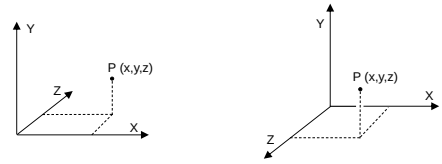
CSc 165
Computer Game Architecture

3 - Fundamentals of 3D Systems

3D Coordinate Systems

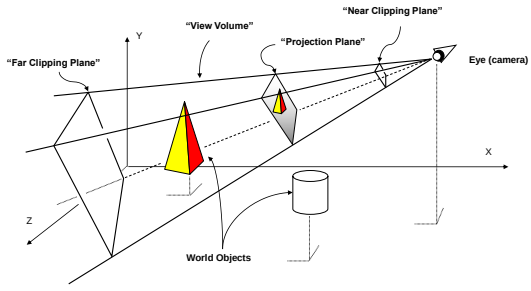


Left-handed Coordinate System **Right-handed Coordinate System**

Points can be represented in *homogeneous form:*

P = [x y z 1]

"Synthetic Camera" Paradigm

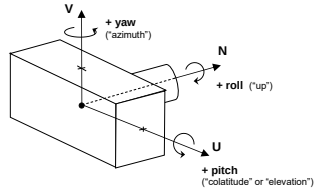


The "UVN" Camera

Two important camera attributes:

- *Location*
- *Orientation of* ***UVN*** *axes*

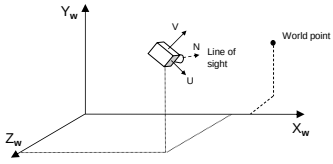
  Note the UVN coordinate system is *left-handed*



Generalized Camera Control

Player controls position & orientation

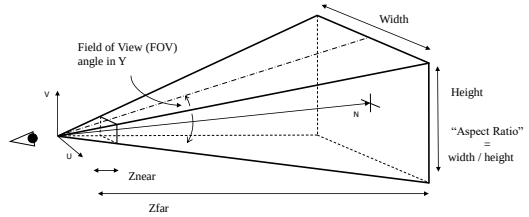
- "World" points must be converted to "camera" points
- Game *engine* should handle this (it's game-independent)



Additional Camera Settings

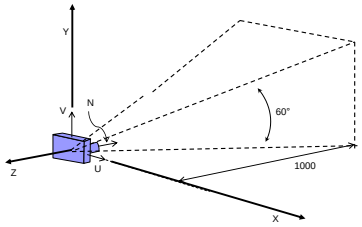
FOVY, **Aspect**, **Near** & **Far** (Clipping)

- Controls "projection" onto 2D plane (& screen)
- Again, game *engine* should handle details

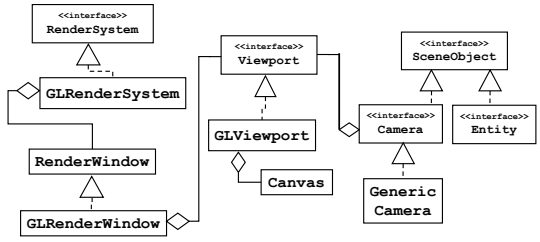


Default Camera Values

- Loc = [0 0 0], looking down *negative* Z
- V = Y, U = X, N = -Z
- fovY = 60°, aspect=1, near=0.01, far=1000



RAGE Camera/Display Class Structure

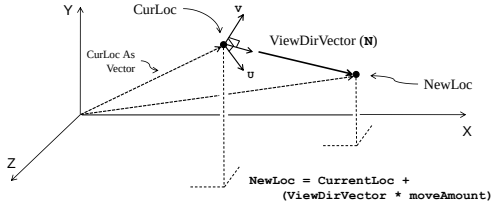


RAGE's Camera Interface

```
//This interface defines the functions provided by all camera implementations.
//Cameras operate in either "camera" or "node" mode - meaning they get their
// location/orientation either internally or from the node they are attached to.
public interface Camera
{   // get the camera's location/orientation
    public Vector3f getPo();
    public Vector3f getRt();
    public Vector3f getUp();
    public Vector3f getFd();
    public char getMode();  // 'c'=camera, 'n'=node

    //modify the camera's location/orientation (note that it is the user's
    //responsibility to insure the camera axes remain mutually perpendicular)
    public void setPo(Vector3f v);
    public void setRt(Vector3f v);
    public void setUp(Vector3f v);
    public void setFd(Vector3f v);
    public void setMode(char m);  // 'c'=camera, 'n'=node
    public void setHUD(string h);

    public Frustum getFrustum();  // frustum is set at time of construction.
    public void renderScene();    // (not called by client game)

    //... other methods to be seen later ...
}
```

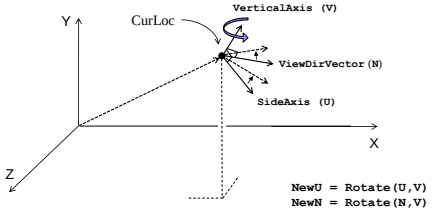

Camera Manipulation

example: "Move Forward" ==
change <u>location</u> along the *<u>view direction</u>*

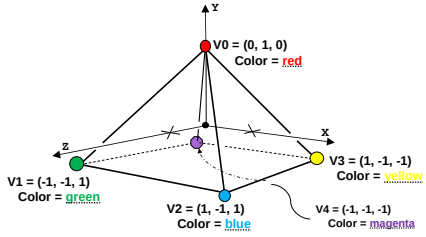


Camera Manipulation

example: "RotateLeft" == *(yaw left)*
rotate <u>U and N</u> around the *<u>V</u> axis*

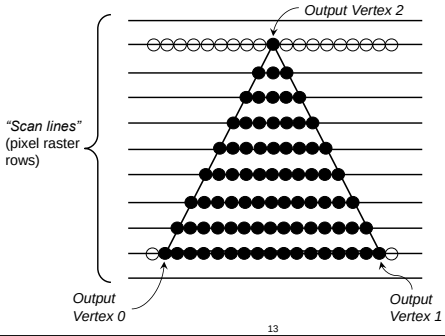


Defining Simple 3D Models

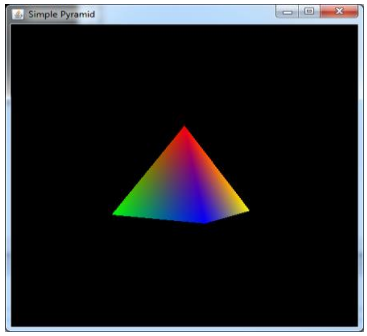


A 2x2x2 "**Pyramid**" Centered At The Origin

Rasterization



Interpolation Result



Pyramid Data Structure
(non-indexed)

Vertices

	X	Y	Z
0	0	1	0
1	-1	-1	1
2	1	-1	1
3	0	1	0
4	1	-1	-1

etc…

one "vertex"

Geometry

Triangles

0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14
5	15	16	17

one "vertex index"

Indexes

Pyramid Data Structure
(indexed)

Vertices

	X	Y	Z
0	0	1	0
1	-1	-1	1
2	1	-1	1
3	1	-1	-1
4	-1	-1	-1

one "vertex"

Geometry

Triangles

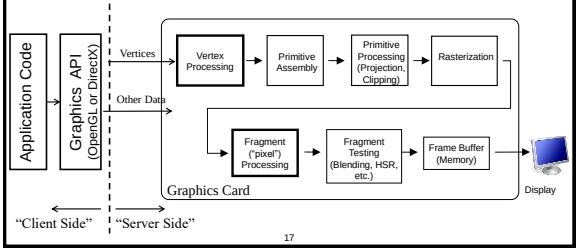
0	0	1	2
1	0	2	3
2	0	3	4
3	0	4	1
4	1	4	2
5	4	3	2

one "vertex index"

Indexes

The Graphics "Pipeline"

- Implemented by the *combination of the graphics driver (software),* and *graphics (hardware) card*
- Can be FIXED-FUNCTION or PROGRAMMABLE, depending on implementation *version*



3D Transformations

Needed for a wide variety of operations:

- Modeling
- Positioning & orienting objects in the "3D virtual world"
- Camera positioning ("viewing")
- Creating the 2D screen view of the 3D world view ("projection")
- Making objects move, grow, spin, fly, etc.

Translation (column-major form):

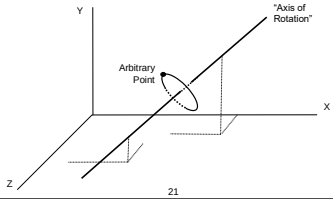
$$\begin{pmatrix} (x+T_x) \\ (y+T_y) \\ (z+T_z) \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Scaling (column-major form):

$$\begin{pmatrix} (x*S_x) \\ (y*S_y) \\ (z*S_z) \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3D Rotation

- **Recall 2D rotations can be "about any point"**
  - For simplicity we define only 2D rotation "about the origin"
  - Other rotations require translation to/from the origin
- **Similarly, 3D rotations can be "about any line" (any "axis of rotation") :**



Euler's Theorem

"Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point." [Leonard Euler, 1707-1783]

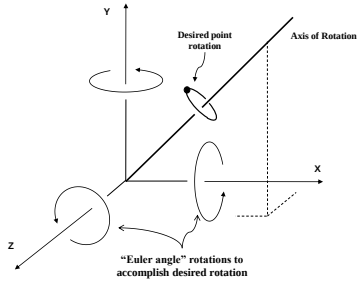
This is equivalent to saying:

Rotation about an arbitrary line through the origin can be accomplished by an equivalent set of rotations about the X, Y, and Z axes.

Thus we can rotate about an arbitrary axis as follows:

1. **Translate the axis so it goes through the origin,**
2. **Rotate by the appropriate "Euler angles" about X, Y, and Z, and**
3. **"Undo" the translation**

Visualizing Euler's Theorem



3D Rotation Transforms

Rotation about X by θ:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

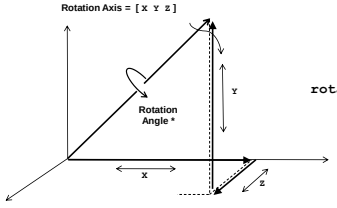
Rotation about Y by θ:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation about Z by θ:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation in Angle/Axis Form



`rotate(60, x, y, z)`

* **Positive rotation** = CCW *as seen from vector (axis) head, looking toward tail at origin (right hand rule)*

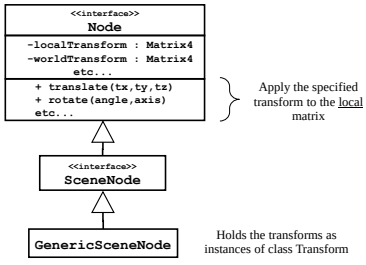
Representing Transforms

Package **`rml`** *("RAGE Math Library")*

- Class **`Matrix4`** : a 4x4 ("3D") matrix
  - Methods for specifying translation, rotation, & scaling, obtaining transpose and inverse, etc.
  - Similar to Java's **`AffineTransform`** (but 3D)
- Class **`Vector4`** : a 4-element ("3D") vector
  - Methods for most common vector operations: add, dot- and cross-product, magnitude, normalize…
  - Useful for representing, for example, a ***rotation axis***

SceneNode Hierarchy

Every object in a scene is an instance of **`SceneNode`**, which is a **`ray.rage.scene.Node`**, which provides translate, rotate, and scale matrices, or combined into a single "transform" matrix.



"Perspective" matrix

```
q = 1 / tan(fieldOfView/2);
A = q / aspectRatio;
B = (near + far) / (near - far);
C = (2.0 * near * far) / (near - far);
```

The perspective transformation matrix is then:

$$\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Lighting

Real world lights have a *frequency spectrum*

- White light: all (visible) frequencies
- Colored light: restricted frequency distribution

Simplified model:

Light "characteristics"

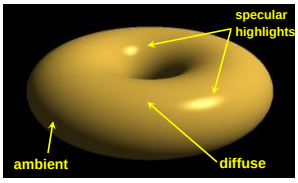
- Ambient, Diffuse, Specular "reflection characteristics"
- Red, Green, Blue "intensities"

Light "type"

- Positional, Directional, …

The "ADS" lighting model

- Ambient reflection simulates a low-level illumination that equally affects everything in the scene.
- Diffuse reflection brightens objects to various degree depending on the light's angle of incidence.
- Specular reflection conveys the shininess of an object by strategically placing a highlight of appropriate size on the object's surface where light is reflected most directly towards our eyes.



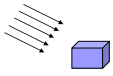
Light Types

Point source

- Location, intensity

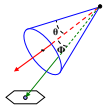
Directional ("distant")

- Direction, intensity

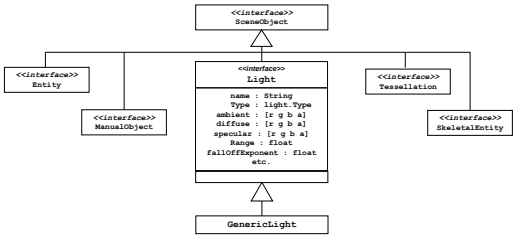


Spot

- Location, direction, intensity, coneAngle, fallOffRate



RAGE Light Classes



Materials

Models the reflectance characteristics of surfaces. Usually modeled in ADS with four components:

- Ambient, Diffuse, and Specular
- Shininess (to determine size of specular highlights)

some common materials

material	*ambient RGBA* *diffuse RGBA* *specular RGBA*	*shininess*
Gold	0.24725, 0.1995, 0.0745, 1.0 0.75164, 0.60648, 0.22648, 1.0 0.62828, 0.5558, 0.36607, 1.0	51.2
Jade	0.135, 0.2225, 0.1575, 0.95 0.54, 0.89, 0.63, 0.95 0.3162, 0.3162, 0.3162, 0.95	12.8
Pearl	0.25, 0.20725, 0.20725, 0.922 1.00, 0.829, 0.829, 0.922 0.2966, 0.2966, 0.2966, 0.922	11.264
Silver	0.19225, 0.19225, 0.19225, 1.0 0.50754, 0.50754, 0.50754, 1.0 0.50827, 0.50827, 0.50827, 1.0	51.2

Barradeu, N., http://www.barradeau.com/nicoptere/dump/materials.html

ADS lighting computations

$$I_{observed} = I_{ambient} + I_{diffuse} + I_{specular}$$

Ambient computation is the simplest:

$$I_{ambient} = Light_{ambient} * Material_{ambient}$$

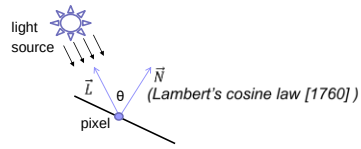
Note that each item has R, G, and B components. So the computations actually are as follows:

$$I^{red}_{ambient} = Light^{red}_{ambient} * Material^{red}_{ambient}$$

$$I^{green}_{ambient} = Light^{green}_{ambient} * Material^{green}_{ambient}$$

$$I^{blue}_{ambient} = Light^{blue}_{ambient} * Material^{blue}_{ambient}$$

Diffuse computation depends on the angle of incidence between the light and the surface:



(Lambert's cosine law [1760])

$$I_{diffuse} = Light_{diffuse} * Material_{diffuse} * cos(\theta)$$

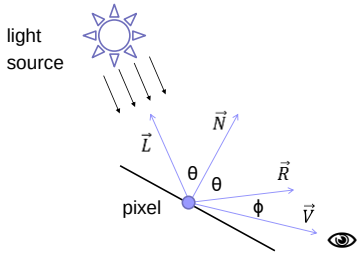
Rightmost term determined simply using dot product:

$$I_{diffuse} = Light_{diffuse} * Material_{diffuse} * (\hat{N} \bullet \hat{L})$$

Only include this term if the surface is exposed to the light:

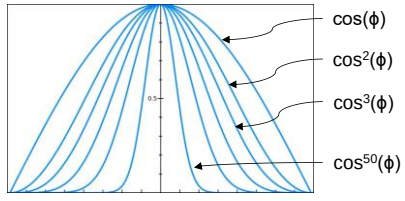
$$I_{diffuse} = Light_{diffuse} * Material_{diffuse} * \max((\hat{N} \bullet \hat{L}), 0)$$

Specular computation depends on the angle of reflection of the light on the surface, and the viewing angle of the eye.



"Shininess" modeled with a falloff function.

Expresses how quickly the specular contribution reduces to zero as the angle ϕ grows.



$$I_{spec} = Light_{spec} * Material_{spec} * \max(0, (\hat{R} \bullet \hat{V})^{n})$$