Computer Graphics: Model and View Transformations

Part 2 – Lecture 1

Christof Lutteroth

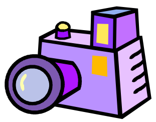
- Just became a permanent lecturer
- From Berlin, Germany
- My research interests:
  model-based SE, HCI, DBMS, CG, …
- Contact details:

lutteroth@cs.auckland.ac.nz
Phone 373-7599 84478
Office: 303 - 494 (4th level CompSci building)

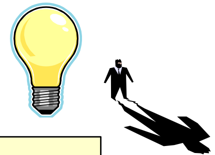
- If you have questions, come to my office at any time ☺

Second Half

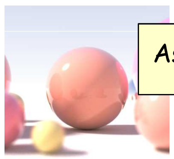
Viewing and Projection



Illumination and Shading



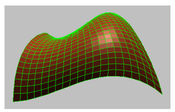
Ray Tracing



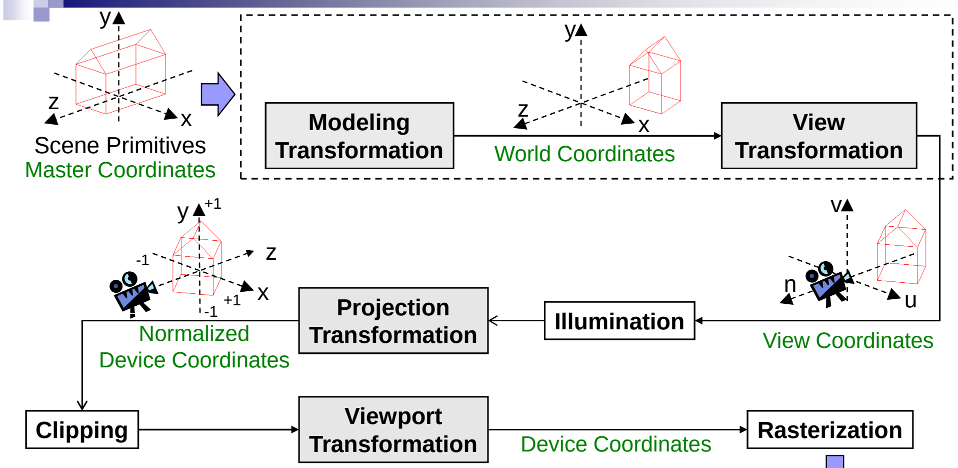
Assignment 3

Color Theory & Rasterization

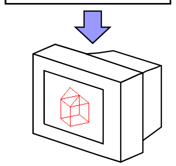
Curve and Surface Design



Raytracing image thanks to Tim Babb



THE OPENGL RENDERING PIPELINE

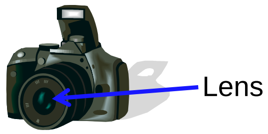


The Camera Analogy

1. **Model Transformations**
   Arranging objects in a scene

2. **View Transformation**
   Positioning the camera

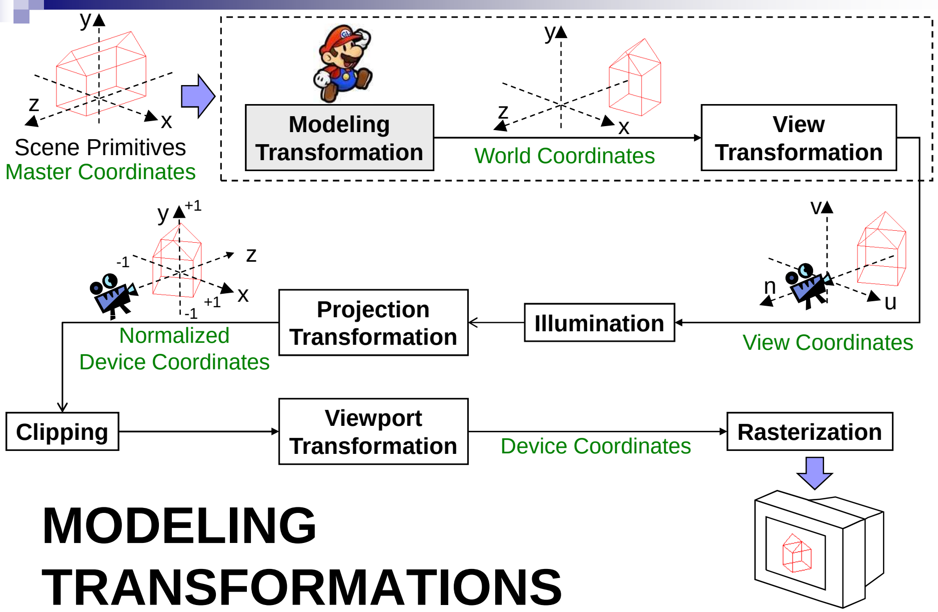
3. **Projection**
   Choosing a lens & taking a photo



4. **Viewport Transformation**
   Printing a photo

OpenGL Rendering Pipeline

- State machine: set up state of rendering pipeline
  - Choose which part of the pipeline should be modified, e.g. `glMatrixMode(MODEL_VIEW)`
  - Set how it should be modified, e.g. `glTranslatef(…)`, `glRotatef(…)`, …
- Now send scene primitives down the pipeline, e.g. `glBegin(GL_TRIANGLES) … glEnd()`
- All primitives are automatically transformed by the pipeline



MODELING TRANSFORMATIONS

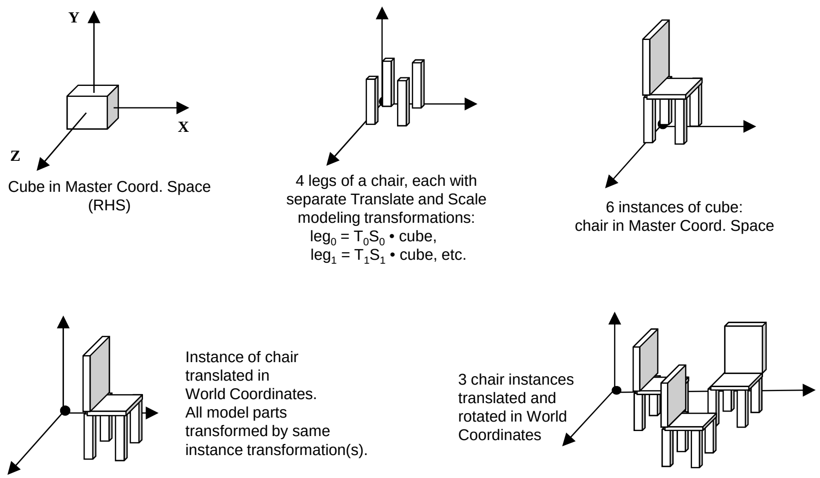
Modeling Transformations Recap

- Translation, rotation, scaling: each corresponds to a matrix

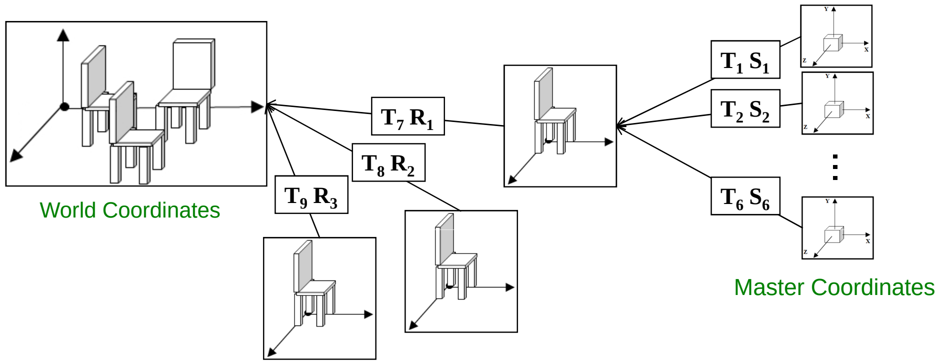
```
glMatrixMode(MODEL_VIEW);
glLoadIdentity();                  // matrix I
glTranslatef(tx, ty, tz);          // matrix T
glRotatef(angle, ux, uy, uz);      // matrix R
glScalef(sx, sy, sz);              // matrix S
```

- Now all vertices P are transformed into *P'* with
  $P' = \mathbf{M}_{\mathbf{ModelView}}\, P = (\mathbf{I\ T\ R\ S})\, P = \mathbf{I\ T\ R}\, P^{(1)} = \mathbf{I\ T}\, P^{(2)} = \mathbf{I}\, P^{(3)} = P^{(3)}$
- The order of transformation matters! Rightmost matrix applied first.
- **Matrix Stack** helps to apply different transforms to different objects
  - Topmost matrix is currently used for transforms
  - `glPushMatrix()` puts a copy of topmost matrix onto stack
  - `glPopMatrix()` removes topmost matrix

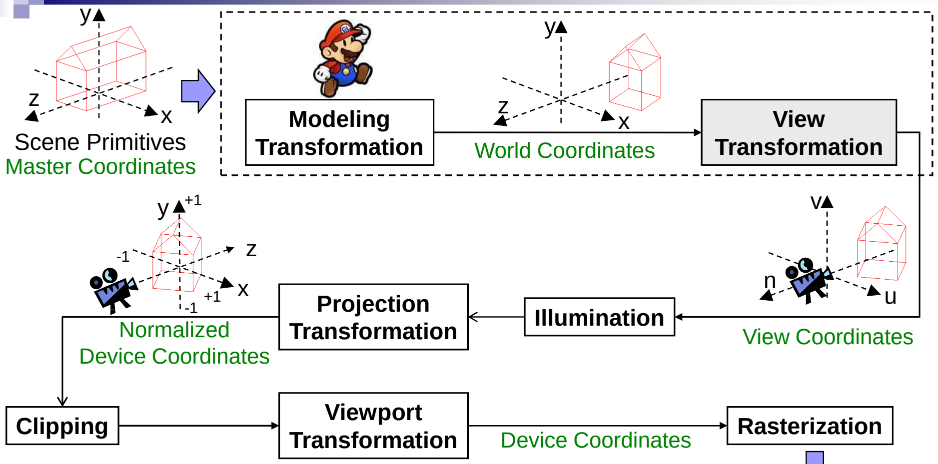
Modeling Transformations Example

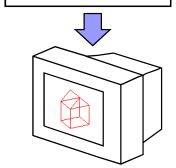


Scene Graph and Matrix Stack



- Composing complex objects from simple parts (parent - children)
- Create a method for every object: cube, chair
- Push matrix before drawing child object, pop after returning to parent object

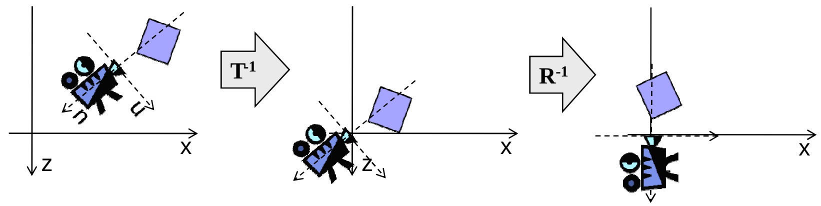




VIEW TRANSFORMATION

View Transformation

- Camera is at the origin looking down negative Z axis
- Could change camera position with translation **T** and rotation **R**
- But instead of rotating and moving camera, transform our scene inversely so that the camera sees what we want it to see:



- In other words: we translate and rotate **view coordinate system** so that it is aligned with world coordinate system
- Viewing transform can be done as the last transform in $\mathbf{M}_{\mathbf{ModelView}}$ (i.e. must be set first in program)

Specifying View Position & Orientation

How to write an OpenGL program that sets the view for a camera…

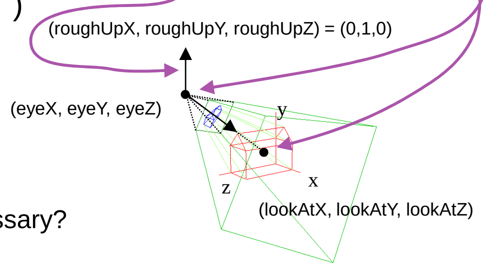
1. Given an **camera (eye) position** and a **point to look at**?
2. Given an **eye translation** and a **rotation**?
3. For an **airplane flight simulator** (simulating the view out the front window) where the simulator position and orientation are controlled via pilot commands that set the plane's **pitch**, **yaw**, and **roll**?
4. Mounted on a **pilot's helmet** (simulating the pilot's eye view such as in a virtual reality head mounted display) where the pilot can move (translate) and rotate his head **within the airplane's cockpit**?
5. Mounted on the end of a **multi-jointed robot arm**, such as the NASA Space Shuttle Canadian arm?

- You already know the answer to #1, use gluLookAt(). But, there is no single gl, glu, or glut function for #2-#5 !

Specifying View Position & Orientation

- **Solution:** OpenGL program that sets view position & orientation given eye position and a point to look at. Use `gluLookAt()`
- Need:
  - Eyepoint
  - View direction
  - Something that specifies camera rotation around its axis `roughUp` may be any vector not parallel to (eye-look) vector. Along with the (eye-look) vector it defines the plane in which the true up vector must lie.
    Question: why is roughUp necessary?

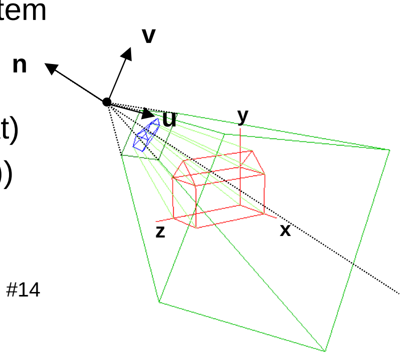
```
gluLookAt(
    eyeX, eyeY, eyeZ,
    lookAtX, lookAtY, lookAtZ,
    roughUpX, roughUpY, roughUpZ
)
```



The View Coordinate System (UVN)

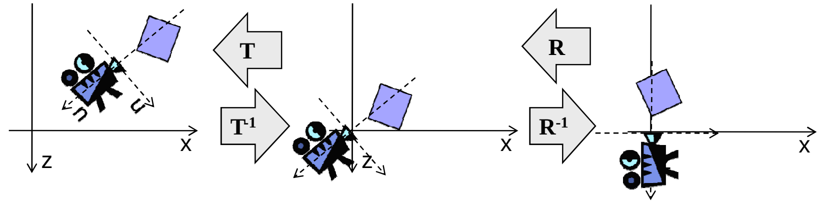
a.k.a. *Eye Coordinate System* or *Camera Coordinate System*

- From the Eye and LookAt points plus the approximate Up vector, can derive UVN Coordinate system (Eye Coords.) basis vectors:
  - **n** = Normalised(Eye – LookAt)
  - **u** = Normalised(Cross(**Up, n**))
  - **v** = Cross(**n, u**)



Alternate definition: Burkhard's notes, 5.1 slide #14

The View Transformation Matrix **V**



- To set up camera, we could rotate it (**R**) then translate it (**T**)
- V must do the inverse: V = $(\mathbf{T}\,\mathbf{R})^{-1} = \mathbf{R}^{-1}\,\mathbf{T}^{-1}$

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- What matrix **R** aligns an object with new basis vectors **u v n** ? We are looking for the inverse $\mathbf{R}^{-1}$ of that matrix.

The View Transformation Matrix V

1. From Part 1: we can rotate an object to be aligned with new basis vectors **u v n** by multiplying with:

$$\mathbf{R} = \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Rotation matrixes such as R are orthogonal, i.e. $\mathrm{col}_i \bullet \mathrm{col}_j = 0$ for $i \neq j$, and $\mathrm{col}_i \bullet \mathrm{col}_i = 1$
3. For an orthogonal matrix R: $\mathbf{R}^{-1} = \mathbf{R}^{T}$

$$\mathbf{V} = \mathbf{R}^{-1}\mathbf{T}^{-1} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -\mathbf{eye}\bullet\mathbf{u} \\ v_x & v_y & v_z & -\mathbf{eye}\bullet\mathbf{v} \\ n_x & n_y & n_z & -\mathbf{eye}\bullet\mathbf{n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

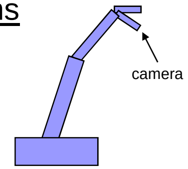
The World as Seen from a Robotic Arm

- View specified as a general **instance transformation**
  - Calls to `glRotatef()` for Euler angle rotations and to `glTranslatef()` for a translation to orient and position the camera (but, no scale).
  - Transformation matrix $\mathbf{M}$ that transforms System 1's coordinate frame (World Coord.) to System 2's frame (Eye Coord.) is:

    $$\mathbf{M} = \mathbf{T}\,\mathbf{R}_x\,\mathbf{R}_y\,\mathbf{R}_z$$

    Matrix $\mathbf{V}$, that transforms points from World to Eye Coordinates is

    $$\mathbf{V} = \mathbf{M}^{-1} = (\mathbf{T}\,\mathbf{R}_x\,\mathbf{R}_y\,\mathbf{R}_z)^{-1} = \mathbf{R}_z^{-1}\,\mathbf{R}_y^{-1}\,\mathbf{R}_x^{-1}\,\mathbf{T}^{-1}$$
  - **Note:** t_x, t_y, t_z are in World Coords., NOT relative to camera orientation.
- Specified as a **hierarchy** of instance transformations
  Example: camera on the gripper of a robot arm

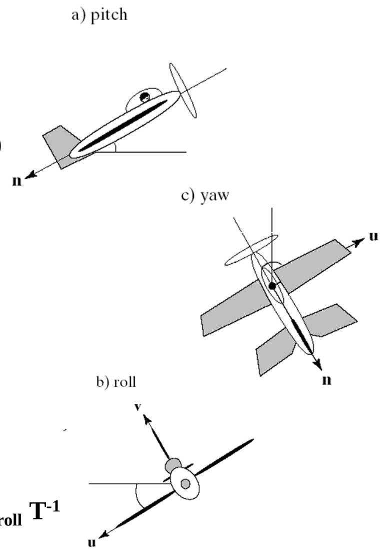


  - Arm hierarchy, joints: base, lower arm, upper arm, gripper
  - Instance transformation of gripper

    $$\mathbf{M} = \mathbf{T}_B\,\mathbf{R}_{By}\,\mathbf{T}_{LA}\,\mathbf{R}_{LAx}\,\mathbf{R}_{LAy}\,\mathbf{T}_{UA}\,\mathbf{R}_{UAx}\,\mathbf{R}_{UAy}\,\mathbf{T}_G\,\mathbf{R}_{Gx}\,\mathbf{R}_{Gy}\,\mathbf{R}_{Gz}$$
  - View transformation for camera attached to gripper, $\mathbf{V} = \mathbf{M}^{-1}$

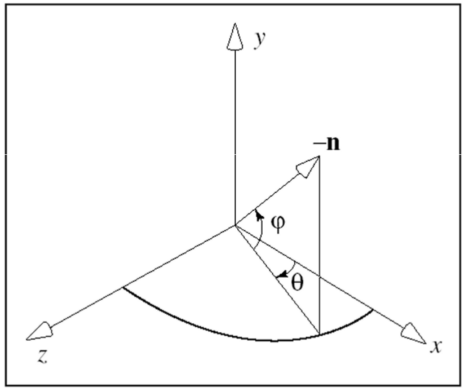
The World as Seen from an Aeroplane

- View specified as **pitch, yaw, roll**
  - Euler angle specification, normally applied: $\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch}$
  - *pitch* = angle **n** axis makes with plane $\mathbf{Y} = 0$ (horizontal)
    same as rotation about **u** axis
  - *yaw* = angle **u** axis makes with plane $\mathbf{Z} = 0$
    same as rotation about **v** axis
    (also known as *heading* or *bearing*)
  - roll = angle **u** axis makes with plane $\mathbf{X} = 0$
    same as rotation about **n** axis
  - Graphics applications often use a "no-roll" camera – pitch and yaw only
  - $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch}$, $\mathbf{V} = \mathbf{M}^{-1}$
    $\mathbf{V} = (\mathbf{T}\,\mathbf{R}_{roll}\,\mathbf{R}_{yaw}\,\mathbf{R}_{pitch})^{-1} = \mathbf{R}^{-1}_{pitch}\,\mathbf{R}^{-1}_{yaw}\,\mathbf{R}^{-1}_{roll}\,\mathbf{T}^{-1}$

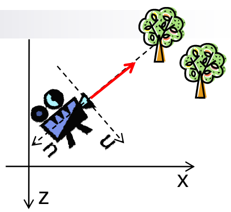


The World as Seen from an Aeroplane 2

- View specified as **azimuth, elevation**
  (tilt, optional but uncommon)
  - Euler angle specification, normally applied: $\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth}$
  - *azimuth* = angle **u** axis makes with the plane $\mathbf{Z} = 0$
    same as rotation about **v** axis, same as yaw
  - *elevation* = angle **n** axis makes with the plane $\mathbf{Y} = 0$ (horizontal)
    same as pitch
  - $\mathbf{M} = \mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth}$
    $\mathbf{V} = \mathbf{M}^{-1}$
    $= (\mathbf{T}\,\mathbf{R}_{elevation}\,\mathbf{R}_{azimuth})^{-1}$
    $= \mathbf{R}^{-1}_{azimuth}\,\mathbf{R}^{-1}_{elevation}\,\mathbf{T}^{-1}$



Moving our Camera



Problem:

How to move our camera relative to view direction?

(Which direction is "forward" for the camera?)

→ need to convert movements relative to view orientation into movements relative to world coords.

Solution: Slide function

- Translates movement along **u, v, n** axes to movement along **x, y, z**
- Given: movement vector $\mathbf{d_2} = (d_u, d_v, d_n)$ in view coords.
- Wanted: movement vector $\mathbf{d_1} = (d_x, d_y, d_z)$ in world coords.
- Solution: rotate the movement vector so that it is aligned with **u, v, n**

$$d_1 = \begin{pmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{pmatrix} d_2$$

For rotation matrix explanation see Burkhard's notes, 5.8 slide #40

SUMMARY

Summary

1. Vertices are automatically transformed by ModelView matrix: $P' = \mathbf{M}_{\mathbf{ModelView}}\, P = (\mathbf{V}\,\mathbf{M})\, P$
2. But instead of rotating and moving camera, transform our scene so that the camera sees what we want it to see
3. **V** is the inverse of the transformation we would use to set up the camera position and orientation

This Friday no lecture!!!
Come and visit the SE part 4 exhibition ☺

References:

- Model transformations: Hill, Chapter 5
- View Transformation: Hill, Chapter 7.22
- More View Transformations & Sliding: Hill, Chapter 7.3

Quiz

1. Given the camera setup transformations R_1, T_1, R_2 (applied in the given order), how do you determine the view transformation matrix V?
2. Create example matrixes R_1, T_1, R_2 and calculate V.
3. How do you translate movements in view coords. into world coords.?