goals

• note: these notes supersede the book.
• 1. we now have the **RigTform** data type to represent rigid body matrices
• for object modeling (say a robot’s lower arm), we still want to have scaling.
• 2. in our code we have duplicated code for drawing each object
  – it would be cleaner to keep some kind of data structure around to represent the scene. a “scene graph”.
• 3. in many cases, we want to manipulate an object, like a robot hierarchically.
  – we have an elbow frame so we can rotate this joint.
  – but when we rotate a shoulder joint, we want the elbow joint to move along with it.
• so we want to encode these relationships in the scene graph

lets start with Scales: problem

• as mentioned, we can’t put scales in our **RigTform**
• more fundamentally, if we apply one of our Rot matrices wrt to a non uniformly scaled frame,
  – say putting a rotation to the right of a scale matrix
• ... we will get wackiness (demo).
• so we should probably keep all of our scale transforms on the right side of any matrix sequence.

scales: solution

• for the frame associated with drawable object
  – which we will soon store in a ‘‘*shape node’’
  – a bone, as opposed to a joint,
• ... we will store an explicit *separate* affine matrix (not a RigTform)
• so if \( \vec{l} \) is an rhon elbow frame, then for the lower arm bone, which is an elongated cube, we will store (in its shape node) a fixed matrix which is of the form \( B := (\text{Trans} \ast \text{Scale}) \)
• and define the bone’s frame as \( \vec{b} = \vec{l} B \)
  – (reading left to right) the translation puts a frame at the center of the bone, and the scale elongates the frame.
• then we can use for the bone’s object coordinates, those of a canonical cube
• during manipulation, we will update \( \vec{b} \) by rotating it as desired.
• but we will not mess with the shape node data, \( B \).
• and we will never try to do any rotation wrt \( \vec{b} \).
• actually, in our *SgGeometryShapeNode*, we will allow one to set \( B \) as \( TR_x R_y R_z S \)

hierarchy

• lets imagine a shoulder frame \( \vec{s} \) and an elbow frame \( \vec{l} \).
• if we rotate the shoulder, we want the elbow frame to rotate as well.
• ie. we want the relationship between \( \vec{s} \) and \( \vec{l} \) to remain fixed
• this means \( \vec{l} = \vec{s} L \) where \( L \) is a fixed RigTform.
• so lets have one “transform node” for the shoulder, and one for the elbow.
• lets store the elbow node as a “child” of the shoulder node
• lets store \( L \) in the elbow node.
• when we want to rotate the elbow, we will update the \( L \) in the elbow’s node.
• if we want to rotate the shoulder, we leave \( L \) alone and do something at the shoulder transform node.

**scene graph**

• if we follow this logic, this will naturally lead us to a scene graph
  a rooted tree of \( \text{SgNodes} \).
• we have two kinds \( \text{SgTransformNodes} \) and \( \text{SgShapeNodes} \).
  – transform nodes return RBTs (in some representation)
  – shape nodes return general affine transforms, as Matrix4s, and can draw themselves

**root**

• at the root we have a transform node, which represents the world frame \( \vec{w}^t \).
  – its \( \text{getRbt()} \) returns the identity \( \text{RigidTform} \).
  – for this, we will use the \( \text{SgRootNode} \) type, a type derived from \( \text{SgTransformNode} \).

**children: transformation**

• each transformation node can have child nodes representing dependent frames.
• a child transformation node stores a \( \text{RigidTform} \) relating its rhon frame to its parent
  – examples: robot object: \( \vec{s} = \vec{w}^t O \), shoulder: \( \vec{s} = \vec{a}^t S \), elbow: \( \vec{l} = \vec{s}^t L \).
  – for this will use the \( \text{SgRbtNode} \) type, derived from \( \text{SgTransformNode} \).

**children: shape**

• each transformation node can have child nodes for things to draw, a \( \text{SgShapeNode} \)
• a shape node returns an affine matrix relating its (non rhon) frame to its parent
  – lower arm bone stores \( B \) describing \( \vec{b}^t = \vec{t}^t B \)
• shape can also draw itself.
• we will use the \( \text{SgGeometryShapeNode} \) type, derived from \( \text{SgShapeNode} \), that stores a Matrix4, a color and a pointer to a \( \text{Geometry} \) object
  – our lower arm’s geometry will just be a cube

**instancing**

• the cube geometry object can be shared between many shape nodes
• this avoids data duplication

**our scene**

• in our scene the root will have children for the skyCam, the ground plane, and each robot.
  – later on, we will also put the lights in the scene graph
• our global pointers to Rbts and geometry should all be replaced by node pointers
• to draw the scene, in display we call \text{drawStuff} which calls
Drawer drawer(invEyeRbt, curSS)
g_world->accept(drawer);

what happens inside of Drawer

• the tree is recursively traversed (dfs) starting at the calling node (g_world).
• a “RBT stack” is maintained, starting with $E^{-1}$.
• at each descent, upon “entrance” to a transform node
  – the top of the stack is duplicated and its own transform is right multiplied to the top.
  – so as the traversal goes world, robot, shoulder, elbow, the stack grows: \{ $E^{-1}$ \}, \{ $E^{-1}$, $E^{-1}O$ \}, \{ $E^{-1}$, $E^{-1}O$, $E^{-1}OS$ \}, \{ $E^{-1}$, $E^{-1}O$, $E^{-1}OS$, $E^{-1}OSL$ \}
• when the traversal hits a shape node (say lower arm),
  – it grabs the top of the stack (say $E^{-1}OSL$)
  – right multiplies by the node’s matrix (producing, say $E^{-1}OSLB$)
  – sends the MVM (and NMVM) to the shaders
  – sends the color to the shader.
  – draws the Geometry object.
• before a transform child returns, the stack is popped.

how is this coded Drawer

• the above dfs, stack maintenance, and drawing could have been done in one codeset.
• but it is more convenient to have one set of code that does just the dfs, and another set of code, called the “visitor”, which does anything else.
• for this we will have a data type \textbf{class Drawer : public SgNodeVisitor}
• look at drawer.h

picking

• we want to be able to click on an object and “pick it”
• when we enter picking mode (p key and click), we will draw the scene using a solid fragment shader, and each object’s color will identify it.
  – we will not swap the buffers, so this will not appear on the screen.
• then we just have to look at the color of the pixel to find the id.
• when a bone is picked, we will “activate” its parent joint for manipulation.
  – ie. we will grab a pointer to its parent’s SgRbtNode.

picking visitor

• picking will be accomplished by writing a new visitor class \textbf{class Picker : public SgNodeVisitor}
• and we will call

  Picker picker(invEyeRbt, curSS}
g_world->accept(picker);

• this visitor will have its own private \textit{Drawer} instance.
• during traversal, this visitor will keep a “node-pointer stack”
• at a shape node, an id color is computed and associated to the node pointed to at the top of the node stack in a “map” data structure.
  – this color is used to set “uIdColor”
• the picker’s visit functions will also call the associated Drawer’s visit function.
• now the scene has been drawn and the map created
• then the observed pixel valued can be used to get the id which can be used to get a pointer to the node.

accumulated Rbt
• we will also need a function
  ```cpp
  RigTForm getPathAccumRbt
  (shared_ptr<SgTransformNode> source, 
   shared_ptr<SgTransformNode> destination, 
   int offsetFromDestination=0);
  ```
• which gives us the product of the RBTS going from source to dest.
  – this product will not include the source itself, but in our code, it will always be the root anyway.
• this will also be computed using a new visitor class **class RbtAccumVisitor : public SgNodeVisitor** that you will complete
  – its constructor takes in the destination pointer, which it stores.
• this visitor just maintains a matrix stack during traversal.
  – but it exits the traversal when the destination is hit.
  – a return value of false from a visitor will end the traverser!
• with this, we can now draw the archball!
• with this, we can set the eye to be at any frame in the scene.

**joint manipulation**
• one more bit of math.
• suppose the elbow joint $\vec{P} = \vec{s}L$ is activated
• the mouse motion gives us the desired action RBT $M$.
• lets call the auxiliary frame $\vec{a} = \vec{w}A$
• this should be a frame with the eye’s directions and the joint’s center
• this means $A = (C(l))_T * (C(e))_R$
  – where $C$ is the accumulated RBT (starting at the world) that we just described

**joint manipulation updating**
• we already have code for $\text{doMtoOwrtA}$
• its derivation assumed we were going to update $\vec{a} = \vec{w}O$.
• but in our setting we want to update $L$.
  – which represents the relationship $\vec{P} = \vec{s}L$, NOT $\vec{P} = \vec{w}L$,
• so we need to do our work with $\vec{s}$ as our base frame, not $\vec{w}$.
• so we need to calculate an RBT $A_s$ such that $\vec{a} = \vec{w}A = \vec{s}A_s$.
• once we have that, then we can set $L = \text{doMtoOwrtA}(M, L, A_s)$