Assignments

- Assignment 0 grades in Stellar
- Curves & Surfaces due next Wednesday
- We recommend posting questions to Stellar message board
  - We are patrolling the boards frequently, but you can of course notify us by email if it’s urgent
  - Remember that giving good advice to others can help your grade!
Normals for Swept Surfaces

profile \( q(v) \)

trajectory \( c(v) \)
Normals for Swept Surfaces

trajectory $c(v)$

unit tangent of $c(v)$

trajectory $c(v)$

6.837 – Durand

Tuesday, September 28, 2010
Normals for Swept Surfaces

trajectory $c(v)$

unit tangent of $c(v)$

plane $P(v)$ that is perpendicular to tangent

coordinate axes for plane $P$

6.837 – Durand
Normals for Swept Surfaces

plane P(v) that is perpendicular to tangent

profile curve q(u), transformed onto plane P

trajectory c(v)

coordinate axes for plane P

unit tangent of c(v)

Tuesday, September 28, 2010
Normals for Swept Surfaces

plane $P(v)$ that is perpendicular to tangent

profile curve $q(u)$, transformed onto plane $P$

coordinate axes for plane $P$

unit tangent of $c(v)$

point $p(u,v)$

trajectory $c(v)$
Normals for Swept Surfaces

unit tangent of $q(u)$, transformed onto plane $P$

plane $P(v)$ that is perpendicular to tangent

point $p(u,v)$

profile curve $q(u)$, transformed onto plane $P$

coordinate axes for plane $P$

unit tangent of $c(v)$

trajectory $c(v)$
Normals for Swept Surfaces

- **trajectory** $c(v)$
- **unit tangent** of $c(v)$
- **profile curve** $q(u)$, transformed onto plane $P(v)$ that is perpendicular to tangent
- **coordinate axes** for plane $P$
- **point** $p(u,v)$
- **unit tangent** of $c(v)$
- **plane** $P(v)$
Normals for Swept Surfaces

- **unit tangent of** $q(u)$, **transformed onto** plane $P$
- **surface normal** = $x$
- **point** $p(u,v)$
- **coordinate axes** for plane $P$
- **plane** $P(v)$ that **is perpendicular to** tangent
- **profile curve** $q(u)$, **transformed onto** plane $P$
- **trajectory** $c(v)$

6.837 – Durand
Plan

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

- 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, transparency, etc.
  - Group: Collection of subgraphs

- C++ implementation
  - base class Object
    - children, parent
  - derived classes for each node type (group, geometry, etc.)
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, or the watch)
    - 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

```plaintext
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
Hierarchical Transformation of Objects

- A transformation node affects the whole subtree
- Each node has its **local coordinate system**
- Transformations are always specified relative to parent!
  - Aggregate object-to-world transform is the concatenation of all transforms on the way from current node to root

Tuesday, September 28, 2010
Simple Example with Transforms

```
Group {
  numObjects 3
  Transform {
    ZRotate { 45 }
    Group {
      numObjects 3
      Box { <BOX PARAMS> }
      Box { <BOX PARAMS> }
      Box { <BOX PARAMS> }
    } }
  Transform {
    Translate { -2 0 0 }
    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> }
}
```
Sidenote

- Note that we’ve used different nodes for translations, rotations, etc.
- They are all represented by 4x4 matrices and there is no technical reason not to combine them into one matrix
  - It’s often simpler for the user to specify these separately.
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 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)
    ▪ 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** \( T \)!
Traversals Example

- **Root**
  - **Translate** $T_1$
    - **Group** (table, fruits)
      - **Translate** $T_2$
        - **Group** (tabletop, legs)
  - **Rotate** $R_2$
    - **Group** (chair, legs)
      - **Rotate** $R_1$
        - **Group** (basket, fruit)
Traversals Example

- Translate $T_1$
- Rotate $R_2$
- Translate $T_2$
- Rotate $R_1$

$S = I$
Traversing Example

Root

Translate $T_1$

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_1$

Group (basket, fruit)

Rotate $R_2$

Group (chair, legs)

$S = T_1$
Traversals Example

Root

Translate $T_1$

Rotate $R_2$

Group (table, fruits)

Translate $T_2$

Rotate $R_1$

Group (tabletop, legs)

Group (chair, legs)

Group (basket, fruit)

$S = T_1$
Traversing Example

\[
\begin{align*}
S &= T_1 T_2 \\
T_2 &= \text{Translate} \\
R_1 &= \text{Rotate} \\
R_2 &= \text{Rotate} \\
T_1 &= \text{Translate} \\
\end{align*}
\]
Traversing Example

Root

Translate \( T_1 \)

Rotate \( R_2 \)

Group (table, fruits)

Translate \( T_2 \)

Rotate \( R_1 \)

Group (tabletop, legs)

Group (basket, fruit)

\[ S = T_1 T_2 \]
Traversing Example

Root

Translate $T_1$

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_1$

Group (basket, fruit)

Rotate $R_2$

Group (chair, legs)

$S = T_1 \ T_2$
Traversal Example

Translate $T_1$

Root

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_1$

Group (basket, fruit)

Rotate $R_2$

Group (chair, legs)

$S = T_1$
Traversals Example

Root

Translate $T_1$
Group (table, fruits)

Rotate $R_2$
Group (chair, legs)

Translate $T_2$
Group (tabletop, legs)

Rotate $R_1$
Group (basket, fruit)

$S = T_1 R_1$
Traversing Example

S = T₁ R₁
Traversal Example

\[ S = T_1 \, R_1 \]
Traversal Example

Root

Translate $T_1$

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_1$

Group (basket, fruit)

Rotate $R_2$

Group (chair, legs)

$S = T_1$
Traversing Example

Root

Translate $T_1$
Group (table, fruits)
Translate $T_2$
Group (tabletop, legs)

Rotate $R_2$
Group (chair, legs)
Rotate $R_1$
Group (basket, fruit)

$S = T_1$
Traversing Example

Root

Translate $T_1$

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_2$

Group (chair, legs)

Rotate $R_1$

Group (basket, fruit)

$S = I$
Traversing Example

Translate $T_1$ Group (table, fruits)

Rotate $R_2$ Group (chair, legs)

Translate $T_2$ Group (tabletop, legs)

Rotate $R_1$ Group (basket, fruit)

$S = R_2$
Traversal Example

Root

Translate $T_1$

Group (table, fruits)

Translate $T_2$

Group (tabletop, legs)

Rotate $R_2$

Group (chair, legs)

Rotate $R_1$

Group (basket, fruit)

$S = R_2$
Traversing Example

Root

- **Translate** $T_1$
  - **Group** (table, fruits)
    - **Translate** $T_2$
      - **Group** (tabletop, legs)
    - **Rotate** $R_1$
      - **Group** (basket, fruit)
  - **Rotate** $R_2$
    - **Group** (chair, legs)

$S = R_2$
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.

\[ S = T_1 R_1 \]
Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving**
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!
Traversing 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? $\mathbf{T} \mathbf{T}^{-1} = \mathbf{I}$ does not necessarily hold in floating point even when $\mathbf{T}$ is an invertible matrix – you accumulate error
  - Why II? $\mathbf{T}$ 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? $T \cdot T^{-1} = I$ 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!
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. $C' = C \times T$.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be $p' = C \times T \times p$
  - 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. $C' = C \times T$.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be $p' = C \times T \times p$
  - 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()
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.
Forward Kinematics

- 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:

\[ x_h, y_h, z_h, \theta_h, \phi_h, \psi_h \]

For the root, the parameters include a position as well.

Joints are specified by angles.
Draw by Traversing a Tree

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

```c
GLfloatIdentity();
GLfloatMatrix();
    glTranslatef(...);
    glRotate(...);
    drawHips();
    glPushMatrix();
    glTranslate(...);
    glRotate(...);
    drawThigh();
    glTranslate(...);
    glRotate(...);
    drawCalf();
    glTranslate(...);
    glRotate(...);
    drawFoot();
    glPopMatrix();
```
Forward Kinematics

How to determine the world-space position for point $v_s$?
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 $S$

$$v_w = T(x_h,y_h,z_h)R(q_h,f_h,s_h) \ TR(q_t,f_t,s_t) \ TR(q_c) \ TR(q_f,f_f) \ v_s$$
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 $S$

$$v_w = T(x_h, y_h, z_h)R(q_h, f_h, s_h)\ TR(q_t, f_t, s_t)\ TR(q_c)\ TR(q_f, f_f)\ v_s$$

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

parameter vector $p$
Questions?
Inverse Kinematics

- Context: an animator wants to "pose" a character
- Specifying every single angle is tedious and not intuitive
- Simpler interface: directly manipulate position of e.g. hands and feet
- That is, specify $v_w$, infer joint transformations

![Diagram of a character with joint transformations $x_h, y_h, z_h, q_h, f_h, s_h$ and end effector $V_s$.]
Inverse Kinematics

- **Forward Kinematics**
  - Given the skeleton parameters $p$ (position of the root and the joint angles) and the position of the point in local coordinates $v_s$, what is the position of the point in the world coordinates $v_w$?
  - Not too hard, just apply transform accumulated from the root.
Inverse Kinematics

- **Forward Kinematics**
  - Given the skeleton parameters $\mathbf{p}$ (position of the root and the joint angles) and the position of the point in local coordinates $\mathbf{v}_s$, what is the position of the point in the world coordinates $\mathbf{v}_w$?
  - Not too hard, just apply transform accumulated from the root.

- **Inverse Kinematics**
  - Given the current position of the point and the desired new position $\tilde{\mathbf{v}}_w$ in world coordinates, what are the skeleton parameters $\mathbf{p}$ that take the point to the desired position?
Inverse Kinematics

- Given the position of the point in local coordinates \( \mathbf{v}_s \) and the desired position \( \mathbf{v}_w \) in world coordinates, what are the skeleton parameters \( \mathbf{p} \)?

\[
\mathbf{v}_w = \mathbf{S}\left( x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f \right) \mathbf{v}_s = \mathbf{S}(\mathbf{p})\mathbf{v}_s
\]

- Requires solving for \( \mathbf{p} \), given \( \mathbf{v}_s \) and \( \mathbf{v}_w \)
  - Non-linear, and...
It’s Underconstrained

- Count degrees of freedom:
  - We specify one 3D point (3 equations)
  - We usually need more than 3 angles
  - \( p \) usually has tens of dimensions

- Simple geometric example (in 3D):
  - specify hand position, need elbow & shoulder
  - The set of possible elbow location is a circle in 3D
How to tackle these problems?

- Deal with non-linearity:
  Iterative solution (steepest descent)
  - Compute Jacobian matrix of world position w.r.t. angles
    - Jacobian: “If the parameters $\mathbf{p}$ change by tiny amounts, what is the resulting change in the world position $\mathbf{v}_{WS}$?”
    - Then invert Jacobian.
      - This says “if $\mathbf{v}_{WS}$ changes by a tiny amount, what is the change in the parameters $\mathbf{p}$?”
    - But wait! The Jacobian is non-invertible (3xN)
  - Deal with ill-posedness: Pseudo-inverse
    - Solution that displaces things the least
  - Deal with ill-posedness: Prior on “good pose” (more advanced)

- Additional potential issues: bounds on joint angles, etc.
Example: Style-Based IK

- Video
- Prior on “good pose”
- Link to paper: Grochow, Martin, Hertzmann, Popovic: Style-Based Inverse Kinematics, ACM SIGGRAPH 2004
Mesh-Based Inverse Kinematics

- Video

- Doesn’t even need a hierarchy or skeleton: Figure proper transformations out based on a few example deformations!

- Link to paper: Sumner, Zwicker, Gotsman, Popovic: Mesh-Based Inverse Kinematics, ACM SIGGRAPH 2005
That’s All for Today!

Further reading

- OpenGL Matrix Stack and hierarchical model/view transforms
  - [http://www.glprogramming.com/red/chapter03.html](http://www.glprogramming.com/red/chapter03.html)