Transformations in OpenGL

- Translate
- Rotate
- Scale

- Push Matrix
- Pop Matrix

OpenGL Functions

- Transformations in OpenGL are not drawing commands. They are retained as part of the graphics state.
- When drawing commands are issued, the current transformation is applied to the points drawn.
- Transformations are cumulative.

Translation

Offset (tx, ty, tz) is applied to all subsequent coordinates. Effectively moves the origin of coordinate system.

- $x' = x + \mathrm{tx}\ , \quad y' = y + \mathrm{ty}, \quad z' = z + \mathrm{tz}$
- OpenGL function is `glTranslate`
- `glTranslatef`(tx, ty, tz);

Rotation

Expressed as rotation through angle θ about an axis direction (x,y,z) .

- OpenGL function – `glRotatef` (θ, x,y,z). So
  `glRotatef`(30.0, 0.0, 1.0, 0.0)
  rotates by 30° about y-axis.
- Note carefully:
  - `glRotate` wants angles in degrees.
  - C math library (sin, cos etc.) wants angles in radians.
  - $degs = rads * 180/\pi$; $rads = degs * \pi / 180$
- Positive angle? Right hand rule: if the thumb points along the vector of rotation, a positive angle has the fingers curling towards the palm.

Rotation (cont.)

- Frequently the axis is one of the coordinate axes. Common terms:
  - rotation about y-axis is heading/yaw
  - rotation about x-axis is pitch/elevation
  - rotation about z-axis is roll/bank
- 3-d rotation is an extremely difficult topic! There are several different mathematical formulations. Rotations do not commute – the order that transformations are done matters.

Scaling

- Multiply subsequent coordinates by scale factors sx, sy, sz. (Note: these are not a point, not a vector, just 3 numbers)

  $x' = \mathrm{sx} * x\ ,\quad y' = \mathrm{sy} * y,\quad z' = \mathrm{sz} * z$
- Often sx = sy = sz for a *uniform* scaling effect. If the factors are different, the scaling is called *anamorphic*.
- OpenGL function – `glScale` For example,

  ```
  glScalef(0.5,0.5,0.5);
  ```

  would cause all objects drawn subsequently to be half as big.

Order of transformations

- Transformations are cumulative and the order matters:
  - The sequence
    1. Scale 2, 2, 2
    2. Translate by (10, 0, 0)

    will scale subsequent objects by factor of 2 about an origin that is 20 along the *x*-axis
  - The sequence
    1. Rotate 90.0 deg about (0, 1, 0)
    2. Translate by (10, 0, 0)

    will set an origin 10 along the –ve *z*-axis
- For each object, the usual sequence is:
  1. Translate (move the origin to the right location)
  2. Rotate (orient the coordinate axes right)
  3. Scale (get the object to the right size)

Matrix representation

- Every 3-d point can be written as a 4-element vector and every 3-d transformation as a 4x4 matrix. (Yes, FOUR)
- For a point P (x,y,z), a fourth 'dummy' coordinate is appended. Internally the graphics card will treat the point as having 4 elements ($x,y,z,1$). These are called the *homogeneous coordinates* of P. (More on this later.)
- The identity matrix leaves any original vector unchanged:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Matrix representation

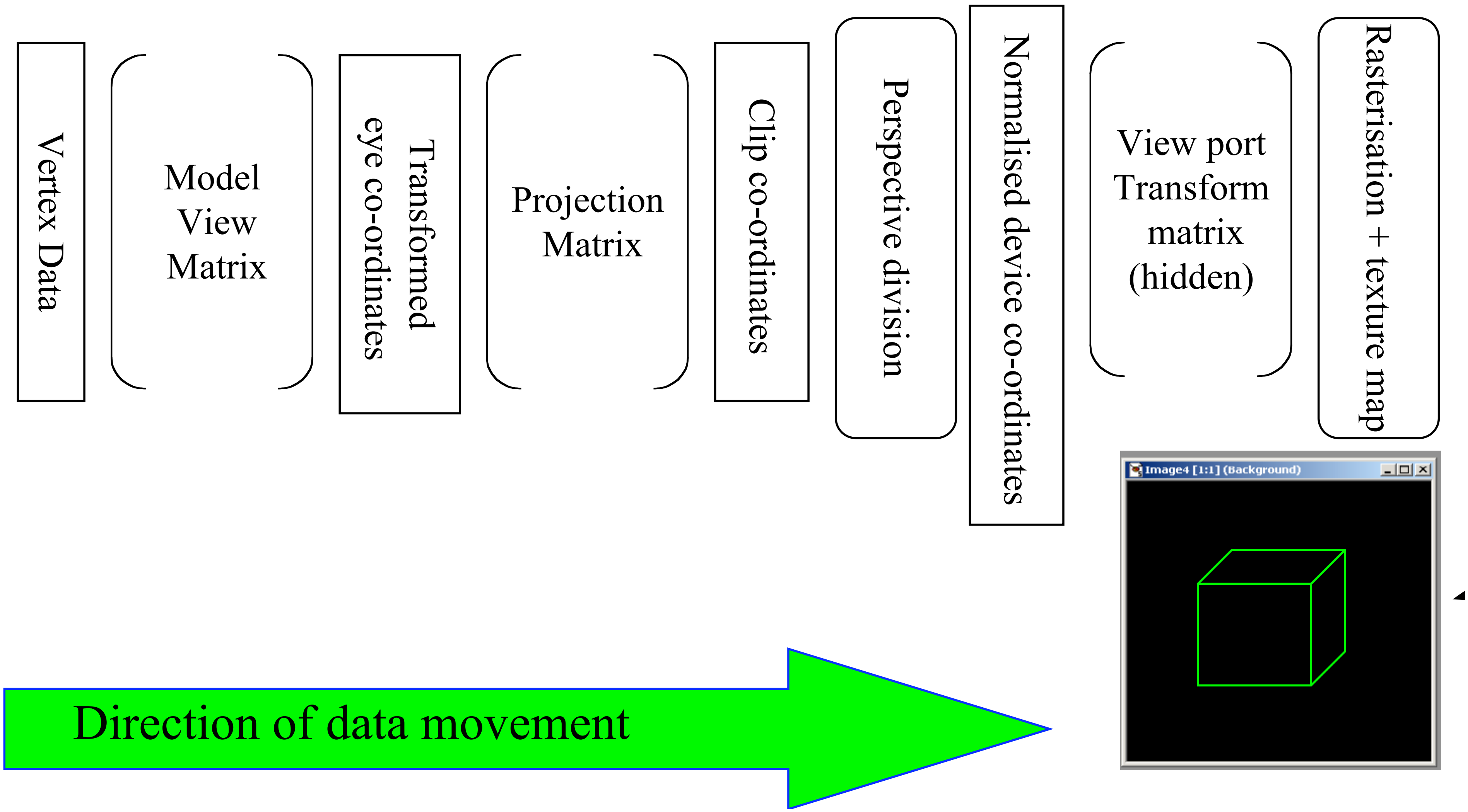
- If points are represented by column vectors, a translation is represented by a matrix with the offset values in the 4th column:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

•A rotation matrix uses the top left 3x3 area. A scaling matrix puts the scale factors on the diagonal.

•A matrix can represent *any* 3-d transformation, including some we haven't covered such as shearing and perspective projection.

The OpenGL pipeline



Matrices and Graphics State

- Each of the transformations above (Model View Matrix, Projection Matrix etc.) is maintained by OpenGL as part of the graphics state. (Current Transformation Matrix CTM)
- `glLoadIdentity` sets the CTM to the identity matrix, for a "fresh start".
- When `glRotate` or similar command is issued, the appropriate transformation matrix is updated.
- Note carefully that the rotation matrix doesn't overwrite the old CTM. It updates CTM by matrix multiplication.
- In fact the CTM is so important that OpenGL can keep several of them in a stack. By popping the stack, you can recover an old and possibly still-useful CTM.

Nested Transformations

- The sequence
  translate 1.5 0 0
  cube
  translate 8.5 0 0
  cube
  will draw two cubes with x centres 1.5 and 10.0 respectively.
- We could create the same image with the sequence
  save state
  translate 1.5 0 0
  cube
  restore state
  save state
  translate 10.0 0 0
  cube
  restore state

- Here both cubes have an absolute translation and the order in which the two cubes are drawn does not matter.

Push and Pop

- `glMatrixMode(GL_MODELVIEW)`
- `glMatrixMode(GL_PROJECTION)`
- `glMatrixMode(GL_TEXTURE)`

- **`glPushMatrix`**`();`
  - Save the state.
  - Push a copy of the CTM onto the stack.
  - The CTM itself is unchanged.
- **`glPopMatrix`**`();`
  - Restore the state, as it was at the last Push.
  - Overwrite the CTM with the matrix at the top of the stack.
- **`glLoadIdentity`**();
  - Overwrite the CTM with the identity matrix.

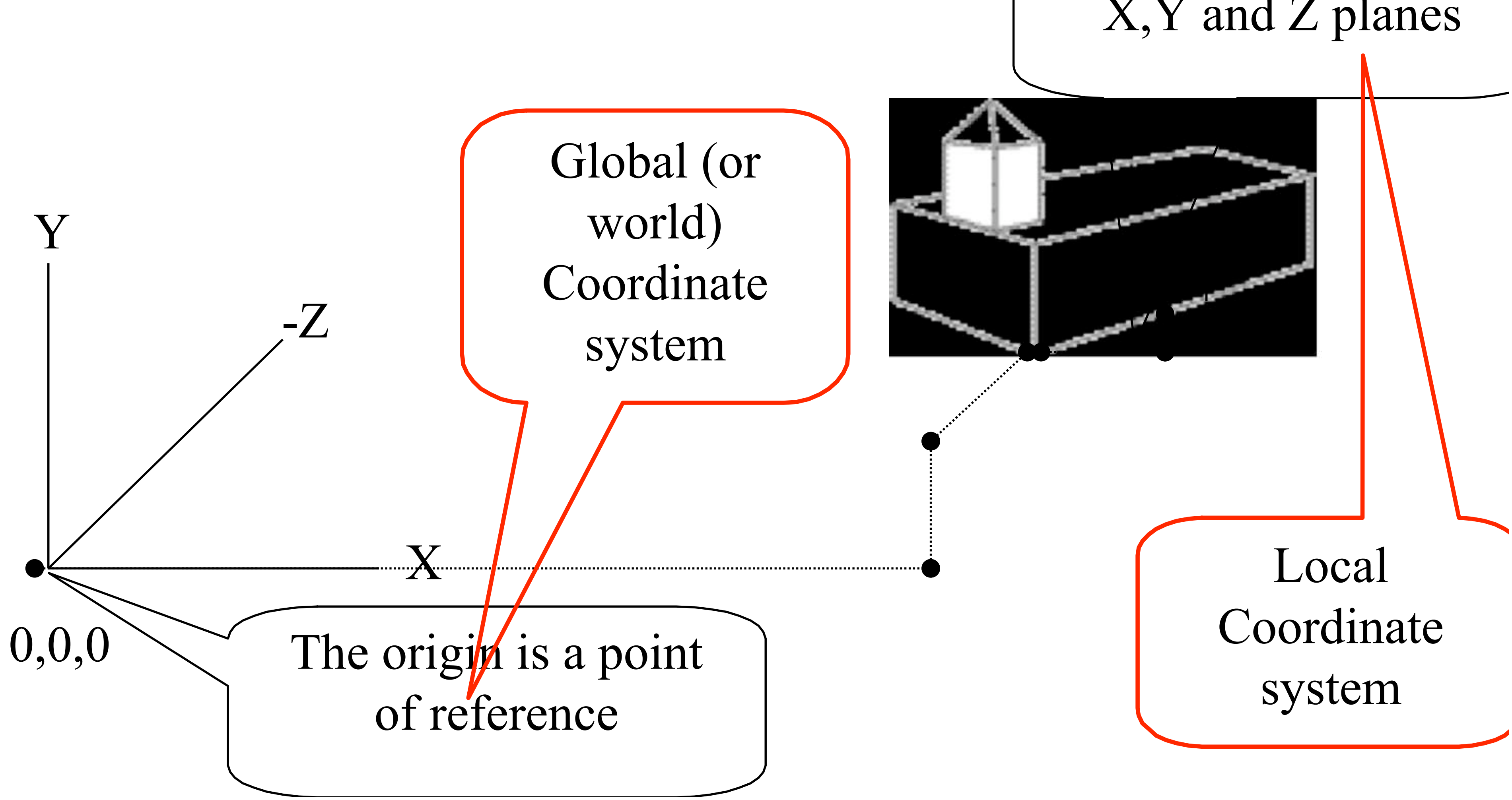
Local coordinate system

- The standard way to construct a complex 3D model is to define each individual part in a local coordinate system. This has whatever origin, and whatever unit, is most convenient.
- Typically we will draw the part centred at the origin, and aligned with the coordinate axes.
- Each part is then transformed relative to some parent before being rendered. Groups of parts may themselves have a parent, and so on up to the final world coordinate system.
- There are many changes of CTM, so many that your head will spin for a couple of weeks. Everyone has to get skilled at this – it's the source of the power and flexibility and (believe it or not) ease of use of a graphics system.

Coordinate systems

So the trick to understanding world co ordinates is to know what the reference point is (and it relation to the origin)

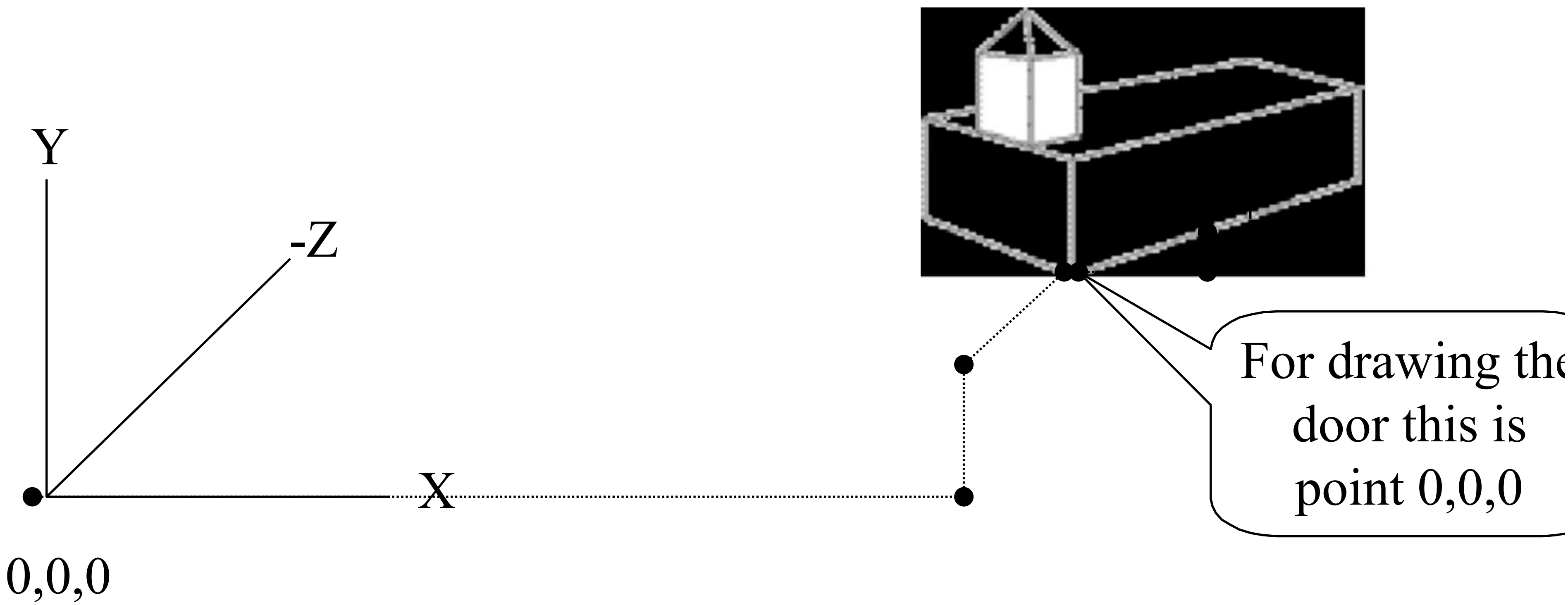
The <u>world</u> coordinate system is the one in which the entire 3D model is defined.



Coordinate systems

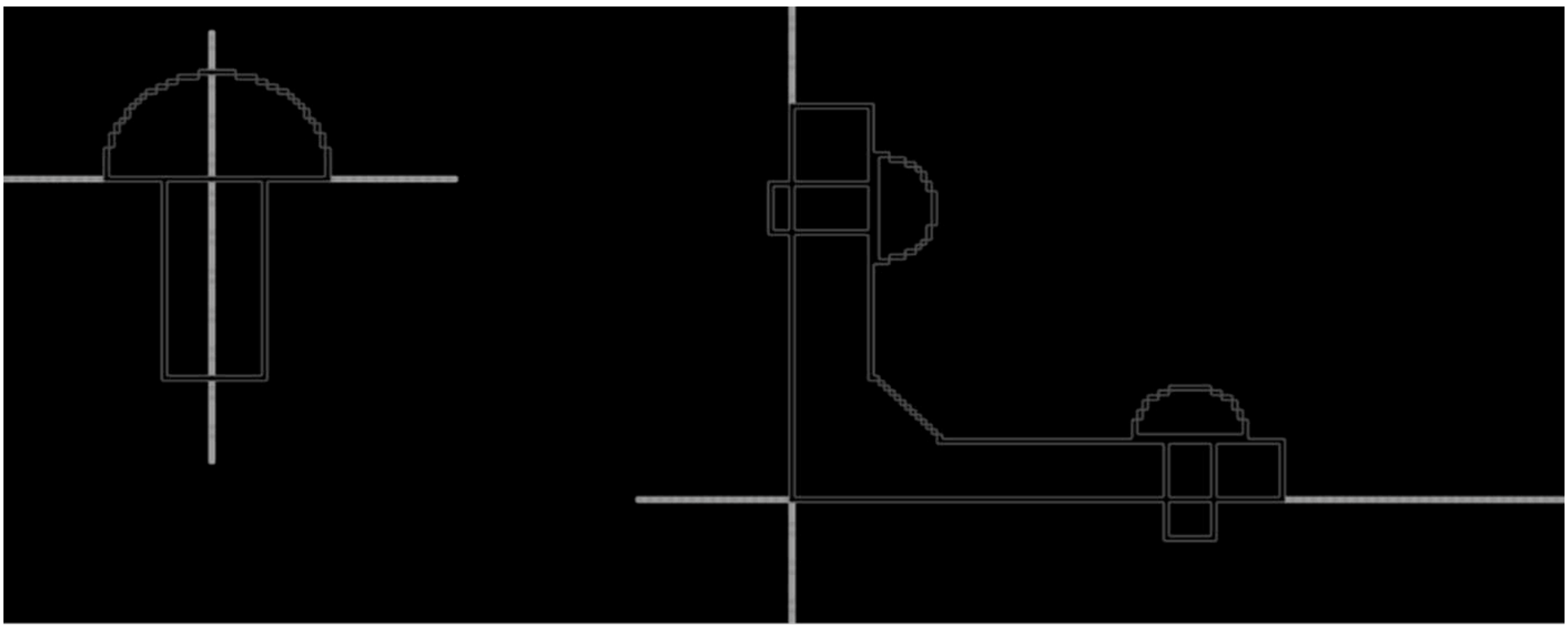
So the trick to understanding world co ordinates is to know what the reference point is (and it relation to the origin)

The world coordinate system is the one in which the entire 3D model is defined.



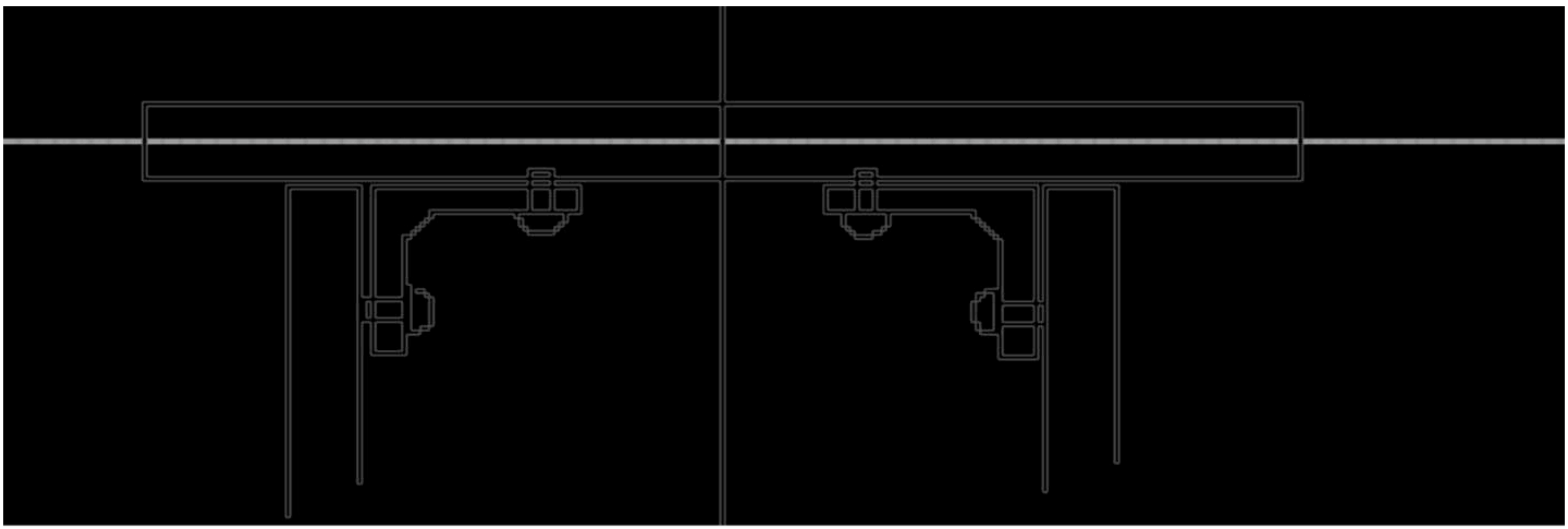
Scene Graphs

- We define a rivet in a local coordinate system, and then translate and rotate each rivet within the coordinate system of a bracket:



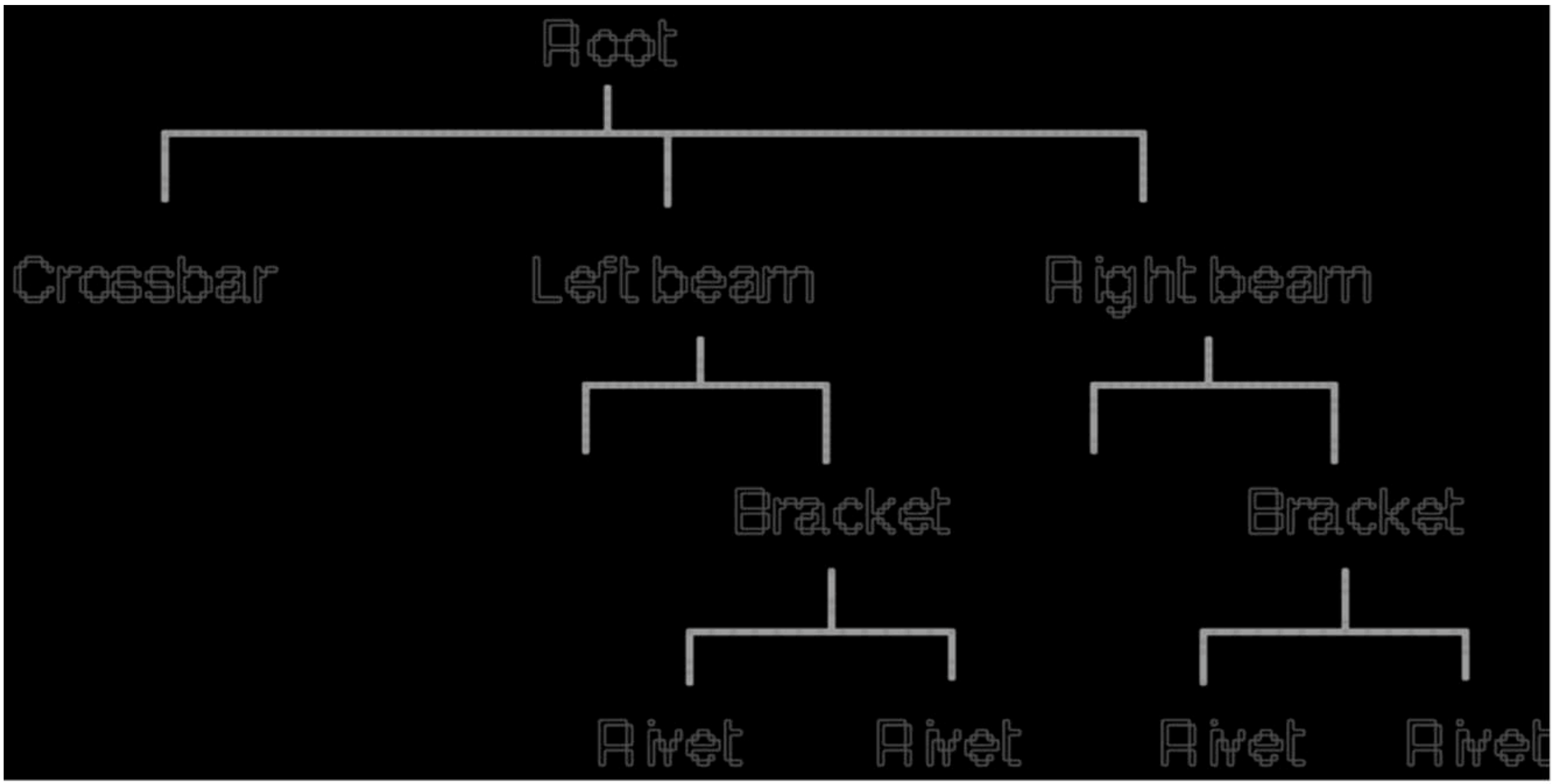
Scene Graphs (cont.)

- Each bracket is in turn translated and rotated within the coordinate system of a subassembly:



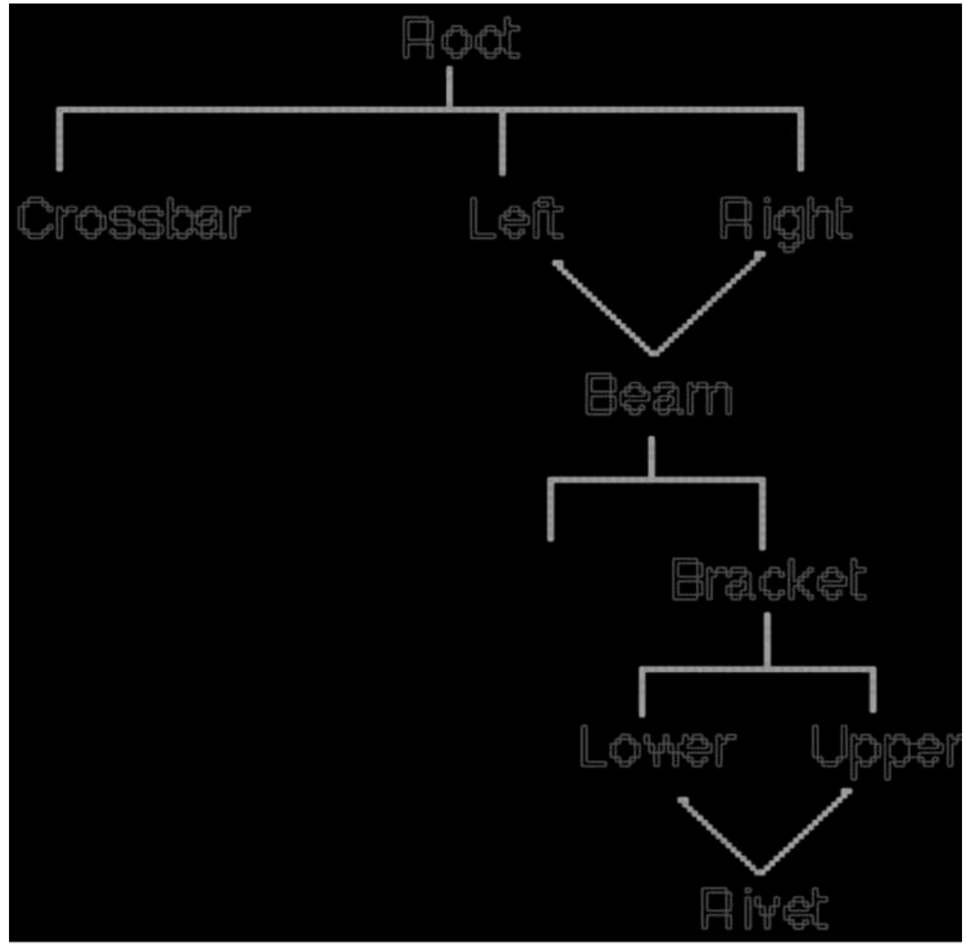
Representation as Graph

- This hierarchical 3D model can be represented as a scene graph:



DAG, Directed Acyclic Graph

- Each node in the scene graph inherits the CTM from its parent. Descending the graph pushes the matrix stack, ascending pops.
- Above, the scene graph was shown as a rooted tree. But a scene graph is better represented as a DAG, directed acyclic graph, in which each node can have more than one predecessor. This picture more accurately reflects the structure of our function calls.



DAG, Directed Acyclic Graph

- May be implicit in Code
  - In the structure of your calls to drawing functions e.g. `drawCar`(), `drawAxle`(), `drawWheel`(), `drawCylinder`()
- Can be in data
  - Such as a mesh (eg .3DS format)
- Can be in a human readable language
  - VRML

VRML
A Scene Graph Language

- Virtual Reality Modelling Language
  - I think Virtual Reality Mark up Language which is sometimes seen is wrong
- It is a scene graph language
- Designed by SGI and is closely related to OpenGL (many features in common)
- Despite early successes VRML has not really caught on
- Some companies are trying to future proof their character meshes by storing them in VRML
- Can be useful to us because it is human readable an relatively easy to understand