Week 2 Scene Trees, Homogenous coordinates, Transformations

Scene Trees

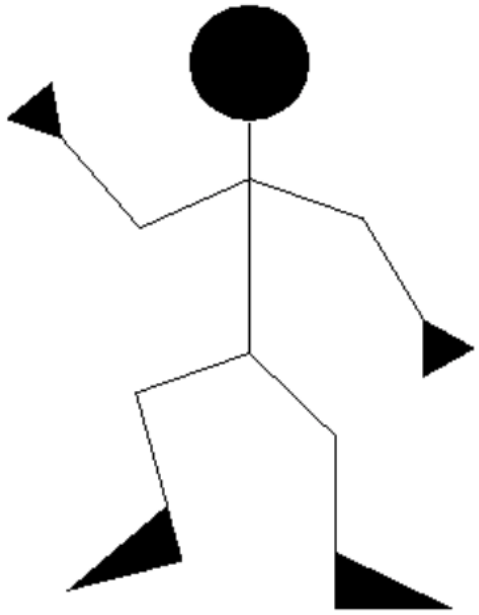
Scene tree describes how different objects in the scene are connected together

To draw (calculate) an compositing object containing lots of components (of this object), instead of drawing it piece by piece (by piece, it means vertex), we represent those complex scenes with a *scene tree*. Each node represents a component and each edge represents the transformation to get from the parent component's coordinate system to the child's

> Scene Tree It is a general data structure commonly used by vector-based graphics editing applications and modern computer games. A scene tree is a collection of nodes in a graph or tree structure. A tree node may have many children but only a single parent. A common feature is the ability to group related shapes and objects into a compound object that can then be moved, transformed, etc, as easily as a single object.

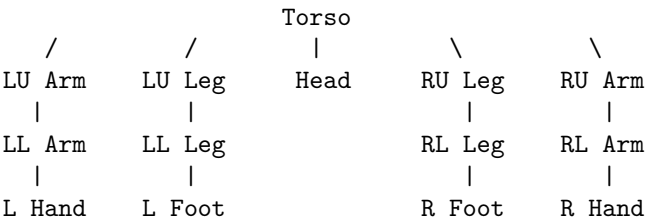
Scene Graph

- Directed acyclic multi-graph
- Each node can have multiple parents
- Multiple edges can go from parent to child
- Shared noes are drawn multiple times

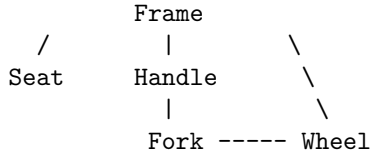


Example

Supposing we have to draw a object demonstrated above, in an abstract way, we can split the whole figure (torso) into subcomponents, including head, legs and arms. But there is still one missing part connecting all of those subcomponents - a torso. In terms of the torso, the components can rotate, scale and transform with regarding to it, which makes the torso the root of our scene tree. Other than that, arms can also be split into upper arm, lower arms and hands.



Scene Graph with multiple parents Bicycle



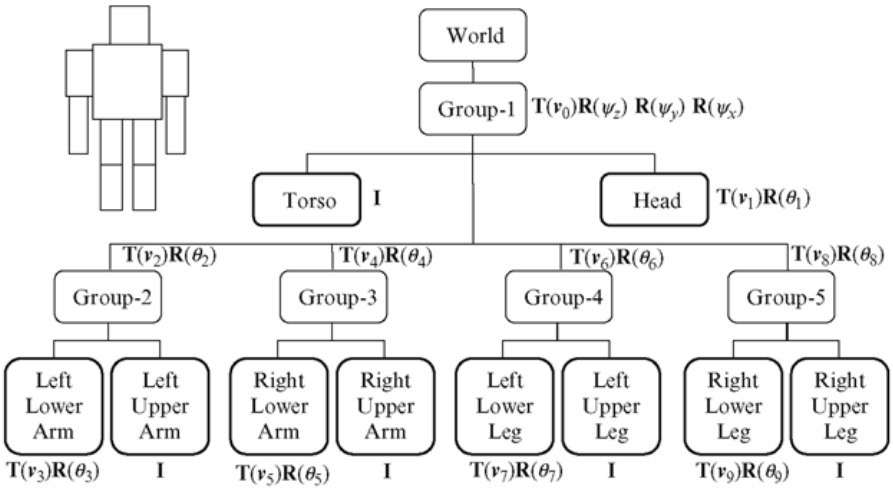


Figure 1: example

More Complicated Example

Implementation

```
drawTree(frame) {
    compute new_frame by transforming frame (TRS):
        translation
        rotation
        scale

    draw the object

    for all children:
        child.drawTree(new_frame)
}
```

Instance Transformation (TRS)

TRS are short for translate(T), rotate(R) and scale(S) and generally abbreviated to M, and each object (i.e., component mentioned above) can be created in their own local coordinate system (i.e., their own frames) and passes that onto its children. When a node in the graph is transformed, all **its children move with it**

Camera

The situation talked about above is speicically assuming that the camera is positioned at the world coordinate (0,0). In actual, camera is also considered to

be an object with its own position, rotation, and scale. The world is rendered as it appears in the camera's local coordinate frame. Aspect ratio $\frac{width}{height}$ should also be taken into consideration: the ratio of the width to the height.

View Transform

View transform converts the world coordinate frame in to camera's local coordinate frame by a series of **inverse** of the transformations that would convert the camera's local coordinate frame into **world coordinate**. Now, consider the camera as the reference frame, rather than that the camera moves, it is the whole world moves (or transforms).

Example

- Moving the camera left = moving the whole world right
- Rotating the camera clockwise = rotating the world anticlockwise
- Growing the camera's view = shrinking the world

Transformation pipeline

Transform performed in 2 stages:

$$P_{camera} \xleftarrow{\text{view}} P_{world} \xleftarrow{\text{model}} P_{local}$$

> Firstly, model transform transforms points in the *local coordinate system* to the *world coordinate system* > Then, view transform transforms points in the *world coordinate systems* to the *camera's coordinate system*

Example:

$$if\ P_{world} = Trans(Rot(Scale(P_{camera})))$$
$$then\ P_{camera} = Scale^{-1}(Rot^{-1}(Tran^{-1}(P_{world})))$$

Implementing a camera in UNSWgraph

```
CoordFrame2D viewFrame = CoordFrame2D.identity()
        /**
         * start with the last operation in world coordinate
         * and get the inverse of each parameter in operations (i.e., what -1 does)
         */
        .scale(1/myScale, 1/myScale)
        .rotate(-myAngle)
        .translate(-myPos.getX(), -myPos.getY());
```

Set camera as a node in the scene tree The relationship between the camera and the scene tree is actually that whenever the camera is transformed, we actually transform the world to adapt the camera. Hence, there should be a trace from the camera back to the root of scene tree, and of course a trace from camera to all the drawable object in the scene tree as well. More specifically, we need to compute the camera's transformations in world coordinates (and then get the inverse) in order to compute the view transform.

Transformations in the coordinate frame

Points A point can be described as a displacement from the origin:

$$P = p_i x + p_j y + \phi$$

where p_i is the modulus of the vector along i-aixs and ϕ is the origin

Transformation To convert **P** to a different coordinate frame:

1. Move the coordiante frame
    1. Convert i-axis: $i = p_i i' + p_j j'$
    2. Convert j-axis: $j = p_i i' + p_j j'$
    3. Convert origin: $\phi = p_x x' + p_y y'$
2. Convert the point based on the previous displacement but in the new coordinate frame

where x' is the previous x-aixs

Homogenous coordinates Point:

$$P = \begin{pmatrix} p_1 \\ p_2 \\ \mathbf{1} \end{pmatrix}$$

Vector:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \mathbf{0} \end{pmatrix}$$

In general, first two elements are coming from two axes repectively, while the thrid showing whether it is a point (1 for point, 0 for vector) or not. Because of this unique property (0/1 binary relationship), vectors can be summed altogether to get a new vector while a point can be the result of sum of one point and one vector. But two points cannot be performed addition in this case (resulting in 2 at the end).

Affine transformation

Matrices in this form (0s with the 1 at the end of the bottom row) are called affine transformation

1. Points

Transformations between coordintate frames can be represented as matrices:

$$Q = \mathbf{M}P$$

$$\begin{pmatrix} q1 \\ q2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & \phi_1 \\ x_2 & y_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

Affine Transformation is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation. However, an affine transformation does not necessarily preserve angles between lines or distances between lines or distinces between points.

2. Vectors

$$v = \mathbf{M}u$$

$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & \phi_1 \\ x_2 & y_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix}$$

Basic transformations

- Translation
- Rotation
- Scale
- Shear (happens whenever non-uniform scaling and then rotating happens)

Properties

- Collinearity: Affine transformation preserves the straight lines
  - $\mathbf{M}(A + t\mathbf{v}) = \mathbf{M}A + t\mathbf{M}\mathbf{v}$ It is still a vector.
- Parallelism: They maintain the parallel lines
  - Performing affine transformation on two parallel lines inside the coordinate system is essentially performing affine transformation on two parallel lines altogether.
- Ratio of lengths: They maintain the relative distances
- They don't necessarily preserve the angle or area
  - Because of non-uniform scaling
- (*) Convexity
- (*) Barycentres

2-D Transformation Translation here is to translate a point in its local coordinate frame to the point according to world coordinate frame

Translating a vector has no effect, since it is a displacement not a specific a point

- Translation

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \phi_1 \\ 0 & 1 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

- Rotation about the origin counter-clockwise

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix}$$

Applying homogeneous coordinate method for vectors (replace 1 with 0)

- Scale by factor (s_x, s_y) about the origin

$$\begin{pmatrix} s_x p_1 \\ s_y p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} s_x v_1 \\ s_y v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v)2 \\ 0 \end{pmatrix}$$

Likewise to vectors

- Shear
  - It can occur by scaling axes non-uniformly and then rotate
  - It does not preserve the angles
  - It can be avoided by uniforming scaling
  - Horizontal - sheared along y-axis (i.e., x-axis is like what it is expected)
  - Vertical - sheared along x-axis

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h(\text{if horizontal else } 0) & 0 \\ v(\text{if vertical else } 0) & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 0 \end{pmatrix}$$

Matrix

OpenGL stores matrix in column-major order.

```
float[] values = new float[] {
      1, 0, 0, // x
      0, 1, 0, // y
      0, 0, 1  // phi
};
```

Composing/Decomposing Transformations

Composing Transformation We can combine series of transformations by post-mulitplying their matrices. The reason why we can do it is that composition of two affine transformations is also affine.

$$\mathbf{M} = \mathbf{M}_T\mathbf{M}_R\mathbf{M}_SQ = \mathbf{M}P = \mathbf{M}_T\mathbf{M}_R\mathbf{M}_SP$$

Decomposing Transformation Every 2D affine transformation can be decomposed as:

$$\mathbf{M} = \mathbf{M}_T\mathbf{M}_R\mathbf{M}_S\mathbf{M}_{SHEAR}$$

If scaling is always uniform in both axes, then shear can be eliminated.

Example Consider, we now have a matrix form to be decomposed

$$\begin{pmatrix} i_1 & j_1 & \phi_1 \\ i_2 & j_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix}$$

It is important to make the assumption that **uniform scaling and hence no shear**

$$translation = (\phi_1, \phi_2, 1)^T$$
$$rotation = atan2(i_2, i_1)$$
$$scale = |i|$$

`atan2` here means to watch the angle for [0,180) [-180, 0)

A sloid example: Given $\begin{pmatrix} 0 & -2 & 1 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$

Translation: $(1, 2, 1)^T$ Rotation: $atan2(2, 0) = 90$ Scale: $|i| = |j| = 2$ Is shear: NO, cause $\mathbf{i} \cdot \mathbf{j} = 0$

Reparenting

Consider a scene tree, where bottle was originally held by the table (bottle is the child of table). Then, a person picks up the bottle by his hand (bottle is reparenting to the hand - child of the person).

Exercise

1. Translate by `(1,0.5)` then plot point `P = (0.5,0.5)` in local frame. What is the point in world coordinate?

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1 \\ 1 \end{pmatrix}$$

   Hence, the world coordinate for this point is [1.5, 1].

   Alternatively, we can also draw the identity frame and translate the frame with [1, 0.5] and then draw the point [0.5, 0.5] in the new frame. And see the world coordinate of point.

2. What would the matrix for scaling -1 in the x and y direction look like?

   `x-axis` and `y-axis` are flipped over. Is the same as `rotate(180)` and `rotate(-180)`

3. What would the matrix for rotating by 180 degrees look like?

$$\begin{pmatrix} cos(180) & -sin(180) & 0 \\ sin(180) & cos(180) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

4. What would the matrix be for the transform defined by this coordinate frame?

```
CoordFrame2D frame = CoordFrame2D.identity()
        .translate(1,2)
        .rotate(90);
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos(90) & -sin(90) & 0 \\ sin(90) & cos(90) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

   Identity frame can be ignored above

5. Suppose we continue from our last example and do the following `frame = frame.scale(2,2);`. What is the matrix for this transformation?

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -2 & 1 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

6. Given

$$\begin{pmatrix} 1.414 & -1.414 & 0.5 \\ 1.414 & 1.414 & -2 \\ 0 & 0 & -1 \end{pmatrix}$$

   1. What are the axes of the coordinate frame this matrix represents? What is the origin?

      - i - [1.414, 1.414]
      - j - [-1.414, 1.414]
      - ϕ - [0.5, -2]

   2. What is the scale of each axis?

      - $|i| = \sqrt{1.414^2 + 1.414^2} = 2$
      - $|j| = 2$

   3. What is the angle of each axis?

      `math.atan2(y,x)`

      - $\theta_i = atan2(1.414, 1.414) = 45$
      - $\theta_j = atan2(1.414, -1.414) = 135$

   4. Are the axes perpendicular (is shear)?

      $\mathbf{i} \cdot \mathbf{j} = 0$

      Or, from the (3) the difference between i-axis and j-axis is 90 degree.

      Hence, it is perpendicular.

7. Build a scene tree for modelling the solar system.

   One of the wrong tree:

```
Sun --- Mercury
    \-- Venus
    \-- Earth -- Moon
```

   The reason this scene tree is wrong is because that the moon is roatating whenever the earth is rotating. It does not mean that the concept is wrong (i.e., the moon rotating around the earth), while it is just the moon is rotating in a different speed or ratio relative to the earth. Hence, the moon should not be directly attached to the earth. Pivot objects are a good way to connect elements which have a cmmon components to their movement but don't have a direct parent-child relationship.

Solution:

```
Sun --- Mercury
    \-- Venus
    \-- EarthPivot --- Earth
                   \-- Moon
```