Shadows

![Illustration of shadows cast by two spheres](image-url)
Shadows

for each pixel do
    compute viewing ray
    if ( ray hits an object with t in [0, inf] ) then
        compute n
        evaluate shading model and set pixel to that color
    else
        set pixel color to the background color
for each pixel do
    compute viewing ray
    if ( ray hits an object with t in [0, inf] ) then
        compute n
        evaluate shading model and set pixel to that color
    else
        set pixel color to the background color
for each pixel do
  compute viewing ray
  if ( ray hits an object with t in [0, inf] ) then
    compute n
    // e.g., phong shading
  for each light
    add light’s ambient component
    compute shadow ray
    if ( ! shadow ray hits an object )
      add light’s diffuse and specular components
  else
    set pixel color to the background color
Reflections

- Reflective_Shader subclass of Phong shader
for each pixel do
  compute viewing ray
  if ( ray hits an object with t in [0, inf] ) then
    compute n
    evaluate shading model and set pixel to that color
  else
    set pixel color to the background color
for each pixel do
  compute viewing ray
  if ( ray hits an object with t in [0, inf] ) then
    compute n
    evaluate shading model and set pixel to that color
  else
    set pixel color to the background color
for each pixel do
  compute viewing ray
  pixel color = cast_ray(viewing ray)

**cast_ray:**
  if ( ray hits an object with t in [0, inf] ) then
    compute n
    return color = shade_surface
  else
    return color = to the background color

**shade_surface:**
  color = ...
  compute reflected ray
  return color = color + k * cast_ray(reflected ray)
Distribution Ray Tracing
Anti-aliasing

[Shirley and Marschner]
Soft Shadows

[Shirley and Marschner]
Soft Focus

[Diagram showing light rays and focus plane]
Fuzzy Reflections
Acceleration Structures
Acceleration Structures
Bounding boxes

$t \in [t_{x\min}, t_{x\max}]$

$t \in [t_{y\min}, t_{y\max}]$

$t \in [t_{x\min}, t_{x\max}] \cap [t_{y\min}, t_{y\max}]$
Uniform Spatial Partitioning

[Shirley and Marschner]
Bounding Volume Hierarchy
Graphics Pipeline
Z-buffer Rendering

- Z-buffering is very common approach, also often accelerated with hardware
- OpenGL is based on this approach
Pipelining operations

An arithmetic pipeline that computes \( c + (a \times b) \)

By pipelining the arithmetic operation, the **throughput**, or rate at which data flows through the system, has been **doubled**
If the pipeline had more boxes, the **latency**, or time it takes one datum to pass through the **system**, would be higher
**throughput and latency must be balanced**
3D graphics pipeline

vertes → [Vertex processor] → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels

Geometry: objects – made of primitives – made of vertices
Vertex processing: coordinate transformations and color
Clipping and primitive assembly: output is a set of primitives
Rasterization: output is a set of fragments for each primitive
Fragment processing: update pixels in the frame buffer

the pipeline is best when we are doing the same operations on many data sets
-- good for computer graphics!! where we process larges sets of vertices and pixels in the same manner
1. Geometry: objects – made of primitives – made of vertices
2. Vertex processing: coordinate transformations and color
3. Clipping and primitive assembly: use clipping volume. must be primitive by primitive rather than vertex by vertex. therefore vertices must be assembled into primitives before clipping can take place. output is a set of primitives.
4. Rasterization: primitives are still in terms of vertices -- must be converted to pixels. e.g., for a triangle specified by 3 vertices, the rasterizer must figure out which pixels in the frame buffer fill the triangle. output is a set of fragments for each primitive. a fragment is like a potential pixel. fragments can carry depth information used to figure out if they lie behind other fragments for a given pixel.
5. Fragment processing: update pixels in the frame buffer. some fragments may not be visible. texture mapping and bump mapping. blending.
3D graphics pipeline

- optimized for drawing 3D triangles with shared vertices
- map 3D vertex locations to 2D screen locations
- shade triangles and draw them in back to front order using a z-buffer
- speed depends on # of triangles
- most operations on vertices can be represented using a 4D coordinate space - 3D position + homogeneous coordinate for perspective viewing
- 4x4 matrices and 4-vectors

- use varying level of detail – fewer triangles for distant objects
1. construct shapes from primitives – points, lines, polygons, images, bitmaps, (mathematical descriptions of objects) – specify the model
Primitives and Attributes
Choice of primitives

• Which primitives should an API contain?
  • small set - supported by hardware, or
  • lots of primitives - convenient for user
Choice of primitives

• Which primitives should an API contain?
  ➡ small set - supported by hardware
• lots of primitives - convenient for user

Performance is in 10s millions polygons/sec -- portability, hardware support key
Choice of primitives

• Which primitives should an API contain?
  - small set - supported by hardware
  - lots of primitives - convenient for user

GPUs are optimized for points, lines, and triangles
Choice of primitives

- Which primitives should an API contain?
  - small set - supported by hardware
  - lots of primitives - convenient for user

GPUs are optimized for **points, lines, and triangles**

In other geometric shapes will be built out of these
Two classes of primitives

**Geometric**: points, lines, polygons

**Image**: arrays of pixels

Angel and Shreiner
Point and line segment types

Angel and Shreiner
Polygons

- Multi-sided planar element composed of edges and vertices.
- Vertices (singular vertex) are represented by points.
- Edges connect vertices as line segments.
Valid polygons

- Simple
- Convex
- Flat
Valid polygons

- Simple
- Convex
- Flat
Open GL polygons

- Only triangles are supported (in latest versions)

GL_POINTS  GL_TRIANGLES  GL_TRIANGLE_STRIP  GL_TRIANGLE_FAN
triangulation
as long as triangles are not collinear, they will be simple, flat, and convex -- easy to render
Sample attributes

- Color: `glClearColor(1.0, 1.0, 1.0, 1.0);`
- Point size: `glPointSize(2.0);`
- Line width: `glLineWidth(3.0);`