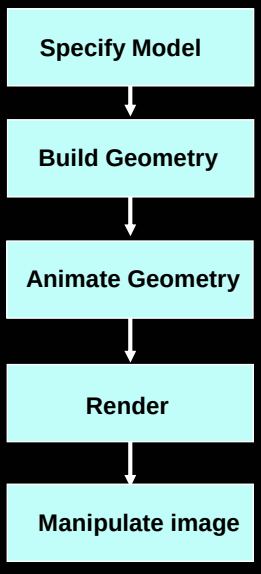
Hierarchical Modeling

A lesson in stick person anatomy.
or
Choosing the right parameters.
Hierarchical transformations.
The matrix stack.

See Angel 9.1-9.7

Staying Oriented (in the course)



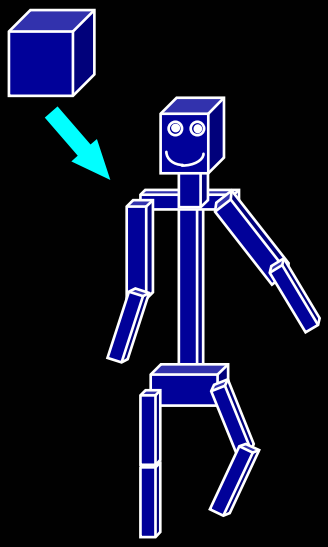
The framework for
the topics we're covering

transformations,
hierarchies

time dependent
transformations

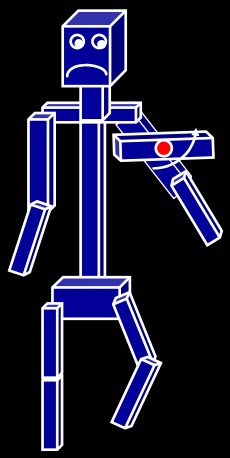
raster ops,
paint

Modeling with Transformations



- You've learned everything you need to know to make a stick person out of cubes.
- Just translate, rotate, and scale each one to get the right size, shape, position, and orientation.
- Looks great--until you try to make it move.

The Right Control Knobs



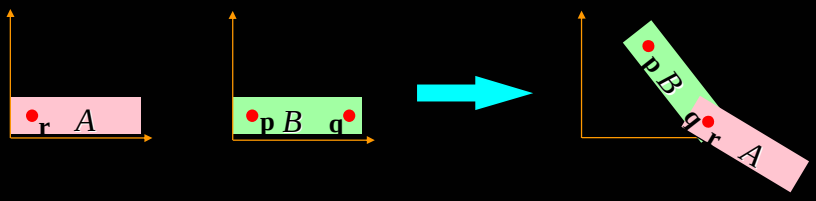
- As soon as you want to change something, the model falls apart
- Reason: the thing you're modeling is *constrained* but your model doesn't know it
- What we need:
  - some sort of representation of *structure*
  - a set of "control knobs" (parameters) that make it easy to move our stick person through *legal configurations*
- This kind of control is convenient for static models, and *vital* for animation!
- Key is to structure the transformations in the right way: using a hierarchy

Hierarchical Modeling Example

"Number One" Playgroup - Duran Duboi
Issue 141: SIGGRAPH 2002 Electronic Theater Program

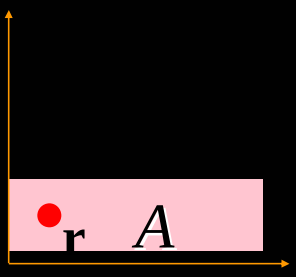
```
HIERARCHY
  ROOT Hips {
     OFFSET 0.00 0.00 0.00
     CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
     JOINT Torso {
        OFFSET 0.00 5.21 0.00
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT Neck {
            OFFSET 0.00 18.65 0.00
            CHANNELS 3 Zrotation Xrotation Yrotation
            ...
        }
      }
      JOINT LeftUpLeg {
         OFFSET 3.91 0.00 0.00
         CHANNELS 3 Zrotation Xrotation Yrotation
         JOINT LeftLowLeg {
             OFFSET 0.00 -18.34 0.00
             CHANNELS 3 Zrotation Xrotation Yrotation
             ...
```

Making an Articulated Model



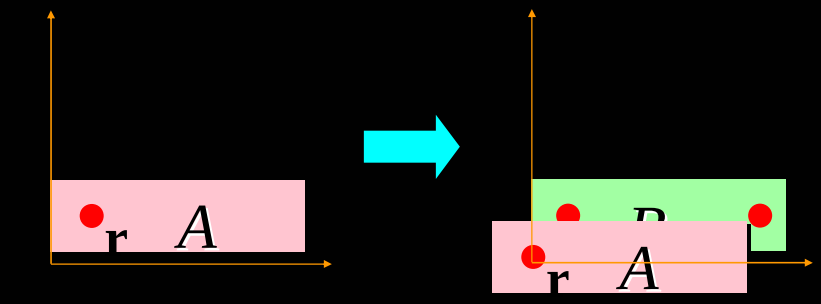
- A minimal 2-D jointed object:
  - Two pieces, *A* (“forearm”) and *B* (“upper arm”)
  - Attach point q on *B* to point r on *A* (“elbow”)
  - Desired control knobs:
    - » T: shoulder position (point at which p winds up)
    - » u: shoulder angle (*A* and *B* rotate together about p)
    - » v: elbow angle (*A* rotates about r, which stays attached to q)

Making an Arm, step 1



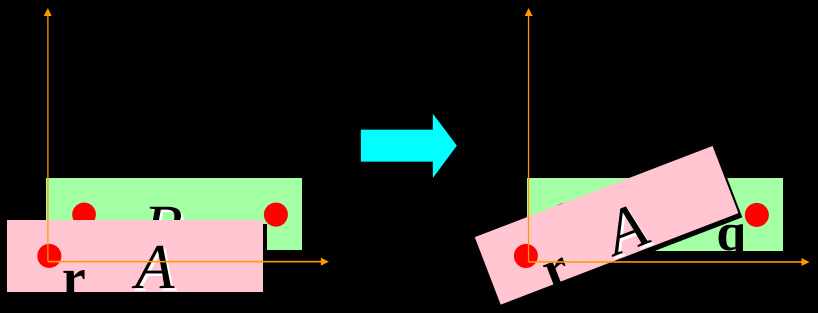
- Start with *A* and *B* in their untransformed configurations (*B* is hiding behind *A*)
- First apply a series of transformations to *A*, leaving *B* where it is…

Making an Arm, step 2



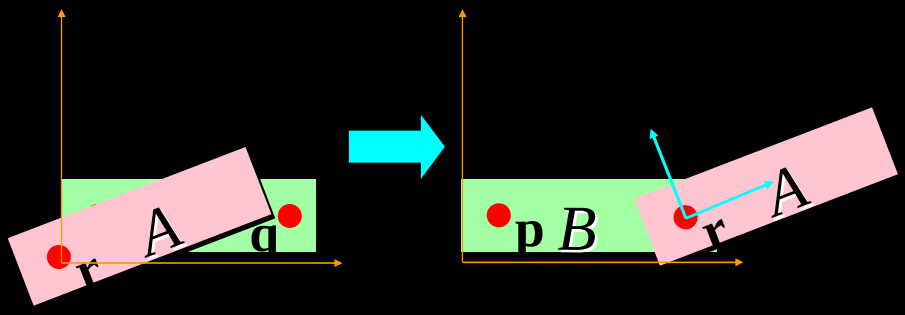
- Translate by -r, bringing r to the origin
- You can now see *B* peeking out from behind *A*

Making an Arm, step 3



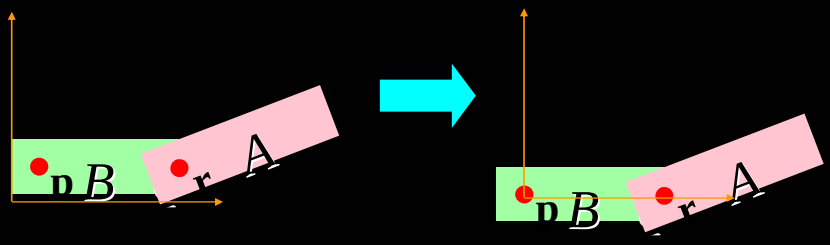
- Next, we rotate *A* by v (the “elbow” angle)

Making an Arm, step 4



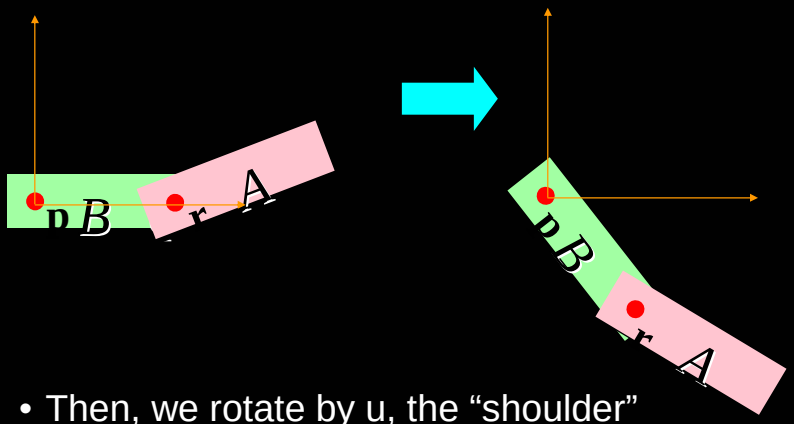
- Translate *A* by q, bringing r and q together to form the elbow joint
- We can regard q as the origin of the *lower arm coordinate system*, and regard A as being in this coordinate system.

Making an Arm, step 5



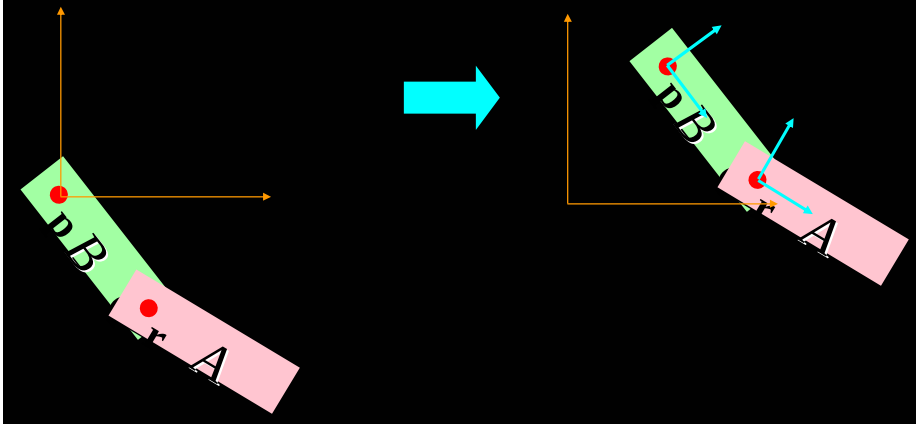
- From now on, each transformation applies to *both A* and *B* (This is important!)
- First, translate by -p, bringing p to the origin
- *A* and *B* both move together, so the elbow doesn't separate!

Making an Arm, step 6



- Then, we rotate by u, the “shoulder” angle
- Again, *A* and *B* rotate together

Making an Arm, step 7

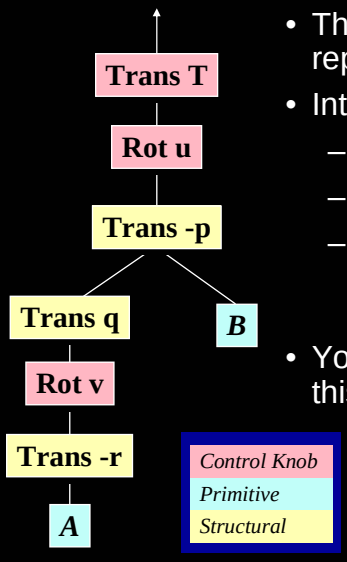


- Finally, translate by T, bringing the arm where we want it
- p is at origin of *upper arm coordinate system*

So What Have We Done?

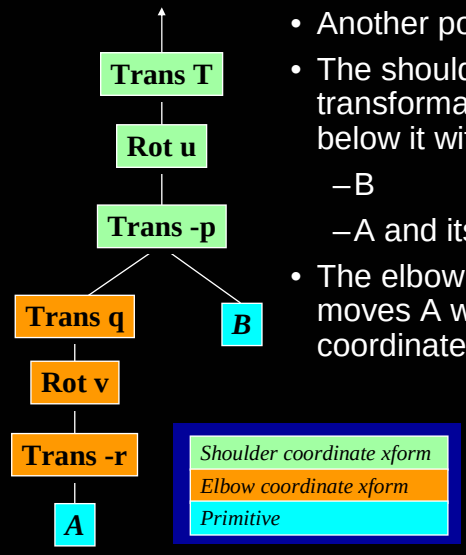
- Seems more complicated than just translating and rotating each piece separately
- But the model is easy to modify/animate:
  - Remember the transformation sequence, and the parameters you used—they're part of the model.
  - Whenever the parameters change, reapply all of the transformations and draw the result
    - » The model will not fall apart!!!
- Note:
  - u, v, and T are parameters of the model.
  - but p, q, and r are *structural constants.*
  - Changing u,v, or T wiggles the arm
  - Changing p,q, or r *dismembers it* (useful only in video games!)

Transformation Hierarchies



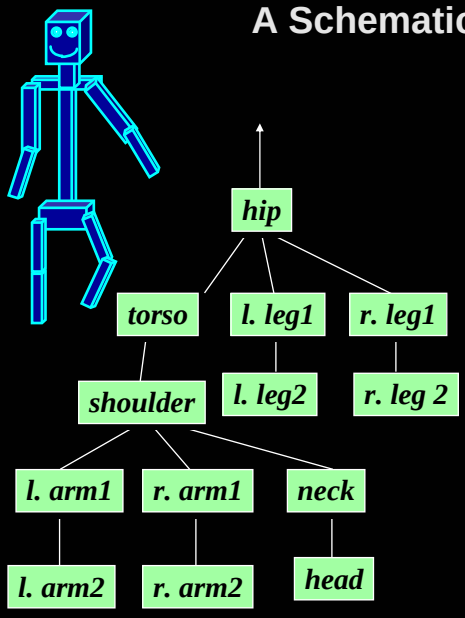
- This is the build-an-arm sequence, represented as a tree
- Interpretation:
  - Leaves are geometric primitives
  - Internal nodes are transformations
  - Transformations apply to everything under them—start at the bottom and work your way up
- You can build a wide range of models this way

Transformation Hierarchies



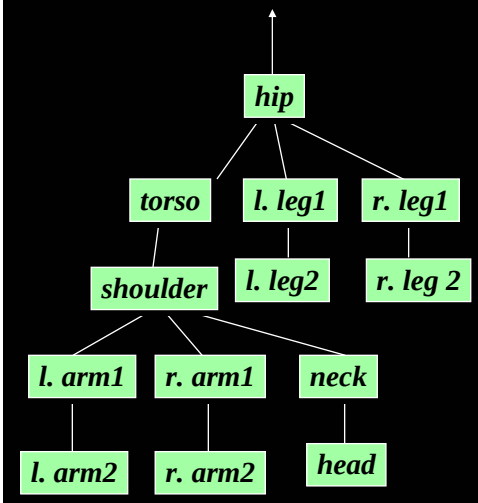
- Another point of view:
- The shoulder coordinate transformation moves everything below it with respect to the shoulder:
  - B
  - A and its transformation
- The elbow coordinate transformation moves A with respect to the shoulder coordinate transform

A Schematic Humanoid



- Each node represents
  - rotation(s)
  - geometric primitive(s)
  - struct. transformations
- The root can be anywhere. We chose the hip *(can re-root)*
- Control knob for each joint angle, plus global position and orientation
- A realistic human would be *much* more complex

Directed Acyclic Graph



- This is a graph, so you can re-root it (make head the root)
- It's *directed*, rendering traversal only follows links one way.
- It's *acyclic*, to avoid infinite loops in rendering.
- Not necessarily a tree.
  - e.g. l.arm2 and r.arm2 primitives might be two instantiations (one mirrored) of the same geometry

```
HIERARCHY
  ROOT Hips {
     OFFSET 0.00 0.00 0.00
     CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
     JOINT Torso {
        OFFSET 0.00 5.21 0.00
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT Neck {
            OFFSET 0.00 18.65 0.00
            CHANNELS 3 Zrotation Xrotation Yrotation
            ...
        }
      }
      JOINT LeftUpLeg {
         OFFSET 3.91 0.00 0.00
         CHANNELS 3 Zrotation Xrotation Yrotation
         JOINT LeftLowLeg {
             OFFSET 0.00 -18.34 0.00
             CHANNELS 3 Zrotation Xrotation Yrotation
             ...
```

What Hierarchies Can and Can't Do

- Advantages:
  - Reasonable control knobs
  - Maintains structural constraints
- Disadvantages:
  - Doesn't always give the "right" control knobs trivially
    - » e.g. hand or foot position - re-rooting may help
  - Can't do closed kinematic chains easily (keep hand on hip)
  - Missing other constraints: do not walk through walls
- Hierarchies are a vital tool for modeling and animation

So What Have We Done?

- Forward Kinematics
  - Given the model and the joint angles, where is the end effector?
    - » In graphics compute this so you know where to draw
    - » In robotics compute this to know how to control the end effector
- Inverse Kinematics
  - Given a desired location of the end effector, what are the required joint angles to put it there.
    - » In robotics, required to place the end effector near to objects in real world

Inverse Kinematics is useful in animation as well

Kinematics is easy, IK is hard because of redundancy.

Implementing Hierarchies

- Building block: a *matrix stack* that you can push/pop
- Recursive algorithm that descends your model tree, doing transformations, pushing, popping, and drawing
- Tailored to OpenGL's state machine architecture (or vice versa)
- Nuts-and-bolts issues:
  - What kind of nodes should I put in my hierarchy?
  - What kind of interface should I use to construct and edit hierarchical models?
- Extensions:
  - expressions, languages.

The Matrix Stack

- Idea of Matrix Stack:
  - LIFO stack of matrices with push and pop operations
  - *current transformation matrix* (product of all transformations on stack)
  - transformations modify matrix at the top of the stack
- Recursive algorithm:
  - load the identity matrix
  - for each internal node:
    - » push a new matrix onto the stack
    - » concatenate transformations onto current transformation matrix
    - » recursively descend tree
    - » pop matrix off of stack
  - for each leaf node:
    - » draw the geometric primitive using the current transformation matrix

Relevant OpenGL routines

`glPushMatrix(), glPopMatrix()`

push and pop the stack. push leaves a copy of the current matrix on top of the stack

`glLoadIdentity(), glLoadMatrixd(M)`

load the Identity matrix, or an arbitrary matrix, onto top of the stack

`glMultMatrixd(M)`

multiply the matrix C on top of stack by M. C = CM

`glRotatef(theta,x,y,z), glRotated(…)`

axis/angle rotate. "f" and "d" take floats and doubles, respectively

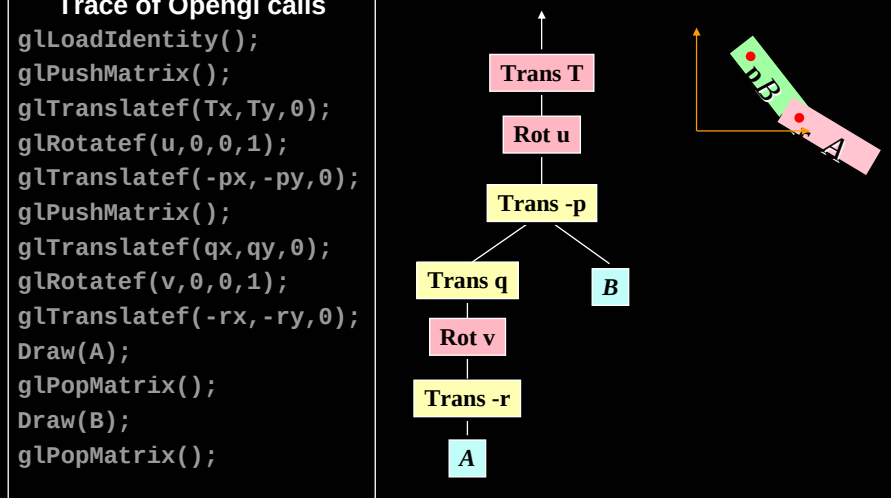
`glTranslatef(x,y,z), glScalef(x,y,z)`

translate, rotate. (also exist in "d" versions.)

Two-link arm, revisited, in OpenGL

Trace of Opengl calls

```
glLoadIdentity();
glPushMatrix();
glTranslatef(Tx,Ty,0);
glRotatef(u,0,0,1);
glTranslatef(-px,-py,0);
glPushMatrix();
glTranslatef(qx,qy,0);
glRotatef(v,0,0,1);
glTranslatef(-rx,-ry,0);
Draw(A);
glPopMatrix();
Draw(B);
glPopMatrix();
```



Building Hierarchies: What Should Transformation Nodes Do?

- Separate nodes for translation, rotation and scale
  - + lots of flexibility
  - – many nodes making select-and-click difficult
- Nodes perform multiple transformations in hard-wired sequence, e.g. rotate-translate-scale
  - + less complex tree
  - – hard-wired sequences are less flexible

Hardwired Group Transformation Sequence

- Must select a good hard-wired sequence that the user will think is intuitive
  - – Rule of thumb: scale before rotate
    - » avoid object shearing during rotation
  - – Rule of thumb: rotate before translate
    - » make sure rotation occurs about correct point
- Occasionally this sequence won't be enough - a more flexible scheme is required

Example Transformation Sequence

- Parameters (2D)
  - (cx, cy): center of rotation and scaling
  - (sx, sy): scaling
  - theta: rotation
  - (tx, ty): translation

- Full sequence of primitive transformations:
  - trans(-cx, -cy) *move center to origin*
  - scale(sx,sy) *scale*
  - rot(theta) *rotate*
  - trans(cx,cy) *move center back*
  - trans(tx,ty) *translate (can combine with previous)*

Variables and Expressions

- Better control can come from the transformation parameters being functions of other variables
- Simple example:
  - a clock with second, minute and hour hands
  - hands should rotate together
  - express all the motions in terms of a "seconds" variable
  - whole clock is animated by varying the seconds parameter

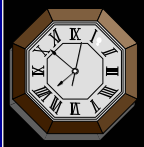
$$m = \frac{s}{60}$$

$$h = \frac{m}{60}$$

$$\theta_s = \frac{2\pi s}{60}$$

$$\theta_m = \frac{2\pi m}{60}$$

$$\theta_h = \frac{2\pi h}{12}$$



Or arms and legs of a walking human figure

Getting Expressions into Your Models

- Some commercial systems (e.g. Maya) have expression-evaluating facilities.
- Some high-end systems (e.g. Pixar's in-house system) contain full-blown embedded interpreted languages — most of their models are really programs.
- If you write your models in a general-purpose language, interpreted or not, you get this for free.
- The trick is to avoid losing too much speed in the process.
- The example on the next slide shows (very schematically) how you might go about writing C code to draw a complex hierarchical model.

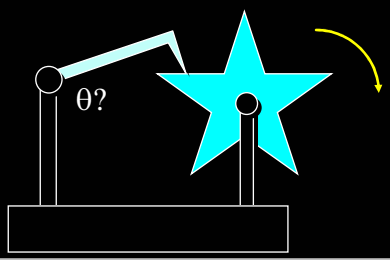
Models as Code: draw-a-bug.

```
void draw_bug(walk_phase_angle, xpos, ypos zpos){
  pushmatrix
  translate(xpos,ypos,zpos)
  calculate all six sets of leg angles based on
    walk phase angle.
  draw bug body
  for each leg:
    pushmatrix
    translate(leg pos relative to body)
    draw_bug_leg(theta1 & theta2 for that leg)
    popmatrix
  popmatrix
}

void draw_bug_leg(float theta1, float theta2){
  glPushMatrix();
  glRotatef(theta1,0,0,1);
  draw_leg_segment(SEGMENT1_LENGTH)
  glTranslatef(SEGMENT1_LENGTH,0,0);
  glRotatef(theta2,0,0,1);
  draw_leg_segment(SEGMENT2_LENGTH)
  glPopMatrix();
}
```

Hard Examples

- A walking humanoid that swings its arms and bobs its head, under control of a single variable, so it walks when you "turn the crank." (you'd have extra parameters for walking style, of course.)
- In the figure below, what expression would you use to calculate the arm's rotation angle to keep the tip on the star-shaped wheel as the wheel rotates???
- This gets arbitrarily hard. There's got to be a better way to do constraints. We'll get back to this topic when we do animation.



Announcements