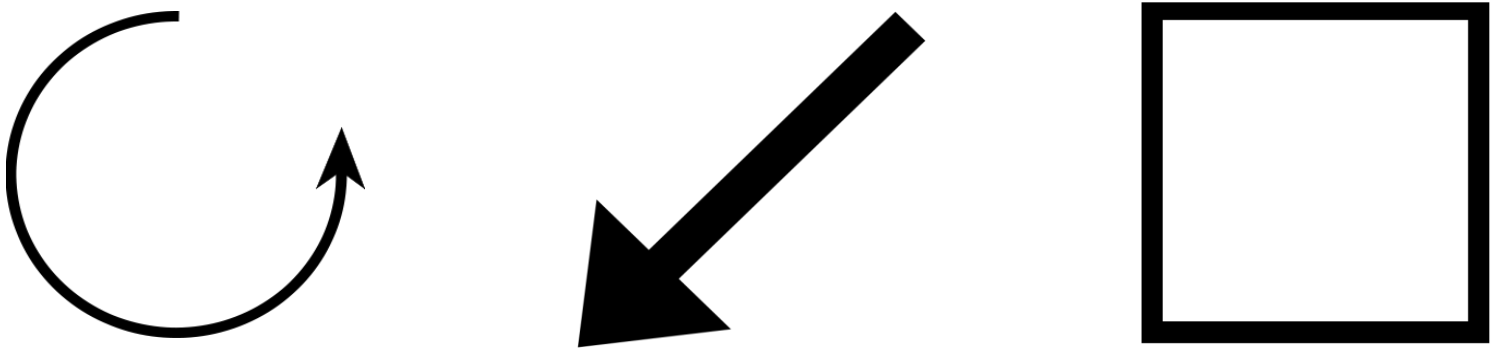
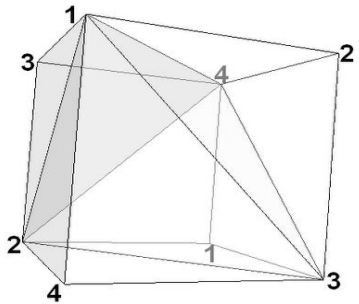


Geometric Transformations

2D and 3D

How do we use Geometric Transformations? (1/2)

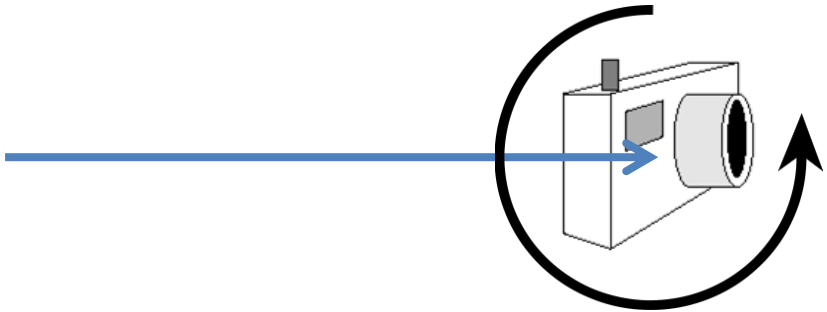
- Objects in a scene at the lowest level are a collection of vertices…



- These objects have location, orientation, size
- Correspond to transformations: Translation (***T***), Rotation (***R***), and Scaling (***S***)

How do we use Geometric Transformations? (2/2)

- A scene has a camera/view point from which the scene is viewed
- The camera has some location and some orientation in 3-space …



- These correspond to Translation and Rotation transformations
- Need other types of viewing transformations as well - learn about them shortly

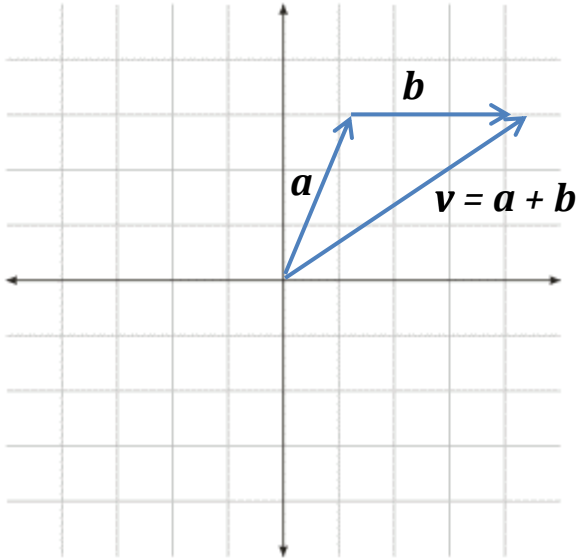
Some Linear Algebra Concepts...

- 3D coordinate geometry
- Vectors in 2 space and 3 space
- Dot product and cross product – definitions and uses
- Vector and matrix notation and algebra
- Identity matrix
- Multiplicative associativity
  - E.g. $A(BC) = (AB)C$
- Matrix transpose and inverse – definition, use, and calculation
- Homogeneous coordinates $(x, y, z, \boldsymbol{w})$

You will need to understand these concepts!

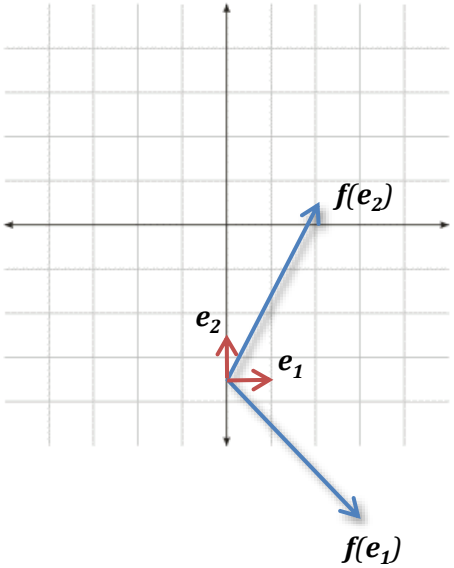
Linear Transformations (1/3)

- We represent vectors as ***bold-italic*** letters (***v***) and scalars as *italic* letters (*c*)
- Recall that a *basis* for a vector space is a set of vectors with the following properties:
  1. The vectors in the set are linearly independent
  2. Any vector in the vector space can be expressed as a linear combination of the basis vectors: $\mathbf{V} = c_1\mathbf{V}_1 + c_2\mathbf{V}_2$
- Multiplying a vector by a scalar changes the vector's magnitude



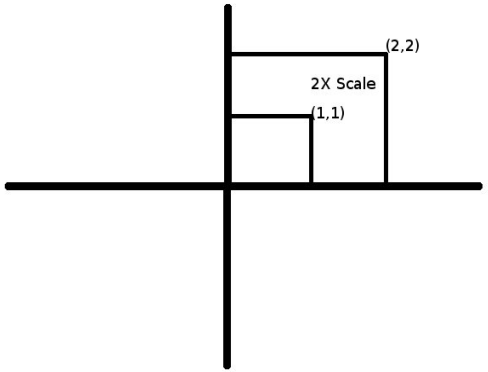
Linear Transformations (2/3)

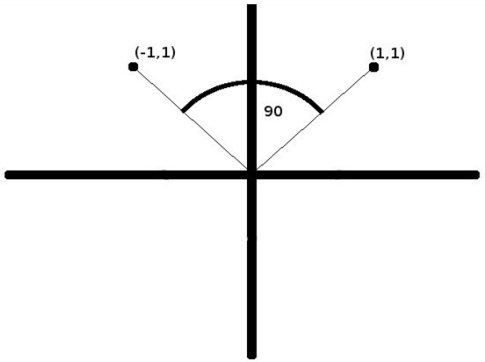
- Definition of a *linear function* ***f***:
  - $f(v+w) = f(v) + f(w)$ for all v and w in the domain of f
  - $f(cv) = cf(v)$ for all scalars c and elements v in the domain
- Both properties must be satisfied for the function ***f*** to be linear
- Example: $f(x) = f(x_1, x_2) := (3x_1+2x_2, -3x_1+4x_2)$
- Now let v and w be two elements in the domain of f
  - $f(v+w) = f(v_1+w_1, v_2+w_2)$
    $= (3(v_1+w_1)+2(v_2+w_2), -3(v_1+w_1)+4(v_2+w_2))$
    $= (3v_1+2v_2, -3v_1+4v_2) + (3w_1+2w_2, -3w_1+4w_2)$
    $= f(v) + f(w)$
  - We can check the second property the same way
- Properties:
  - Leaves origin invariant
  - Maps parallelograms to (possibly distorted) parallelograms
  - If M is invertible, there is a sequence of rotations, scales and shears that performs the mapping



Linear Transformations (3/3)

- Graphical use: transformations of points around the origin (**leaves the origin invariant**)
  - These include *Scaling* and *Rotations*
  - *Translation* is not a linear function (moves the origin)
  - Any linear transformation of a point will result in another point in the same coordinate system, transformed about the origin
- Aside: How do we know the origin is invariant from the definition of linearity?





Linear Transformations as Matrices (1/2)

- Linear transformations can be represented as invertible (non-singular) matrices
- Let's start with 2D transformations. These can be represented by 2x2 matrices:

$$\boldsymbol{T} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

- A transformation of an arbitrary column vector $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ has the form:

$$T\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} ax_1 + bx_2 \\ cx_1 + dx_2 \end{bmatrix}$$

Linear Transformations as Matrices (2/2)

- Let $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ be the standard basis vectors: $\boldsymbol{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $\boldsymbol{e_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- Now substitute each basis vector for $\boldsymbol{x}$ to get:

$$T\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ c \end{bmatrix} \qquad T\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

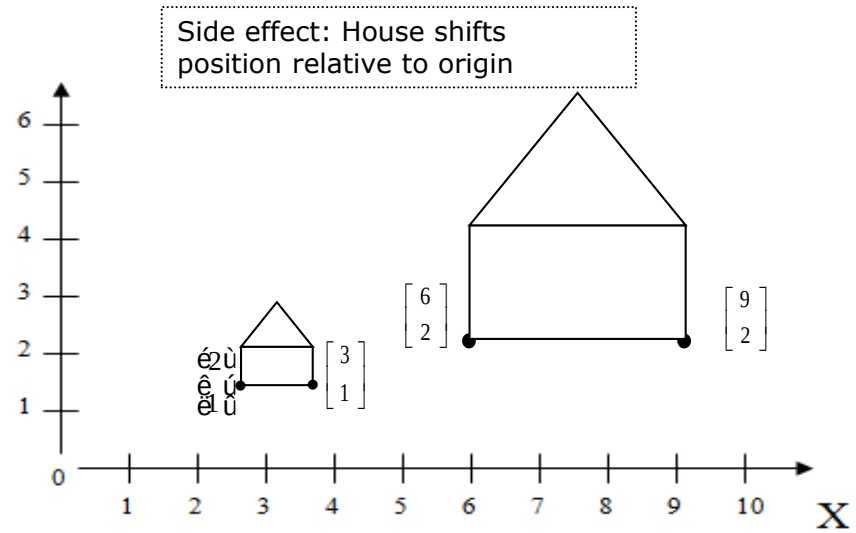
- Notice that the columns of the matrix representation of our transformation matrix $\boldsymbol{T}$ are precisely $\boldsymbol{T}$ applied to $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$:

$$\boldsymbol{T}(\boldsymbol{e_1}) = \begin{bmatrix} a \\ c \end{bmatrix} \quad \boldsymbol{T}(\boldsymbol{e_2}) = \begin{bmatrix} b \\ d \end{bmatrix}$$

- **This gives us a strategy for deriving transformation matrices!**
- We can derive the columns of a transformation matrix one by one by considering how our desired transformation affects each of the standard unit vectors.

Scaling in 2D (1/2)

- Scale ***x*** by 3, ***y*** by 2 ($S_x = 3,\ S_y = 2$)
  - $\boldsymbol{v} = \begin{bmatrix} x \\ y \end{bmatrix}$ (original vertex); $\boldsymbol{v'} = \begin{bmatrix} x' \\ y' \end{bmatrix}$ (new vertex)
- $\boldsymbol{v' = Sv}$
- Derive $\boldsymbol{S}$ by determining how $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ should be transformed
  - $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow s_x * e_1 = \begin{bmatrix} s_x \\ 0 \end{bmatrix}$ (scale in X by S_x)
  - $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow s_y * e_2 = \begin{bmatrix} 0 \\ s_y \end{bmatrix}$ (scale in Y by S_y)
- Thus we obtain $S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$



Scaling in 2D (2/2)

- ***S*** is a diagonal matrix; we can quickly check using matrix multiplication that our derivation is correct:

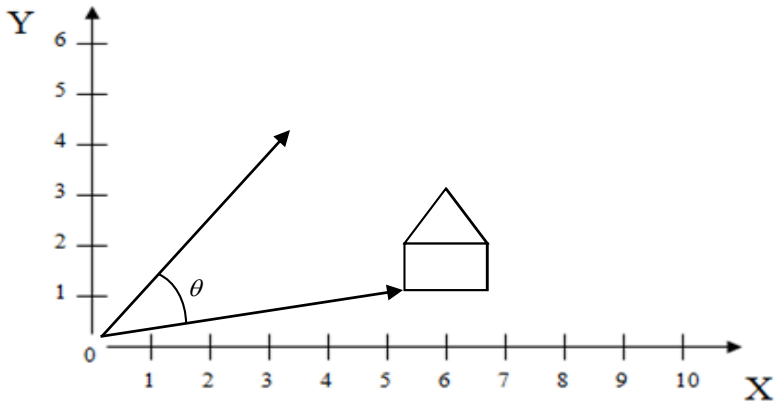
$$\boldsymbol{Sv} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

- ***S*** multiplies each coordinate of ***v*** by the appropriate scaling factor, as expected
- In general, the i^{th} entry of ***Dv***, where ***D*** is diagonal, is (***D[i,i]*** * ***v[i]***)

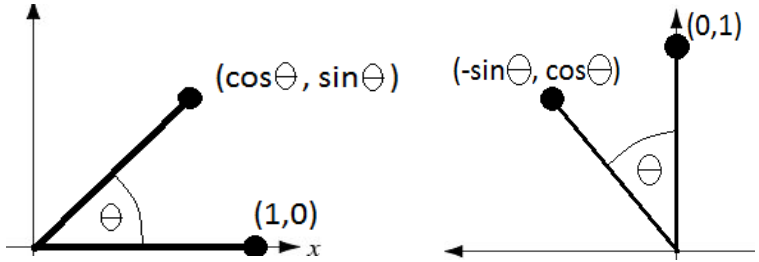
- Properties of scaling to look out for:
  - Does not preserve angles between lines in the plane (except when scaling is uniform, i.e. $\boldsymbol{s_x} = \boldsymbol{s_y}$)
  - If the object doesn't start at the origin, scaling will move it closer to or farther from the origin (often not desired)

Rotation in 2D (1/2)

- Rotate by θ about the origin
- $\boldsymbol{v}' = \boldsymbol{R}_{\theta}\boldsymbol{v}$, where
  - $\boldsymbol{v} = \begin{bmatrix} x \\ y \end{bmatrix}$ (original vertex)
  - $\boldsymbol{v}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$ (new vertex)



- Derive $\boldsymbol{R}_{\theta}$ by determining how $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ should be transformed:
  - $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} \cos q \\ \sin q \end{bmatrix}$ (first column of $\boldsymbol{R}_{\theta}$)
  - $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -\sin q \\ \cos q \end{bmatrix}$ (second column of $\boldsymbol{R}_{\theta}$)



$$\therefore R_q = \begin{bmatrix} \cos q & -\sin q \\ \sin q & \cos q \end{bmatrix}$$

Rotation in 2D (2/2)

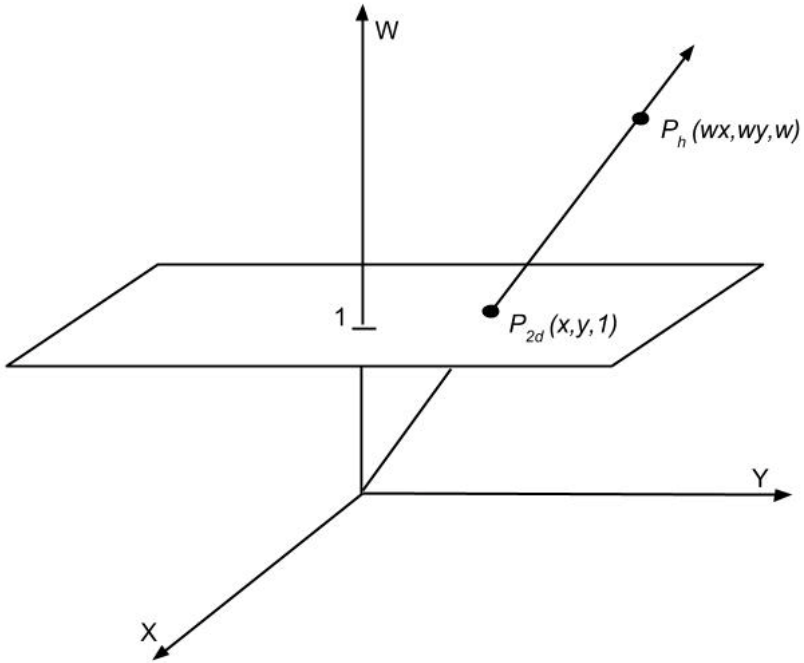
- Let's try matrix-vector multiplication
- $\boldsymbol{R_\theta v} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} = \boldsymbol{v'}$
  - $x' = x\cos q - y\sin q$
  - $y' = x\sin q + y\cos q$
- Other properties of rotation:
  - Preserves lengths in objects and angles between parts of objects (rigid-body rotation)
  - For objects not centered at the origin, an unwanted translation might be introduced (rotation is always about the origin)

What about translation?

- Translation is not a linear transformation (the origin is not invariant)
- Therefore, it can't be represented as a 2x2 invertible matrix
- **Question:** Is there another solution?
- **Answer:** Yes, $\boldsymbol{v}' = \boldsymbol{v} + \boldsymbol{t}$, where $\boldsymbol{t} = \begin{bmatrix} dx \\ dy \end{bmatrix}$
- However, using vector addition is not consistent with our method of treating transformations as matrices
- If we could treat all transformations in a consistent manner, i.e., with matrix representation, then could combine transformations by composing their matrices
- Let's try using a matrix again
- How? **Homogeneous Coordinates:** add an additional dimension, the *w*-axis, and an extra coordinate, the *w*-component
  - Thus 2D -> 3D (effectively the hyperspace for embedding 2D space)

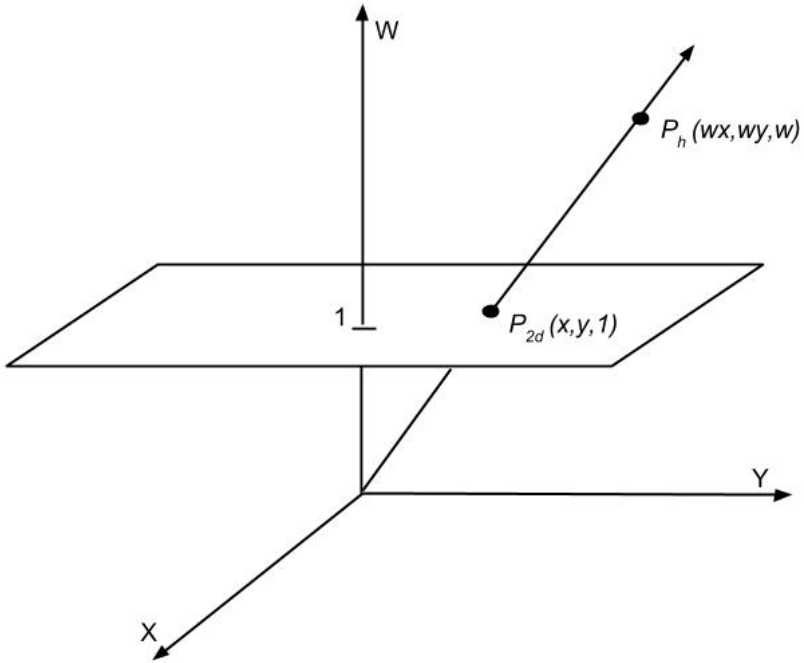
Homogeneous Coordinates (1/3)

- Allow expression of all three 2D transformations as 3x3 matrices
- We start with the point P_{2d} on the *xy* plane and apply a mapping to bring it to the *w*-plane in the hyperspace
  - $P_{2d}(x,y) \rightarrow P_h(wx, wy, w), w \neq 0$
- The resulting *(x',y')* coordinates in our new point P_h are different from the original *(x,y)*, since $x' = wx$, $y' = wy$
  - $P_h(x', y', w), w \neq 0$



Homogeneous Coordinates (2/3)

- Once we have this point, we can apply a homogenized version of our ***T***, ***R*** and ***S*** transformation matrices (next slides) to get a new point in the hyperspace
- Finally, we want to obtain the corresponding point in 2D-space, so perform the inverse of the previous mapping (divide all entries by *w*)
- The vertex $\boldsymbol{v} = \begin{bmatrix} x \\ y \end{bmatrix}$ is now represented as $\boldsymbol{v} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$



Homogeneous Coordinates (3/3)

- The transformations we use will always map **points** in the hyperplane defined by ***w = 1*** to other such points. (That way, we don't have to divide by *w* to get our equivalent point in 2D)
- In other words, we want our transformations ***T*** to map points $\boldsymbol{v} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ to points $\boldsymbol{v'} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$
- How do we achieve this with the matrices we have already derived?
- For linear transformations (i.e. scaling and rotation), embed the existing matrix in the upper-left of a new 3x3 matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Back to Translation

- Our translation matrix (***T***) can now be represented by embedding the translation vector in the right column:

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

- To verify that this is the right matrix, multiply it by our homogenized point:

$$\boldsymbol{Tv} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix} = \boldsymbol{v'}$$

- Coordinates have been translated, and ***v'*** is still homogeneous

Transformations Homogenized

- Let's homogenize our all matrices! Doesn't affect linearity of scaling and rotation
- Our new transformation matrices look like this…

Transformation	Matrix
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Rotation	$\begin{bmatrix} \cos q & -\sin q & 0 \\ \sin q & \cos q & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Translation	$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$

- Note: These transformations are called affine transformations, which means they preserve ratios of distances between points on a straight line

Examples

- Scaling: Scale by 15 in the *x* direction, 17 in the *y*

$$\begin{bmatrix} 15 & 0 & 0 \\ 0 & 17 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotation: Rotate by 123^o

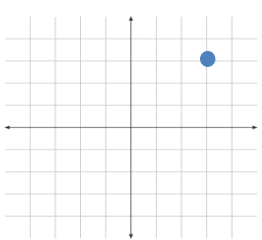
$$\begin{bmatrix} \cos(123) & -\sin(123) & 0 \\ \sin(123) & \cos(123) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

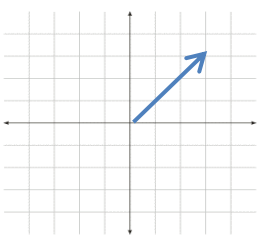
- Translation: Translate by -16 in the *x* direction, +18 in the *y*

$$\begin{bmatrix} 1 & 0 & -16 \\ 0 & 1 & 18 \\ 0 & 0 & 1 \end{bmatrix}$$

Before we continue! Vectors vs. Points

- Up until now, we've only used the notion of a point in our 2D space
- We now present a distinction between points and vectors





- We used **homogeneous coordinates** to more conveniently represent translation; hence **points** are represented as $(x, y, 1)^T$
- A **vector** can be rotated/scaled, but not translated (can think of it as always starting at origin), so don't use the homogeneous coordinate: $(x, y, 0)^T$
- That way, the translation matrix won't have any affect on our vectors.
- For now, let's focus on just our points (typically vertices)

Inverses

- When we want to undo a transformation, we'll need to find the matrix's inverse.
- Thanks to homogenization, they are all invertible!

Transformation	Matrix Inverse	Does it make sense?
Scaling	$\begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	If you scale something by factor a, the inverse is scaling by $1/a$
Rotation	$\begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	Inverse of rotation by θ is rotation by $-\theta$. The properties $sin(-\theta) = -sin(\theta)$ and $cos(-\theta) = cos(\theta)$ give this matrix. Also, the matrix is orthonormal, so inverse is just the transpose (see next slide).
Translation	$\begin{bmatrix} 1 & 0 & -dx \\ 0 & 1 & -dy \\ 0 & 0 & 1 \end{bmatrix}$	If you translate by $\boldsymbol{x}$, the inverse is translation by $-\boldsymbol{x}$

A moment of appreciation for linear algebra

- The inverse of a rotation matrix M is just its transpose M^T! That's really convenient, so let's understand how it works using orthonormal matrices
- Take a rotation matrix $M = [v_1 \quad v_2 \quad v_3]$ (where each v_i is a vector)
- First note some properties of M
  - The columns are orthogonal to each other: $v_i \bullet v_j = 0 \quad (i \neq j)$
  - Columns have unit length: $||v_i|| = 1$
- Let's see what multiplying M^T and M produces:

$$\begin{bmatrix} v_{1x} & v_{1y} & v_{1z} \\ v_{2x} & v_{2y} & v_{2z} \\ v_{3x} & v_{3y} & v_{3z} \end{bmatrix} \begin{bmatrix} v_{1x} & v_{2x} & v_{3x} \\ v_{1y} & v_{2y} & v_{3y} \\ v_{1z} & v_{2z} & v_{3z} \end{bmatrix} = \begin{bmatrix} v_1 \bullet v_1 & v_1 \bullet v_2 & v_1 \bullet v_3 \\ v_2 \bullet v_1 & v_2 \bullet v_2 & v_2 \bullet v_3 \\ v_3 \bullet v_1 & v_3 \bullet v_2 & v_3 \bullet v_3 \end{bmatrix}$$

- Using the properties above, we see that this is the identity matrix, so $M^T = M^{-1}$

More with Homogeneous Coordinates

Some uses we'll see later:

- Placing sub-objects in parent's coordinate system to construct hierarchical scene graph
  - Transforming primitives in their own coordinate systems
- View volume normalization
  - Mapping arbitrary view volume into canonical view volume along *z*-axis
- Parallel (orthographic, oblique) and perspective projections
- Perspective transformation (turn viewing pyramid into a cuboid to turn perspective projection into parallel projection) after perspective foreshortening

Composition of Transformations (2D) (1/2)

- We now have a number of tools at our disposal; we can combine them!
- An object in a scene uses many transformations in sequence. How do we represent this in terms of functions?
- Transformation is a function; by associativity, we can compose functions:
  - $(f \circ g)(i)$
- This is the same as first applying g, then applying f:
  - $f(g(i))$
- Consider our functions f and g as matrices (M_1 and M_2) and our input as a vector v
- Our composition is equivalent to $M_1 M_2 v$

Composition of Transformations (2D) (2/2)

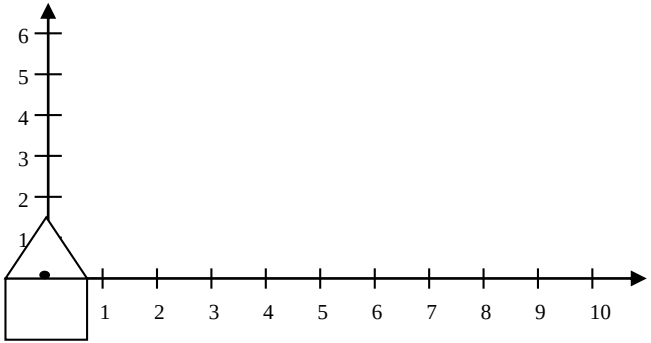
- We can now form compositions of transformation matrices to form a more complex transformation
- For example, ***TRSv***, which scales a point, then rotates it, then translates it:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

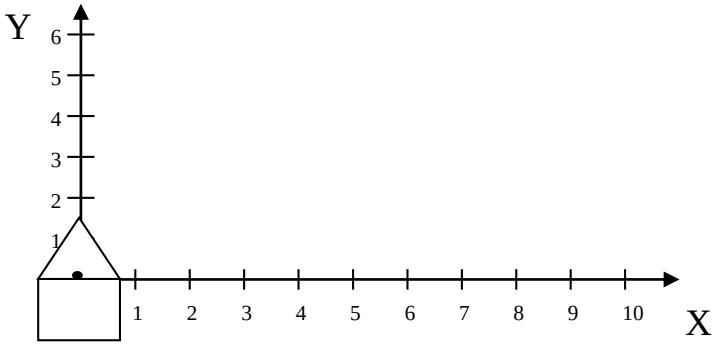
- Note that we apply the matrices in sequence right to left. We can use associativity to compose them first; it is often useful to be able to apply a single matrix if, for example, we want to use it to transform many points at once.
- **Important: order matters!** Matrix multiplication is **NOT commutative**.
- Be sure to check out the Transformation Game at http://www.cs.brown.edu/exploratories/freeSoftware/repository/edu/brown/cs/exploratories/applets/transformationGame/transformation_game_guide.html
- Let's see an example…

Not commutative

Translate by
$x = 6, y = 0$, then
rotate by 45°

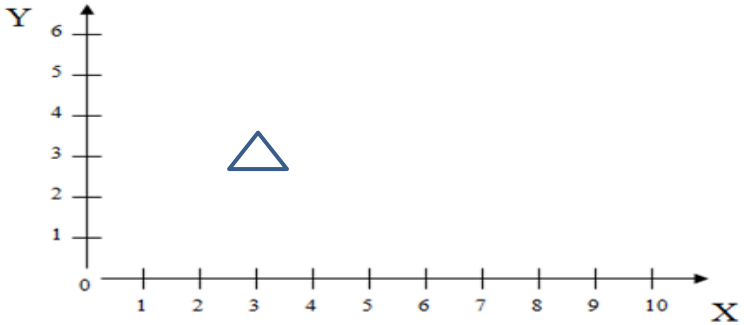


Rotate by 45°,
then translate
by $x = 6, y = 0$

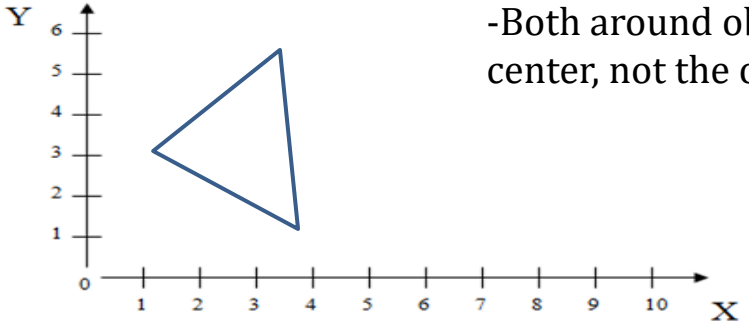


Composition (an example) (2D) (1/2)

- Start:



Goal:



-Rotate 90°
-Uniform scale 4x
-Both around object's center, not the origin

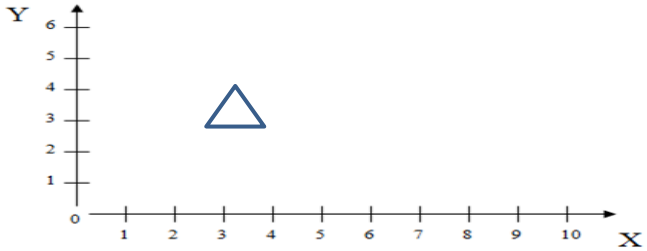
- Important concept: make the problem simpler
- Translate object to origin first, scale, rotate, and translate back:

$$T^{-1}RST = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90 & -\sin 90 & 0 \\ \sin 90 & \cos 90 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

- Apply to all vertices

Composition (an example) (2D) (2/2)

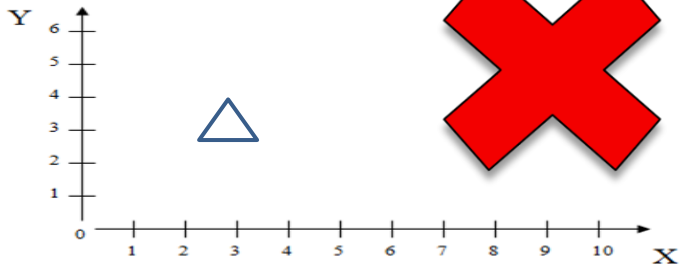
- $T^{-1}RST$



- But what if we mixed up the order? Let's try $RT^{-1}ST$:

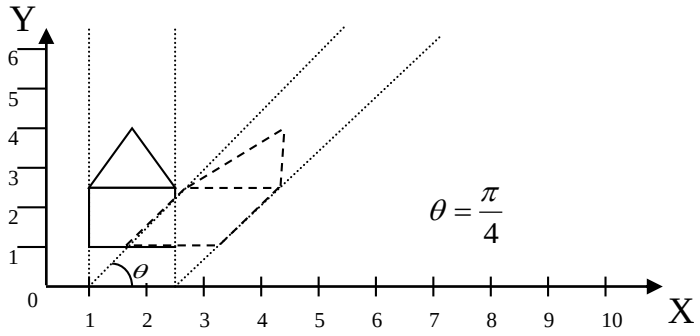
$$\begin{bmatrix} \cos 90 & -\sin 90 & 0 \\ \sin 90 & \cos 90 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

- Oops! We scaled properly, but when we rotated the object, it's center was not at the origin, so its position was shifted. Order matters!



Aside: Skewing/shearing

- "Skew" an object to the side, like shearing a card deck by displacing each card relative to the previous one
  - What physical situations mirror this behavior?
- Squares become parallelograms; *x*-coordinates skew to right, *y* stays the same
- Notice that the base of the house (at $y = 1$) remains horizontal but shifts right. Why?



$$Skew_\theta = \begin{bmatrix} 1 & \frac{1}{\tan\theta} \\ 0 & 1 \end{bmatrix}$$

2D non-homogeneous

$$Skew_\theta = \begin{bmatrix} 1 & \frac{1}{\tan\theta} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2D homogeneous

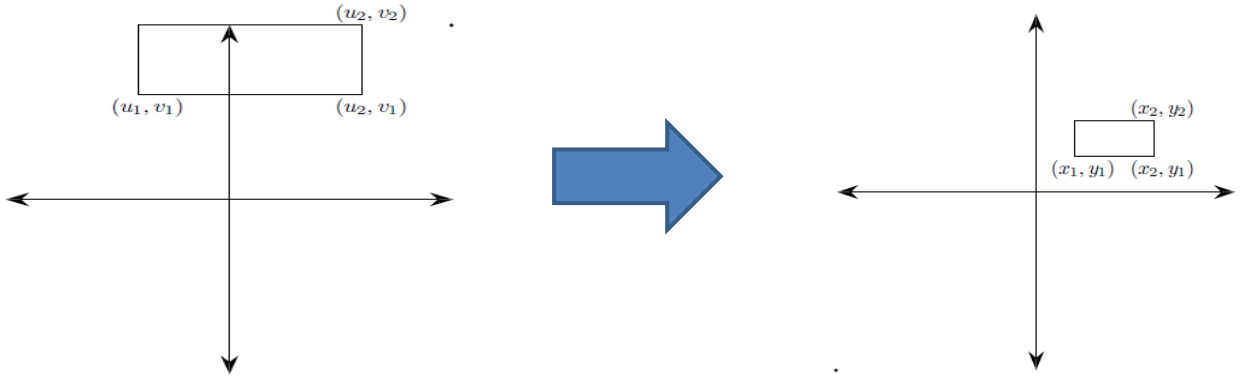
Inverses Revisited

- What is the inverse of a sequence of transformations?
  - $(M_1M_2...M_n)^{-1} = M_n^{-1}M_{n-1}^{-1}...M_1^{-1}$
- Inverse of a sequence of transformations is the composition of the inverses of each transformation in reverse order (why?)
- Say we want to do the opposite transformation of the example on slide 27. What will our sequence look like?
  - $(T^{-1}RST)^{-1} = T^{-1}S^{-1}R^{-1}T$
- We still translate to the origin first, then translate back at the end!

$$T^{-1}S^{-1}R^{-1}T = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/3 & 0 & 0 \\ 0 & 1/3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90 & \sin 90 & 0 \\ -\sin 90 & \cos 90 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

Aside: Windowing Transformations (CG terminology)

- Windowing transformation maps contents of 2D clip rectangle ("window") to a "viewport" rectangle on the screen, e.g., interior canvas ("client area") of a window manager's window; also called window-to-viewport transformation
- Sends rectangle with bounding coordinates (u_1, v_i), (u_2, v_2) to (x_1, y_1), (x_2, y_2)

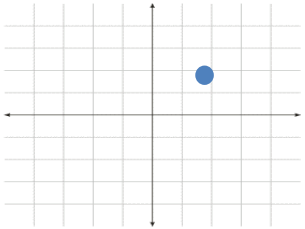


- The transformation matrix here is:

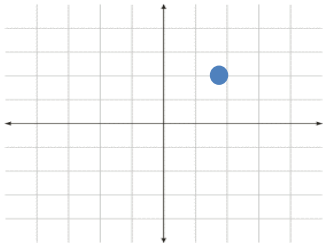
$$\begin{bmatrix} (x_2 - x_1)/(u_2 - u_1) & 0 & (x_1u_2 - x_2u_1)/(u_2 - u_1) \\ 0 & (y_2 - y_1)/(v_2 - v_1) & (y_1v_2 - y_2v_1)/(v_2 - v_1) \\ 0 & 0 & 1 \end{bmatrix}$$

Aside: Transforming Coordinate Axes

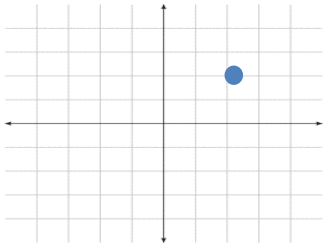
- We understand linear transformations as changing the position of vertices relative to the standard axes
- Can also think of transforming the coordinate axes themselves



Rotation



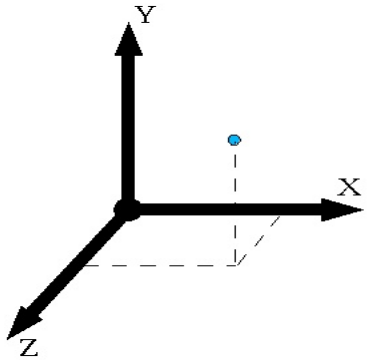
Scaling



Translation

- Just as in matrix composition, be careful of which order you modify your coordinate system

Dimension++ (3D!)



- How should we treat geometric transformations in 3D?
- Just add one more coordinate/axis!
- A point is represented as $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$
- A matrix for a linear transformation ***T*** can be represented as

  $\begin{bmatrix} T(e_1) & T(e_2) & T(e_3) \end{bmatrix}$

  where e_3 is the standard basis vector along the z-axis, $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$
- But remember to use homogeneous coordinates! Embed scale and rotation matrices as upper left submatrices and translation vectors as upper right subvectors of the right column

Transformations in 3D

Transformation	Matrix	Comments
Scaling	$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Looks just like the 2D version. We just added an s_z term.
Rotation	(see next slide)	In 2D, only one axis of rotation; now there are infinitely many! Must take all into account. See next slide...
Translation	$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Similar to the 2D version, just with one more entry *dz*, representing change in the *z*-direction.

Rodrigues' s Formula…

- Rotation by angle θ around vector $\boldsymbol{u} = [u_x \; u_y \; u_z]^T$
  - Note: this is an arbitrary **unit** vector $\boldsymbol{u}$ in *xyz*-space
- Here's a not so friendly rotation matrix

$$\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z \sin\theta & u_x u_z(1-\cos\theta) + u_y \sin\theta \\ u_x u_y(1-\cos\theta) + u_z \sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x \sin\theta \\ u_x u_z(1-\cos\theta) - u_y \sin\theta & u_y u_z(1-\cos\theta) + u_x \sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

- This is called the coordinate form of Rodrigues's formula
- Let's try a different approach…

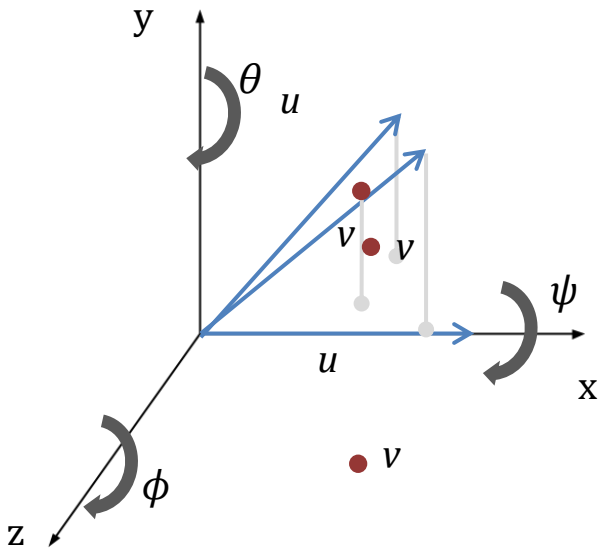
Rotating axis by axis (1/2)

- Every rotation can be represented as the composition of 3 different angles of **counter-clockwise** rotation around 3 axes, namely
  - x axis in the yz plane by ψ; y axis in the xz plane by θ; z axis in the xy plane by ϕ
- Also known as Euler angles, make problem of rotation much easier

$R_{yz}(\psi)$: rotation about x axis by ψ	$R_{zx}(\theta)$: rotation about y axis by θ	$R_{xy}(\phi)$: rotation about z axis by ϕ
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma & 0 \\ 0 & \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- Note these differ only in how the 3x3 submatrix is embedded in the homogeneous matrix, but the row-column order is different for R_{zx}
- You can compose these matrices to form a composite rotation matrix

Rotating axis by axis (2/2)



- It would still be difficult to find the 3 angles to rotate by, given arbitrary axis *u* and specified angle ψ
- Solution? Make the problem easier by mapping *u* to one of the principal axes

- **Step 1**: Find a θ to rotate around *y* axis to put *u* in the *xy* plane
- **Step 2**: Then find a ϕ to rotate around the *z* axis to align *u* with the *x* axis

Now that **u** *is in a convenient alignment, we can do our transformation rotation for vertex* v*:*

- **Step 3**: Rotate *v* by ψ around *x* axis (which is coincident with *u* axis)
- **Step 4**: Finally, undo the alignment rotations (inverse).

The only rotation we've preserved is the one around axis u *by* ψ*, which was our goal*

- Rotation matrix: $\boldsymbol{M} = \boldsymbol{R}_{zx}^{-1}(\theta)\boldsymbol{R}_{xy}^{-1}(\phi)\boldsymbol{R}_{yz}(\psi)\boldsymbol{R}_{xy}(\phi)\boldsymbol{R}_{zx}(\theta)$

Inverses and Composition in 3D!

- Inverses are once again parallel to their 2D versions…

Transformation	Inverse Matrix
Scaling	$\begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation	$\boldsymbol{R_{yz}^{-1}(\psi)}$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & \sin\psi & 0 \\ 0 & -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, $\boldsymbol{R_{zx}^{-1}(\theta)}$ $\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, $\boldsymbol{R_{xy}^{-1}(\phi)}$ $\begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Translation	$\begin{bmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- Composition works exactly the same way…

Example in 3D!

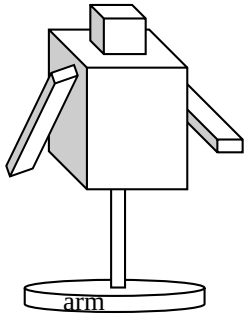
- Let's take some 3D object, say a cube centered at (2,2,2)
- Rotate clockwise in object's space by 30° around *x* axis, 60° around *y*, and 90° around *z*
- Scale in object space by 1 in the *x*, 2 in the *y*, 3 in the *z*
- Translate by (2,2,4) in world space
- Transformation sequence: $\boldsymbol{TT_0^{-1}S_{xy}R_{xy}R_{zx}R_{yz}T_o}$, where $\boldsymbol{T_0}$ translates to (0,0):

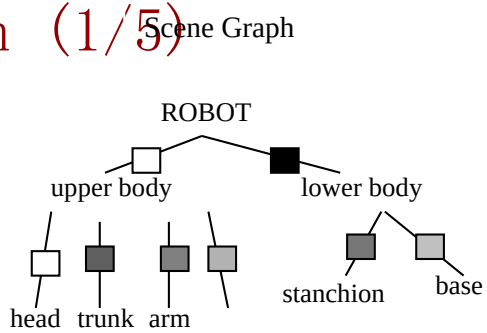
$$\begin{bmatrix}1&0&0&2\\0&1&0&2\\0&0&1&4\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&2\\0&1&0&2\\0&0&1&2\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&2&0&0\\0&0&3&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}\cos 90&\sin 90&0&0\\-\sin 90&\cos 90&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}\cos 60&0&-\sin 60&0\\0&1&0&0\\\sin 60&0&\cos 60&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&\cos 30&\sin 30&0\\0&-\sin 30&\cos 30&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&-2\\0&1&0&-2\\0&0&1&-2\\0&0&0&1\end{bmatrix}$$

$\boldsymbol{T}$ $\boldsymbol{T_0^{-1}}$ $\boldsymbol{S_{xy}}$ $\boldsymbol{R_{xy}}$ $\boldsymbol{R_{zx}}$ $\boldsymbol{R_{yz}}$ $\boldsymbol{T_0}$

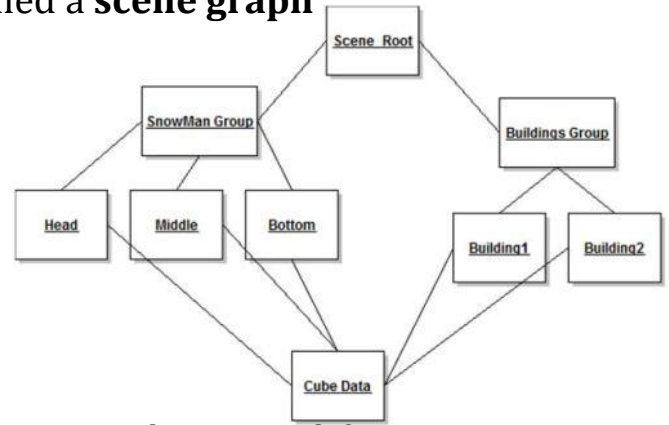
Transformations and the scene graph (1/5)

- Objects are typically composites:





- 3D scenes are often stored in a directed acyclic graph (DAG) called a **scene graph**
  - WPF (Windows Presentation Foundation)
  - Open Scene Graph (used in the Cave)
  - X3D ™ (VRML ™ was a precursor to X3D)
  - most game engines
- Typical scene graph format:
  - **objects** (cubes, sphere, cone, polyhedra etc.):
  - stored as nodes (default: unit size at origin)
  - **attributes** (color, texture map, etc.): stored as separate nodes
  - **Transformations**: also nodes



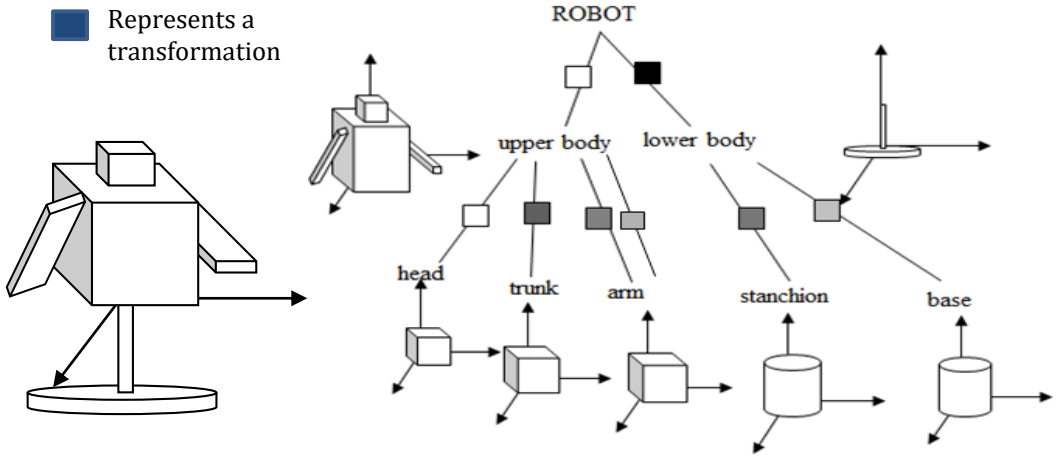
Example scenegraph from a game engine

Transformations and the scene graph (2/5)

- For your assignments use simplified format:
  - Attributes stored as a components of each object node (no separate attribute node)
  - A transform node affects its subtree
  - Only leaf nodes are graphical objects.
  - All internal nodes that are not transform nodes are object group nodes

Transformations and the scene graph (3/5)

- **Step 1:** Various transformations are applied to each of the leaves (object primitives—head, base, etc.)
- **Step 2:** Transformations are then applied to groups of objects (form upper and lower body, etc…)



This format means that instead of designing new primitives for every single shape we need, we can just apply transformations to a smaller set of primitives to form complex composite 3D shapes.

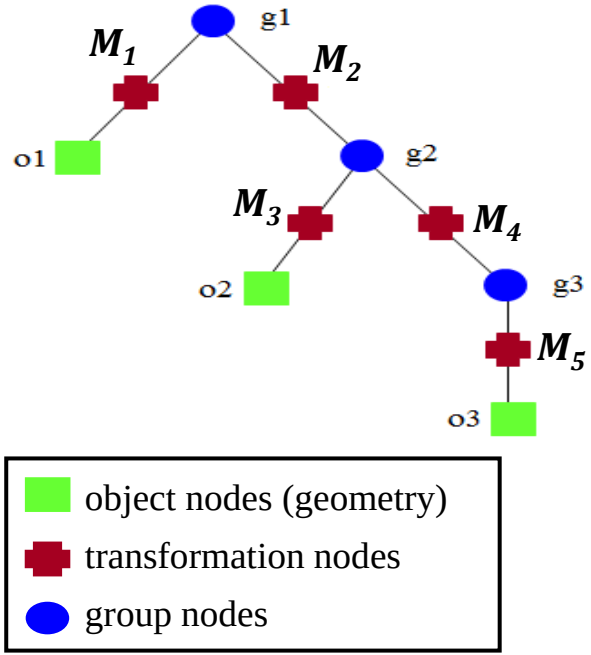
Together the above hierarchy of transformations forms the "robot" scene as a whole

Transformations and the scene graph (4/5)

- A **cumulative transformation matrix (*CTM*)** builds as you move up the tree.

- Note that higher level transformation matrices are appended to the front of the sequence

- Example:
  - For object 1 (o1), $CTM = M_1$
  - For o2, $CTM = M_2M_3$
  - For o3, $CTM = M_2M_4M_5$
  - For a vertex *v* in o3, position in world coordinate system is $CTM\ v = (M_2M_4M_5)\ v$



Transformations and the scene graph (5/5)

- You can easily reuse groups of objects (sub-trees in the scene graph) if they have been defined already
- This might occur if you have multiple similar components to your scene. For example, the robot's 2 arms
- Here, group 3 has been used twice.
- Transformations defined within group 3 itself do not change; there are different ***CTM***s for each use of group 3 as a whole
- T_0T_1 vs. $T_0T_2T_4$