The 3D Graphics Rendering Pipeline

Animação e Visualização Tridimensional

João Madeiras Pereira
Which Visualization?

Interactive Real-Time Rendering

- 3D scenes
- Realism
- Real-Time
- Interaction

Application paradigm: games
The process of creating a 2-D Image from a 3-D model (scene) is called “rendering.” The rendering pipeline has three functional stages. The speed of the pipeline is that of its slowest stage.
Graphics API vs. Application API

- **Graphics API**
  - Support *rendering pipeline for polygon stream*
  - Supported by graphics hardware
  - **Examples**
    - OpenGL, Direct 3D

- **Application API**
  - A *software engine that calls graphics API for rendering polygons in a scene*
  - Requires scene management, real-time rendering modules, animation modules, etc.
  - **Home made programs vs. programming using application API**
    - Either one will call graphics API for rendering polygons
  - **Examples**
    - OpenInventer (visual simulation), WTK (VR engine), Unreal3 (game engine)....
Low level rendering pipeline

- Set up a polygonal mesh for the scene
- Culling back-facing polygons
  (in world coord. sys., or screen coord. sys. (prefer))
- 3D-to-2D projection and clipping
  - Apply viewing transformation
  - Compute lighting
  - Apply projection
  - Clip polygons against view volume
    (in 3D screen space or homogeneous space)
- Rasterization
  - Scan convert polygons
  - Apply hidden surface removal
  - Shade pixels by incremental shading methods
  - Texture accessing

©2014/2015, AVT
OpenGL rendering pipeline(s)
The Graphics Rendering Pipeline

Old rendering pipelines were done in software (slow) Modern pipeline architecture uses parallelism and buffers. The application stage is implemented in software, while the other stages are *hardware-accelerated*.
The Rendering Pipeline

Application → Geometry → Rasterizer
The application stage

✓ Is done entirely in software by the CPU;
✓ It reads Input devices (such as gloves, mouse);
✓ It changes the coordinates of the virtual camera;
✓ It performs collision detection and collision response (based on object properties) for haptics;
✓ One form of collision response if force feedback.
Application stage optimization...

✓ Reduce model complexity (models with less polygons – less to feed down the pipe);

Low res. Model
~ 600 polygons

Higher resolution model
134,754 polygons.
Application stage optimization...

✓ Reduce floating point precision (single precision instead of double precision)
✓ minimize number of divisions
✓ Since all is done by the CPU, to increase speed a dual-processor (super-scalar) architecture is recommended.
Computing Architectures

The Rendering Pipeline

Application → Geometry → Rasterizer

©2014/2015, AVT

Rendering pipeline
Detail Steps

• Set up polygonal meshes for the scene
• 3D-to-2D projection and lighting stage
  – By polygon basis
  – Apply viewing transformation to transform polygon to view coordinate system
  – Compute lighting (Phong model) at each vertex
  – Apply projection transformation to transform polygon to device normalized coordinates
  – View-volume clipping
    • Clipped and transformed into screen coordinates
The Graphics Rendering Pipeline (revisited)

Application → Geometry → Rasterizer

The Geometry Functional Sub-Stages

- Model & View Transformation
- Lighting
- Projection
- Clipping
- Screen Mapping
The Rendering Pipeline

Application → Geometry → Rasterizer

The Geometry Functional Sub-Stages

Model & View Transformation → Lighting → Projection → Clipping → Screen Mapping

©2014/2015, AVT
Model and Viewing Transformations:

✓ Model transforms link object coordinates to world coordinates. By changing the model transform, the same object can appear several times in the scene.

We call these *instances*.

\[
\begin{bmatrix}
1 & 0 & 0 & p_{1x}(t) \\
0 & 1 & 0 & p_{1y}(t) \\
0 & 0 & 1 & p_{1z}(t) \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Virtual ball

\[
\begin{bmatrix}
1 & 0 & 0 & p_{5x}(t) \\
0 & 1 & 0 & p_{5y}(t) \\
0 & 0 & 1 & p_{5z}(t) \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Instance 2 of Virtual ball

Instance 5

World system or coordinate
Viewing Transformation

✓ The **Viewing Transform** transform objects from World Coord. to Camera coord. (eye space)

✓ Its matrix captures the position and orientation of the virtual camera in the virtual world;

✓ The camera is located at the origin of the camera coordinate system, looking in the negative Z axis, with Y pointing upwards, and X to the right.

✓ Normalized vector \( \mathbf{n} = \text{lookAt} - \text{CamPos} \)

✓ Right-handed frame in OGL but \( \mathbf{n} = -\mathbf{z_v} \)
Viewing Transform

- Default (and initial) VRC left-handed frame:
  - Basis vectors $u = [1 \ 0 \ 0]$; $v = [0 \ 1 \ 0]$ and $n = [0 \ 0 \ -1]$
  - Origin: VRP

- Moving the camera implies to **orient** it and then **translate** it to the new VRP ($v_{rp_x} \ v_{rp_y} \ v_{rp_z}$):
  $$T[VRP] \bullet R.$$

- Two ways of understanding this transformation:
  - One simple frame (WCS): moving the virtual camera is equivalent to transform the objects by using the inverse camera transformation:
    $$(T[VRP] \bullet R)^{-1}, \Leftrightarrow R^{-1} \bullet T[-VRP]$$
  - Align both frames: First, we perform a translation $T[-VRP]$ and then a rotation. Corresponde em efectuar uma translação seguida de uma rotação

- Viewing Transform: $M_{view} = R_{rot} \bullet T_{trans}$
- $T_{trans} = T[-VRP]$
- And **rotation**???
Calculating the camera rotation

- Each column of a rotation matrix represents the axis of the resulting frame.

- The camera frame is initially aligned with World frame, except that it is pointing in the negative camera z-direction.

\[
u: [1 \ 0 \ 0] \rightarrow [u_x \ u_y \ u_z]
\]

\[
v: [0 \ 1 \ 0] \rightarrow [v_x \ v_y \ v_z]
\]

\[
-n: [0 \ 0 \ 1] \rightarrow [-n_x \ -n_y \ -n_z]
\]

\[
R_{\text{Camera}} = \begin{bmatrix}
u_x & v_x & -n_x & 0 \\
u_y & v_y & -n_y & 0 \\
u_z & v_z & -n_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Object rotation is \((R_{\text{Camera}})^{-1}\). Rotation matrix is orthogonal; so inverting it means to calculate its transpose. **The rows of the resulting matrix are the vectors** \(u, v\) **and** \(-n\).
Viewing Transformation

\[ R_{\text{rot}} = \begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
 -n_x & -n_y & -n_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \quad T_{\text{trans}} = \begin{bmatrix}
  1 & 0 & 0 & -\text{VRP}_x \\
  0 & 1 & 0 & -\text{VRP}_y \\
  0 & 0 & 1 & -\text{VRP}_z \\
  0 & 0 & 0 & 1
\end{bmatrix} \]

\[ M_{\text{vis}} = R_{\text{rot}} \bullet T_{\text{trans}} \]

\[ M_{\text{vis}} = \begin{bmatrix}
  u_x & u_y & u_z & -u \bullet \text{VRP} \\
  v_x & v_y & v_z & -v \bullet \text{VRP} \\
 -n_x & -n_y & -n_z & n \bullet \text{VRP} \\
  0 & 0 & 0 & 1
\end{bmatrix} \]
Back-Faces Culling

- In World coord. sys or Camera coord. sys (prefer)

- Based on dot product between face normal and viewing vector
The lighting sub-stage

✓ It calculates the vertex color based on:
  ▪ type and number of simulated light sources;
  ▪ the lighting model;
  ▪ the surface material properties;
  ▪ atmospheric effects such as fog or smoke.
✓ Lighting results in object shading which makes the scene more realistic.
Computing architectures

\[ I_\lambda = I_{a\lambda} K_a O_{d\lambda} + \]
\[ f_{\text{att}} I_{p\lambda} [K_d O_{d\lambda} \cos \theta + K_s O_{s\lambda} \cos^n \alpha] \]

where:
- \( I_\lambda \) is the intensity of light of wavelength \( \lambda \);
- \( I_{a\lambda} \) is the intensity of ambient light;
- \( K_a \) is the surface ambient reflection coefficient;
- \( O_{d\lambda} \) is the object diffuse color;
- \( f_{\text{att}} \) is the atmospheric attenuation factor;
- \( I_{p\lambda} \) is the intensity of point light source of wavelength \( \lambda \);
- \( K_d \) is the diffuse reflection coefficient;
- \( K_s \) is the specular reflection coefficient;
- \( O_{s\lambda} \) is the specular color;
Phong Lightning Model

\[ I = k_a L_a + k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max((\mathbf{r} \cdot \mathbf{v})^\alpha, 0) \]

One light source
Phong Lightning Model

• Several \((i)\) Light Sources
  – Specular Component

\[
I_s = k_s \sum_{i} L_{is} (\vec{r}_i \cdot \vec{v})^\alpha
\]

– Diffuse Component

\[
I_d = k_d \sum_{i} L_{id} (\vec{l}_i \cdot \vec{n})
\]
Phong Lightning Model
Atmosferic Attenuation

• Intensity varies with distance between surface and light source

• Given by:

\[ I = \frac{L}{a + bd + cd^2} \]

• Where
  – \( d \): distance from object to light source
  – \( a, b, c \): empirical constants
  – \( L \): intensity at light source
Phong Lightning Model

\[
I = \sum_i \frac{1}{a + b d_i + c d_i^2} \left( k_d L_{id} \max(\vec{l}_i \cdot \vec{n}) + k_s L_{is} \max(\vec{r}_i \cdot \vec{v})^\alpha \right) + k_a L_{ia}
\]
Phong Lightning Model

• Object

- $k_{ra}$, $k_{ga}$, $k_{ba}$: ambient reflection coefficients
- $k_{rd}$, $k_{gd}$, $k_{bd}$: diffuse reflection coefficient
- $k_{rs}$, $k_{gs}$, $k_{bs}$: specular reflection coefficient

• Light Source

- $L_{ra}$, $L_{ga}$, $L_{ba}$: ambient light intensity
- $L_{rd}$, $L_{gd}$, $L_{bd}$: diffuse light intensity
- $L_{rs}$, $L_{gs}$, $L_{bs}$: specular light intensity

©2014/2015, AVT
Phong Lightning Model
Color Control

• Defined by 3x3 matrices
  – Object
    \[
    \begin{bmatrix}
    k_{ra} & k_{ga} & k_{ba} \\
    k_{rd} & k_{gd} & k_{bd} \\
    k_{rs} & k_{gs} & k_{bs}
    \end{bmatrix}
    \]
  – Light Source
    \[
    \begin{bmatrix}
    L_{ra} & L_{ga} & L_{ba} \\
    L_{rd} & L_{gd} & L_{ba} \\
    L_{rs} & L_{gs} & L_{bs}
    \end{bmatrix}
    \]
Blinn Approximation

• Computation of $\vec{r}$ is expensive
  – Instead, *halfway vector* $\vec{h}$ is used
    • Vector normal a uma hipotética faceta reflectora pura
    • Vector médio normalizado

$$\vec{h} = \frac{\vec{l} + \vec{v}}{|\vec{l} + \vec{v}|}$$
Blinn Approximation

• Estimation of specular component
  – Use
    \[(\vec{n} \cdot \vec{h})^\beta\]
  – Instead of
    \[(\vec{r} \cdot \vec{v})^\alpha\]

  – Selecting a value for \(\beta\) that guarantees:
    \[(\vec{n} \cdot \vec{h})^\beta \approx (\vec{r} \cdot \vec{v})^\alpha\]
Modified Phong Lightning Model (Blinn-Phong Lightning Model)

\[ I = k_a L_a + \sum_{i} \frac{1}{a + b d_i + c d_i^2} \left( k_d L_i \max(\vec{l}_i \cdot \vec{n}) + k_s L_i \max(\vec{n}_i \cdot \vec{h}_i)^\alpha \right) \]
The lighting sub-stage optimization...

✓ It takes less computation for fewer lights in the scene;

✓ The simpler the shading model, the less computations (and less realism):
  - Wire-frame models;
  - Flat shaded models;
  - Gouraud shaded;
  - Phong shaded.
Iluminação e Materiais em OpenGL

- **Activar cálculo da Iluminação**
  - `glEnable(GL_LIGHTING)`
  - Depois de activado, `glColor()` é ignorado

- **Activar individualmente cada fonte de luz**
  - `glEnable(GL_LIGHTi)`  // `i=0,1...`

- **Escolher parâmetros do modelo de iluminação**
  - `glLightModelfv(pname, params)`
  - Exemplo: Definir luz ambiente global

  ```c
  GLfloat amb[] = {0.2f, 0.2f, 0.2f, 1.0};
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
  ```

©2014/2015, AVT
Iluminação e Materiais em OpenGL

• Propriedades das fontes de luz:
  – $I_a$, $I_s$ e $I_d$
  – Posição e direcção (se aplicável)
  – Atenuação, spot cut-off e spot exponent

```c
GLfloat ambient[] = { 0.0, 0.1, 0.0, 1.0 };    
GLfloat diffuse[] = { 0.0, 1.0, 1.0, 1.0 };    
GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };  
(...)                                       
gllightfv(GL_LIGHT0, GL_AMBIENT, ambient);    
gllightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);    
gllightfv(GL_LIGHT0, GL_SPECULAR, specular); 
(...)                                       
```

©2014/2015, AVT
Iluminação e Materiais em OpenGL

- Propriedades das fontes de luz:
  - $I_s$, $I_d$ e $I_a$
  - Posição e direcção (se aplicável)
  - Atenuação, spot cut-off e spot exponent

```c
GLfloat position[4] = {2.0, 2.0, 0.0, 1.0}  
GLfloat direction[4] = {-1.0, -1.0, 0.0, 0.0}  
(...)
// point light
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

```c
// directional light
glLightfv(GL_LIGHT1, GL_POSITION, direction);
(...)
```
Iluminação e Materiais em OpenGL

• Propriedades das fontes de luz:
  – \( I_s, I_d \) e \( I_a \)
  – Posição e direcção (se aplicável)
  – Atenuação, spot cut-off e spot exponent

\[
f_i = \frac{1}{a + bd_i + cd_i^2}
\]

(...)

```c
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a)
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, b)
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c)
(...)
```

©2014/2015, AVT
Iluminação e Materiais em OpenGL

- Propriedades das fontes de luz:
  - \( I_s, I_d \) e \( I_a \)
  - Posição e direcção (se aplicável)
  - Atenuação, spot cut-off e spot exponent \( (\alpha) \)

\[
\begin{align*}
\text{glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction)} \\
\text{glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, theta) // 0~180} \\
\text{glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, alpha) // 0~128}
\end{align*}
\]

©2014/2015, AVT
Iluminação e Materiais em OpenGL

• Propriedades das fontes de luz:
  – $I_s$, $I_d$ e $I_a$
  – Posição e direcção (se aplicável)
  – Atenuação, spot cut-off e spot exponent ($\alpha$)

```cpp
(...)
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction)
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, theta) // 0~180
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, alpha) // 0~128
(...)
```

$$
\text{spotlight\_effect}= \begin{cases} 
1 & \text{if } (!\text{spotlight}) \\
0 & \text{if } (!\text{inspot}) \\
\max(v \cdot d, 0) & \text{otherwise}
\end{cases}
$$
Iluminação e Materiais em OpenGL

• Material das superfícies:
  – $k_s$, $k_d$ e $k_a$
  – $k_{emissive}$ – Componente “emissiva” do material

```c
GLfloat matambient[] = { 0.5, 0.5, 0.5, 1.0 };  
GLfloat mat_diffuse[] = { 0.6, 0.6, 0.0, 1.0 };  
GLfloat mat_specular[] = { 0.8, 0.8, 0.8, 1.0 };  
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};  
GLfloat mat_shine = 100.0
```

```c
(...)  
glMaterialfv (GL_FRONT, GL_AMBIENT, mat_ambient);  
glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse);  
glMaterialfv (GL_FRONT, GL_SPECULAR, mat_specular);  
glMaterialfv (GL_FRONT, GL_EMISSION, mat_emission);  
glMaterialf (GL_FRONT, GL_SHININESS, mat_shine);
(...)```
Iluminação e Materiais em OpenGL

Vertex Color

\[\text{Vertex Color} = \text{emission}_{\text{material}} \]
\[+ \text{ambient}_{\text{lightmodel}} \ast \text{ambient}_{\text{material}}\]
\[+ \sum_{i=0}^{n-1} \left( \frac{1}{a_i + b_i d_i + c_i d_i^2} \ast \text{spotlight\_effect}_i \right) \]
\[\ast \left( \text{ambient}_{\text{light}_i} \ast \text{ambient}_{\text{material}} \right)\]
\[+ \text{diffuse}_{\text{light}_i} \ast \text{diffuse}_{\text{material}} \ast \max\{l_i \cdot n, 0\}\]
\[+ \text{specular}_{\text{light}_i} \ast \text{specular}_{\text{material}} \ast \max\{h_i \cdot n, 0\}^{\text{shininess}_i} \right)\]

©2014/2015, AVT
Iluminação e Materiais em OpenGL

| attenuation_factor_i | \[
| \frac{1}{a_i + b_i d_i + c_i d_i^2}
|\]

| spotlight_effect_i | \[
| \begin{cases}
| 1 & \text{if } \text{GL\_SPOT\_CUTOFF} = 180.0 \\
| 0 & \text{if } \max\{v_i \cdot d_i, 0\} < \cos(\text{GL\_SPOT\_CUTOFF}) \\
| \max\{v_i \cdot d_i, 0\}, & \text{otherwise}
|\end{cases}
|\]

| ambient_term_i | ambient_light \ast ambient_material |

| diffuse_term_i | diffuse_light_i \ast diffuse_material \ast \max\{l_i \cdot n, 0\} |

| specular_term_i | specular_light_i \ast specular_material \ast \max\{h_i \cdot n, 0\}^{\text{shininess}} |

**Vertex Color**

\[
\text{Vertex Color} = \sum_{i=1}^{n-1} \text{contribution}_{\text{lightsource}_i} + \text{emission}_{\text{material}} + \text{ambient}_{\text{light model}} \ast \text{ambient}_{\text{material}}
\]

\[
\text{contribution}_{\text{lightsource}_i} = \text{attenuation}_{\text{factor}_i} \ast \text{spotlight}_{\text{effect}_i} \ast (\text{ambient}_{\text{term}_i} + \text{diffuse}_{\text{term}_i} + \text{specular}_{\text{term}_i})
\]
VR Modeling

The Rendering Pipeline

Application → Geometry → Rasterizer

The Geometry Functional Sub-Stages

Model & View Transformation → Lighting → Projection → Clipping → Screen Mapping

©2014/2015, AVT
Projection Transformations:

✓ Models what portion (volume) of the virtual world the camera actually sees. There are two kinds of projections, parallel projection and perspective projection.
Orthogonal View-Volume
The portion of the virtual world seen by the camera at a given time is limited by front and back “clipping planes”. These are at $z=n$ and $z=f$. Only what is within the viewing cone (also called frustum) is sent down the rendering pipe.
Projection transformation

- Transforms the view volume to a normalized view volume and does the orthogonal projection
- Advantages:
  - Easy for hardware 3-D clipping
  - Clipping can be unified for perspective and parallel projection
- Involves translation, shearing, and scaling
Canonical Mapping from Frustum:

✓ The perspective transform maps the viewing volume to a unit cube with extreme points at (-1,-1,-1) and (1,1,1). This is also called the canonical (normalized) view volume.

The projection transformation matrix is:

\[
T'_{\text{projection}} = \begin{bmatrix}
2n/(r-l) & 0 & -(r+l)/(r-l) & 0 \\
0 & 2n/(t-b) & -(t+b)/(t-b) & 0 \\
0 & 0 & (f+n)/(f-n) & -2fn/(f-n) \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\]
VR Modeling

The Rendering Pipeline

Application → Geometry → Rasterizer

The Geometry Functional Sub-Stages

Model & View Transformation → Lighting → Projection → Clipping → Screen Mapping

©2014/2015, AVT
Clipping Transformation:

Since the frustum maps to the unit cube, only objects inside it will be rendered. Some objects are partly inside the unit cube (ex. the line and the rectangle). Then they need to be “clipped”. The vertex $V_1$ is replaced by new one at the intersection between the line and the viewing cone, etc.
VR Modeling

The Rendering Pipeline

Application → Geometry → Rasterizer

The Geometry Functional Sub-Stages

Model & View Transformation → Lighting → Projection → Clipping → Screen Mapping

©2014/2015, AVT
**Screen Mapping (Viewport Transformation):**

- The scene is rendered into a window with corners \((x_1, y_1), (x_2, y_2)\).

- Screen mapping is a translation followed by a scaling that affects the \(x\) and \(y\) coordinates of the primitives (objects), but not their \(z\) coordinates. Screen coordinates plus \(z \in [-1, 1]\) are passed to the rasterizer stage of the pipeline. In OpenGL this coordinate space is called Window Space.

©2014/2015, AVT
The rendering speed vs. surface polygon type

✓ The way surfaces are described influences rendering speed.

✓ If surfaces are described by triangle meshes, the rendering will be faster than for the same object described by independent quadrangles or higher-order polygons. This is due to the graphics board architecture which may be optimized to render triangles.
Câmara em OpenGL

• Por omissão:
  – Câmara na origem do referencial do mundo, aponta para –z
  – Volume de visualização é cubo centrado na origem c/ lado 2
Especificação em OpenGL

```c
glMatrixMode(GL_MODELVIEW)
gluLookAt()
...
```

```c
glMatrixMode(GL_PROJECTION)
...
```
Abertura do Volume Perspectivo

FOV: Field of View

\[ \tan \left( \frac{\Theta_V}{2} \right) = \frac{h}{D} \]

\( \Theta_V \): abertura vertical

\[ \tan \left( \frac{\Theta_W}{2} \right) = \frac{w}{D} \]

\( \Theta_W \): abertura horizontal

Vista lateral do volume

Vista topo do volume
void glOrtho(
    GLdouble left,
    GLdouble right,
    GLdouble bottom,
    GLdouble top,
    GLdouble nearVal,
    GLdouble farVal);

\[
\begin{pmatrix}
  \frac{2}{\text{right}-\text{left}} & 0 & 0 & t_x \\
  0 & \frac{2}{\text{top}-\text{bottom}} & 0 & t_y \\
  0 & 0 & \frac{-2}{\text{far}-\text{near}} & t_z \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
t_x = -\frac{\text{right}+\text{left}}{\text{right}-\text{left}}
\]
\[
t_y = -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}}
\]
\[
t_z = -\frac{\text{far}+\text{near}}{\text{far}-\text{near}}
\]
void `glFrustum`

\[
\begin{bmatrix}
\frac{2 \text{ near}}{\text{right-left}} & 0 & A & 0 \\
0 & \frac{2 \text{ near}}{\text{top-bottom}} & B & 0 \\
0 & 0 & C & D \\
0 & 0 & -1 & 0
\end{bmatrix}
\]

\[a = \frac{\text{right+left}}{\text{right-left}}\]

\[b = \frac{\text{top+bottom}}{\text{top-bottom}}\]
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

\[
P = \begin{bmatrix}
\frac{\cot \frac{\text{fovy}}{2}}{\text{aspect}} & 0 & 0 & 0 \\
0 & \frac{\cot \frac{\text{fovy}}{2}}{2} & 0 & 0 \\
0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
Exemplo em OpenGL

Calcule as dimensões da janela de visualização do *frustum* simétrico definido por:

```
gluPerspective(120.0f, 1.33f , 15.0, 120.0)
```

Nota:
Sinopse do comando `gluPerspective`:

```c
void gluPerspective(   GLdouble fovy,
                     GLdouble aspect,
                     GLdouble near,
                     GLdouble far)
```
Computing Architectures

The Rendering Pipeline

Application → Geometry → Rasterizer

©2014/2015, AVT
\[
\begin{pmatrix}
x_{eye} \\
y_{eye} \\
z_{eye} \\
w_{eye}
\end{pmatrix} = M_{modelView} \cdot \begin{pmatrix}
x_{obj} \\
y_{obj} \\
z_{obj} \\
w_{obj}
\end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix}
x_{obj} \\
y_{obj} \\
z_{obj} \\
w_{obj}
\end{pmatrix}
\]

\[
\begin{pmatrix}
x_{eye} \\
y_{eye} \\
z_{eye} \\
w_{eye}
\end{pmatrix} = \left(M_{modelView}^{-1}\right)^T \cdot \begin{pmatrix}
x_{obj} \\
y_{obj} \\
z_{obj} \\
w_{obj}
\end{pmatrix}
\]

\[
\begin{pmatrix}
x_{clip} \\
y_{clip} \\
z_{clip} \\
w_{clip}
\end{pmatrix} = M_{projection} \cdot \begin{pmatrix}
x_{eye} \\
y_{eye} \\
z_{eye} \\
w_{eye}
\end{pmatrix}
\]

\[
\begin{pmatrix}
x_{ndc} \\
y_{ndc} \\
z_{ndc}
\end{pmatrix} = \begin{pmatrix}
x_{clip}/w_{clip} \\
y_{clip}/w_{clip} \\
z_{clip}/w_{clip}
\end{pmatrix}
\]

\[
\begin{pmatrix}
x_w \\
y_w \\
z_w
\end{pmatrix} = \begin{pmatrix}
\frac{w}{2}x_{ndc} + (x + \frac{w}{2}) \\
\frac{h}{2}y_{ndc} + (y + \frac{h}{2}) \\
\frac{f-n}{2}z_{ndc} + \frac{f+n}{2}
\end{pmatrix}
\]
Computing Architectures

The Rendering Pipeline

Application → Geometry → Rasterizer

©2014/2015, AVT
The Rasterizer Stage

- Performs operations in hardware for speed;
- Converts vertices information from the geometry stage (x,y,z, color, texture) into pixel information on the screen;
- The pixel color information is in color buffer;
- The pixel z-value is stored in the Z-buffer (has same size as color buffer);
- Assures that the primitives that are visible from the point of view of the camera are displayed.
The Rasterizer Stage - continued

- The scene is rendered in the back buffer;
- It is then swapped with the front buffer which stores the current image being displayed;
- This process eliminates flicker and is called “double buffering”;
- All the buffers on the system are grouped into the frame buffer.
OpenGL rendering pipeline(s)
Detail Steps in OpenGL

• **Rasterization**
  – Input are transformed vertices and associated colors and texture coordinates
  – Scan convert polygons, and perform per-pixel operations
    • yield fragments consisting of depths, color, alpha value, and texture coordinates by linear interpolation, except for the texture coordinates, which are done in a perspectively correct way

• **Fragment processing**
  – Shading
  – Access texture maps via a lookup and does blending
    • Also handle multi-texturing
  – Fog

• **Per-fragment operations**
  – Alpha channel, stencil, depth tests etc.
  – Fragments passing all tests are written to frame-buffer

• **Frame-buffer operations**
Shading Techniques

- **Wire-frame** is simplest – only shows polygon visible edges;
- The **flat shaded** model assigns same color to all pixels on a polygon (or side) of the object;
- **Gouraud** or smooth shading interpolates colors inside the polygons based on the color of the edges;
- **Phong shading** interpolates the vertex normals before calculating the light intensity based on the model described – most realistic shading model.
Local illumination methods

Flat shading model

Gouraud shading model

Phong shading model

I_p = I_b - (I_b - I_a) \frac{x_{b-p}}{x_b-x_a}
Computing architectures

Wire-frame model

Flat shading model

Gouraud shading model

©2014/2015, AVT
Scene illumination

✓ Local methods (Flat shaded, Gouraud shaded, Phong shaded) treat objects in isolation. They are computationally faster than global illumination methods;

✓ Global illumination treats the influence of one object on another object’s appearance. It is more demanding from a computation point of view but produces more realistic scenes.
Flat shaded Utah Teapot

Phong shaded Utah Teapot
Global scene illumination

✓ The inter-reflections and shadows cast by objects on each other.
How to create textures:

✔ Models are available online in texture “libraries” of cars, people, construction materials, etc.

✔ Custom textures from scanned photographs or
✔ Using an interactive paint program to create bitmaps
Texturing methods

- Objects rendered using Phong reflection model and Gouraud or Phong interpolated shading often appear rather ‘plastic’ and ‘floating in air’
- Texture effects can be added to give more realistic looking surface appearance
  - Texture mapping
    - Texture mapping uses pattern to be put on a surface of an object
  - Light maps
    - Light maps combine texture and lighting through a modulation process
  - Bump Mapping
    - Smooth surface is distorted to get variation of the surface
  - Environmental mapping
    - Environmental mapping (reflection maps) – enables ray-tracing like output
Where does mapping take place?

- Most mapping techniques are implemented at the end of the rendering pipeline.
- Texture mapping as a part of shading process, but a part that is done on a fragment-to-fragment basis.
  - Very efficient because few polygons make it past the clipper.
How to set \((s,t)\) texture coordinates?

- Set the coordinates manually
  - Set the texture coordinates for each vertex ourselves
- Automatically compute the coordinates
  - Use an algorithm that sets the texture coordinates for us