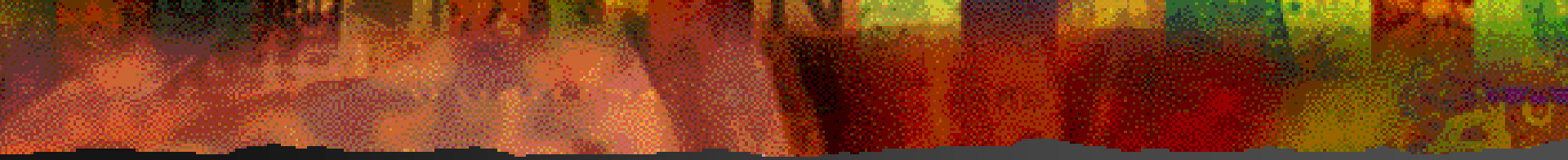


Skeletons

CSE169: Computer Animation

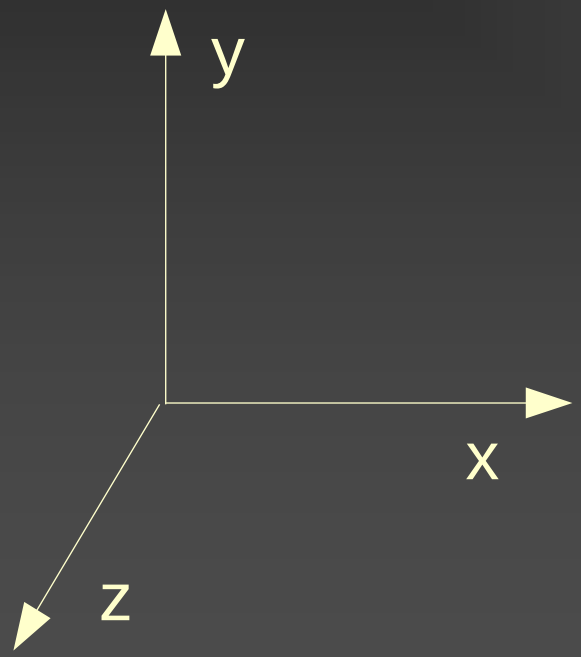
Instructor: Steve Rotenberg

UCSD, Winter 2020

Matrix Review

Coordinate Systems

- Right handed coordinate system



3D Models

- Let's say we have a 3D model that has an array of position vectors describing its shape
- We will group all of the position vectors used to store the data in the model into a single array: $\mathbf{v}_n$ where $0 \leq n \leq$ NumVerts-1
- Each vector $\mathbf{v}_n$ has components v_{nx} v_{ny} v_{nz}

Translation

- Let's say that we want to move our 3D model from its current location to somewhere else…
- In technical jargon, we call this a *translation*
- We want to compute a new array of positions $\mathbf{v}'_n$ representing the new location
- Let's say that vector **d** represents the relative offset that we want to move our object by
- We can simply use: $\mathbf{v}'_n = \mathbf{v}_n + \mathbf{d}$

  to get the new array of positions

Transformations

$$\mathbf{v}'_n = \mathbf{v}_n + \mathbf{d}$$

- This translation represents a very simple example of an object *transformation*
- The result is that the entire object gets moved or *translated* by **d**
- From now on, we will drop the $_n$ subscript, and just write

$$\mathbf{v}' = \mathbf{v} + \mathbf{d}$$

remembering that in practice, this is actually a loop over several *different* $\mathbf{v}_n$ vectors applying the *same* vector **d** every time

Transformations

$$\mathbf{v}' = \mathbf{v} + \mathbf{d}$$

- Always remember that this compact equation can be expanded out into

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

- Or into a system of linear equations:

$$v'_x = v_x + d_x$$

$$v'_y = v_y + d_y$$

$$v'_z = v_z + d_z$$

Rotation

- Now, let's rotate the object in the *xy* plane by an angle θ, as if we were spinning it around the *z* axis

$$v'_x = \cos(\theta)v_x - \sin(\theta)v_y$$
$$v'_y = \sin(\theta)v_x + \cos(\theta)v_y$$
$$v'_z = v_z$$

- Note: a *positive* rotation will rotate the object *counterclockwise* when the rotation axis (z) is pointing *towards* the observer

Rotation

$$v'_x = \cos(\theta)v_x - \sin(\theta)v_y$$

$$v'_y = \sin(\theta)v_x + \cos(\theta)v_y$$

$$v'_z = v_z$$

- We can expand this to:

$$v'_x = \cos(\theta)v_x - \sin(\theta)v_y + 0v_z$$

$$v'_y = \sin(\theta)v_x + \cos(\theta)v_y + 0v_z$$

$$v'_z = 0v_x + 0v_y + 1v_z$$

- And rewrite it as a matrix equation:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- Or just:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

Rotation

- We can represent a z-axis rotation transformation in matrix form as:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

or more compactly as:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

where

$$\mathbf{M} = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

- We can also define rotation matrices for the *x*, *y*, and *z* axes:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Linear Transformations

- Like translation, rotation is an example of a *linear transformation*
- True, the rotation contains *nonlinear* functions like sin()'s and cos()'s, but those ultimately just end up as constants in the actual linear equation
- We can generalize our matrix in the previous example to be:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v} \qquad \mathbf{M} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$

Linear Equation

- A general linear equation of 1 variable is:

$$f(v) = av + d$$

where *a* and *d* are constants

- A general linear equation of 3 variables is:

$$f(v_x, v_y, v_z) = f(\mathbf{v}) = av_x + bv_y + cv_z + d$$

- Note: there are no *nonlinear* terms like $v_x v_y$, v_x^2, $\sin(v_x)$…

System of Linear Equations

- Now let's look at 3 linear equations of 3 variables v_x, v_y, and v_z

$$v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1$$

$$v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2$$

$$v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3$$

- Note that all of the a_n, b_n, c_n, and d_n are constants (12 in total)

Matrix Notation

$$v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1$$

$$v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2$$

$$v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3$$

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v} + \mathbf{d}$$

Translation

- Let's look at our translation transformation again:

$$\mathbf{v}' = \mathbf{v} + \mathbf{d}$$

$$v'_x = v_x + d_x$$
$$v'_y = v_y + d_y$$
$$v'_z = v_z + d_z$$

- If we really wanted to, we could rewrite our three translation equations as:

$$v'_x = 1v_x + 0v_y + 0v_z + d_x$$
$$v'_y = 0v_x + 1v_y + 0v_z + d_y$$
$$v'_z = 0v_x + 0v_y + 1v_z + d_z$$

Identity

- We can see that this is equal to a transformation by the identity matrix

$$v'_x = 1v_x + 0v_y + 0v_z + d_1$$
$$v'_y = 0v_x + 1v_y + 0v_z + d_2$$
$$v'_z = 0v_x + 0v_y + 1v_z + d_3$$

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Identity

- Multiplication by the *identity matrix* does not affect the vector

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{v} = \mathbf{I} \cdot \mathbf{v}$$

Uniform Scaling

- We can apply a uniform scale to our object with the following transformation

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- If $s>1$, then the object will grow by a factor of s in each dimension
- If $0<s<1$, the object will shrink
- If $s<0$, the object will be *reflected* across all three dimensions, leading to an object that is 'inside out'

Non-Uniform Scaling

- We can also do a more general *nonuniform scale*, where each dimension has its own scale factor

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

which leads to the equations:

$$v'_x = s_x v_x$$
$$v'_y = s_y v_y$$
$$v'_z = s_z v_z$$

Reflections

- A reflection is a special type of scale operation where one of the axes is negated causing the object to reflect across a plane
- For example, a reflection along the x-axis would look like:

$$\begin{bmatrix} v_x{}' \\ v_y{}' \\ v_z{}' \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- Given an arbitrary 3x3 matrix **M**, we can tell if it has been reflected by checking if the matrix determinant is negative

Shears

- A *shear* is a translation along one axis by an amount proportional to the value along a different axis
- It causes a deformation similar to writing with *italics* (i.e., it causes a rectangle to deform into a parallelogram)
- For example a shear along x proportional to the value of y would look like:

$$\begin{bmatrix} v_x{}' \\ v_y{}' \\ v_z{}' \end{bmatrix} = \begin{bmatrix} 1 & h_{xy} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

Multiple Transformations

- If we have a vector **v**, and an x-axis rotation matrix $\mathbf{R}_x$, we can generate a rotated vector **v′**:

$$\mathbf{v}' = \mathbf{R}_x(\theta) \cdot \mathbf{v}$$

- If we wanted to then rotate that vector around the y-axis, we could simply:

$$\mathbf{v}'' = \mathbf{R}_y(\phi) \cdot \mathbf{v}'$$

$$\mathbf{v}'' = \mathbf{R}_y(\phi) \cdot (\mathbf{R}_x(\theta) \cdot \mathbf{v})$$

Multiple Transformations

- We can extend this to the concept of applying any sequence of transformations:

$$\mathbf{v}' = \mathbf{M}_4 \cdot \left(\mathbf{M}_3 \cdot \left(\mathbf{M}_2 \cdot \left(\mathbf{M}_1 \cdot \mathbf{v}\right)\right)\right)$$

- Because matrix algebra obeys the *associative* law, we can regroup this as:

$$\mathbf{v}' = \left(\mathbf{M}_4 \cdot \mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1\right) \cdot \mathbf{v}$$

- This allows us to *concatenate* them into a single matrix:

$$\mathbf{M}_{total} = \mathbf{M}_4 \cdot \mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1$$

$$\mathbf{v}' = \mathbf{M}_{total} \cdot \mathbf{v}$$

- Note: matrices do NOT obey the *commutative* law, so the order of multiplications is important

Matrix Dot Matrix

$$\mathbf{L} = \mathbf{M} \cdot \mathbf{N}$$

$$\begin{bmatrix} l_{11} & l_{12} & l_{13} \\ l_{21} & l_{22} & l_{23} \\ l_{31} & l_{32} & l_{33} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \cdot \begin{bmatrix} n_{11} & n_{12} & n_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{bmatrix}$$

$$l_{12} = m_{11}n_{12} + m_{12}n_{22} + m_{13}n_{32}$$

Multiple Rotations & Scales

- We can combine a sequence of rotations and scales into a single matrix
- For example, we can combine a y-rotation, followed by a z-rotation, then a non-uniform scale, and finally an x-rotation:

$$\mathbf{M} = \mathbf{R}_x(\gamma) \cdot \mathbf{S}(\mathbf{s}) \cdot \mathbf{R}_z(\beta) \cdot \mathbf{R}_y(\alpha)$$

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

Multiple Translations

- We can also take advantage of the associative property of vector addition to combine a sequence of translations
- For example, a translation along vector $\mathbf{t}_1$ followed by a translation along $\mathbf{t}_2$ and finally $\mathbf{t}_3$ can be combined:

$$\mathbf{d} = \mathbf{t}_1 + \mathbf{t}_2 + \mathbf{t}_3$$

$$\mathbf{v}' = \mathbf{v} + \mathbf{d}$$

Combining Transformations

- We see that we can combine a sequence of rotations and/or scales
- We can also combine a sequence of translations
- But what if we want to combine translations *with* rotations/scales?

Homogeneous Transformations

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

$$v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1$$

$$v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2$$

$$v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3$$

$$1 = 0 v_x + 0 v_y + 0 v_z + 1$$

Homogeneous Transformations

- So we've basically taken the 3x3 rotation/scale matrix and the 3x1 translation vector from our original equation and combined them into a new 4x4 matrix (for today, we will always have [0 0 0 1] in the bottom row of the matrix)
- We also replace our 3D position vector **v** with its 4D version [v_x v_y v_z 1]
- Using 4x4 transformation matrices allows us to combine rotations, translations, scales, shears, and reflections into a single matrix
- For rendering, we use the bottom row as well to perform perspective projections, but for animation, we are mainly concerned with placing objects in 3D space, not rendering them into a 2D image, so we will almost always have [0 0 0 1] on the bottom row

Homogeneous Transformations

- For example, a translation by vector **r** followed by a *z*-axis rotation can be represented as:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & r_x \\ 0 & 1 & 0 & r_y \\ 0 & 0 & 1 & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

Rotation Matrices

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translations

- A 4x4 translation matrix that translates an object by the vector **r** is:

$$\mathbf{T}(\mathbf{r}) = \begin{bmatrix} 1 & 0 & 0 & r_x \\ 0 & 1 & 0 & r_y \\ 0 & 0 & 1 & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pivot Points

- The standard rotation matrices pivot the object about an axis through the origin
- What if we want the pivot point to be somewhere else?
- The following transformation performs a z-axis rotation pivoted around the point **r**

$$\mathbf{M} = \mathbf{T}(\mathbf{r}) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{T}(-\mathbf{r})$$

General 4x4 Matrix

- All of the matrices we've see so far have [0 0 0 1] in the bottom row
- The product formed by multiplying any two matrices of this form will also have [0 0 0 1] in the bottom row
- We can say that this set of matrices forms a multiplicative group of 3D linear transformations
- We can construct any matrix in this group by multiplying a sequence of basic rotations, translations, scales, and shears

$$\begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General 4x4 Matrix

- Assuming we have [0 0 0 1] in the bottom row, we see that there are 12 different numbers in the upper 3x4 portion of the 4x4 matrix
- There are also 12 degrees of freedom for an object undergoing a linear transformation in 3D space
- 3 of those are represented by the three translational axes
- 3 of them are for rotation in the 3 planes (xy, yz, xz)
- 3 of them are scales along the 3 main axes
- and the last 3 are shears in the 3 main planes (xy, yz, xz)

- The 3 numbers for translation are easily decoded (d_x, d_y, d_z)
- The other 9 numbers, however, are encoded into the 9 numbers in the upper 3x3 portion of the matrix

Affine Transformations

- All of the transformations we've seen so far are examples of *affine* transformations
- If we have a pair of parallel lines and transform them with an affine transformation, they will remain parallel
- Affine transformations are fast to compute and very useful throughout computer graphics

Position vs. Direction Vectors

- We will almost always treat vectors as having 3 coordinates (x, y, and z)
- However, when we actually transform them by a 4x4 matrix, we expand them to 4 coordinates
- Vectors representing a position in 3D space are expanded into 4D as:

$$\begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}$$

- Vectors representing direction (like a normal or an axis of rotation) are expanded as:

$$\begin{bmatrix} v_x & v_y & v_z & 0 \end{bmatrix}$$

Position Transformation

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

$$v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1$$

$$v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2$$

$$v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3$$

$$1 = 0 v_x + 0 v_y + 0 v_z + 1$$

Direction Transformation

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 0 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

$$v'_x = a_1 v_x + b_1 v_y + c_1 v_z$$

$$v'_y = a_2 v_x + b_2 v_y + c_2 v_z$$

$$v'_z = a_3 v_x + b_3 v_y + c_3 v_z$$

$$0 = 0 v_x + 0 v_y + 0 v_z + 0$$

Object Space

- The space that an object is defined in is called *object space* or *local space*
- Usually, the object is located at or near the origin and is aligned with the xyz axes in some reasonable way
- The units in this space can be whatever we choose (i.e., meters, etc.)
- A 3D object would be stored on disk and in memory in this coordinate system
- When we go to draw the object, we will want to transform it into a different space

World Space

- We will define a new space called *world space* or *global space*
- This space represents a 3D world or scene and may contain several objects placed in various locations
- Every object in the world needs a matrix that transforms its vertices from its own object space into this world space
- We will call this the object's *world matrix*, or often, we will just call it the object's matrix
- For example, if we have 100 chairs in the room, we only need to store the object space data for the chair once, and we can use 100 different matrices to transform the chair model into 100 locations in the world

ABCD Vectors

- We mentioned that the translation information is easily extracted directly from the matrix, while the rotation information is encoded into the upper 3x3 portion of the matrix
- Is there a geometric way to understand these 9 numbers?

$$\begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ABCD Vectors

- In fact there is! The 9 constants make up 3 vectors called **a**, **b**, and **c**. If we think of the matrix as a transformation from object space to world space, then the **a** vector is essentially the object's x-axis rotated into world space, **b** is its y-axis in world space, and **c** is its z-axis in world space. **d** is of course the position in world space.

$$\begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rigid Matrices

- If the **a**, **b**, and **c** vectors are all unit length and perpendicular to each other, we say that the upper 3x3 matrix is *orthonormal*
- If the upper 3x3 portion is orthonormal, we say that the 4x4 matrix is *rigid*, meaning that it describes an object that is only translated and rotated (in other words, it will not have any scale or shears which distort the object)
- A rigid object in 3D space has 6 degrees of freedom (DOFs). As we saw earlier, a general 3D transformation has 12 variables. 6 of them represent the rigid DOFs (rotation & translation) and the other 6 represent the linear *deformations* (scales & shears)

Rigid Matrices

- If a 4x4 matrix represents a rigid transformation, then the upper 3x3 portion will be orthonormal

$$|\mathbf{a}| = |\mathbf{b}| = |\mathbf{c}| = 1$$

$$\mathbf{a} = \mathbf{b} \times \mathbf{c}$$

$$\mathbf{b} = \mathbf{c} \times \mathbf{a}$$

$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

Character Skeletons

Kinematics

- Kinematics: The analysis of motion independent of physical forces. Kinematics deals with position, velocity, acceleration, and their rotational counterparts, orientation, angular velocity, and angular acceleration.
- Forward Kinematics: The process of computing world space geometric data from local DOFs
- Inverse Kinematics: The process of computing a set of DOFs that causes some world space goal to be met (I.e., place the hand on the doorknob…)
- Note: Kinematics is an entire branch of mathematics and there are several other aspects of kinematics that don't fall into the 'forward' or 'inverse' description

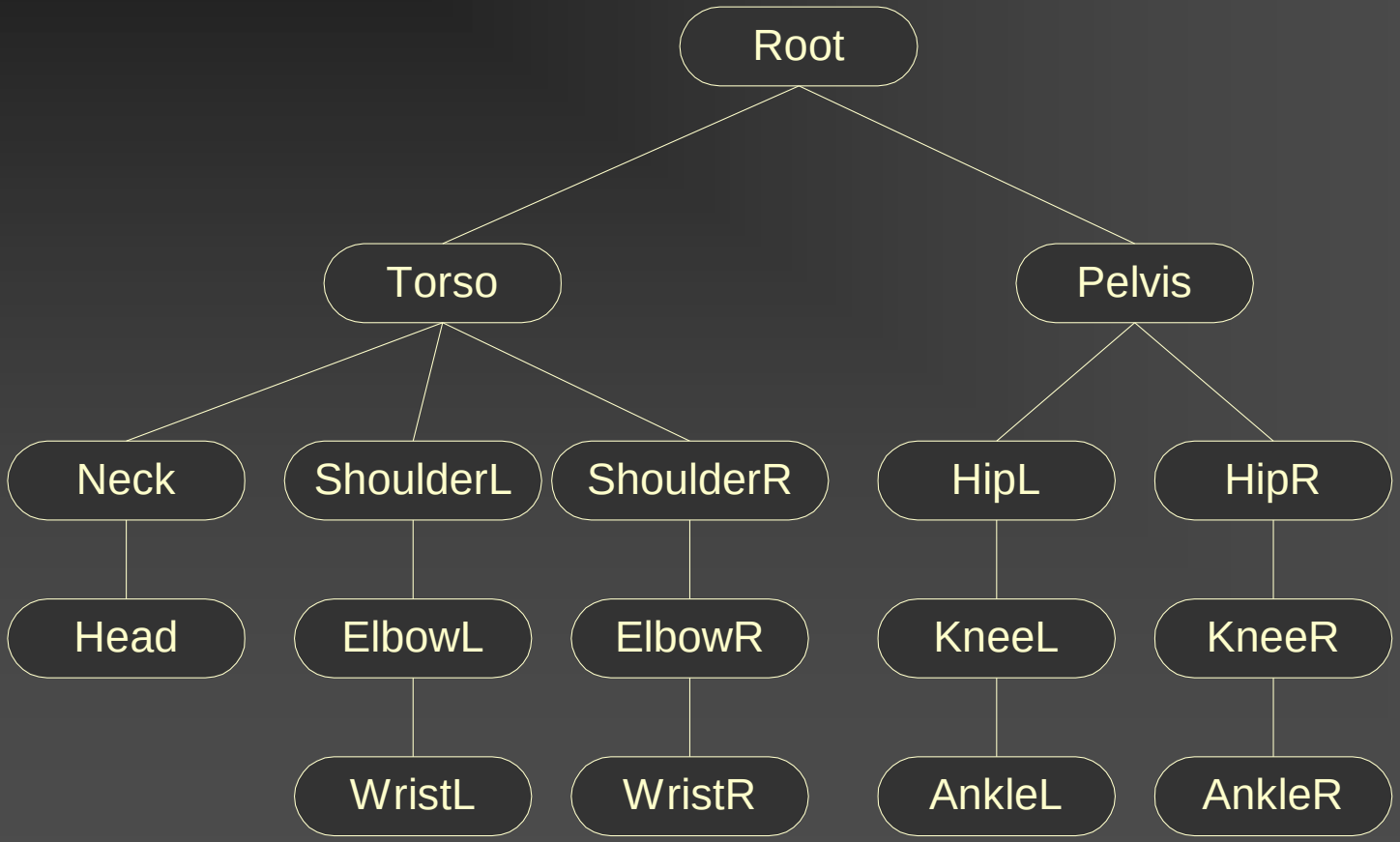
Skeletons

- Skeleton: A pose-able framework of joints arranged in a tree structure. The skeleton is used as an invisible armature to manipulate the skin and other geometric data of the character
- Joint: A joint allows relative movement within the skeleton. Joints are essentially 4x4 matrix transformations. Joints can be rotational, translational, or some non-realistic types as well
- Bone: Bone is really just a synonym for joint for the most part. For example, one might refer to the shoulder joint or upper arm bone (humerus) and mean the same thing

DOFs

- Degree of Freedom (DOF): A variable φ describing a particular axis or dimension of movement within a joint
- Joints typically have around 1-3 DOFs ($\varphi_1 \ldots \varphi_N$)
- Changing the DOF values over time results in the animation of the skeleton
- In later weeks, we will extend the concept of a DOF to be any animate-able parameter within the character rig
- Note: in a mathematical sense, a free rigid body has 6 DOFs: 3 for position and 3 for rotation

Example Joint Hierarchy



Joints

- Core Joint Data
  - DOFs (N floats)
  - Local matrix: **L**
  - World matrix: **W**
- Additional Data
  - Joint offset vector: **r**
  - DOF limits (min & max value per DOF)
  - Type-specific data (rotation/translation axes, constants…)
  - Tree data (pointers to children, siblings, parent…)

Skeleton Posing Process

1. Specify all DOF values for the skeleton (done by higher level animation system)

2. Recursively traverse through the hierarchy starting at the root and use forward kinematics to compute the world matrices (done by skeleton system)

3. Use world matrices to deform skin & render (done by skin system)

Note: the matrices can also be used for other things such as collision detection, FX, etc.

Forward Kinematics

- In the recursive tree traversal, each joint first computes its local matrix **L** based on the values of its DOFs and some formula representative of the joint type:

  Local matrix $\mathbf{L} = \mathbf{L}_{joint}(\varphi_1, \varphi_2, \ldots, \varphi_N)$

- Then, world matrix **W** is computed for each joint by concatenating its local matrix **L** with the *world matrix of the parent joint*

  World matrix $\mathbf{W} = \mathbf{W}_{parent} \cdot \mathbf{L}$

Joint Offsets

- It is convenient to have a 3D offset vector **r** for every joint which represents its pivot point relative to its parent's matrix

$$\mathbf{L}_{offset} = \begin{bmatrix} 1 & 0 & 0 & r_x \\ 0 & 1 & 0 & r_y \\ 0 & 0 & 1 & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

DOF Limits

- It is nice to be able to limit a DOF to some range (for example, the elbow could be limited from 0º to 150º)
- Usually, in a realistic character, all DOFs will be limited except the ones controlling the root

Skeleton Rigging

- Setting up the skeleton is an important and early part of the rigging process
- Sometimes, character skeletons are built before the skin, while other times, it is the opposite
- To set up a skeleton, an artist uses an interactive tool to:
  - Construct the tree
  - Place joint offsets
  - Configure joint types
  - Specify joint limits
  - Possibly more…

Poses

- Once the skeleton is set up, one can then adjust each of the DOFs to specify the pose of the skeleton
- We can define a pose Φ more formally as a vector of N numbers that maps to a set of DOFs in the skeleton

$$\Phi = [\varphi_1 \; \varphi_2 \; \ldots \; \varphi_N]$$

- A pose is a convenient unit that can be manipulated by a higher level animation system and then handed down to the skeleton
- Usually, each joint will have around 1-3 DOFs, but an entire character might have 100+ DOFs in the skeleton
- Keep in mind that DOFs can be also used for things other than joints, as we will learn later…

Joint Types

Joint Types

- Rotational
  - Hinge: 1-DOF
  - Universal: 2-DOF
  - Ball & Socket: 3-DOF
    - Euler Angles
    - Quaternions
- Translational
  - Prismatic: 1-DOF
  - Translational: 3-DOF (or any number)
- Compound
  - Free
  - Screw
  - Constraint
  - Etc.
- Non-Rigid
  - Scale
  - Shear
  - Etc.
- Design your own...

Hinge Joints (1-DOF Rotational)

- Rotation around the x-axis:

$$\mathbf{L}_{Rx}(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & r_x \\ 0 & \cos\theta_x & -\sin\theta_x & r_y \\ 0 & \sin\theta_x & \cos\theta_x & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hinge Joints (1-DOF Rotational)

- Rotation around the y-axis:

$$\mathbf{L}_{Ry}(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & r_x \\ 0 & 1 & 0 & r_y \\ -\sin\theta_y & 0 & \cos\theta_y & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hinge Joints (1-DOF Rotational)

- Rotation around the z-axis:

$$\mathbf{L}_{Rz}(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & r_x \\ \sin\theta_z & \cos\theta_z & 0 & r_y \\ 0 & 0 & 1 & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hinge Joints (1-DOF Rotational)

- Rotation around an arbitrary unit axis **a**:

$$\mathbf{L}_{Ra}(\theta) =$$

$$\begin{bmatrix} a_x^2 + c_\theta(1-a_x^2) & a_x a_y(1-c_\theta) - a_z s_\theta & a_x a_z(1-c_\theta) + a_y s_\theta & r_x \\ a_x a_y(1-c_\theta) + a_z s_\theta & a_y^2 + c_\theta(1-a_y^2) & a_y a_z(1-c_\theta) - a_x s_\theta & r_y \\ a_x a_z(1-c_\theta) - a_y s_\theta & a_y a_z(1-c_\theta) + a_x s_\theta & a_z^2 + c_\theta(1-a_z^2) & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Universal Joints (2-DOF)

- For a 2-DOF joint that first rotates around x and then around y:

$$\mathbf{L}_{Rxy}(\theta_x, \theta_y) = \begin{bmatrix} c_y & s_x s_y & c_x s_y & r_x \\ 0 & c_x & -s_x & r_y \\ -s_y & s_x c_y & c_x c_y & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Different matrices can be formed for different axis combinations

Ball & Socket (3-DOF)

- For a 3-DOF joint that first rotates around x, y, then z:

$$\mathbf{L}_{Rxyz}(\theta_x, \theta_y, \theta_z) = \begin{bmatrix} c_y c_z & s_x s_y c_z - c_x s_z & c_x s_y c_z + s_x s_z & r_x \\ c_y s_z & s_x s_y s_z + c_x c_z & c_x s_y s_z - s_x c_z & r_y \\ -s_y & s_x c_y & c_x c_y & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Different matrices can be formed for different axis combinations

Quaternions

$$\mathbf{q} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}$$

$$|\mathbf{q}| = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2} = 1$$

$$\mathbf{q} = \begin{bmatrix} a_x \sin\frac{\theta}{2} & a_y \sin\frac{\theta}{2} & a_z \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

$$\mathbf{L}_Q(\mathbf{q}) = \begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_wq_z & 2q_xq_z+2q_wq_y & r_x \\ 2q_xq_y+2q_wq_z & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_wq_x & r_y \\ 2q_xq_z-2q_wq_y & 2q_yq_z+2q_wq_x & 1-2q_x^2-2q_y^2 & r_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Prismatic Joints (1-DOF Translation)

- 1-DOF translation along an arbitrary axis **a**:

$$\mathbf{L}_{Ta}(t) = \begin{bmatrix} 1 & 0 & 0 & r_x + t \cdot a_x \\ 0 & 1 & 0 & r_y + t \cdot a_y \\ 0 & 0 & 1 & r_z + t \cdot a_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translational Joints (3-DOF)

- For a more general 3-DOF translation:

$$\mathbf{L}_{Txyz}(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & r_x + t_x \\ 0 & 1 & 0 & r_y + t_y \\ 0 & 0 & 1 & r_z + t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Other Joints

- Compound
  - Free
  - Screw
  - Constraint
  - Etc.
- Non-Rigid
  - Scale (1 axis, 3 axis, volume preserving…)
  - Shear
  - Etc.

Programming Project #1: Skeleton

Project 1 Assignment

- Load a skeleton from a ‘.skel’ file and display it in 3D
- All joints in the skeleton should be 3-DOF rotational joints
- Due: Thursday, January 16, by 4:50pm

Software Architecture

- Object oriented
- Make objects for things that should be objects
- Avoid global data & functions
- Encapsulate information
- Provide useful interfaces
- Put different objects in different files

Sample Code

- Some sample code is provided on the course web page listed as ‘project0’
- It is an object oriented demo of a spinning cube
- Classes:
  - Shader
  - Model
  - Camera
  - SpinningCube
  - Tokenizer
  - Tester

Sample Skel File

```
balljoint root {
    [data for root]
    balljoint head {
        [data for head]
        [children of head]
    }
    balljoint leg_l {
        [data for leg]
        [children of leg]
    }
    [more children of root]
}
```

Skel File Data Tokens

offset	x y z	(joint offset vector)
boxmin	x y z	(min corner of box to draw)
boxmax	x y z	(max corner of box to draw)
rotxlimit	min max	(x rotation DOF limits)
rotylimit	min max	(y rotation DOF limits)
rotzlimit	min max	(z rotation DOF limits)
pose	x y z	(values to pose DOFs)
balljoint	name { }	(child joint)

Possible Object Breakdown

- One should consider making objects (classes) for the following:
  - DOF
  - Joint
  - Skeleton

Common Routines

- Many classes will need functions for some or all of the following:
  - Constructor / destructor
  - Initialize
  - Load
  - Update (move things, pose, animate…)
  - Draw
  - Reset

What is a DOF?

- Data
  - Value
  - Min, max
- Functions
  - SetValue() (can clamp value at the time of setting)
  - GetValue()
  - SetMinMax()…

What is a Joint?

- Data
  - Local & World matrices
  - Array of DOFs
  - Tree data (child/sibling/parent pointers, etc.)
- Functions
  - Update() (recursively generate local matrix & concatenate)
  - Load()
  - AddChild()
  - Draw()
- Note: One could also make a Joint base class and derive various specific joint types. In this case, it would be a good idea to make a virtual function for MakeLocalMatrix() that the base traversal routine calls

What is a Skeleton?

- Data
  - Joint tree (might only need a pointer to the root joint)
- Functions
  - Load
  - Update (traverses tree & computes joint matrices)
  - Draw

Tree Data Structures

- The skeleton requires only the most basic N-tree data structure
- The main thing the tree needs is an easy way to perform a depth-first traversal
- There are various ways to implement the tree, but I suggest that each Joint just stores a std::vector of pointers to its child joints

Update & Draw

```
void Joint::Update(Matrix &parent) {
    … // Compute LocalMatrix
    … // Compute WorldMatrix
    … // Recursively call Update() on children
}

void Joint::Draw() {
   .. // Do some OpenGL
   .. // Recursively call Draw() on children
}
```

Load

```
bool Skeleton::Load(const char *file) {
    Tokenizer token;
    token.Open(file,"skel"));
    token.FindToken("balljoint"));

    // Parse tree
    Root=new Joint;
    Root->Load(token);

    // Finish
    token.Close();
    return true;
}
```

```
bool Joint::Load(Tokenizer &t) {
    token.FindToken("{"));
    while(1) {
        char temp[256];
        token.GetToken(temp);
        if(strcmp(temp,"offset")==0) {
            Offset.x=token.GetFloat();
            Offset.y=token.GetFloat();
            Offset.z=token.GetFloat();
        }
        else // Check for other tokens
        else if(strcmp(temp,"balljoint")==0) {
            Joint *jnt=new Joint;
            jnt->Load(token);
            AddChild(*jnt);
        }
        else if(strcmp(temp,"}")==0) return true;
        else token.SkipLine();    // Unrecognized token
    }
}
```