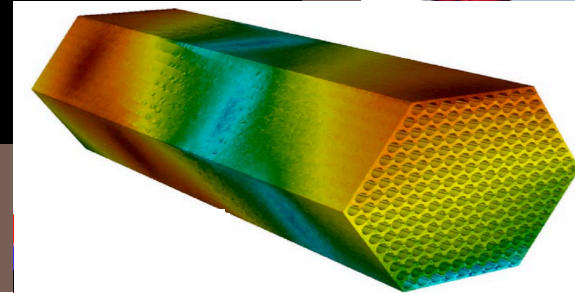
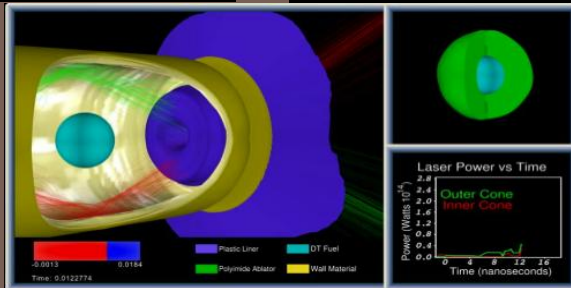
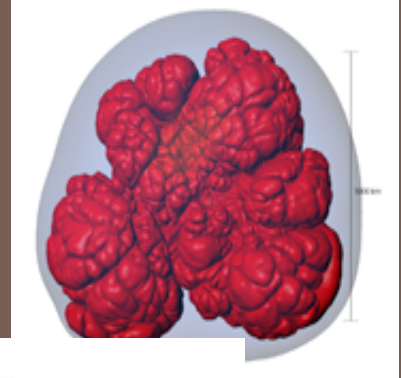
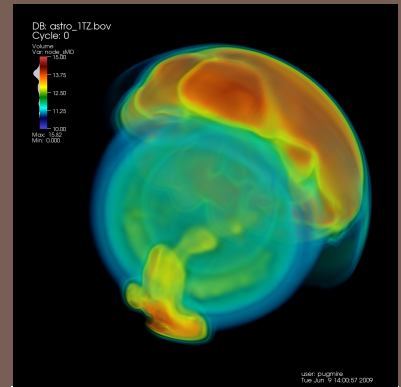
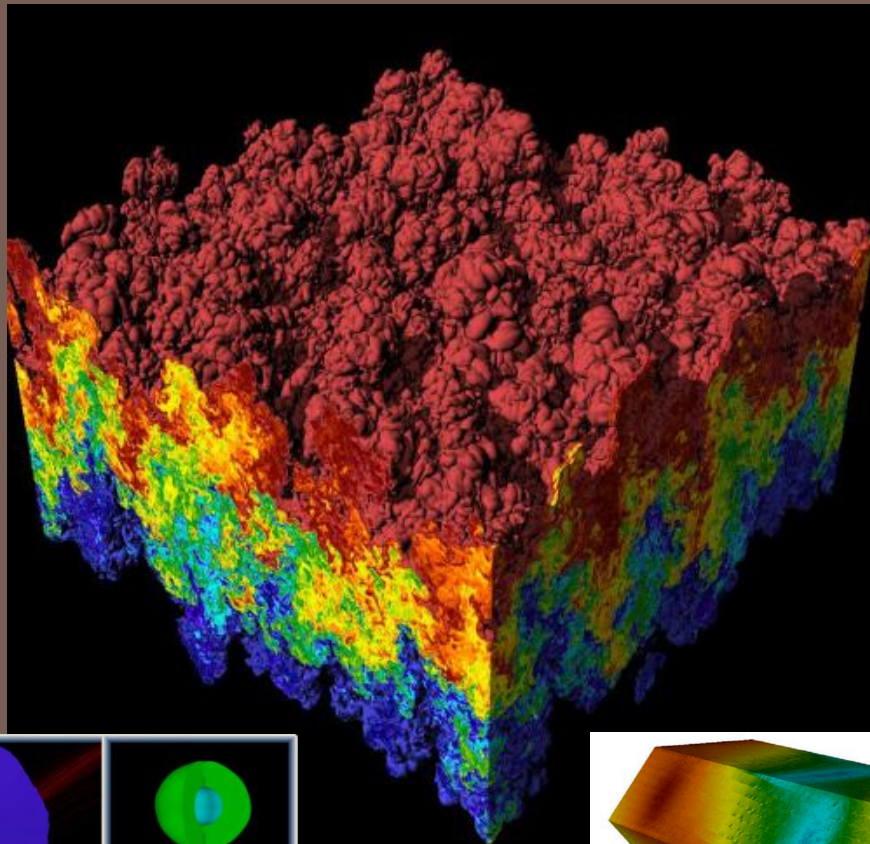
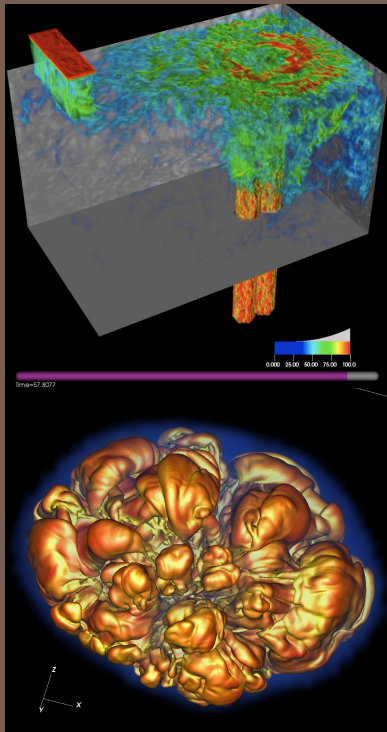


CIS 441/541: Introduction to Computer Graphics

Lecture 14: tests, interactors, raytracing, collision detection, animation



March 7th, 2019

Hank Childs, University of Oregon

Talk about the test



Project G



- ❑ Blender
- ❑ WebGL #1
- ❑ WebGL #2
- ❑ CUDA
- ❑ Bezier
- ❑ Computer Vision
- ❑ Shaders

Schedule



- Upcoming:
 - Project G Mass OH?
 - Live code project 1?

Late Passes



- Will bring forms to final

Hank OH



- Primary purpose of Hank's OH is now to help with self-defined projects
- Also can help with 1A-1F,2A,2B
- Continues as Thurs/Fri 1130-1230



Transparent Geometry



The University of New Mexico

Compositing and Blending

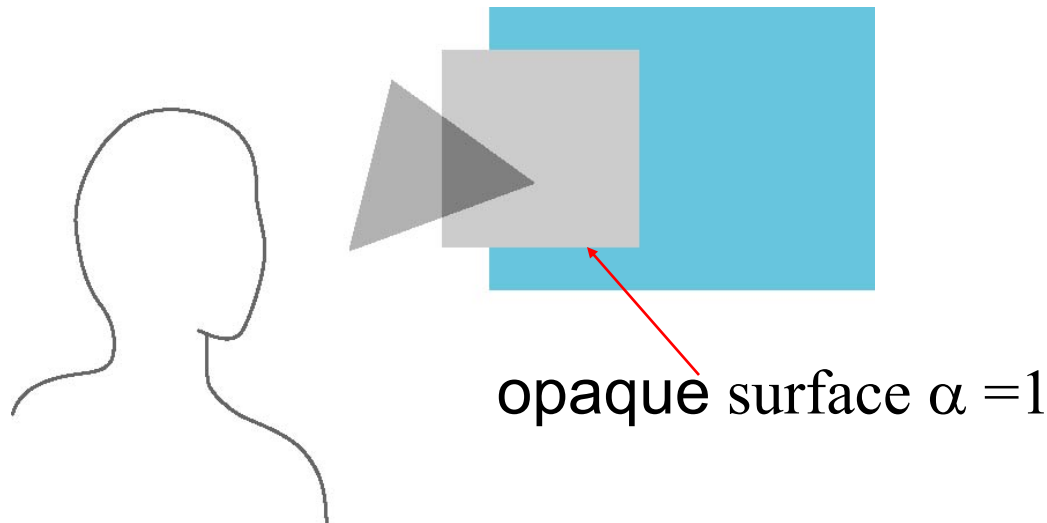
Ed Angel

Professor of Computer Science,
Electrical and Computer
Engineering, and Media Arts
University of New Mexico



Opacity and Transparency

- Opaque surfaces permit no light to pass through
 - Transparent surfaces permit all light to pass
 - Translucent surfaces pass some light
- translucency = $1 - \text{opacity } (\alpha)$





Transparency

- If you have an opaque red square in front of a blue square, what color would you see?
 - Red
- If you have a 50% transparent red square in front of a blue square, what color would you see?
 - Purple
- If you have a 100% transparent red square in front of a blue square, what color would you see?
 - Blue



(One) Formula For Transparency

- Front = (Fr,Fg,Fb,Fa)
 - a = alpha, transparency factor
 - Sometimes percent
 - Typically 0-255, with 255 = 100%, 0 = 0%
- Back = (Br,Bg,Bb,Ba)
- Equation = $(Fa * Fr + (1 - Fa) * Br,$
 $Fa * Fg + (1 - Fa) * Bg,$
 $Fa * Fb + (1 - Fa) * Bb,$
 $Fa + (1 - Fa) * Ba)$



Transparency

- If you have an 25% transparent red square (255,0,0) in front of a blue square (0,0,255), what color would you see (in RGB)?
 - (192,0,64)
- If you have an 25% transparent blue square (0,0,255) in front of a red square (255,0,0), what color would you see (in RGB)?
 - (64,0,192)



Implementation

- Per pixel storage:
 - RGB: 3 bytes
 - Alpha: 1 byte
 - Z: 4 bytes
- Alpha used to control blending of current color and new colors

Vocab term reminder: fragment

- Fragment is the contribution of a triangle to a single pixel

Scanline algorithm

- Determine rows of pixels triangles can possibly intersect
 - Call them rowMin to rowMax
 - rowMin: ceiling of smallest Y value
 - rowMax: floor of biggest Y value
- For r in [rowMin → rowMax] ; do
 - Find end points of r intersected with triangle
 - Call them leftEnd and rightEnd
 - For c in [ceiling(leftEnd) → floor(rightEnd)] ; do
 - ImageColor(r, c) ← triangle color



Examples

- Imagine pixel (i, j) has:
 - RGB = 255/255/255
 - Alpha=255
 - Depth = -0.5
- And we contribute fragment:
 - RGB=0/0/0
 - Alpha=128
 - Depth = -0.25
- What do we get?
- Answer: 128/128/128, Z = -0.25
- What's the alpha?



Examples

- Imagine pixel (i, j) has:
 - RGB = 255/255/255
 - Alpha=128
 - Depth = -0.25
- And we contribute fragment:
 - RGB=0/0/0
 - Alpha=255
 - Depth = -0.5
- What do we get?
- Answer: (probably) 128/128/128, Z = -0.25
- What's the alpha?



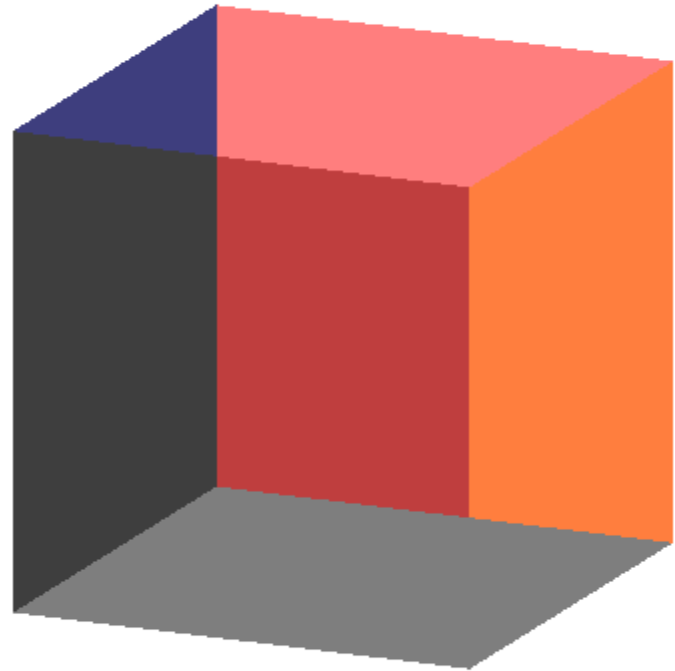
System doesn't work well for transparency

- Contribute fragments in this order:
 - $Z=-0.1$
 - $Z=-0.9$
 - $Z=-0.5$
 - $Z=-0.4$
 - $Z=-0.6$
- Model is too simple. Not enough info to resolve!



Order Dependency

- Is this image correct?
 - Probably not
 - Polygons are rendered in the order they pass down the pipeline
 - Blending functions are order dependent

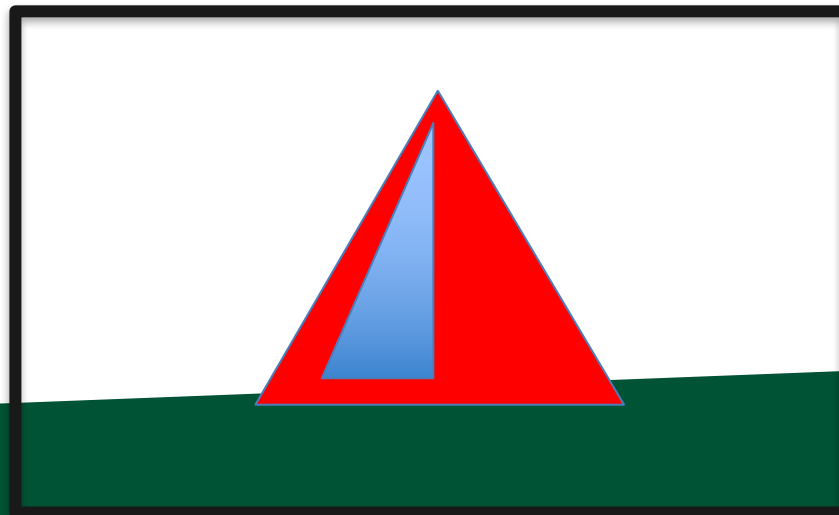
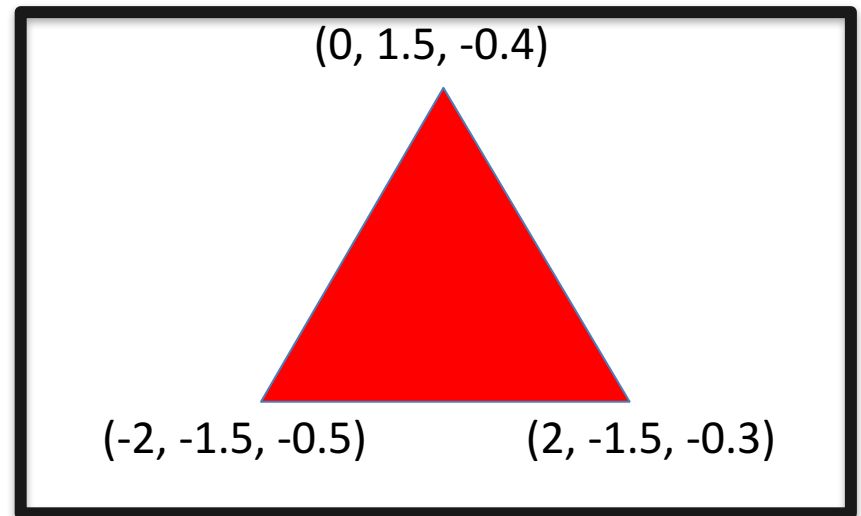
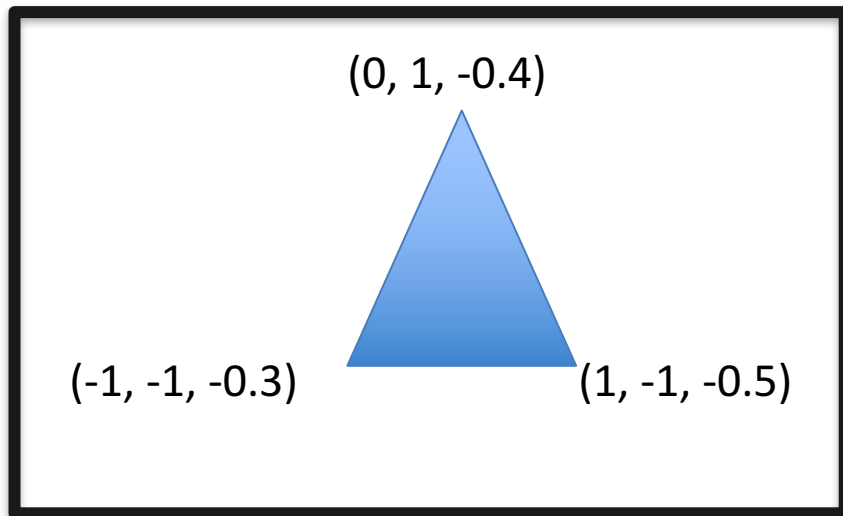




How do you sort?

- 1) Calculate depth of each triangle center.
- 2) Sort based on depth
 - Not perfect, but good
- In practice: sort along X, Y, and Z and use “dominant axis” and only do “perfect sort” when rotation stops

But there is a problem...



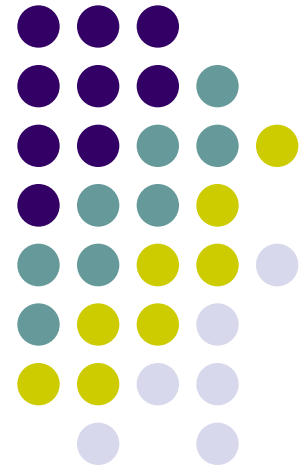


Depth Peeling

- a multi-pass technique that renders transparent polygonal geometry without sorting
- Pass #1:
 - render as opaque, but note opacity of pixels placed on top
 - treat this as “top layer”
 - save Z-buffer and treat this as “max”
- Pass #2:
 - render as opaque, but ignore fragments beyond “max”
- repeat, repeat...

Introduction to Ray Tracing

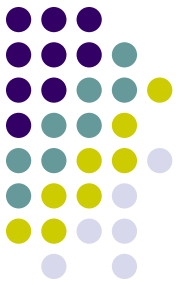
Dr. Xiaoyu Zhang
Cal State U., San Marcos



Classifying Rendering Algorithms



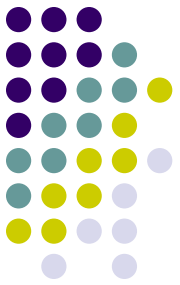
- One way to classify rendering algorithms is according to the type of light interactions they capture
- For example: The OpenGL lighting model captures:
 - Direct light to surface to eye light transport
 - Diffuse and rough specular surface reflectance
 - It actually doesn't do light to surface transport correctly, because it doesn't do shadows
- We would like a way of classifying interactions: *light paths*



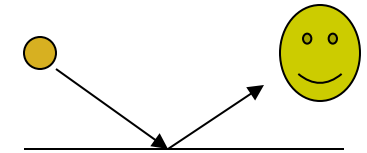
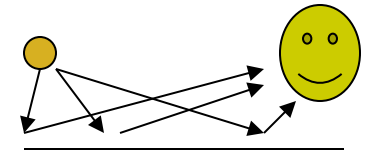
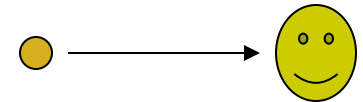
Classifying Light Paths

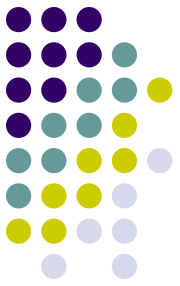
- Classify light paths according to where they come from, where they go to, and what they do along the way
- Assume only two types of surface interactions:
 - Pure diffuse, D
 - Pure specular, S
- Assume all paths of interest:
 - Start at a light source, L
 - End at the eye, E
- Use regular expressions on the letters D, S, L and E to describe light paths
 - Valid paths are $L(D|S)^*E$

Simple Light Path Examples

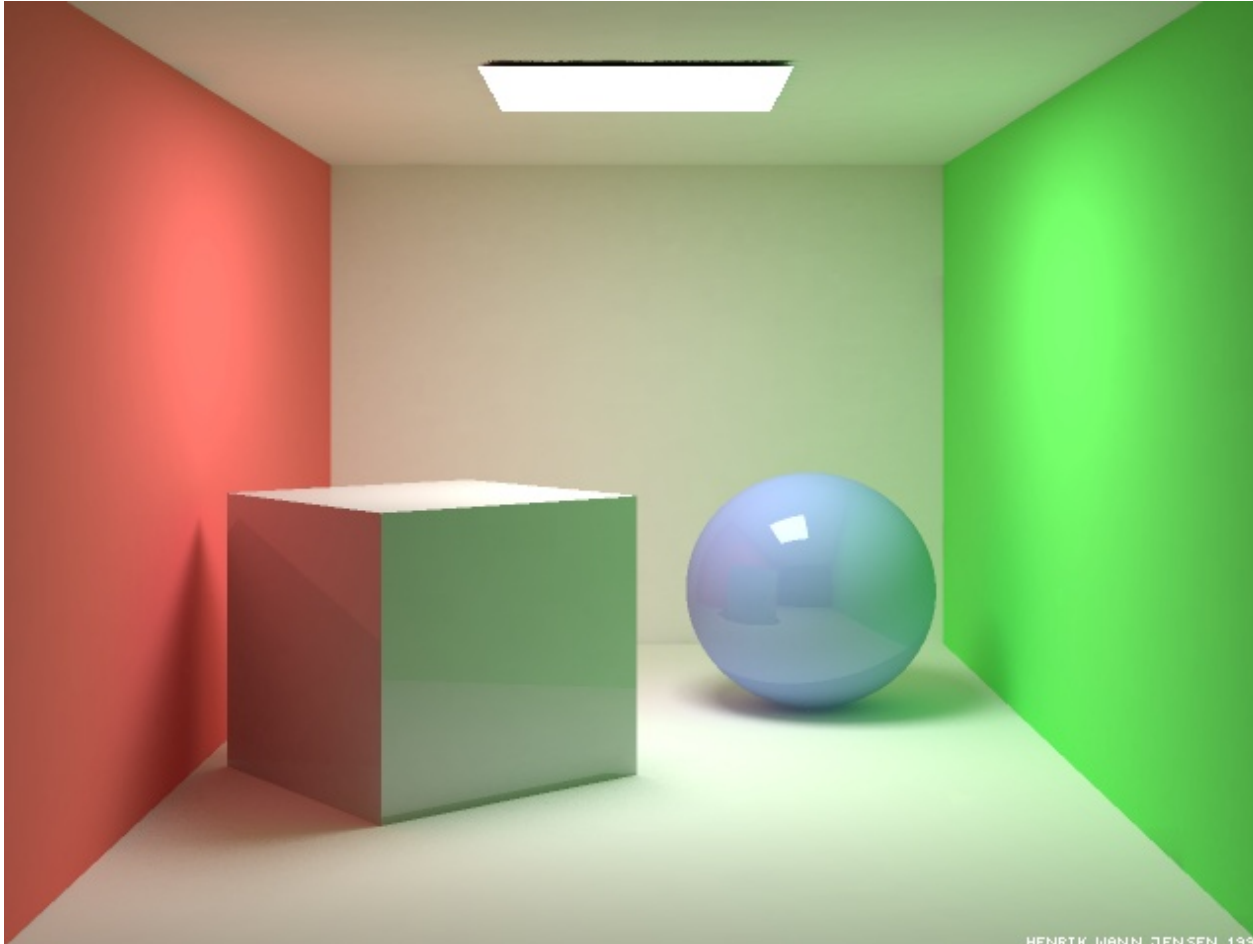


- LE
 - The light goes straight from the source to the viewer
- LDE
 - The light goes from the light to a diffuse surface that the viewer can see
- LSE
 - The light is reflected off a mirror into the viewer's eyes
- L(S|D)E
 - The light is reflected off either a diffuse surface or a specular surface toward the viewer
- Which do OpenGL (approximately) support?





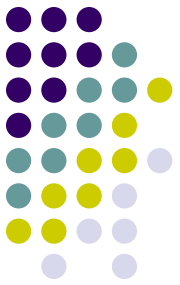
More Complex Light Paths



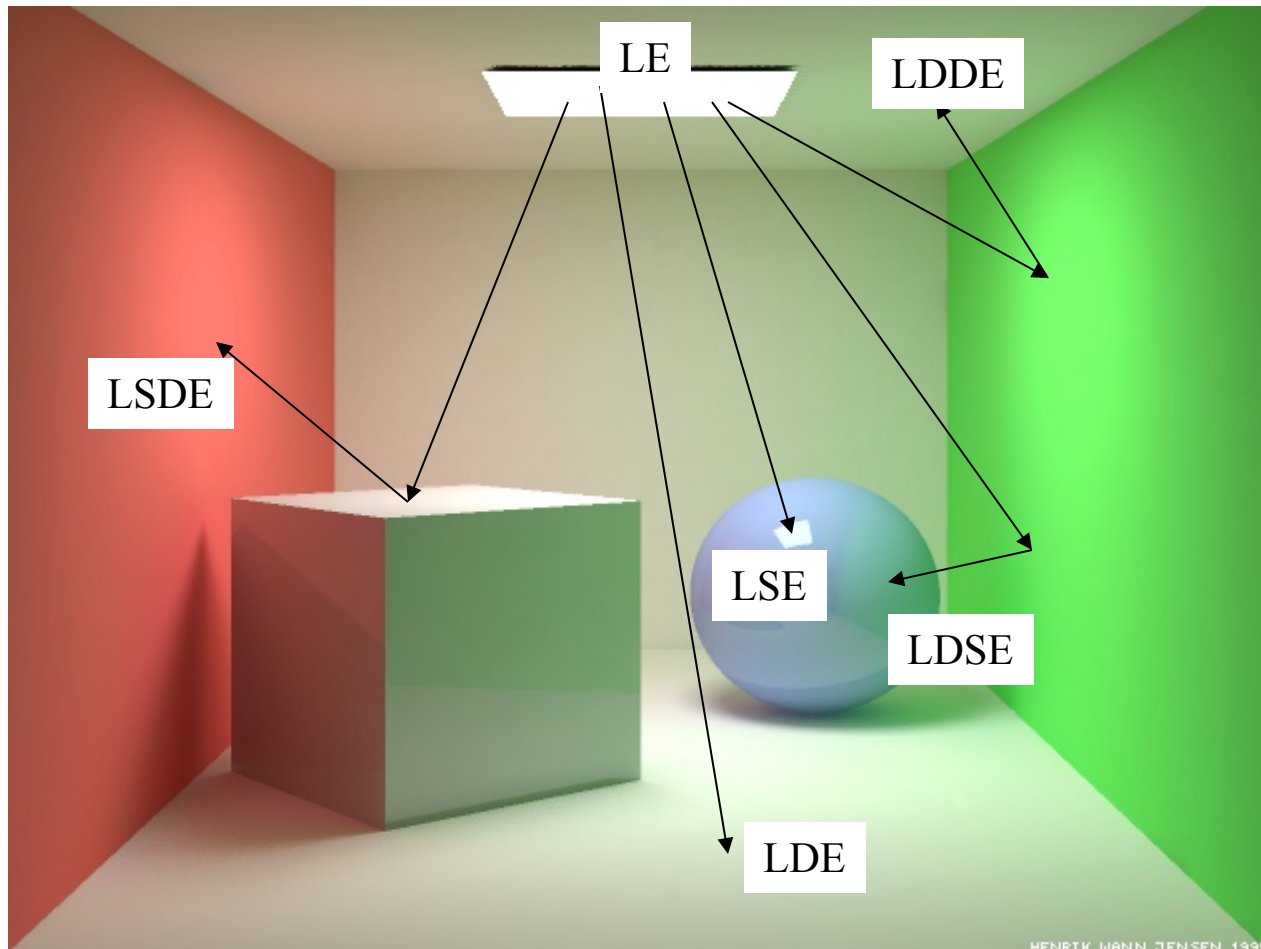
HENRIK WANN JENSEN 1995

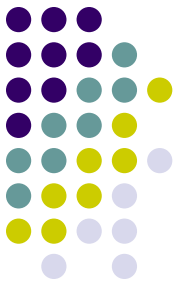
- Find the following:
 - LE
 - LDE
 - LSE
 - LDDE
 - LDSE
 - LSDE

Radiosity Cornell box,
due to Henrik wann
Jensen,
<http://www.gk.dtu.dk/~hwj>, rendered with
ray tracer



More Complex Light Paths

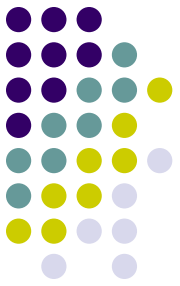


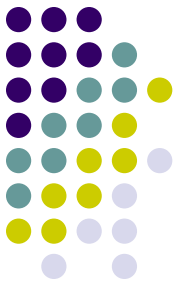


The OpenGL Model

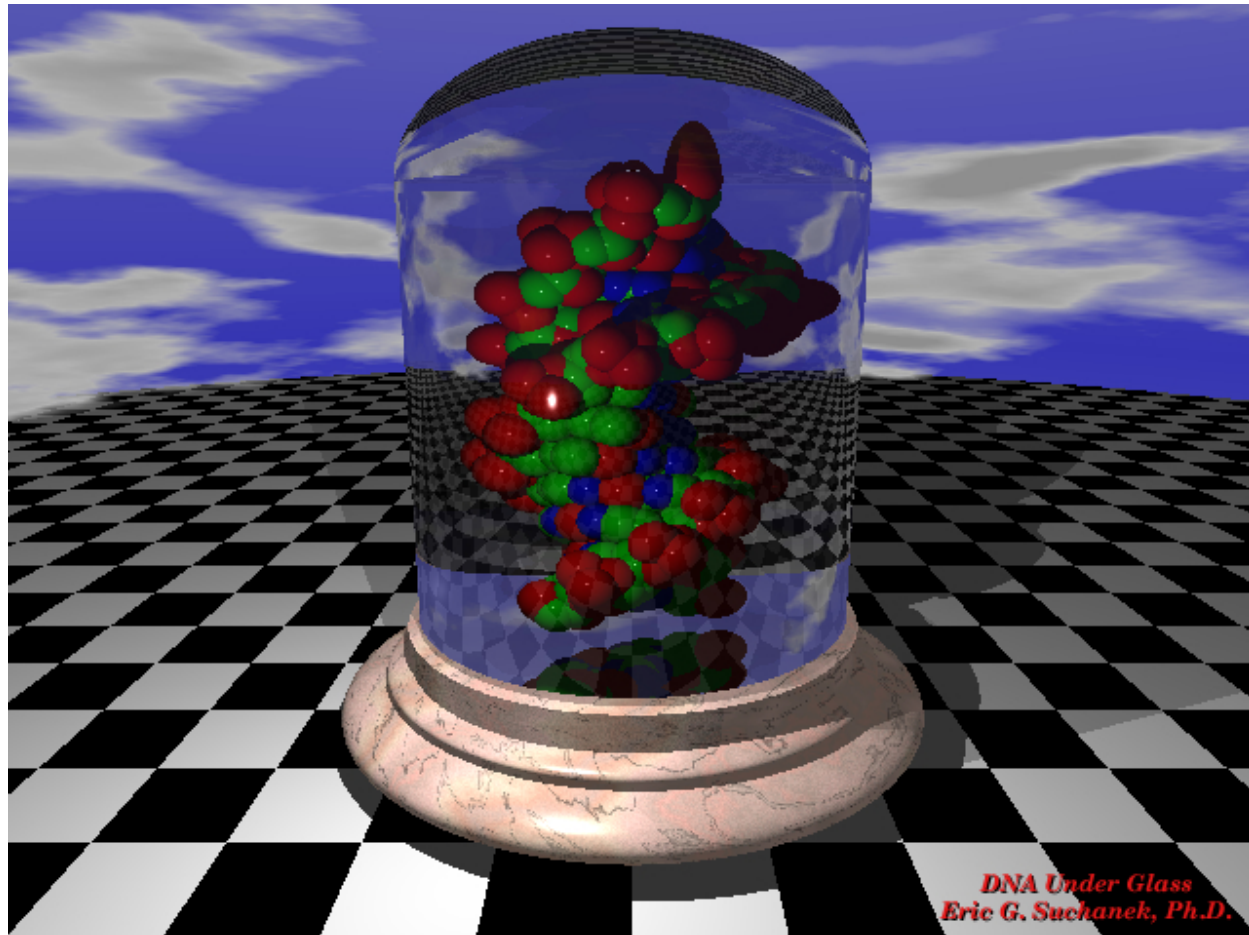
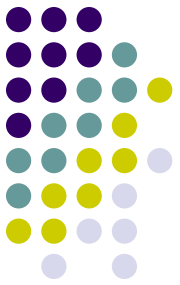
- The “standard” graphics lighting model captures only $L(D|S)E$
- It is missing:
 - Light taking more than one diffuse bounce: LD^*E
 - Should produce an effect called color bleeding, among other things
 - Approximated, grossly, by ambient light
 - Light refracted through curved glass
 - Consider the refraction as a “mirror” bounce: $LDSE$
 - Light bouncing off a mirror to illuminate a diffuse surface: $LS+D+E$
 - Many others
 - Not sufficient for photo-realistic rendering

Raytraced Images

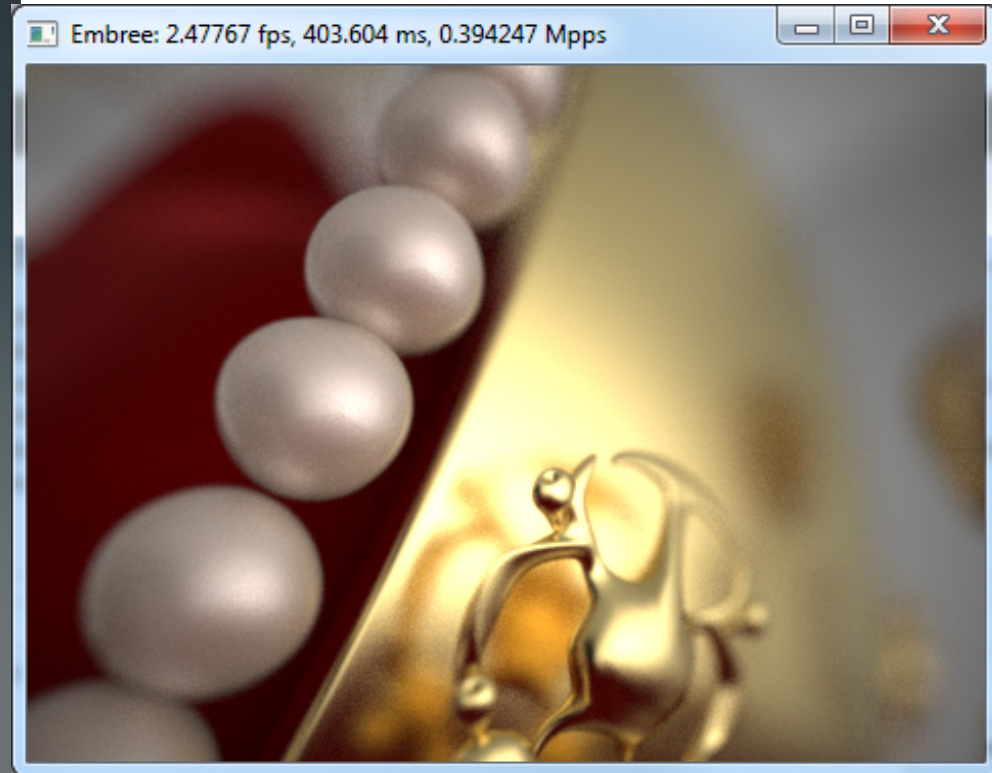




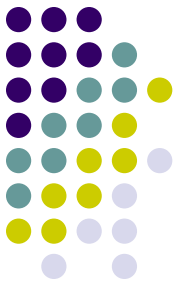
Kettle, Mike
Miller, POV-
Ray



The previous slides now look like amateur hour...



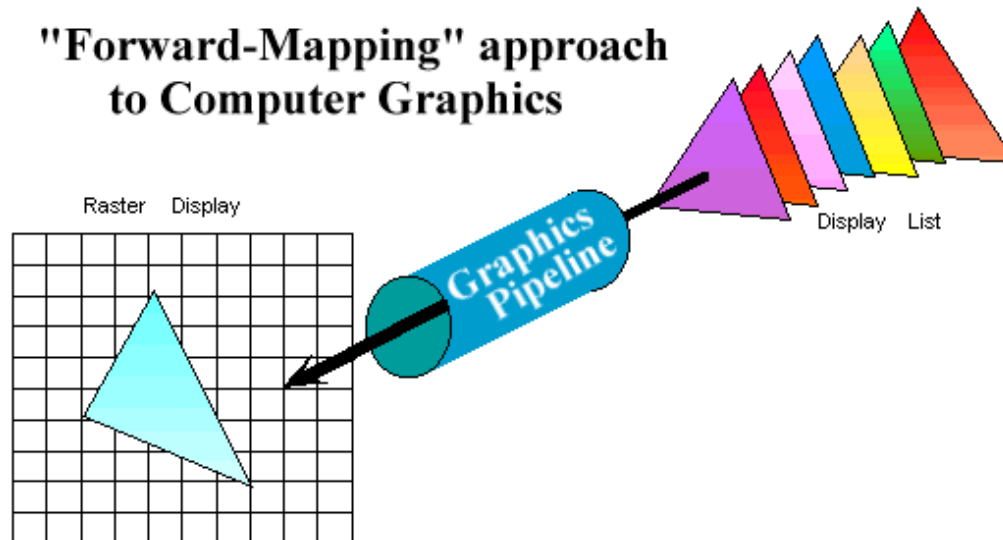
Model courtesy of Martin Lubich, www.loramel.net
HDR light courtesy of Lightmap Ltd, www.lightmap.co.uk



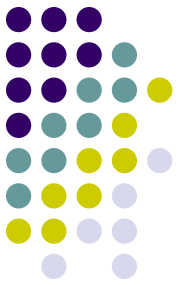
Graphics Pipeline Review

- Properties of the Graphics Pipeline
 - Primitives are transformed and projected (not depending on display resolution)
 - Primitives are processed one at a time
 - Forward-mapping from geometrical space to image space

**"Forward-Mapping" approach
to Computer Graphics**

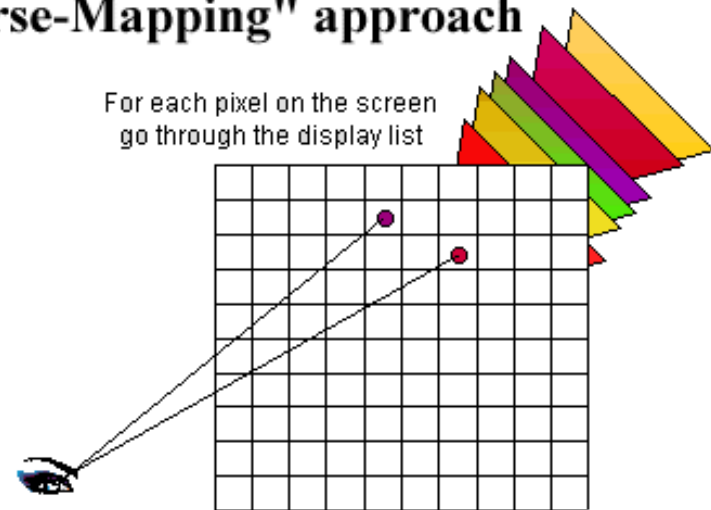


Alternative Approaches: Ray CASTING (not Ray TRACING)



Ray-casting searches along lines of sight, or rays, to determine the primitive that is visible along it.

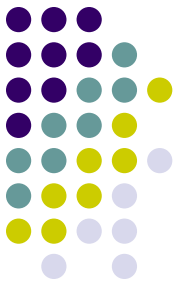
"Inverse-Mapping" approach



Properties of ray-casting:

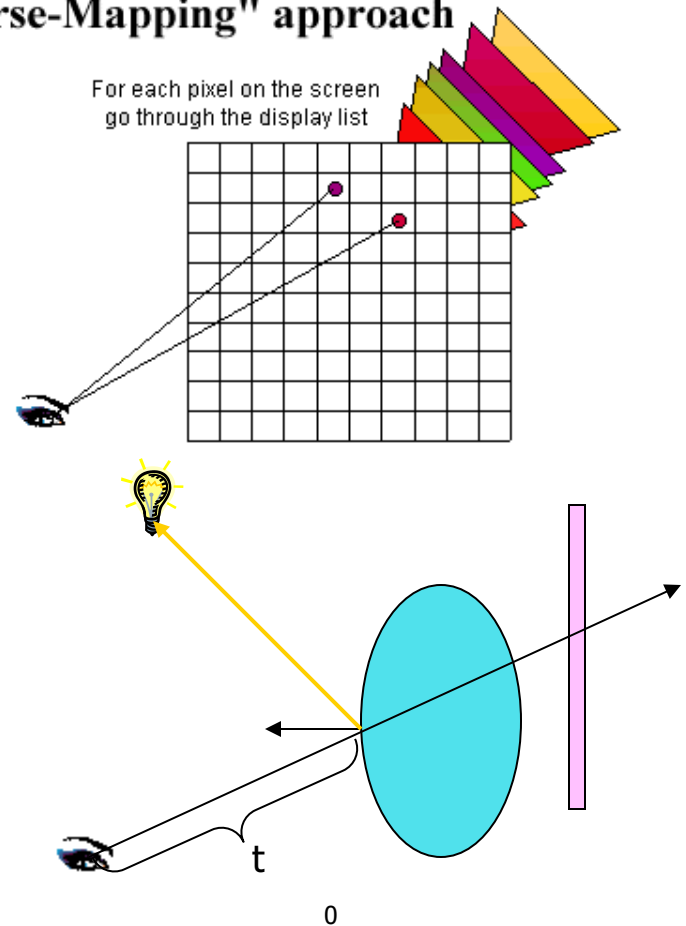
- Go through all primitives at each pixel
- Image space sample first
- Analytic processing afterwards

Ray Casting Overview



- For every pixel shoot a ray from the eye through the pixel.
- For every object in the scene
 - Find the point of intersection with the ray closest to (and in front of) the eye
 - Compute normal at point of intersection
- Compute color for pixel based on point and normal at intersection closest to the eye (e.g. by Phong illumination model).

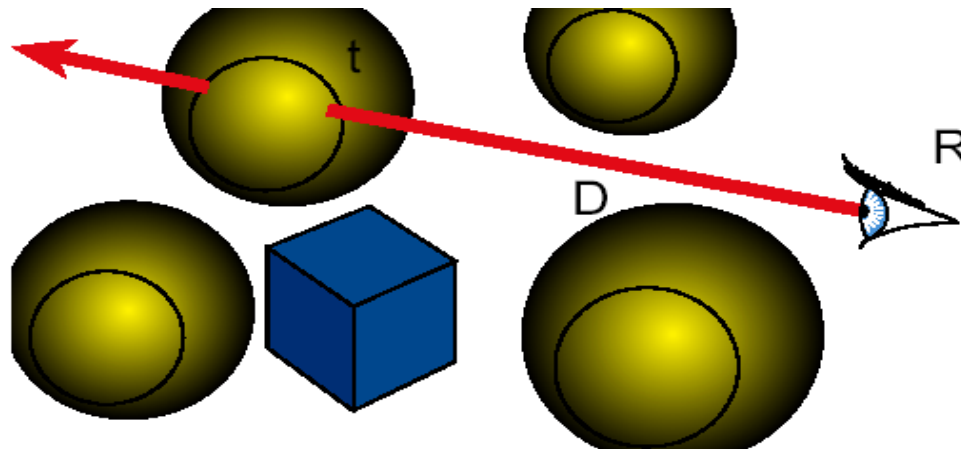
"Inverse-Mapping" approach

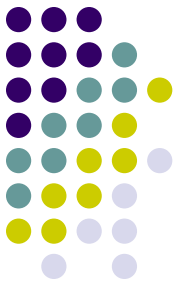




Ray Casting

- Ray Cast (Point R , Ray D) {
 foreach object in the scene
 find minimum $t > 0$ such that $R + t D$ hits object
 if (object hit)
 return object
 else return background object
}

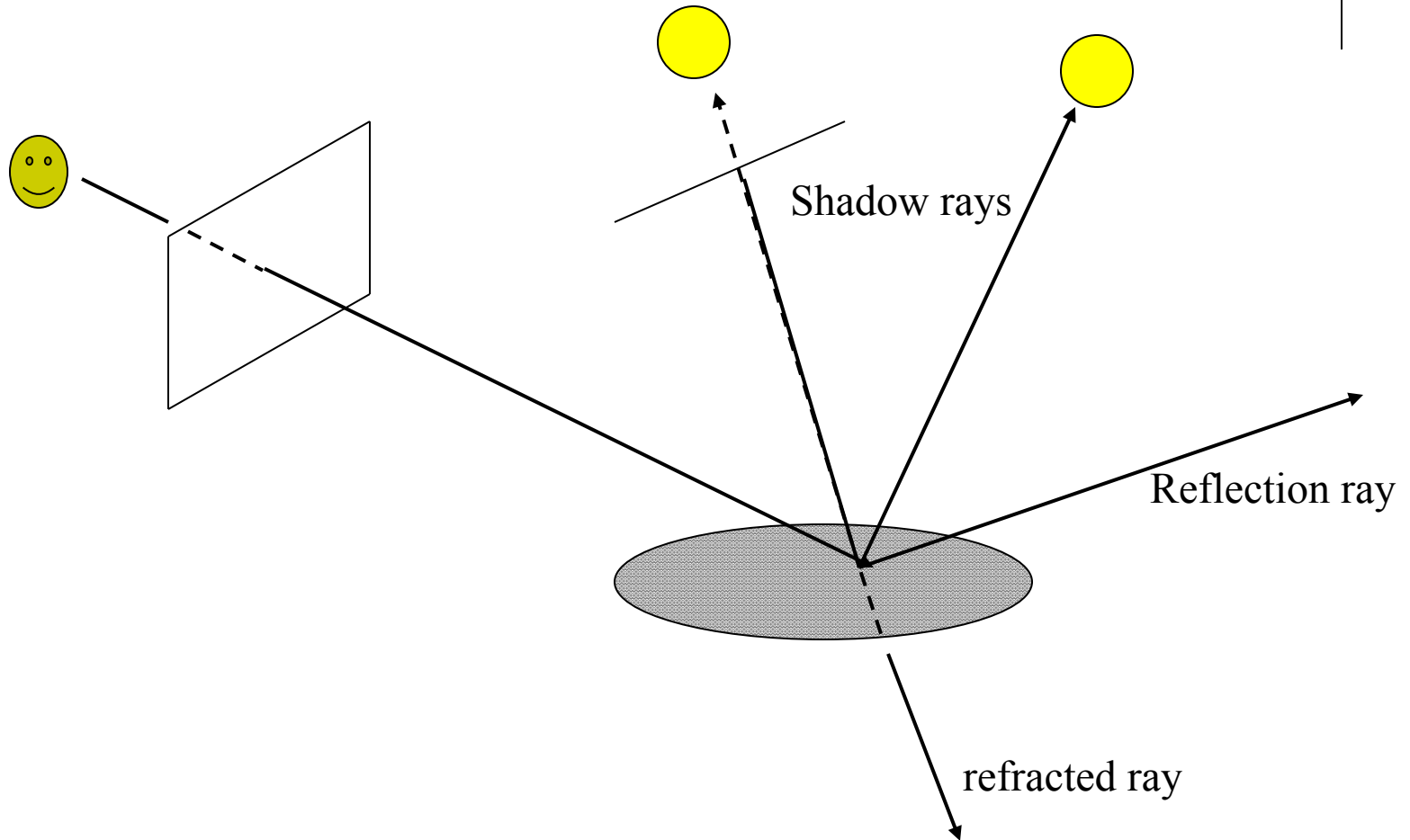


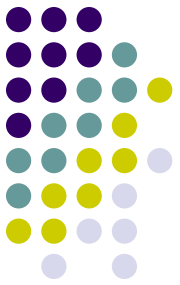


Raytracing

- Cast rays from the eye point the same way as ray casting
 - Builds the image pixel by pixel, one at a time
- Cast additional rays from the hit point to determine the pixel color
 - Shoot rays toward each light. If they hit something, then the object is shadowed from that light, otherwise use “standard” model for the light
 - Reflection rays for mirror surfaces, to see what should be reflected in the mirror
 - Refraction rays to see what can be seen through transparent objects
 - Sum all the contributions to get the pixel color

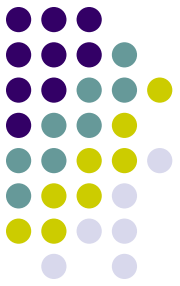
Raytracing





Recursive Ray Tracing

- When a reflected or refracted ray hits a surface, repeat the whole process from that point
 - Send out more shadow rays
 - Send out new reflected ray (if required)
 - Send out a new refracted ray (if required)
 - Generally, reduce the weight of each additional ray when computing the contributions to surface color
 - Stop when the contribution from a ray is too small to notice or maximum recursion level has been reached



Raytracing Implementation

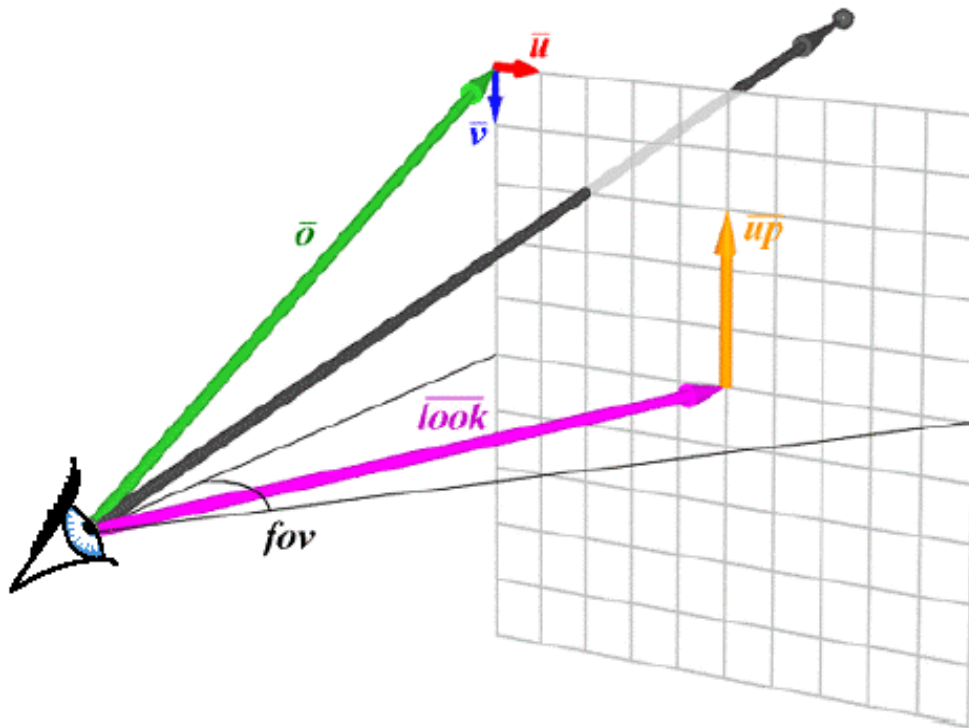
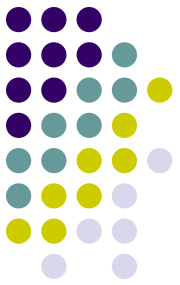
- Raytracing breaks down into two tasks:
 - Constructing the rays to cast
 - Intersecting rays with geometry
- The former problem is simple vector arithmetic
- Intersection is essentially root finding (as we will see)
 - Any root finding technique can be applied
- Intersection calculation can be done in world coordinates or model coordinates



Constructing Rays

- Define rays by an initial point and a direction: $\mathbf{x}(t) = \mathbf{x}_0 + t\mathbf{d}$
- Eye rays: Rays from the eye through a pixel
 - Construct using the eye location and the pixel's location on the image plane. $\mathbf{X}_0 = \mathbf{eye}$
- Shadow rays: Rays from a point on a surface to the light.
 - $\mathbf{X}_0 =$ point on surface
- Reflection rays: Rays from a point on a surface in the reflection direction
 - Construct using laws of reflection. $\mathbf{X}_0 =$ surface point
- Transmitted rays: Rays from a point on a transparent surface through the surface
 - Construct using laws of refraction. $\mathbf{X}_0 =$ surface point

From Pixels to Rays



$$\vec{r}_u = \frac{\vec{look} \times \vec{up}}{|\vec{look} \times \vec{up}|}$$

$$\vec{r}_v = \frac{\vec{look} \times \vec{r}_u}{|\vec{look} \times \vec{r}_u|}$$

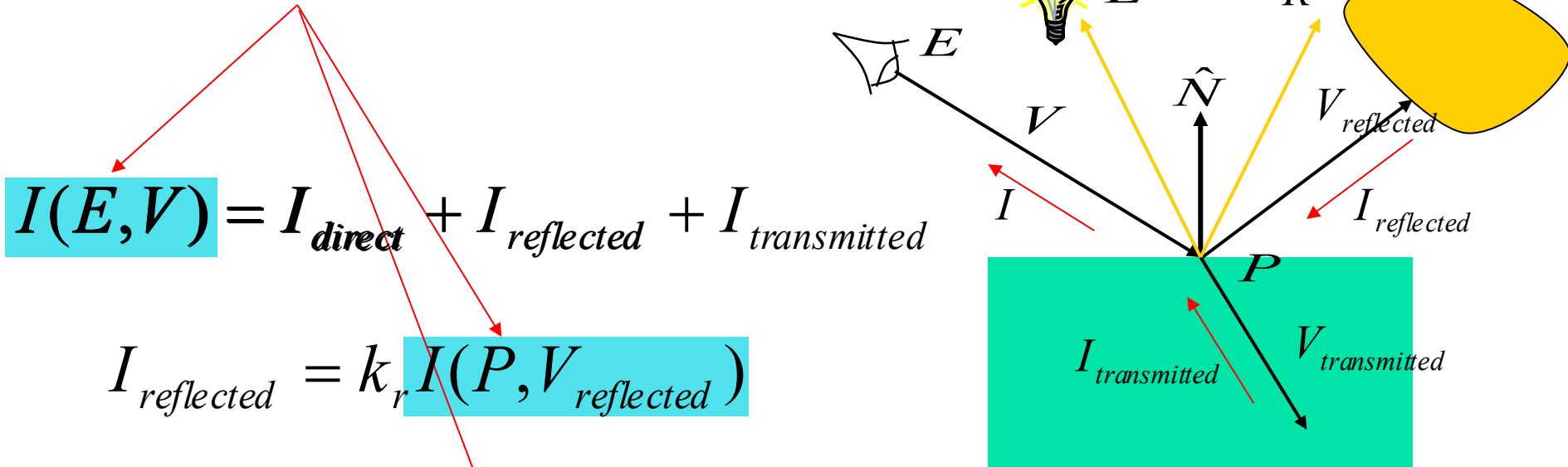
$$\Delta \vec{x}^r = \frac{2 \tan(\text{fov}_x / 2)}{W} \vec{r}_u$$

$$\Delta \vec{y}^r = \frac{2 \tan(\text{fov}_y / 2)}{H} \vec{r}_v$$

$$\vec{r}_{d(i,j)} = \frac{\vec{look}}{|\vec{look}|} + \frac{(2i+1-W)}{2} \Delta \vec{x}^r + \frac{(2j+1-H)}{2} \Delta \vec{y}^r$$

Ray Tracing Illumination

Recursive



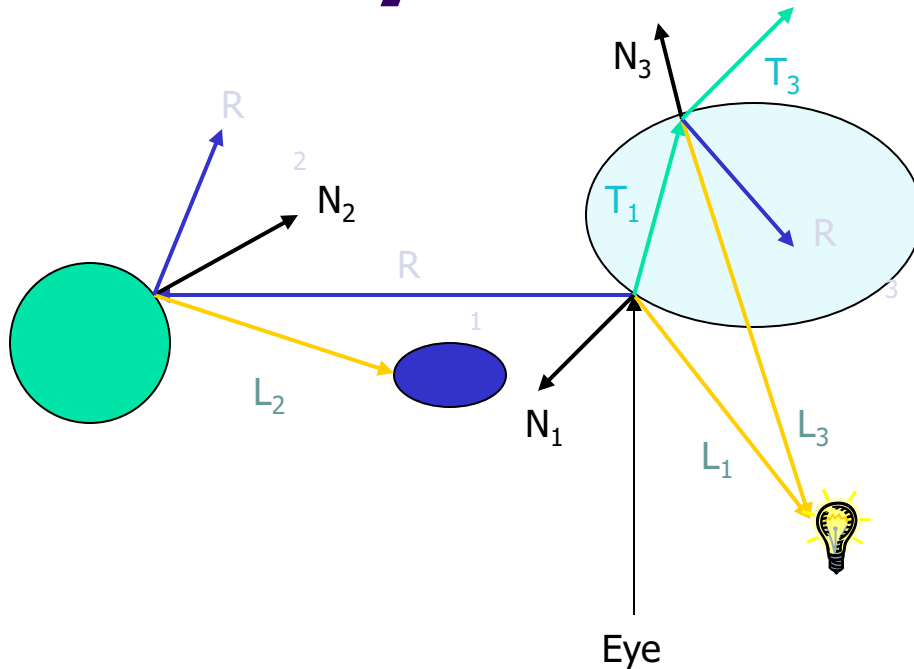
$$I(E, V) = I_{direct} + I_{reflected} + I_{transmitted}$$

$$I_{reflected} = k_r I(P, V_{reflected})$$

$$I_{transmitted} = k_t I(P, V_{transmitted})$$

$$I_{direct} = k_a I_{ambient} + I_{light} \left[k_d (\hat{N} \cdot \hat{L}) + k_s (-\hat{V} \cdot \hat{R})^{n_{shiny}} \right]$$

The Ray Tree



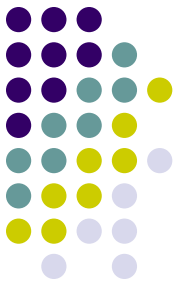
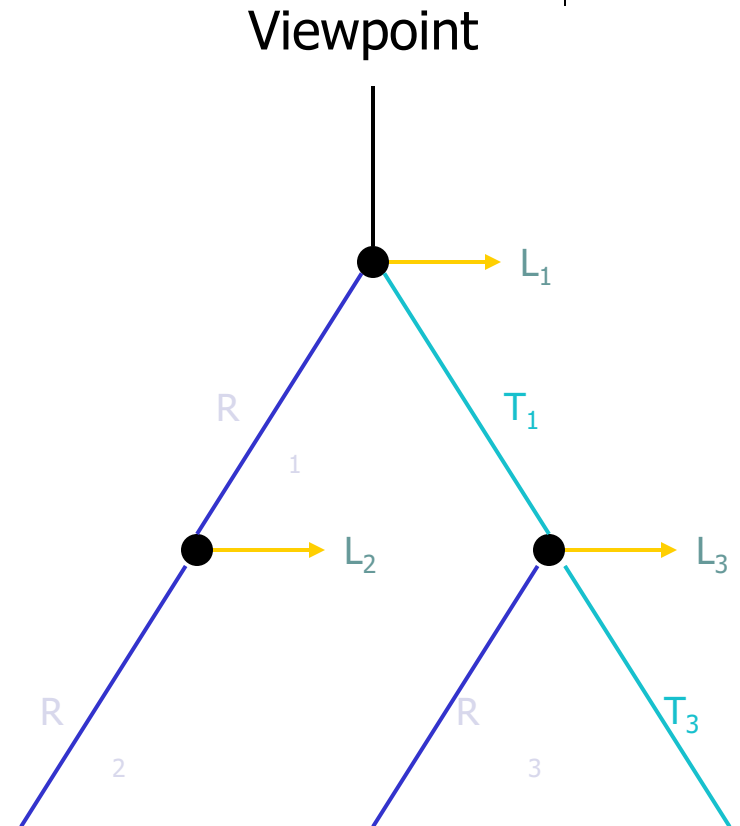
N_i surface normal

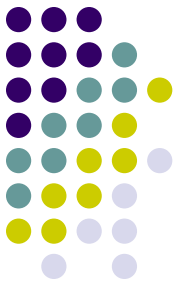
R_i reflected ray

L_i shadow ray

T_i transmitted (refracted) ray

CS 535
Pseudo-code

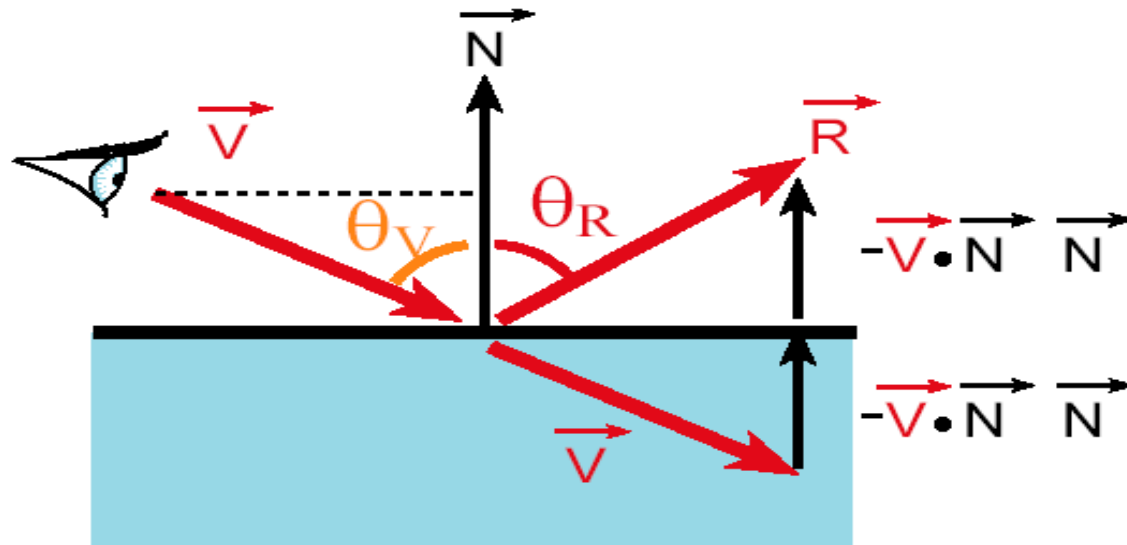


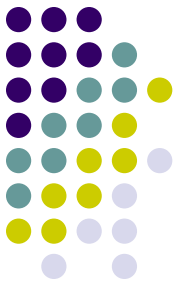


Reflection

- Reflection angle = view angle

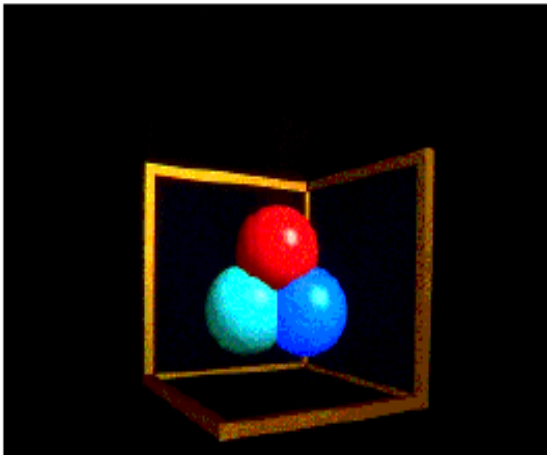
$$\vec{R} = \vec{V} - 2(\vec{V} \bullet \vec{N})\vec{N}$$



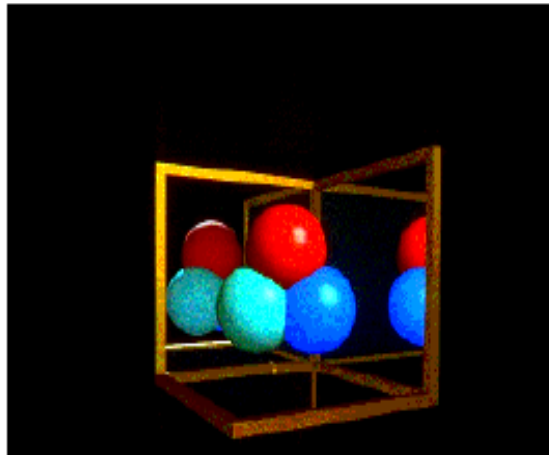


Reflection

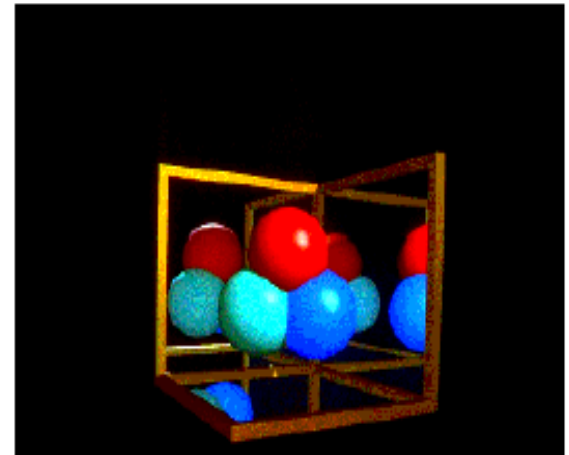
- The maximum depth of the tree affects the handling of refraction
- If we send another reflected ray from here, when do we stop? 2 solutions (complementary)
 - Answer 1: Stop at a fixed depth.
 - Answer 2: Accumulate product of reflection coefficients and stop when this product is too small.



0 recursion

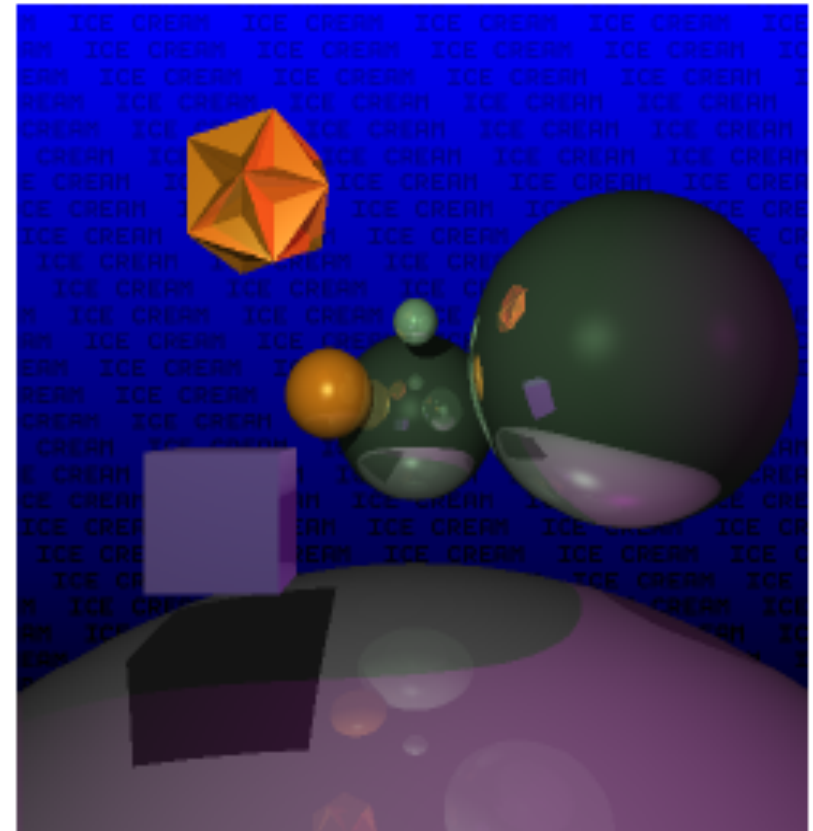
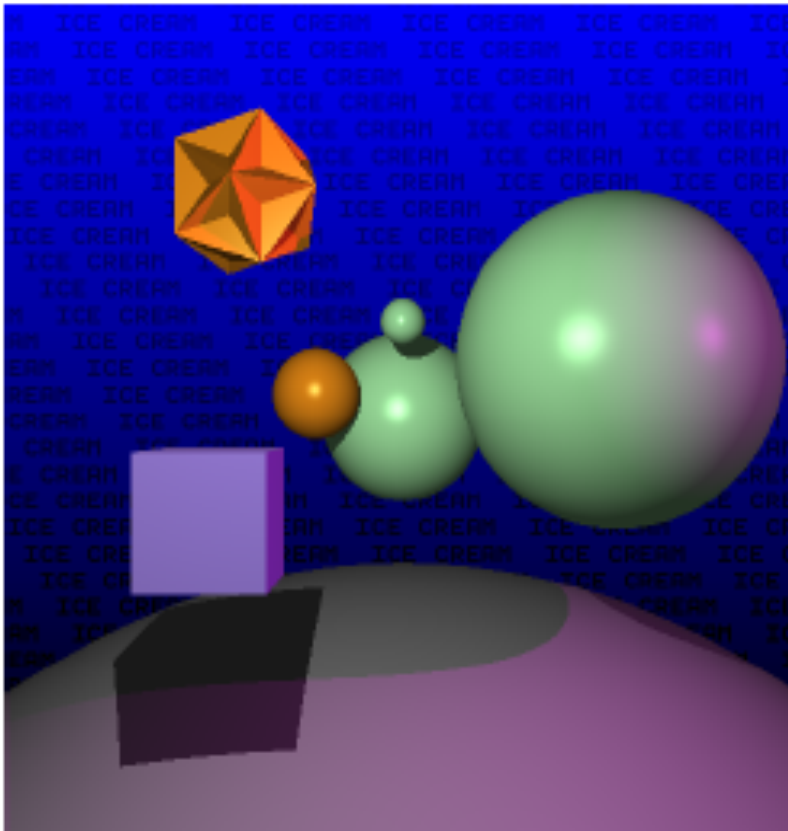
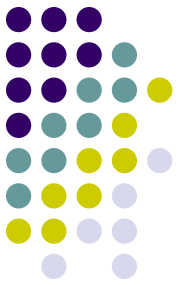


1 recursion



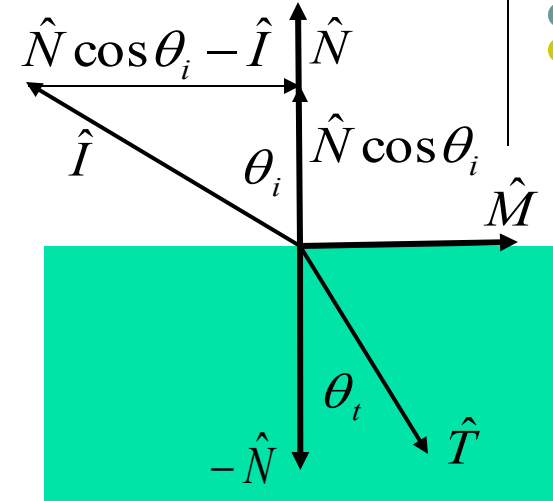
2 recursions

Reflection



Refraction

Snell's Law $\frac{\sin \theta_t}{\sin \theta_i} = \frac{\eta_i}{\eta_t} = \eta_r$



Note that \hat{I} is the negative of the incoming ray

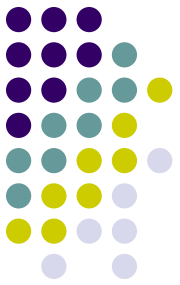


Pseudo Code for Ray Tracing

```
rgb lsou;           // intensity of light source
rgb back;           // background intensity
rgb ambi;           // ambient light intensity

Vector L            // vector pointing to light source
Vector N            // surface normal
Object objects [n]  // list of n objects in scene
float Ks [n]         // specular reflectivity factor for each object
float Kr [n]         // refractivity index for each object
float Kd [n]         // diffuse reflectivity factor for each object
Ray r;

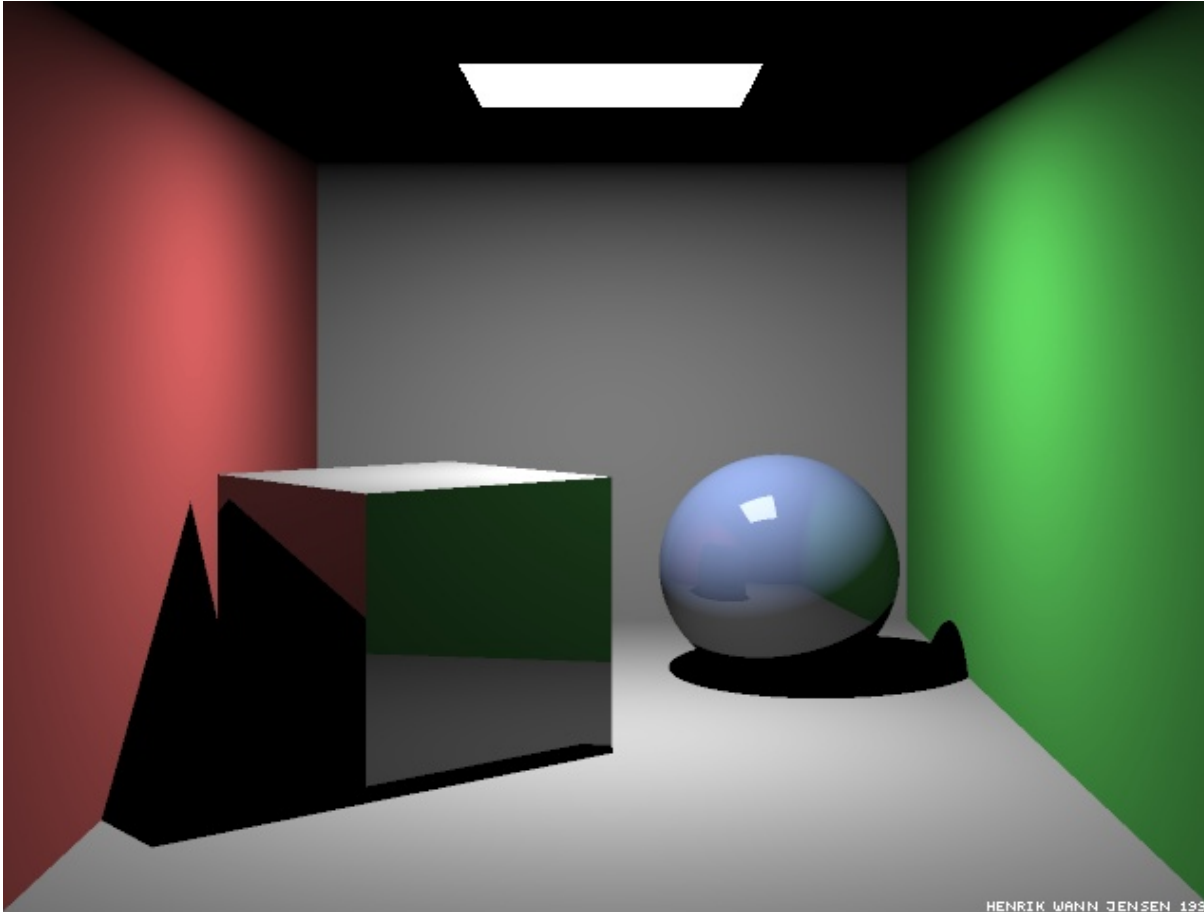
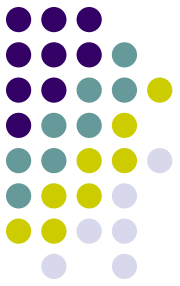
void raytrace() {
    for (each pixel P of projection viewport in raster order) {
        r = ray emanating from viewer through P
        int depth = 1; // depth of ray tree consisting of multiple paths
        the pixel color at P = intensity(r, depth)
    }
}
```



```
rgb intensity (Ray r, int depth) {
    Ray  flec, frac;
    rgb   spec, refr, dull, intensity;

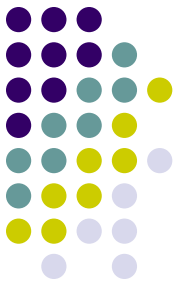
    if (depth >= 5) intensity = back;
    else {
        find the closest intersection of r with all objects in scene
        if (no intersection) {
            intensity =back;
        } else {
            Take closest intersection which is object[j]
            compute normal N at the intersection point
            if (Ks[j] >0) { // non-zero specular reflectivity
                compute reflection ray flec;
                refl = Ks[j]*intensity(flec, depth+1);
            } else refl =0;
            if (Kr[j]>0) { // non-zero refractivity
                compute refraction ray frac;
                refr = Kr[j]*intensity(frac, depth+1);
            } else refr =0;
            check for shadow;
            if (shadow) direct = Kd[j]*ambi
            else direct = Phong illumination computation;
            intensity = direct + refl +refr;
        } }
    return intensity; }
```

Raytraced Cornell Box



Which paths
are missing?

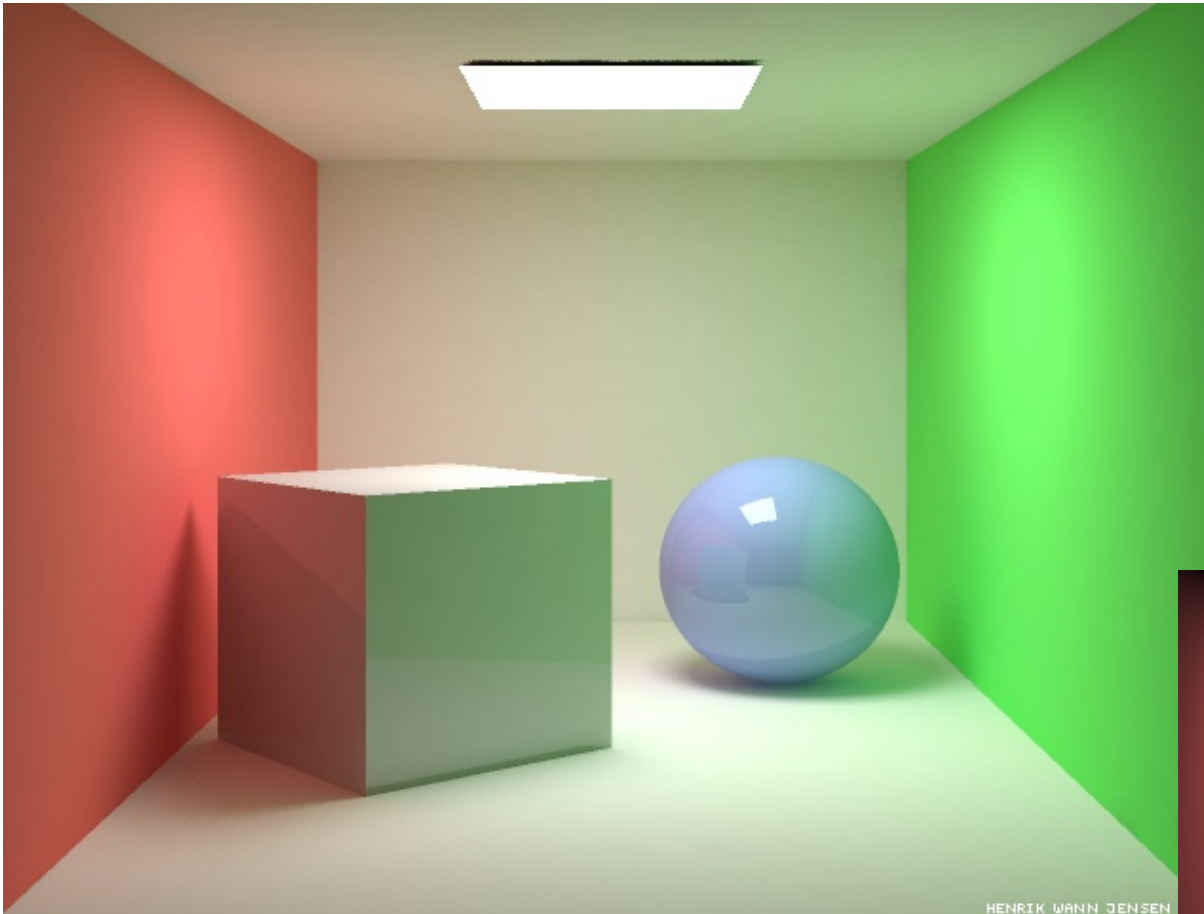
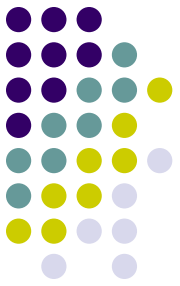
Ray-traced Cornell box, due
to Henrik Jensen,
<http://www.gk.dtu.dk/~hwj>



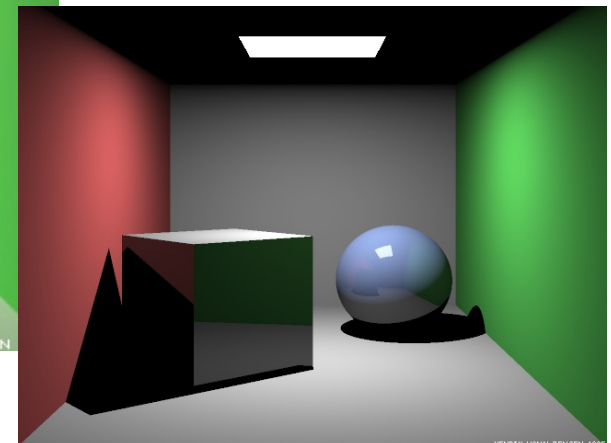
Paths in RayTracing

- Ray Tracing
 - Captures LDS*E paths: Start at the eye, any number of specular bounces before ending at a diffuse surface and going to the light
- Raytracing cannot do:
 - LS*D+E: Light bouncing off a shiny surface like a mirror and illuminating a diffuse surface
 - LD+E: Light bouncing off one diffuse surface to illuminate others
- Basic problem: The raytracer doesn't know where to send rays out of the diffuse surface to capture the incoming light
- Also a problem for rough specular reflection
 - Fuzzy reflections in rough shiny objects
- Need other rendering algorithms that get more paths

A Better Rendered Cornell Box



HENRIK WANN JENSEN



HENRIK WANN JENSEN 1999

Level of detail

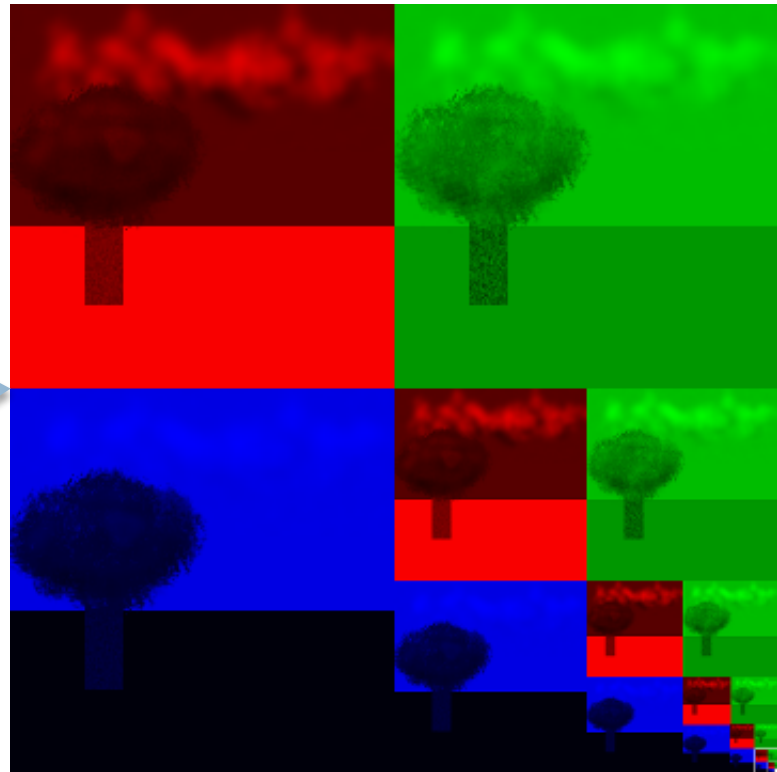


Mipmaps



- ❑ Mipmaps: pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts.
- ❑ Widely used in 3D computer games, flight simulators and other 3D imaging systems.
- ❑ In use, it is called “mipmapping.”
- ❑ The letters “MIP” in the name are an acronym of the Latin phrase multum in parvo, meaning “much in little”.

Mipmaps



Level of detail (LOD) techniques



- level of detail: decreasing the complexity of some 3D object representations, because they
 - are far away
 - are moving fast
 - are not important
- increases the efficiency of rendering by decreasing the workload on graphics pipeline stages
 - reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast.

Types of LOD



- Two types:
 - Discrete LoD (DLoD)
 - Continuous LoD (CLoD)



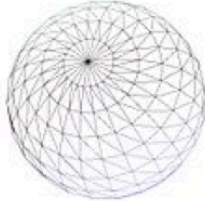
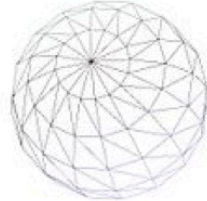

Discrete LoD (DLOD)

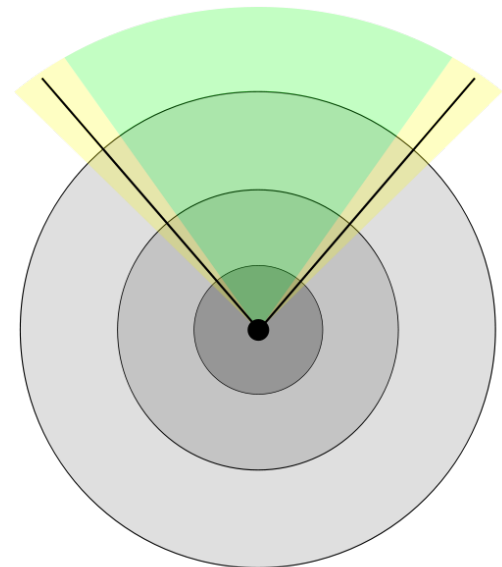


□ Discrete LoD (DLoD)

- Make a fixed amount of models, ranging from highest quality to coarse approximation & render appropriate one based on importance factor
- Fastest in practice, but leads to “popping”



Visual impact comparisons and measurements

Image					
Vertices	~5500	~2880	~1580	~670	140
Notes	Maximum detail, for closeups.				Minimum detail, very far objects.



Discrete LoD example



	Brute	DLOD
Rendered images	 A rendered image of a scene with several spheres of varying sizes and colors (green, blue, purple) on a white background with scattered black dots. The image shows significant aliasing artifacts, particularly around the edges of the spheres.	 A rendered image of the same scene as the Brute image, but using Discrete Level of Detail (DLOD). The image shows significantly fewer aliasing artifacts, with smoother edges and less visible noise.
Render time	27.27 ms	1.29 ms
Scene vertices (thousands)	2328.48	109.44

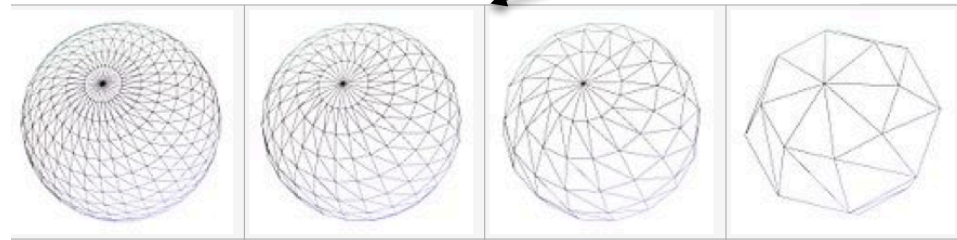
OK, how do we create coarse versions?



□ How do we take



and make these?



□ Answer: surface decimation

Decimation of Triangle Meshes

Paper by W.J.Schroeder et.al

Presented by Guangfeng Ji



Goal

- ✦ Reduce the total number of triangles in a triangle mesh
- ✦ Preserve the original topology and a good approximation of the original geometry



Overview

- ★ A multiple-pass algorithm
- ★ During each pass, perform the following three basic steps on every vertex:
 - Classify the local geometry and topology for this given vertex
 - Use the decimation criterion to decide if the vertex can be deleted
 - If the point is deleted, re-triangulate the resulting hole.
- ★ This vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met.



Feature Edge

- ★ A feature edge exists if the angle between the surface normals of two adjacent triangles is greater than a user-specified “feature angle”.



Characterize Local Geometry and Topology

- ★ Each vertex is assigned one of five possible classifications:
 - Simple vertex
 - Complex vertex
 - Boundary vertex
 - Interior edge vertex
 - Corner vertex



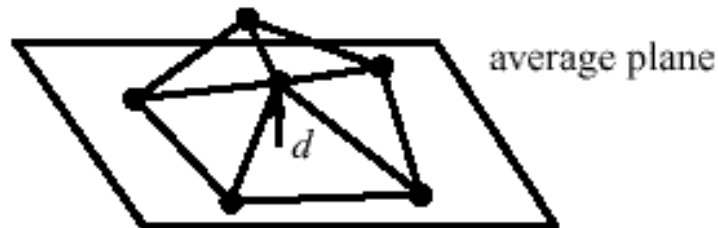
Evaluate the Decimation Criteria

- ✳ Complex vertices are not deleted from the mesh.
- ✳ Use the distance to plane criterion for simple vertices.
- ✳ Use the distance to edge criterion for boundary and interior edge vertices.
- ✳ Corner vertex?



Criterion for Simple Vertices

- ✱ Use the distance to plane criterion.
- ✱ If the vertex is within the specified distance to the average plane, it can be deleted. Otherwise, it is retained.



Overview

- ★ A multiple-pass algorithm
- ★ During each pass, perform the following three basic steps on every vertex:
 - Classify the local geometry and topology for this given vertex
 - Use the decimation criterion to decide if the vertex can be deleted
 - If the point is deleted, re-triangulate the resulting hole.
- ★ This vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met.



Continuous LoD



- Continuous LOD (CLOD)
 - considers the polygon mesh being rendered as a function which must be evaluated requiring to avoid excessive errors which are a function of some heuristic (usually distance) themselves.
 - The given "mesh" function is then continuously evaluated and an optimized version is produced according to a tradeoff between visual quality and performance.

Terrain Rendering



Terrain rendering



- Wikipedia:
 - Terrain rendering covers a variety of methods of depicting real-world or imaginary world surfaces.
 - Most common terrain rendering is the depiction of Earth's surface. ← Hank disagrees when it comes to CG
- Used in various applications to give an observer a frame of reference

Terrain rendering structure



- Actors:
 - terrain database,
 - a central processing unit (CPU),
 - a dedicated graphics processing unit (GPU),
 - a display.
- Software application is configured to start at initial location in the world space.
- The output of the application is screen space representation of the real world on a display.

Terrain rendering details



- CPU identifies and loads terrain data corresponding to initial location from the terrain database
- CPU applies the required transformations to build a mesh of points that can be rendered by the GPU
- Data sent to GPU and GPU completes geometrical transformations, creating screen space objects (i.e., polygons) that create a picture closely resembling the location of the real world.

Generation



- The main tension is between the # of processed polygons and the # of rendered polygons.
 - A very detailed picture of the world might use billions of data points.
- Virtually all terrain rendering applications use level of detail to manage number of data points processed by CPU and GPU.
 - There are several modern algorithms for terrain surfaces generating.

Example: ROAM



- ROAM: Real-time optimally adapting mesh.
 - Continuous level of detail algorithm that optimizes terrain meshes.
 - Premise: sometimes it is more effective to send a small amount of unneeded polygons to the GPU, rather than burden the CPU with LOD (Level of Detail) calculations.
 - Result: produce high quality displays while being able to maintain real-time frame rates.
 - ROAM provides control over scene quality versus performance in order to provide HQ scenes while retaining real-time frame rates on hardware.

ROAM in action

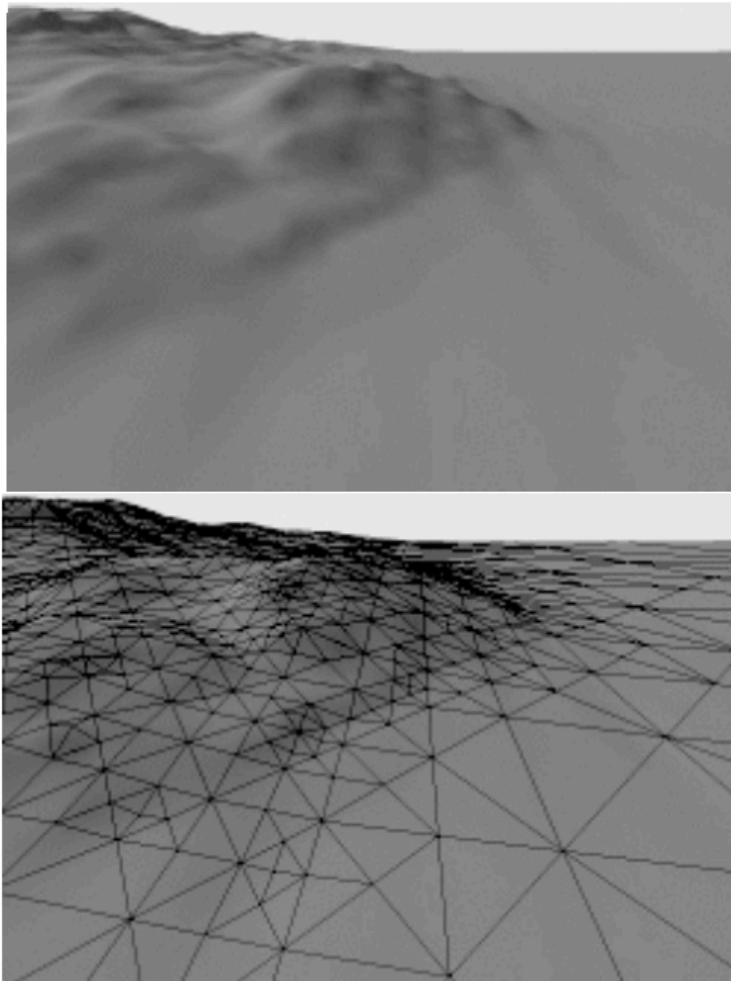


Figure 1: Example of ROAM terrain.

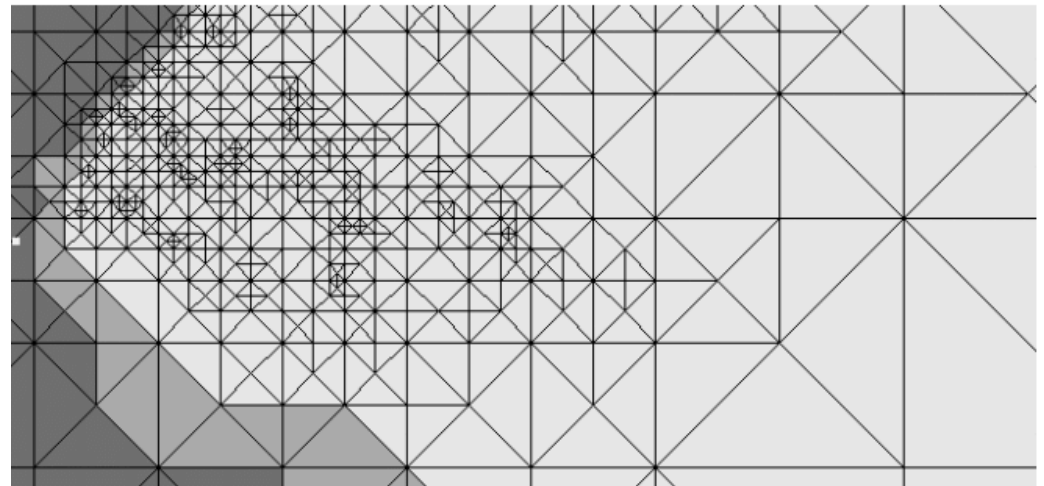


Figure 2: Triangulation for example frame, with eye looking right.

ROAM internals



to the midpoint v_c of its *base edge* (v_0, v_1) . The *left child* of T is $T_0 = (v_c, v_a, v_0)$, while the *right child* of T is $T_1 = (v_c, v_1, v_a)$. The rest of the triangle bintree is defined by recursively repeating this splitting process.

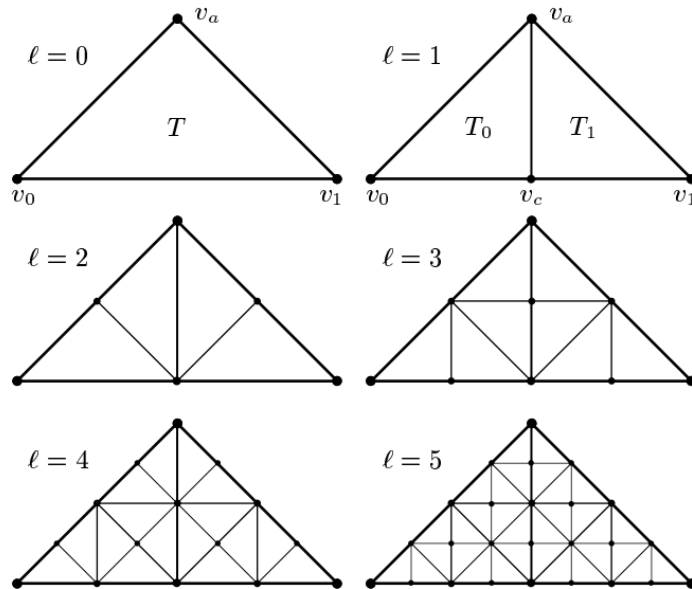


Figure 3: Levels 0–5 of a triangle bintree.

4.2 Dynamic Continuous Triangulations

Meshes in world space are formed by assigning world-space positions $w(v)$ to each bintree vertex. A set of bintree triangles forms a continuous mesh when any two triangles either overlap nowhere, at a common vertex, or at a common edge. We refer to such continuous meshes as *bintree triangulations* or simply *triangulations*. Figure 4 shows a typical neighborhood about a triangle T within a

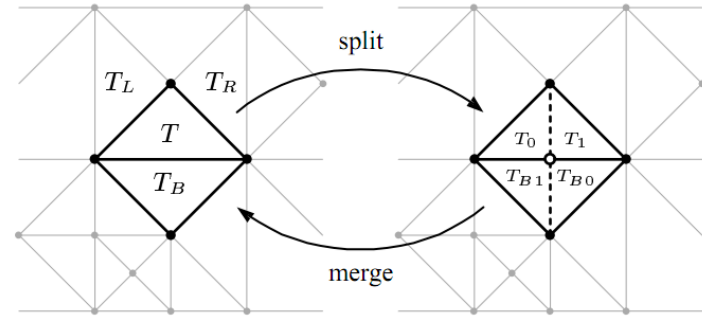
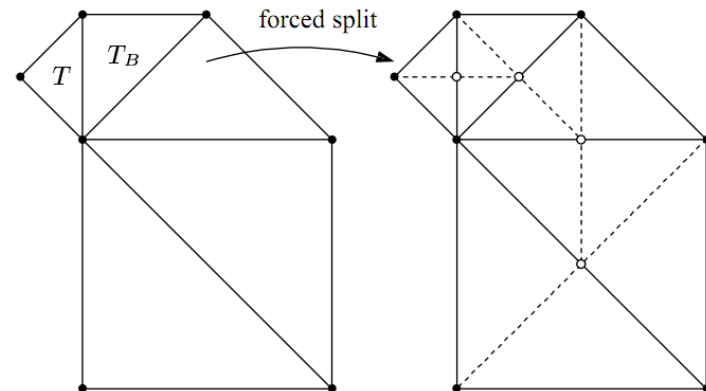


Figure 4: Split and merge operations on a bintree triangulation. A typical neighborhood is shown for triangle T on the left.

A triangle T in a triangulation cannot be split immediately when its base neighbor T_B is from a coarser level. To force T to be split, T_B must be forced to split first, which may require further splits in a recursive sequence. A case requiring a total of four splits is depicted in Figure 5. Such *forced splits* are needed for the optimization algorithm in Section 5.



Spatial Search Structures



Spatial Search Data Structures

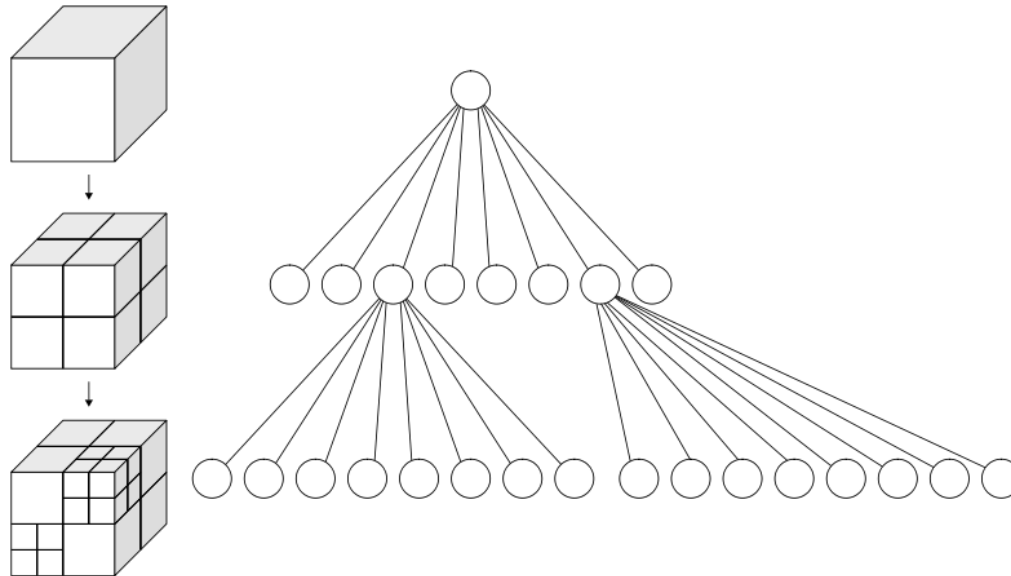


- Organize geometry so you can quickly find geometry in a given region.
- Examples:
 - Octree
 - k-d tree
 - Binary space partitioning
- Usages:
 - Collision detection
 - Culling
 - Ray tracing

Octrees



- Oct + tree = octree (one 't')
- Tree data structure
 - Internal node: has eight children, corresponding to 8 octants
 - Leaf node: contains some number of points

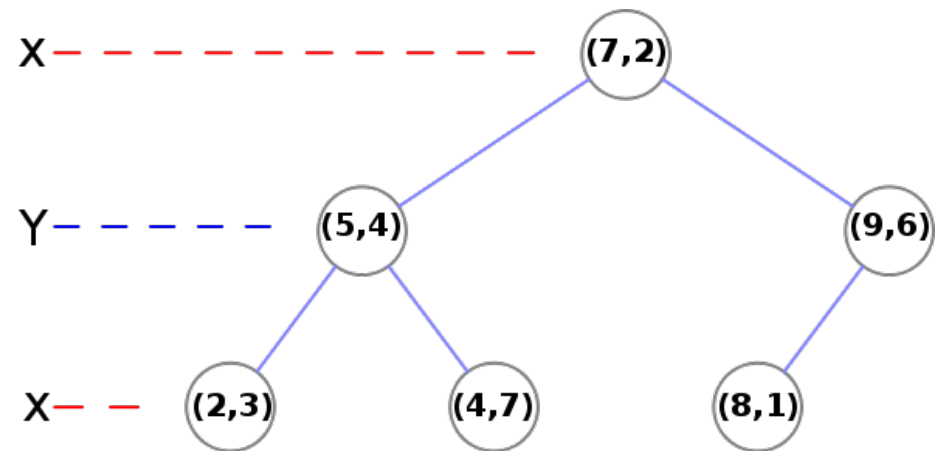
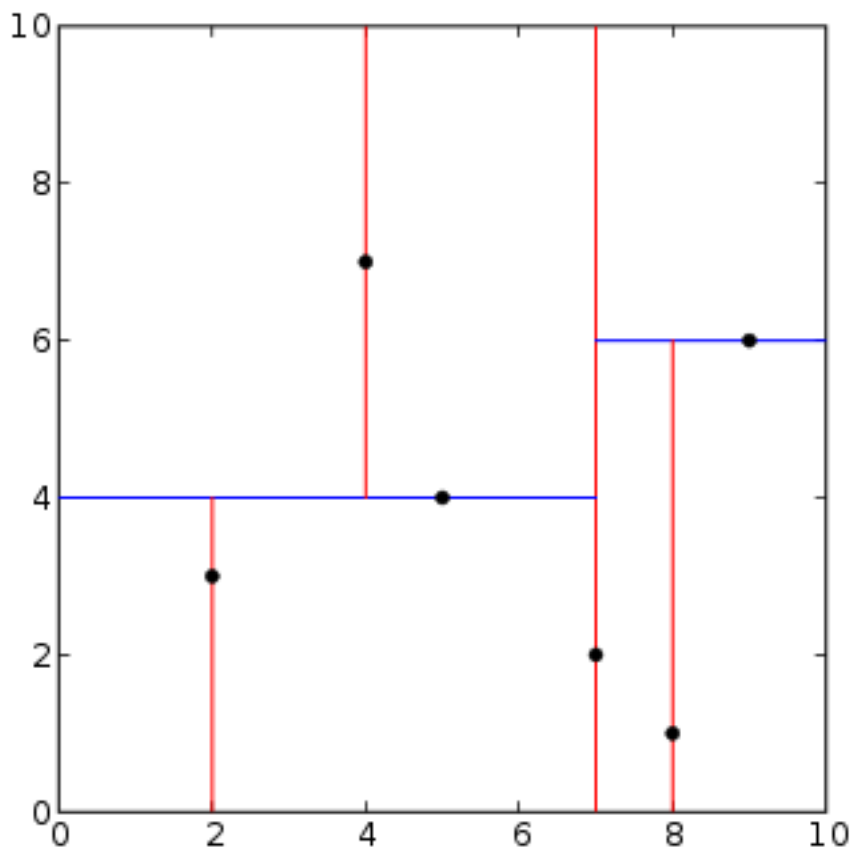




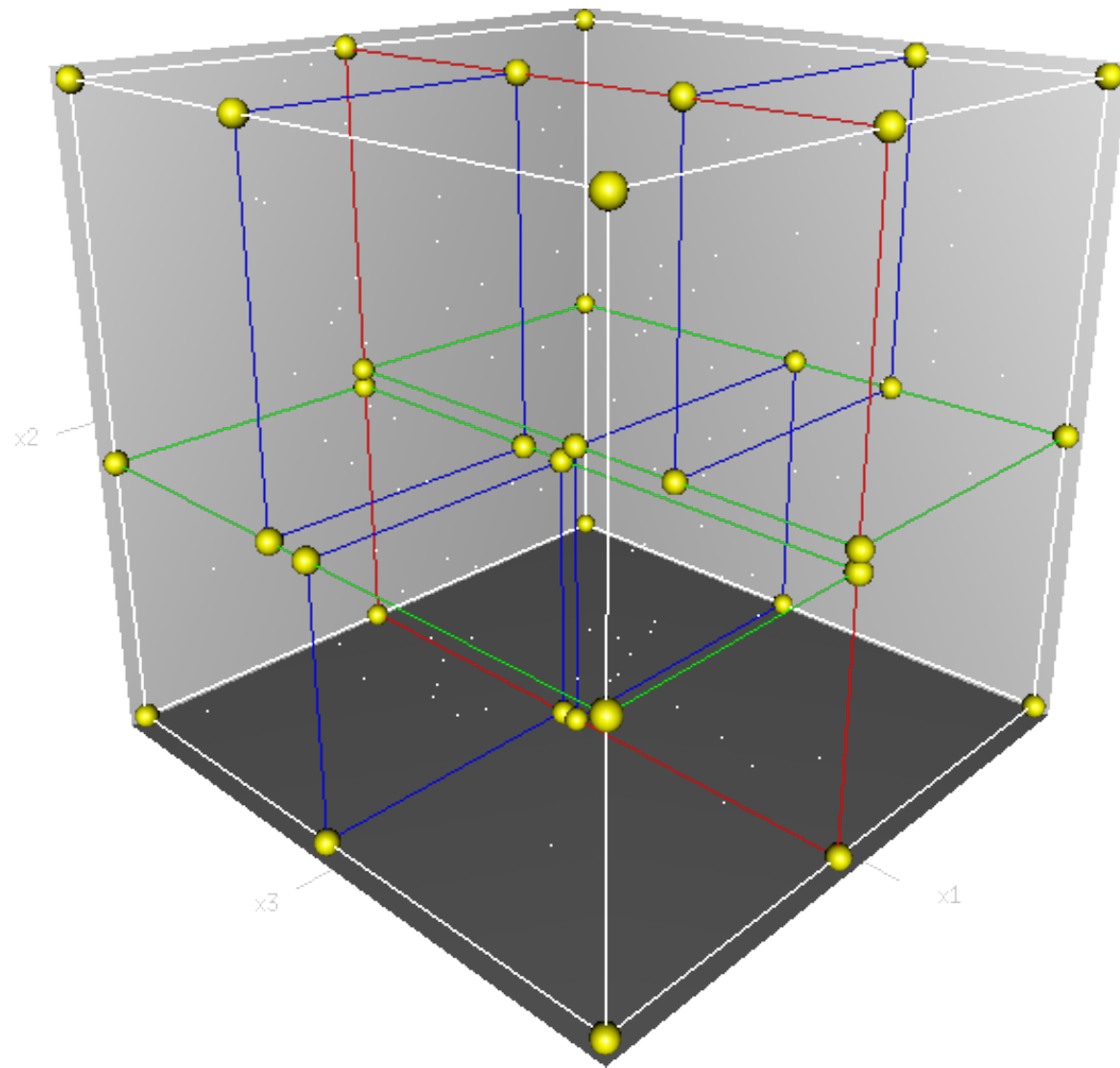
k-d trees

- k-d = k-dimensional tree
 - We are interested in $k=3$
- Node roles:
 - Every leaf node is a point
 - Every internal node divides space into two parts using a plane. Also contains a point.
 - Each plane is axis-aligned, i.e., $X=a$, $Y=b$, or $Z=c$.
 - Alternate between axes as you descend the tree

k-d tree



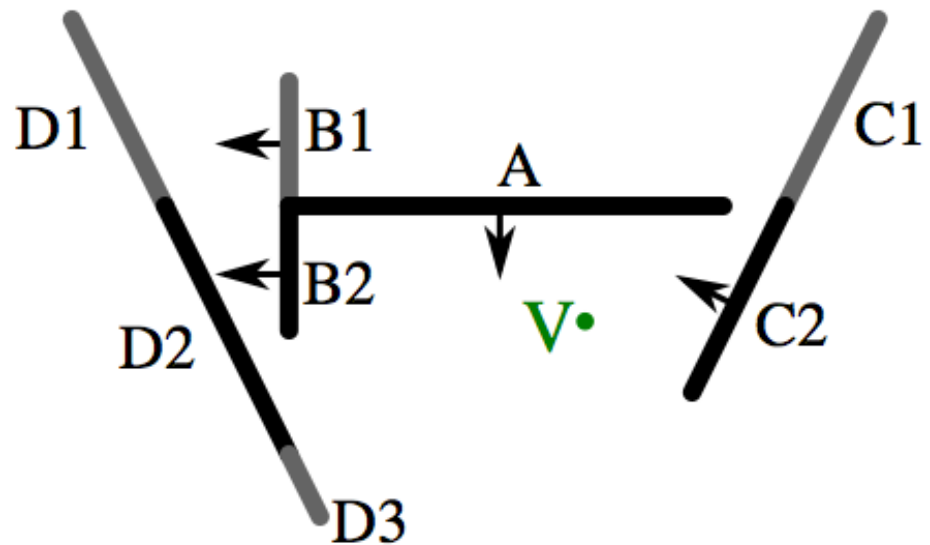
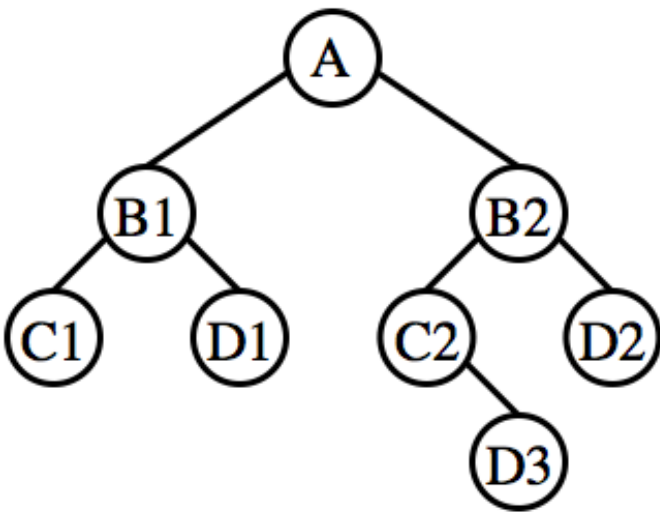
k-d tree



Binary Space Partitioning



- General form of k-d trees, using arbitrary planes, not just $X=a$, $Y=b$, $Z=c$
- Associated tree (BSP trees) can be used for spatial searches.



View Frustum Culling



View Frustum Culling



- Viewing frustum culling: the process of removing objects that lie completely outside the viewing frustum from the rendering process.
- Rendering these objects would be a waste of time since they are not directly visible.

View Frustum Culling



- Spatial search structures (octree, k-d, BSP) can dramatically accelerate view frustum culling.
 - Need correct granularity though.
- Speedups are very application-dependent
 - Best case scenario: you are zoomed in on very complex scene
 - Worst case scenario: you are zoomed out on simple scene

Collision Detection



Collision Detection



- Collision detection: as objects in the scene move, figure out when they collide and perform appropriate action (typically bouncing)
- Game setting: 30 FPS, meaning 0.033s to figure out what to render and render it.
 - Use spatial structures to accelerate searching

Collision Detection



- Two flavors:
 - A priori
 - before the collision occurs
 - calculate the trajectory of each object and put in collision events right before they occur
 - A posteriori
 - after the collision occurs
 - with each advance, see if anything has hit or gotten close
- Both use spatial search structures (octree, k-d tree to identify collisions)

Shaders



Shaders



- Shader: computer program used to do “shading”
- “Shading”: general term that covers more than just shading/lighting
 - Used for many special effects
- Increased control over:
 - position, hue, saturation, brightness, contrast
- For:
 - pixels, vertices, textures

Motivation: Bump Mapping



□ Idea:

- typical rasterization, calculate fragments
- fragments have normals (as per usual)
- also interpolate texture on geometry & fragments
 - use texture for “bumps”
 - take normal for fragment and displace it by “bump” from texture

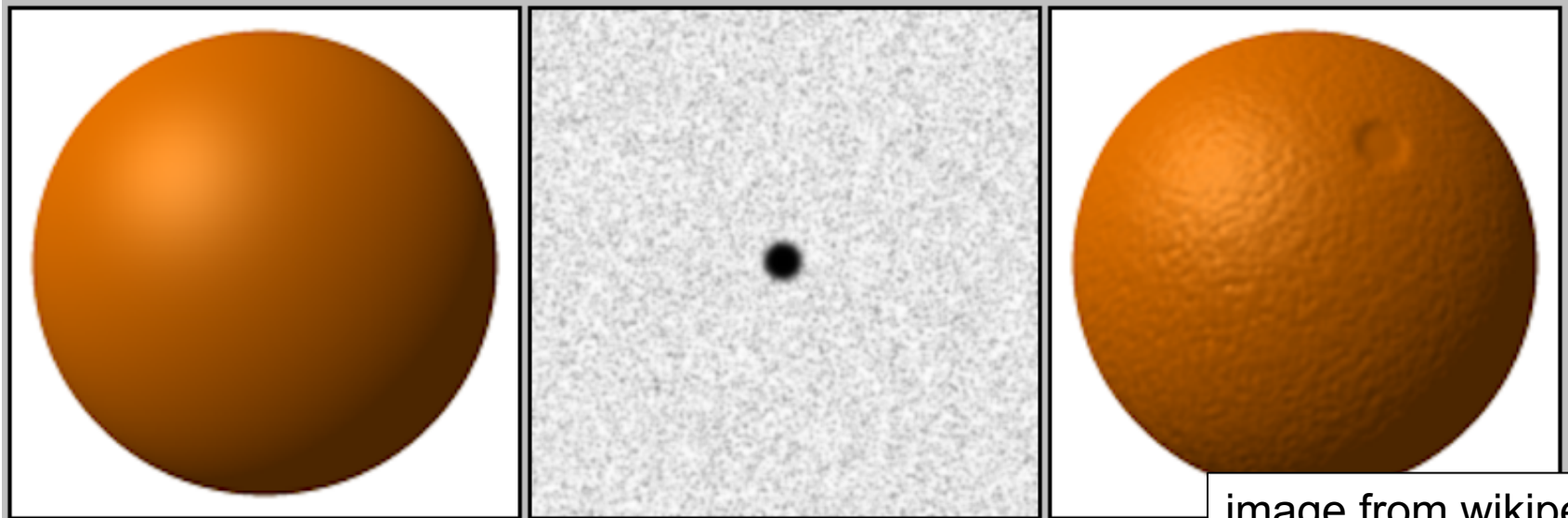


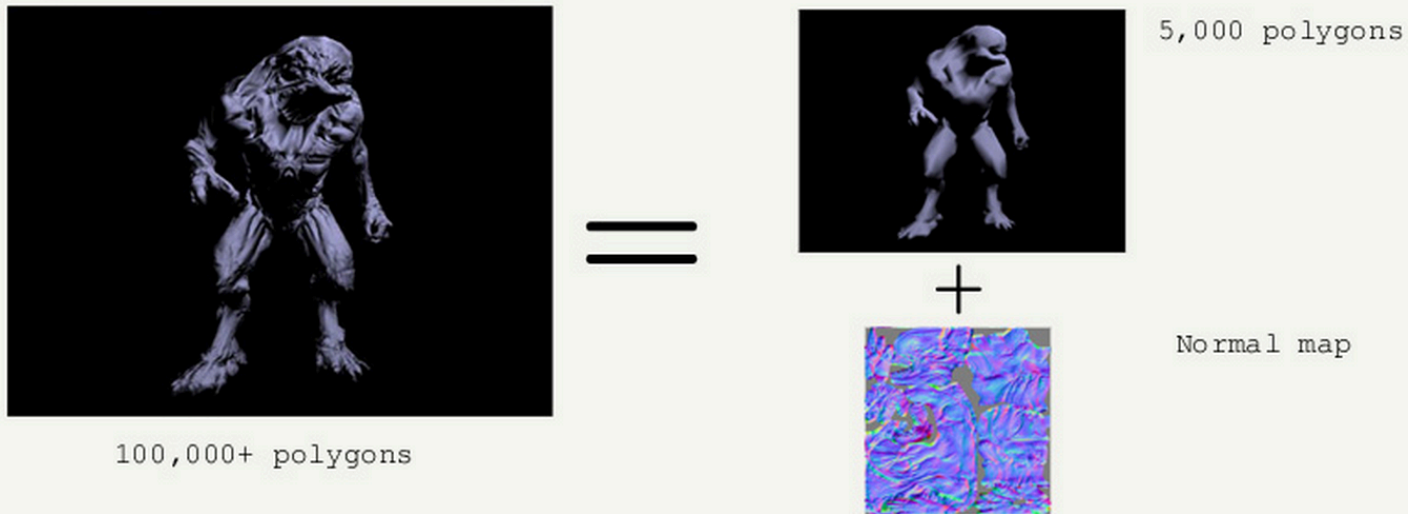
image from wikipedia

Bump Mapping Example



Concept

BumpMapping allows designers to express their creativity through a 100,000+ polygons creature. Once art is done, a low poly model (5000 polygons) is automatically generated along with a normal map.



At runtime, details are added back by combining the low model with the normal map.

Results



How to do Bump Mapping?



- Answer: easy to imagine doing it in your Project 1A-1F infrastructure
 - You have total control
- But what OpenGL commands would do this?
 - Not possible in V1 of the GL interface, which is what we have learned
- It is possible with various extensions to OpenGL
 - We will learn to do this with shaders

Shading Languages



- ❑ shading language: programming language for graphics, specifically shader effects
- ❑ Benefits: increased flexibility with rendering
- ❑ OpenGL (as we know it so far): fixed transformations for color, position, of pixels, vertices, and textures.
- ❑ Shader languages: custom programs, custom effects for color, position of pixels, vertices, and textures.

ARB assembly language



- ARB: low-level shading language
 - at same level as assembly language
- Created by OpenGL Architecture Review Board (ARB)
- Goal: standardize instructions for controlling GPU
- Implemented as a series of extensions to OpenGL
- You don't want to work at this level, but it was an important development in terms of establishing foundation for today's technology

GLSL:



OpenGL Shading Language

- GLSL: high-level shading language
 - also called GLSLang
 - syntax similar to C
- Purpose: increased control of graphics pipeline for developers, but easier than assembly
 - This is layer where developers do things like “bump mapping”
- Benefits:
 - Benefits of GL (cross platform: Windows, Mac, Linux)
 - Support over GPUs (NVIDIA, ATI)
 - HW vendors support GLSL very well

Other high-level shading languages



- Cg (C for Graphics)
 - based on C programming language
 - outputs DirectX or OpenGL shader programs
 - deprecated in 2012
- HLSL (high-level shading language)
 - used with MicroSoft Direct3D
 - analogous to GLSL
 - similar to CG
- RSL (Renderman Shading Language)
 - C-like syntax
 - for use with Renderman: Pixar's rendering engine

Relationship between GLSL and OpenGL



Versions [\[edit\]](#)

GLSL versions have evolved alongside specific versions of the OpenGL API. It is only with OpenGL versions 3.3 and above that the GLSL and OpenGL major and minor version numbers match. These versions for GLSL and OpenGL are related in the following table:

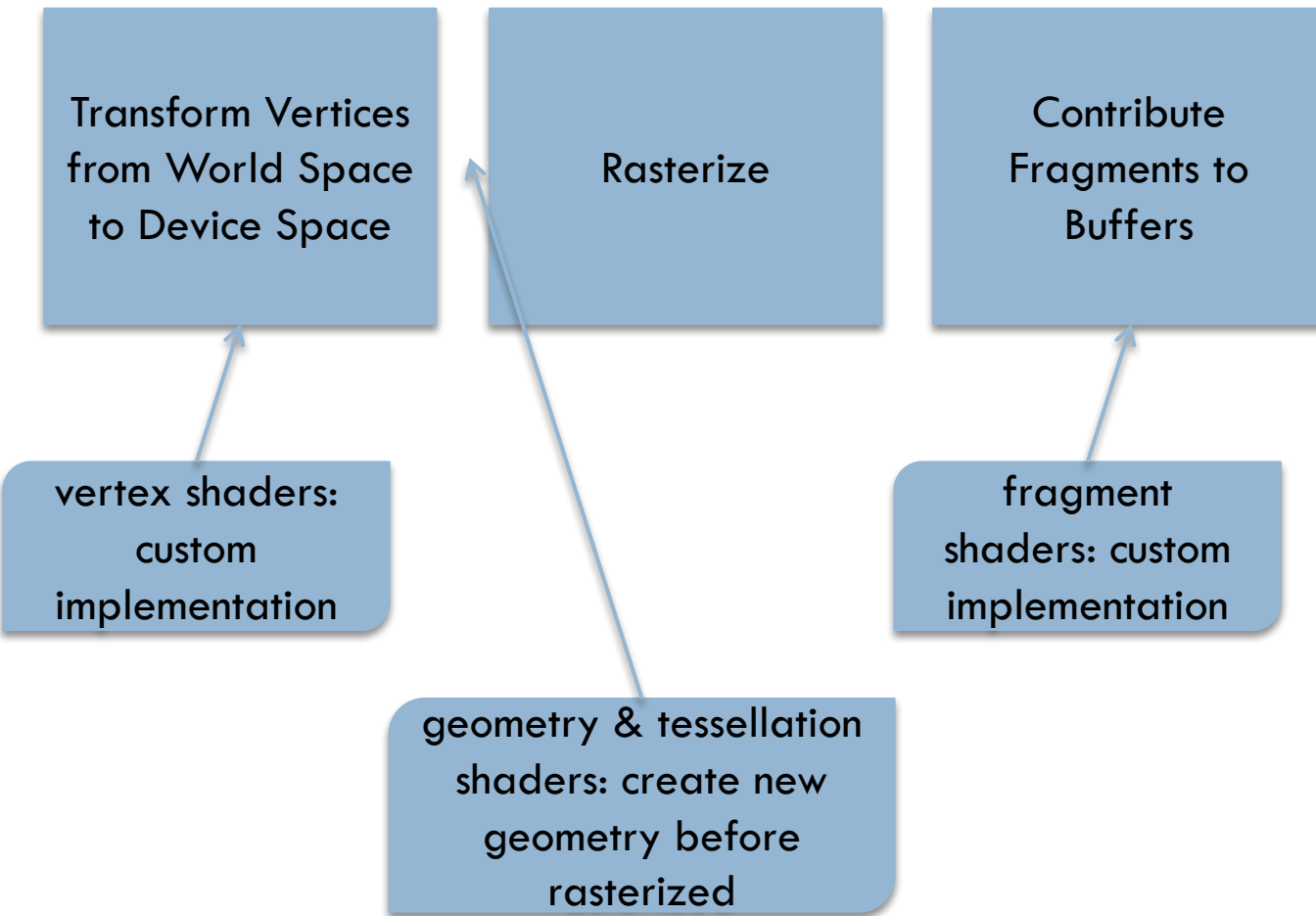
GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59 ^[1]	2.0	April 2004	#version 110
1.20.8 ^[2]	2.1	September 2006	#version 120
1.30.10 ^[3]	3.0	August 2008	#version 130
1.40.08 ^[4]	3.1	March 2009	#version 140
1.50.11 ^[5]	3.2	August 2009	#version 150
3.30.6 ^[6]	3.3	February 2010	#version 330
4.00.9 ^[7]	4.0	March 2010	#version 400
4.10.6 ^[8]	4.1	July 2010	#version 410
4.20.11 ^[9]	4.2	August 2011	#version 420
4.30.8 ^[10]	4.3	August 2012	#version 430
4.40 ^[11]	4.4	July 2013	#version 440
4.50 ^[12]	4.5	August 2014	#version 450

4 Types of Shaders



- ❑ Vertex Shaders
 - ❑ Fragment Shaders
 - ❑ Geometry Shaders
 - ❑ Tessellation Shaders
-
- ❑ It is common to use multiple types of shaders in a program and have them interact.

How Shaders Fit Into the Graphics Pipeline



- You can 0 or 1 of each shader type
- Vertex & fragment: very common
- Geometry & tessellation: less common
 - adaptive meshing

Vertex Shader



- Run once for each vertex
- Can: manipulate position, color, texture
- Cannot: create new vertices
- Primary purpose: transform from world-space to device-space (+ depth for z-buffer).
 - However: A vertex shader replaces the transformation, texture coordinate generation and lighting parts of OpenGL, and it also adds texture access at the vertex level
- Output goes to geometry shader or rasterizer

Geometry Shader



- ❑ Run once for each geometry primitive
- ❑ Purpose: create new geometry from existing geometry.
- ❑ Output goes to rasterizer
- ❑ Examples: glyphing, mesh complexity modification
- ❑ Formally available in GL 3.2, but previously available in 2.0+ with extensions
- ❑ Tessellation Shader: doing some of the same things
- ❑ Available in GL 4.0



Fragment Shader

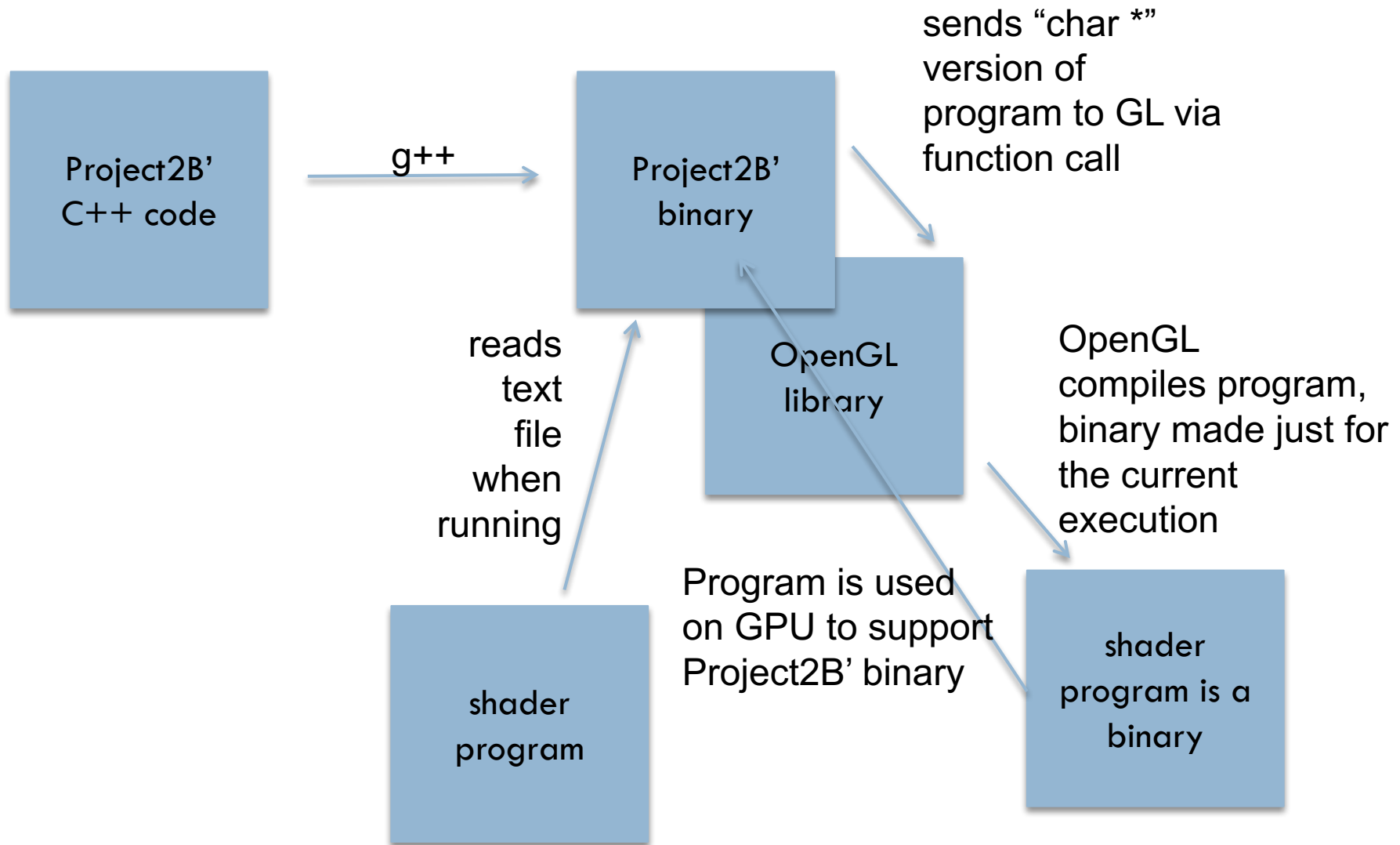
- ❑ Run once for each fragment
- ❑ Purpose: replaces the OpenGL 1.4 fixed-function texturing, color sum and fog stages
- ❑ Output goes to buffers
- ❑ Example usages: bump mapping, shadows, specular highlights
- ❑ Can be very complicated: can sample surrounding pixels and use their values (blur, edge detection)
- ❑ Also called pixel shaders

How to Use Shaders



- ❑ You write a shader program: a tiny C-like program
- ❑ You write C/C++ code for your application
- ❑ Your application loads the shader program from a text file
- ❑ Your application sends the shader program to the OpenGL library and directs the OpenGL library to compile the shader program
- ❑ If successful, the resulting GPU code can be attached to your (running) application and used
- ❑ It will then supplant the built-in GL operations

How to Use Shaders: Visual Version





Compiling Shader

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);
```

Compiling Shader: inspect if it works



```
if(isCompiledVS == GL_FALSE)
{
    cerr << "Did not compile VS" << endl;

    GLint maxLength = 0;
    glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

    // The maxLength includes the NULL character
    std::vector<GLchar> errorLog(maxLength);
    glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
    cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
    exit(EXIT_FAILURE);
}
```

Compiling Multiple Shaders



```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);

if(isCompiledVS == GL_FALSE)
{
    cerr << "Did not compile VS" << endl;

    GLint maxLength = 0;
    glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

    // The maxLength includes the NULL character
    std::vector<GLchar> errorLog(maxLength);
    glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
    cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
    exit(EXIT_FAILURE);
}

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
std::string fragmentProgram = loadFileToString("fs.glsl");
const char *fragment_shader_source = fragmentProgram.c_str();
GLint const fragment_shader_length = strlen(fragment_shader_source);
glShaderSource(fragmentShader, 1, &fragment_shader_source, &fragment_shader_length);
glCompileShader(fragmentShader);
GLint isCompiledFS = 0;
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &isCompiledFS);
```

Attaching Shaders to a Program



```
GLuint program = glCreateProgram();  
glAttachShader(program, vertexShader);  
glAttachShader(program, fragmentShader);  
  
glLinkProgram(program);  
  
glDetachShader(program, vertexShader);  
glDetachShader(program, fragmentShader);
```

Inspecting if program link worked...



```
GLint isLinked = 0;
glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
if(isLinked == GL_FALSE)
{
    GLint maxLength = 0;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

    //The maxLength includes the NULL character
    std::vector<GLchar> infoLog(maxLength);
    glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
    cerr << "Couldn't link" << endl;
    cerr << "Log says " << &(infoLog[0]) << endl;

    exit(EXIT_FAILURE);
}
```

BUT: this doesn't work in VTK...



- VTK has its own shader handling, and it doesn't play well with the GL calls above...

```
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();  
pgm->SetContext(renWin);
```

```
vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();  
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);  
std::string vertexProgram = loadFileToString("v_vs.glsl");  
vertexShader->SetSourceCode(vertexProgram.c_str());  
vertexShader->SetContext(pgm->GetContext());
```

```
pgm->GetShaders()->AddItem(vertexShader);
```

```
vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();  
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);  
std::string fragmentProgram = loadFileToString("v_fs.glsl");  
fragmentShader->SetSourceCode(fragmentProgram.c_str());  
fragmentShader->SetContext(pgm->GetContext());
```

```
pgm->GetShaders()->AddItem(fragmentShader);
```

```
((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
```

note: VTK6.1 much better for shaders than 6.0

Simplest Vertex Shader



```
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Many built-in variables.

Some are input.

Some are required output (gl_Position).

Simplest Vertex Shader (VTK version)



```
void propFuncVS(void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

VTK uses special names

propFuncVS: vertex shader

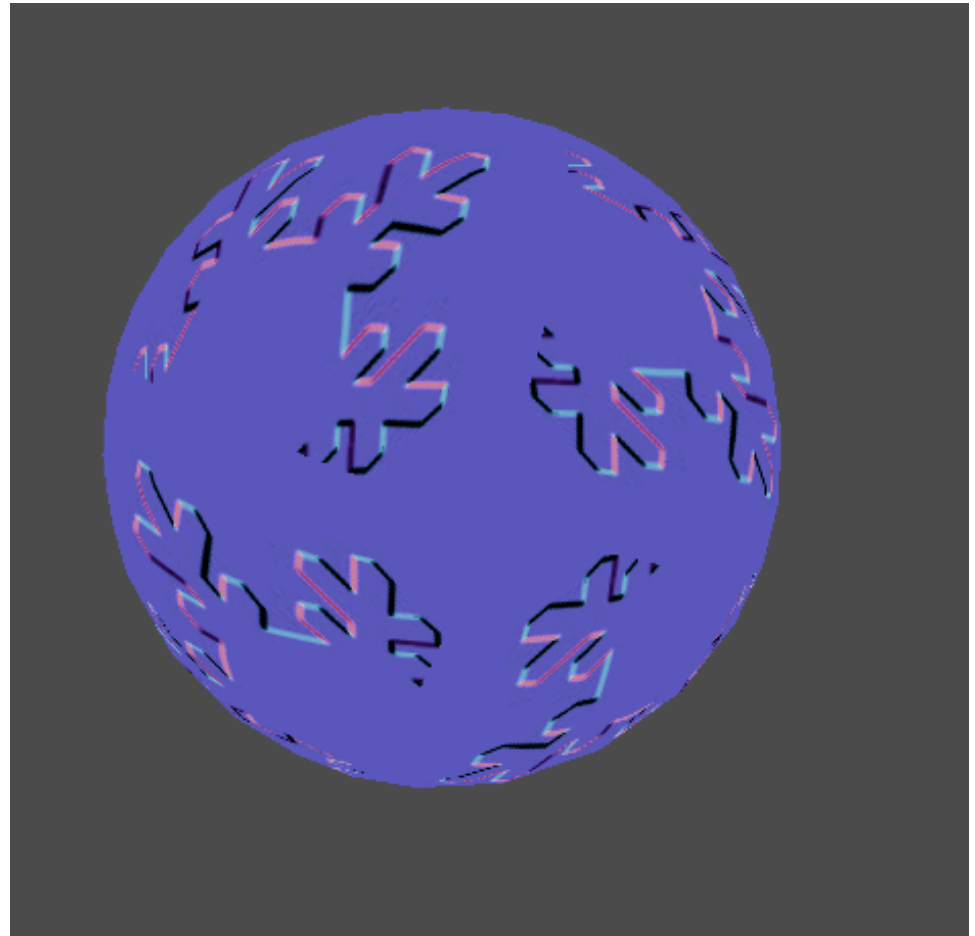
propFuncFS: fragment shader

somehow it changes these into “main” just in time...

Bump-mapping with GLSL



bump map texture



output



Will need to load a texture...

```
// from swiftless.com
GLuint LoadTexture( const char * filename, int width, int height )
{
    GLuint texture;
    unsigned char * data;
    FILE * file;

    //The following code will read in our RAW file
    file = fopen( filename, "rb" );

    if ( file == NULL ) return 0;
    data = (unsigned char *)malloc( width * height * 3 );
    fread( data, width * height * 3, 1, file );

    fclose( file );

    glGenTextures( 1, &texture ); //generate the texture with the loaded data
    glBindTexture( GL_TEXTURE_2D, texture ); //bind the texture to it's array

    glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE ); //set texture environment parameters

    //And if you go and use extensions, you can use Anisotropic filtering textures which are of an
    //even better quality, but this will do for now.
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );

    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );

    //Here we are setting the parameter to repeat the texture instead of clamping the texture
    //to the edge of our shape.
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    //Generate the texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

    free( data ); //free the texture

    return texture; //return whether it was successfull
}
```

Need to put 2D textures on our triangles...



```
class Triangle
{
public:
    double      X[3];
    double      Y[3];
    double      Z[3];
    double      Tu[3];
    double      Tv[3];
};
```

```
void DrawSphere()
{
    int recursionLevel = 3;
    Triangle t;
    t.X[0] = 1;
    t.Y[0] = 0;
    t.Z[0] = 0;
    t.Tu[0] = 0;
    t.Tv[0] = 0;
    t.X[1] = 0;
    t.Y[1] = 1;
    t.Z[1] = 0;
    t.Tu[1] = 1;
    t.Tv[1] = 0;
    t.X[2] = 0;
    t.Y[2] = 0;
    t.Z[2] = 1;
    t.Tu[2] = 1;
    t.Tv[2] = 1;
    std::vector<Triangle> list;
    list.push_back(t);
    for (int r = 0 ; r < recursionLevel ; r++)
    {
        list = SplitTriangle(list);
    }
}
```

```
std::vector<Triangle> SplitTriangle(std::vector<Triangle> &list)
{
    std::vector<Triangle> output(4*list.size());
    for (unsigned int i = 0 ; i < list.size() ; i++)
    {
        double mid1[5], mid2[5], mid3[5];
        mid1[0] = (list[i].X[0]+list[i].X[1])/2;
        mid1[1] = (list[i].Y[0]+list[i].Y[1])/2;
        mid1[2] = (list[i].Z[0]+list[i].Z[1])/2;
        mid1[3] = (list[i].Tu[0]+list[i].Tu[1])/2;
        mid1[4] = (list[i].Tv[0]+list[i].Tv[1])/2;
        mid2[0] = (list[i].X[1]+list[i].X[2])/2;
        mid2[1] = (list[i].Y[1]+list[i].Y[2])/2;
        mid2[2] = (list[i].Z[1]+list[i].Z[2])/2;
        mid2[3] = (list[i].Tu[1]+list[i].Tu[2])/2;
        mid2[4] = (list[i].Tv[1]+list[i].Tv[2])/2;
        mid3[0] = (list[i].X[0]+list[i].X[2])/2;
        mid3[1] = (list[i].Y[0]+list[i].Y[2])/2;
        mid3[2] = (list[i].Z[0]+list[i].Z[2])/2;
        mid3[3] = (list[i].Tu[0]+list[i].Tu[2])/2;
        mid3[4] = (list[i].Tv[0]+list[i].Tv[2])/2;
        output[4*i+0].X[0] = list[i].X[0];
        output[4*i+0].Y[0] = list[i].Y[0];
        output[4*i+0].Z[0] = list[i].Z[0];
        output[4*i+0].Tu[0] = list[i].Tu[0];
        output[4*i+0].Tv[0] = list[i].Tv[0];
        output[4*i+0].X[1] = mid1[0];
        output[4*i+0].Y[1] = mid1[1];
        output[4*i+0].Z[1] = mid1[2];
        output[4*i+0].Tu[1] = mid1[3];
        output[4*i+0].Tv[1] = mid1[4];
        output[4*i+0].X[2] = mid3[0];
        output[4*i+0].Y[2] = mid3[1];
        output[4*i+0].Z[2] = mid3[2];
        output[4*i+0].Tu[2] = mid3[3];
        output[4*i+0].Tv[2] = mid3[4];
        output[4*i+1].X[0] = list[i].X[1];
        output[4*i+1].Y[0] = list[i].Y[1];
        output[4*i+1].Z[0] = list[i].Z[1];
        output[4*i+1].Tu[0] = list[i].Tu[1];
        output[4*i+1].Tv[0] = list[i].Tv[1];
    }
}
```

Need to set up shaders and textures...



```
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();
pgm->SetContext(renWin);
```

```
vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);
//std::string vertexProgram = loadFileToString("vs.glsl");
std::string vertexProgram = loadFileToString("v_vs.glsl");
vertexShader->SetSourceCode(vertexProgram.c_str());
vertexShader->SetContext(pgm->GetContext());
```

```
pgm->GetShaders()->AddItem(vertexShader);
```

```
vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);
//std::string fragmentProgram = loadFileToString("light_fs.glsl");
std::string fragmentProgram = loadFileToString("v_fs.glsl");
fragmentShader->SetSourceCode(fragmentProgram.c_str());
fragmentShader->SetContext(pgm->GetContext());
```

```
pgm->GetShaders()->AddItem(fragmentShader);
```

```
((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
win3Actor->GetProperty()->ShadingOn();
```

```
GLuint texture = LoadTexture("normal_map.raw", 256, 256);
glEnable(GL_TEXTURE_2D);
int texture_location = glGetUniformLocation(fragmentShader->GetId(), "color_texture");
glUniform1i(texture_location, 0);
glBindTexture(GL_TEXTURE_2D, texture);
```

So what is the vertex shader program?...



```
void propFuncVS(void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;

    // Set the position of the current vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

And what is the fragment shader program?...



```
uniform sampler2D color_texture;
uniform sampler2D normal_texture;

void propFuncFS(void)
{
    // Extract the normal from the normal map
    vec3 normal = normalize(texture2D(normal_texture, gl_TexCoord[0].st).rgb * 2.0 - 1.0);

    // Determine where the light is positioned (this can be set however you like)
    vec3 light_pos = normalize(vec3(1.0, 1.0, 1.5));

    // Calculate the lighting diffuse value
    float diffuse = max(dot(normal, light_pos), 0.0);

    vec3 color = diffuse * texture2D(color_texture, gl_TexCoord[0].st).rgb;

    // Set the output color of our current pixel
    gl_FragColor = vec4(color, 1.0);
}
```

Rotations



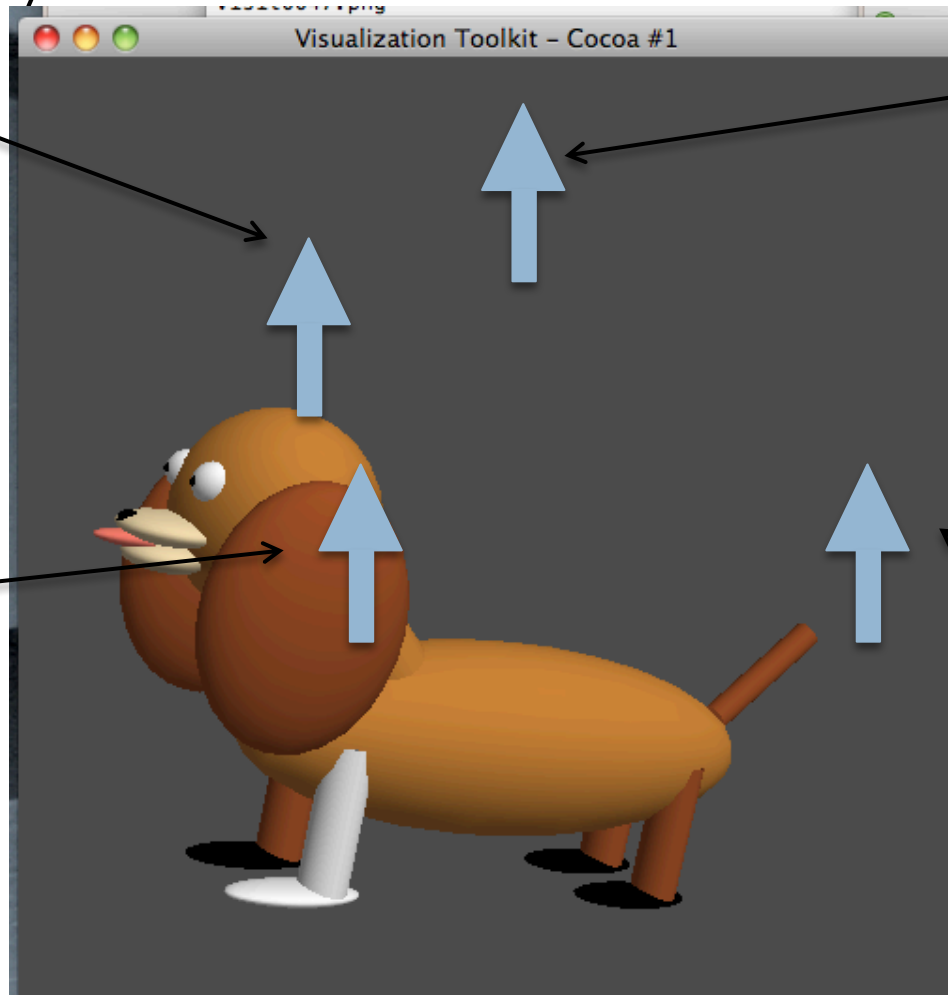


Improved rotations

□ Project 2A/2B:

Click here
→
Medium rotations
around
combination of up
& “up cross view”
for as long as you
hold button down.

Click here
→
Small rotations
around up axis
for as long as
you hold button
down.



Click here
→
Large rotations
around “up
cross view” for
as long as you
hold button
down.

Click here
→
Large rotations
around up axis
for as long as
you hold button
down.

Improved rotations: trackball



Only rotates
while trackball is
spun

Improved rotations: trackball



- ❑ Idea: approximate trackball interface with traditional mouse interface
- ❑ Camera movement occurs when the mouse is moving
- ❑ The camera does not move when the button is clicked, but the mouse does not move

Improved rotations: trackball

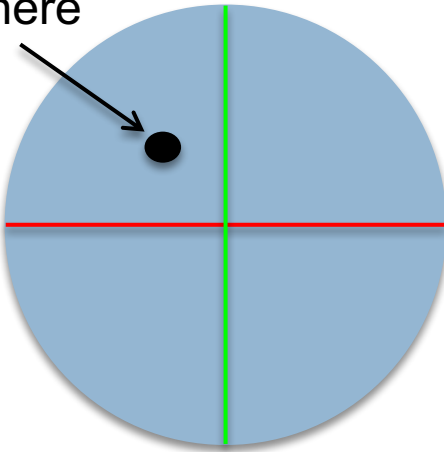


- Imagine your scene is contained within a sphere.
- When you push the mouse button, the cursor is over a pixel and the ray corresponding to that pixel intersects the sphere.
- Idea: every subsequent mouse movement (while the button is pushed) should rotate the intersection of the sphere so that it is still under the cursor.

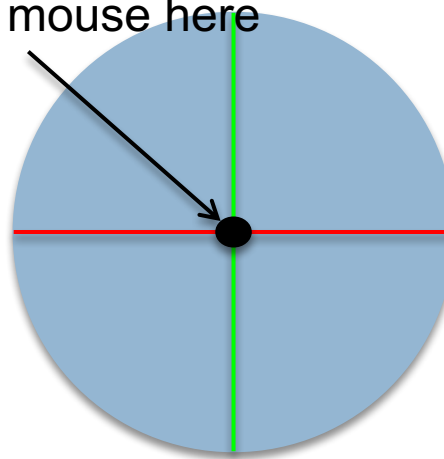
Improved rotations: trackball



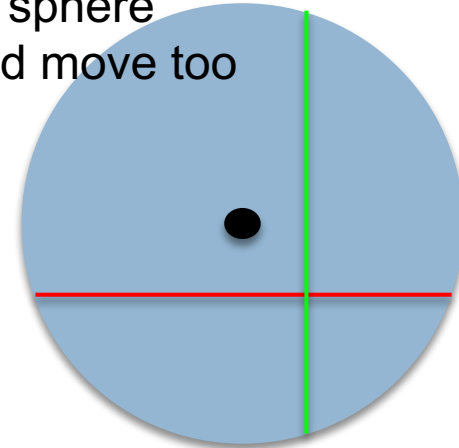
Click here



Move mouse here



Then sphere
should move too



How to do the rotations?



- Best way: use quaternions
 - Number system that extends complex numbers
 - Applies to mechanics to 3D space
 - Would be a very long lecture!
- Simple way:
 - Take “dx” and “dy” in pixels, and then do
 - RotateAroundUp(dy*factor);
 - RotateAroundUpCrossView(dx*factor);
 - Factors vary based on level of zoom
 - Can create weird effects based on order of rotations
 - Users rarely notice in practice

Parallel Renderin





Large Scale Visualization with Cluster Computing

Linux Cluster Institute Workshop

October 1, 2004

**Kenneth Moreland
Sandia National Laboratories**



Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy's National Nuclear Security Administration
under contract DE-AC04-94AL85000.



The Graphics Pipeline

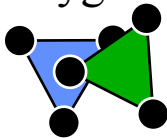
Points



Lines



Polygons



Rendering Hardware

Geometric Processing

Translation
Lighting
Clipping

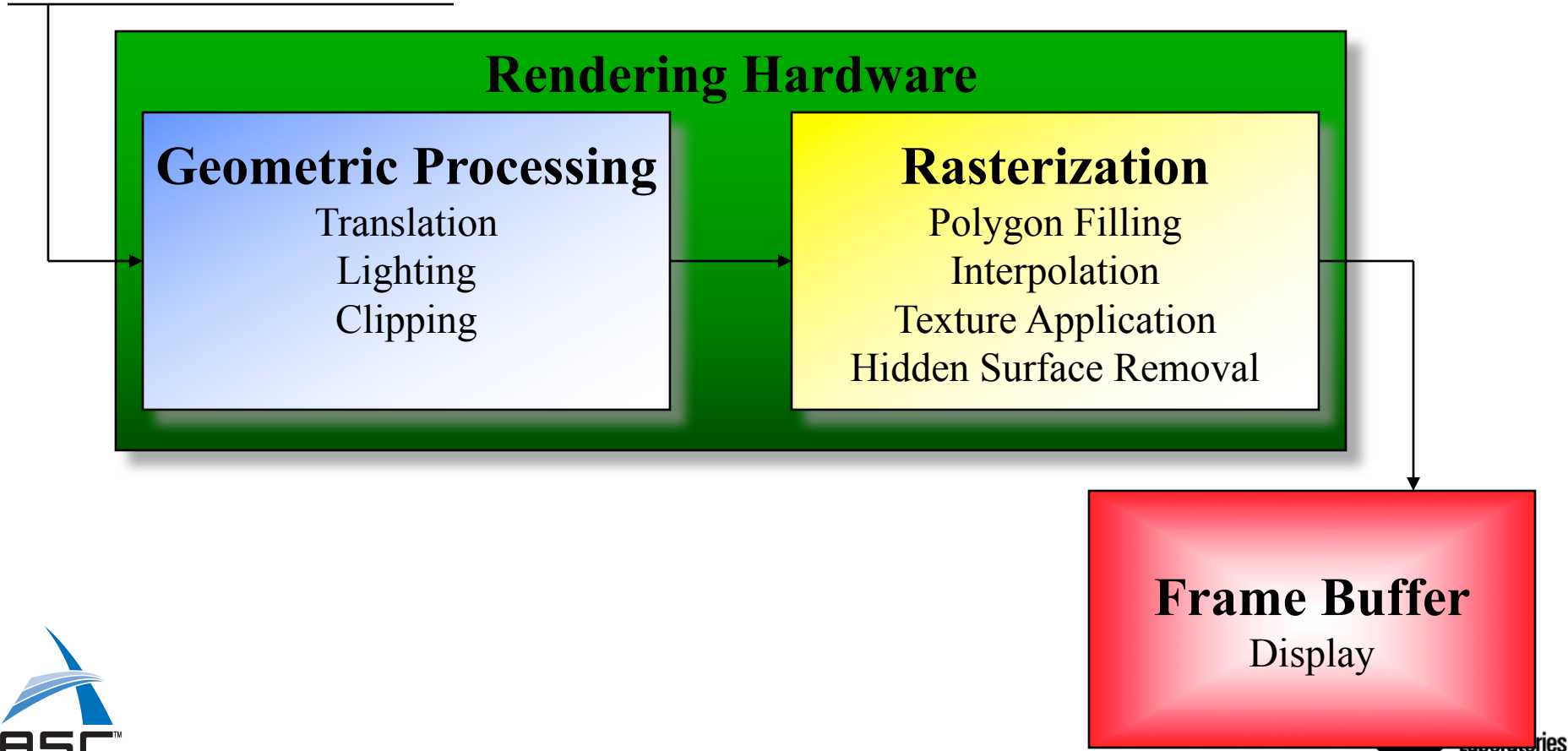
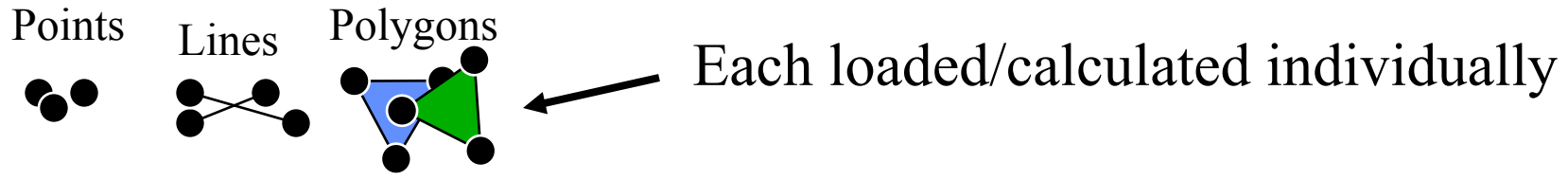
Rasterization

Polygon Filling
Interpolation
Texture Application
Hidden Surface Removal

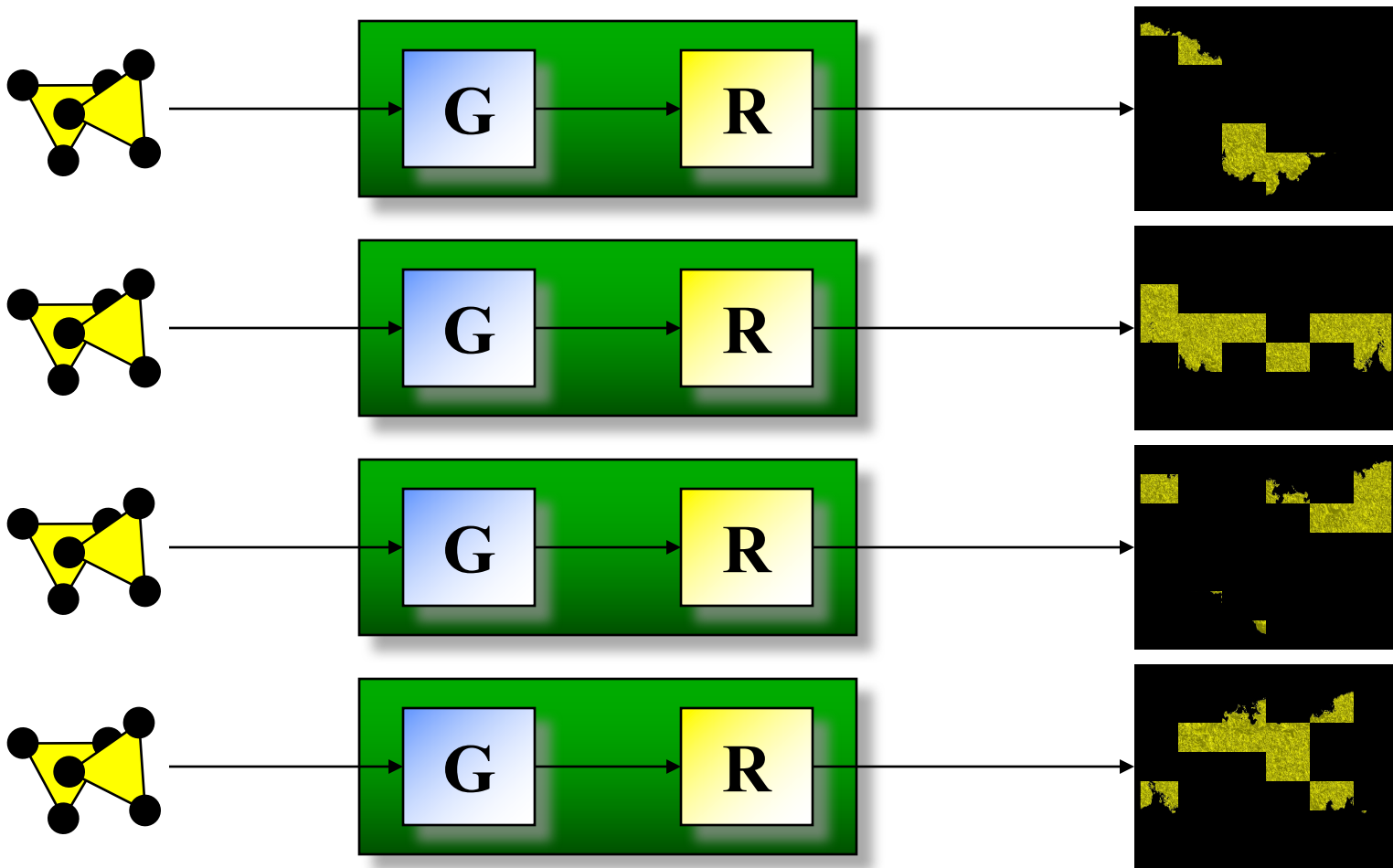
Frame Buffer

Display

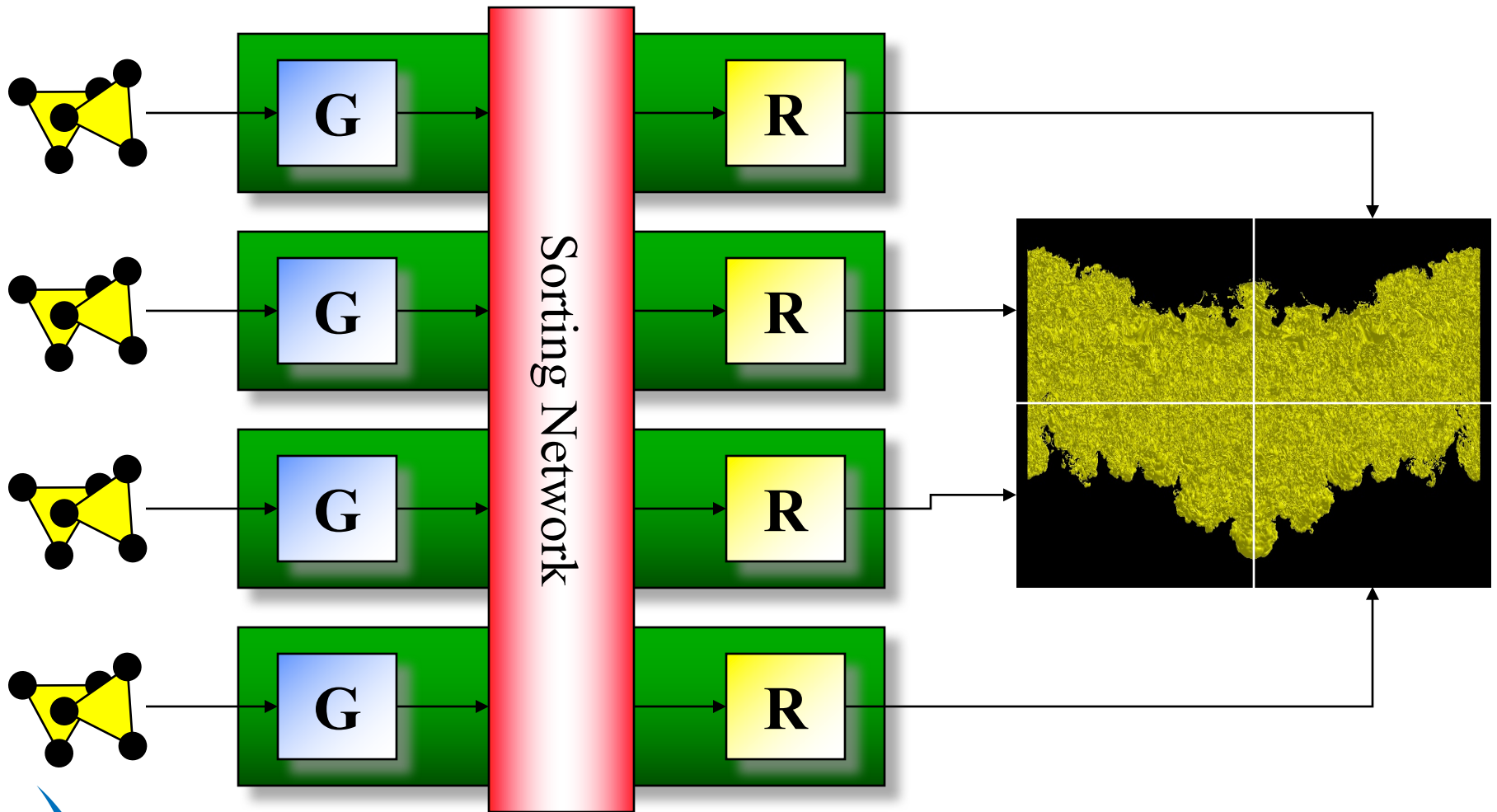
Parallel Graphics Pipelines



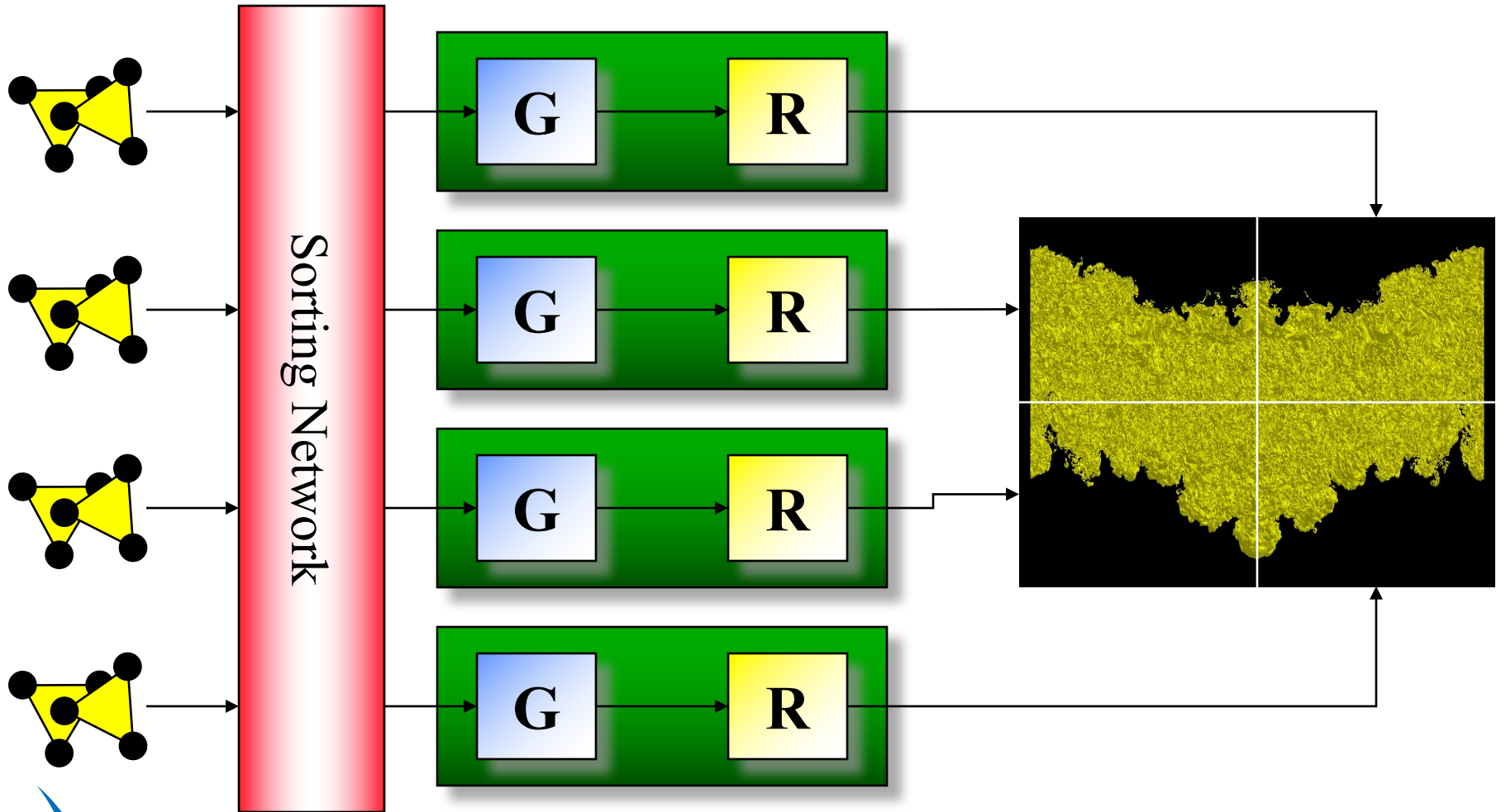
Parallel Graphics Pipelines



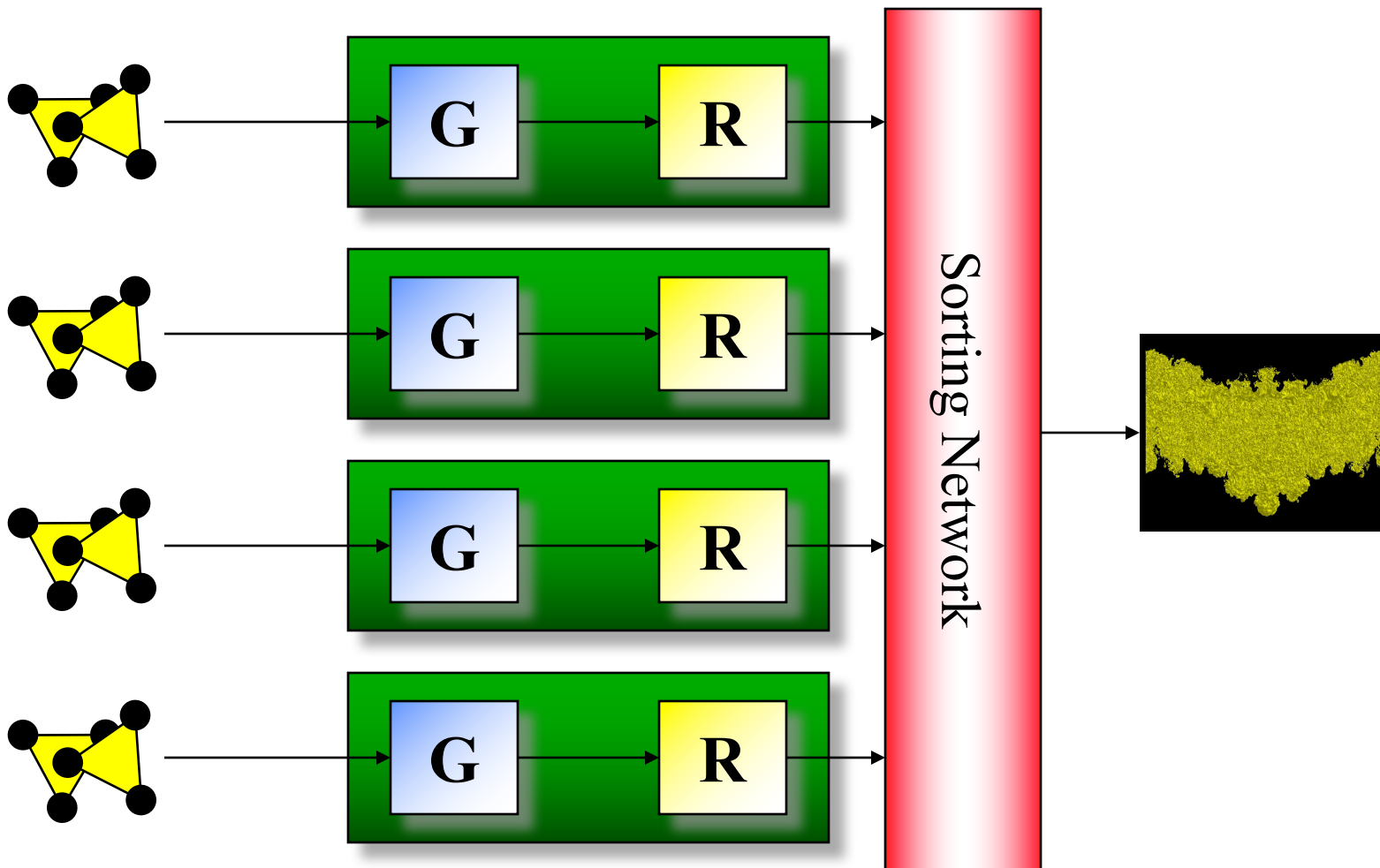
Sort Middle Parallel Rendering



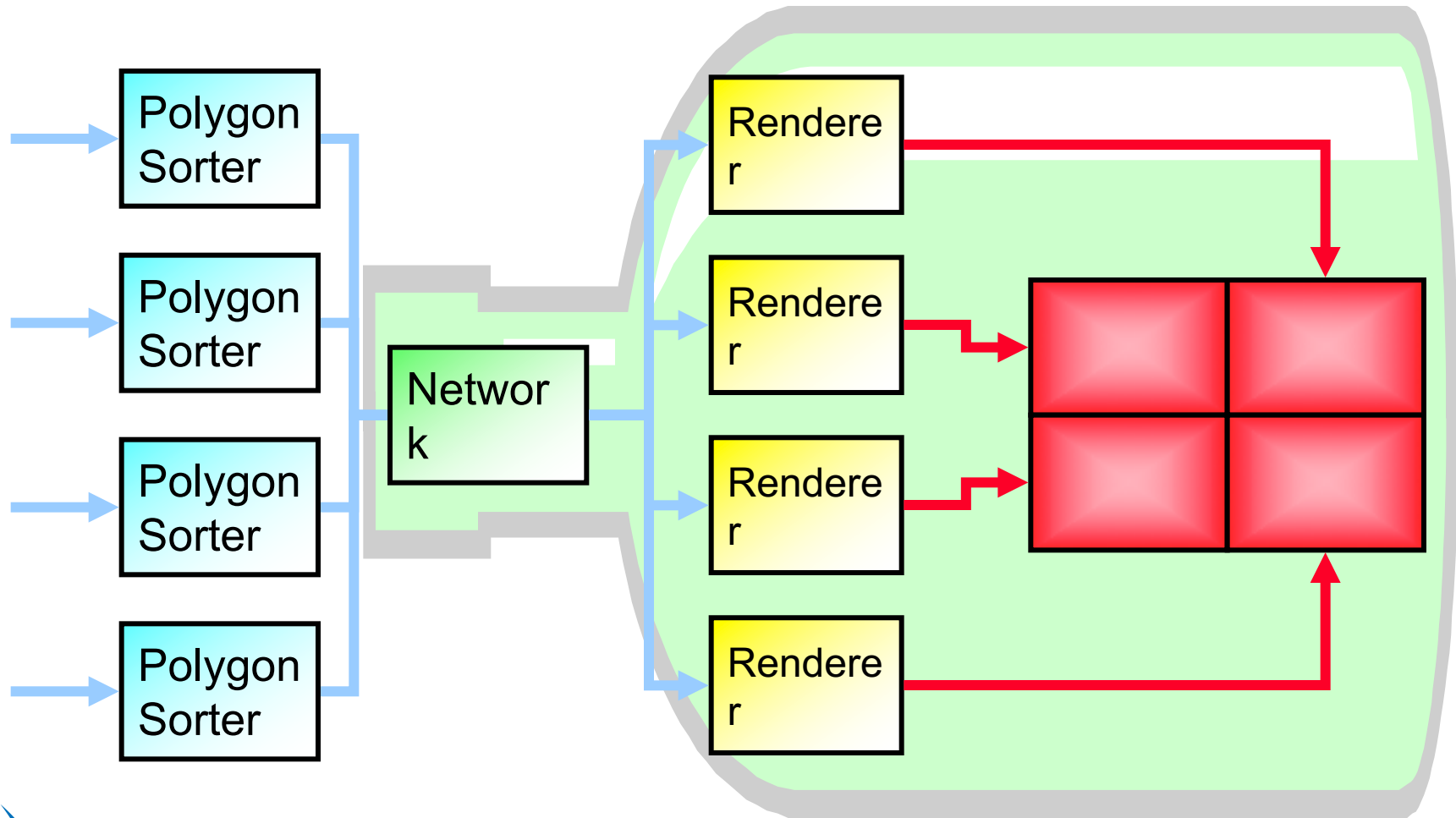
Sort First Parallel Rendering



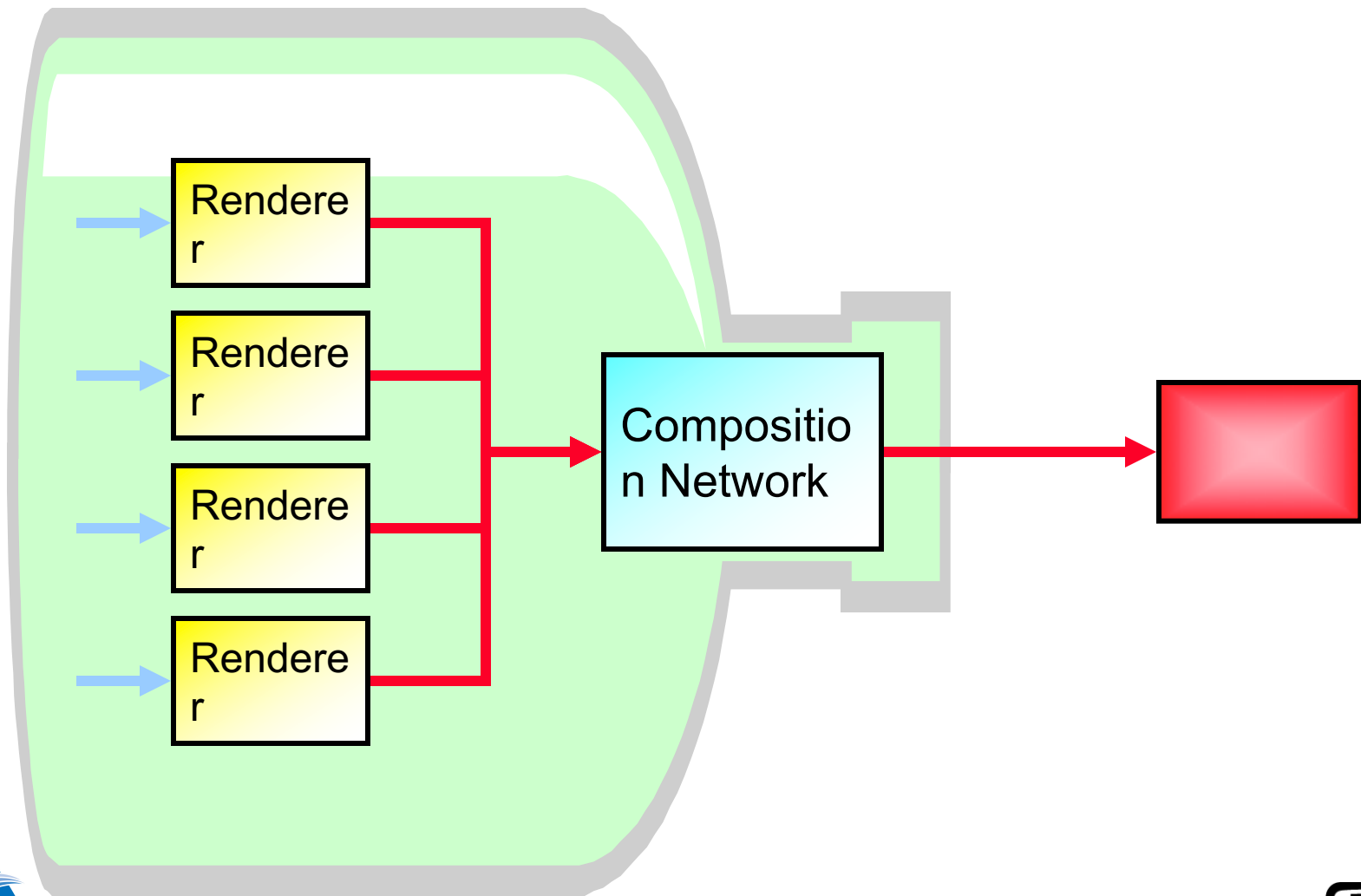
Sort Last Parallel Rendering



Sort-First Bottleneck



Sort-Last Bottleneck



Clockwise / Counter-Clockwise



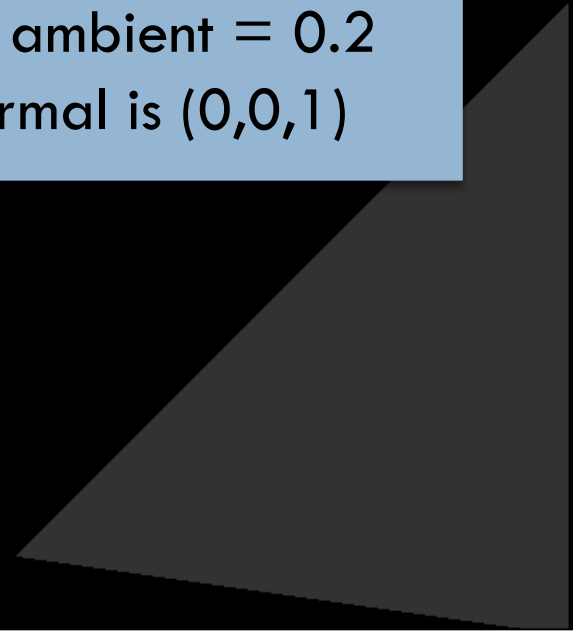
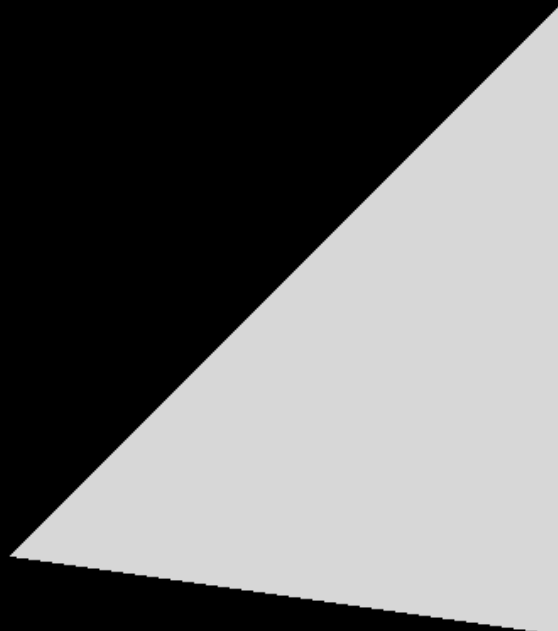


What is going on here?

Details:

Diffuse = 0.8, ambient = 0.2

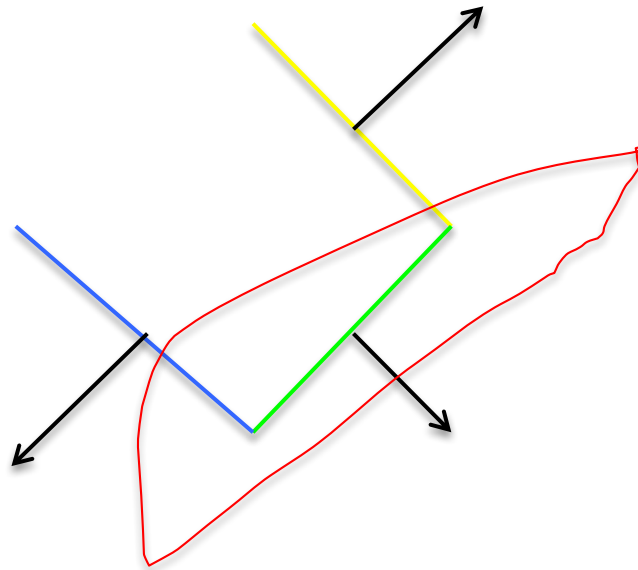
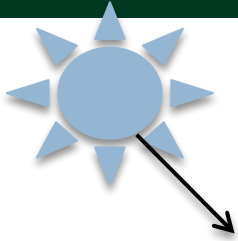
Default GL normal is (0,0,1)



```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_TRIANGLES);
    glVertex3f(-10, -10, -10);
    glVertex3f(10, -10, 10);
    glVertex3f(10, 10, 10);
    glEnd();
}
```

```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_TRIANGLES);
    glVertex3f(-10, -10, -10);
    glVertex3f(10, 10, 10);
    glVertex3f(10, -10, 10);
    glEnd();
}
```

But wait...



If you have an open surface, then there is a “back face”. The back face has the opposite normal.

How can we deal with this case?

Idea #1: encode all triangles twice, with different normals

Idea #2: modify diffuse lighting model

This is called two-sided lighting

Diffuse light = $\text{abs}(\mathbf{L} \cdot \mathbf{N})$

Reminder: open surface, closed surface



- Closed surface:

- you could drop it in water and it would float (i.e., no way for water to get inside)
- no way to see the inside

- Open surface:

- water can get to any part of the surface
- you can see the inside

Front face / back face



- Front face: the face that is “facing outward”
- Back face: the face that is “facing inward”
- These distinctions are meaningful for closed surfaces
- They are not meaningful for open surfaces
 - This is why we did “two-sided lighting”

What is going on here?



Answer:

OpenGL is giving us diffuse+ambient on the left, but only ambient on the right (rather, diffuse == 0).

Why? ... it is interpreting the left side as a “front face” and the right side as a “back face”.

(if we spun the camera, the left would be gray, and the right would be white)

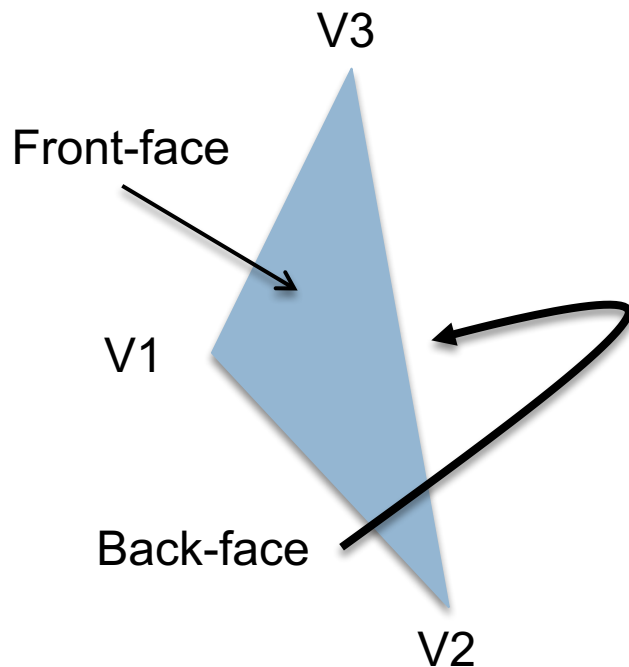
```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_TRIANGLES);
    glVertex3f(-10, -10, -10);
    glVertex3f(10, -10, 10);
    glVertex3f(10, 10, 10);
    glEnd();
}
```

```
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_TRIANGLES);
    glVertex3f(-10, -10, -10);
    glVertex3f(10, 10, 10);
    glVertex3f(10, -10, 10);
    glEnd();
}
```

Determining Front Face and Back Face



- The front face and back face is determined by convention
- Convention #1:

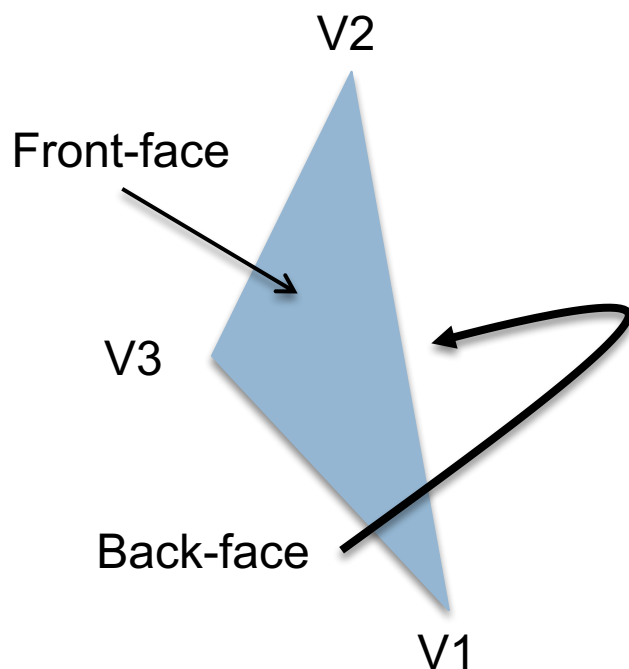


V1, V2, and V3 are arranged counter-clockwise around the front-face

Determining Front Face and Back Face



- The front face and back face is determined by convention
- Convention #1:

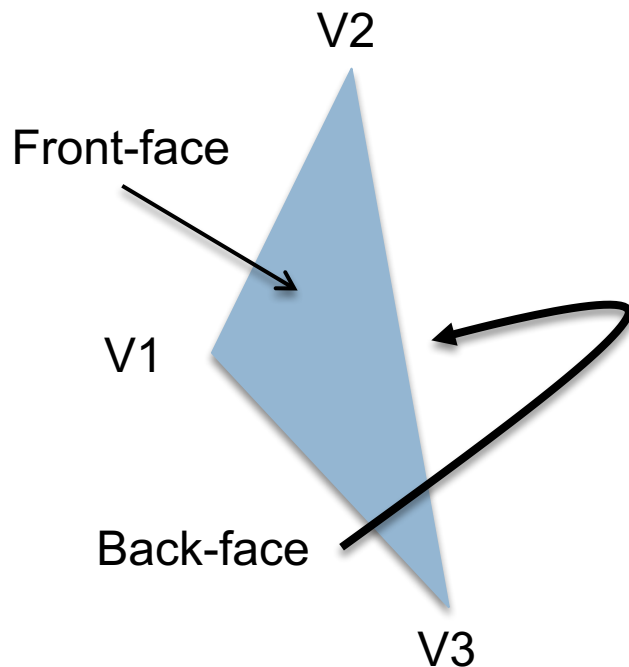


Which vertex is specified first doesn't matter, as long as they are arranged counter-clockwise around the front-face

Determining Front Face and Back Face



- The front face and back face is determined by convention
- Convention #2:

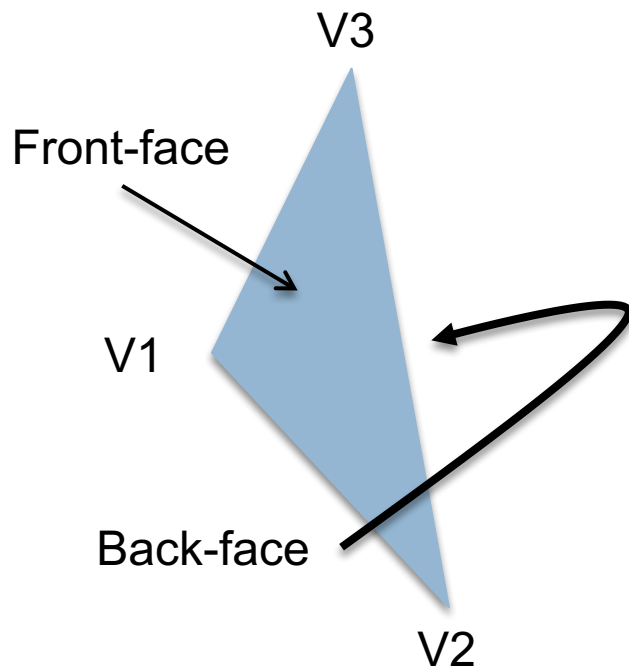


V1, V2, and V3 are arranged clockwise around the front-face

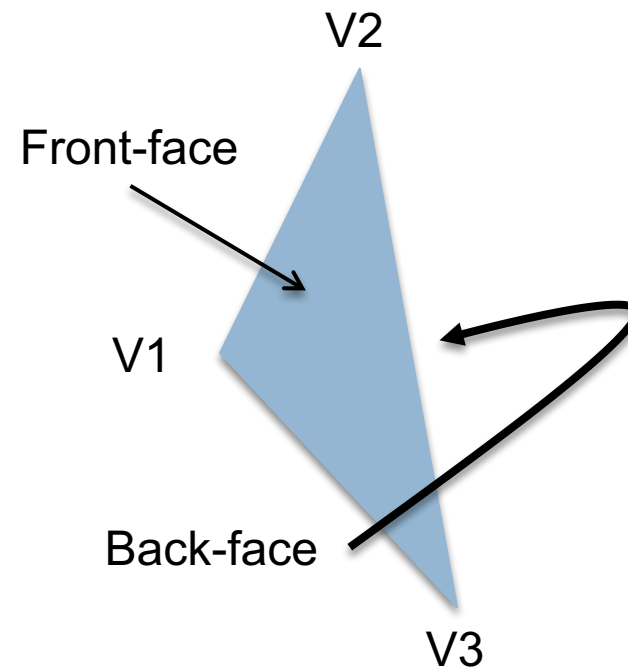
Determining Front Face and Back Face



- The front face and back face is determined by convention
- Convention #1:



Convention #2:



glFrontFace



Name

glFrontFace — define front- and back-facing polygons

C Specification

```
void glFrontFace(GLenum mode );
```

Python Specification

```
glFrontFace(mode ) → None
```

Parameters

mode

Specifies the orientation of front-facing polygons. `GL_CW` and `GL_CCW` are accepted. The initial value is `GL_CCW`.

Description

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call [glEnable](#) and [glDisable](#) with argument `GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `glFrontFace` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `GL_CCW` to *mode* selects counterclockwise polygons as front-facing; `GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

Why front face / back face is important



- Reason #1:
 - Need to know this if you are doing lighting calculations
- Reason #2:
 - Culling

Culling



- ❑ Idea: some triangles can't affect the picture, so don't render them.
- ❑ Question: how can this happen?
- ❑ Answer #1: geometry outside [0->width, 0->height] in device space
- ❑ Answer #2: closed surface, and the back face is facing the camera

`glCullFace` — specify whether front- or back-facing facets can be culled

C Specification

```
void glCullFace(GLenum mode);
```

Parameters

mode

Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants **GL_FRONT**, **GL_BACK**, and **GL_FRONT_AND_BACK** are accepted. The initial value is **GL_BACK**.

Description

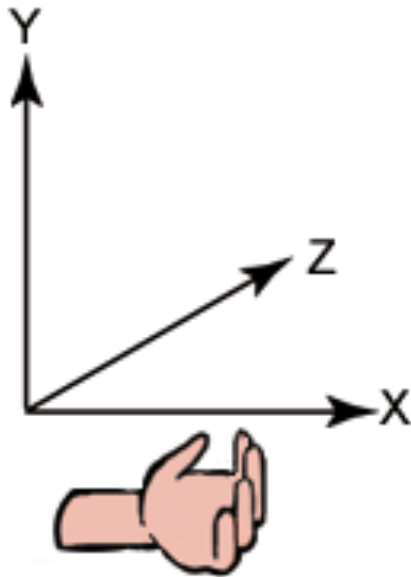
glCullFace specifies whether front- or back-facing facets are culled (as specified by *mode*) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the **glEnable** and **glDisable** commands with the argument **GL_CULL_FACE**. Facets include triangles, quadrilaterals, polygons, and rectangles.

glFrontFace specifies which of the clockwise and counterclockwise facets are front-facing and back-facing. See **glFrontFace**.

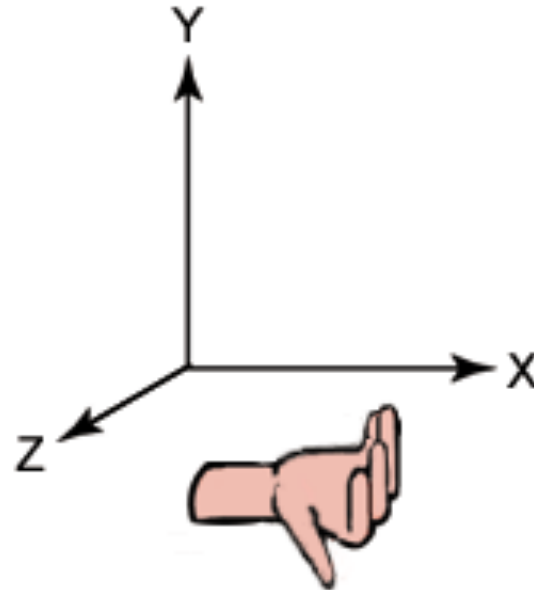
Something you should know about: Left- and right-handed coordinates



Left-handed
Cartesian Coordinates



Right-handed
Cartesian Coordinates



OpenGL: right-handed
DirectX: left-handed