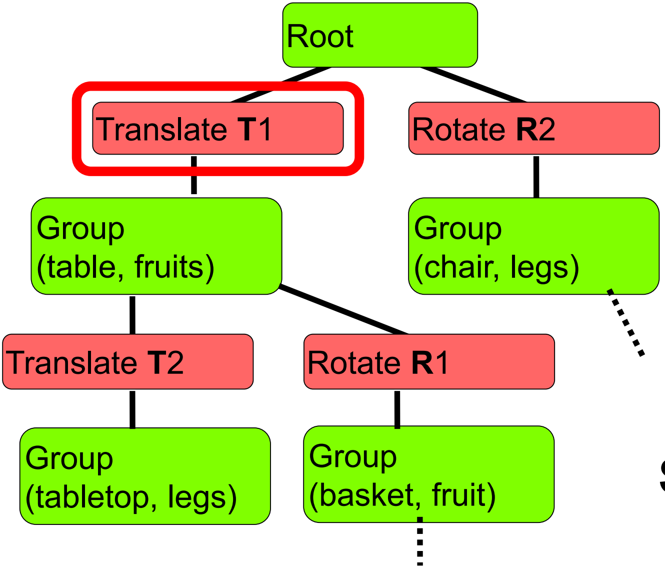


Traversal Example



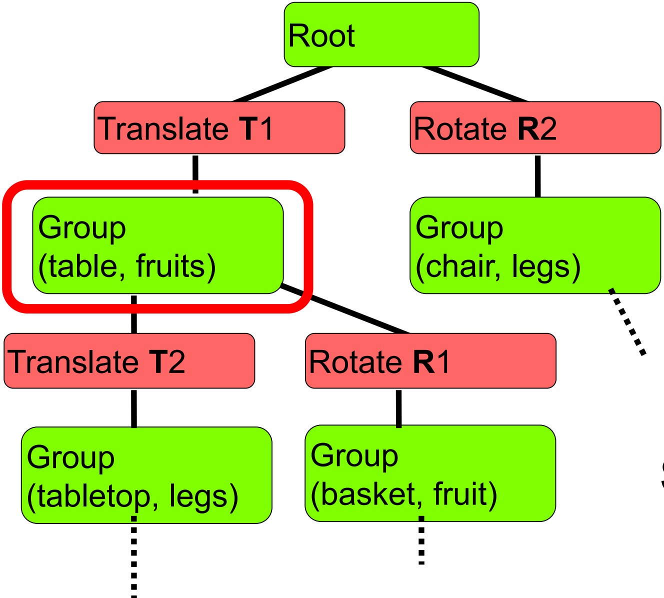
S = **I**

Traversal Example



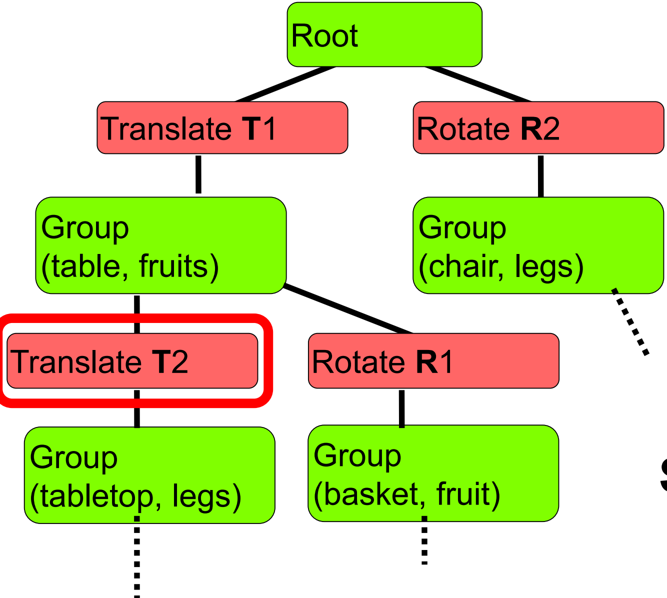
S = T1

Traversal Example



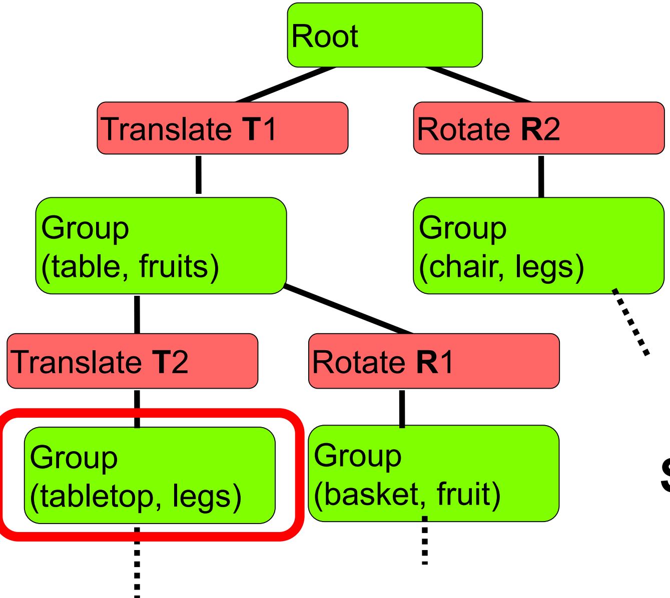
S = T1

Traversal Example



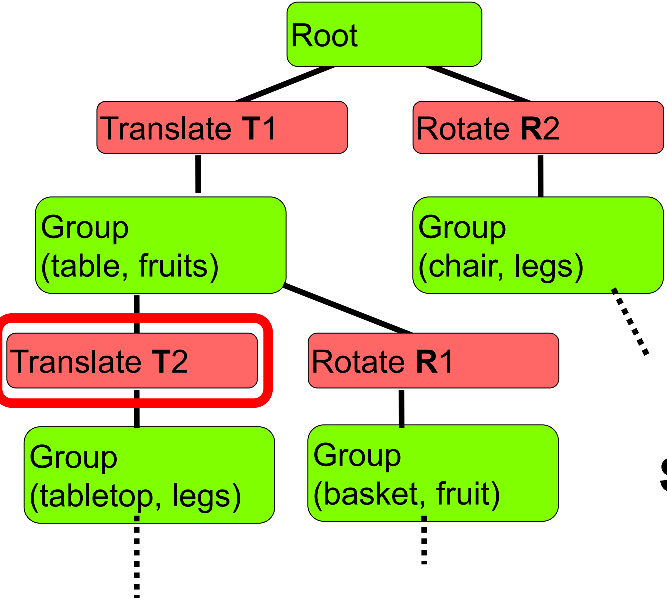
S = T1 T2

Traversal Example



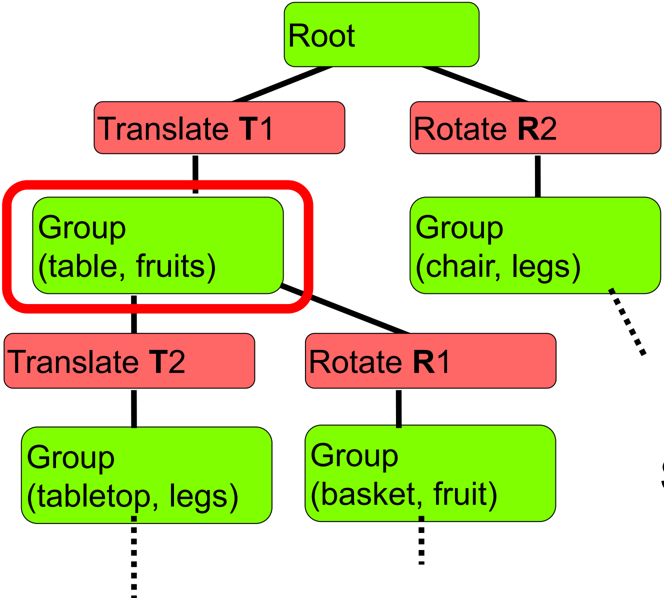
S = T1 T2

Traversal Example



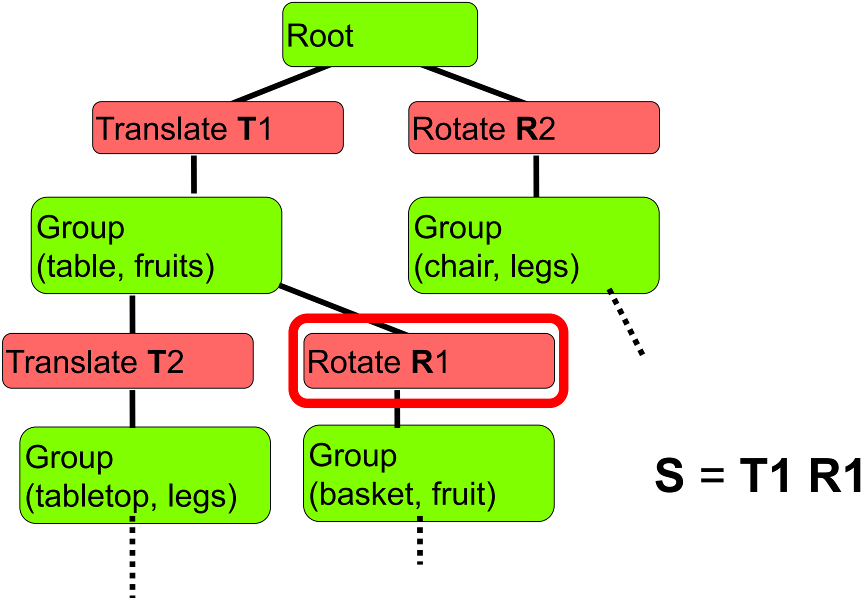
S = T1 T2

Traversal Example

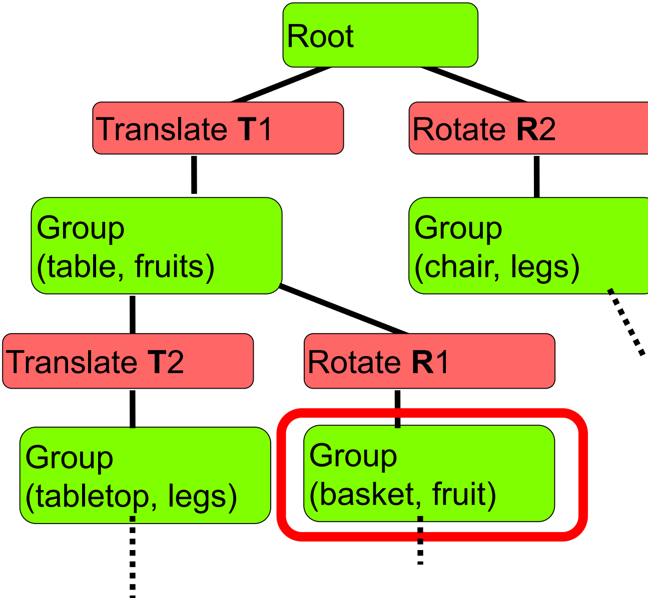


S = T1

Traversal Example

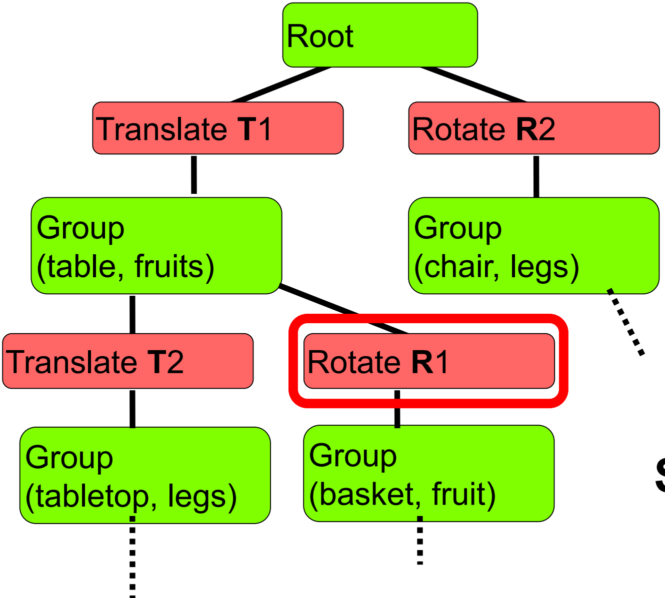


Traversal Example



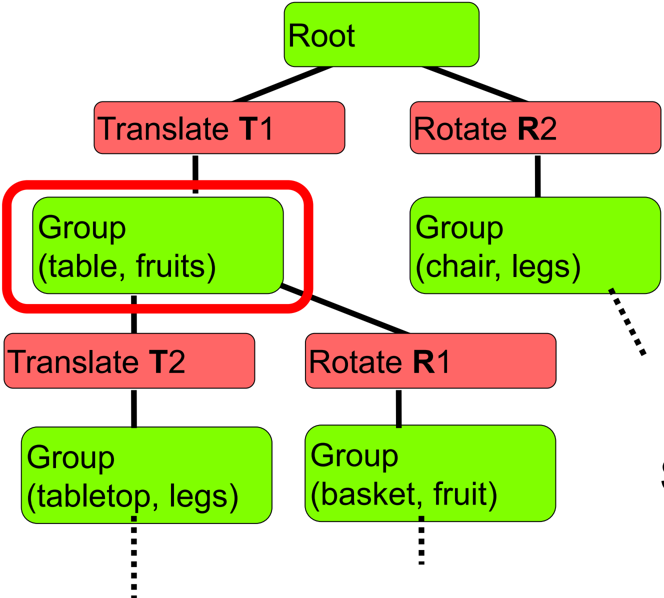
S = T1 R1

Traversal Example



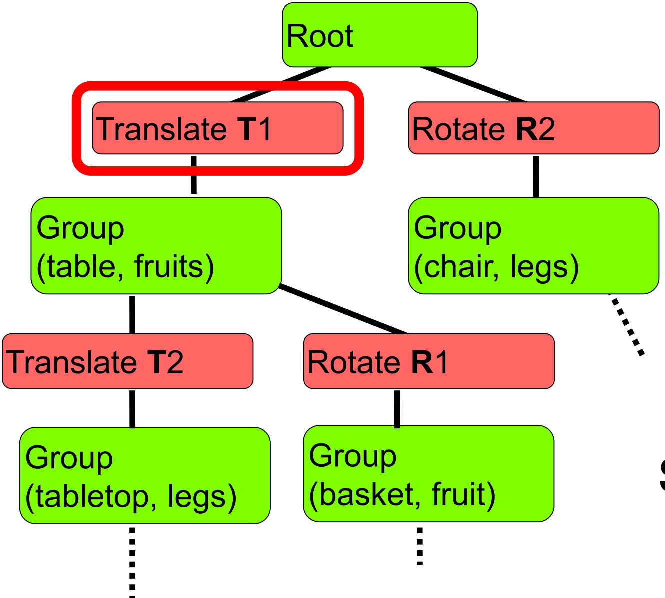
S = T1 R1

Traversal Example



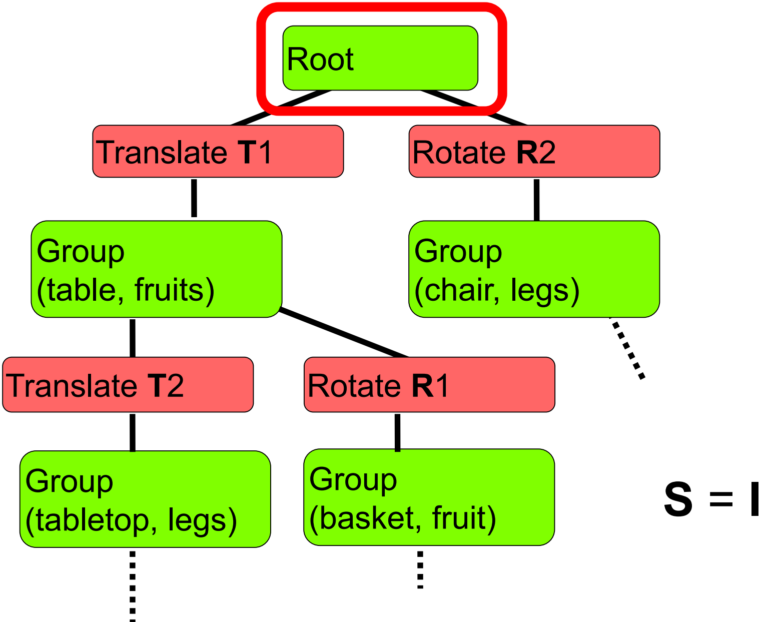
S = T1

Traversal Example

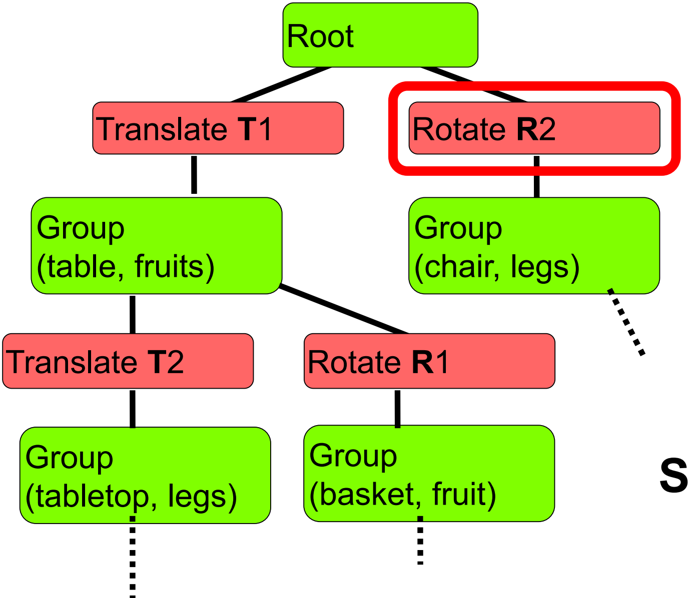


S = T1

Traversal Example

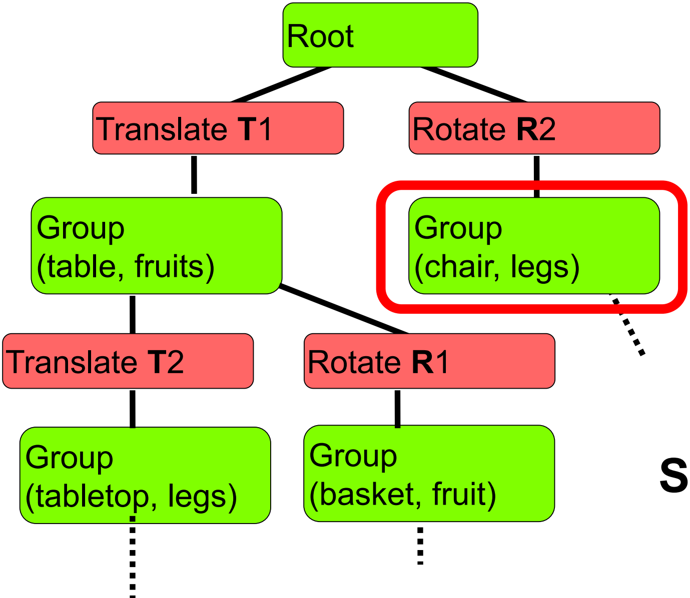


Traversal Example



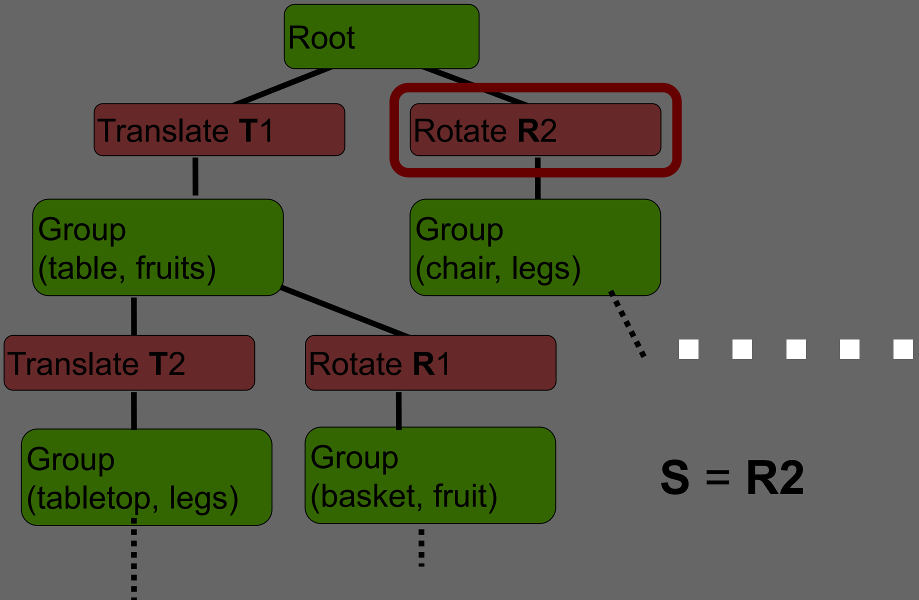
S = R2

Traversal Example



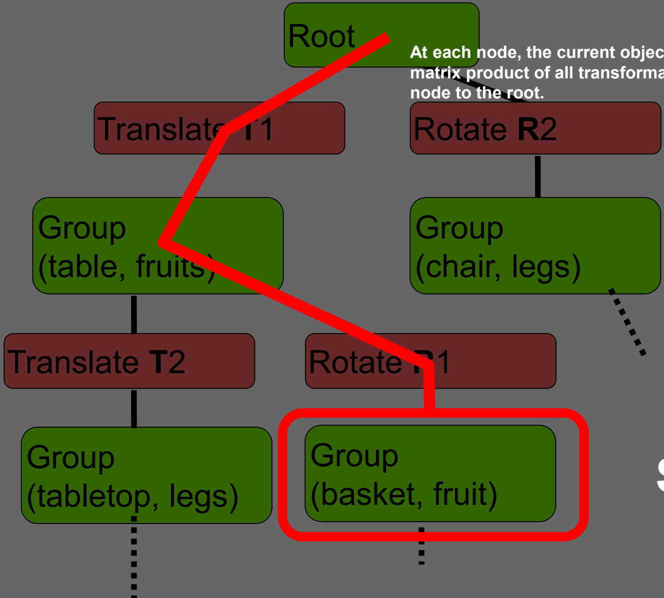
S = R2

Traversal Example



S = R2

Traversal Example



At each node, the current object-to-world transformation is the matrix product of all transformations found on the way from the node to the root.

S = T1R1

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix **(Why?)**

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T*****T**-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn’t exist!

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T*****T**-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn’t exist!

Can you think of a data structure suited for this?

Traversal State – Stack

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T*****T**-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn't exist!

- **Solution:** Keep state variables in a **stack**
  - Push current state when entering node, update current state
  - Pop stack when leaving state-changing node
  - See what the stack looks like in the previous example!

Questions?

Plan

- Hierarchical Modeling, Scene Graph
- OpenGL matrix stack
- Hierarchical modeling and animation of characters
  - Forward and inverse kinematics

Hierarchical Modeling in OpenGL

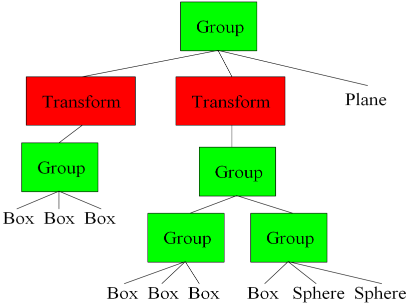
- The OpenGL Matrix Stack implements what we just did!

- Commands to change current transformation
  - glTranslate, glScale, etc.
- Current transformation is part of the OpenGL state, i.e., all following draw calls will undergo the new transformation
  - Remember, a transform affects the whole subtree
- Functions to maintain a matrix stack
  - glPushMatrix, glPopMatrix
- Separate stacks for modelview (object-to-view) and projection matrices

When You Encounter a Transform Node

- Push the current transform using glPushMatrix()
- Multiply current transform by node's transformation
  - Use glMultMatrix(), glTranslate(), glRotate(), glScale(), etc.
- Traverse the subtree
  - Issue draw calls for geometry nodes
- Use glPopMatrix() when done.

- Simple as that!



More Specifically…

- An OpenGL transformation call corresponds to a matrix **T**
- The call multiplies current modelview matrix **C** by **T** from the right, i.e. **C′** = **C** * **T**.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be **p′** = **C** * **T** * **p**
  - Vertices are column vectors on the right in OpenGL
  - This implements hierarchical transformation directly!

More Specifically...

- An OpenGL transformation call corresponds to a matrix **T**
- The call multiplies current modelview matrix **C** by **T** from the right, i.e. **C′** = **C** * **T**.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be **p′** = **C** * **T** * **p**
  - Vertices are column vectors on the right in OpenGL
  - This implements hierarchical transformation directly!

- At the beginning of the frame, initialize the current matrix by the viewing transform that maps from world space to view space.
  - For instance, glLoadIdentity() followed by gluLookAt()

TIEA311 - Code!

Let us revisit the Assignment 0 example that draws a teapot.

Can we use the “theory” just presented to make a controlled scene of a couple of more teapots in their own object coordinates wrt. the world and the camera?

Does real OpenGL code look the way it was just “promised”?

[live, on-screen, if Visual Studio is working also in today’s lecture room]

Questions?

- Further reading on OpenGL
  Matrix Stack and hierarchical model/view transforms
  - http://www.glprogramming.com/red/chapter03.html

- It can be a little confusing if you don't think the previous through, but it's really quite simple in the end.
  - I know very capable people who after 15 years of experience still resort to brute force (trying all the combinations) for getting their transformations right, but it's such a waste :)

Plan

- Hierarchical Modeling, Scene Graph
- OpenGL matrix stack
- Hierarchical modeling and animation of characters
  - Forward and inverse kinematics

Animation

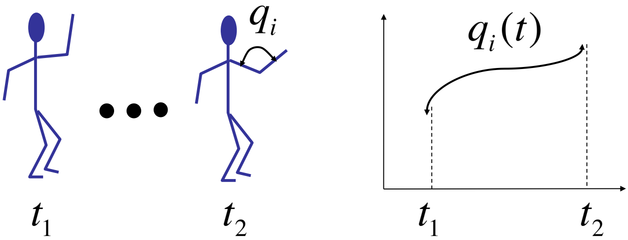
- Hierarchical structure is essential for animation
  - Eyes move with head
  - Hands move with arms
  - Feet move with legs
  - …

- Without such structure the model falls apart.

Articulated Models

- **Articulated models** are rigid parts connected by joints
  - each joint has some angular degrees of freedom

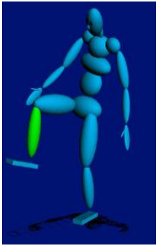
- Articulated models can be animated by specifying the joint angles as functions of time.



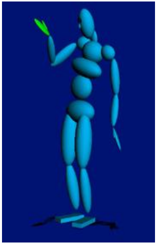
Joints and bones

- Describes the positions of the body parts as a function of joint angles.
  - Body parts are usually called “bones”
- Each joint is characterized by its degrees of freedom (dof)
  - Usually rotation for articulated bodies

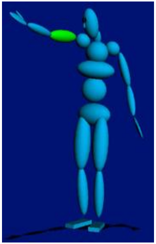
1 DOF: knee



2 DOF: wrist



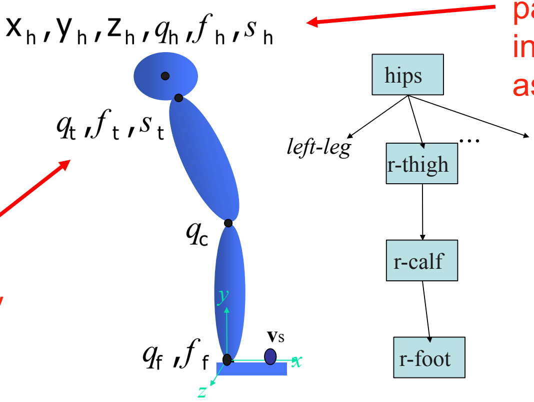
3 DOF: arm



Skeleton Hierarchy

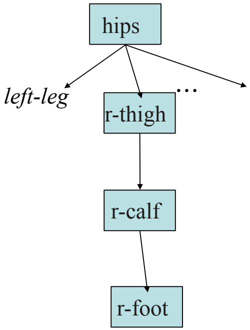
- Each bone position/orientation described relative to the parent in the hierarchy:

For the root, the parameters include a position as well



Joints are specified by angles.

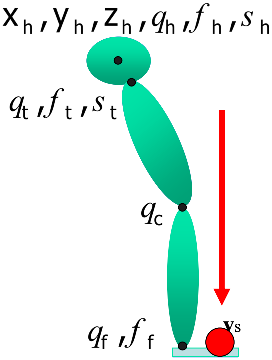
Draw by Traversing a Tree




```
glLoadIdentity();
glPushMatrix();
  glTranslatef(…);
  glRotate(…);
  drawHips();
  glPushMatrix();
    glTranslate(…);
   glRotate(…);
   drawThigh();
   glTranslate(…);
   glRotate(…);
   drawCalf();
   glTranslate(…);
   glRotate(…);
   drawFoot();
  glPopMatrix();
```
left-leg

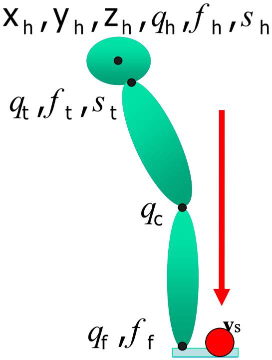
- Assumes drawing procedures for thigh, calf, and foot use joint positions as the origin for a drawing coordinate frame

Forward Kinematics



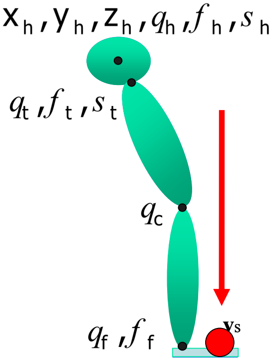
How to determine the world-space position for point **v**s?

Forward Kinematics



Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the joint angles between here and root.

Forward Kinematics



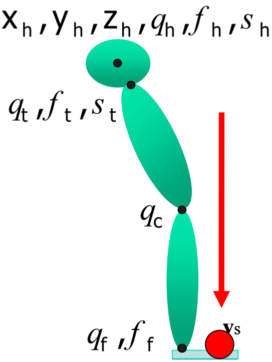
Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the joint angles between here and root.

Note that the angles have a non-linear effect.

This product is **S**

$$\mathbf{v}_w = \boxed{\mathbf{T}(x_h, y_h, z_h)\mathbf{R}(q_h, f_h, s_h)\ \mathbf{TR}(q_t, f_t, s_t)\ \mathbf{TR}(q_c)\ \mathbf{TR}(q_f, f_f)}\ \mathbf{v}_s$$

Forward Kinematics



Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the joint angles between here and root.

Note that the angles have a non-linear effect.

This product is **S**

$$\mathbf{v}_w = \mathbf{T}(x_h, y_h, z_h)\mathbf{R}(q_h, f_h, s_h)\ \mathbf{TR}(q_t, f_t, s_t)\ \mathbf{TR}(q_c)\ \mathbf{TR}(q_f, f_f)\ \mathbf{v}_s$$

$$v_w = S\left(\underbrace{x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f}_{p}\right) v_s = S(p)v_s$$

parameter vector **p**

Questions?

TIEA311 - Today in Jyväskylä

Today (if Visual Studio allows):

- Assignment 2 and 4 live. Some C++ language features weren't used in the earlier ones. Also, Assignment 4 is a bit larger code with less functionality implemented in the starter pack. Warm ups are done. Now we start working!
- C++ static member functions (i.e., "static methods")
- C++ object instantiation using constructors, operator overloading, temporary objects, pass-by-value vs. pass-by-reference
- C++ (and C) pass-by-pointer
- C++ pointer types and inheritance
- Dots, asterisks, ampersands, and arrows in C++ (and C)

C++

- 3 ways to pass arguments to a function
  - by value, e.g. float f(float x)
  - by reference, e.g. float f(float &x)
    - f can modify the value of x
  - by pointer, e.g. float f(float *x)
    - x here is a just a memory address
    - motivations:
      less memory than a full data structure if x has a complex type
      dirty hacks (pointer arithmetic),but just do not do it
    - clean languages do not use pointers
    - kind of redundant with reference
    - arrays are pointers

Pointers

- Can get it from a variable using &
  - often a BAD idea. see next slide
- Can be dereferenced with *
  - float *px=new float; // px is a memory address to a float
  - *px=5.0; //modify the value at the address px
- Should be instantiated with new. See next slide

Pointers, Heap, Stack

- Two ways to create objects
  - The BAD way, on the stack
    - myObject *f() {
      - myObject x;
      - ...
      - return &x
    - will crash because x is defined only locally and the memory gets de-allocated when you leave function f
  - The GOOD way, on the heap
    - myObject *f() {
      - myObject *x=new myObject;
      - ...
      - return x
    - but then you will probably eventually need to delete it

Segmentation Fault

- When you read or, worse, write at an invalid address
- Easiest segmentation fault:
  - float *px; // px is a memory address to a float
  - *px=5.0; //modify the value at the address px
  - Not 100% guaranteed, but you haven't instantiated px, it could have any random memory address.
- 2nd easiest seg fault
  - Vector<float> vx(3);
  - vx[9]=0;

Segmentation Fault

- TERRIBLE thing about segfault: the program does not necessarily crash where you caused the problem
- You might write at an address that is inappropriate but that exists
- You corrupt data or code at that location
- Next time you get there, crash

- When a segmentation fault occurs, always look for pointer or array operations before the crash, but not necessarily at the crash

Debugging

- Display as much information as you can
  - image maps (e.g. per-pixel depth, normal)
  - OpenGL 3D display (e.g. vectors, etc.)
  - cerr<< or cout<< (with intermediate values, a message when you hit a given if statement, etc.)
- Doubt everything
  - Yes, you are sure this part of the code works, but test it nonetheless
- Use simple cases
  - e.g. plane z=0, ray with direction (1, 0, 0)
  - and display all intermediate computation

Questions?

TIEA311 - Today in Jyväskylä (in Finnish)

The “steps of Jarno” (Ajattelumallia tehtävien ratkaisuun):

1. Luentomateriaali
2. Tehtävänanto (muista mitä aiemmissa tehtävissä on tehty/annettu)
3. Hae lähdekoodi ja testaa sen toiminta
4. Yhdistä teoria tehtävään ja lähdekoodiin, ymmärrä kokonaisuus
5. Hahmottele kevyt ”speksi” esim. paperille UML, prosessikaavio, ...

   ————————————————————-

6. Tee osatehtävä 1
7. Päivitä ”speksi”
8. Tee osatehtävä 2
9. Päivitä ”speksi”

   ...

TIEA311 - Today in Jyväskylä

The time allotted for this week's graphics lectures is now over.

Next lecture happens in 6 days and 4 hours.

The teacher will now tell his view about what **could be useful activities** for you during that time period.

$\rightarrow$ see lecture video.

Make notes, if you have to.

Even if he forgets to say it, **remember to rest, too**!