The 3D Graphics Rendering Pipeline

Animação e Visualização Tridimensional

João Madeiras Pereira

Which Visualization?

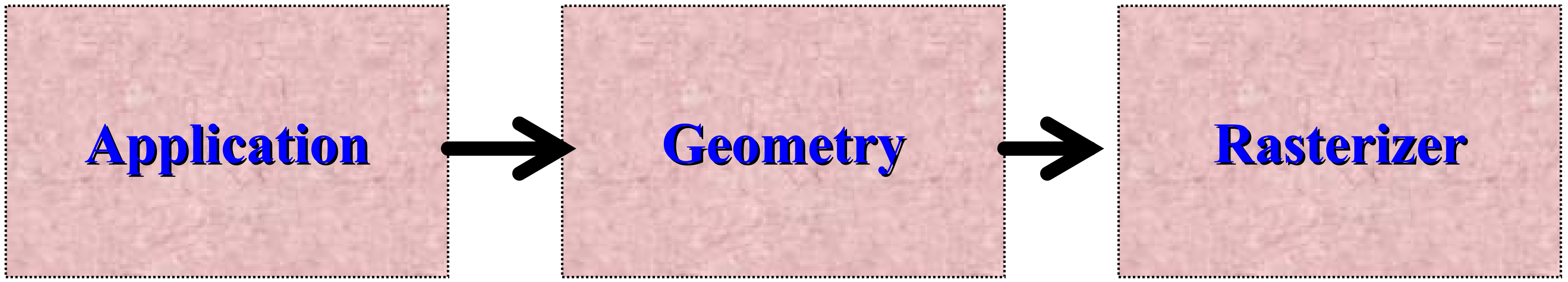
Interactive Real-Time Rendering

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

Application paradigm: games

The Graphics Rendering Pipeline

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.



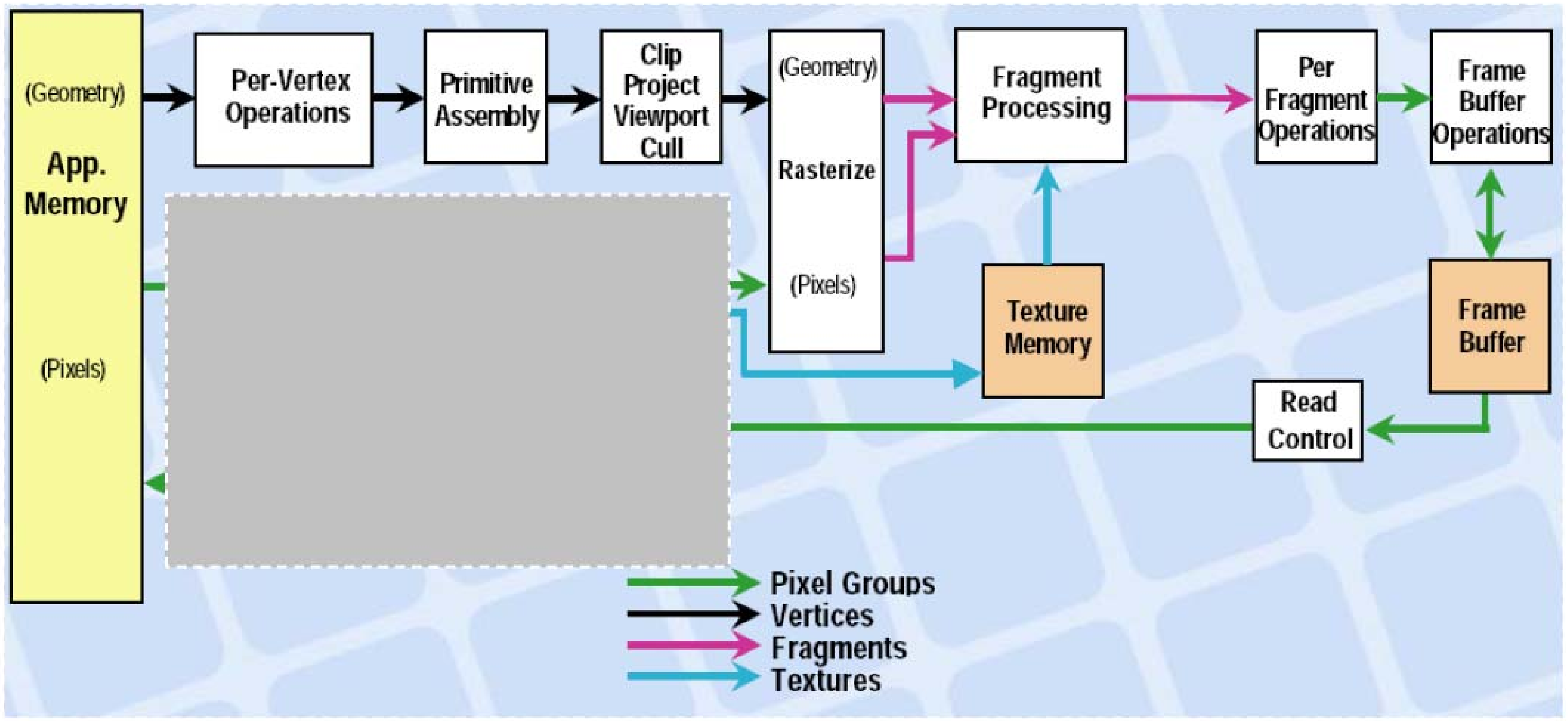
Graphics API vs. Application API

- **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

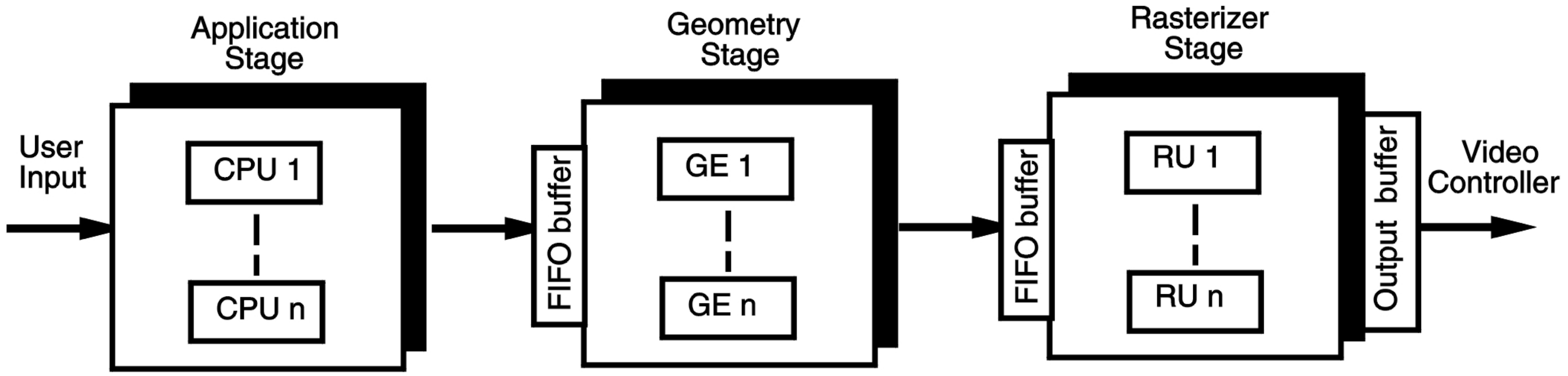
- Set up a polygonal mesh for the scene
- Culling back-facing polygons
  (in world coord. sys., or screen coord. sys. (prefer))
- 3D-to-2D projection and clipping
  - Apply viewing transformation
  - Compute lighting
  - Apply projection
  - Clip polygons against view volume
    (in 3D screen space or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

OpenGL rendering pipeline(s)

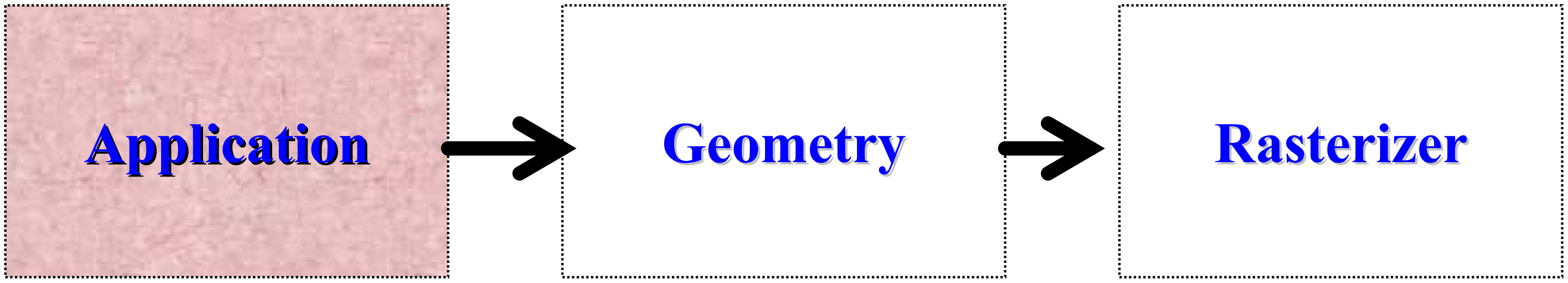


The Graphics Rendering Pipeline

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



The application stage

✓ Is done entirely in software by the CPU;

✓ It reads Input devices (such as gloves, mouse);

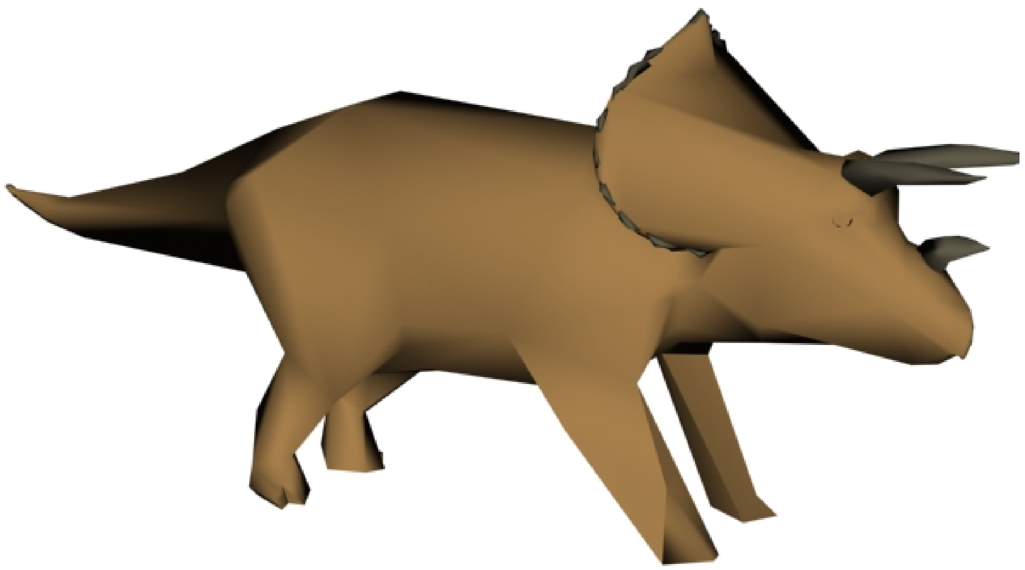
✓ It changes the coordinates of the virtual camera;

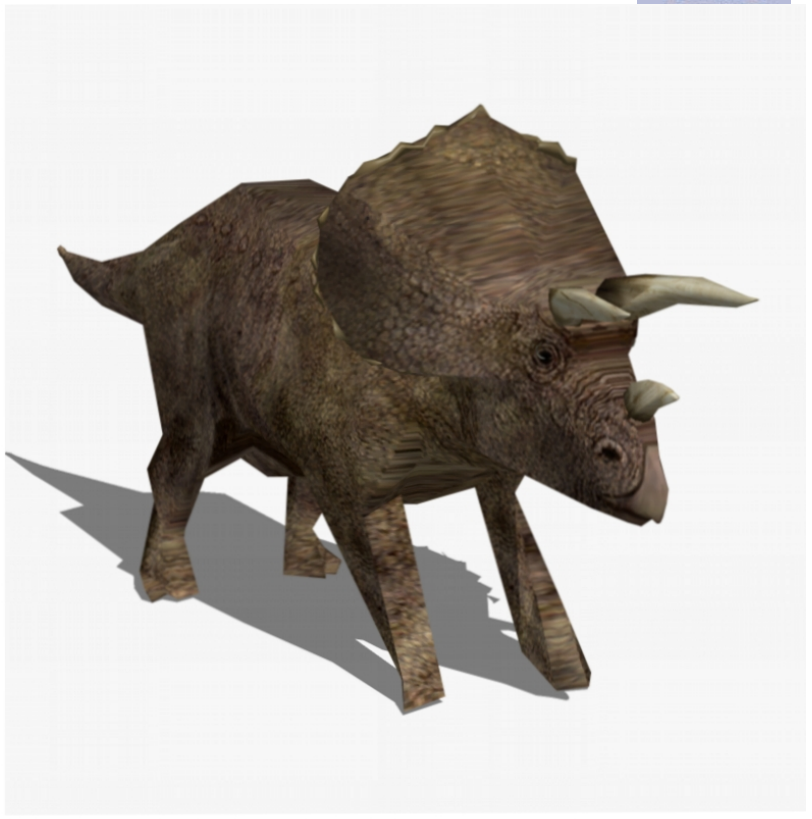
✓ It performs collision detection and collision response (based on object properties) for haptics;

✓ 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

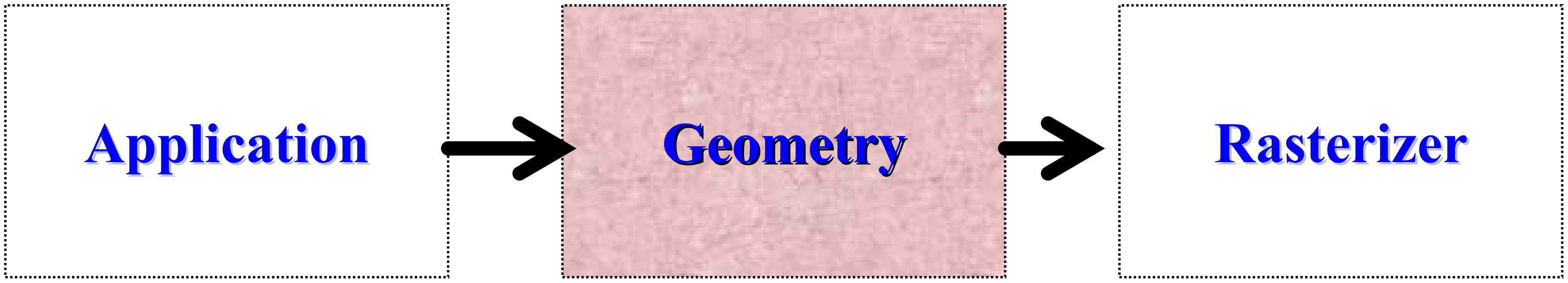
Higher resolution model
134,754 polygons.

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.

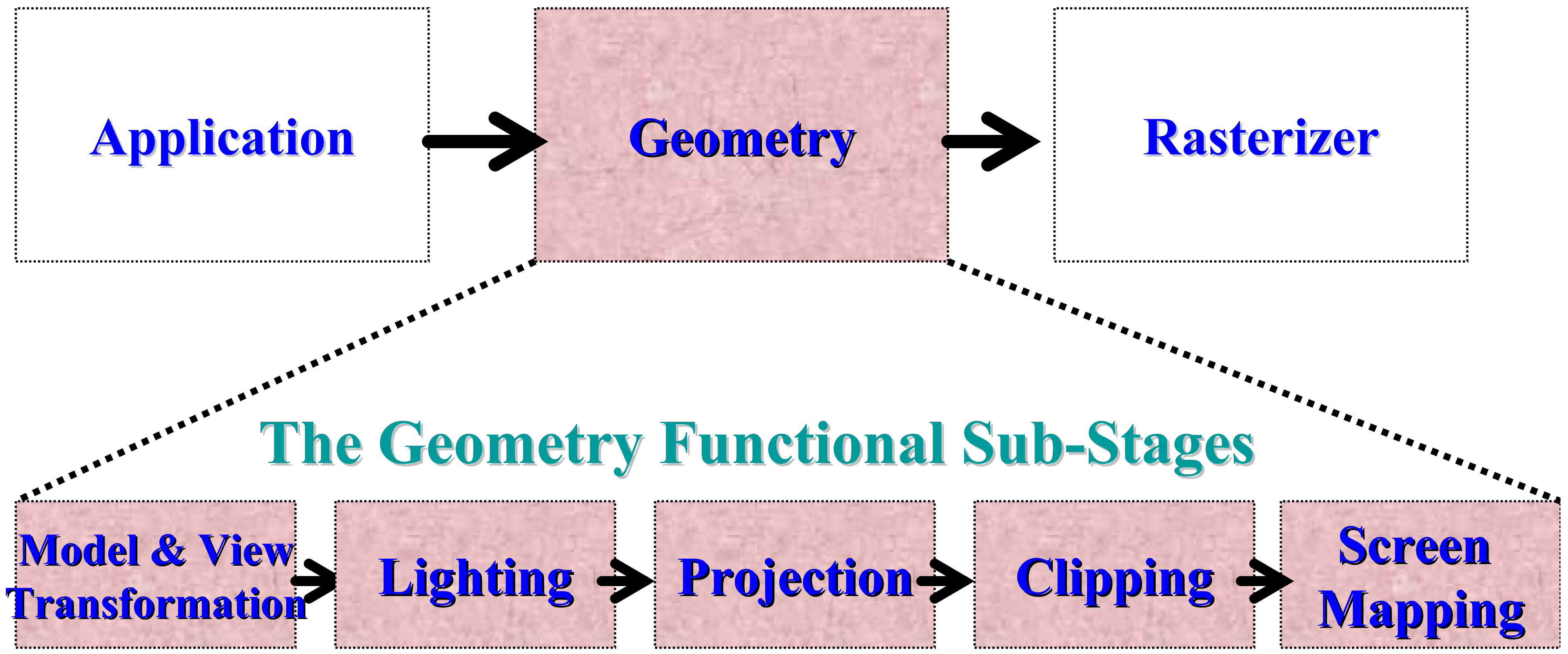
The Rendering Pipeline



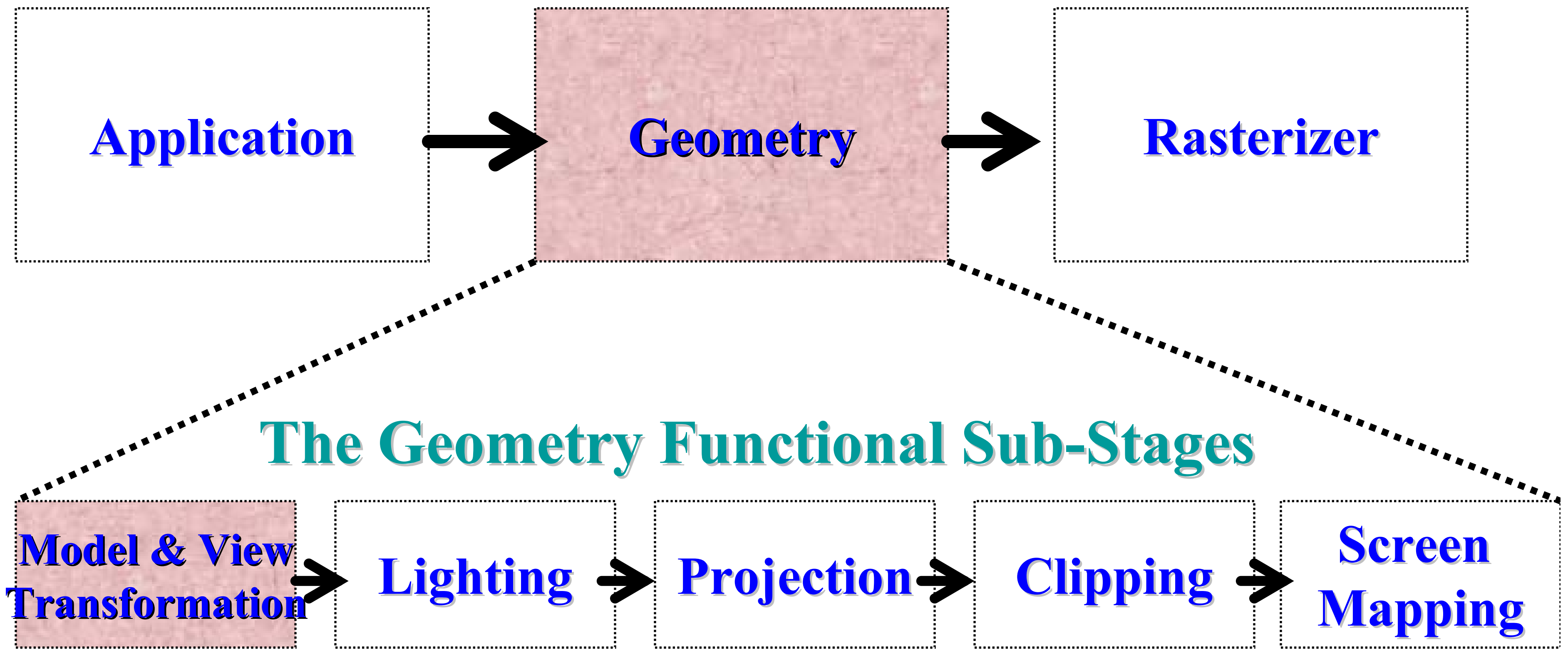
Detail Steps

- 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)

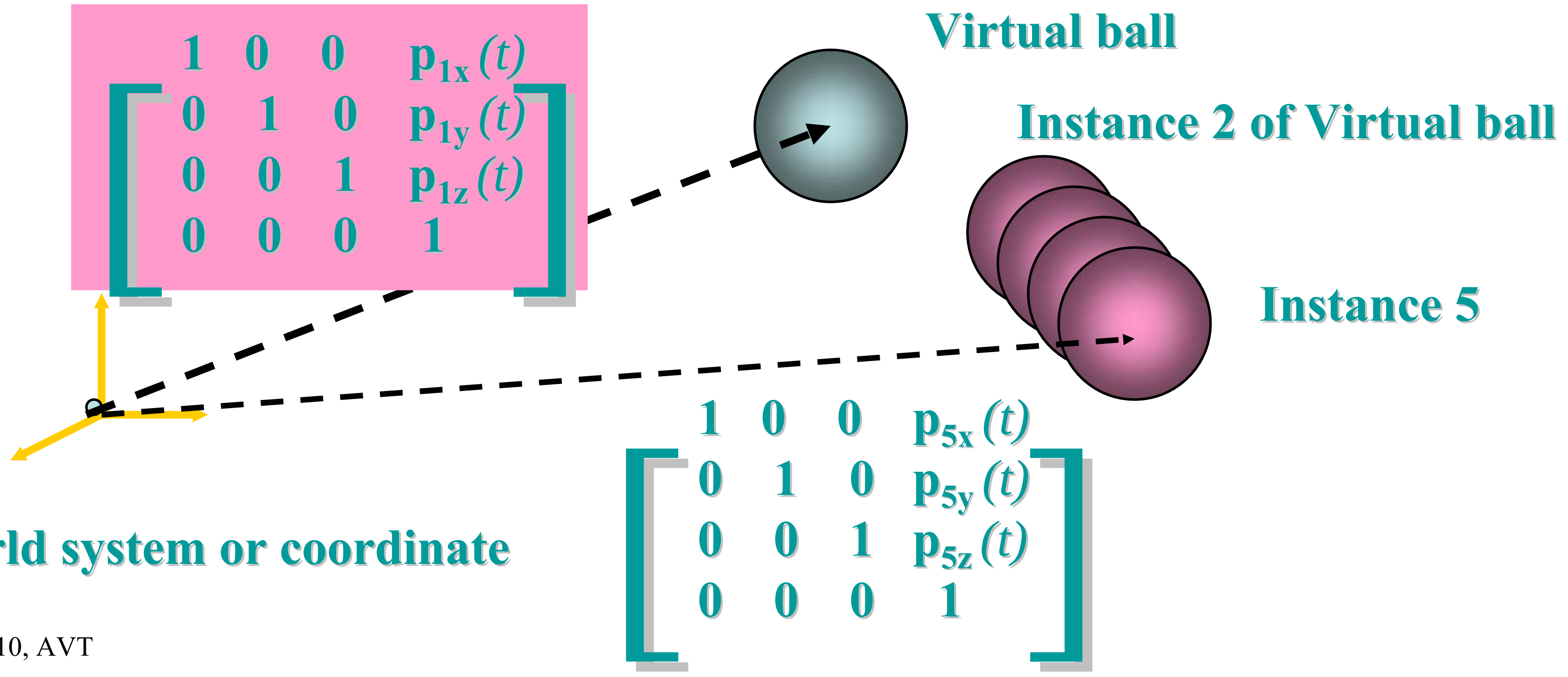


The Rendering Pipeline



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***.

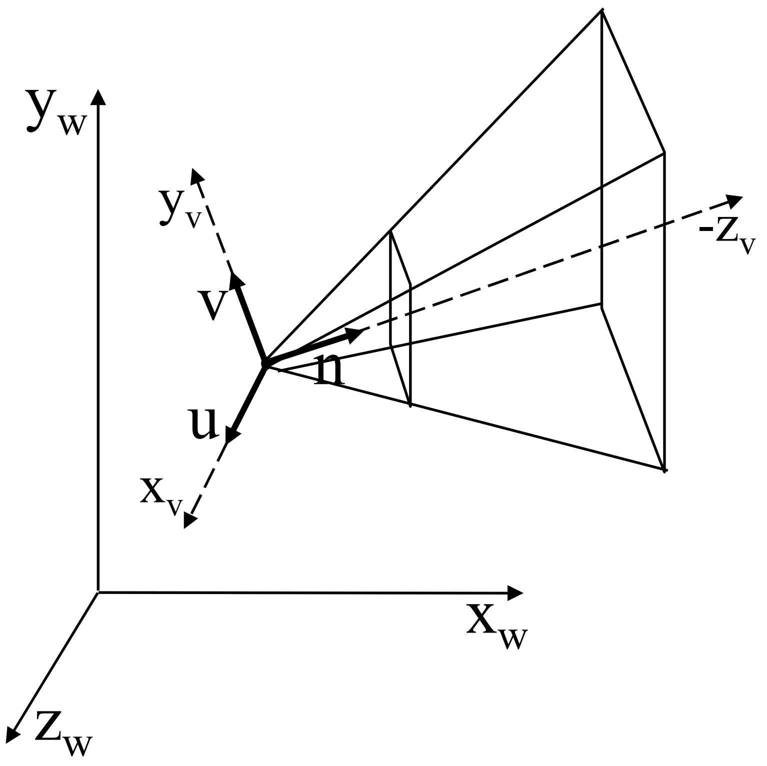


$$\begin{bmatrix} 1 & 0 & 0 & p_{1x}(t) \\ 0 & 1 & 0 & p_{1y}(t) \\ 0 & 0 & 1 & p_{1z}(t) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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 n = lookAt – CamPos
- ✓Left-handed frame



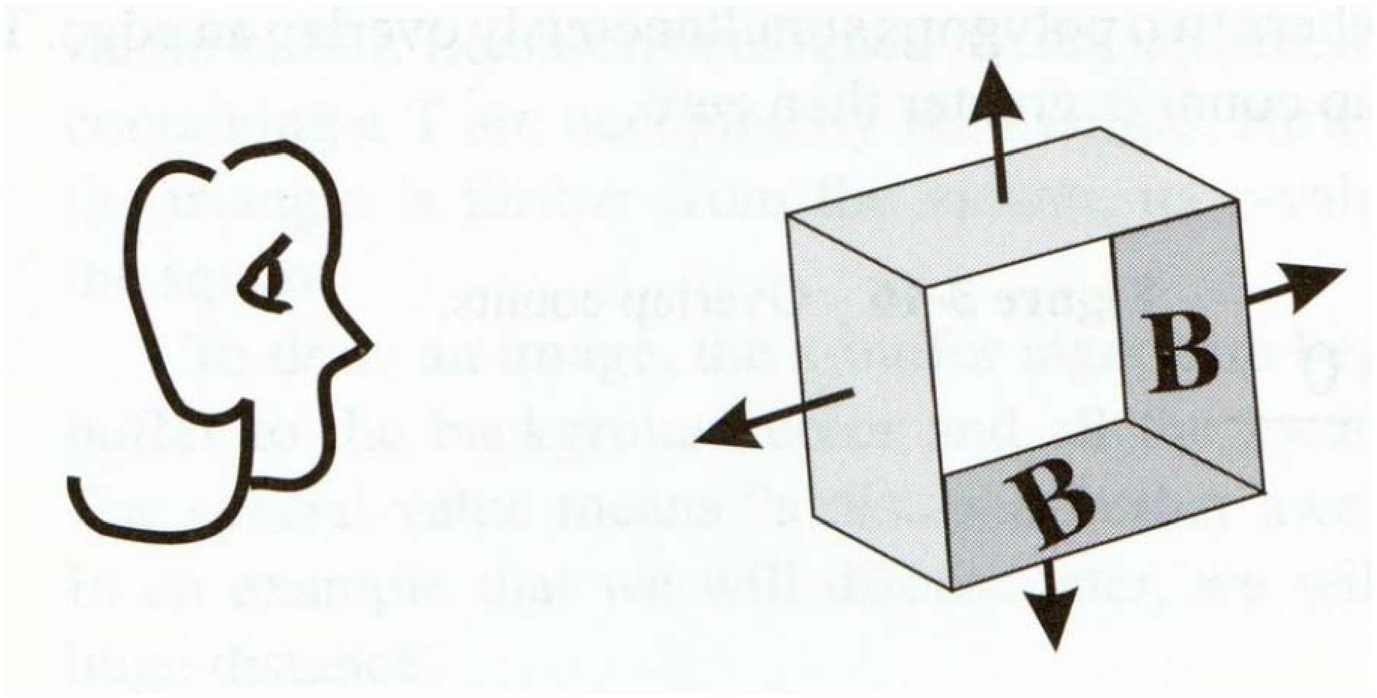
Viewing Transformation

$$R_{rot} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ -n_x & -n_y & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_{trans} = \begin{bmatrix} 1 & 0 & 0 & -VRP_x \\ 0 & 1 & 0 & -VRP_y \\ 0 & 0 & 1 & -VRP_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Viewing Transformation $\mathbf{M_{vis} = R_{rot} \bullet T_{trans}}$**

$$M_{vis} = \begin{bmatrix} u_x & u_y & u_z & -u \bullet VRP \\ v_x & v_y & v_z & -v \bullet VRP \\ -n_x & -n_y & -n_z & n \bullet VRP \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Back-Faces Culling



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

•Based on dot product between face normal and viewing vector

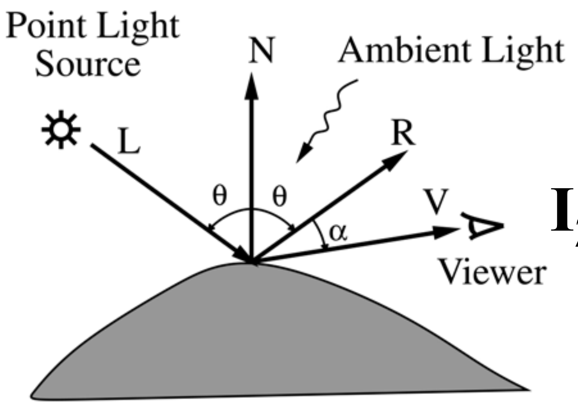
The lighting sub-stage

✓ It calculates the vertex color based on:

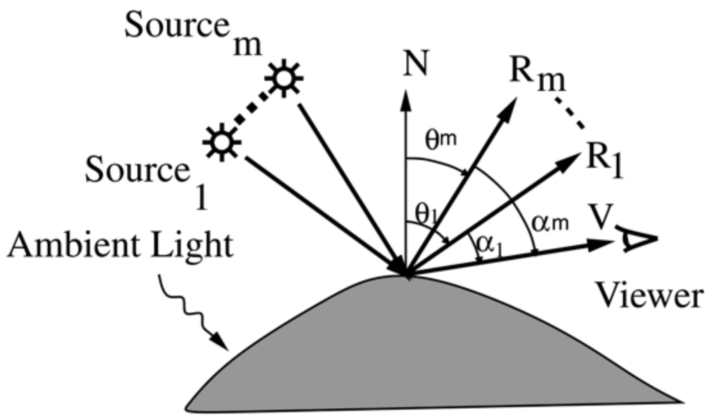
- type and number of simulated light sources;
- the lighting model;
- the surface material properties;
- atmospheric effects such as fog or smoke.

✓ Lighting results in object shading which makes the scene more realistic.

Computing architectures



a)



b)

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

where: $\mathbf{I_\lambda}$ is the intensity of light of wavelength $\boldsymbol{\lambda}$**;**

$\mathbf{I_{a\lambda}}$ is the intensity of ambient light;

$\mathbf{K_a}$ is the surface ambient reflection coefficient;

$\mathbf{O_{d\lambda}}$ is the object diffuse color;

$\mathbf{f_{att}}$ is the atmospheric attenuation factor;

$\mathbf{I_{p\lambda}}$ is the intensity of point light source of wavelength λ;

$\mathbf{K_d}$ is the diffuse reflection coefficient;

$\mathbf{K_s}$ is the specular reflection coefficient;

$\mathbf{O_{s\lambda}}$ is the specular color;

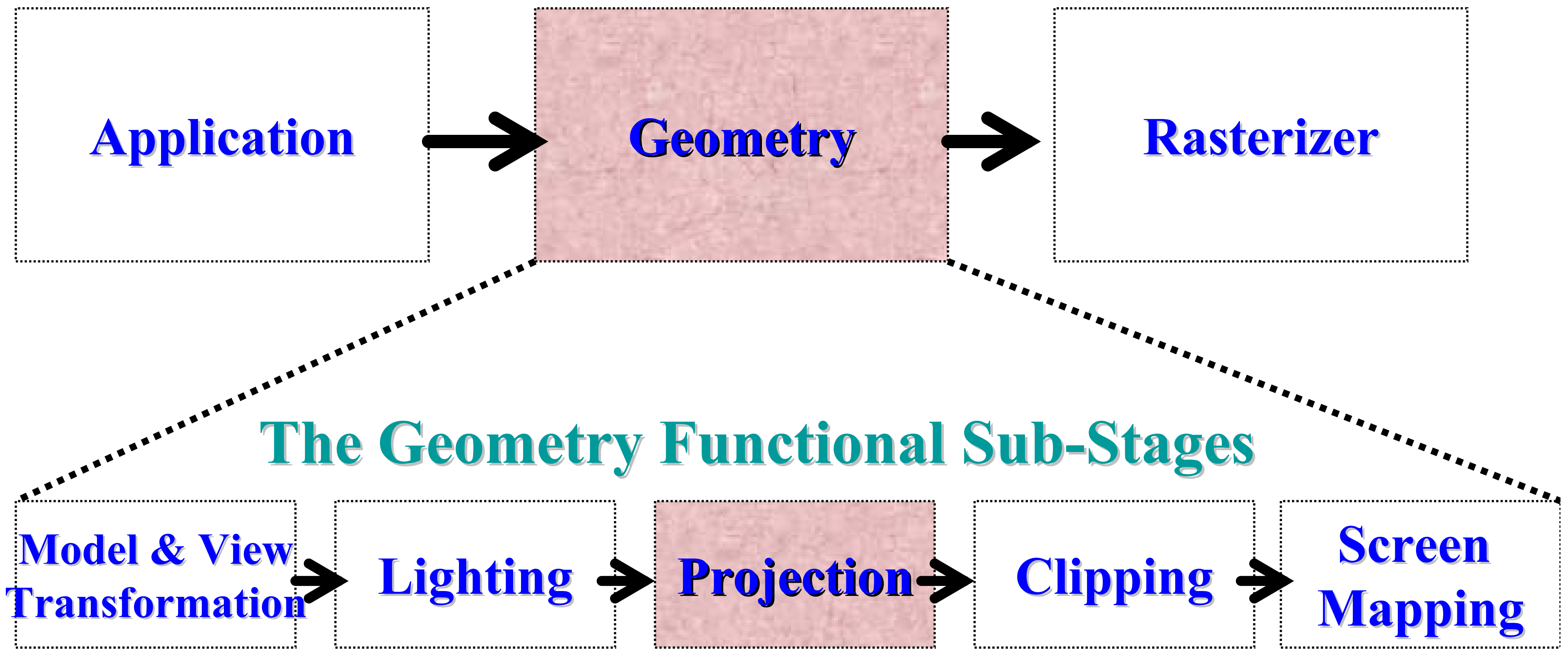
The lighting sub-stage optimization…

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

✓ The simpler the shading model, the less computations (and less realism):

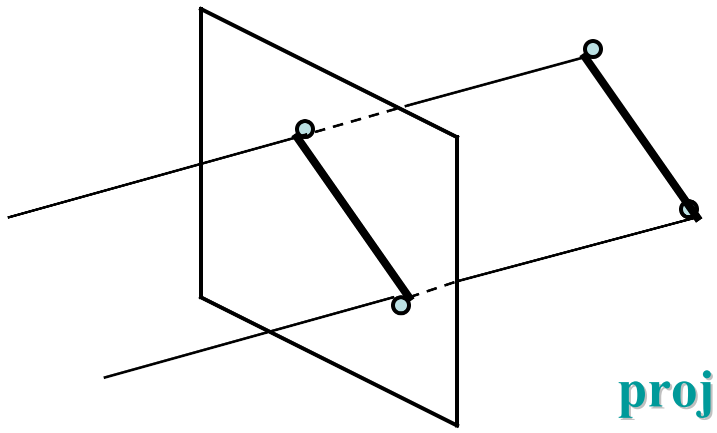
- Wire-frame models;
- Flat shaded models;
- Gouraud shaded;
- Phong shaded.

The Rendering Pipeline

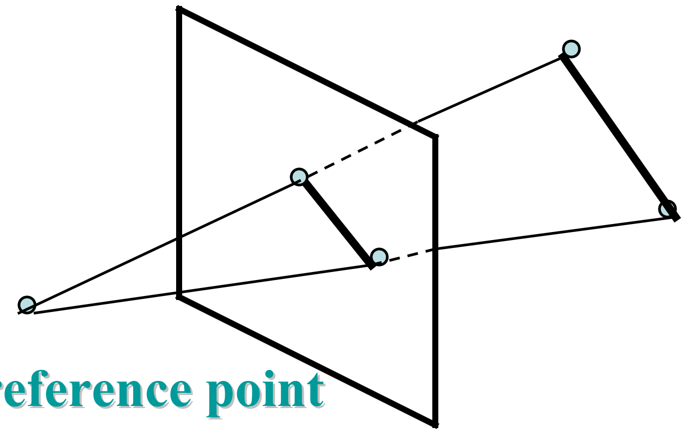


Projection Transformations:

✓ Models what portion (volume) of the virtual world the camera actually sees. There are two kinds of projections, ***parallel*** projection and ***perspective*** projection

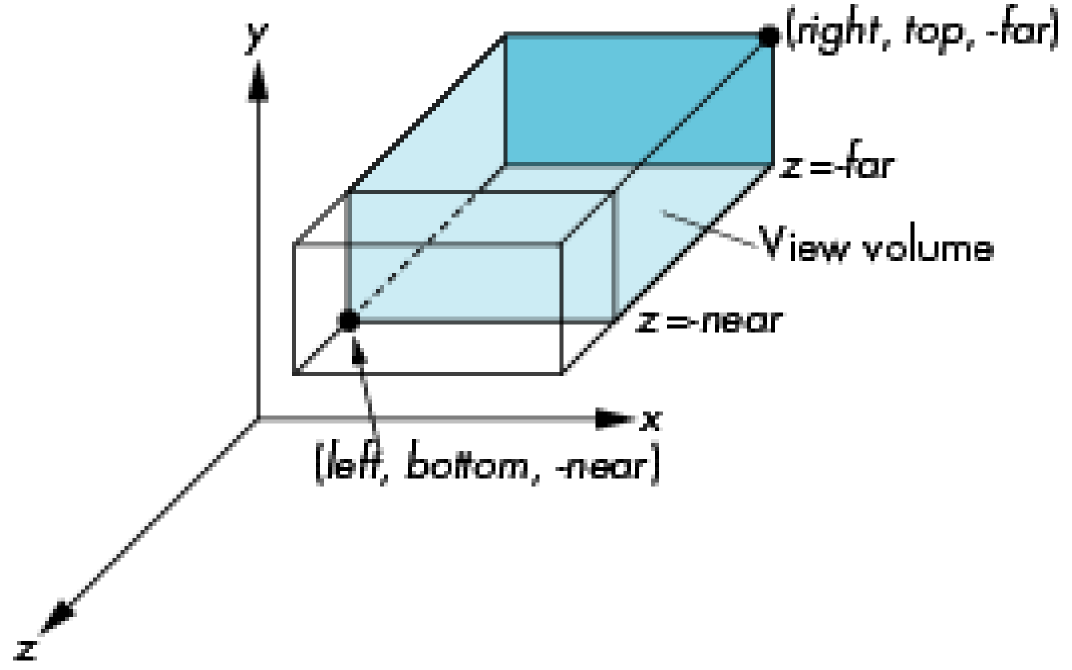


Parallel projection



Perspective projection

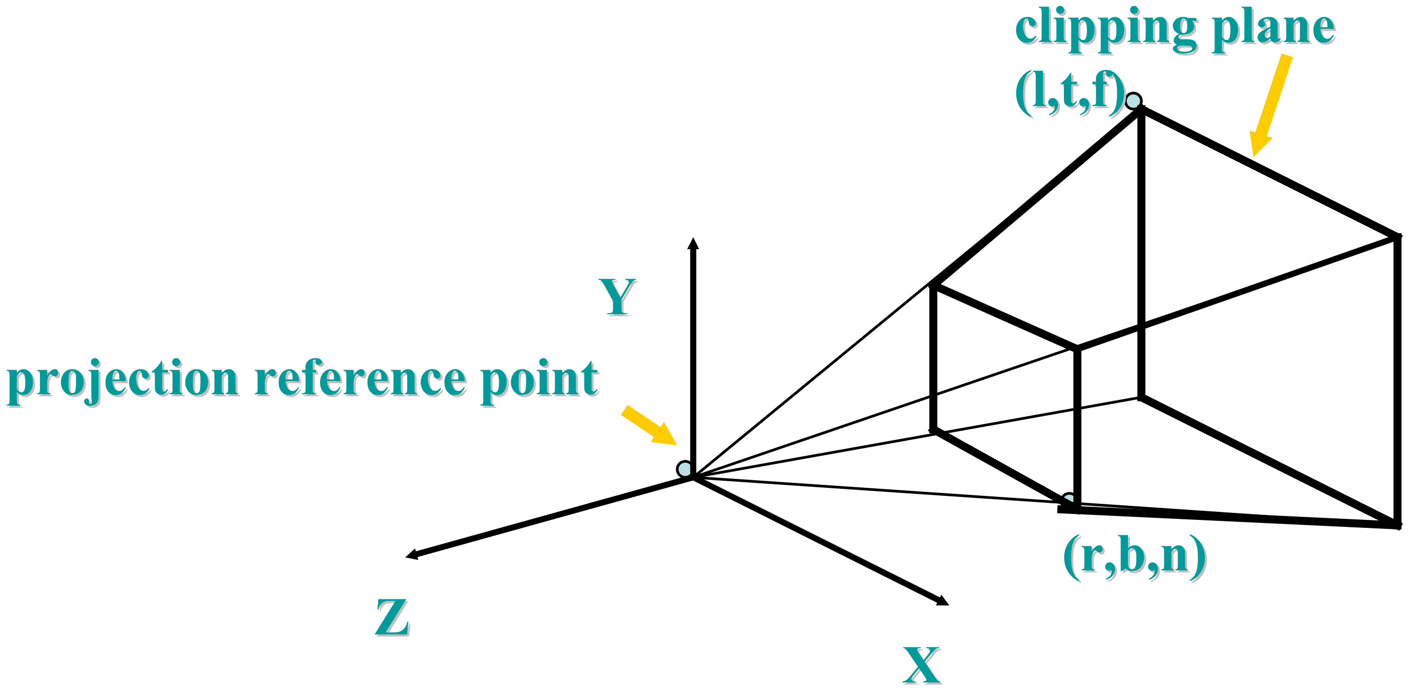
Orthogonal View-Volume



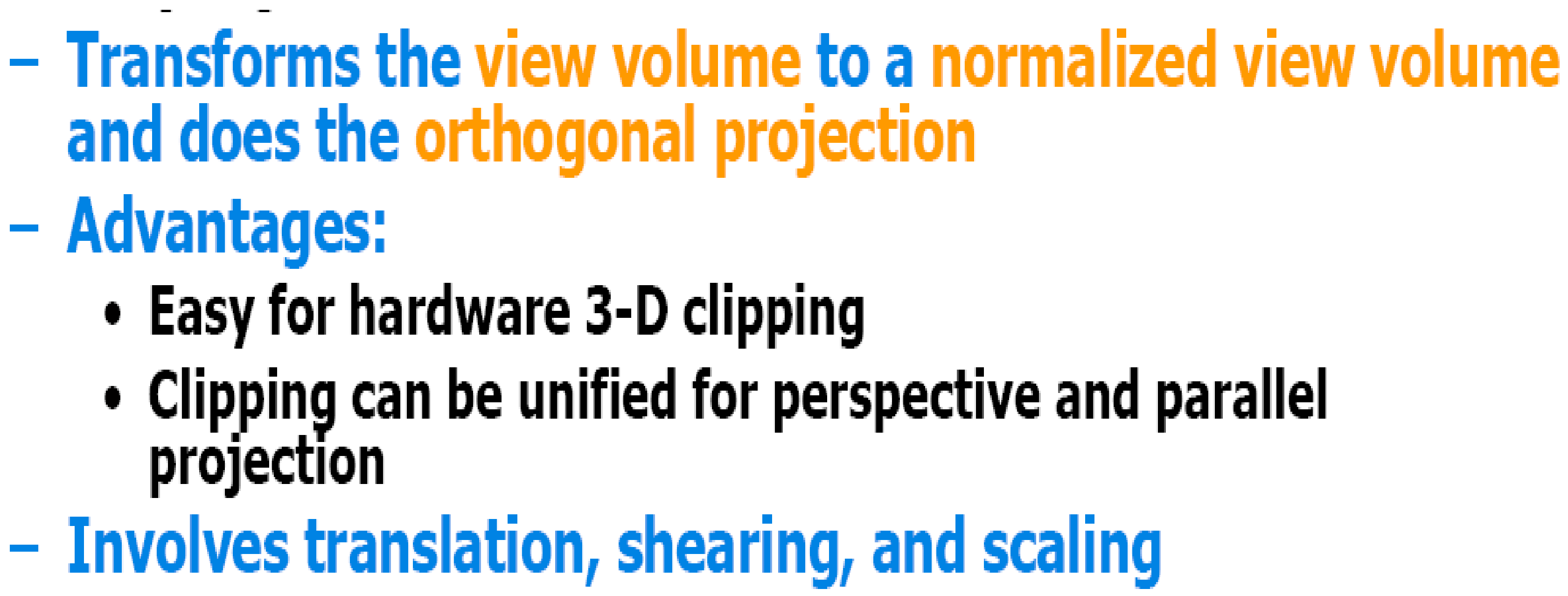
Frustum

✓ 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

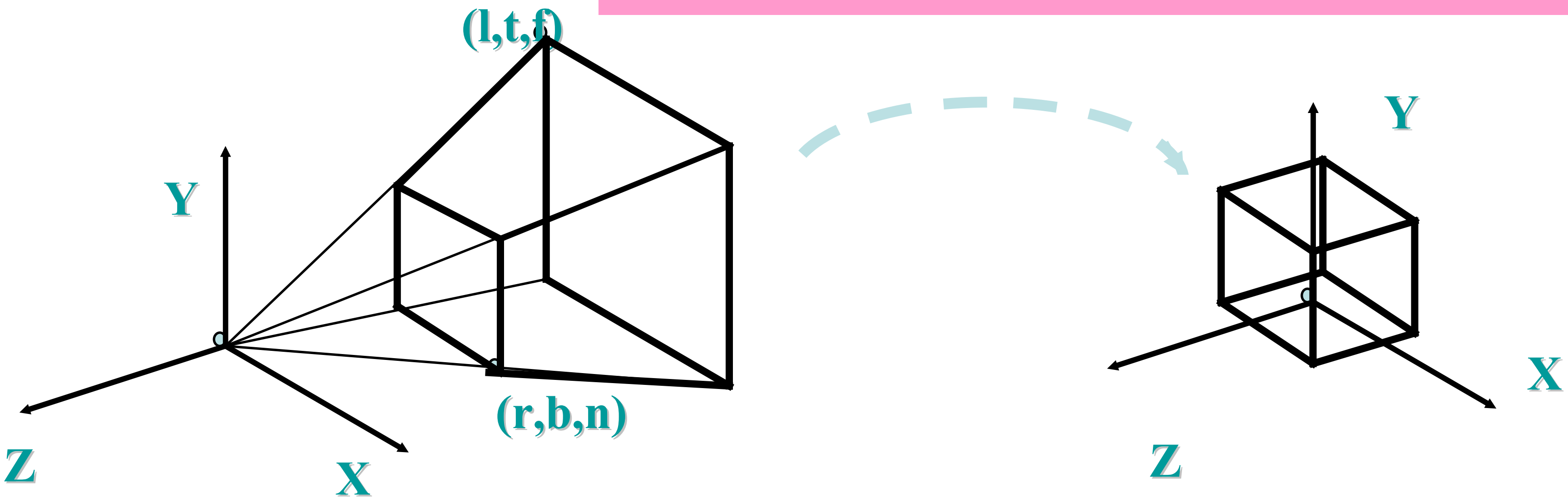


- Transforms the view volume to a normalized view volume and does the orthogonal projection
- Advantages:
  - Easy for hardware 3-D clipping
  - Clipping can be unified for perspective and parallel projection
- Involves translation, shearing, and scaling

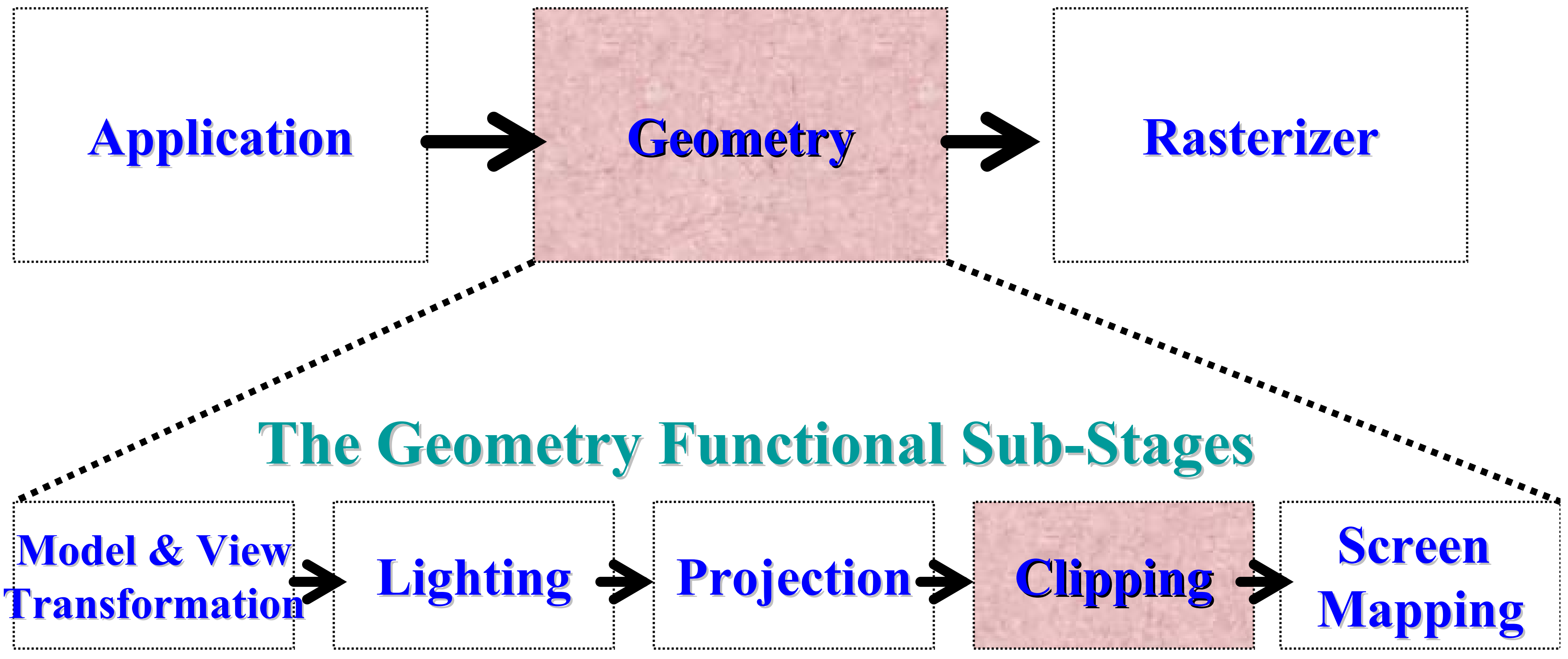
Canonical Mapping from Frustum:

✓ 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***.

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

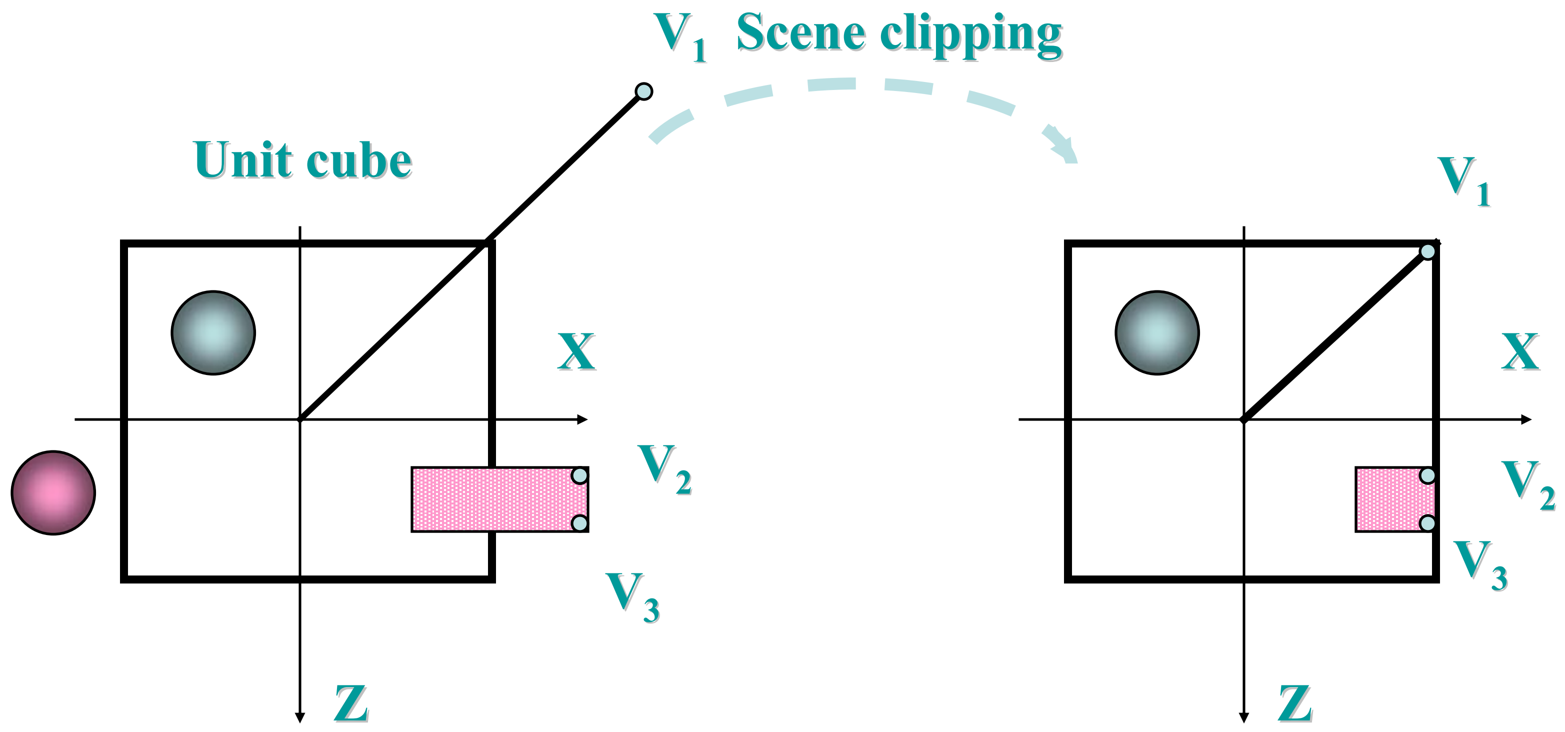


The Rendering Pipeline

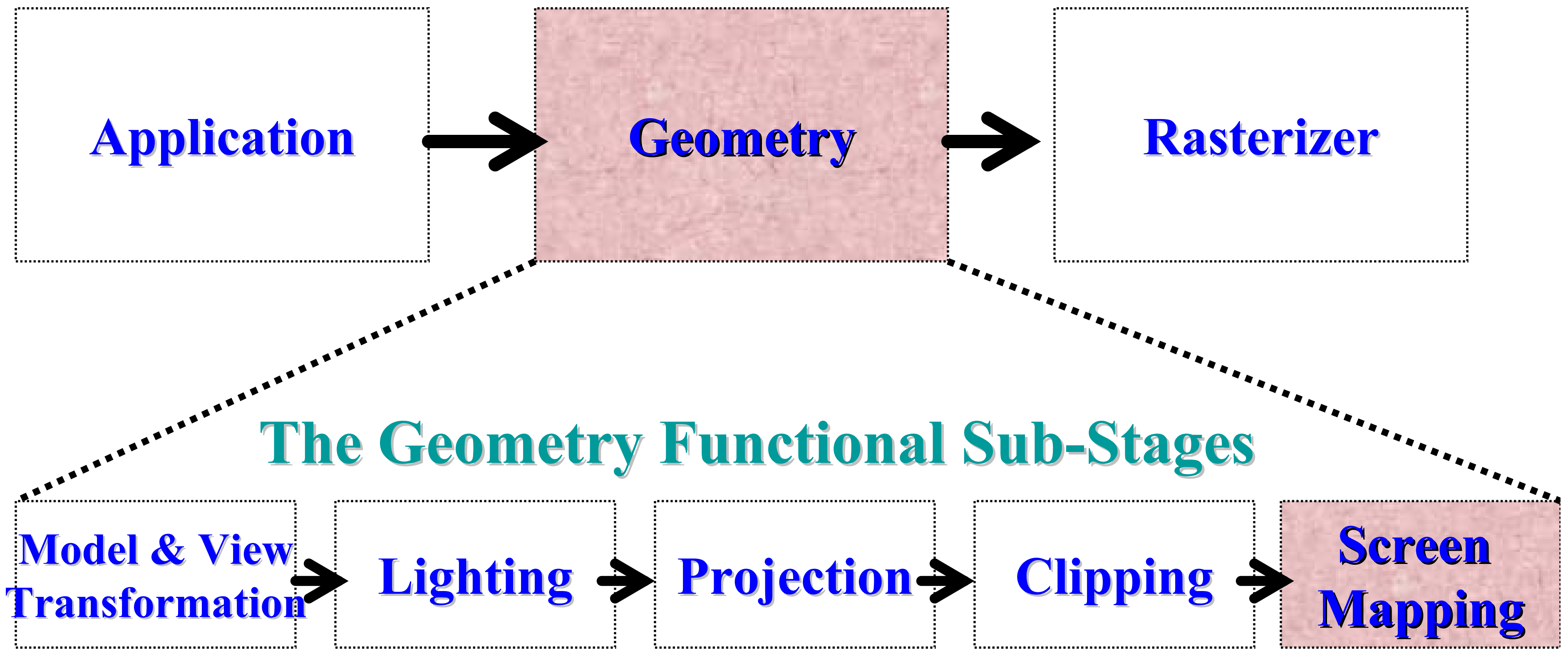


Clipping Transformation:

✓ 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.



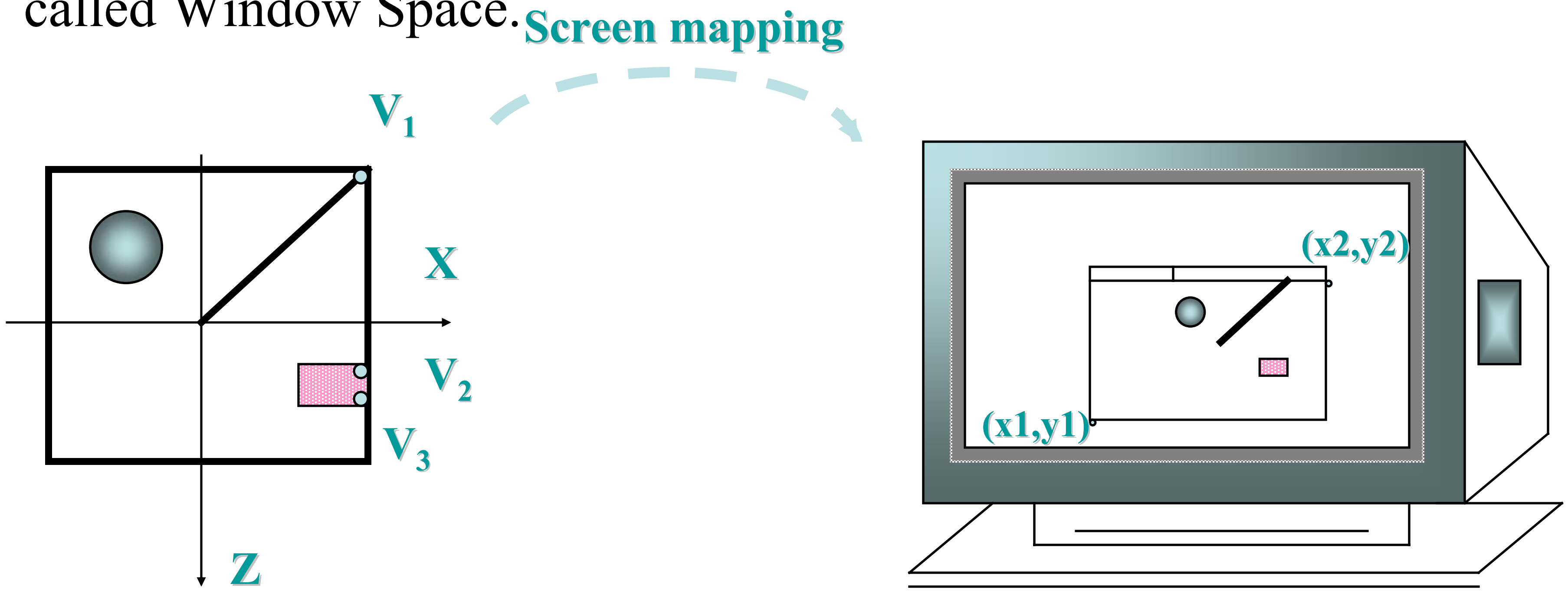
The Rendering Pipeline



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.

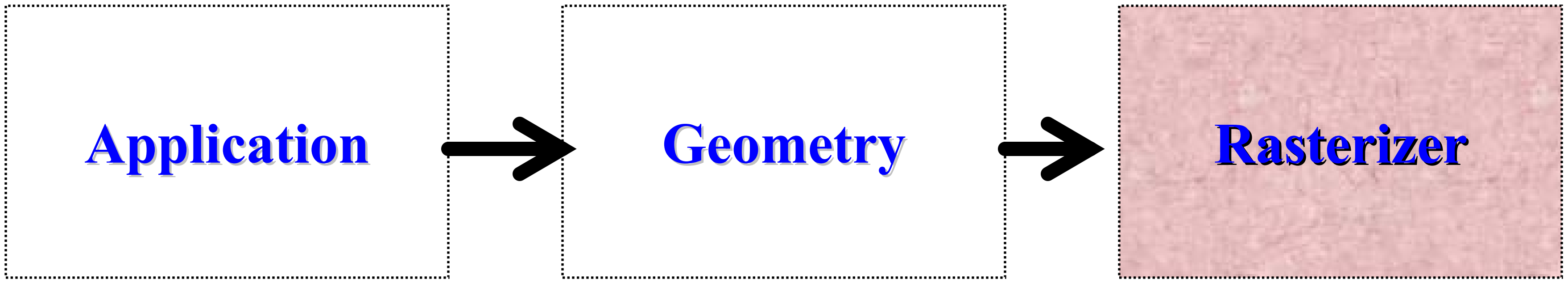


The rendering speed vs. surface polygon type

✓ 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.

The Rendering Pipeline



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***;

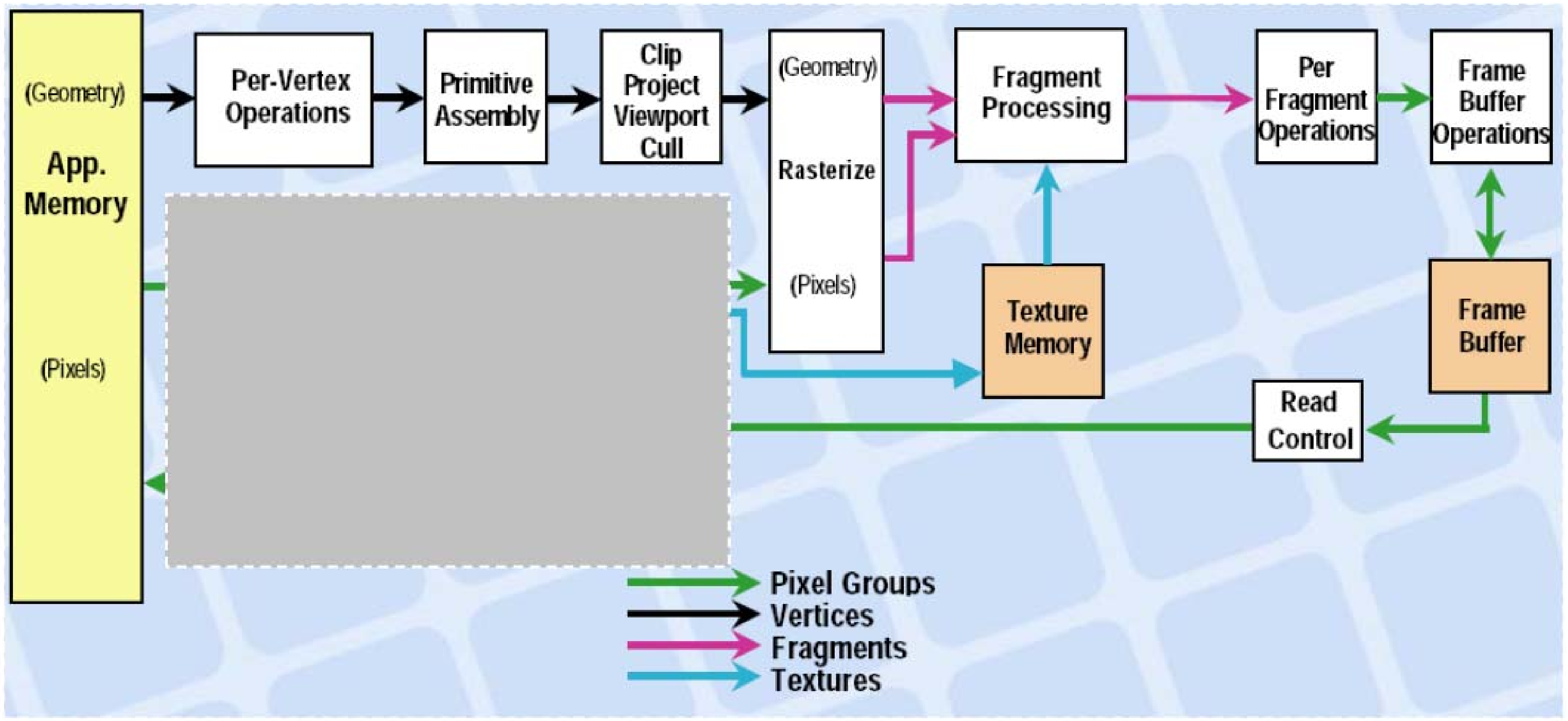
✓ The pixel z-value is stored in the ***Z-buffer*** (has same size as color buffer);

✓ 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

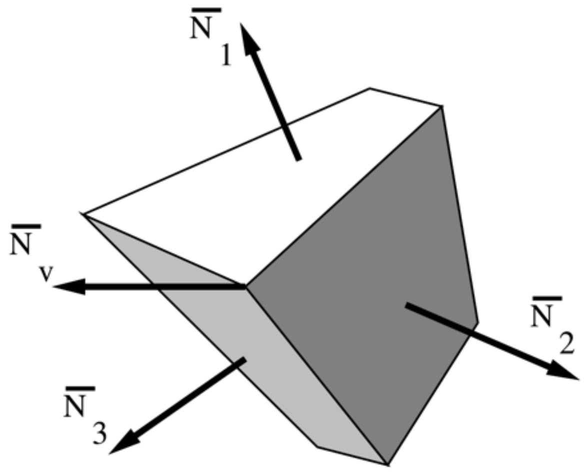
✓ ***Wire-frame*** is simplest – only shows polygon visible edges;

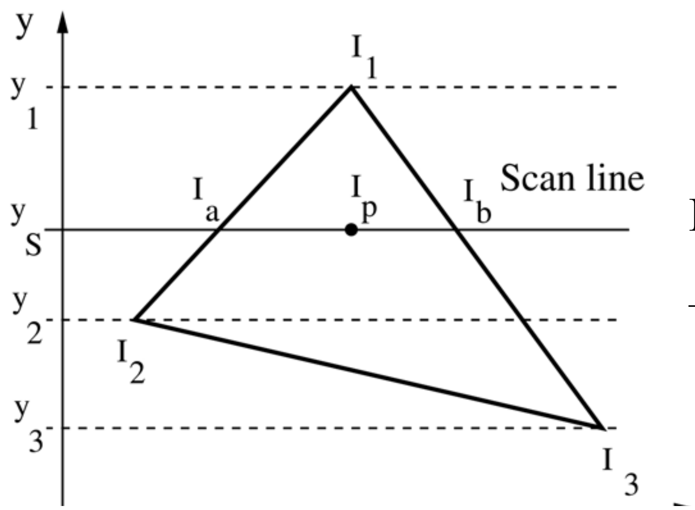
✓ The ***flat shaded*** model assigns same color to all pixels on a polygon (or side) of the object;

✓ ***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

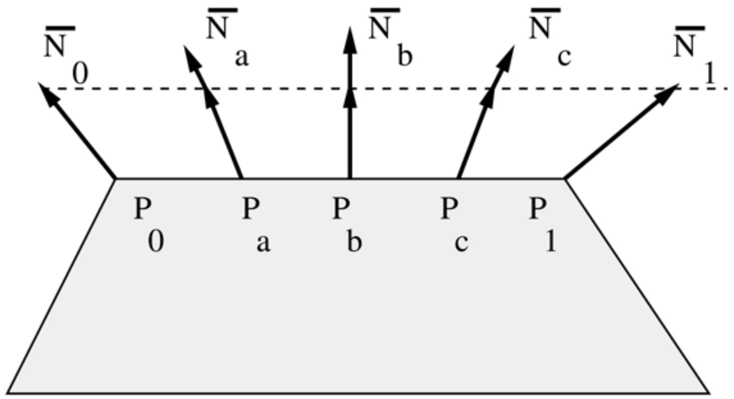




$$I_p = I_b - (I_b - I_a)\,\frac{x_b - x_p}{x_b - x_a}$$

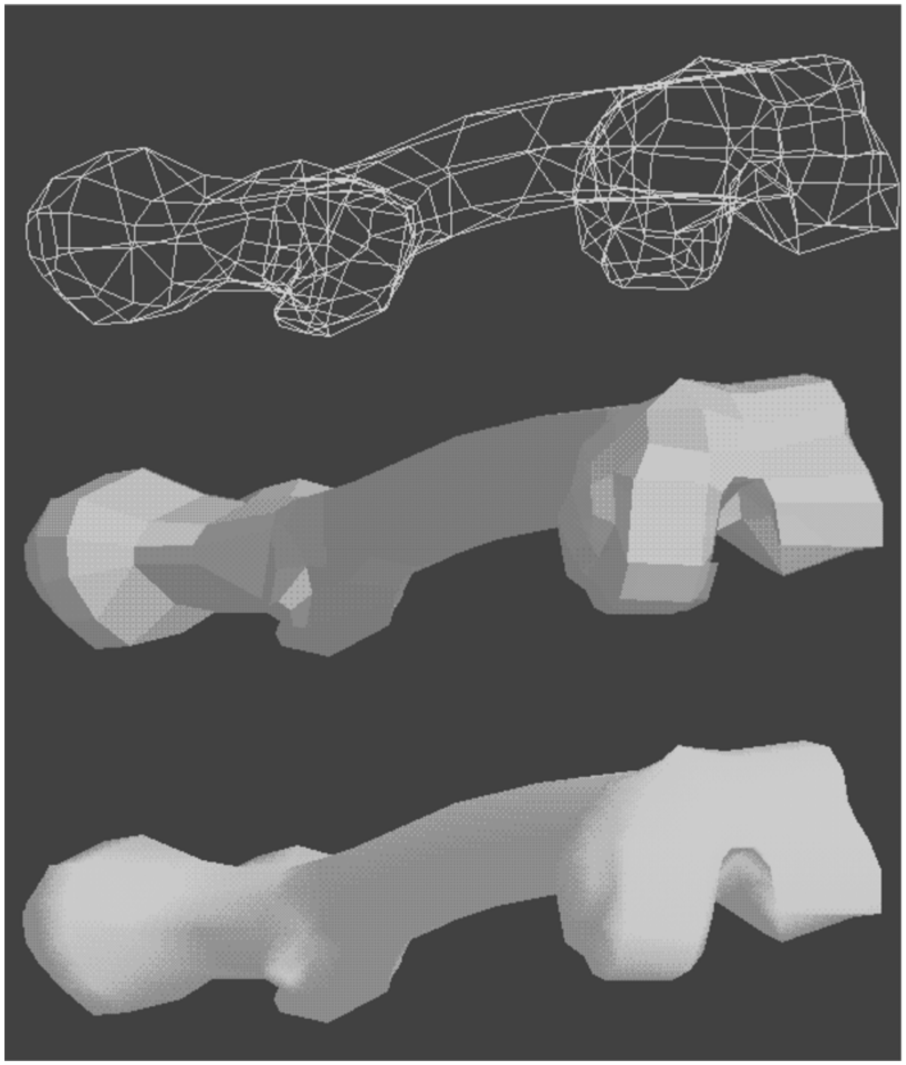
Flat shading model

Gouraud shading model



Phong shading model

Computing architectures



Wire-frame model

Flat shading model

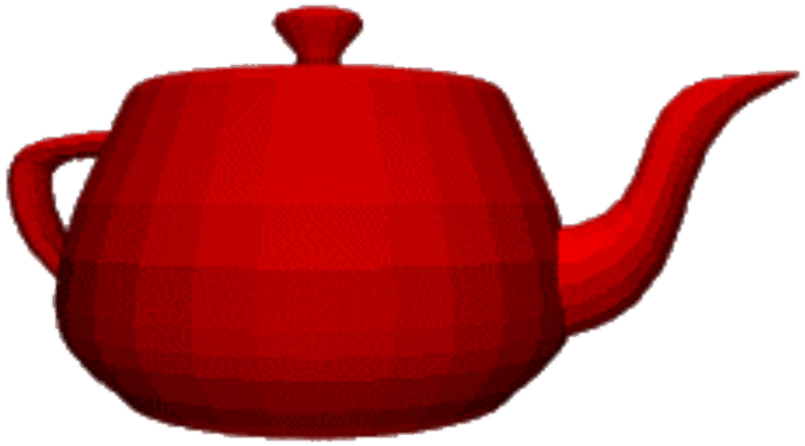
Gouraud shading model

Scene illumination

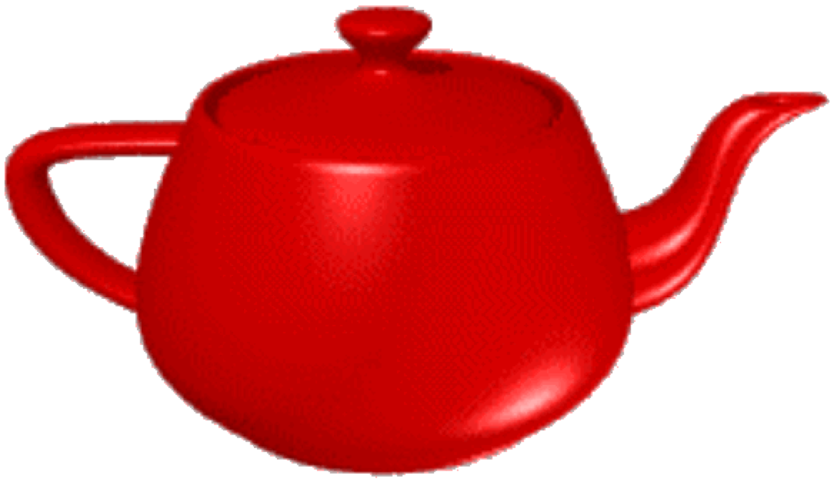
- ✓ Local methods (Flat shaded, Gouraud shaded, Phong shaded) treat objects in isolation. They are computationally faster than global illumination methods;

- ✓ 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

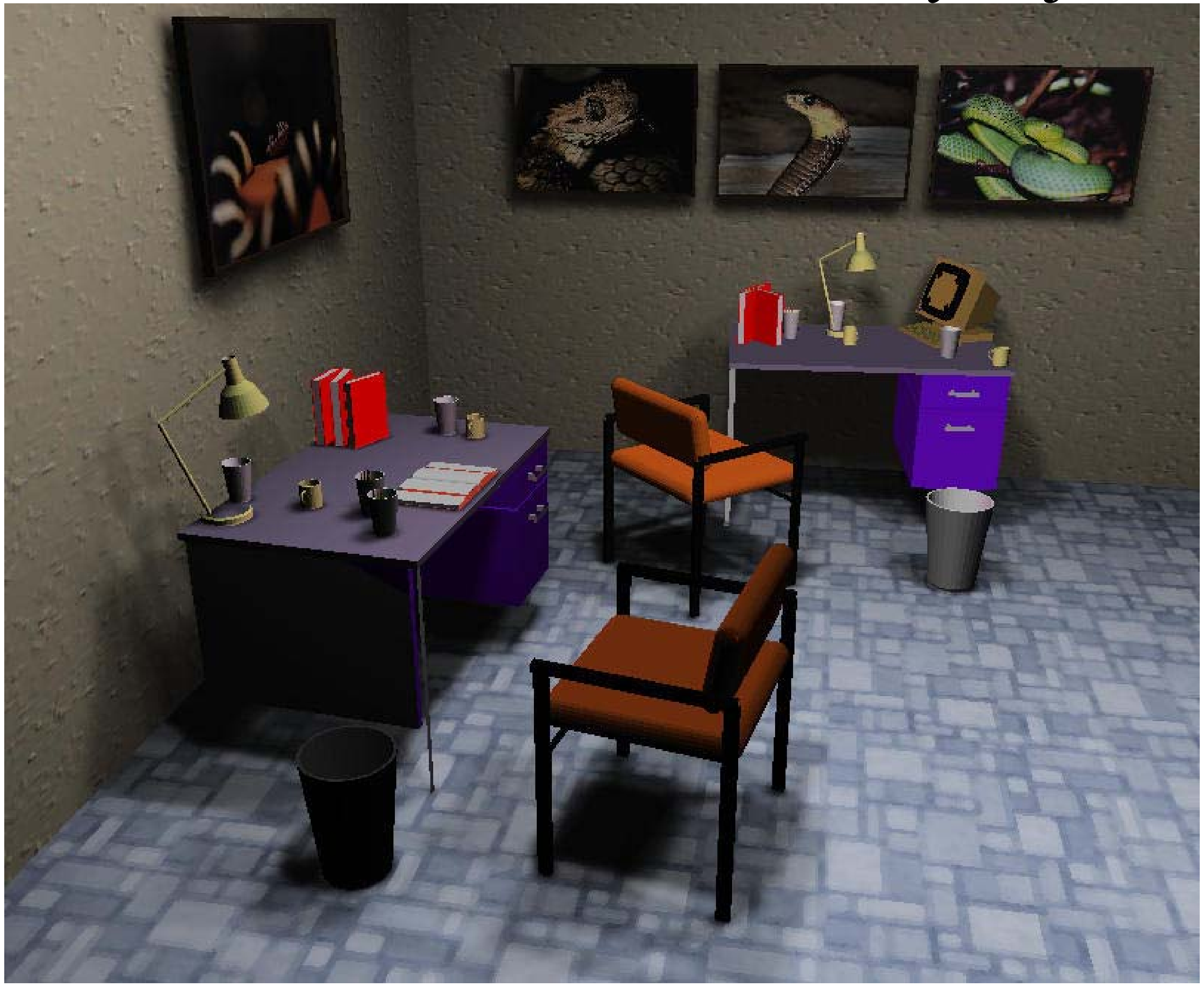


Phong shaded Utah Teapot



Global scene illumination

✓ The inter-reflections and shadows cast by objects on each other.



How to create textures:

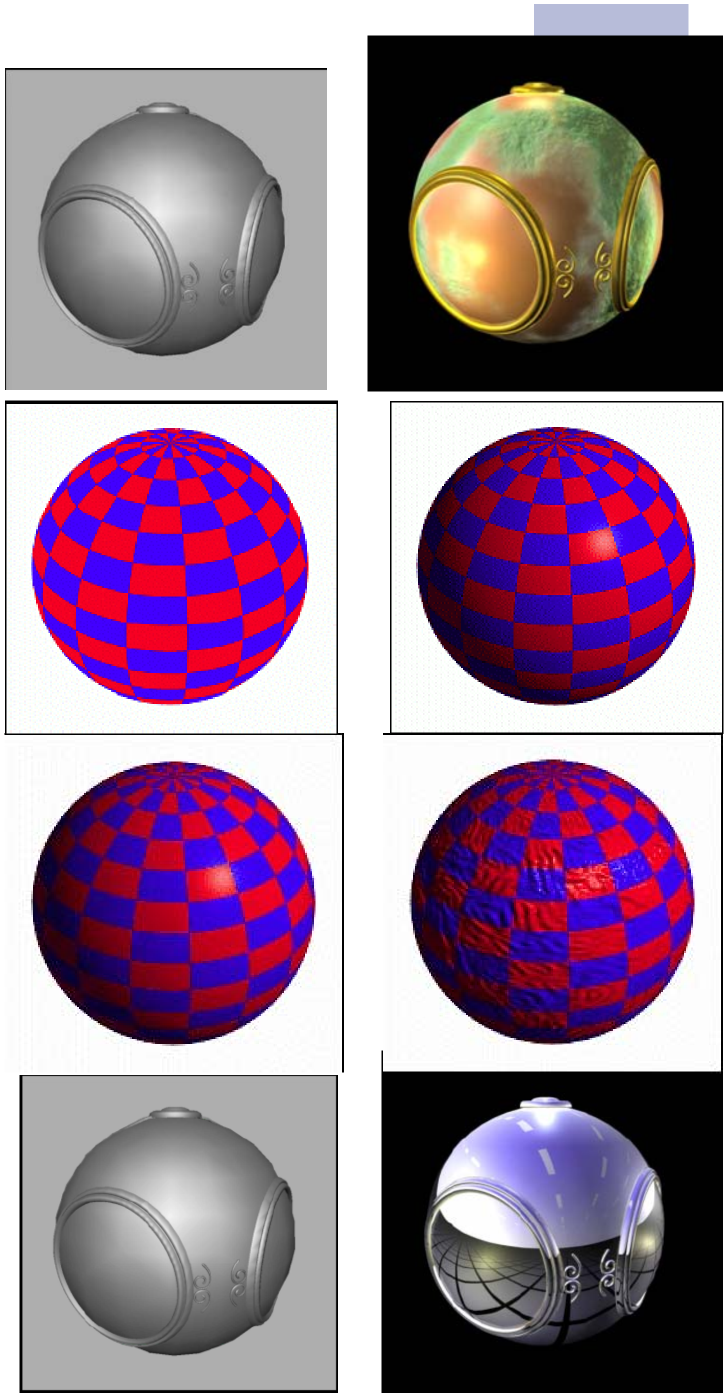
✓ Models are available on line in texture "libraries" of cars, people, construction materials, etc.

✓ Custom textures from scanned photographs or

✓ Using an interactive paint program to create bitmaps

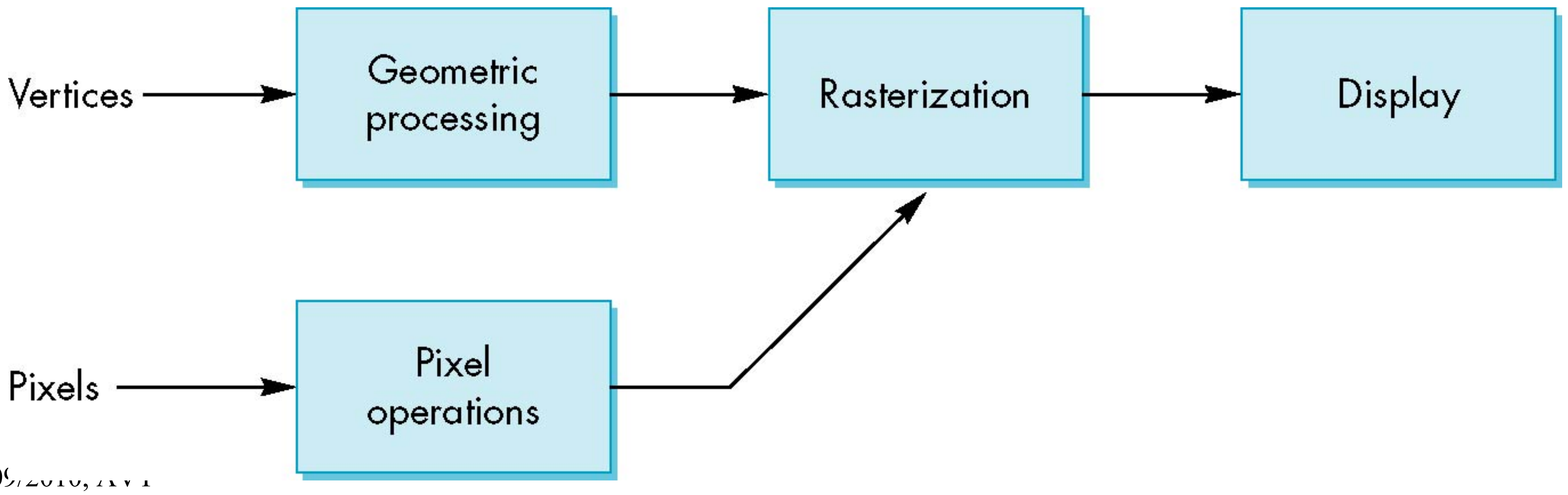
Texturing methods

- **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