EECS 487: Interactive Computer Graphics

Lecture 12:

- Vertex passing: triangle strip/fan, vertex array, drawing modes
- From 3D to 2D Overview
- Viewing transform

OpenGL: Drawing Triangles

You can draw multiple triangles between `glBegin(GL_TRIANGLES)` and `glEnd()`:

```
float v1[3], v2[3], v3[3], v4[3];
...
glBegin(GL_TRIANGLES);
  glVertex3fv(v1); glVertex3fv(v2); glVertex3fv(v3);
  glVertex3fv(v1); glVertex3fv(v3); glVertex3fv(v4);
glEnd();
```

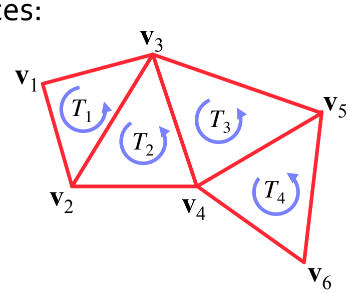
Each triangle sent to the rendering pipeline as 3 vertices at a time ⇒ vertex duplication, not efficient

- each must be transformed and lit
- fewer vertices = faster drawing

OpenGL: Triangle Strips

An OpenGL triangle strip primitive reduces this redundancy by sharing vertices:

```
glBegin(GL_TRIANGLE_STRIP);
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3); // T1
  glVertex3fv(v4); // T2
  glVertex3fv(v5); // T3
  glVertex3fv(v6); // T4
glEnd();
```

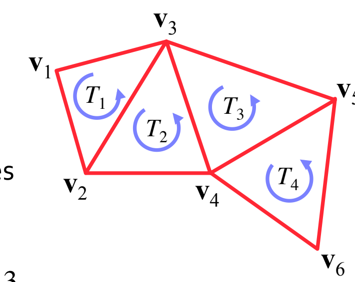


- triangle 1 is $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$
- triangle 2 is $\mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_4$ (why not $\mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4$?)
- triangle 3 is $\mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5$
- triangle 4 is $\mathbf{v}_5, \mathbf{v}_4, \mathbf{v}_6$

When looking at the front side, the vertices go counterclockwise (right-hand rule)

- n odd ⇒ T_n: n , $n+1$, $n+2$
- n even ⇒ T_n: $n+1$, n , $n+2$
- n starts at 1

Triangle Strips



Without strips:

- 8 triangles * 3 vertices = 24 vertices

With strips:

- use 1 vertex per triangle instead of 3
  - startup cost $\mathbf{v}_1$, $\mathbf{v}_2$, then
  - $\mathbf{v}_3(T_1)$, $\mathbf{v}_4(T_2)$, $\mathbf{v}_5(T_3)$, $\mathbf{v}_6(T_4)$, $\mathbf{v}_7(T_5)$, $\mathbf{v}_8(T_6)$, $\mathbf{v}_9(T_7)$, $\mathbf{v}_{10}(T_8)$
- total 10 vertices instead of 24
  - 10/8 = 1.125 vertices/triangle
  - 100*10/24 = 37.5% less data

We can expect the geometry stage to run almost 3 times faster!

Akenine-Möller02

How to Create Triangle Strips from a 3D Model?

Manually
- only doable for small models, and not fun…

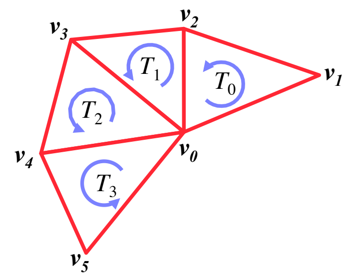
Or write your own program:
- need to know triangle's neighbors
- it's quite simple to make a working strip-creator
- to make a really good one is more work

Or use nvidia's `NVTriStrip` (on the web)

OpenGL: Triangle Fan

The `GL_TRIANGLE_FAN` primitive is another way to eliminate vertex duplication (also 1 vertex per triangle):

```
glBegin(GL_TRIANGLE_FAN);
 glVertex3fv(v0); // start with central point
 glVertex3fv(v1); // build triangles around it
 glVertex3fv(v2); // T0
 glVertex3fv(v3); // T1
 glVertex3fv(v4); // T2
 glVertex3fv(v5); // T3
glEnd();
```

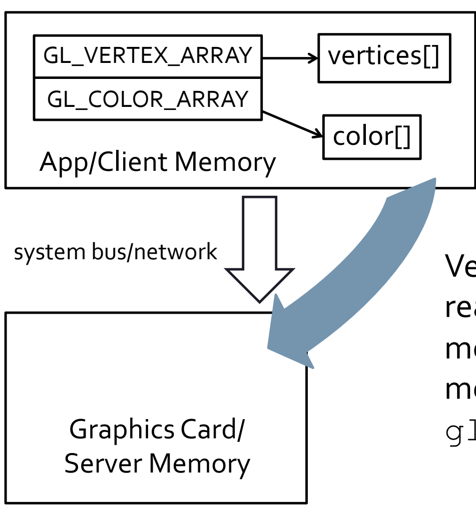


Vertex Arrays

Even with triangle strips, passing each vertex to OpenGL requires a separate function call

Vertex arrays allow for passing an array of vertices to OpenGL with a constant number of function calls
- store vertex data in triangle strip sequentially in application/client-side memory
- pass pointer to this memory to the API
- the API copies the data from memory to GPU/server

Vertex Array



Vertex attribute data read from client memory to server memory whenever `glDraw*()` is called

Vertex Arrays

Enable vertex array in app memory:

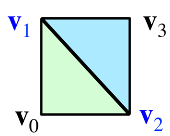
- `glEnableClientState(GL_VERTEX_ARRAY)`

Next, pass array to OpenGL, specifying data format:

- `glVertexPointer(…)`

Array can be accessed in three ways
(array must be in scope when calling these):

- randomly: `glArrayElement(…)`
- sequentially: `glDrawArrays(…)`, with multiple triangle strips, e.g., in a sphere, no vertex sharing across strips, duplicated vertices must be enumerated in array
- indexed: `glDrawElements(…)`, duplicated vertices listed once in array



Vertex Arrays Example

```
float vertices[]= { 1.0, 0.0, 0.0,
                    0.0, 2.0, 1.0,
                    0.0, 1.0, 0.0,
                    0.0, 0.0, 1.0 };
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// void glVertexPointer(Glint size, GLenum type,
//                      GLsizei stride, const GLvoid *pointer);
// size is the number of coordinates per vertex; type specifies the
// data type; stride is the byte offset between vertices (in bytes, e.g.,
// (3*sizeof(float) for 1-vertex stride), stride=0 if the vertices
// are packed back-to-back
// pointer points to array
```

(see http://www.songho.ca/opengl/gl_vertexarray.html for sample)

Li,Siu,Butler

Using the Array

With random access (in display list):

```
glBegin(GL_TRIANGLES);
  glArrayElement(1);
  glArrayElement(0);
  glArrayElement(2);
glEnd();
```

With indexed access (not between `glBegin/glEnd`):

```
unsigned char indices[]= { 1, 0, 2 };
glDrawElements(GL_TRIANGLES, 3,
               GL_UNSIGNED_BYTE, indices);
// void glDrawElements(GLenum mode, GLsizei count,
//                     GLenum type, void *indices);
// mode is connection type, count size of index array,
// type the data type of the index array, choose
// smallest representation necessary, indices the array of indices
```

Merrell

Other Vertex Attributes

Aside from specifying vertex coordinates, you can also specify the following attribute arrays, with 1-to-1 mapping to the vertex array

- `glNormalPointer()`: vertex normal array
- `glColorPointer()`: vertex RGB color array
- `glIndexPointer()`: indexed vertex color array
- `glTexCoordPointer()`: texture coordinates array
- `glEdgeFlagPointer()`: array indicating boundary vertices

Note: corresponding client-state must be enabled:
`GL_NORMAL_ARRAY`, `GL_COLOR_ARRAY`, etc.

Drawing Modes

Immediate drawing mode: graphics system does not store drawn primitives

- `glVertex()`: vertices streamed through to display as soon as specified
- vertex array: vertices copied from client state to graphics card/server as needed (`glDrawElements()` may cache vertices)

Often we want to draw the same object several times, perhaps transformed . . . inefficient to copy the same vertices multiple times, use retained drawing mode (display list, OpenGL 2.1) or vertex buffer object (OpenGL 3.0+)

Ramamooorthi

EECS 487: Interactive Computer Graphics

Lecture 12:

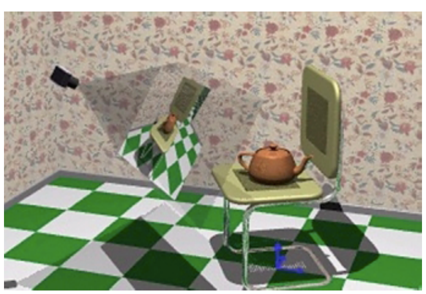
- Vertex passing: triangle strip/fan, vertex array, drawing modes
- From 3D to 2D Overview
- Viewing transform

Getting from 3D to 2D

Given a 3D scene, how do we get it onto a 2D screen?

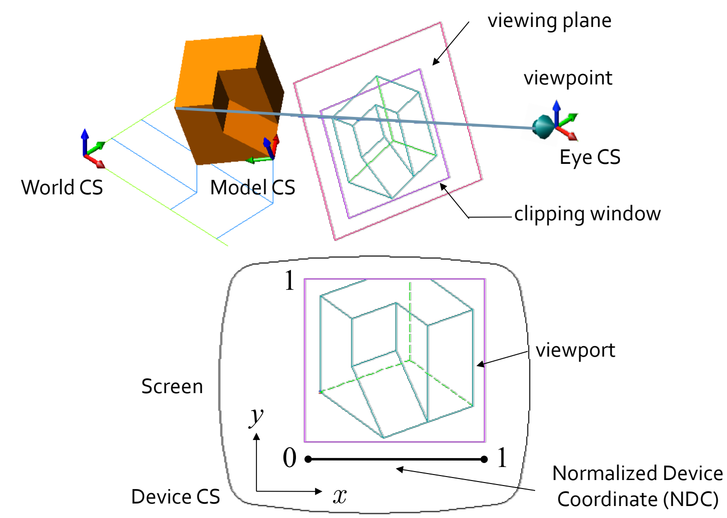
- specify 3D location of all points on the object
- map these points to locations on 2D device monitor

Various coordinate systems used to accomplish this



Lozano-Perez01

Coordinate System Relationships



Leake

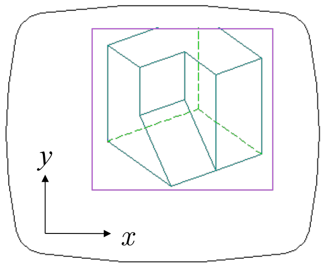
Device Coordinate

Image in clipping window mapped onto viewport area

Location of a pixel in a viewport is expressed in device coordinates (x, y)

Device coordinates are:

- integers
- resolution dependent



Leake

Normalized Device Coordinate

Normalized device coordinate (NDC):

- location of a pixel expressed in terms of percentages of image size
- resolution independent
- (z-values retained)

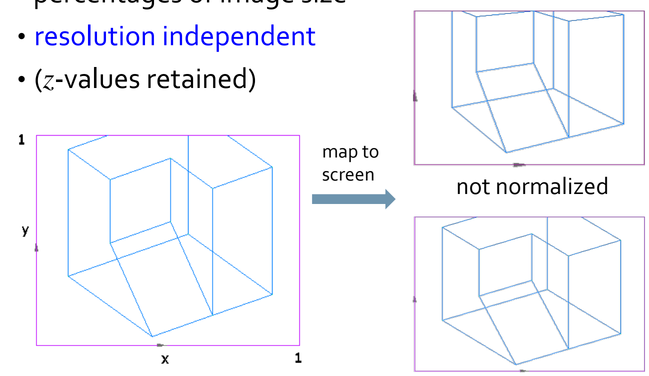


image clipped or occupies only a small region of screen

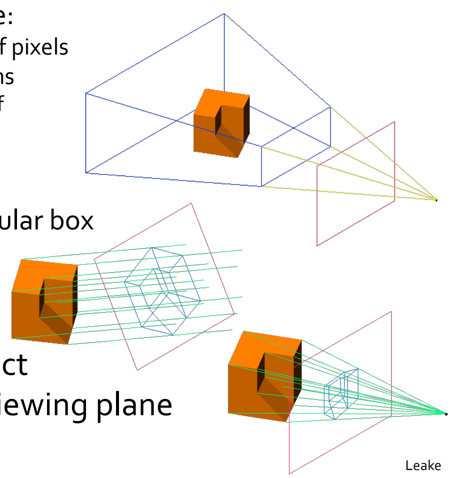
image stretched to fill screen

Leake

Projection Transforms

View volume: part of 3D scene we want to draw, constrained by front, back, and side clipping planes

- drawing window is of finite size:
  - we can only store a finite number of pixels
  - and a discrete, finite range of depths
    - like color, only have a fixed number of depth bits at each pixel
  - points too close or too far away will not be drawn
- for parallel projection: rectangular box
- for perspective projection: truncated pyramid (frustum)

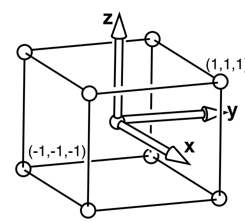


Projection transforms project the view volume onto the viewing plane

Leake

Canonical View Volume

A cube centered at the origin, aligned with the axes, spanning $(-1, -1, -1)$ to $(1,1,1)$



A 3D version of NDC

- decouples projection from window/screen sizes
- parallel sides and equal dimensions of the cvv make many operations, e.g., clipping, easier (cvv is thus a.k.a. clip coordinates)

From View Frustum to Screen Space

Perspective transform:

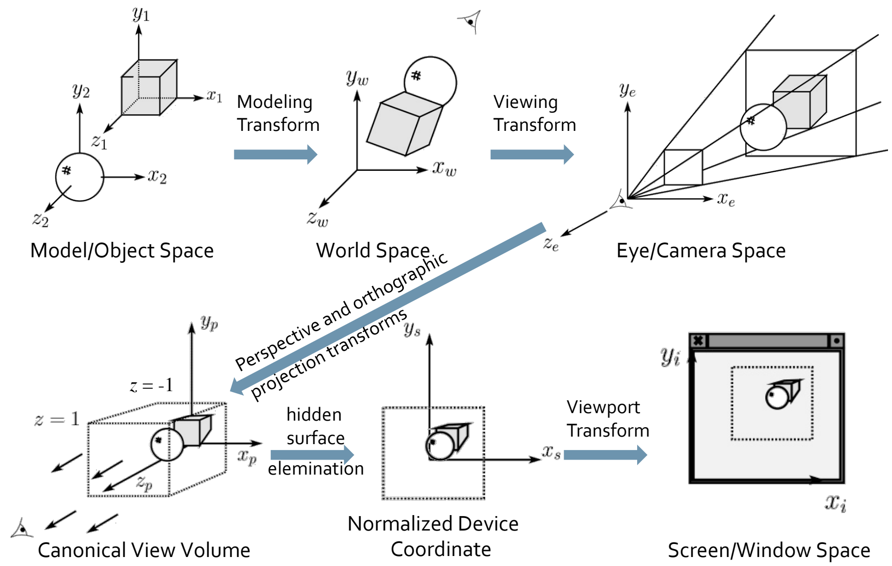
1. perspective project view frustum to orthographic space
2. orthographic/parallel project to canonical view volume
3. map canonical view volume to NDC for display

Viewport transform/screen mapping:

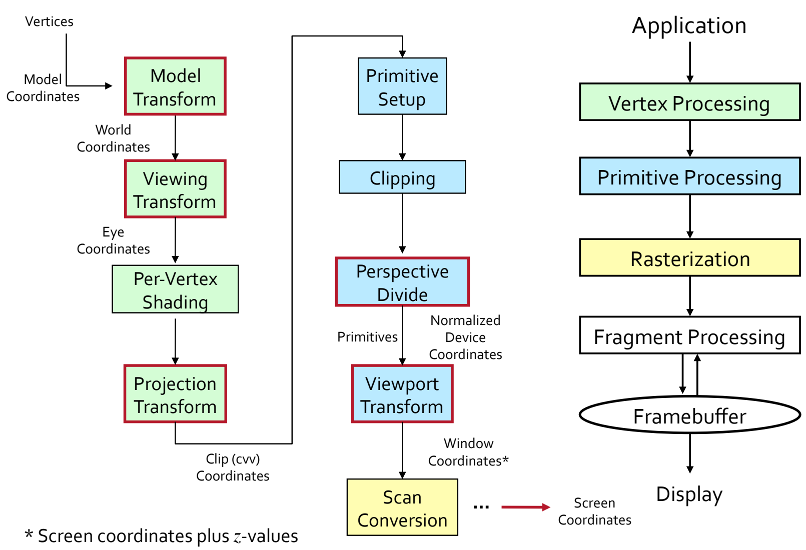
- map NDC to viewport

Illustrated in the next slide …

3D Geometric Pipeline

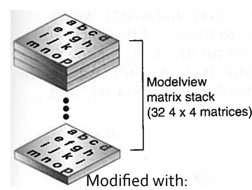


Where in the Pipeline?



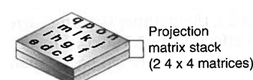
* Screen coordinates plus z-values

OpenGL States: CTMs



Modified with:

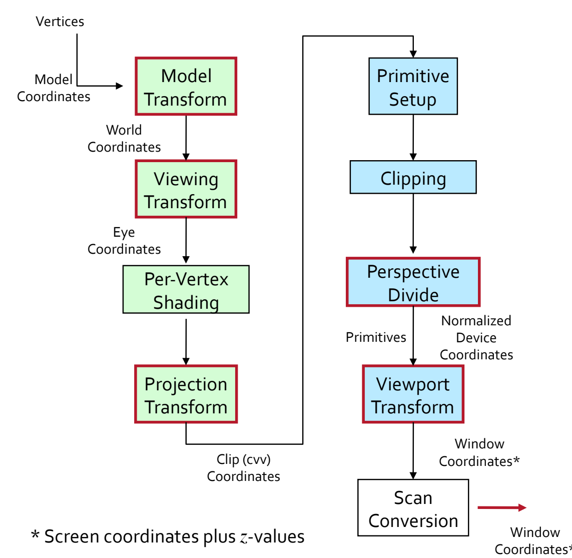
```
glLoadMatrix()
gluLookAt()
glRotate()
glTranslate()
glScale()
etc.
```



Modified with:

```
glLoadMatrix()
glFrustum()
glOrtho()
gluPerspective()
gluOrtho2D()
```

Where in the Pipeline?



* Screen coordinates plus z-values

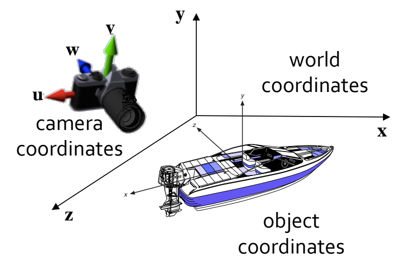
EECS 487: Interactive Computer Graphics

Lecture 12:

- Vertex passing: triangle strip/fan, vertex array, drawing modes
- From 3D to 2D Overview
- Viewing transform

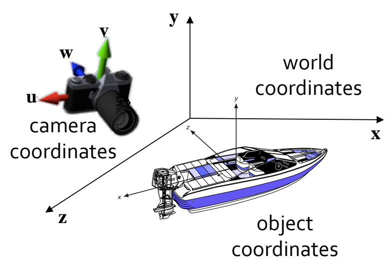
Viewing Transform

Transform the scene such that the camera is at the origin; simplifies projection, visibility and clipping determination, lighting



Viewing Transform

First construct a camera coordinate frame



What is a coordinate frame?

A set of 3 vectors ($\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$) and an origin $\mathbf{o}$ such that:

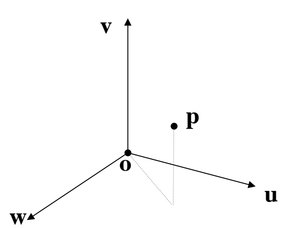
$$\|\mathbf{u}\| = \|\mathbf{v}\| = \|\mathbf{w}\| = 1$$

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{u} \cdot \mathbf{w} = 0$$

$$\mathbf{w} = \mathbf{u} \times \mathbf{v}$$

and for any point $\mathbf{p}$:

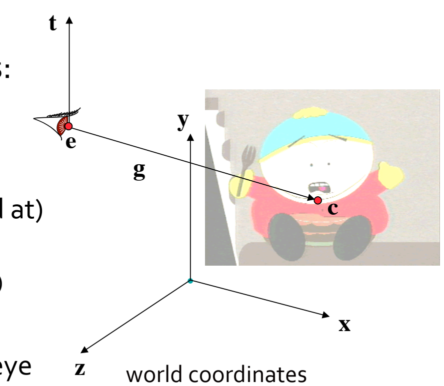
$$\mathbf{p} = \mathbf{o} + (\mathbf{p} \cdot \mathbf{u})\mathbf{u} + (\mathbf{p} \cdot \mathbf{v})\mathbf{v} + (\mathbf{p} \cdot \mathbf{w})\mathbf{w}$$



Camera Coordinate Frame

Given, in world coordinates:

- camera/eye location ($\mathbf{e}$)
- lookat point ($\mathbf{c}$), where camera is pointed to (centered at)
  - or view/gaze direction ($\mathbf{g}$): line from eye to lookat point ($\mathbf{c} - \mathbf{e}$)
- and an up-vector ($\mathbf{t}$): a vector pointing up from the camera/eye



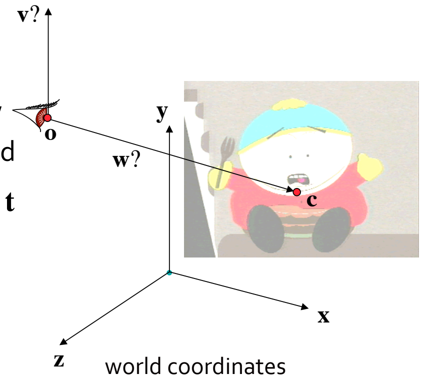
How do we construct a camera coordinate frame?

Harto8

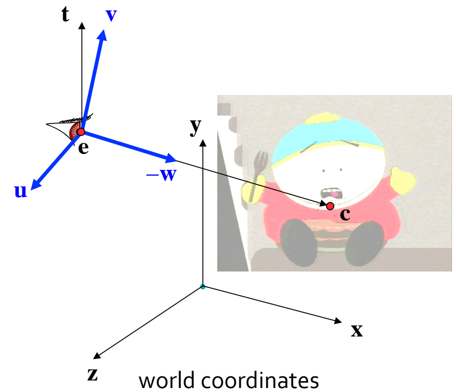
Camera Coordinate Frame

Given $\mathbf{e}$, $\mathbf{c}$, and $\mathbf{t}$ of the camera in world coordinates,

- we can use $\mathbf{e}$ as the origin ($\mathbf{o}$), and
- would like to use $\mathbf{g}$ as $\mathbf{w}$ and $\mathbf{v}$ as $\mathbf{t}$
  - but $\mathbf{g}$ and $\mathbf{t}$ are neither orthogonal nor of unit length!
- furthermore, we need a $\mathbf{u}$



Camera Coordinate Frame



$\mathbf{w} = -(\mathbf{c} - \mathbf{e})/||(\mathbf{c} - \mathbf{e})||$

$\mathbf{u} = (\mathbf{t} \times \mathbf{w})/||(\mathbf{t} \times \mathbf{w})||$

$\mathbf{v} = \mathbf{w} \times \mathbf{u}$

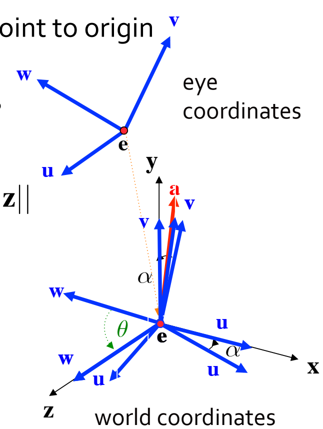
(Called Gram-Schmidt Orthonormalization)

Viewing Transform Implementation

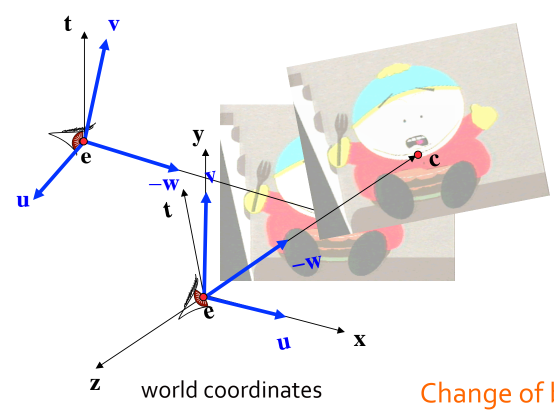
Given $\mathbf{e}$, $\mathbf{c}$, $\mathbf{t}$, and $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ as computed, how do we transform from world to eye coords?

1. Translate by $-\mathbf{e}$, bring camera's viewpoint to origin
2. Rotate $\mathbf{w}$ to the z-axis ($\mathbf{z} = [0,0,1]$):
   - what are the rotation axis and angle?
   - axis: $\mathbf{a} = (\mathbf{w} \times \mathbf{z}) / ||\mathbf{w} \times \mathbf{z}||$
   - angle: $\cos\theta = \mathbf{w} \cdot \mathbf{z}$ and $\sin\theta = ||\mathbf{w} \times \mathbf{z}||$
   - `glRotate(θ,ax,ay,az)`
   - $\mathbf{u}$ and $\mathbf{v}$ are now on the x-y plane
3. Rotate $\mathbf{v}$ by α about the z-axis, to get $\mathbf{v}$ parallel to the y-axis and $\mathbf{u}$ // to $\mathbf{x}$

$\mathbf{p}' = \mathbf{V}\mathbf{p} = \mathbf{R}(\alpha, \mathbf{z})\mathbf{R}(\theta, \mathbf{a})\mathbf{T}(-\mathbf{e})\mathbf{p}$



Viewing Transform



Change of basis!

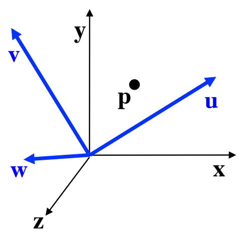
Is there a simpler way than to compute $\mathbf{p}' = \mathbf{V}\mathbf{p} = \mathbf{R}(\alpha, \mathbf{z})\mathbf{R}(\theta, \mathbf{a})\mathbf{T}(-\mathbf{e})\mathbf{p}$?

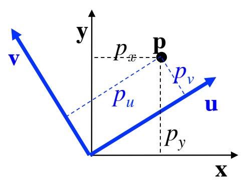
Change of Orthonormal Basis

Given coordinate frames **xyz** (world) and **uvw** (eye) and point $\mathbf{p} = (p_x, p_y, p_z)$

Find: $\mathbf{p} = (p_u, p_v, p_w)$ by transforming the coordinate frame

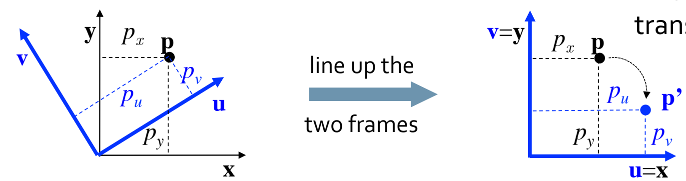
(Easier to visualize in 2D!)





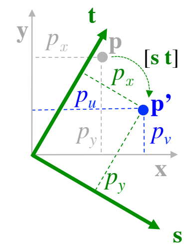
Durand06

Change of Orthonormal Basis



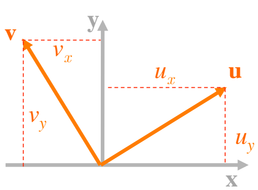
what kind of transform?

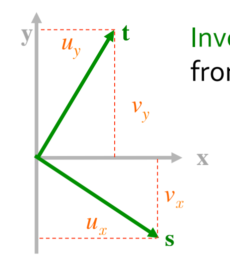
xy: world coordinate frame
uv: eye coordinate frame
(assume $\mathbf{e} = \mathbf{o}$)



inverse transform

Change of Orthonormal Basis





Inverse transformation from **x**, **y**, **z** to **u**, **v**, **w**

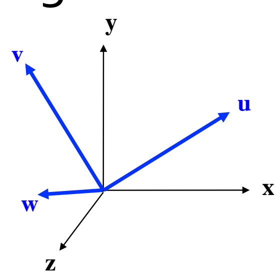
column space: [**u v w**]

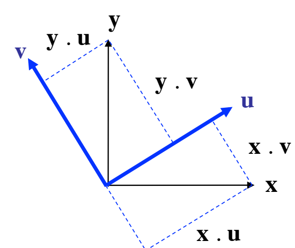
$$\begin{matrix} \mathbf{s} \\ \mathbf{t} \\ \ \end{matrix} \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix}$$

row space: [**s t r**]

$$\begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix}$$

Change of Orthonormal Basis (Algebraic Check)





Expressing the **x**, **y**, **z** (world) bases in terms of **u**, **v**, **w** (eye) (coordinates are length of projections!):

$$\mathbf{x} = (\mathbf{x} \cdot \mathbf{u})\,\mathbf{u} + (\mathbf{x} \cdot \mathbf{v})\,\mathbf{v} + (\mathbf{x} \cdot \mathbf{w})\,\mathbf{w}$$
$$\mathbf{y} = (\mathbf{y} \cdot \mathbf{u})\,\mathbf{u} + (\mathbf{y} \cdot \mathbf{v})\,\mathbf{v} + (\mathbf{y} \cdot \mathbf{w})\,\mathbf{w}$$
$$\mathbf{z} = (\mathbf{z} \cdot \mathbf{u})\,\mathbf{u} + (\mathbf{z} \cdot \mathbf{v})\,\mathbf{v} + (\mathbf{z} \cdot \mathbf{w})\,\mathbf{w}$$

Durand06

Change of Orthonormal Basis

$$\mathbf{x} = (\mathbf{x}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{x}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{x}\cdot\mathbf{w})\,\mathbf{w}$$
$$\mathbf{y} = (\mathbf{y}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{y}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{y}\cdot\mathbf{w})\,\mathbf{w}$$
$$\mathbf{z} = (\mathbf{z}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{z}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{z}\cdot\mathbf{w})\,\mathbf{w}$$

Substitute into equation for **p**:

$$\mathbf{p} = (p_x, p_y, p_z) = p_x\,\mathbf{x} + p_y\,\mathbf{y} + p_z\,\mathbf{z}$$

$$\begin{aligned}\mathbf{p} = {} & p_x\,[\,(\mathbf{x}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{x}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{x}\cdot\mathbf{w})\,\mathbf{w}\,] + \\ & p_y\,[\,(\mathbf{y}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{y}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{y}\cdot\mathbf{w})\,\mathbf{w}\,] + \\ & p_z\,[\,(\mathbf{z}\cdot\mathbf{u})\,\mathbf{u} + (\mathbf{z}\cdot\mathbf{v})\,\mathbf{v} + (\mathbf{z}\cdot\mathbf{w})\,\mathbf{w}\,]\end{aligned}$$

Rewrite:

$$\begin{aligned}\mathbf{p} = {} & [\; p_x(\mathbf{x}\,.\,\mathbf{u}) + p_y(\mathbf{y}\,.\,\mathbf{u}) + p_z(\mathbf{z}\,.\,\mathbf{u}) \;]\,\mathbf{u} + \\ & [\; p_x(\mathbf{x}\,.\,\mathbf{v}) + p_y(\mathbf{y}\,.\,\mathbf{v}) + p_z(\mathbf{z}\,.\,\mathbf{v}) \;]\,\mathbf{v} + \\ & [\; p_x(\mathbf{x}\,.\,\mathbf{w}) + p_y(\mathbf{y}\,.\,\mathbf{w}) + p_z(\mathbf{z}\,.\,\mathbf{w}) \;]\,\mathbf{w}\end{aligned}$$

Durand06

Change of Orthonormal Basis

$$\begin{aligned}\mathbf{p} = {} & [\; p_x(\mathbf{x}\,.\,\mathbf{u}) + p_y(\mathbf{y}\,.\,\mathbf{u}) + p_z(\mathbf{z}\,.\,\mathbf{u}) \;]\,\mathbf{u} + \\ & [\; p_x(\mathbf{x}\,.\,\mathbf{v}) + p_y(\mathbf{y}\,.\,\mathbf{v}) + p_z(\mathbf{z}\,.\,\mathbf{v}) \;]\,\mathbf{v} + \\ & [\; p_x(\mathbf{x}\,.\,\mathbf{w}) + p_y(\mathbf{y}\,.\,\mathbf{w}) + p_z(\mathbf{z}\,.\,\mathbf{w}) \;]\,\mathbf{w}\end{aligned}$$

$$\mathbf{p} = (p_u, p_v, p_w) = p_u\,\mathbf{u} + p_v\,\mathbf{v} + p_w\,\mathbf{w}$$

Expressed in **uvw** (eye) basis:

$$p_u = p_x(\mathbf{x}\,.\,\mathbf{u}) + p_y(\mathbf{y}\,.\,\mathbf{u}) + p_z(\mathbf{z}\,.\,\mathbf{u})$$
$$p_v = p_x(\mathbf{x}\,.\,\mathbf{v}) + p_y(\mathbf{y}\,.\,\mathbf{v}) + p_z(\mathbf{z}\,.\,\mathbf{v})$$
$$p_w = p_x(\mathbf{x}\,.\,\mathbf{w}) + p_y(\mathbf{y}\,.\,\mathbf{w}) + p_z(\mathbf{z}\,.\,\mathbf{w})$$

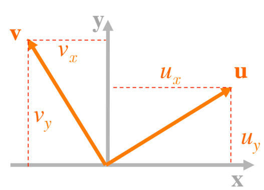
In matrix form:

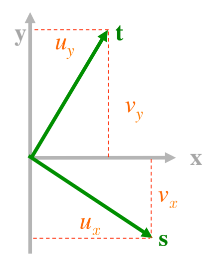
$$\begin{pmatrix} p_u \\ p_v \\ p_w \end{pmatrix} = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

where:
$x_u = \mathbf{x}\cdot\mathbf{u}$
$y_u = \mathbf{y}\cdot\mathbf{u}$
etc.

Durand06

Change of Orthonormal Basis





Inverse transform
from **x, y, z**
to **u, v, w**

$u_x = \mathbf{u}\cdot\mathbf{x}$
$\quad = \mathbf{x}\cdot\mathbf{u} = x_u$

column space: [**u v w**] row space: [**s t r**]

$$\begin{matrix}\mathbf{s}\\ \mathbf{t}\\ \ \end{matrix}\begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} \begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix} = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix}$$

Change of Basis to Eye Coordinate Frame

Translate to eye and transformed to **u, v, w** basis:

$$\mathbf{M}_{world\to eye} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -u_x e_x - u_y e_y - u_z e_z \\ v_x & v_y & v_z & -v_x e_x - v_y e_y - v_z e_z \\ w_x & w_y & w_z & -w_x e_x - w_y e_y - w_z e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And to go all the way from world to cvv:

$$\mathbf{M}_{world\to canonical} = \mathbf{M}_{eye\to canonical}\,\mathbf{M}_{world\to eye}$$

viewing transform

projection transform

$$\mathbf{p}_{canonical} = \mathbf{M}_{world\to canonical}\,\mathbf{p}_{world}$$

Chenney

Viewing Transform in OpenGL 2.1

Position the camera/eye in the scene

```
gluLookAt(eyeX,eyeY,eyeZ,
          centerX,centerY,centerZ,
          upX,upY,upZ);
```

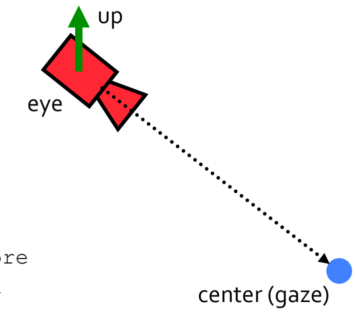
To "fly through" a scene, change viewing transform and redraw scene

- moving camera is equivalent to moving every object in the world relative to a stationary camera

`gluLookAt()` operates on the ModelView matrix just like any modeling transform:

- must come "before in code, after in action" to other transforms

Viewing Transforms



Example:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// viewing transform, comes before
// model transform, applied last
gluLookAt(0.0,0.0,5.0,  // camera at (0,0,5)
          0.0,0.0,0.0,  // gazing at (0,0,0)
          0.0,1.0,0.0); // up is y-axis

// model transform
glRotated(-20.0, 0.0,1.0,0.0);

// then draw
glutSolidTeapot(1.0);
```