CSE 167:
Introduction to Computer Graphics
Lecture #12: Visibility Culling

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2020

Announcements

- Sunday, November 15th at 11:59pm:
  - Late deadline for Project 2
- Sunday, November 22nd at 11:59pm:
  - Homework Project 3 due
- Tomorrow is Veterans Day
  - No discussion
  - Homework project 3 introduction in class today

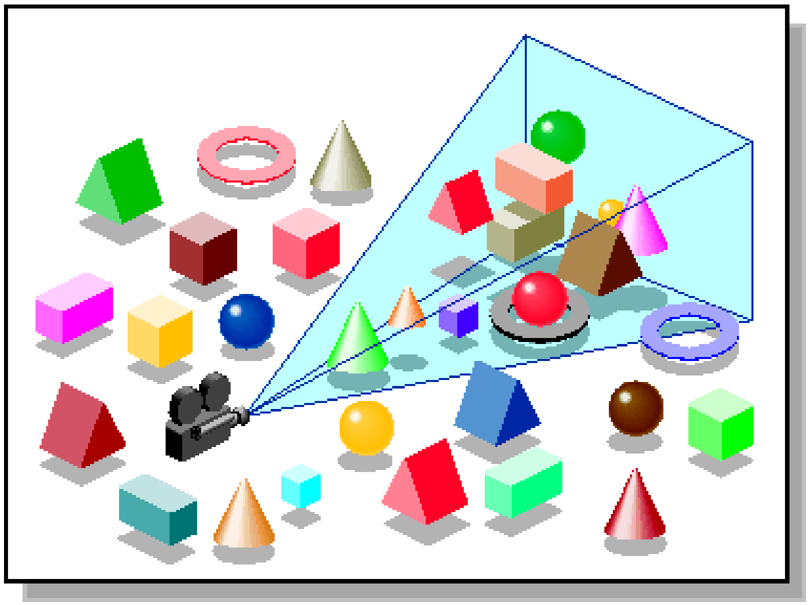
Visibility Culling

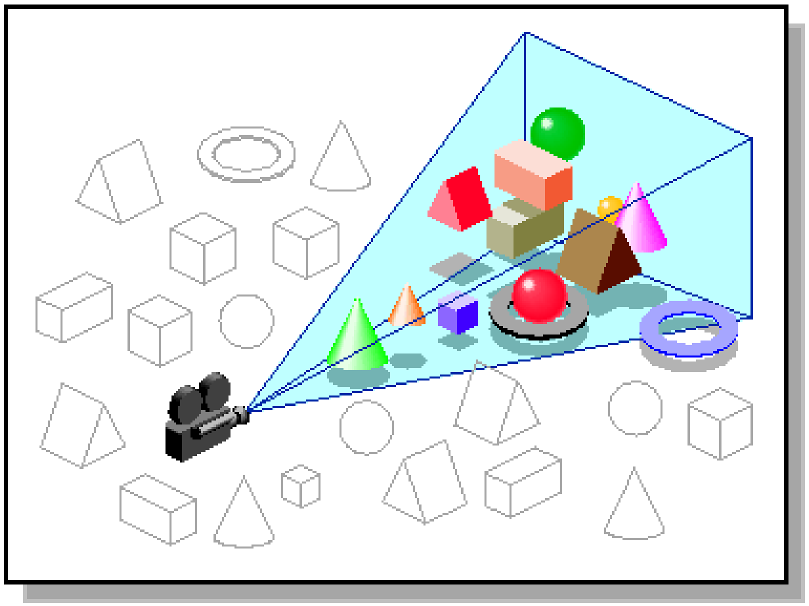
Visibility Culling

- Goal:
  Discard geometry that does not need to be drawn to speed up rendering
- Types of culling:
  - View frustum culling
  - Small object culling
  - Degenerate culling
  - Backface culling
  - Occlusion culling

View Frustum Culling

- Triangles outside of view frustum are off-screen





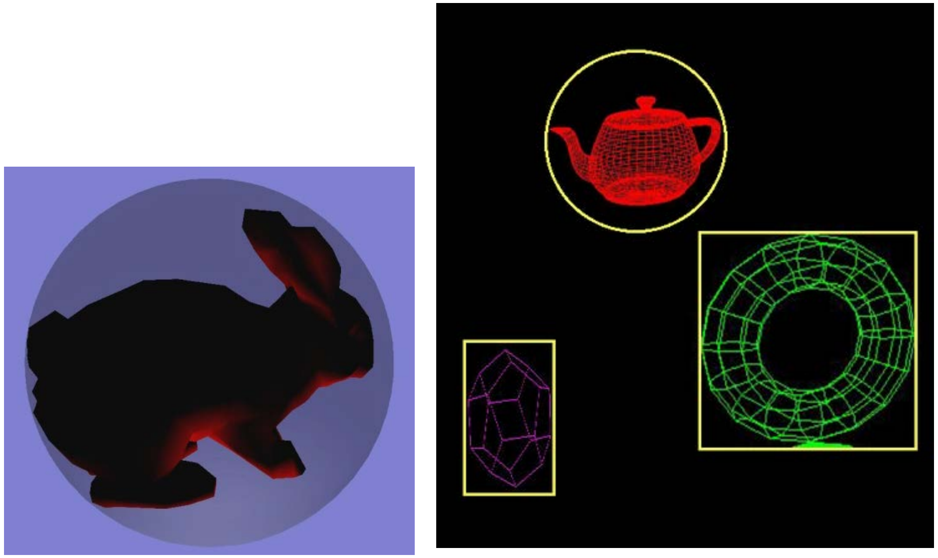
Images: SGI OpenGL Optimizer Programmer's Guide

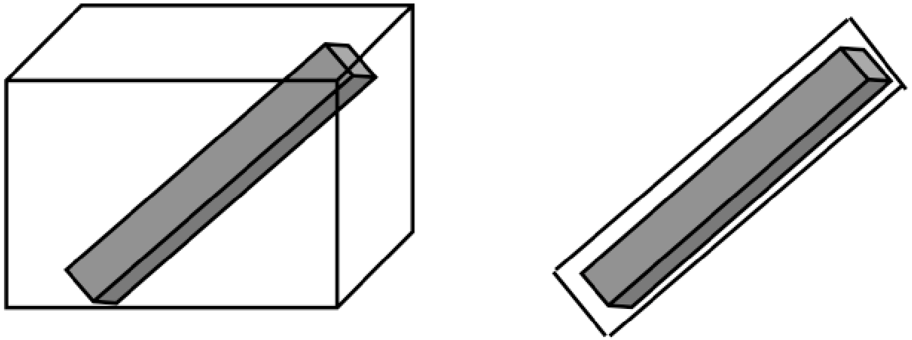
Videos

- Rendering Optimizations - Frustum Culling
  - http://www.youtube.com/watch?v=kvVHp9wMAO8
- View Frustum Culling Demo
  - http://www.youtube.com/watch?v=bJrYTBGpwic
- View Frustum Culling in Action
  - http://giant.gfycat.com/InexperiencedMadKiskadee.webm

Bounding Volumes

- Simple shape that completely encloses an object
- Generally a box or sphere
  - Easier to calculate culling for spheres
  - Easier to calculate tight fits for boxes
- Intersect bounding volume with view frustum instead of each primitive





Bounding Box

- How to cull objects consisting of may polygons?
- Cull bounding box
  - Rectangular box, parallel to object space coordinate planes
  - Box is smallest box containing the entire object

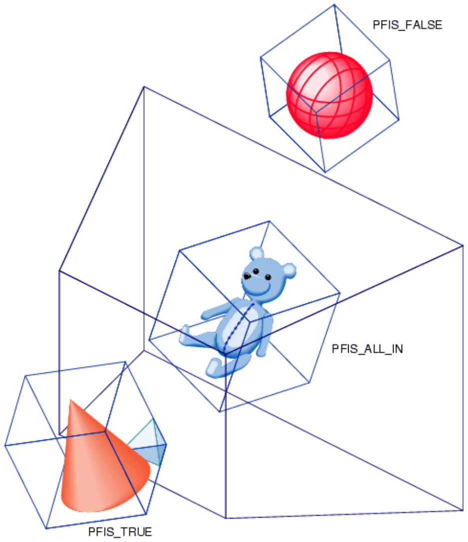
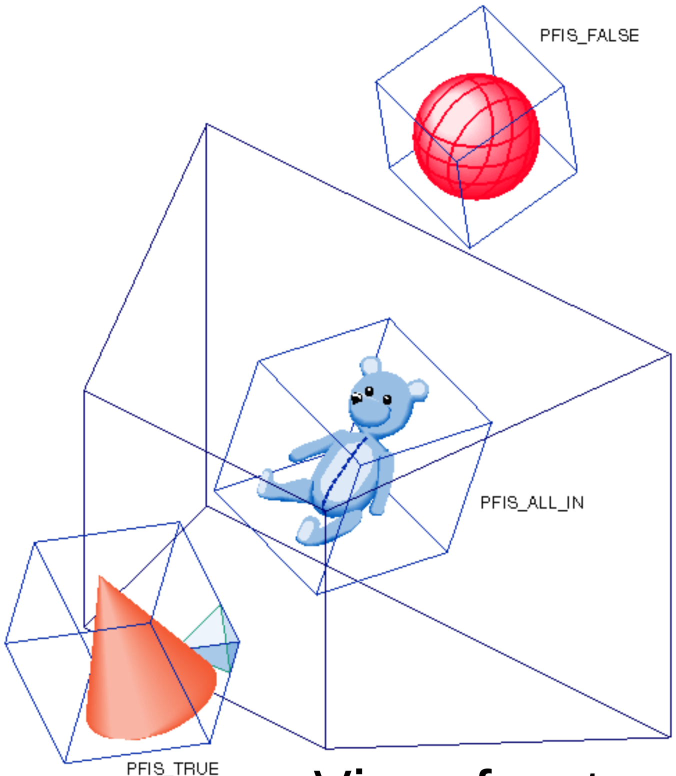


Image: SGI OpenGL Optimizer Programmer's Guide

View Frustum Culling

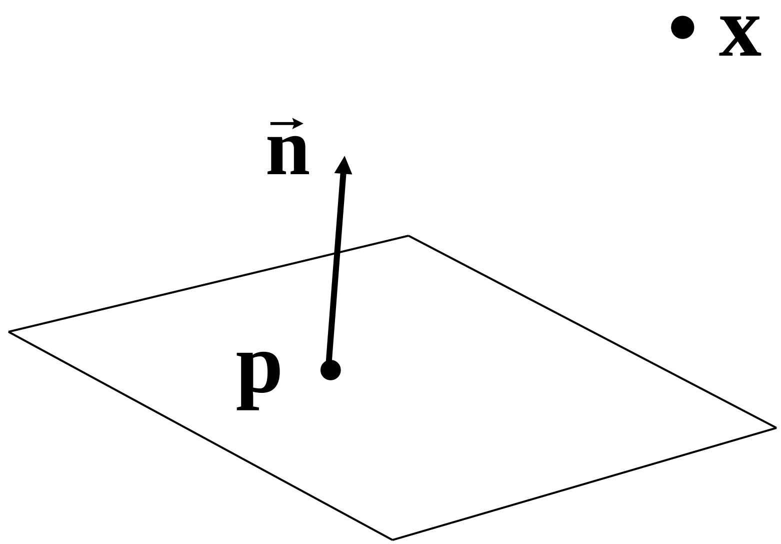
- Frustum defined by 6 planes
- Each plane divides space into "outside", "inside"
- Check each object against each plane
  - Outside, inside, intersecting
- If "outside" of at least one plane
  - Outside the frustum
- If "inside" all planes
  - Inside the frustum
- Else partly inside and partly out



View frustum

Distance to Plane

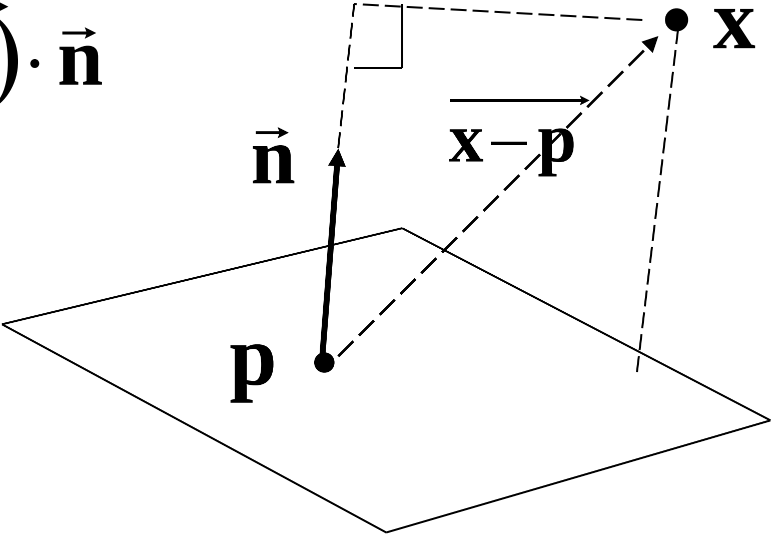
- A plane is described by a point **p** on the plane and a unit normal **n**
- Find the (perpendicular) distance from point **x** to the plane



Distance to Plane

- The distance is the length of the projection of **x-p** onto **n**

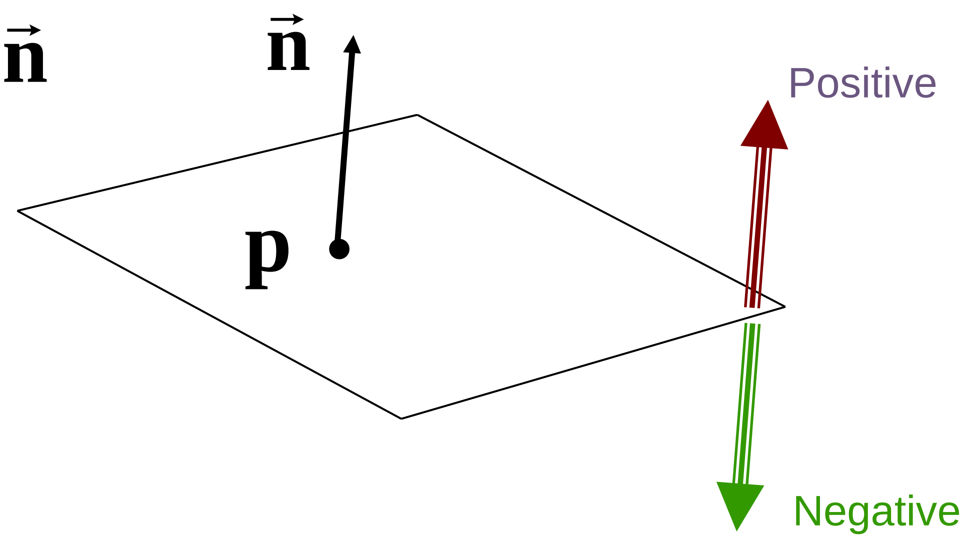
$$dist = \overrightarrow{(\mathbf{x} - \mathbf{p})} \cdot \vec{\mathbf{n}}$$



Distance to Plane

- The distance has a sign
  - positive on the side of the plane the normal points to
  - negative on the opposite side
  - zero exactly on the plane
- Divides 3D space into two infinite half-spaces

$$dist(\mathbf{x}) = \overrightarrow{(\mathbf{x} - \mathbf{p})} \cdot \vec{\mathbf{n}}$$



Distance to Plane

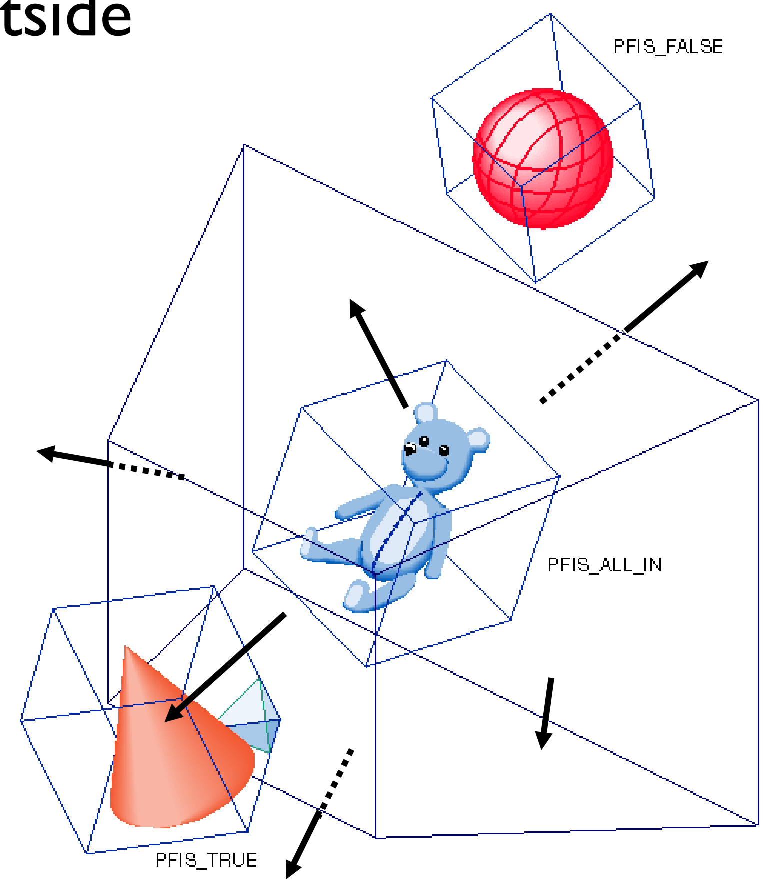
- Simplification

$$dist(\mathbf{x}) = (\mathbf{x} - \mathbf{p}) \cdot \mathbf{n}$$
$$= \mathbf{x} \cdot \mathbf{n} - \mathbf{p} \cdot \mathbf{n}$$
$$dist(\mathbf{x}) = \mathbf{x} \cdot \mathbf{n} - d, \quad d = \mathbf{pn}$$

- *d* is independent of **x**
- *d* is distance from the origin to the plane
- We can represent a plane with just *d* and **n**

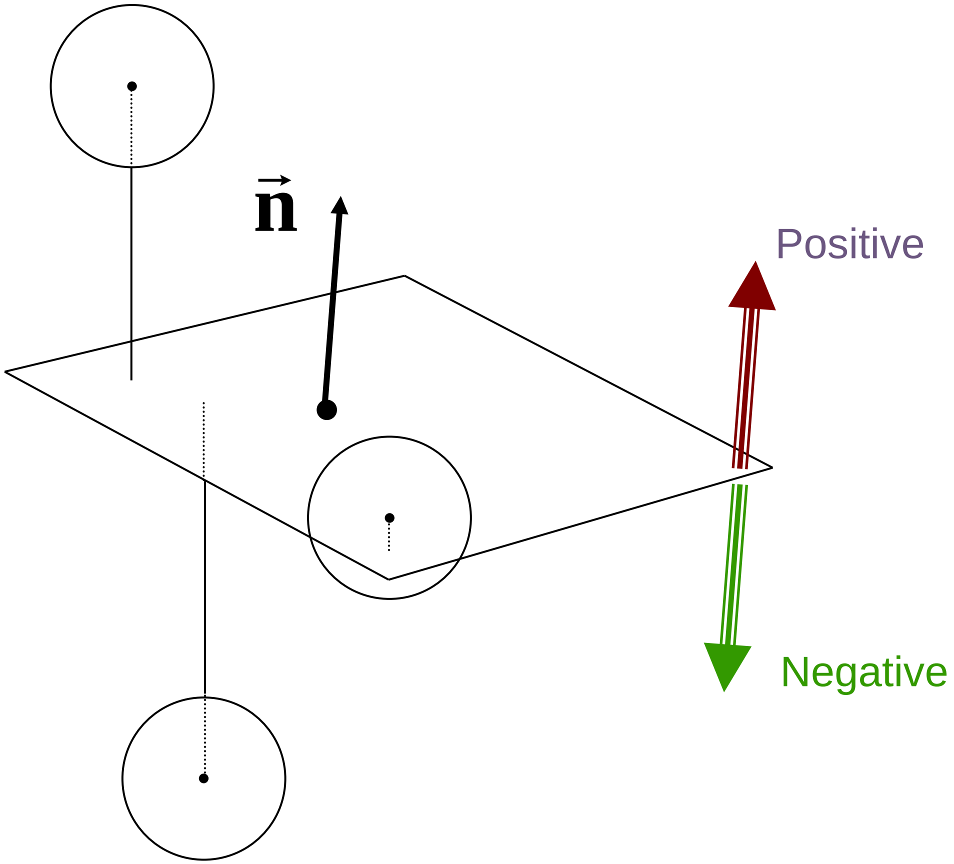
Frustum With Signed Planes

- Normal of each plane points outside
  - "outside" means positive distance
  - "inside" means negative distance



Test Sphere and Plane

- For sphere with radius r and origin $\mathbf{x}$, test the distance to the origin, and see if it is beyond the radius
- Three cases:
  - $dist(\mathbf{x}) > r$
    - completely above
  - $dist(\mathbf{x}) < -r$
    - completely below
  - $-r < dist(\mathbf{x}) < r$
    - intersects

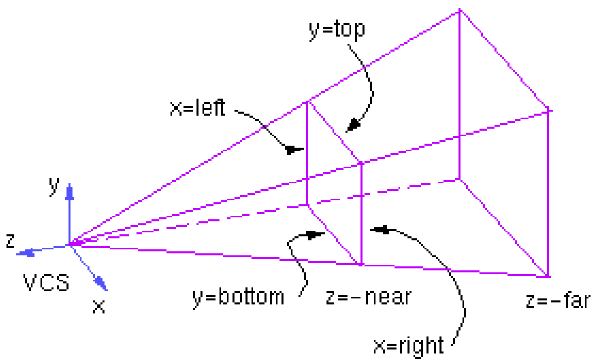


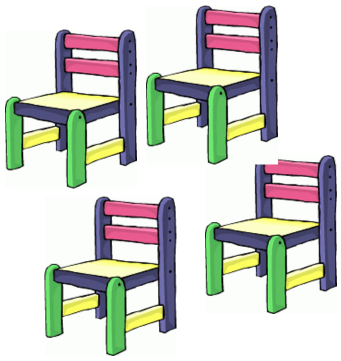
Culling Summary

- Transform view frustum plane equations in camera space.
- Pre-compute the normal $\mathbf{n}$ and value d for each of the six planes.
- Given a sphere with center $\mathbf{x}$ and radius r in camera space.
- For each plane:
  - if $dist(\mathbf{x}) > r$: sphere is outside! (no need to continue loop)
  - add 1 to count if $dist(\mathbf{x}) < -r$
- If we made it through the loop, check the count:
  - if the count is 6, the sphere is completely inside
  - otherwise the sphere intersects the frustum
  - *(can use a flag instead of a count)*

Culling Groups of Objects

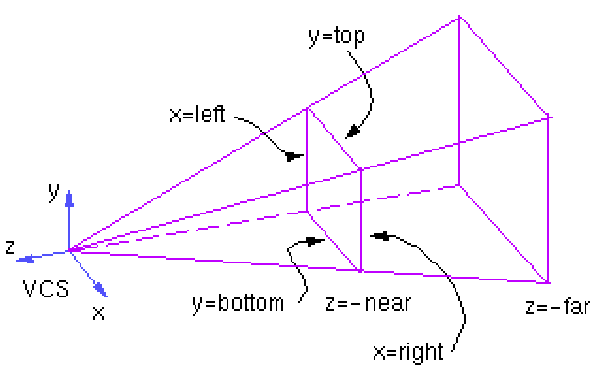
- Want to be able to cull the whole group quickly
- But if the group is partly in and partly out, want to be able to cull individual objects

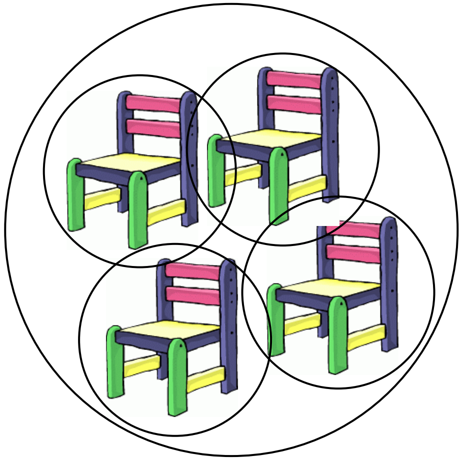




Hierarchical Bounding Volumes

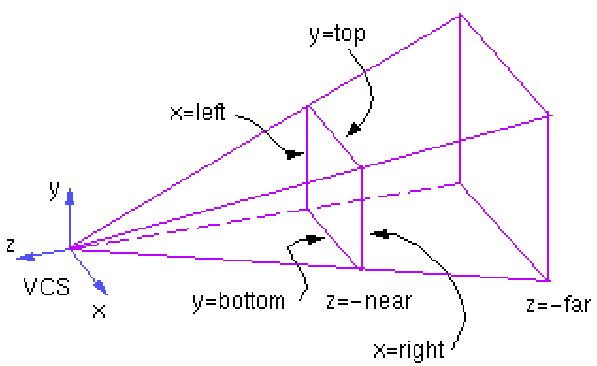
- Given hierarchy of objects
- Bounding volume of each node encloses the bounding volumes of all its children
- Start by testing the outermost bounding volume
  - If it is entirely outside, don't draw the group at all
  - If it is entirely inside, draw the whole group

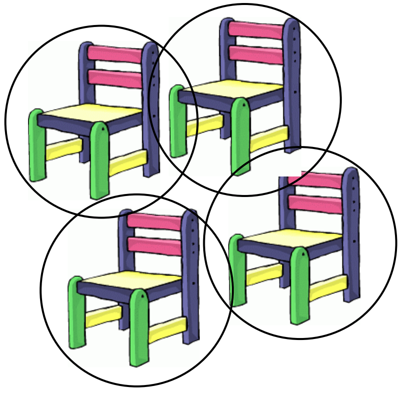




Hierarchical Culling

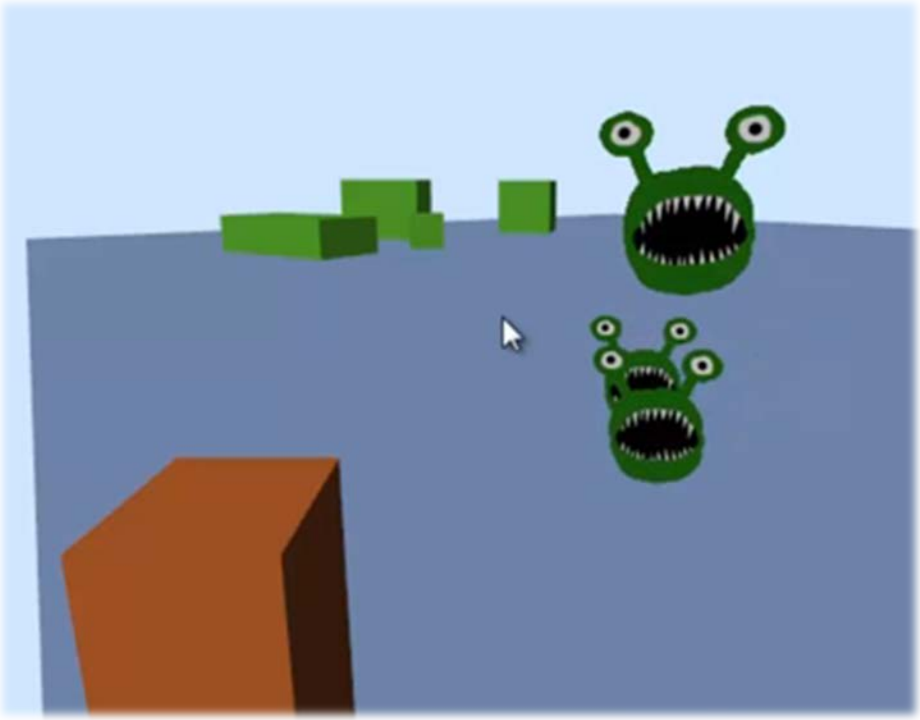
- If the bounding volume is partly inside and partly outside
  - Test each child's bounding volume individually
  - If the child is in, draw it; if it's out cull it; if it's partly in and partly out, recurse.
  - If recursion reaches a leaf node, draw it normally





Video

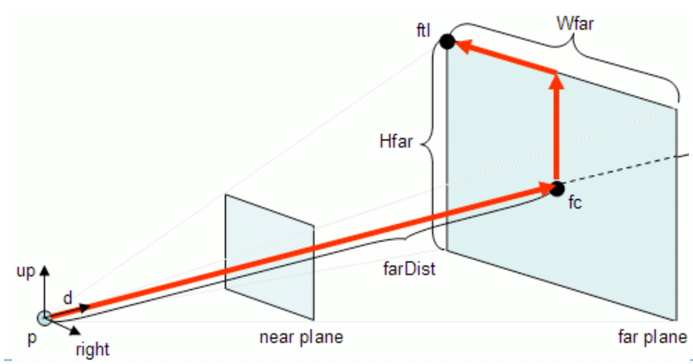
- Math for Game Developers - Frustum Culling
  - http://www.youtube.com/watch?v=4p-E_31XOPM



Find the frustum planes

- p – the camera position
- d – a vector with the direction of the camera's view ray. In here it is assumed that this vector has been normalized
- Wnear – the "width" of the near plane
- nearDist – the distance from the camera to the near plane
- farDist – the distance from the camera to the far plane
- up – the up vector obtained by normalizing (ux, uy, uz) from the last parameters of gluLookAt
- right – the right vector obtained by cross product between up and d.

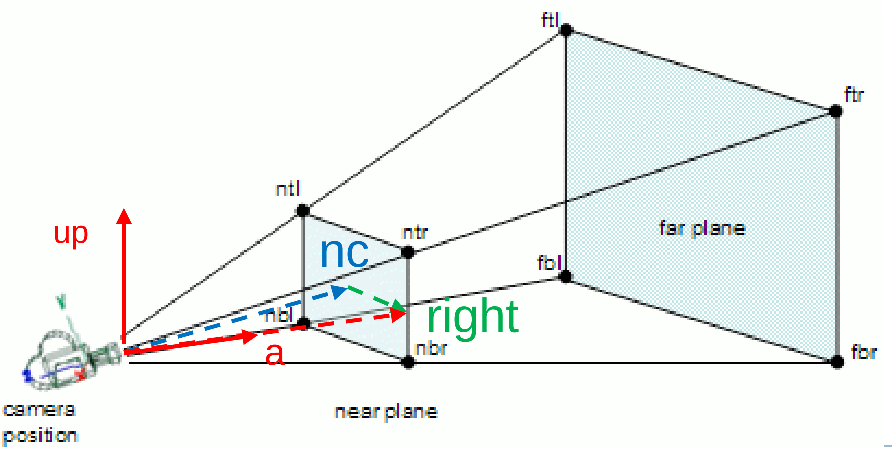
```
nc = p + d * nearDist
fc = p + d * farDist
```



Find the frustum planes

- near plane: d as normal, nc as a point on the plane.
- far plane: –d as normal, fc as a point on the plane.
- right plane: p as a point on the plane. normal can be found in this tutorial, the pseudocode is copied here.

```
nc = p + d * nearDist
a = (nc + right * Wnear / 2) - p
a.normalize()
normalRight = up × a
```



Visibility Culling

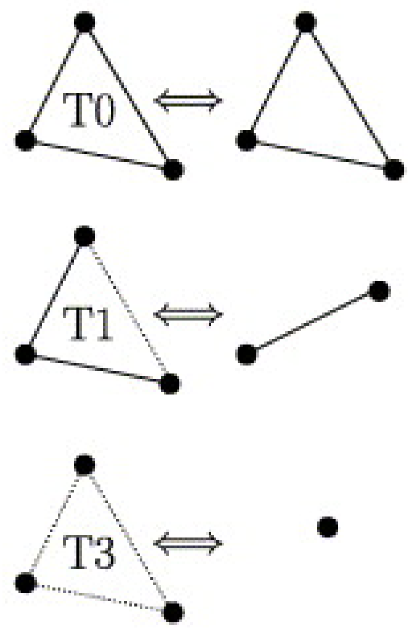
- Goal:
  Discard geometry that does not need to be drawn to speed up rendering
- Types of culling:
  - View frustum culling
  - Small object culling
  - Degenerate culling
  - Backface culling
  - Occlusion culling

Small Object Culling

- Object projects to less than a specified size
  - Cull objects whose screen-space bounding box is less than a threshold number of pixels

Degenerate Culling

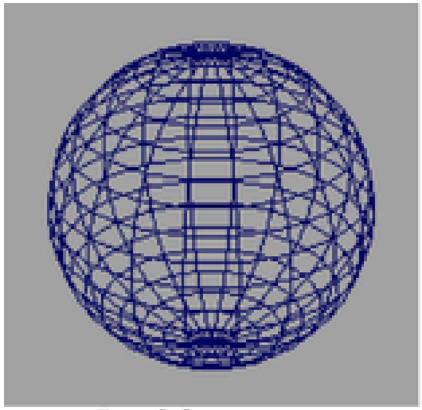
- Degenerate triangle has no area
  - Normal **n**=0
  - All vertices in a straight line
  - All vertices in the same place

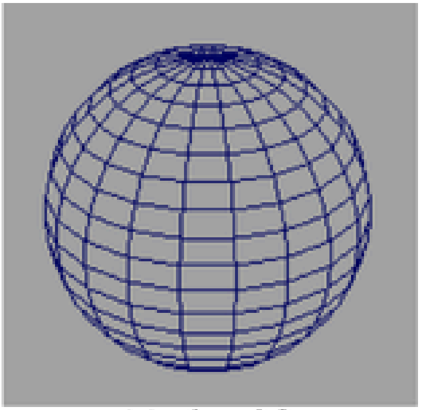


Source: Computer Methods in Applied Mechanics and Engineering, Volume 194, Issues 48–49

Backface Culling

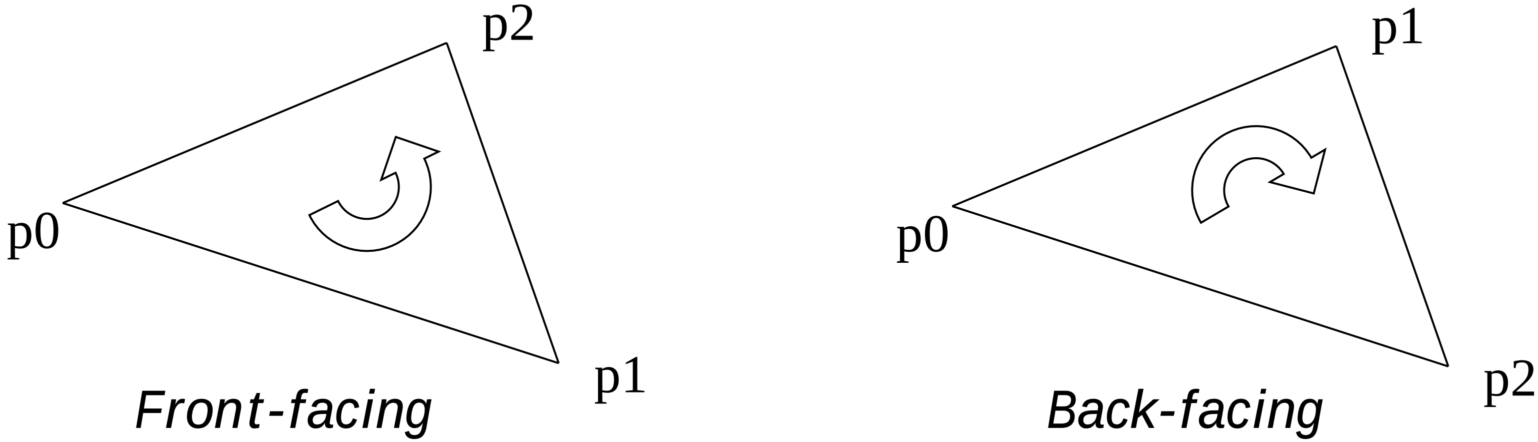
- Consider triangles as "one-sided", i.e., only visible from the "front"
- Closed objects
  - If the "back" of the triangle is facing away from the camera, it is not visible
  - Gain efficiency by not drawing it (culling)
  - Roughly 50% of triangles in a scene are back facing


Backfaces


No backfaces

Backface Culling

- Convention:
  Triangle is front facing if vertices are ordered counterclockwise



Backface Culling

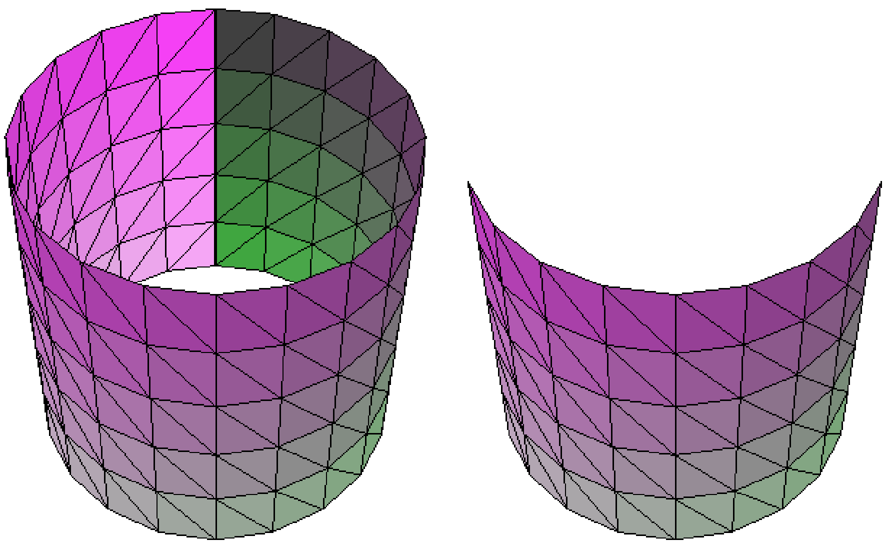
- Compute triangle normal after projection (homogeneous division)

$$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$$

- Third component of $\mathbf{n}$ negative: front-facing, otherwise back-facing
  - Remember: projection matrix is such that homogeneous division flips sign of third component

OpenGL

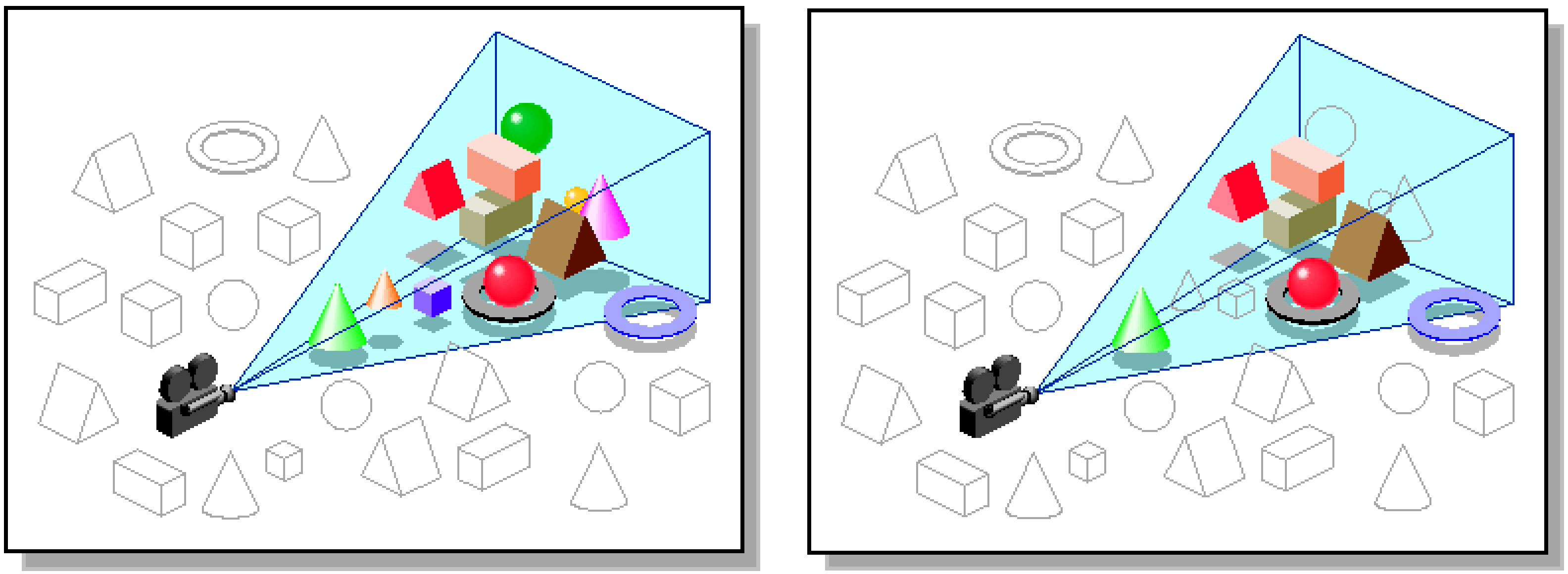
- OpenGL allows one- or two-sided triangles
  - One-sided triangles:
    glEnable(GL_CULL_FACE); glCullFace(GL_BACK)
  - Two-sided triangles (no backface culling):
    glDisable(GL_CULL_FACE)



glDisable(GL_CULL_FACE);   glEnable(GL_CULL_FACE);

Occlusion Culling

- Geometry hidden behind occluder cannot be seen
  - Many complex algorithms exist to identify occluded geometry



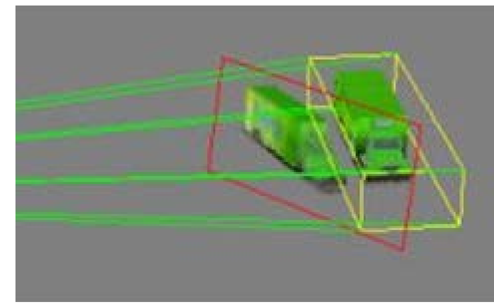
Images: SGI OpenGL Optimizer Programmer's Guide

Video

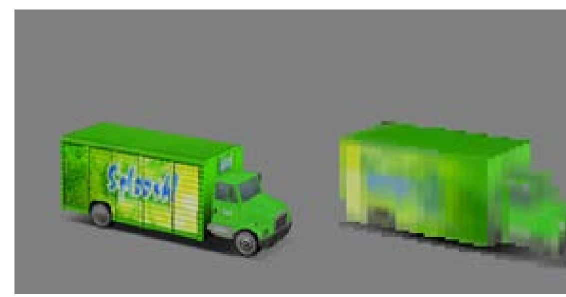
- Umbra 3 Occlusion Culling explained
  - http://www.youtube.com/watch?v=5h4QgDBwQhc

Level-of-Detail Techniques

- Don't draw objects smaller than a threshold
  - Small feature culling
  - Popping artifacts
- Replace 3D objects by 2D impostors
  - Textured planes representing the objects



Impostor generation



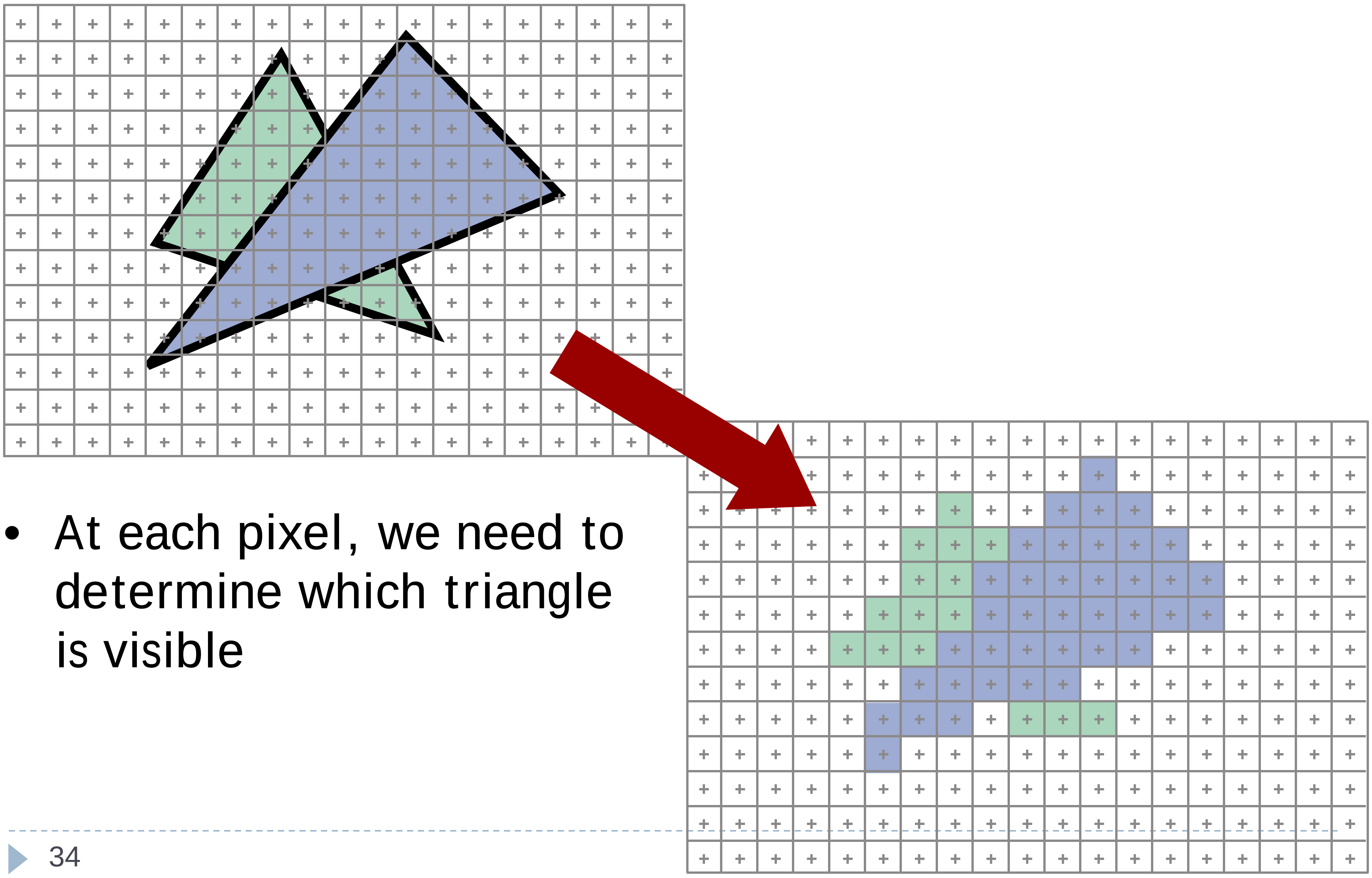
Original vs. impostor

- Adapt triangle count to projected size

Size dependent mesh reduction
(Data: Stanford Armadillo)

Occlusion Detection

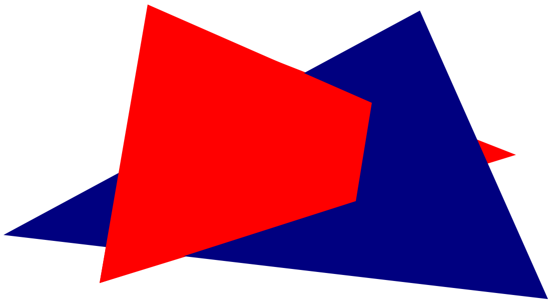
Occlusion



- At each pixel, we need to determine which triangle is visible

Painter's Algorithm

- Paint from back to front
- Need to sort geometry according to depth
- Every new pixel always paints over previous pixel in frame buffer
- May need to split triangles if they intersect



- Intuitive, but slow algorithm
- Still used today to render translucent geometry

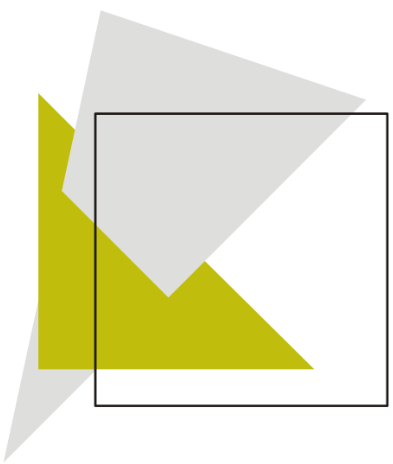
Z-Buffering

- Z-buffer stores depth (z-) value for each pixel
- Z-buffer is dedicated memory in GPU
- Algorithm:
  - Create z-buffer with as many entries as pixels in render window
  - Initialize z-buffer with farthest z value
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller

```
setpixel(int x, int y, color c, float z)
if(z < zbuffer(x,y)) then
{ zbuffer(x,y) = z; color(x,y) = c }
```

- Depth test is performed by GPU → very fast

Z-Buffer Example



∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

+

5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

=

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

+

7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

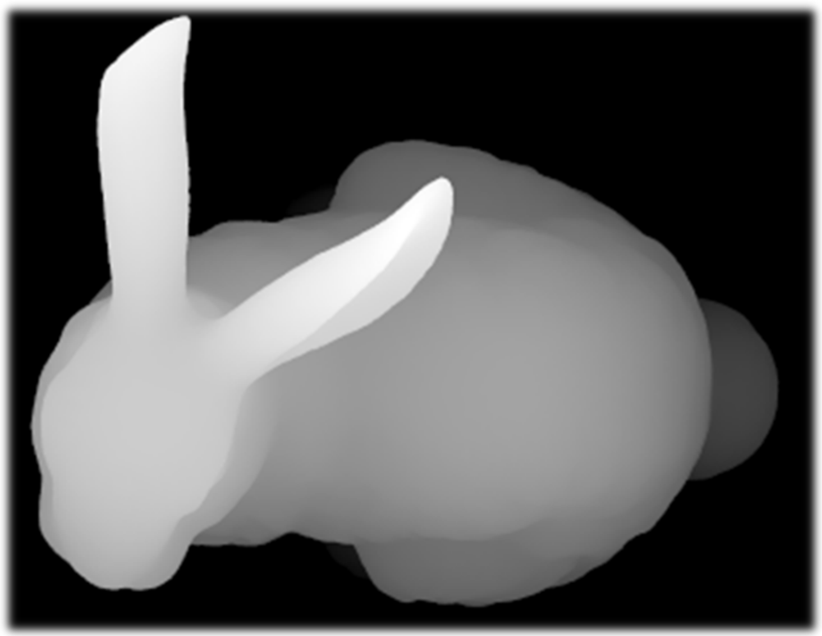
=

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Displaying the Z-Buffer

- Interpret z-buffer values as luminance values
- gl_FragCoord in fragment shader contains depth value
- Output this depth value as a color:

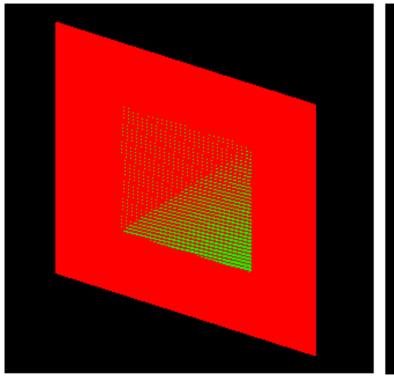
```
void main() { FragColor = vec4(vec3(gl_FragCoord.z), 1.0); }
```



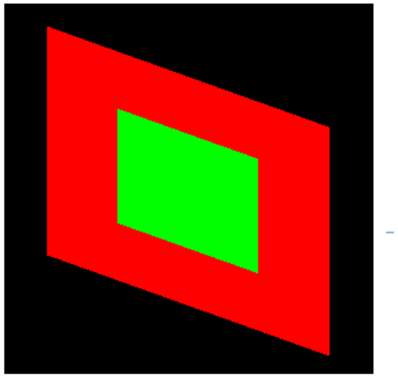
Z-Buffering in OpenGL

- In OpenGL applications:
  - Ask for a depth buffer when you create your GLFW window.
    - glfwOpenWindow(512, 512, 8, 8, 8, 0, **16**, 0, GLFW_WINDOW)
  - Place a call to glEnable(GL_DEPTH_TEST) in your program's initialization routine.
  - Set *zNear* and *zFar* clipping planes (glm::perspective(fovy, aspect, zNear, zFar)) to optimize depth buffer precision: near plane as far away as possible, far plane as close as possible without cutting into scene
  - Add GL_DEPTH_BUFFER_BIT parameter to glClear:
    - glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
- Z-buffer is non-linear: uses smaller depth bins in foreground for greater depth resolution near viewer

Z-Buffer Fighting



Z-buffer fighting

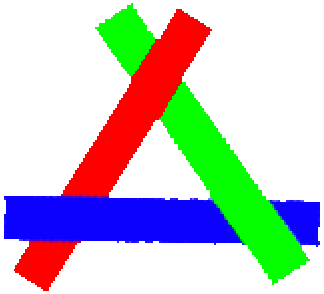


Desired result

- Problem: polygons close together don't get rendered correctly. Errors change with camera perspective → flicker
- Cause: differently colored fragments from different polygons being rasterized to same pixel and depth → not clear which is in front
- Solutions:
  - Move surfaces farther apart, so that fragments rasterize into different depth bins
  - Bring near and far planes closer together
  - Use a higher precision depth buffer. Note that OpenGL often defaults to 16 bit even if your graphics card supports 24 bit or 32 bit depth buffers

Translucent Geometry

- Need to depth sort translucent geometry and render with Painter's Algorithm (back to front)
- Problem: incorrect blending with cyclically overlapping geometry



- Solutions:
  - Back to front rendering of translucent geometry (Painter's Algorithm), after rendering opaque geometry
  - Theoretically: need to store multiple depth and color values per pixel (not practical in real-time graphics)