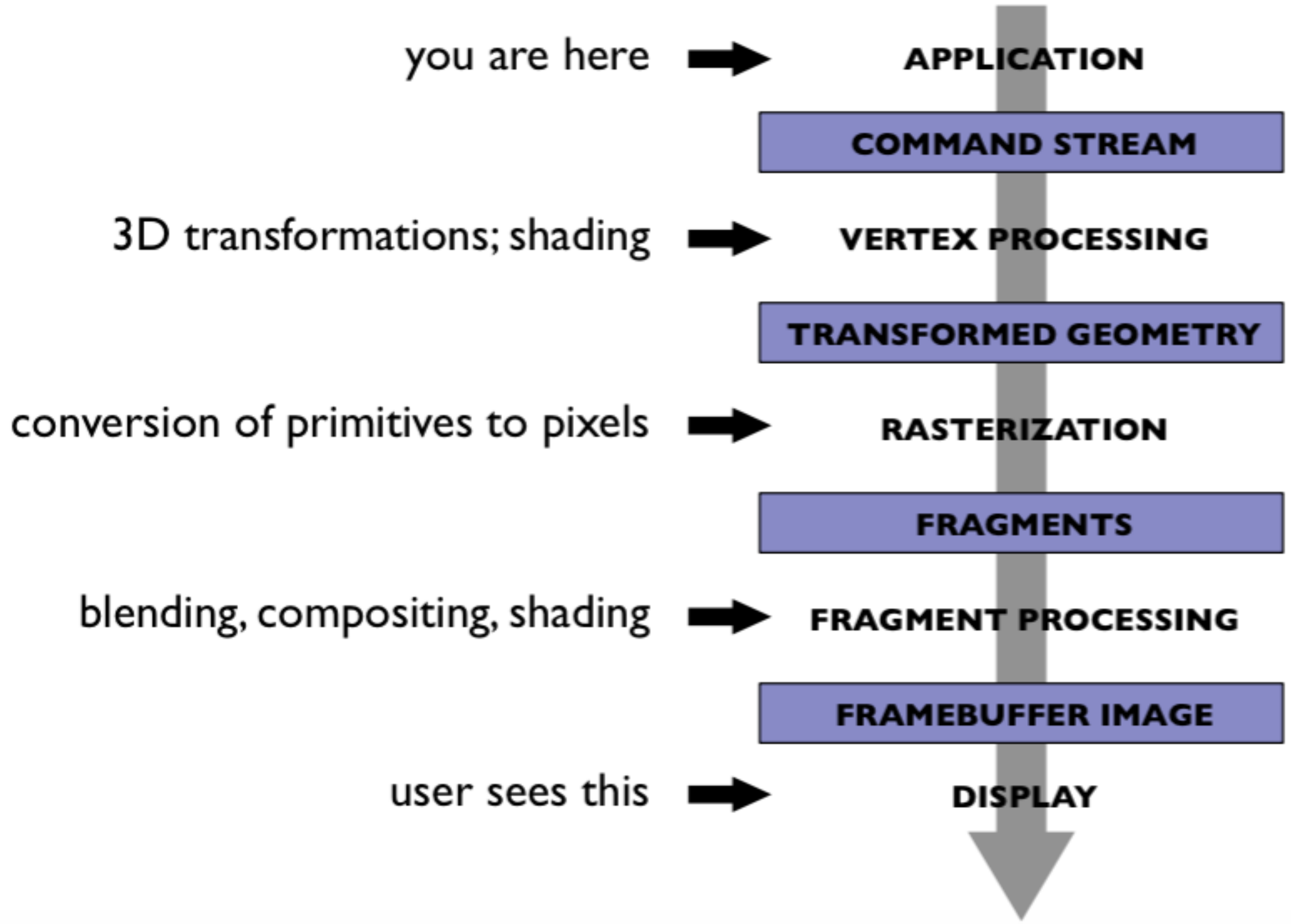
MAC420/5744: Introdução à Computação Gráfica

Marcel P. Jackowski
mjack@ime.usp.br

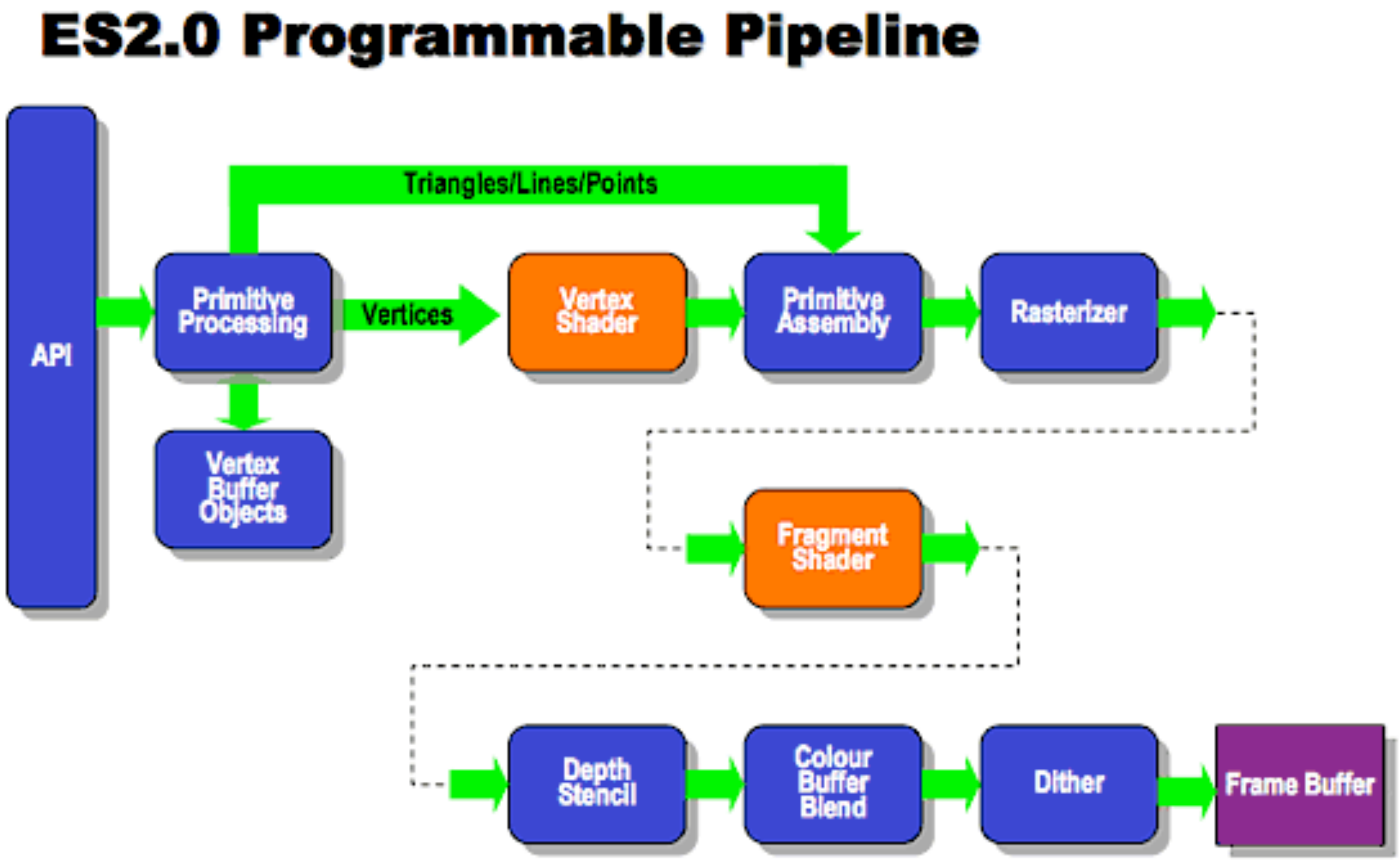
Aula #13: Rasterização II

Some slides by Steve Marschner, Cornell CS4620/5620

Pipeline


you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
user sees this
DISPLAY

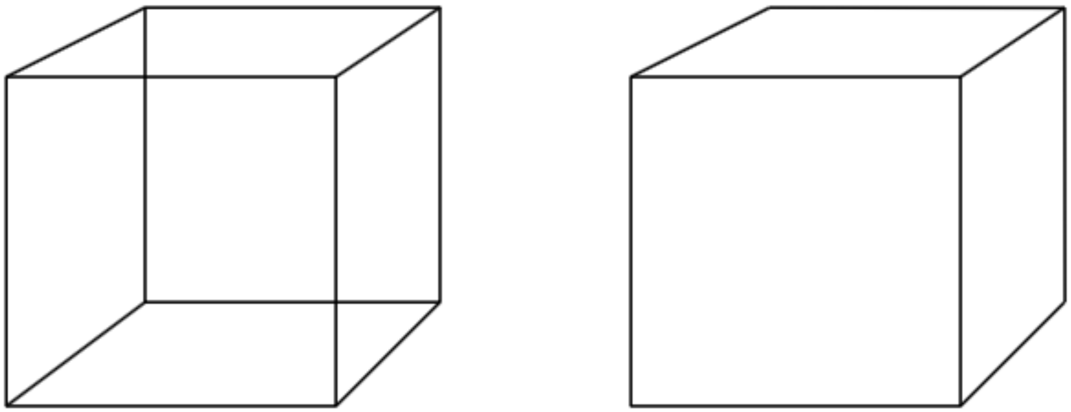
Pipeline OpenGL



Remoção de superfícies escondidas

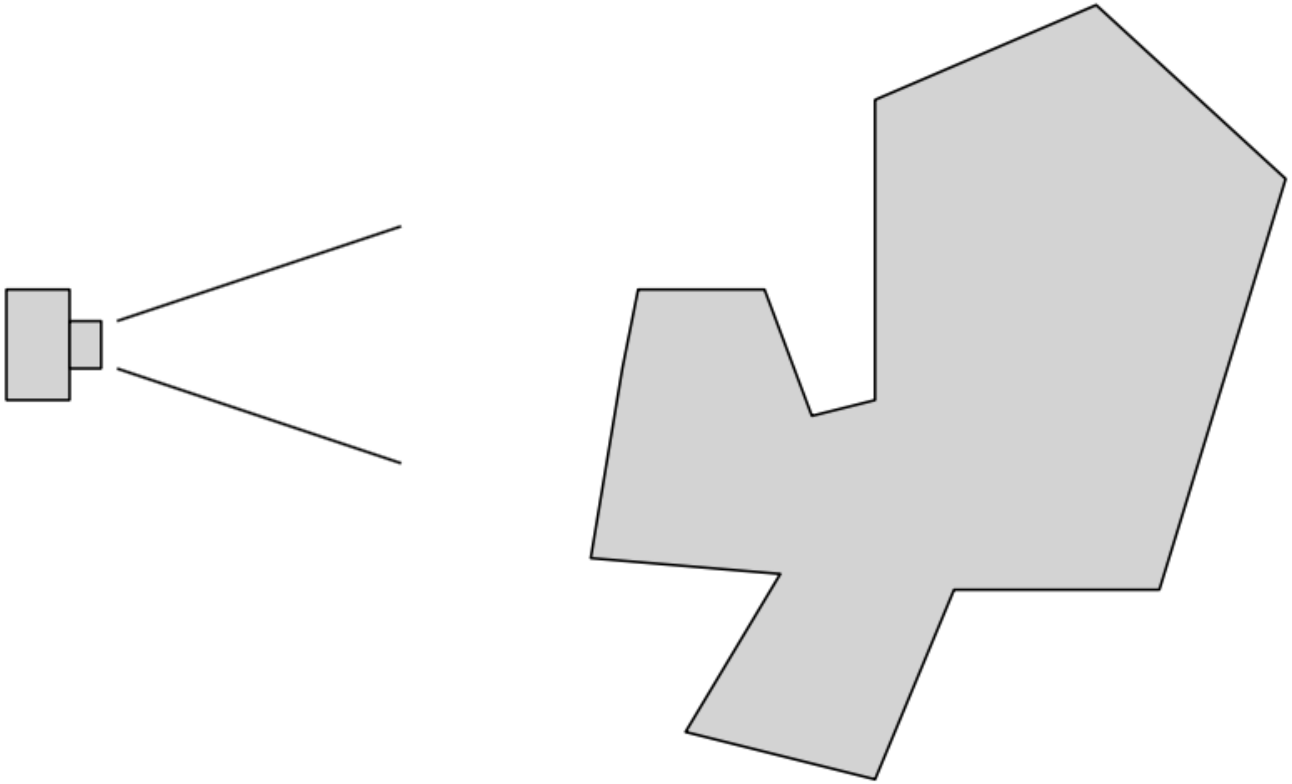
We have discussed how to map primitives to image space

- projection and perspective are depth cues
- occlusion is another very important cue



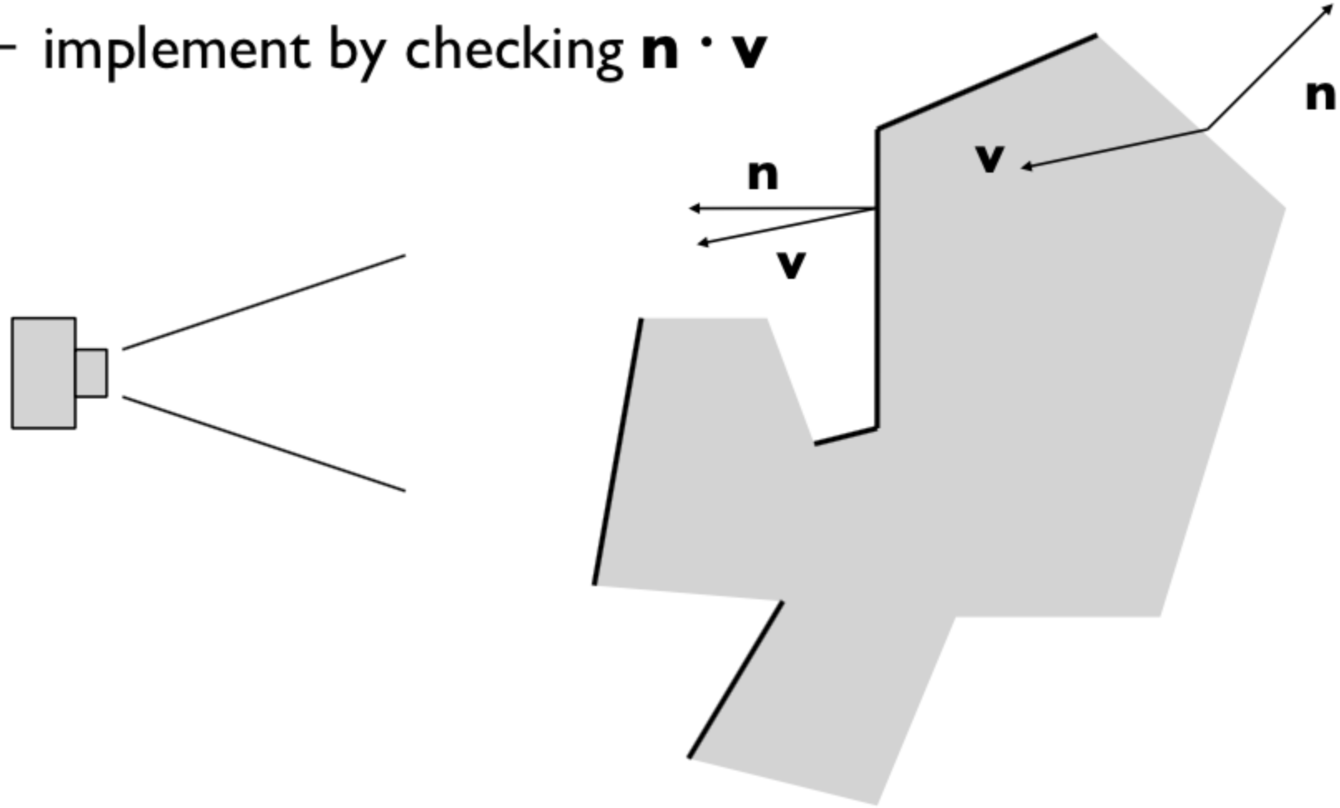
Face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera

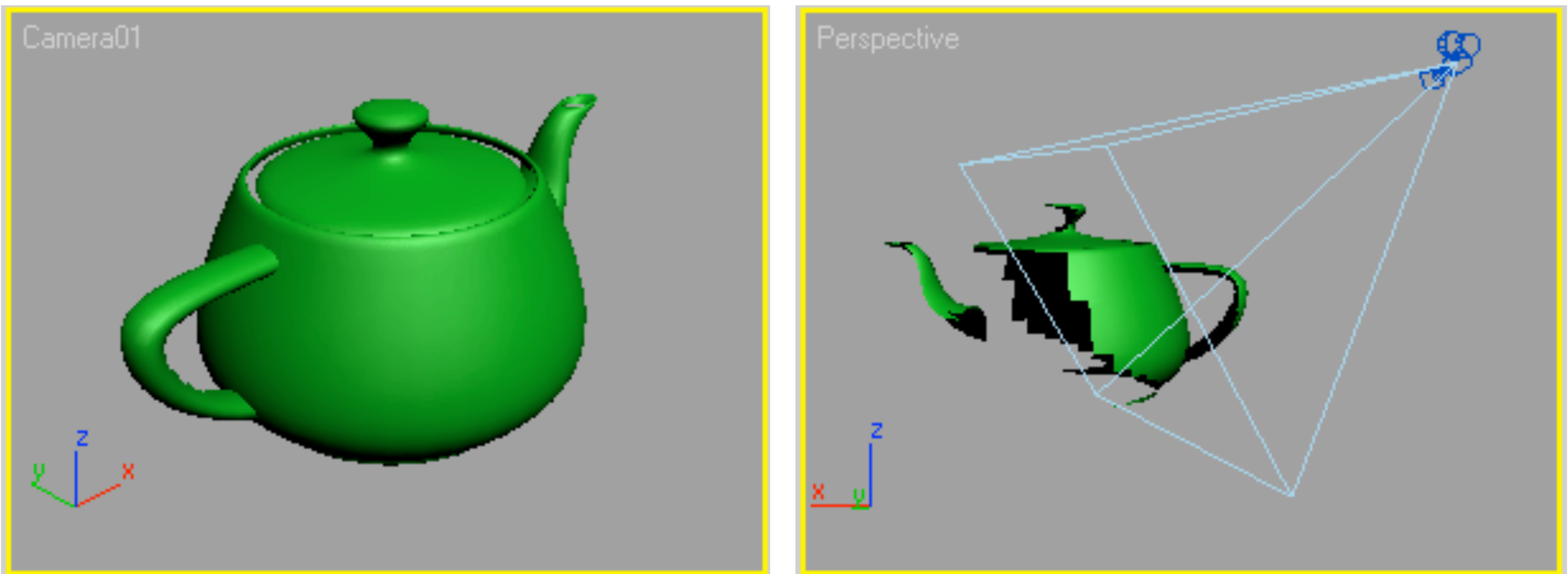


Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**



Back face culling



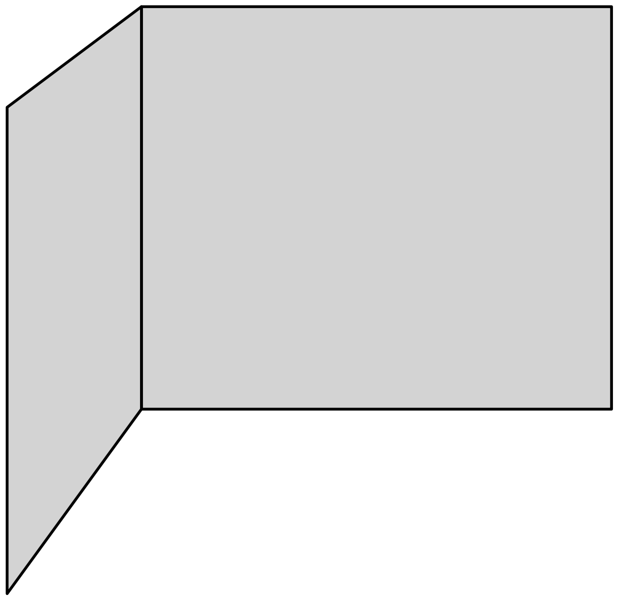

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Algoritmo do pintor

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer

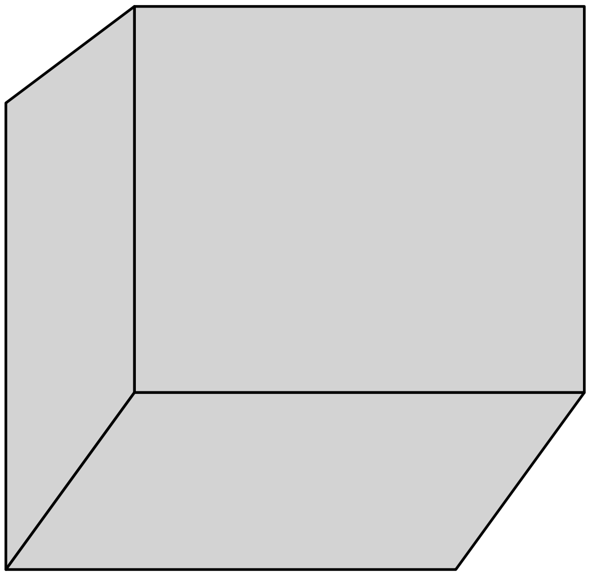
Algoritmo do pintor

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



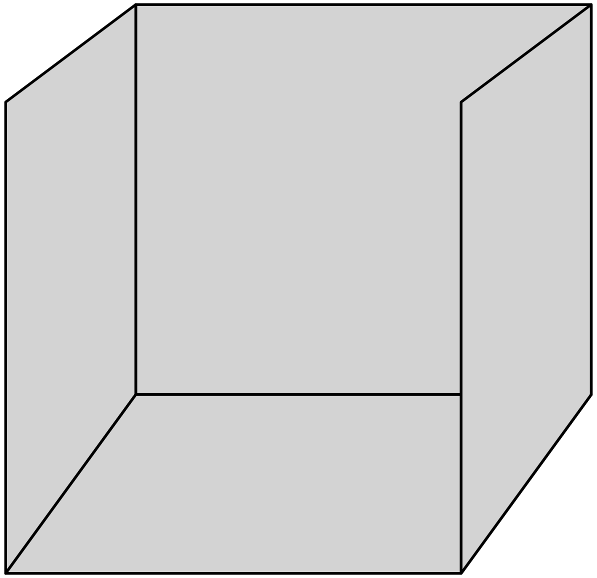
Algoritmo do pintor

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



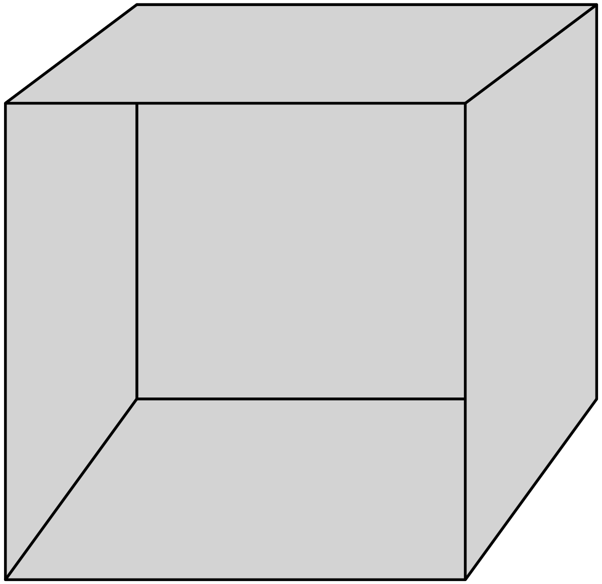
Algoritmo do pintor

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



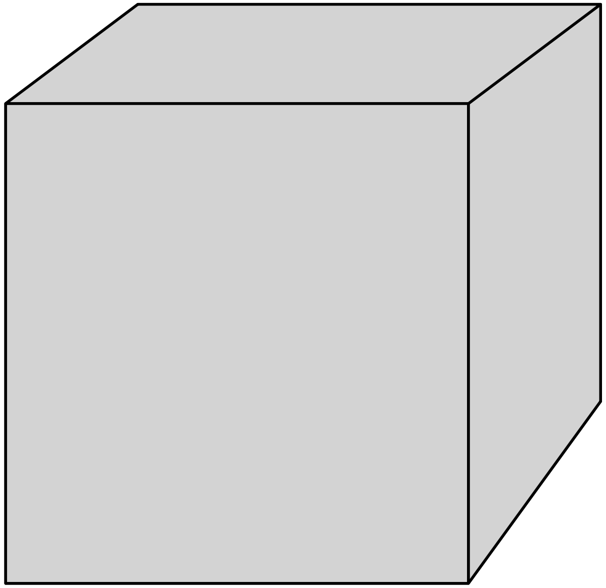
Algoritmo do pintor

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



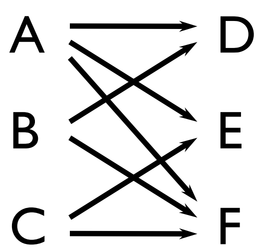
Algoritmo do pintor

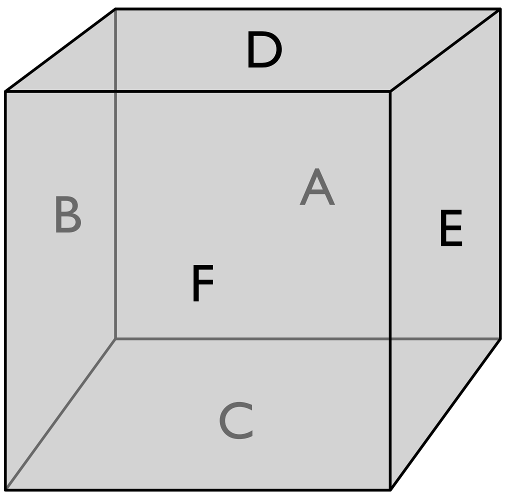
- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



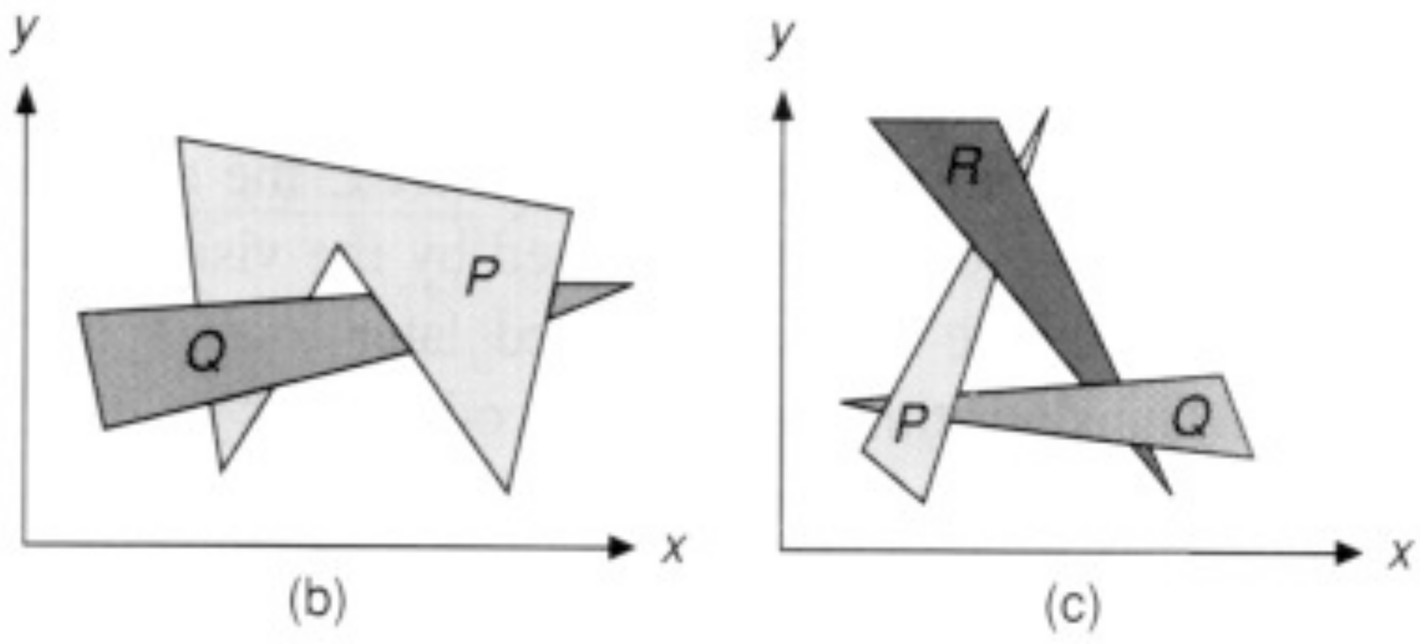
Algoritmo do pintor

- Amounts to a topological sort of the graph of occlusions
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort



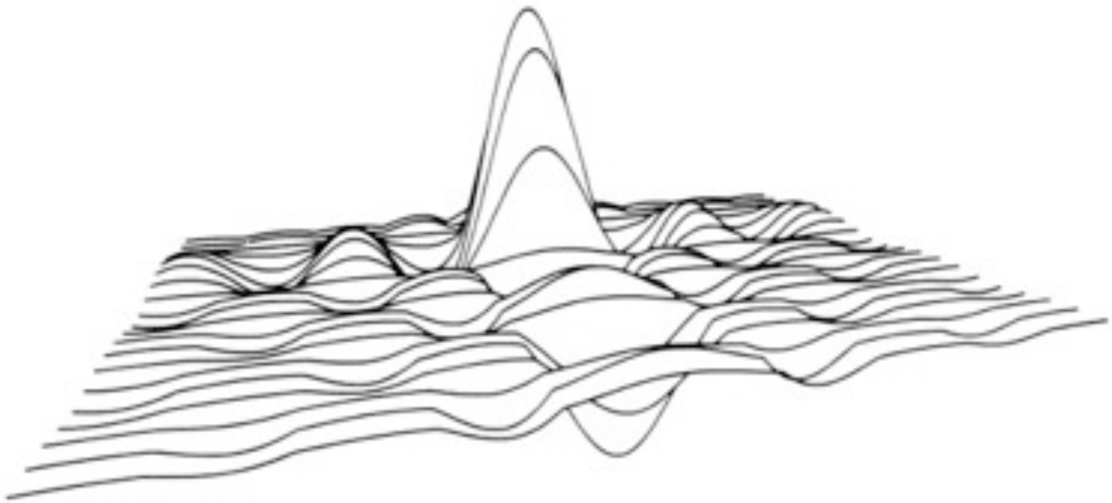


Algoritmo do pintor



Algoritmo do pintor

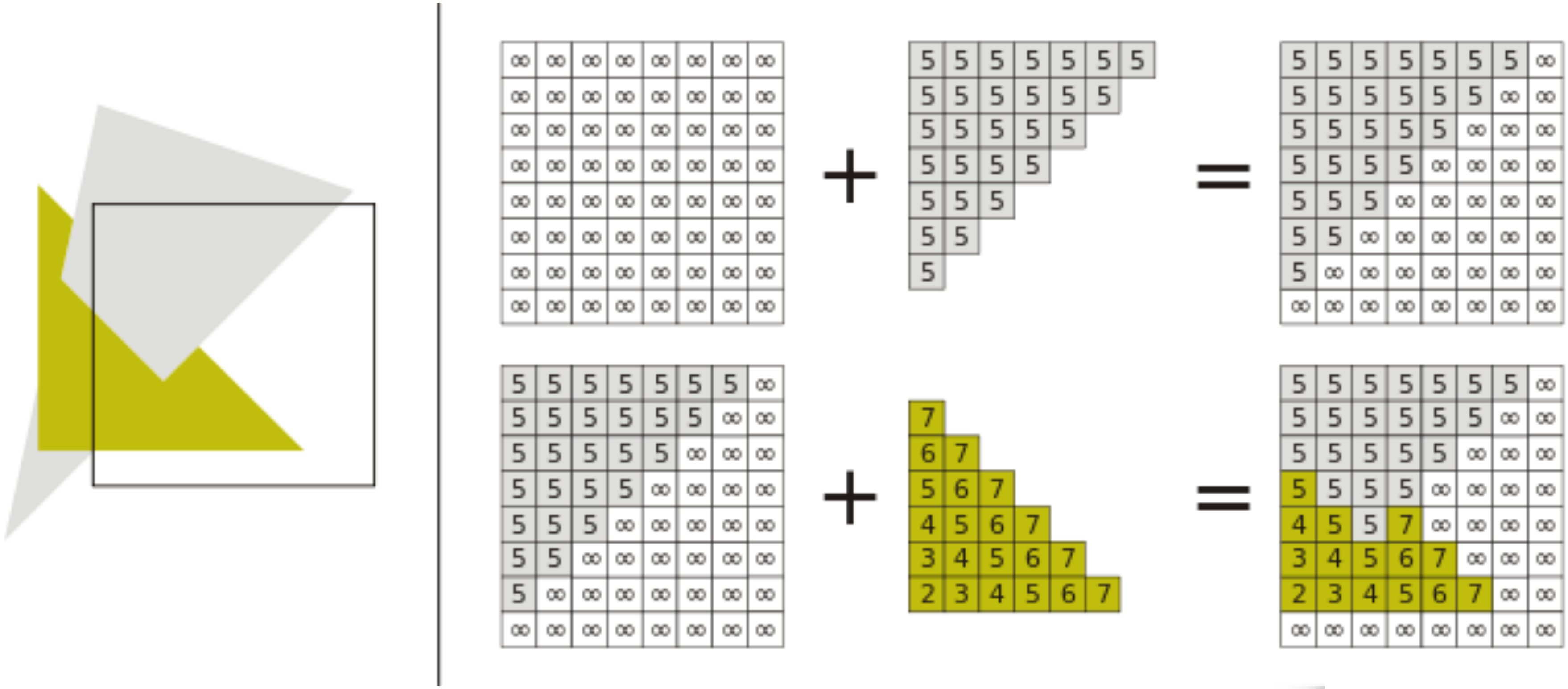
- Useful when a valid order is easy to come by
- Compatible with alpha blending



Método z-buffer

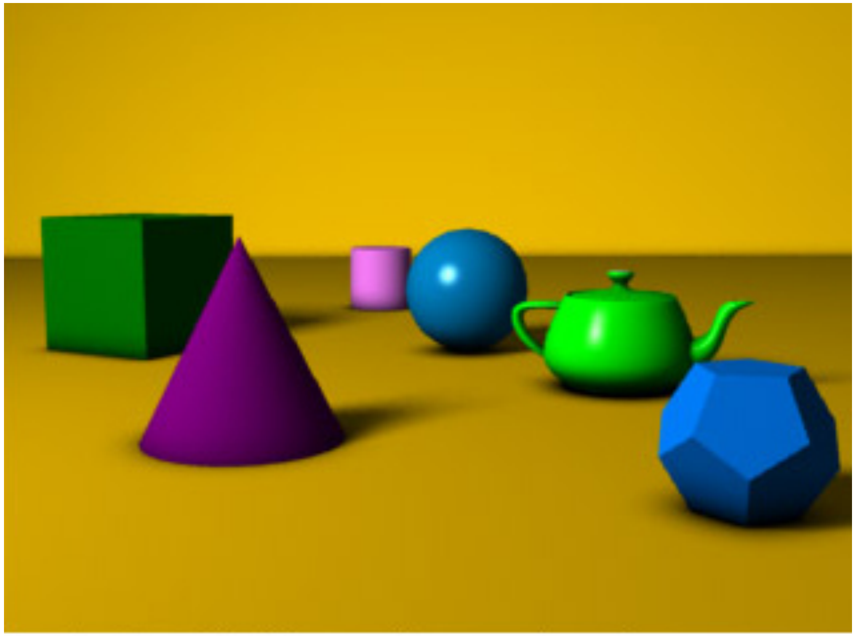
- In many (most) applications maintaining a z sort is too expensive
  - changes all the time as the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - allocate extra channel per pixel to keep track of closest depth so far
  - when drawing, compare object's depth to current closest depth and discard if greater
  - this works just like any other compositing operation

O z-buffer



– another example of a memory-intensive brute force approach that works and has become the standard

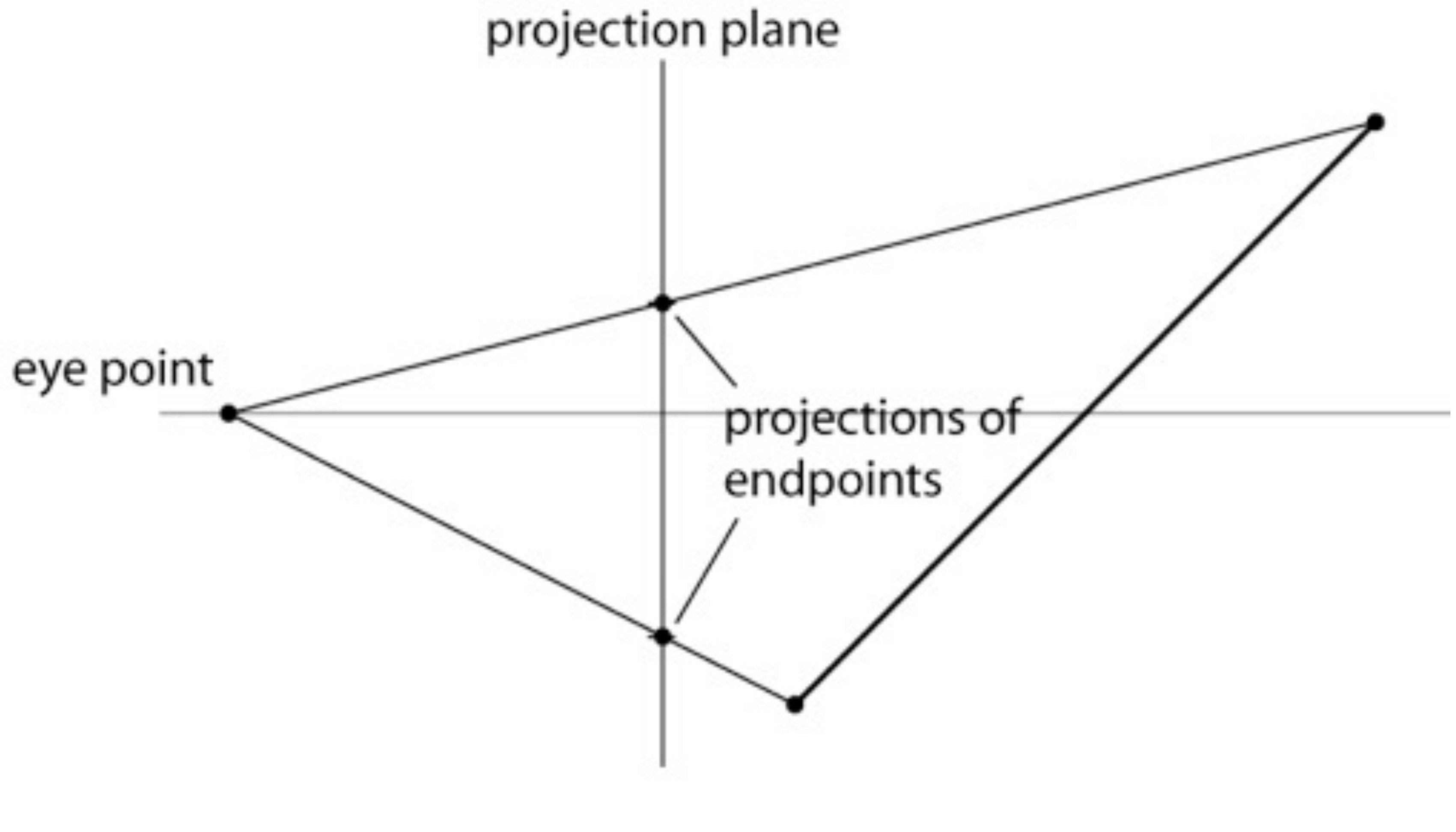
O z-buffer



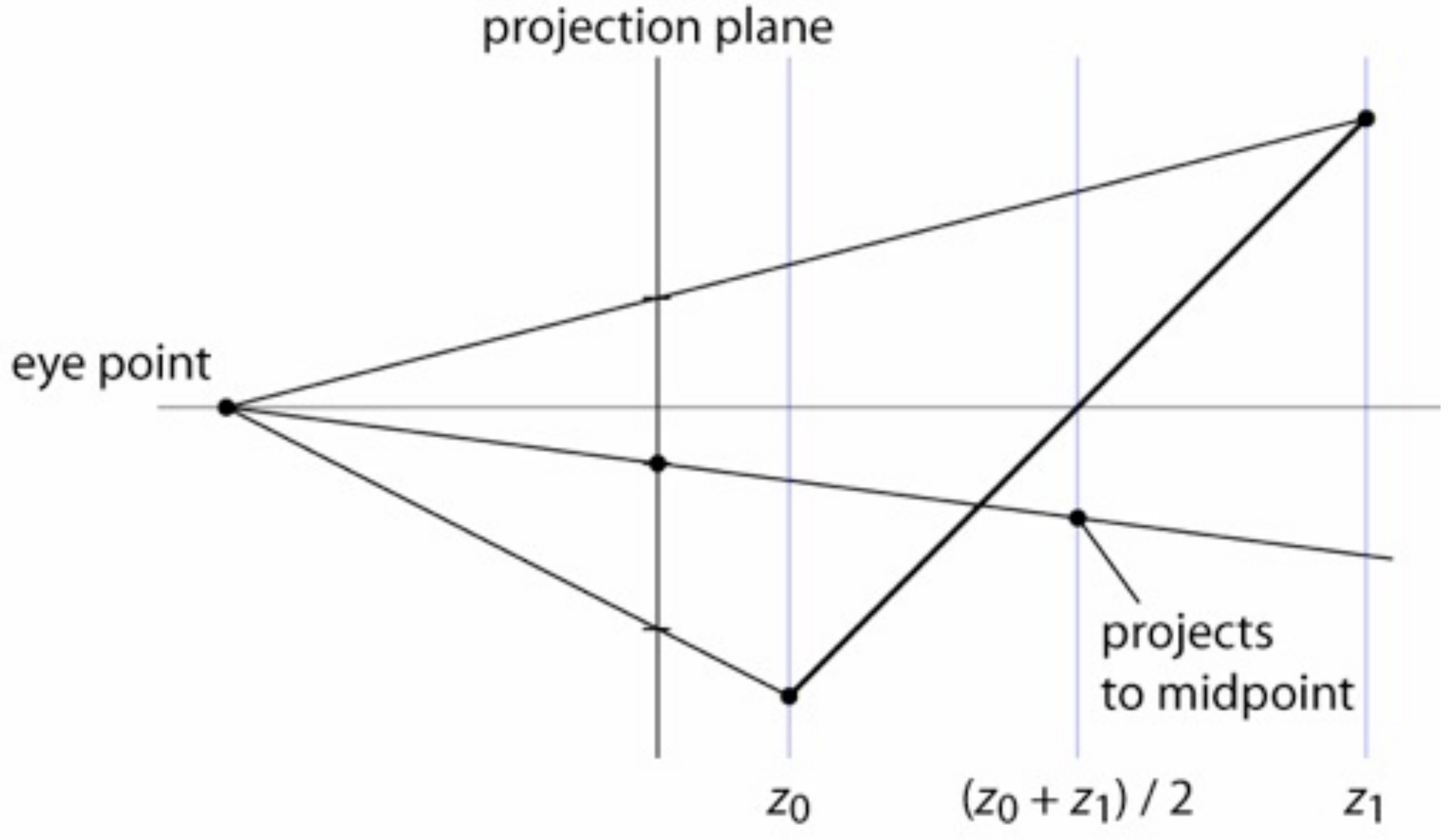
Precisão do z-buffer

- The precision is distributed between the near and far clipping planes
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances

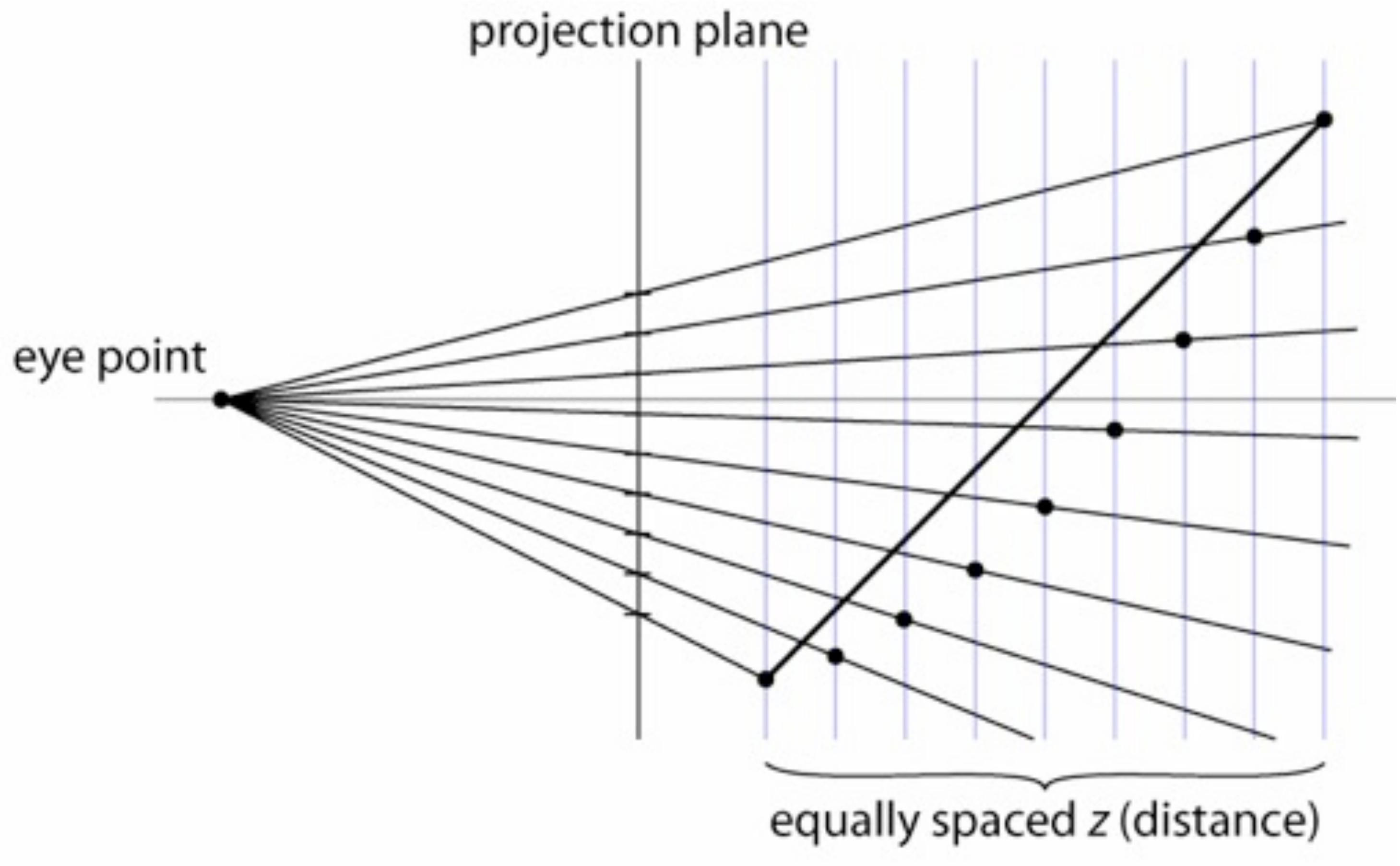
Interpolação de profundidade



Interpolação de profundidade

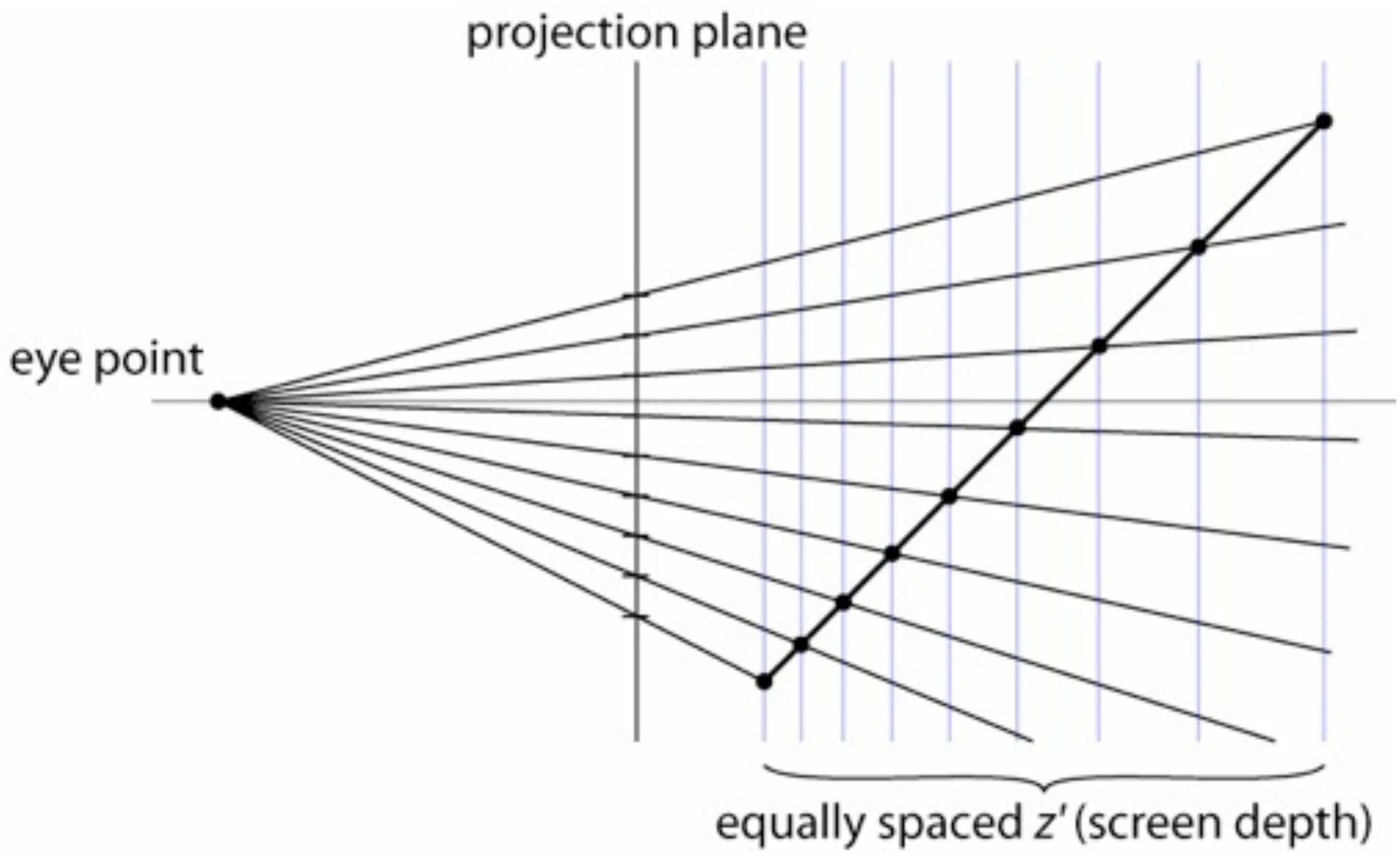


Interpolação de profundidade



linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolação de profundidade



linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolação de perspectiva

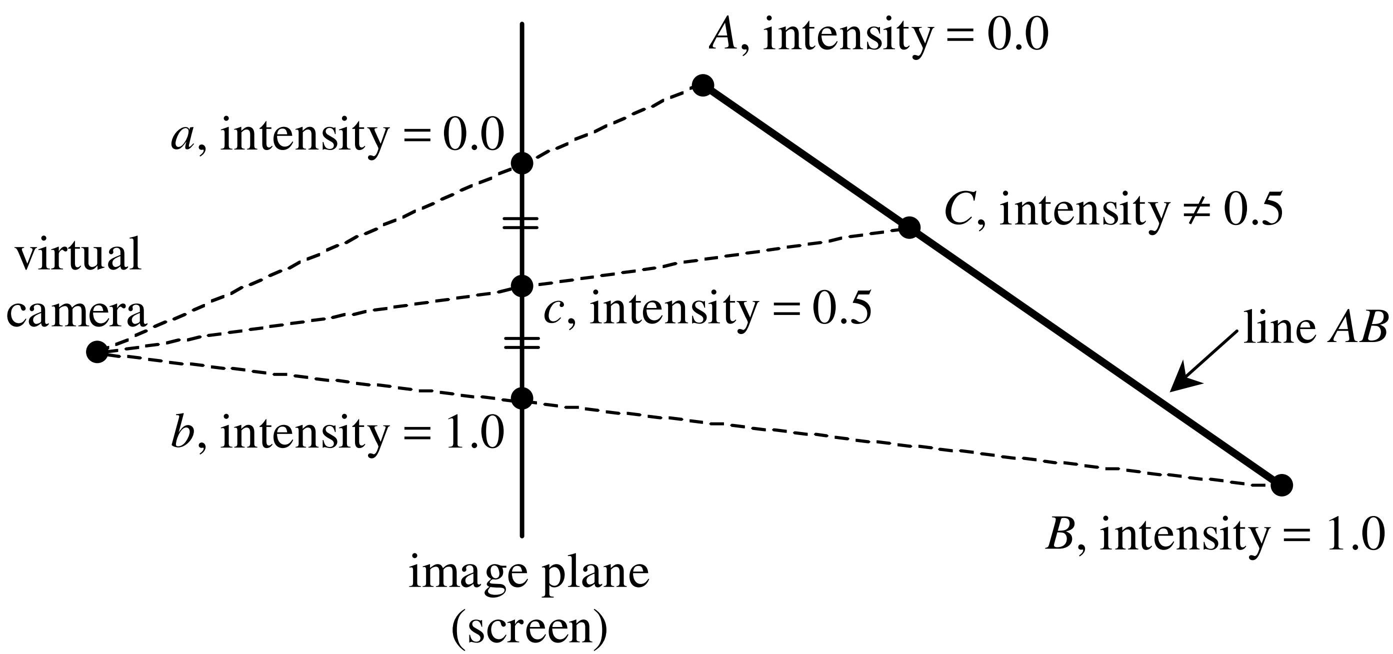
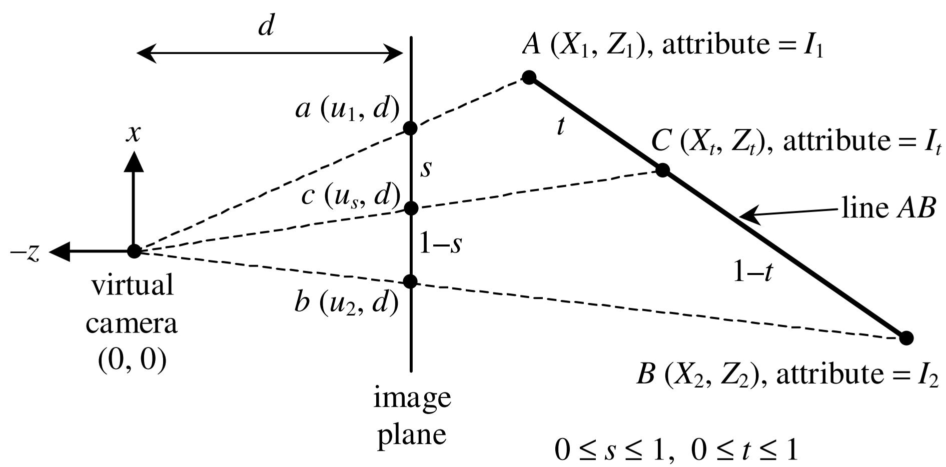


***Figure 1**: Straightforward linear interpolation of attribute values in the screen space (or in the image plane) does not always produce perspective-correct results.*

Interpolação de perspectiva



$$\frac{X_1}{Z_1} = \frac{u_1}{d} \quad \Rightarrow \quad X_1 = \frac{u_1 Z_1}{d}, \tag{1}$$

$$\frac{X_2}{Z_2} = \frac{u_2}{d} \quad \Rightarrow \quad X_2 = \frac{u_2 Z_2}{d}, \tag{2}$$

$$\frac{X_t}{Z_t} = \frac{u_s}{d} \quad \Rightarrow \quad Z_t = \frac{dX_t}{u_s}. \tag{3}$$

By linearly interpolating in the image plane (or screen space), we have

$$u_s = u_1 + s(u_2 - u_1). \tag{4}$$

By linearly interpolating across the primitive in the camera coordinate system, we have

$$X_t = X_1 + t(X_2 - X_1), \tag{5}$$

$$Z_t = Z_1 + t(Z_2 - Z_1), \tag{6}$$

Interpolação de perspectiva

$$t = \frac{sZ_1}{sZ_1 + (1-s)Z_2} \cdot \tag{10}$$

Substituting (10) into (6), we have

$$Z_t = Z_1 + \frac{sZ_1}{sZ_1 + (1-s)Z_2}(Z_2 - Z_1), \tag{11}$$

which can be simplified to

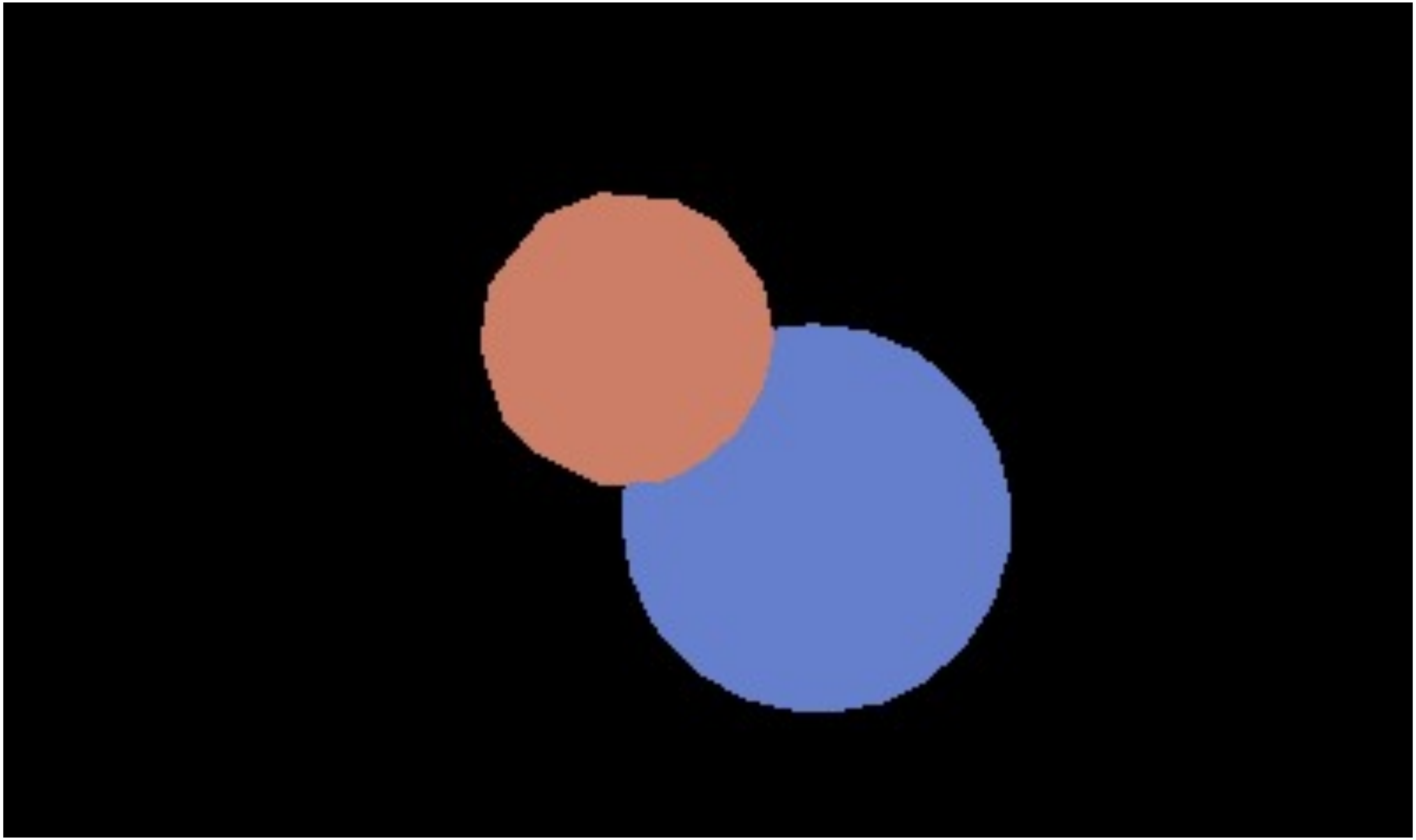
$$Z_t = \frac{1}{\frac{1}{Z_1} + s\left(\frac{1}{Z_2} - \frac{1}{Z_1}\right)}. \tag{12}$$

Equation (12) tells us that the z-value at point c in the image plane can be correctly derived by just *linearly interpolating* between $1/Z_1$ and $1/Z_2$, and then compute the reciprocal of the interpolated result. For z-buffer purpose, the final reciprocal need not even be computed, because all we need is to reverse the comparison operation during z-value comparison.

Pipeline mínimo

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - pass through color
- Fragment stage (output: color)
  - write to color planes

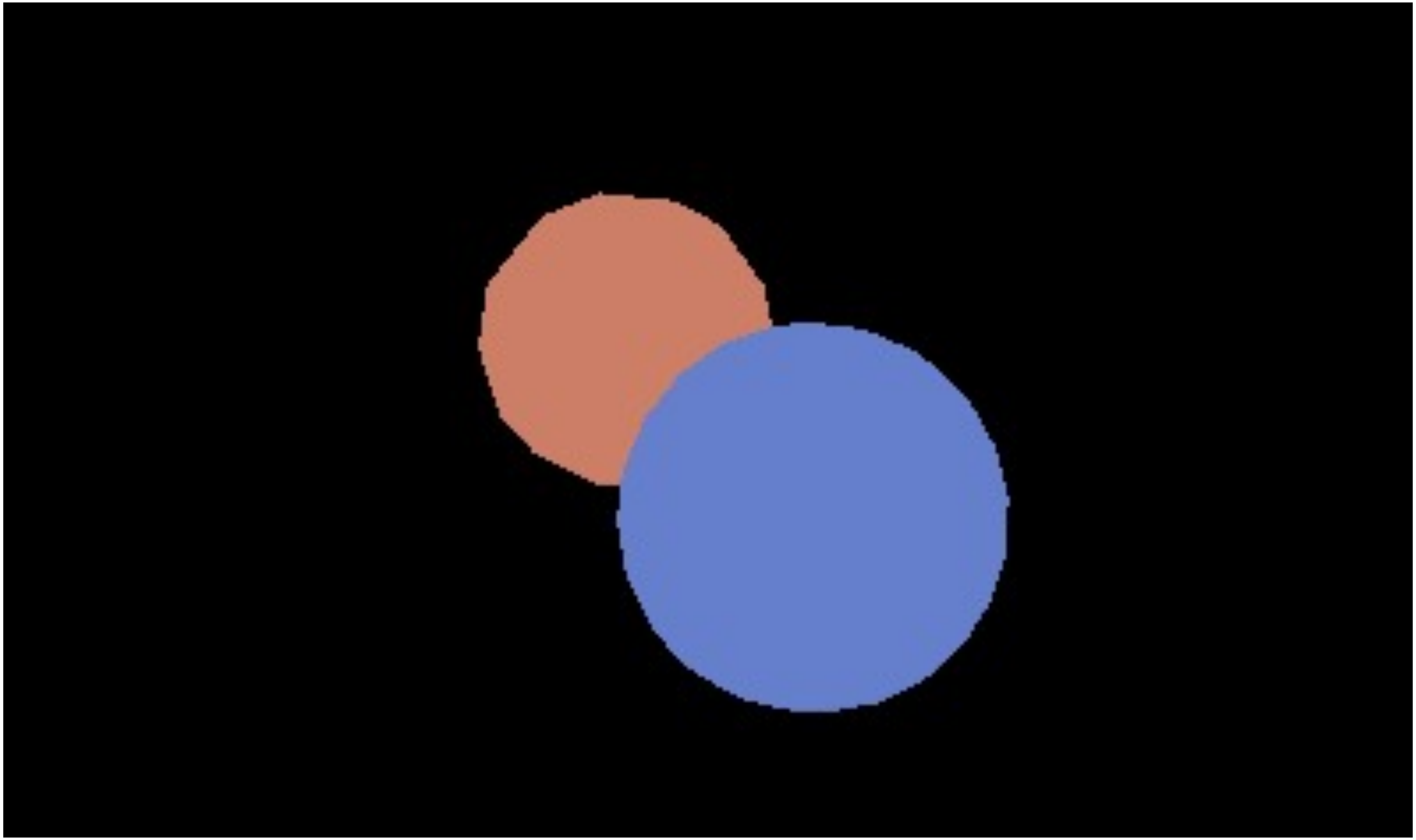
Resultado do pipeline mínimo



Pipeline para um z-buffer básico

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - interpolated parameter: z' (screen z)
  - pass through color
- Fragment stage (output: color, z')
  - write to color planes only if interpolated z' < current z'

Resultado do pipeline com z-buffer



Tonalização facetada (*Flat shading*)

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

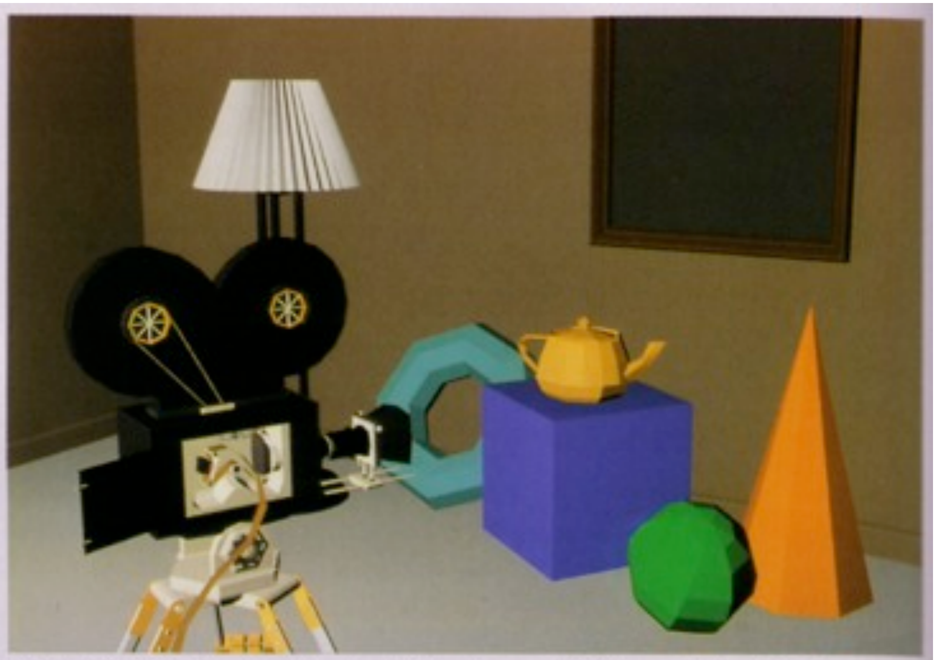


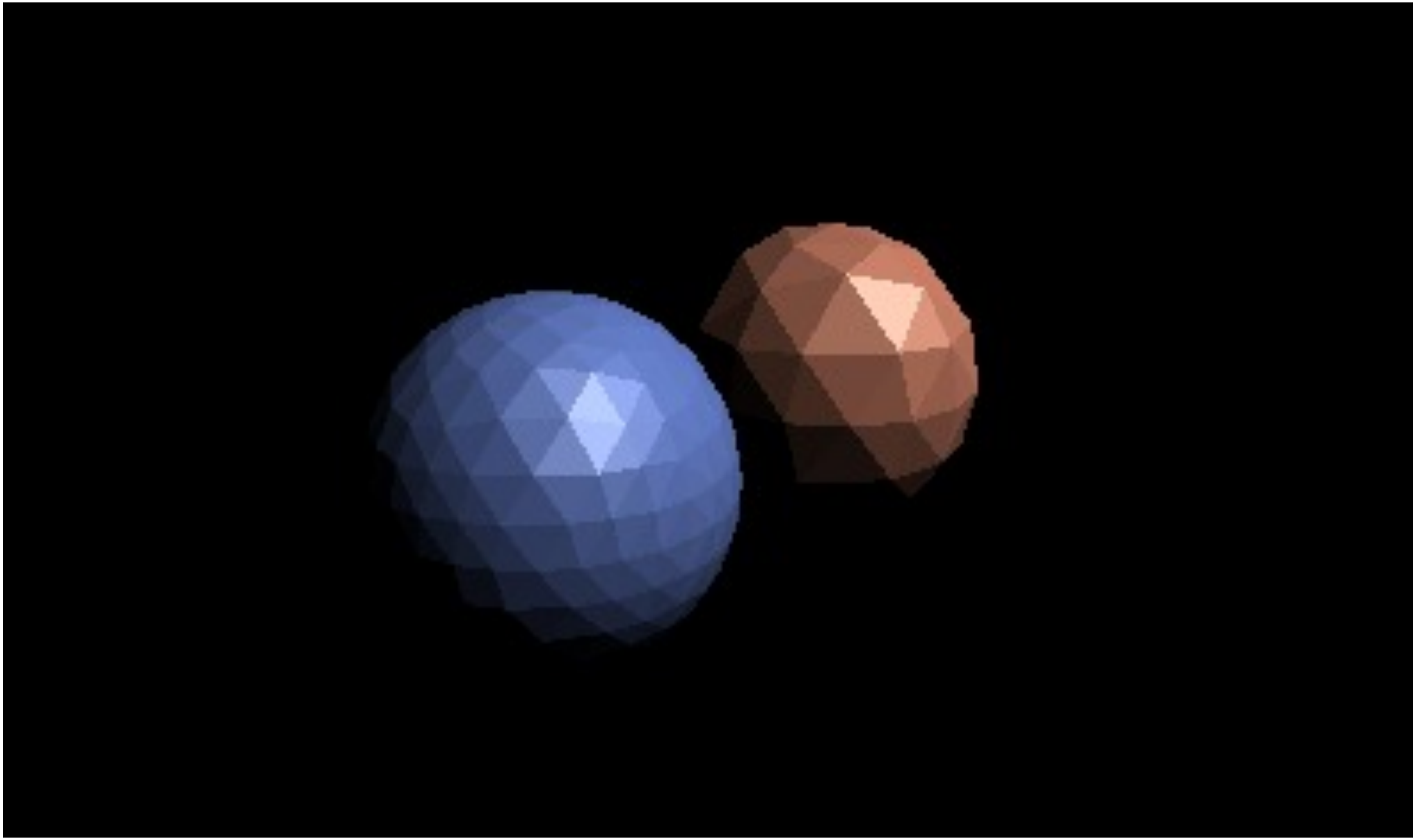
Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline para *Flat Shading*

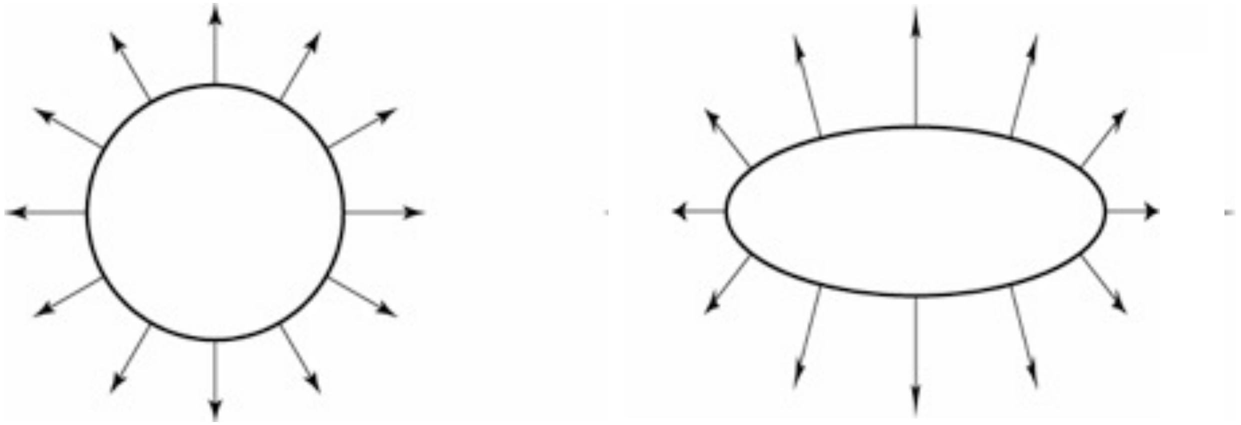
- Vertex stage (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: *z'* (screen *z*)
  - pass through color
- Fragment stage (output: color, *z'*)
  - write to color planes only if interpolated *z'* < current *z'*

Resultado do pipeline para *Flat Shading*



Transformação de vetores normais

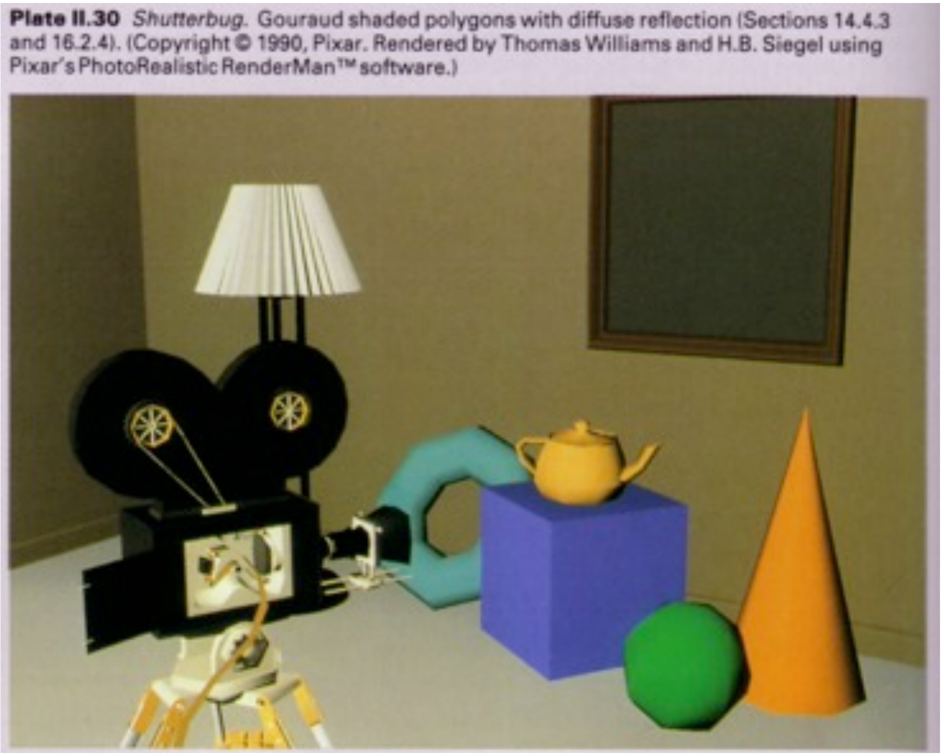
- Transforming surface normals
  - differences of points (and therefore tangents) transform OK
  - normals do not --> use inverse transpose matrix



have: $\mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$
want: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0$
so set $X = (M^T)^{-1}$
then: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

Gouraud shading

- Often we're trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - "Gouraud shading"
  - "Smooth shading"

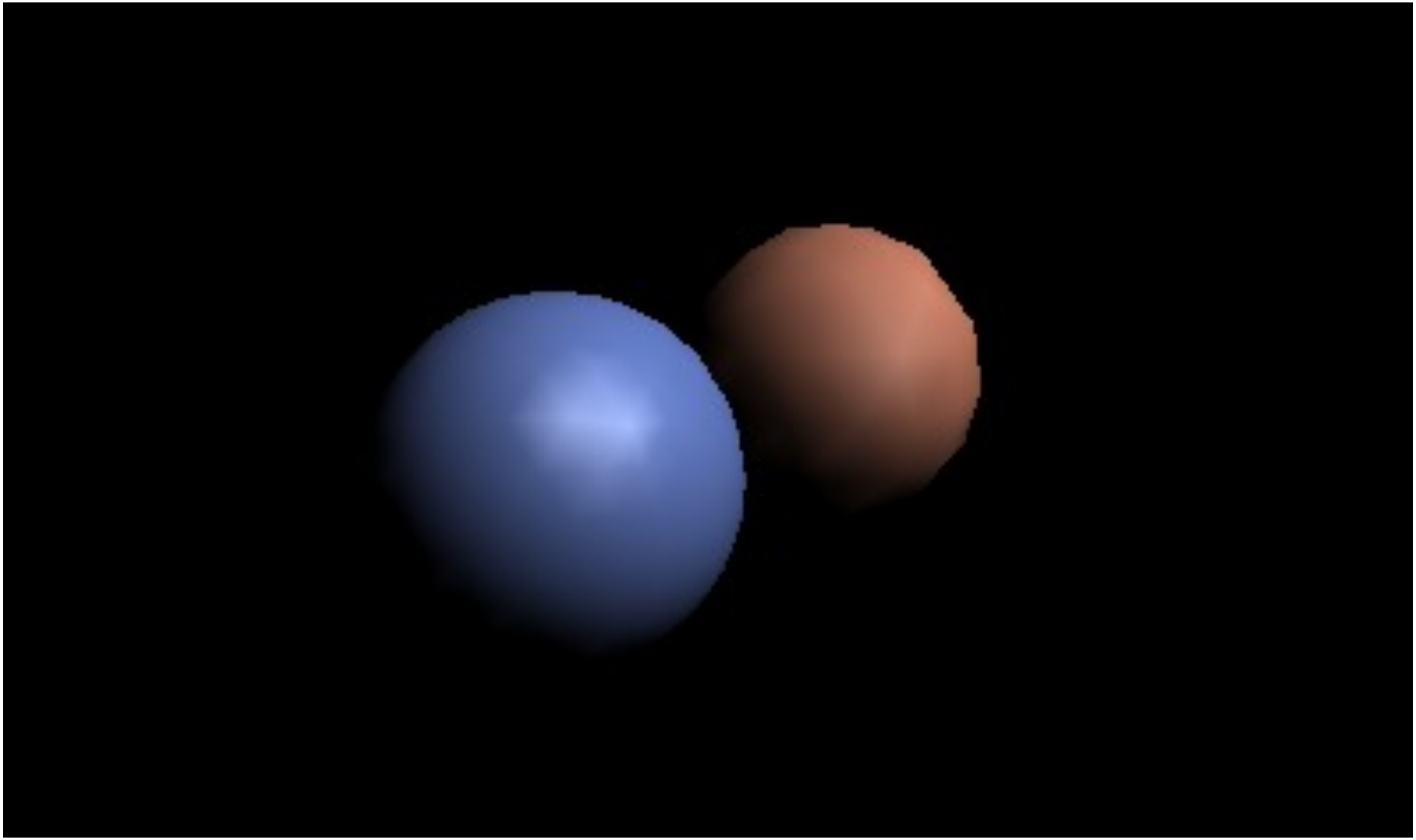


[Foley et al.]

Pipeline para *Gouraud* shading

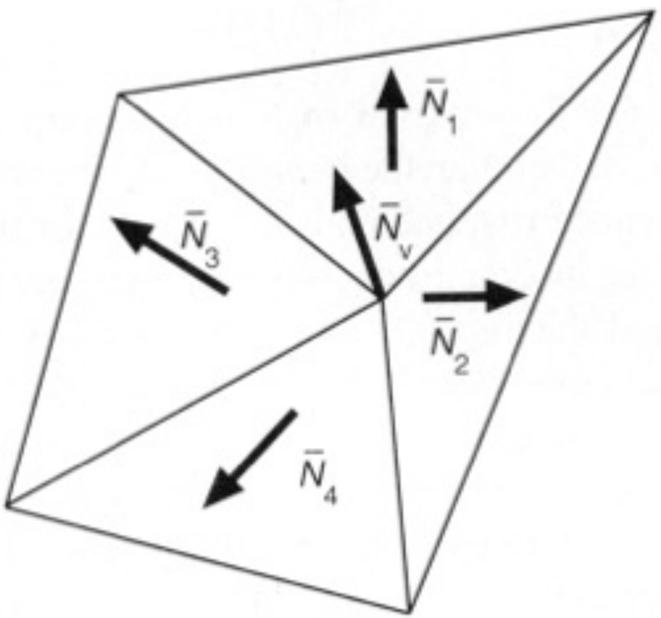
- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - compute shaded color per vertex
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: *z'* (screen *z*); *r*, *g*, *b* color
- Fragment stage (output: color, *z'*)
  - write to color planes only if interpolated *z'* < current *z'*

Resultado com *Gouraud* shading



Normais

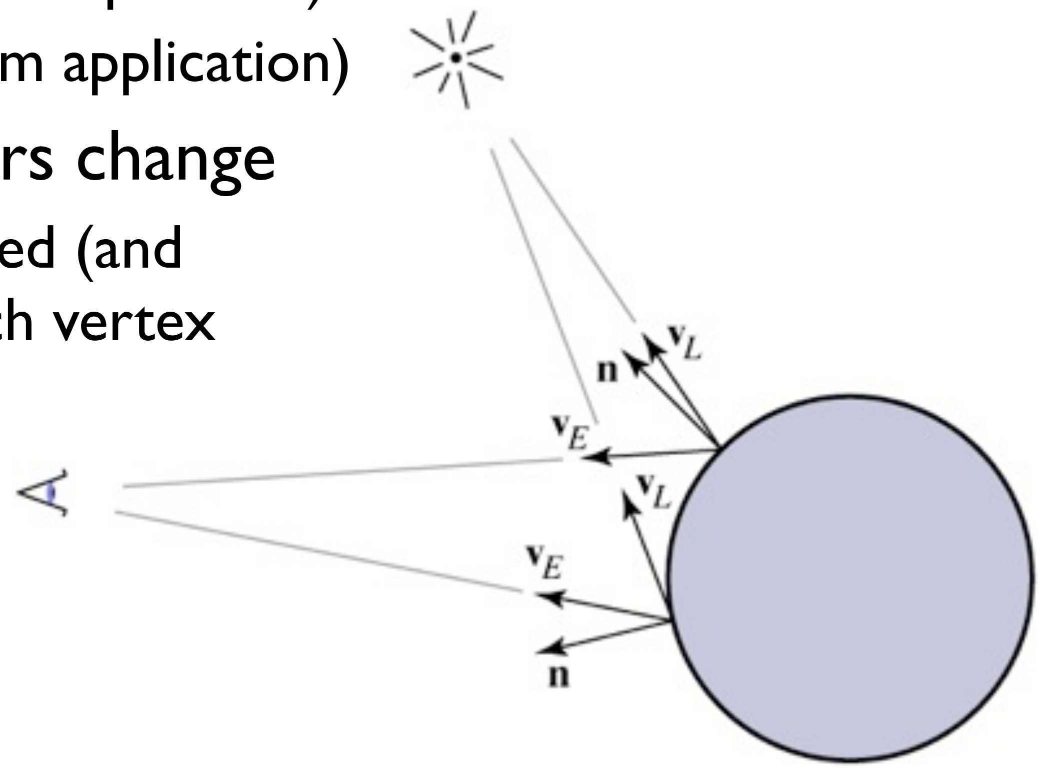
- Need normals at vertices to compute Gouraud shading
- Best to get vtx. normals from the underlying geometry
  - e. g. spheres example
- Otherwise have to infer vtx. normals from triangles
  - simple scheme: average surrounding face normals



$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$

Câmera e luzes

- Phong illumination requires geometric information:
  - light vector (function of position)
  - eye vector (function of position)
  - surface normal (from application)
- Light and eye vectors change
  - need to be computed (and normalized) for each vertex

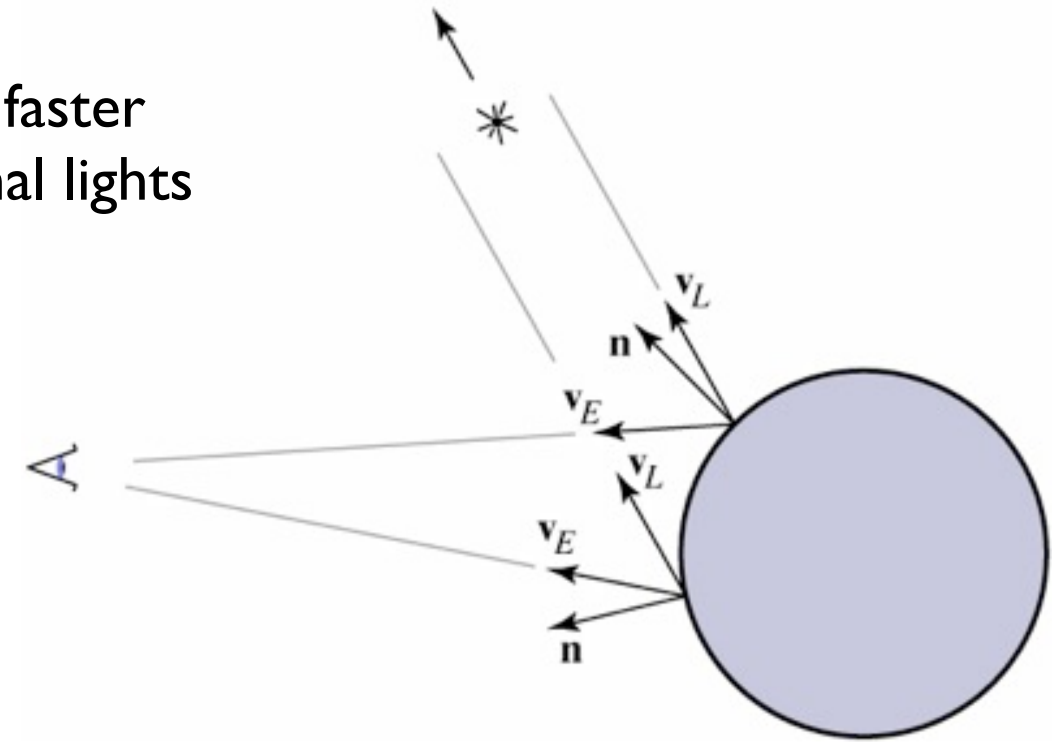


Câmera e luzes locais vs. no infinito

- Look at case when eye or light is far away:
  - distant light source: nearly parallel illumination
  - distant eye point: nearly orthographic projection
  - in both cases, eye or light vector changes very little
- Optimization: approximate eye and/or light as infinitely far away

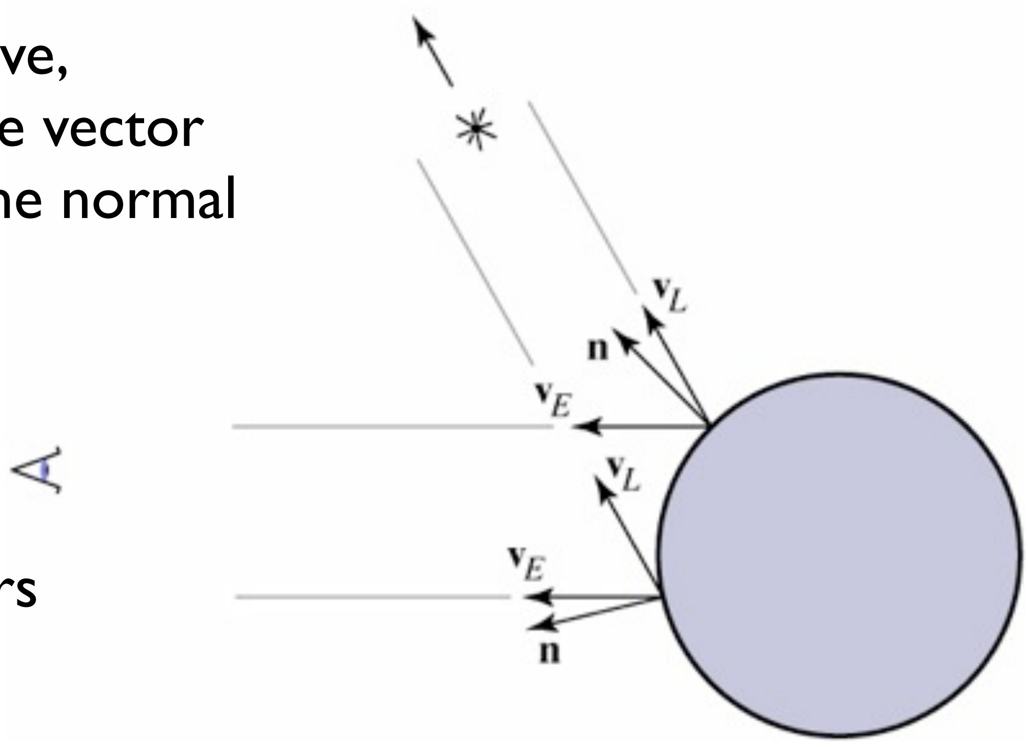
Luzes direcionais

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position $[x \;\; y \;\; z \;\; 0]$
  - many pipelines are faster if you use directional lights



Câmera no infinito

- Orthographic camera
  - projection direction is constant
- "Infinite viewer"
  - even with perspective, can approximate eye vector using the image plane normal
  - can produce weirdness for wide-angle views
  - Blinn-Phong: light, eye, half vectors all constant!



Gouraud shading

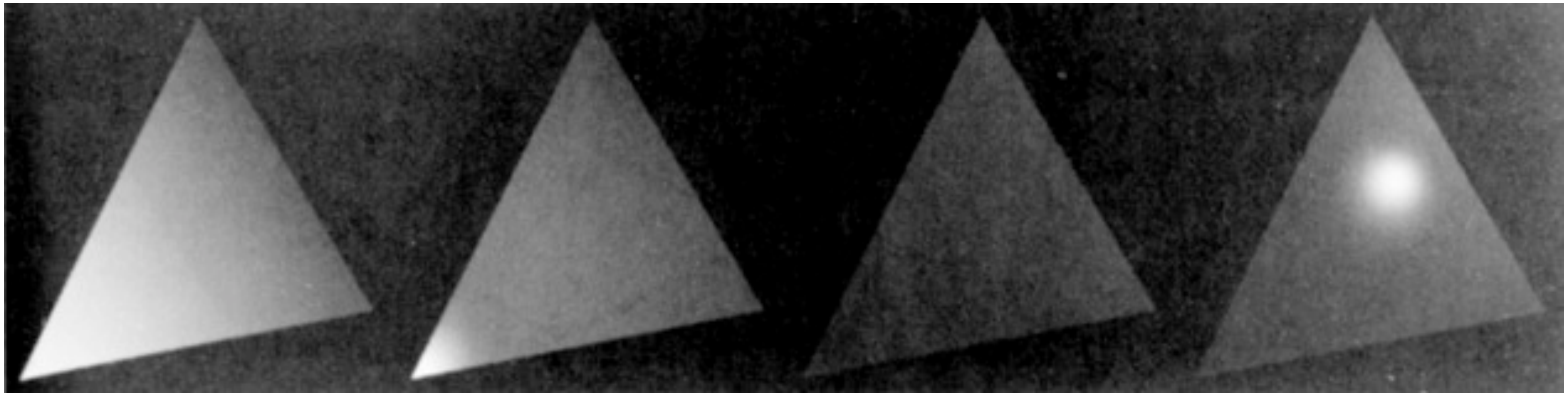
- Can apply Gouraud shading to any illumination model
  - it's just an interpolation method
- Results are not so good with fast-varying models like specular ones
  - problems with any highlights smaller than a triangle

Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Phong shading (por pixel)

- Get higher quality by interpolating the normal
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage



Phong shading

- Bottom line: produces much better highlights

tterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4
yright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using
listic RenderMan™ software.)

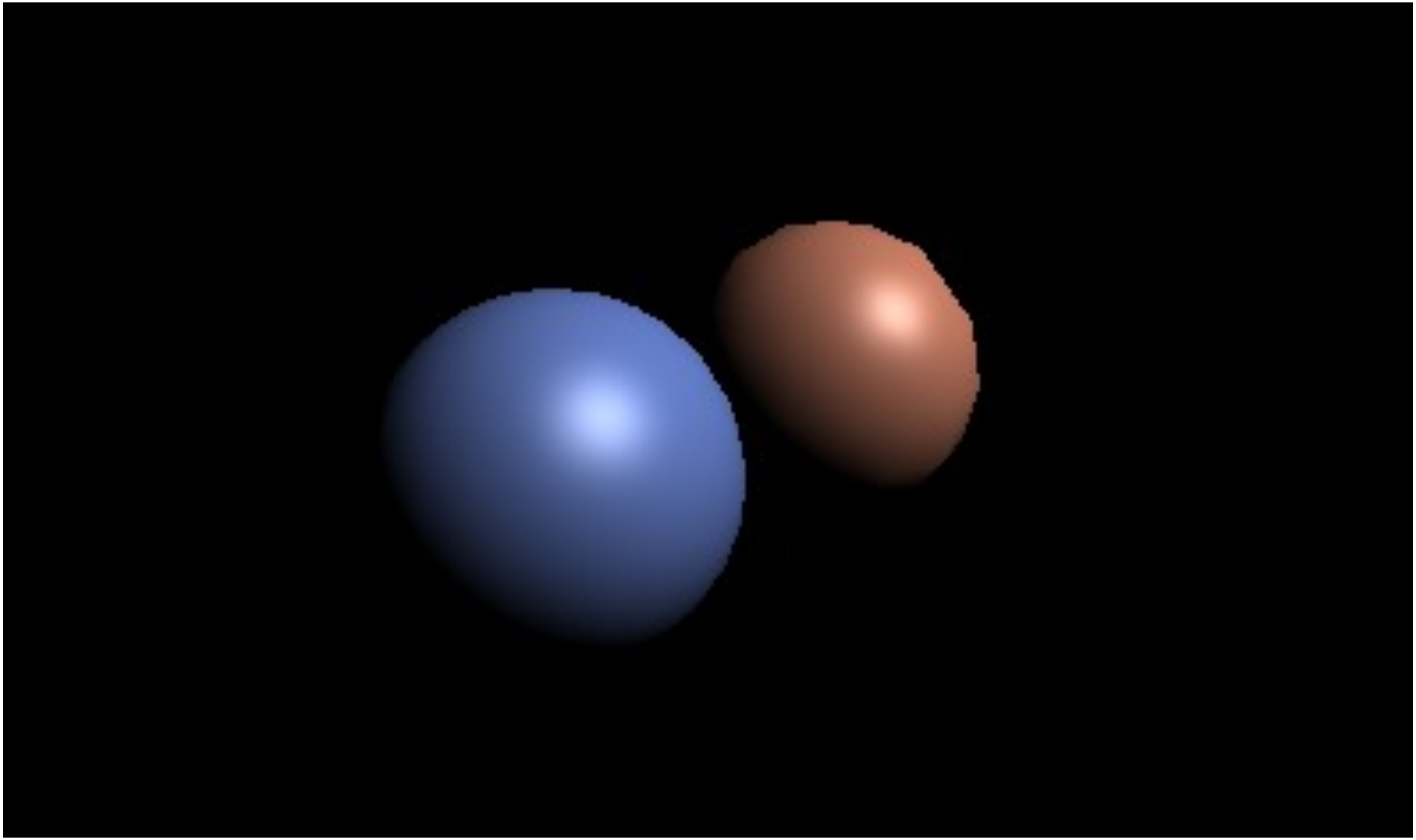
Plate II.32 *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline para *Phong* shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through color
- Rasterizer
  - interpolated parameters: z' (screen z); r, g, b color; x, y, z normal
- Fragment stage (output: color, z')
  - compute shading using interpolated color and normal
  - write to color planes only if interpolated $z' <$ current z'

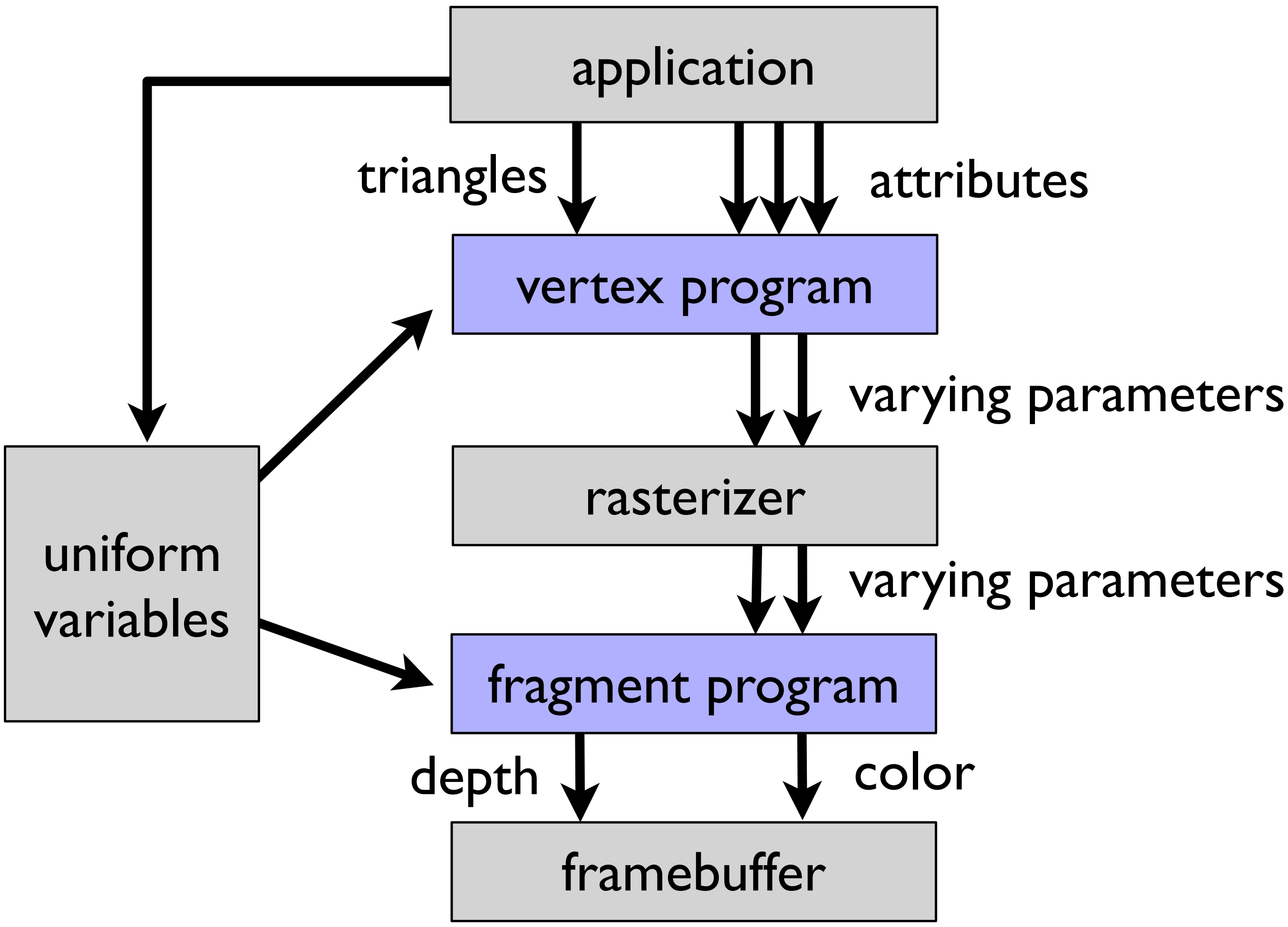
Resultado do pipeline para *Phong*



Pipelines programáveis

- Modern hardware graphics pipelines are flexible
  - programmer defines exactly what happens at each stage
  - do this by writing *shader programs* in domain-specific languages called *shading languages*
  - rasterization is fixed-function, as are some other operations (depth test, many data conversions, …)
- One example: OpenGL and GLSL (**GL S**hading **L**anguage)
  - several types of shaders process primitives and vertices; most basic is the *vertex program*
  - after rasterization, fragments are processed by a *fragment program*

Shaders GLSL



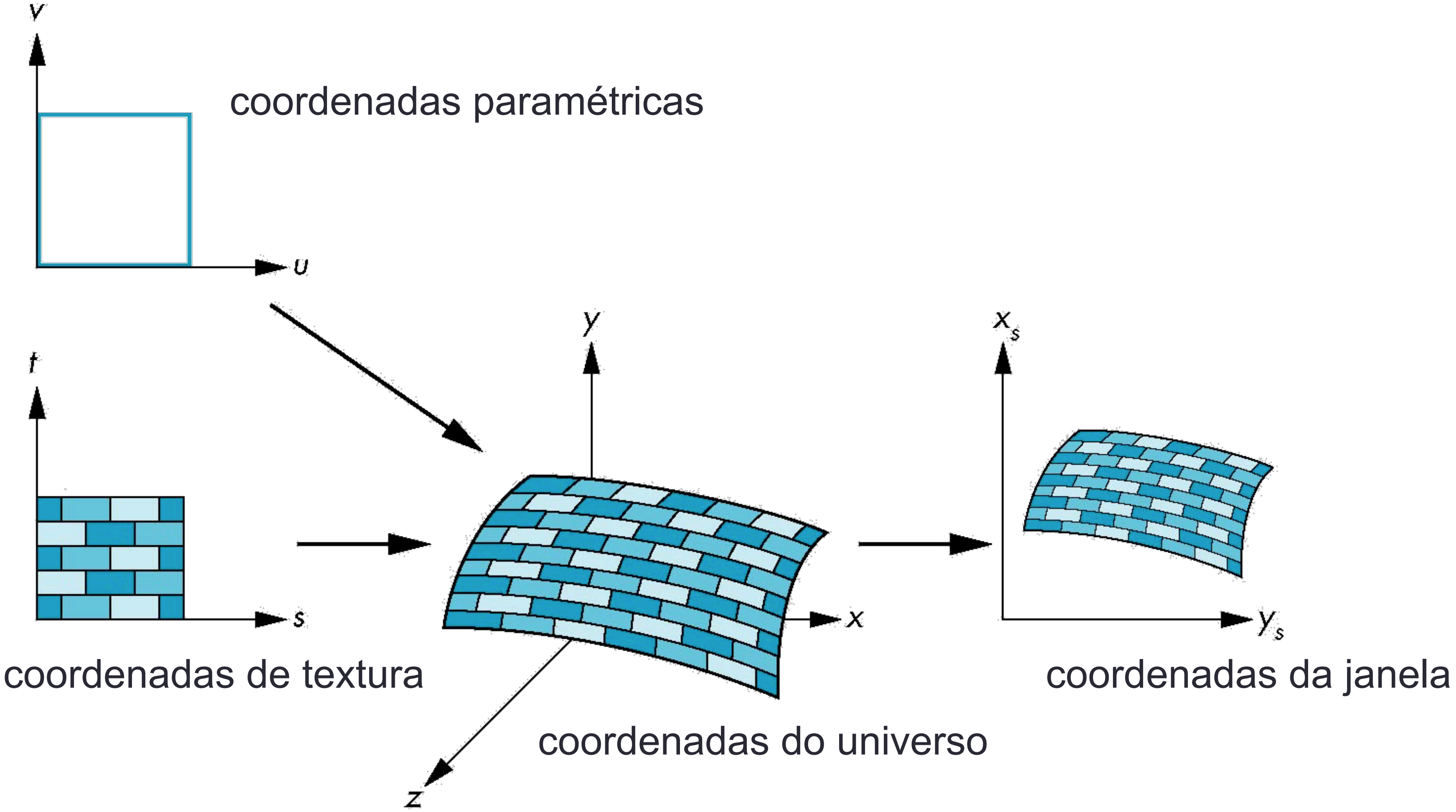
Mapeamento de textura

- Modelos de iluminação não são apropriados para descrever todas as diferenças de cor observáveis em uma superfície
  - Superfícies pintadas com padrões ou imagens
  - Superfícies com padrões de rugosidade
- A princípio, é possível modelar esses detalhes com geometria, usando materiais com propriedades óticas distintas
  - Na prática, esses efeitos são modelados usando uma técnica chamada mapeamento de textura

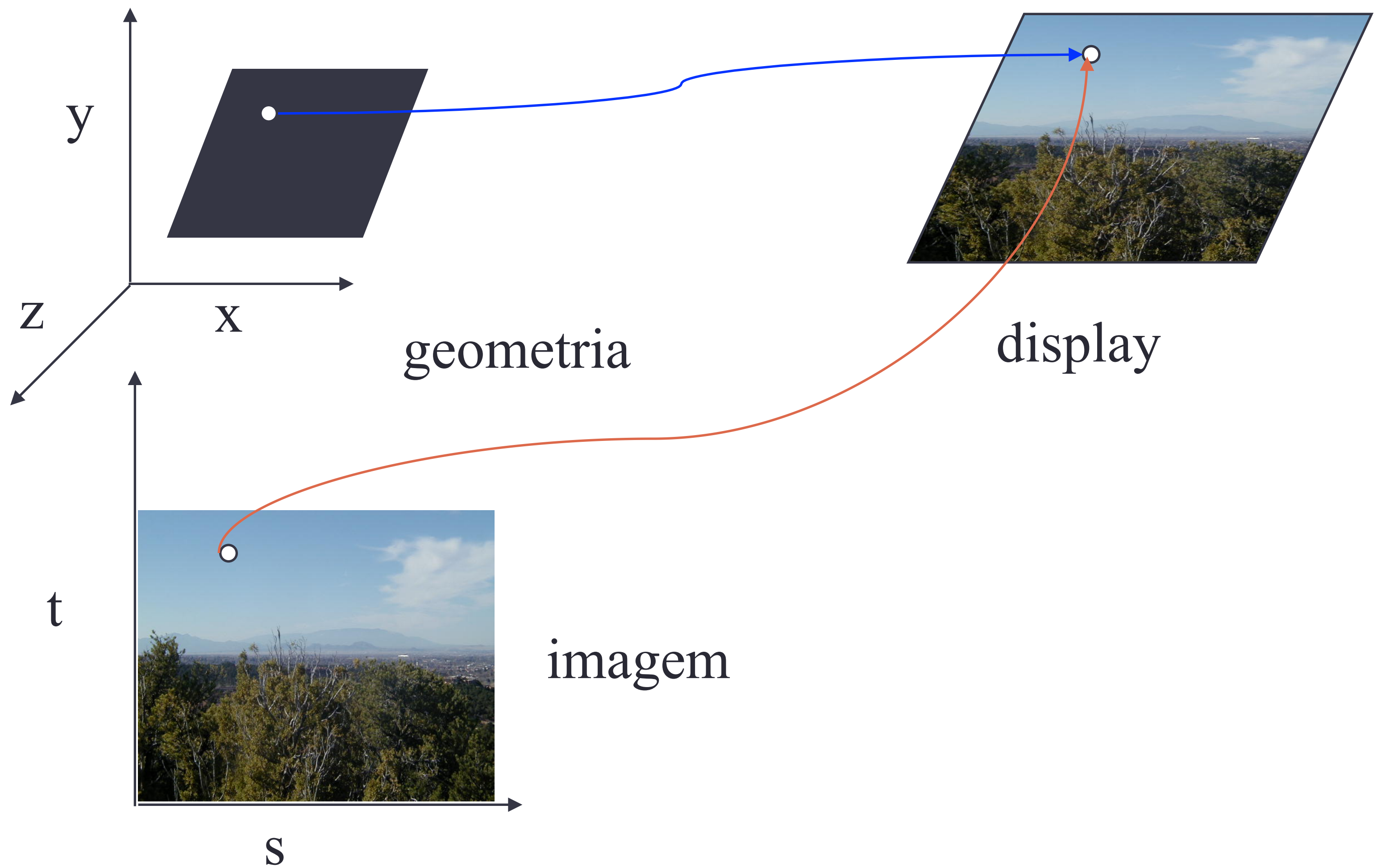
Tipos de textura

- Texturas podem ter componentes RGB ou RGBA ou simplesmente luminosidade (L).
- Podem ter diferentes dimensões:
  - 1D: Arrays
  - 2D: Imagens
  - 3D: Volumes (imagens volumétricas)
- Em versões < 2.0 de OpenGL, os tamanhos de texturas deveriam ser potência de 2.
  - A atual versão do WebGL também.

Mapeamento de textura

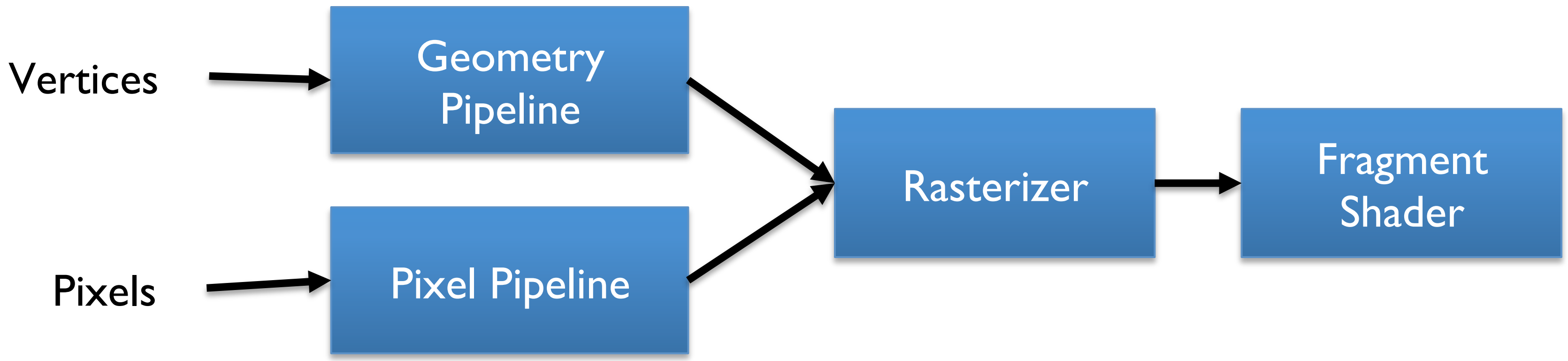


Mapeamento de textura



Mapeamento de textura no pipeline

- Técnicas de mapeamento são implementadas no final do pipeline
  - Eficiente pois poucos polígonos sobrevivem ao processo de recorte



Tarefa de casa

- Leitura livro-texto
  - Shirley and Marschner. Fundamentals of Computer Graphics, CRC Press, 3rd Ed. 2010
  - Capítulo 8