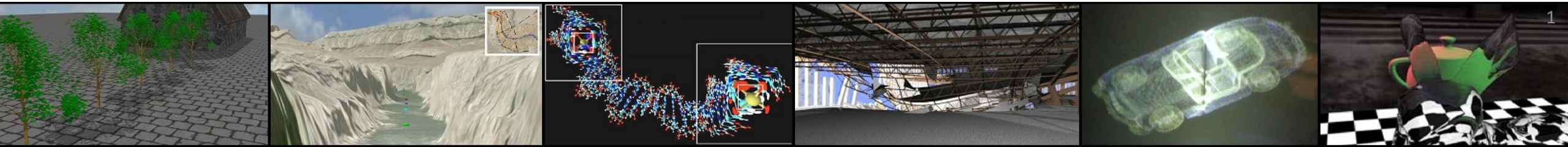
CIS 4930-001: Introduction to Augmented and Virtual Reality

Scene Graph and Acceleration Strategies

Paul Rosen
Assistant Professor
University of South Florida

Some slides from: Anders Backman, Mark Billinghurst, Doug Bowman, David Johnson, Gun Lee, Ivan Poupyrev, Bruce Thomas, Geb Thomas, Anna Yershova, Stefanie Zollman



Motivation

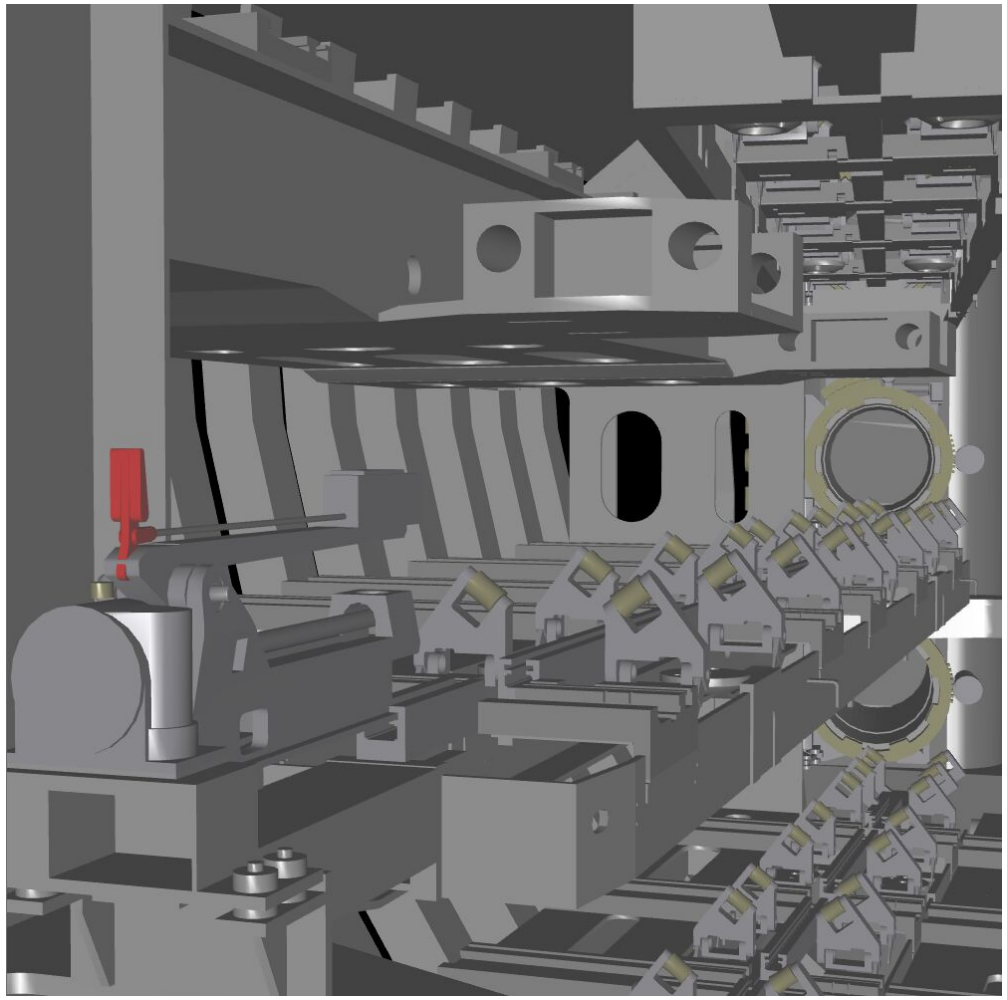
VR scenes need complexity

- Immersion
- People don't bother with VR for easy problems
  - lots of geometry

How do we organize objects?

How do we maintain speed?

- Wait for the graphics cards?



Limitations of Immediate Mode Graphics

When we define a geometric object in an application

Object is passed through the pipeline

To redraw the object, either changed or the same, we must re-execute the code

At the end of the day, we manage objects directly

The Scene Graph

Build a structure that allows:

- More intuitive/organized modeling
- Global analysis of the model
  - To generate OpenGL calls in an efficient way
- CPU-level preprocessing to avoid overloading graphics pipeline
  - Object culling

Scene Graphs

Thinking objects … not vertices

- Scene Graph: Directed Acyclic Graph (DAG)
  - Represents object-based hierarchy of geometry
  - Suits a bottom-up design of objects & scenes
  - Common for most graphic API's

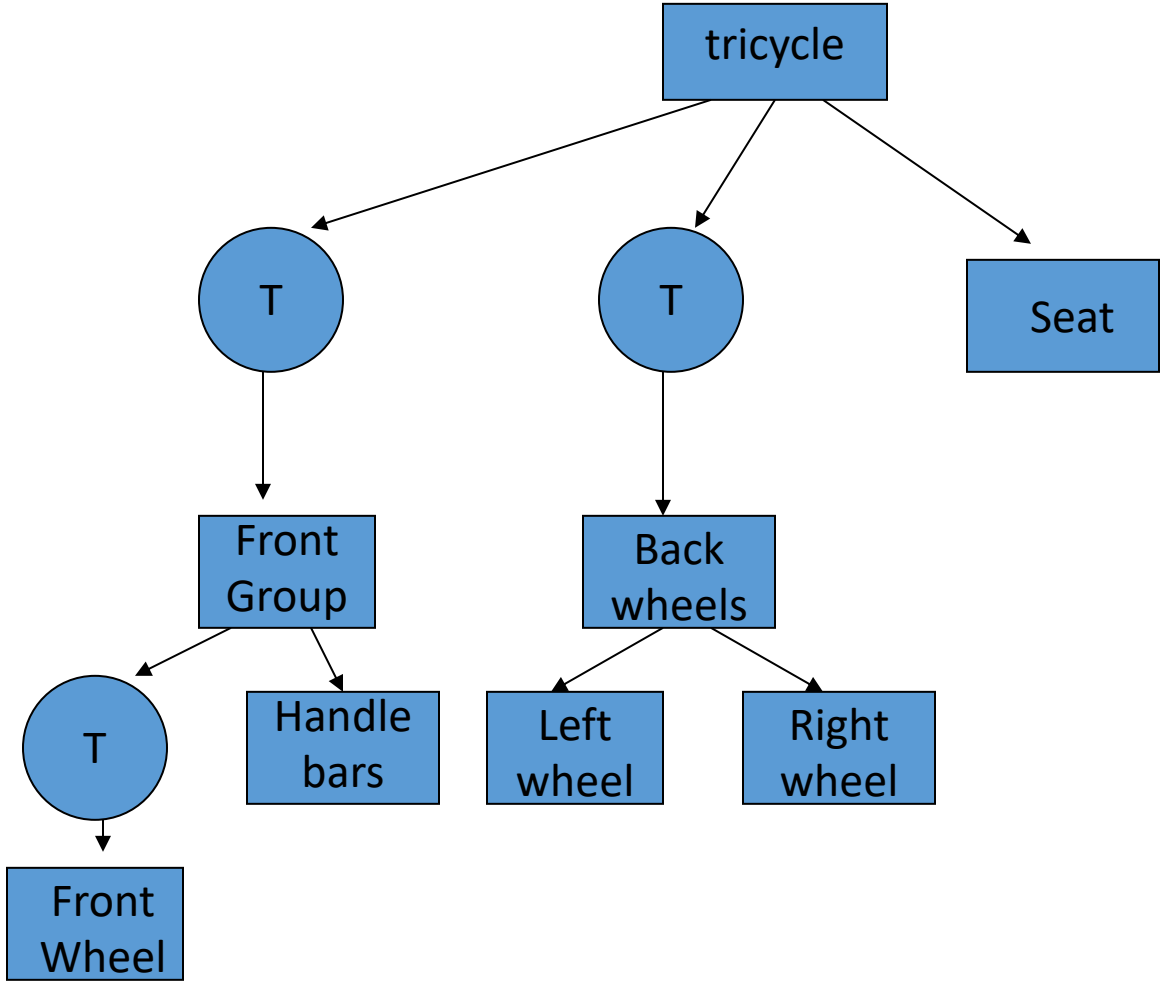
Thinking content … not rendering process

- A scene is a collection of nodes
  - Nodes are: groups, lights, geometry, transforms, sounds, rendering states, etc…
  - Nodes hold pointers to children

Provides the hierarchical framework for grouping objects together, spatially

- Behavior can be built into the scene-graph: collision detection, animations, …
- Abstracts the hardware (i.e. multipass/multitexture)
- Possible to optimize the rendering
  - Sorting of: material state changes; polygons when rendering transparency; etc.

Examples Scene Graph



EXAMPLE OF SOME SCENE-GRAPH ELEMENTS

Group Nodes
- Collects a number of nodes in a logical way
- Level of Detail, Switch nodes

Transformation Nodes
- Moves, rotates, scales its children/siblings

Light Nodes
- Point light, Spotlight, Directional, ...

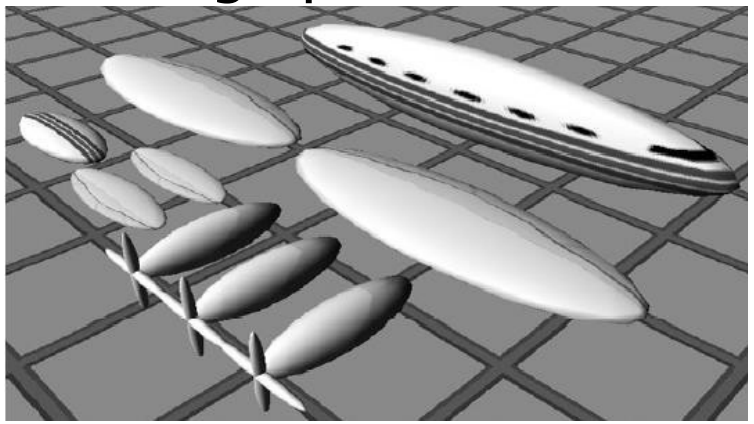
Geometry Nodes
- Usually a leaf node
- Polygons, lines, points, ...
- Material

Others
- Sound, fog, manipulators (animators), Collision, ...

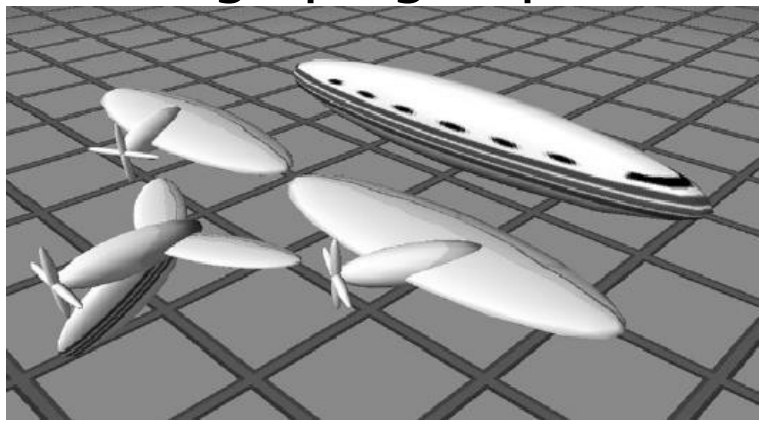
Visitor/Traversers
- Walks through the scene graph and operate on each node
- RenderVisitor, CullVisitor,…

An example

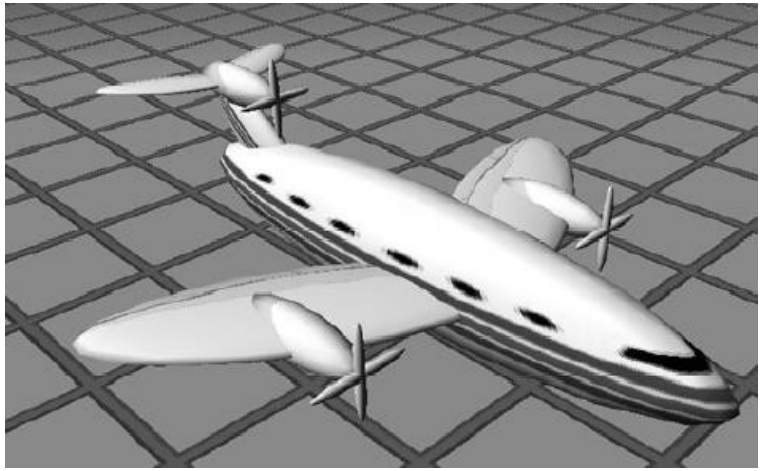
Scene graph items



Scene graph groups



Final Scene



BENEFITS

Transparent use of underlying hardware

Possibly for the scene graph to optimize the rendering of the content

- State sorting (render objects with same shaders, textures, etc. together)
- Rendering order of transparent objects

Sharing of data

- Objects with the same appearance can share state
- Objects with the same geometry can share geometry (smaller memory footprint)

Possible to serialize (and to store or send over the internet)

Rendering Acceleration!

ACCELERATION ALGORITHMS

Never enough horse power

Four performance goals

- Higher FPS
- Higher Resolution
- More realism
- Lower power

We will always need to assist the hardware with clever data structures and algorithms

Spatial Data Structures

Organize data in some n-dimensional space

We want to accelerate queries such as

- Does the following geometry overlap any other geometry?
- Does this ray intersect anything in the scene?

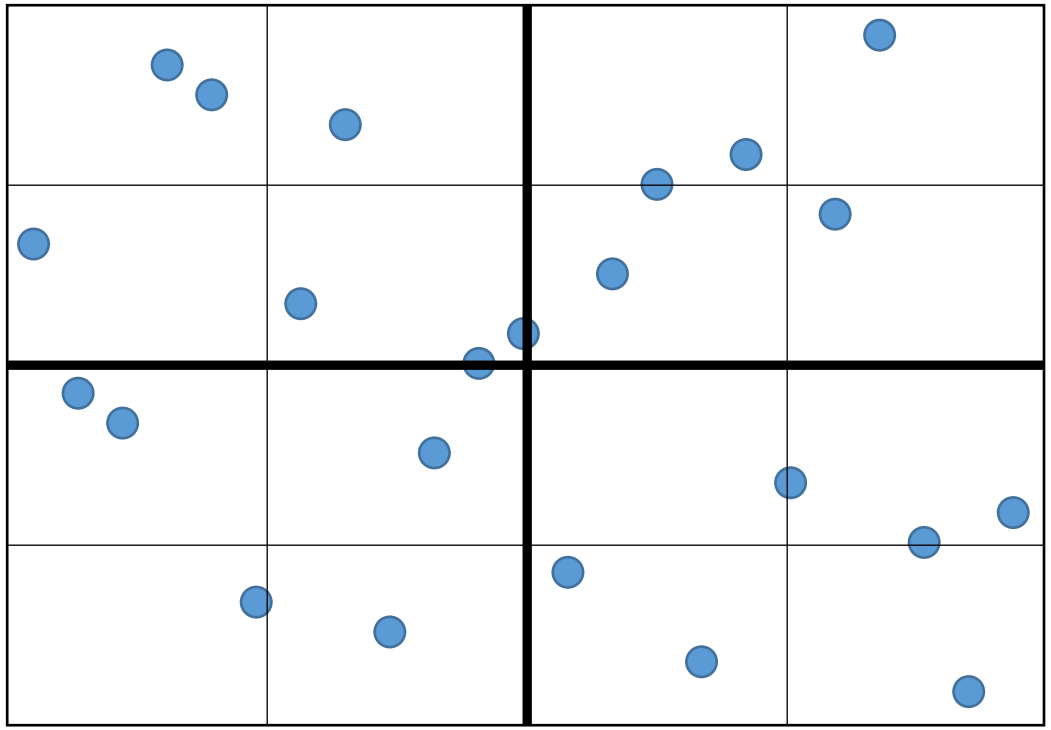
Using a tree structure

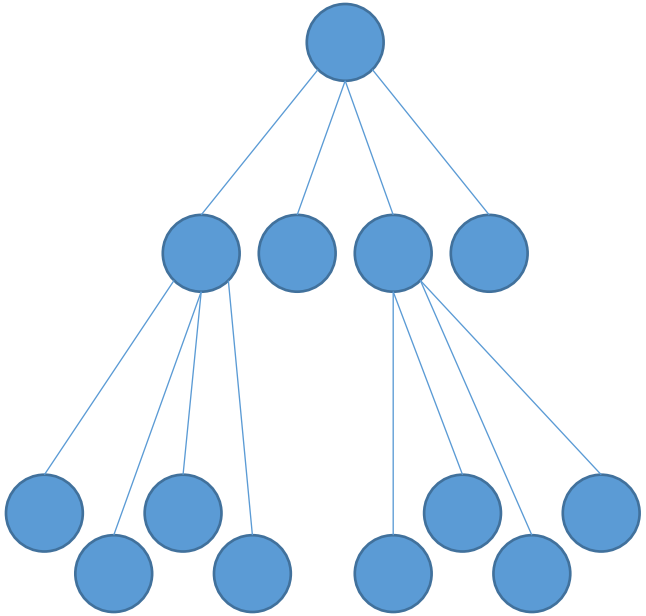
- Construction of a tree can be expensive and should in some cases be done as a preprocess
- Speedups for queries and traversals from O(n) to O(log n)
- Three flavors:
  - Binary Space Partition trees (BSP)
  - Quadtrees/Octrees/Kd-trees
  - Bounding Volume Hierarchies (BVH)

QUADTREES/OCTREES

Recursively divide space in 2D (quadtrees) or 3D (octrees)

At each tree level objects get placed in the quadrant or octant that is "most appropriate".

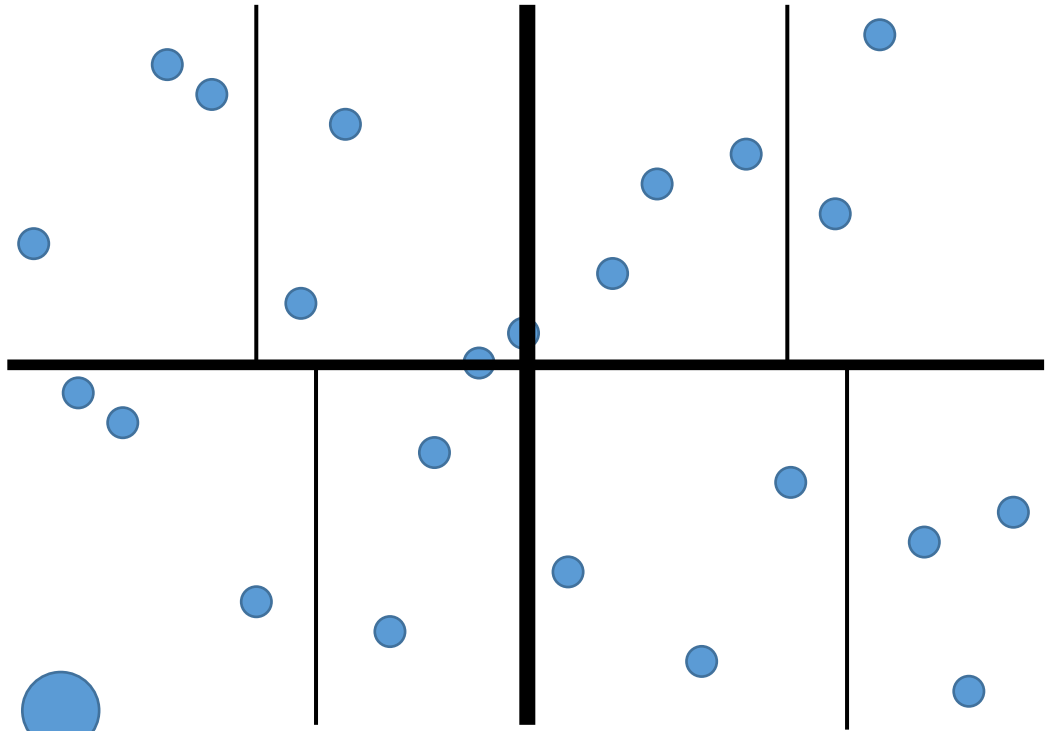


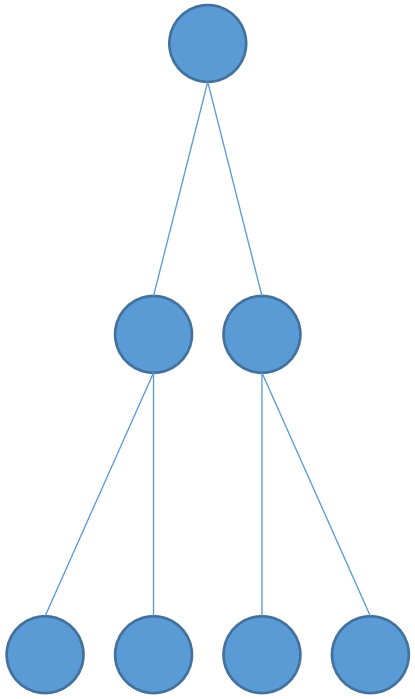


KD-Trees

Recursively divide space in 2 parts by selecting the dimension and cut that best divides the objects

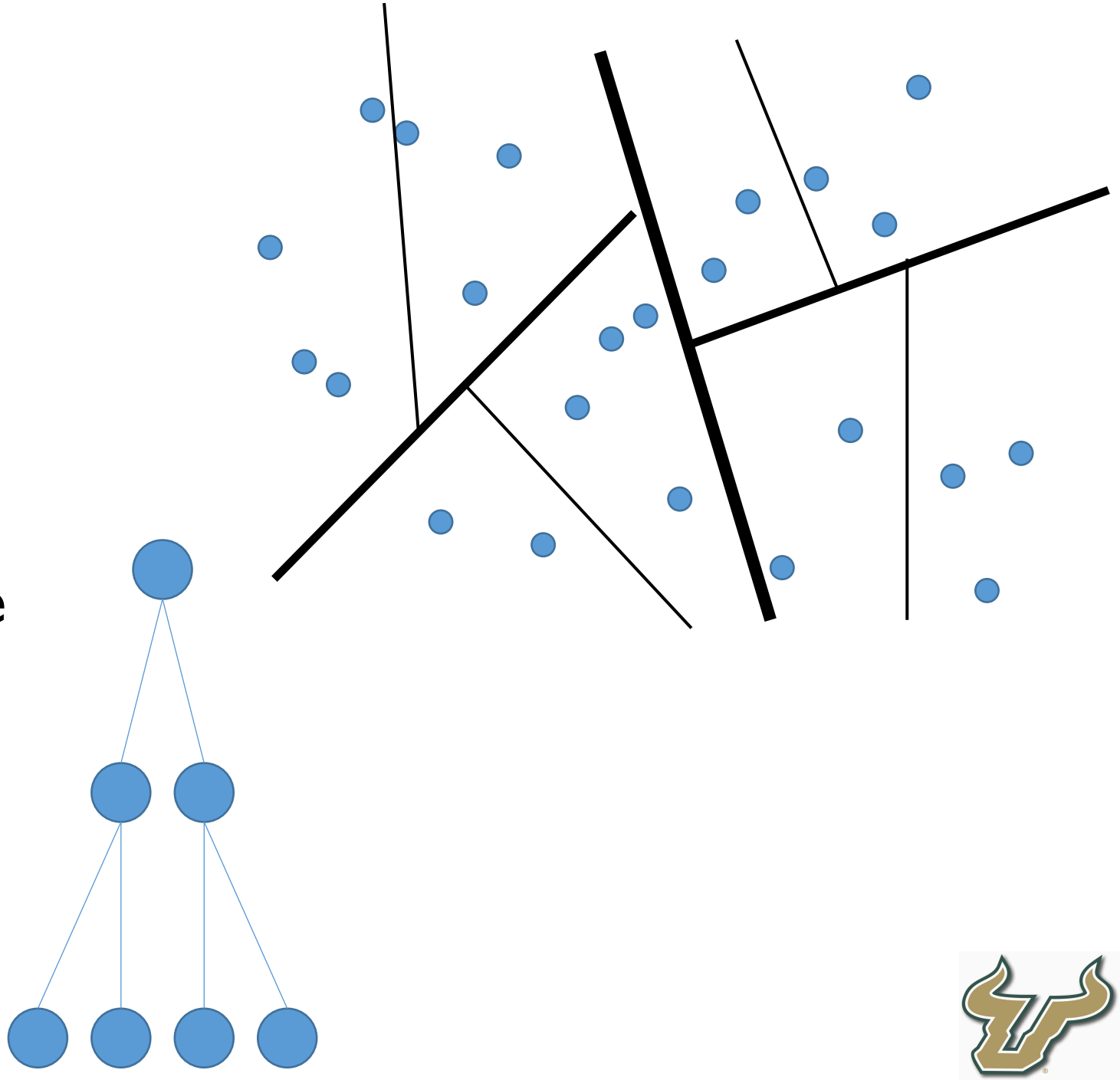
At each level objects get placed in the "most appropriate" division.





Binary Space Partition (BSP)

Similar to kd-tree except cuts no longer need to be axis aligned



BOUNDING VOLUMES

We want a simpler representation of a geometry

- Not for rendering, but for intersection/collision tests

Specs:

- Should fit enclosed geometry as tight as possible
- Should be simple to use in collision test
- Specs are contradictory

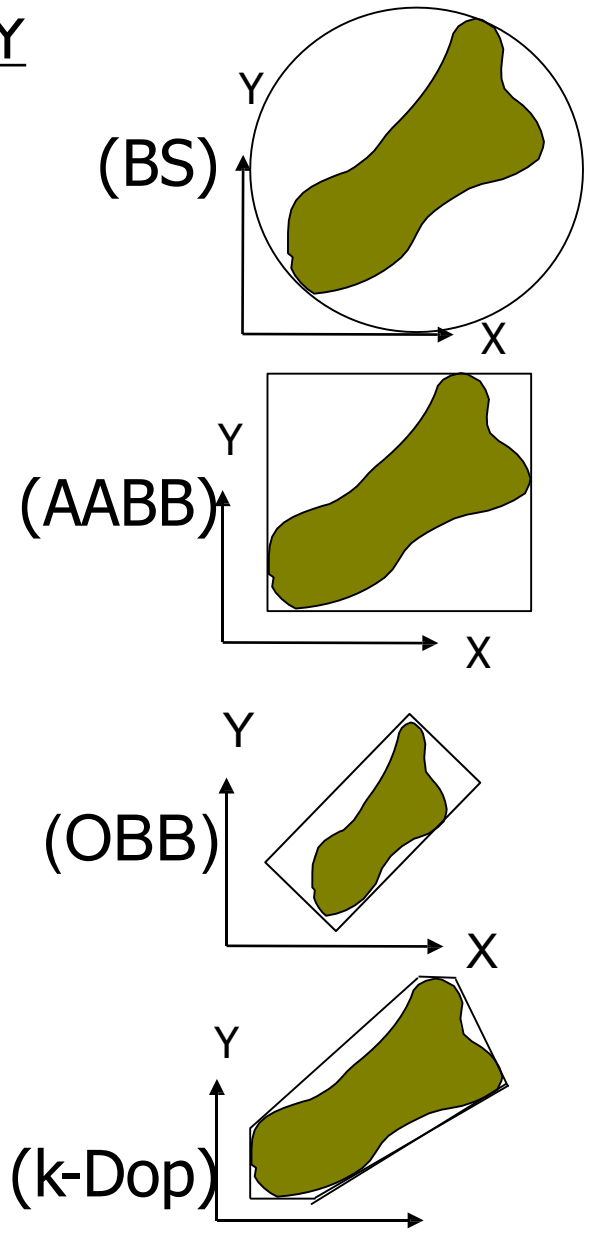
BOUNDING VOLUME GEOMETRY

Bounding spheres
- + Simple to create
- + Simple collision detection
- - Could potentially be a bad choice for long thin objects

Axis Aligned Bounding Box (AABB)
- + Simple to create
- + Simple collision detection
- - A lot of empty space enclosed

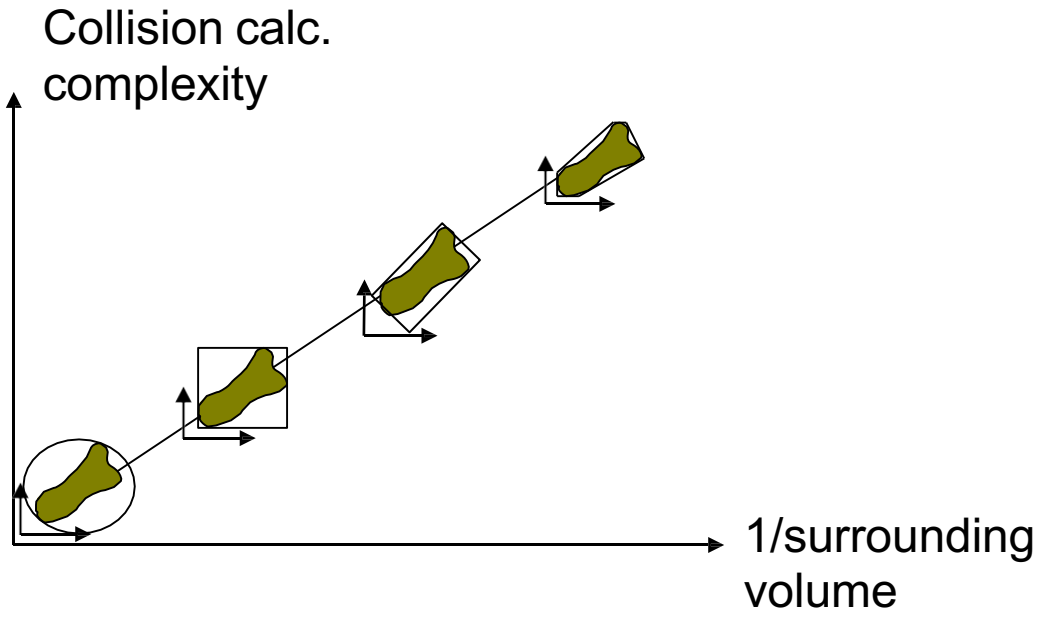
Oriented Bounding Box (OBB)
- + Less empty space enclosed
- - More complicated to create
- - More complicated during collision detection

k-Dop (discrete oriented polytope)
- + Least empty space
- - Most complicated to calculate



BOUNDING VOLUME GEOMETRY

Less empty space – more complex to create and use



"*The fastest rendered polygon, is the one never sent down the rendering pipeline*"

Real-Time Rendering
Akenine-Möller, Haines, and Hoffman

SPEEDING THE GRAPHICS PIPELINE

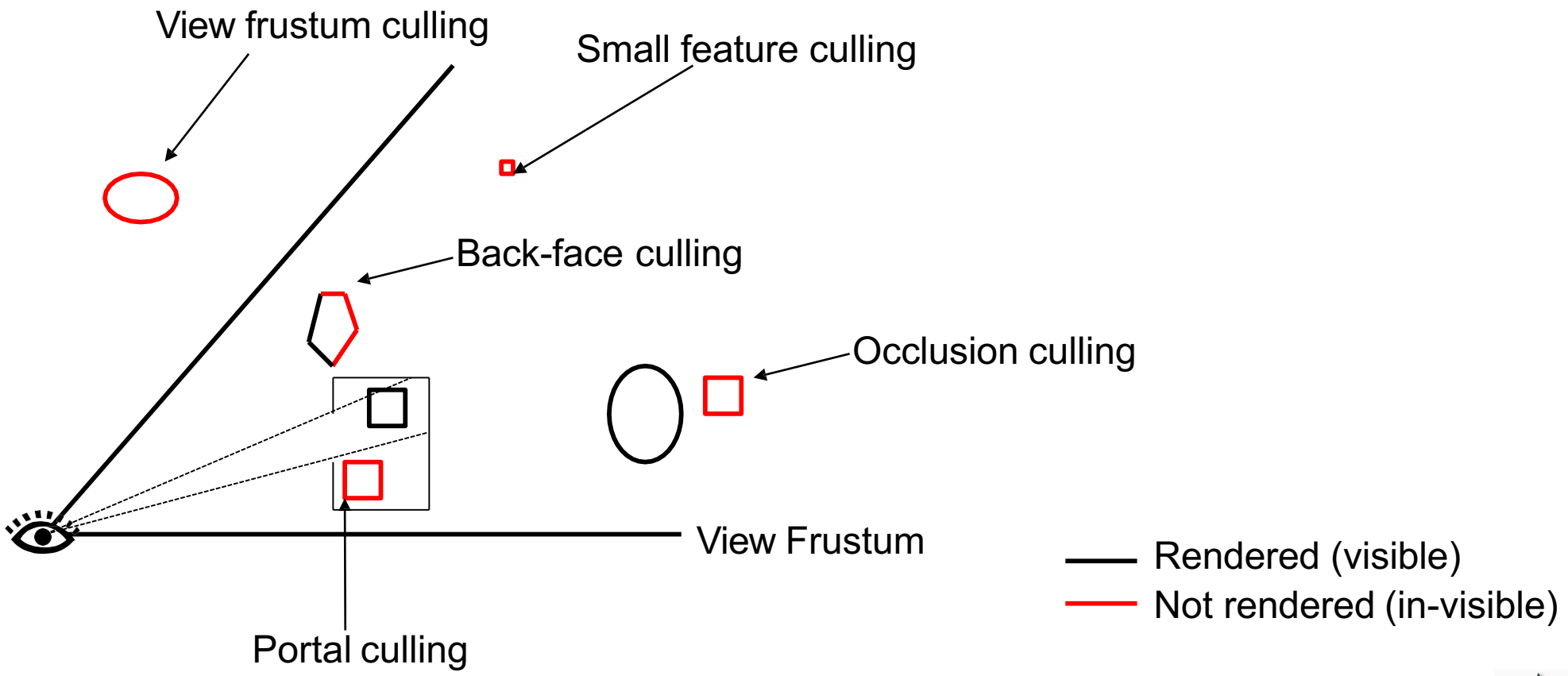
What are some ways the graphics pipeline speeds rendering?

- Off-screen geometry: solved by clipping
- Occluded geometry: solved by Z-buffer
- Clipping/Z-buffering take time linear to the number of primitives/fragments

How can Scene Graphs help?

- Off-screen: **view-frustum culling**, **portal culling**, **contribution culling**
- Occluded by other objects: **occlusion culling**, **backface culling**
- All work on the object level, instead of the primitive level

CULLING


View frustum culling
Small feature culling
Back-face culling
Occlusion culling
View Frustum
Portal culling
Rendered (visible)
Not rendered (in-visible)

View Frustum Culling Goal

Quickly eliminate large portions of the scene which will not be visible in the final image

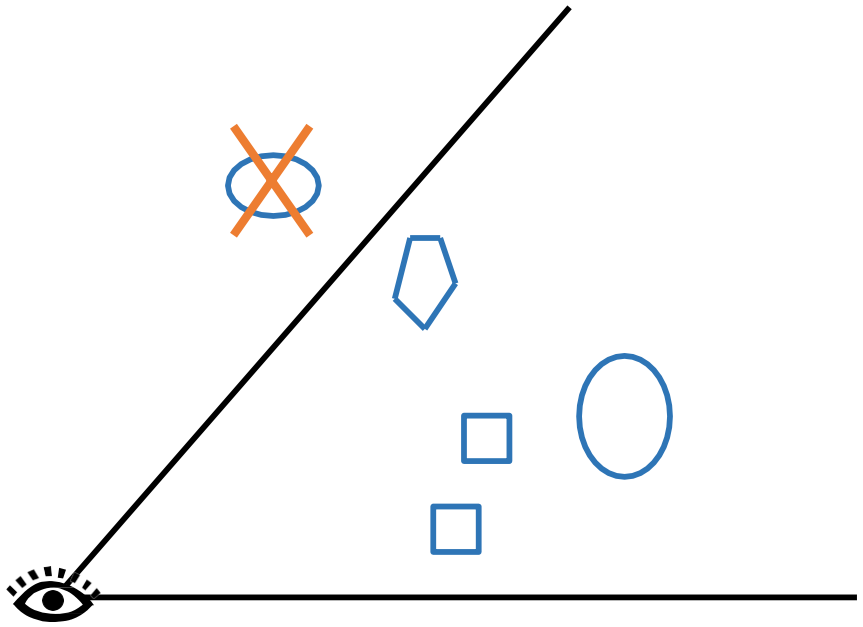
- Not the exact visibility solution, but a quick-and-dirty conservative estimate of which primitives might be visible
  - Z-buffer & clipping this for the exact solution
- This conservative estimate is called the potentially visible set or PVS

VIEW FRUSTUM CULLING

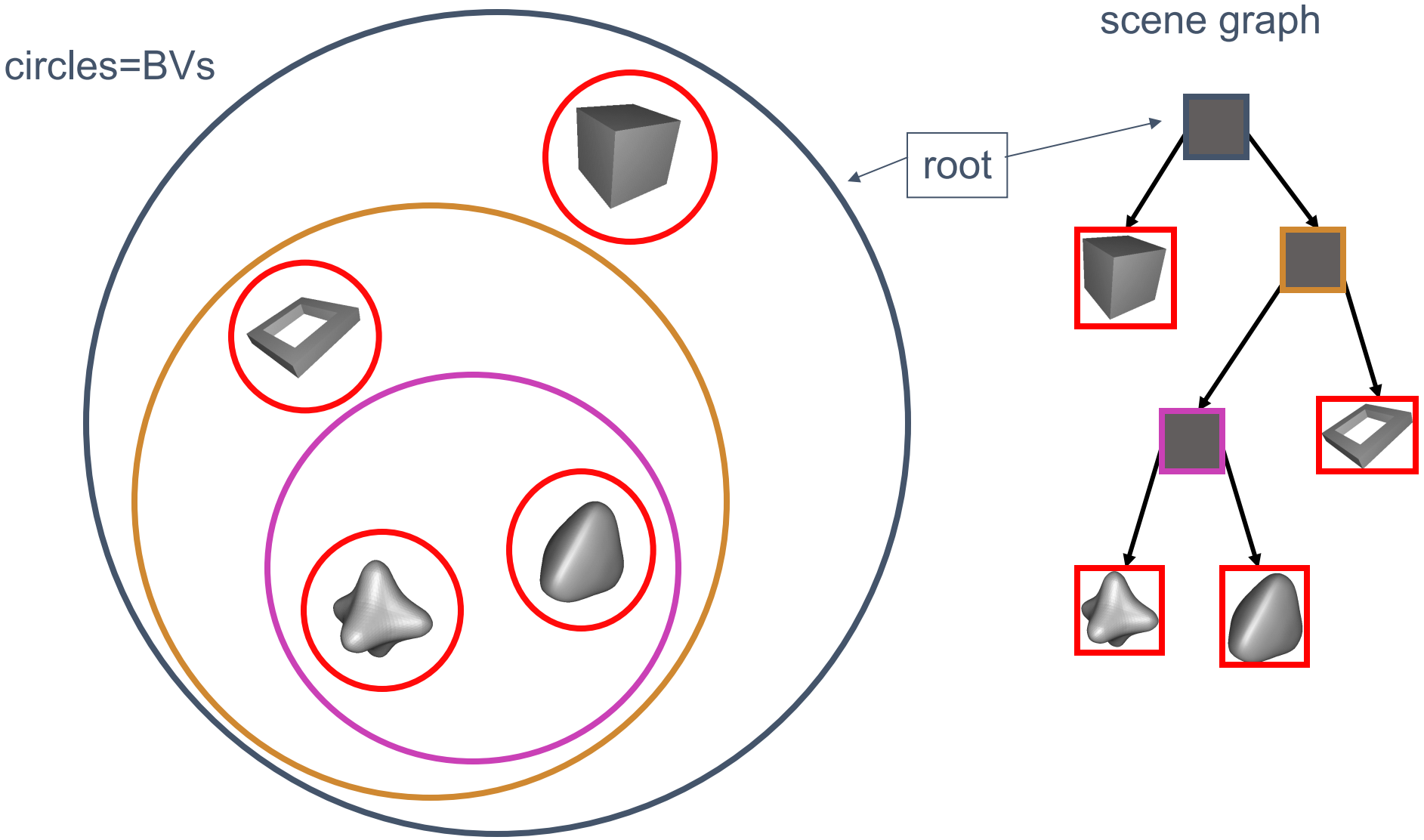
Remove object not inside view frustum (VF)

Complete test is called exclusion/inclusion/intersection test

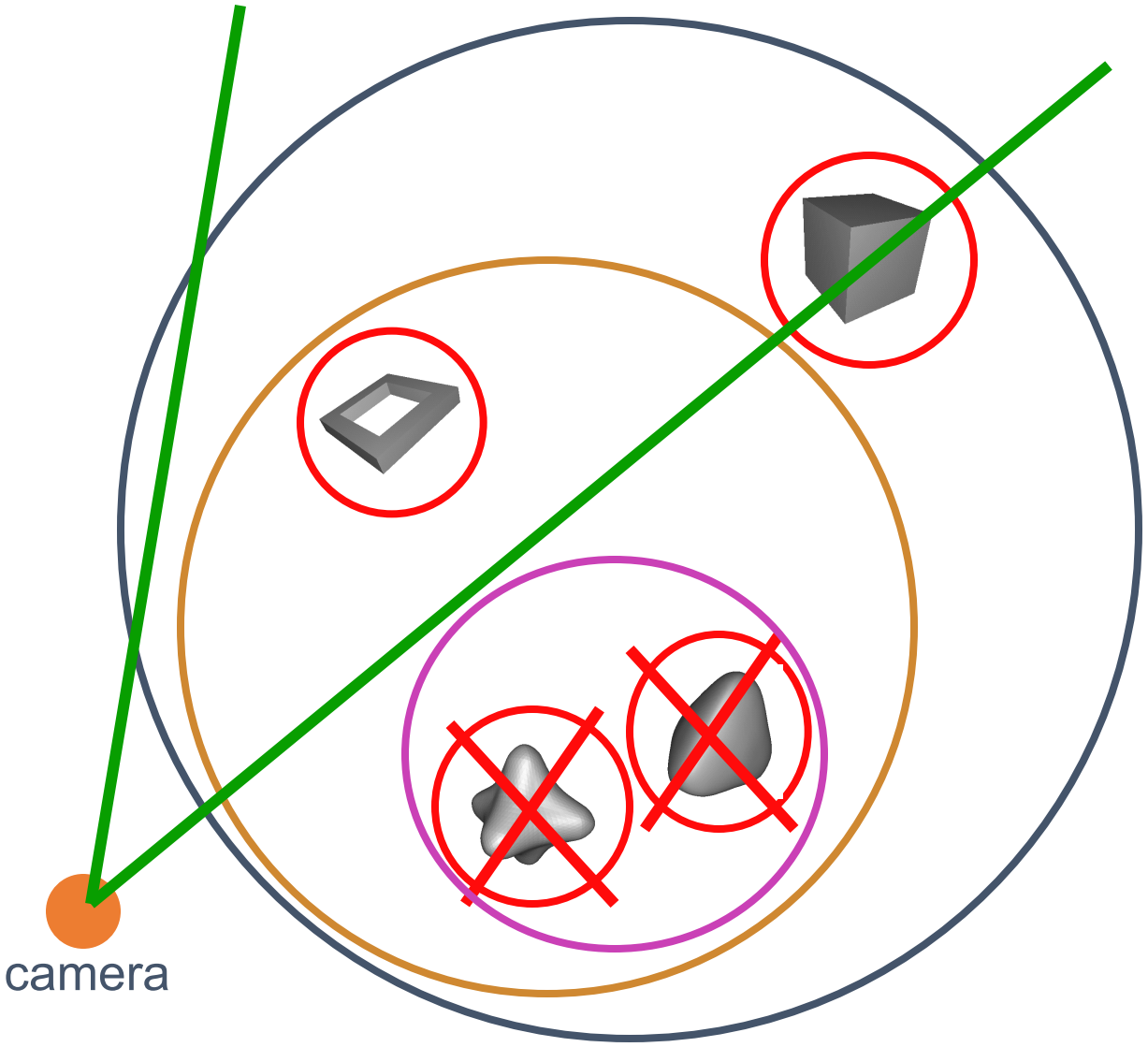
- Exclusion a BV is completely outside the VF, thus not rendered
- Inclusion the BV is completely inside the VF, thus rendered
- Intersection, part of the BV is inside, part is outside. Here we have a few options.

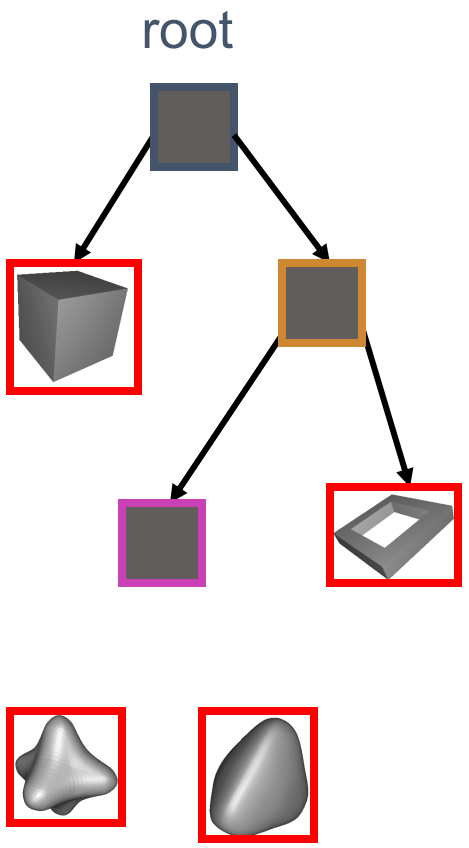


SCENE GRAPH EXAMPLE



Example of Hierarchical View Frustum Culling





Portal Culling

Goal: walk through architectural models (buildings, cities, catacombs)

Divide scene into cells

- Room, hallways
- Each object (furniture etc.) associated to a cell are referenced

Define portals

- Door/windows connect adjacent rooms
- For each cell associate portals to that cell

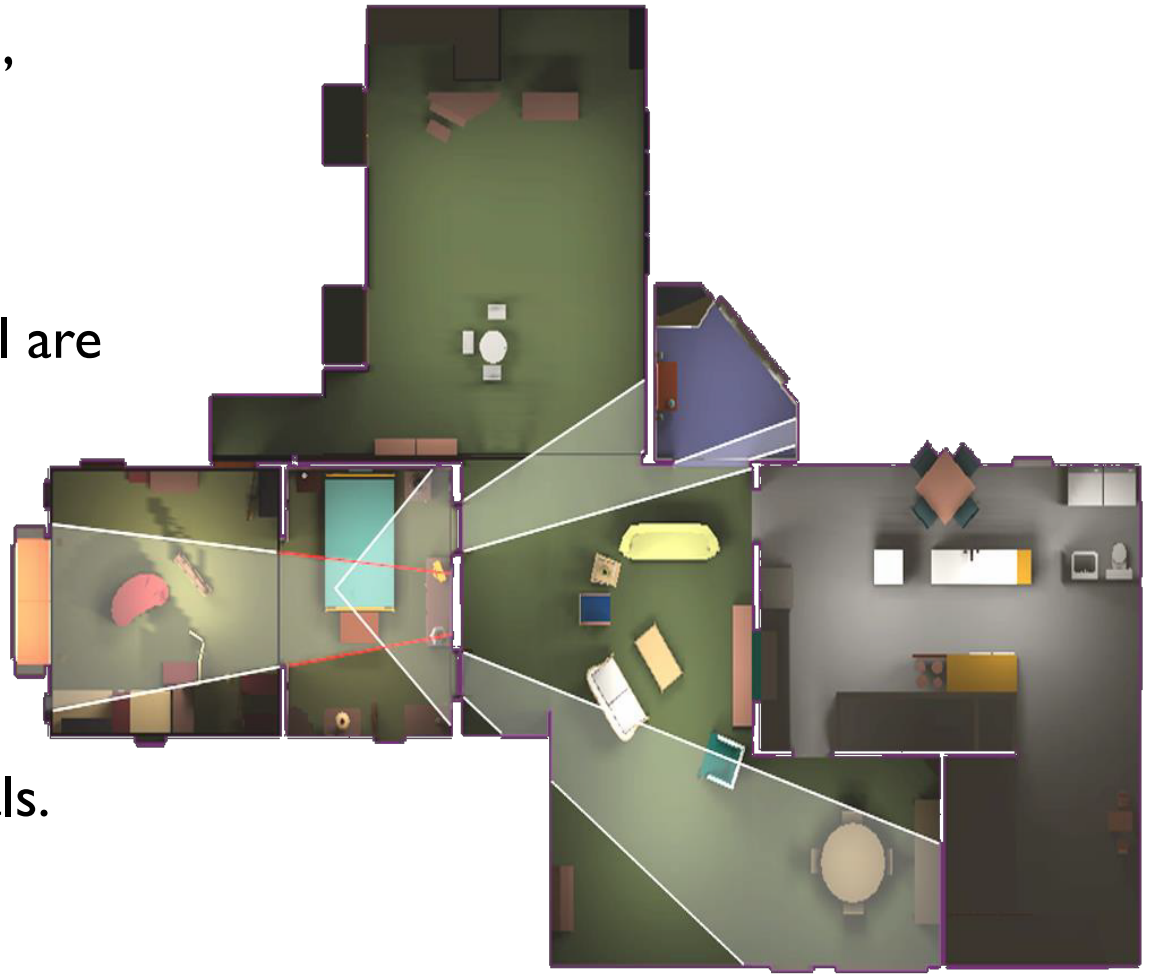
Create an adjacency graph of cells

- Each cell connect to other cells through portals.

Extension of VF culling

- The VF is adjusted to the adjacent portals

Notice: cells only see other cells through portals



Cells & Portals

Starting with cell containing eye-point, traverse graph, rendering visible cells using original View Frustum (VF)

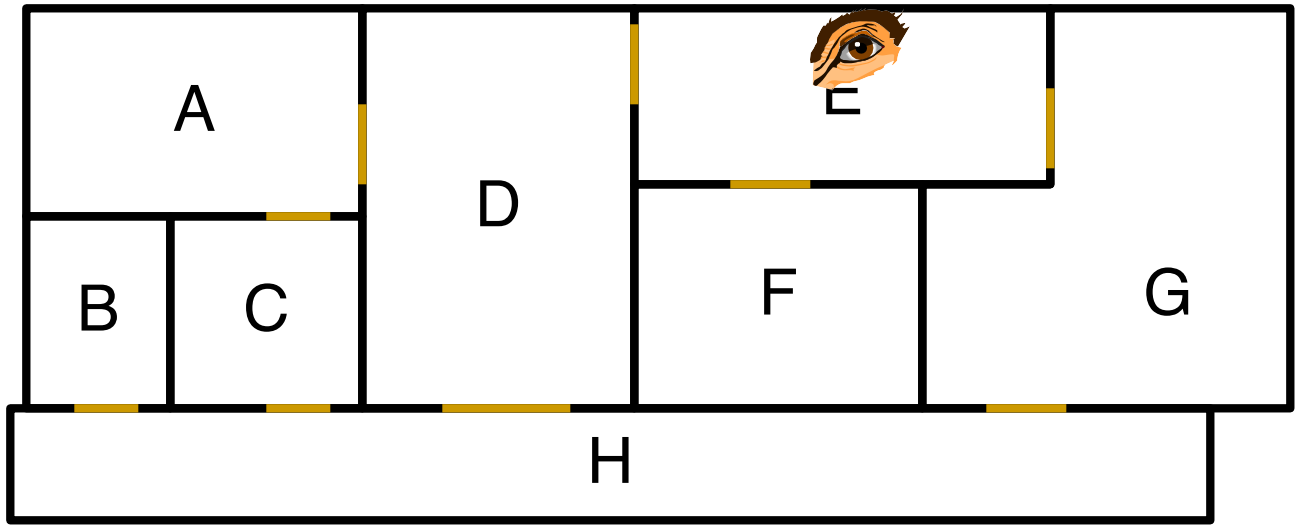
For each Portal visible in VF

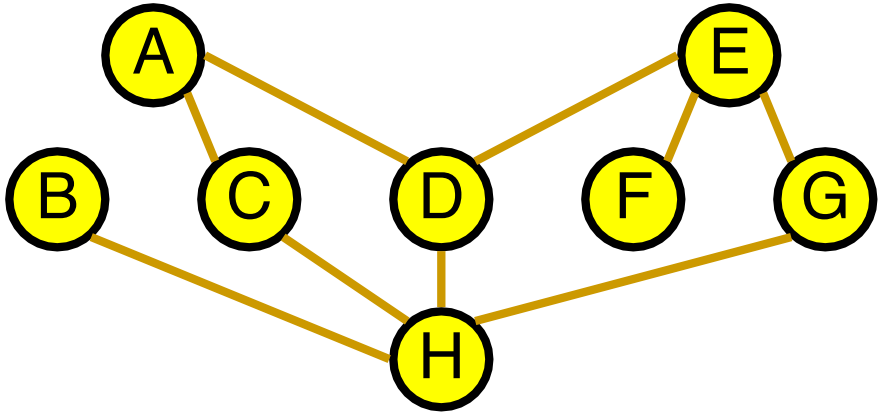
- Reduce VF to fit Portal
- Now render Cell with reduced VF
  - For each portal visible in VF… (recursion)

A cell is only visible if it can be seen through a sequence of portals

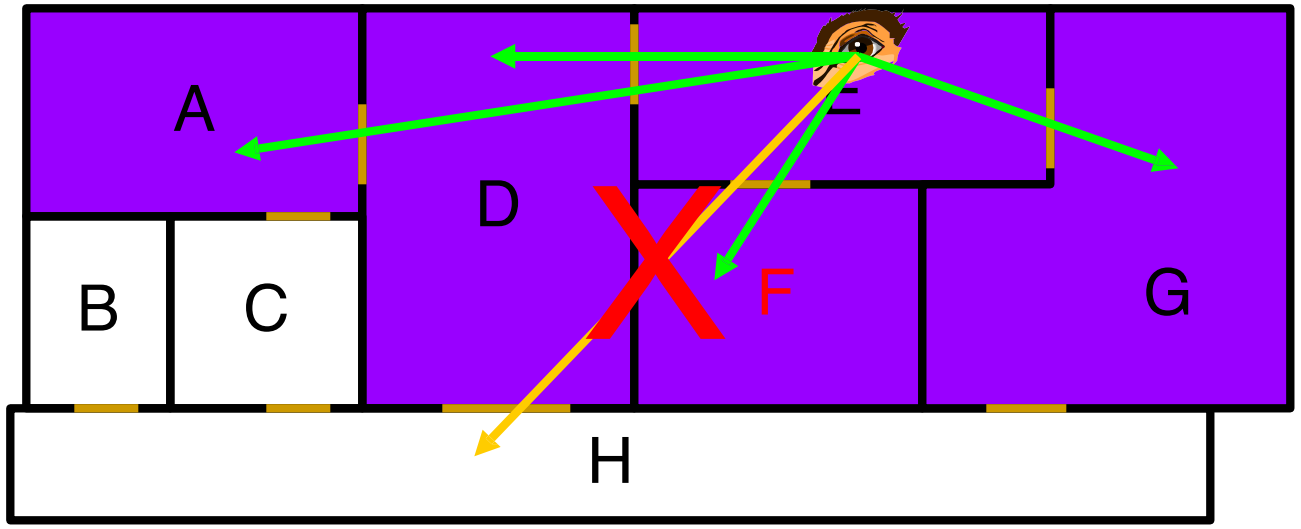
- So cell visibility reduces to testing portal sequences for a line of sight…

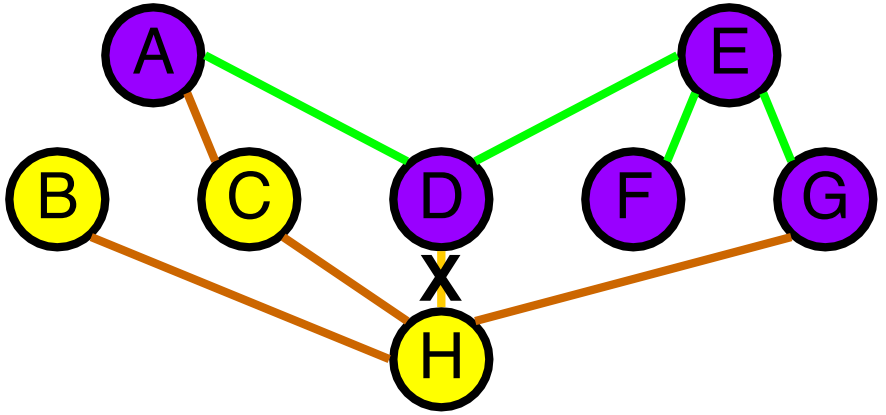
Cells & Portals





Cells & Portals





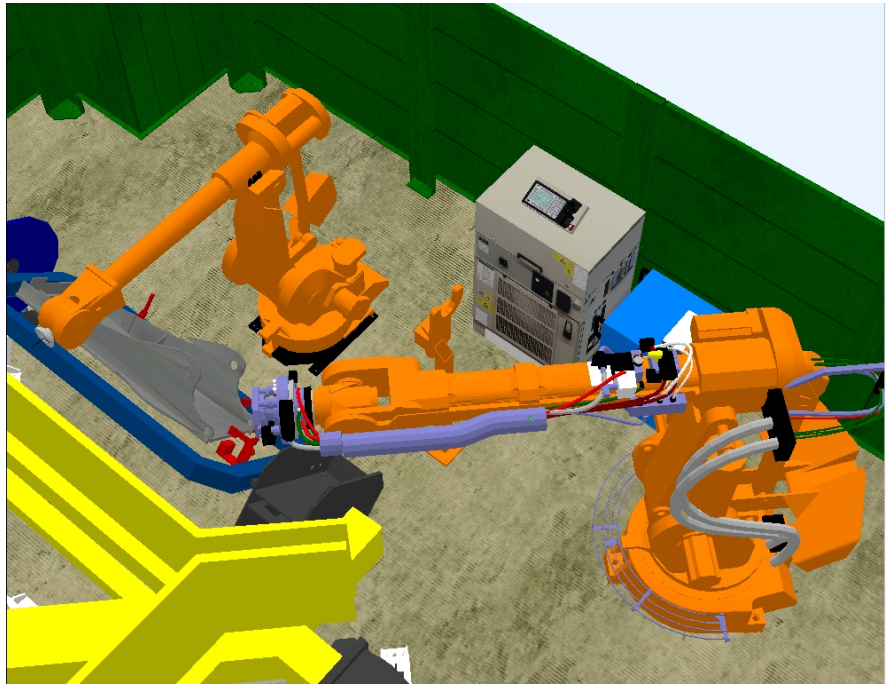
Contribution Culling

Idea: objects whose projected bounding volume occupy less than N pixels are culled

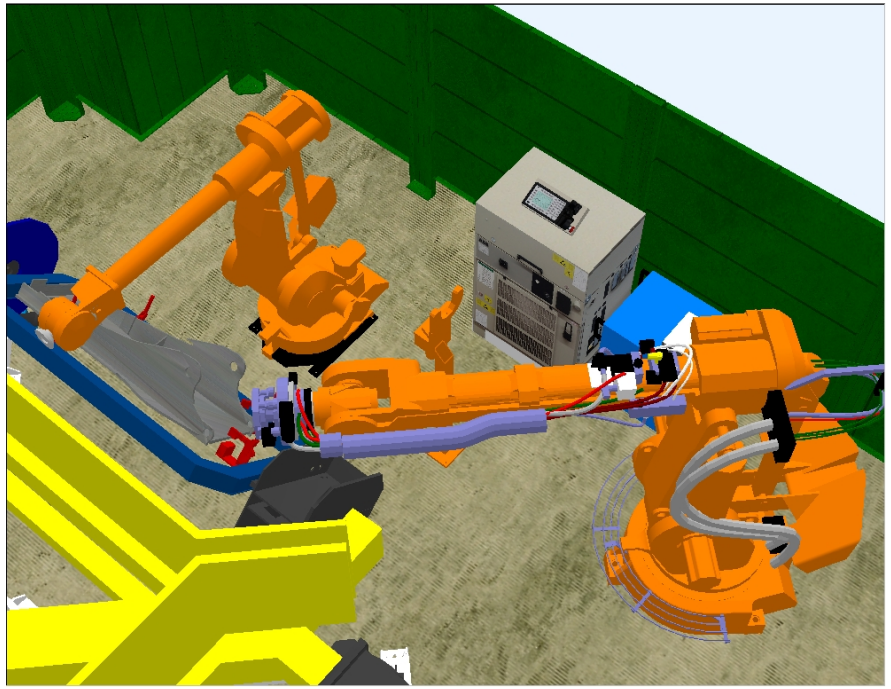
This is an approximative algorithm as the things you cull away may actually contribute to the final image

Advantage: trade-off quality/speed

Example of Contribution Culling



contribution culling OFF



contribution culling ON

Not much visual difference, but 80-400% faster

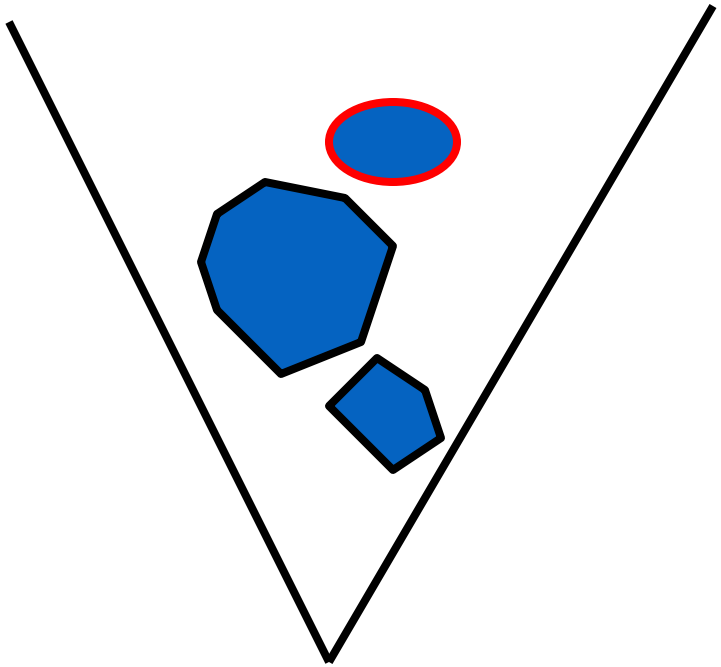
Particularly effective when moving

Images courtesy of ABB Robotics Products, created by Ulf Assarsson

Occlusion Culling

Main idea: Objects that lies completely "behind" another set of objects can be culled

Hard problem to solve efficiently

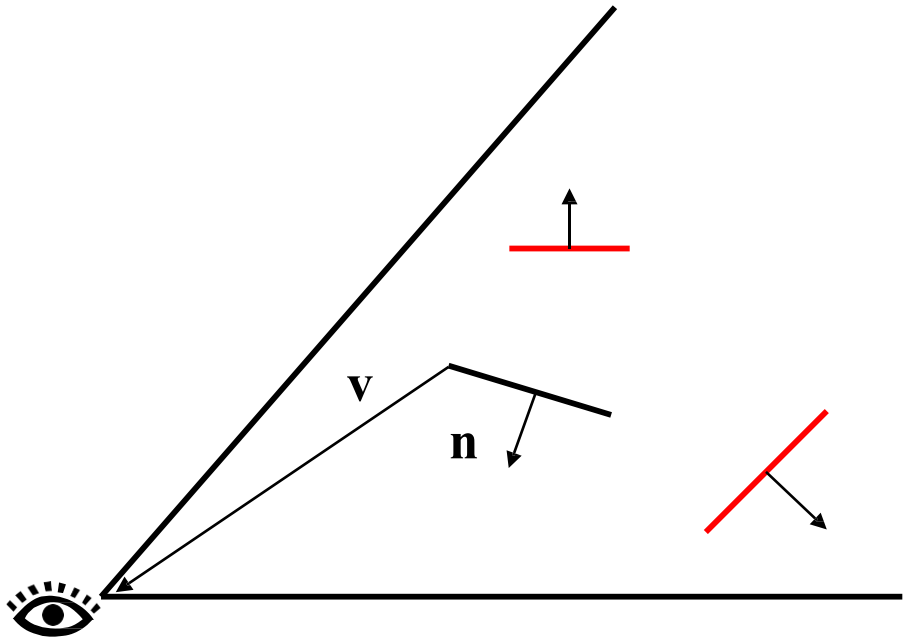


BACKFACE CULLING

If normal of a surface is pointing away from viewer, remove it

Works for closed object where we don't want to see the inside of the object (for example a sphere)

Does not work well with for terrain (most polygons visible)



Simplify Geometry

One polygon

- Sprites
- Billboards
- Impostors

Many polygons

- Level of detail (LOD)

Old School: Sprites

A rectangular shaped image that moves around on the screen

- One-to-one mapping with pixels on the screen
- The earliest PC graphics were sprite

Implementation

- An image texture on a polygon with the use of alpha channel

background **sprite image** **result**

BILLBOARDING

Textured polygon

Combined with alpha texturing and (potentially) animation

- ex. smoke, fire, explosions, vapor trails, clouds

Axial Billboard

Billboards at fixed location rotates to face user

Useful for representing objects with cylindrical symmetry

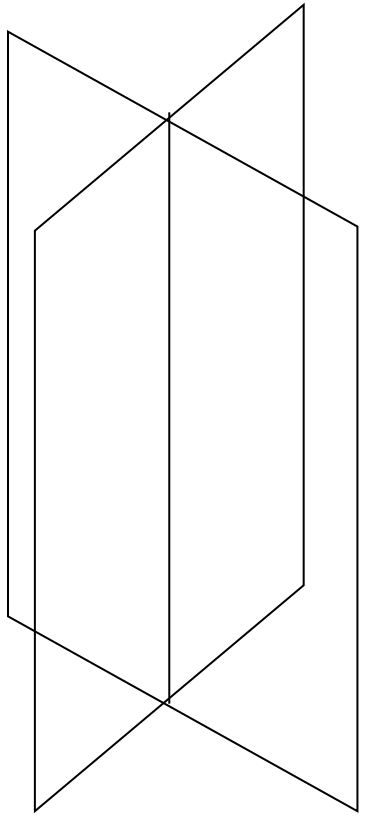
- displaying trees

Multi-Polygon Billboards

Use two polygons at right angles

- No alignment with viewer

Use more polygons for better appearance



IMPOSTORS

Replace 3D geometry with a 2D image

2D image fools viewer into thinking 3D geometry is still there

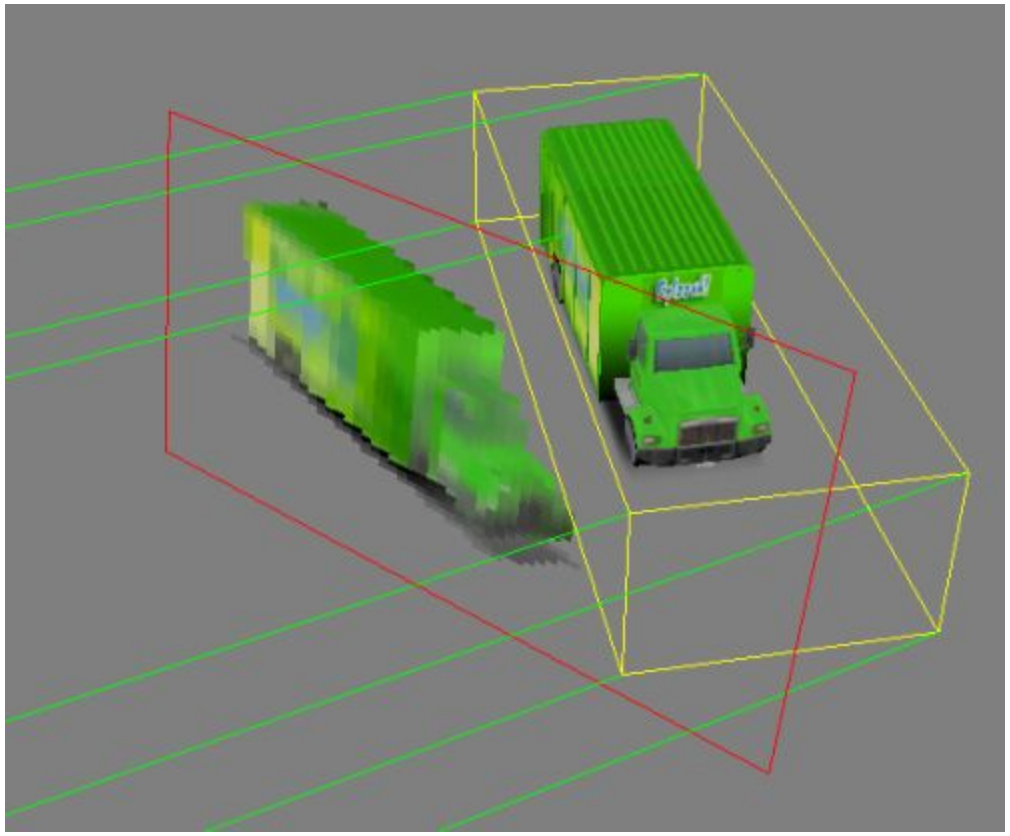
Big limitation: No parallax

Impostors Technique

Basic idea

- Render set of geometry into a large texture: an impostor
- Now render impostor texture instead of the geometry

Only helpful when impostors can be reused across many frames

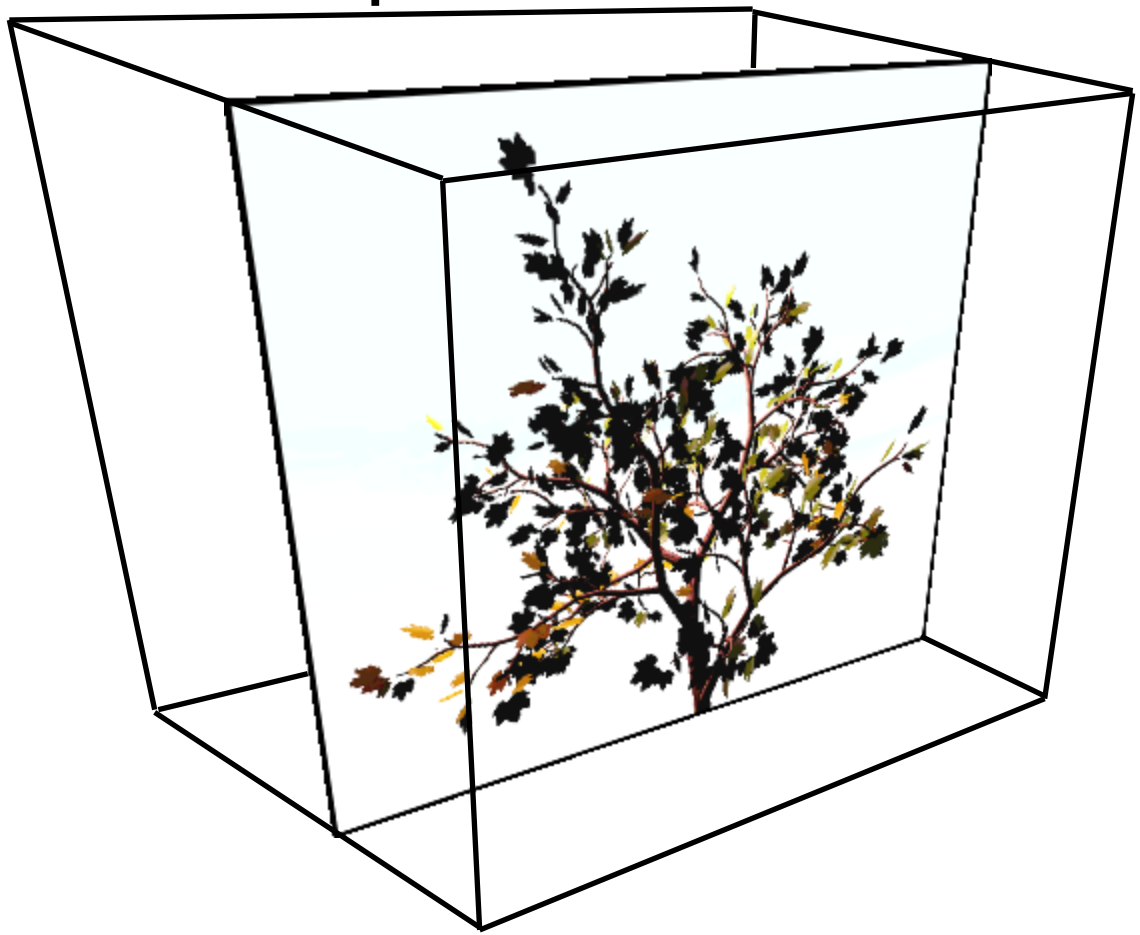


IMPOSTORS: EXAMPLE

We render a set of geometry into an impostor (image/texture)

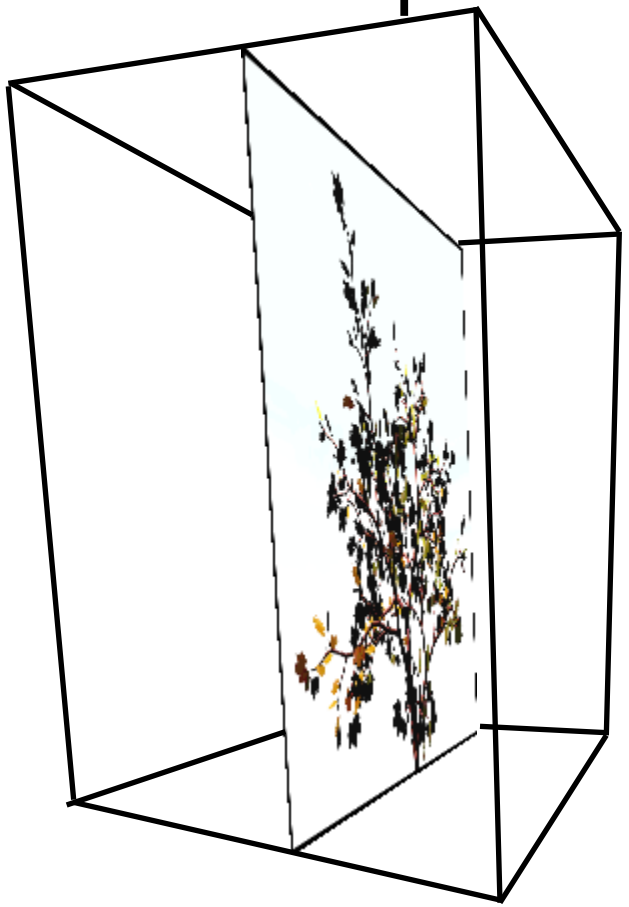
IMPOSTORS: EXAMPLE

We can re-use this impostor in 3D for several frames



IMPOSTORS: EXAMPLE

Eventually, we have to update the impostor



Level of Detail

The problem:

- Geometric datasets can be too complex to render at interactive rates

One solution:

- Simplify the polygonal geometry of small or distant objects
- Known as ***Level of Detail*** or ***LOD***
  - A.k.a. polygonal simplification, geometric simplification, mesh reduction, decimation, multiresolution modeling, …
- Focuses on the fidelity / performance tradeoff

Level-of-Detail Rendering

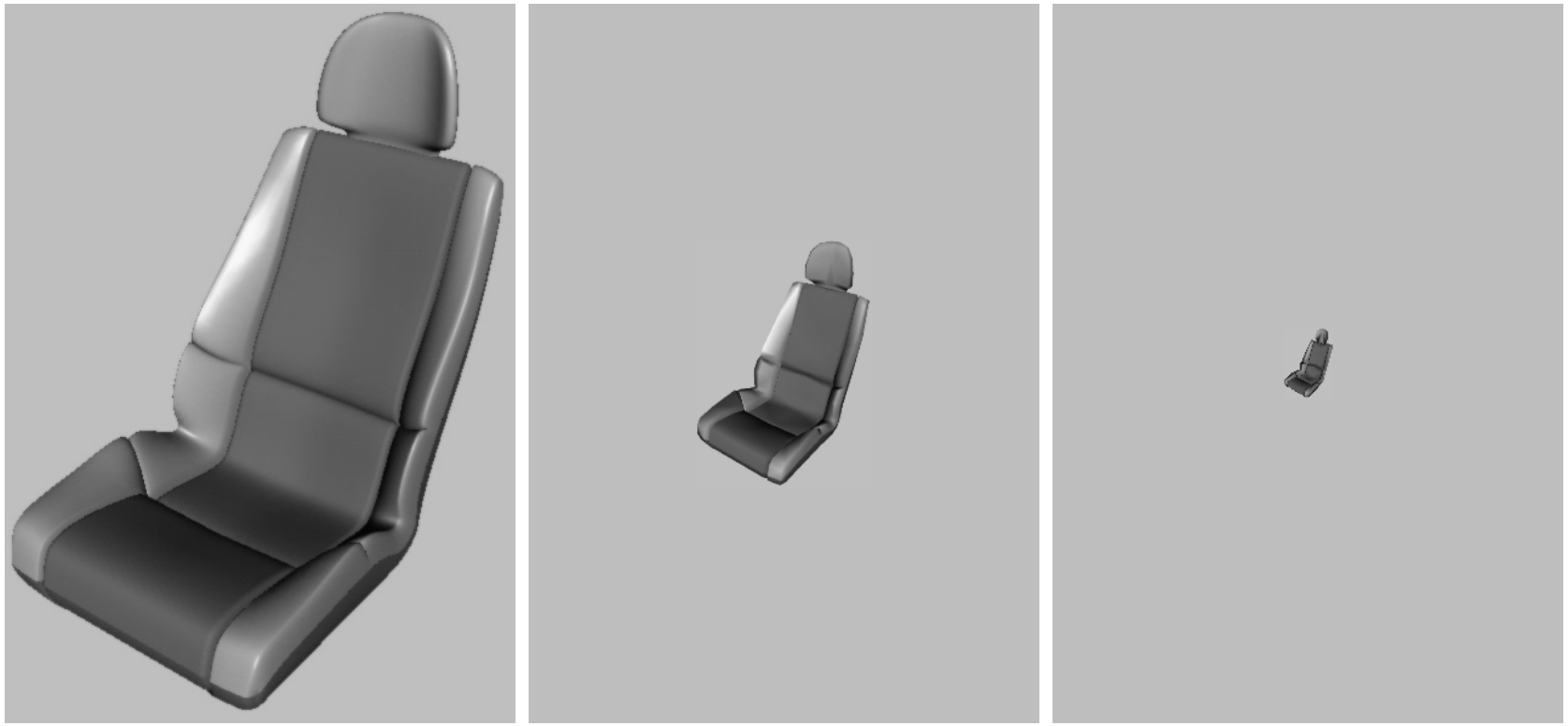
Use different levels of detail at different distances from the viewer

More triangles closer to the viewer

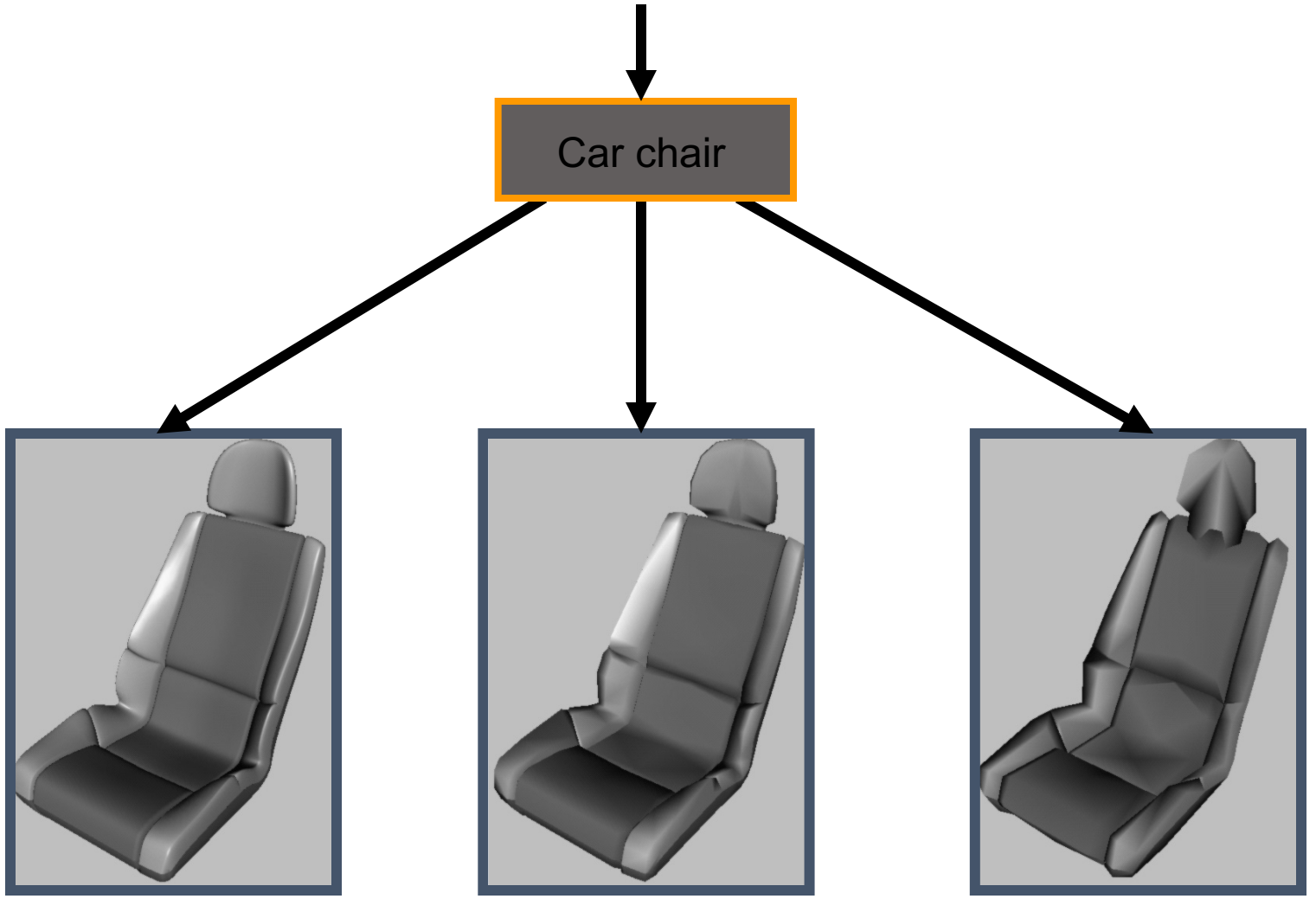
LOD RENDERING

Use area of projection of BV to select appropriate LOD

Not much visual difference, but a lot faster



SCENE GRAPH WITH LODS

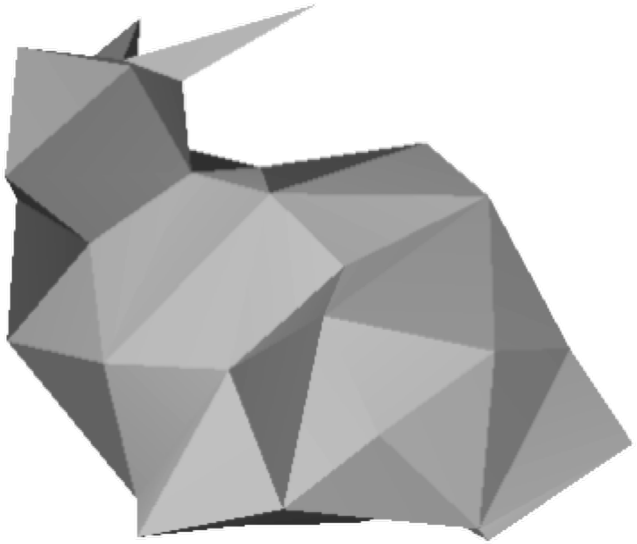


Another Example

69,451 polys

2,502 polys

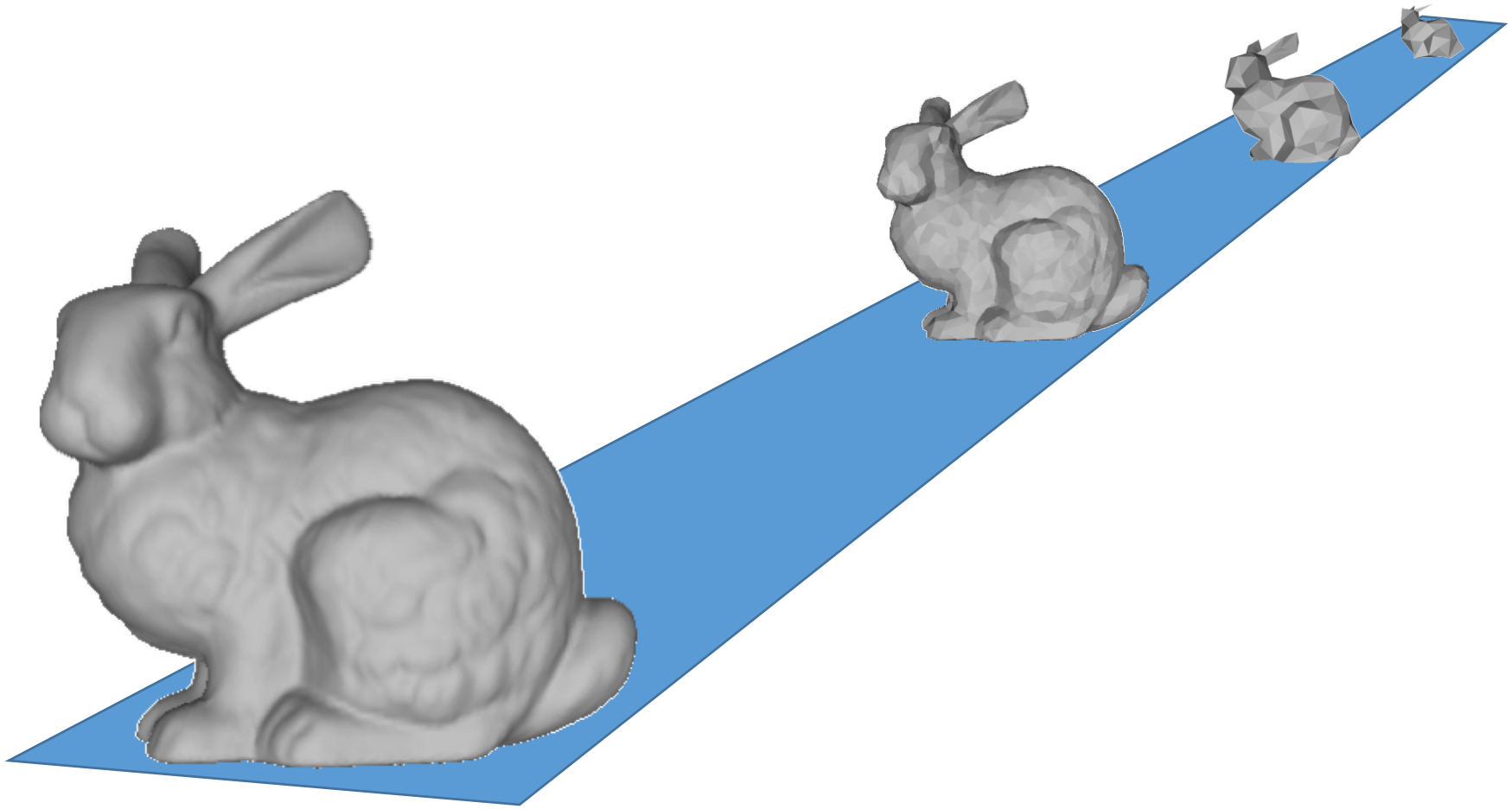
251 polys



76 polys

Courtesy Stanford 3D Scanning Repository

Another Example



Courtesy Stanford 3D Scanning Repository

Level of Detail: The Big Questions

How to generate simpler versions of a complex model?

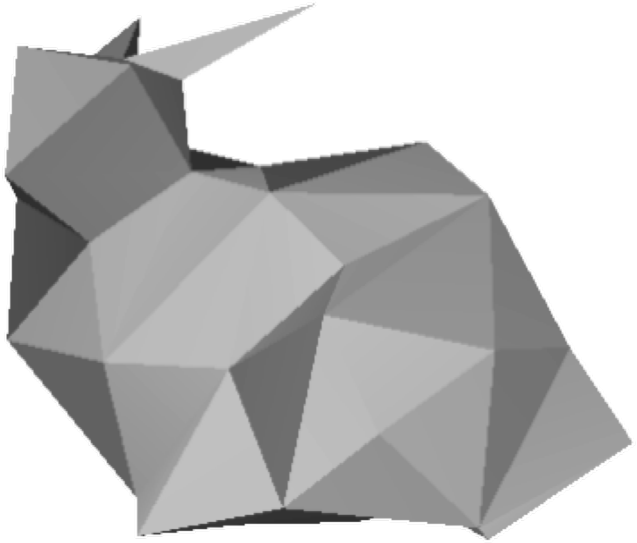
How to evaluate the fidelity of the simplified models?

When to use which LOD of an object?

69,451 polys

2,502 polys

251 polys



76 polys

Courtesy Stanford 3D Scanning Repository

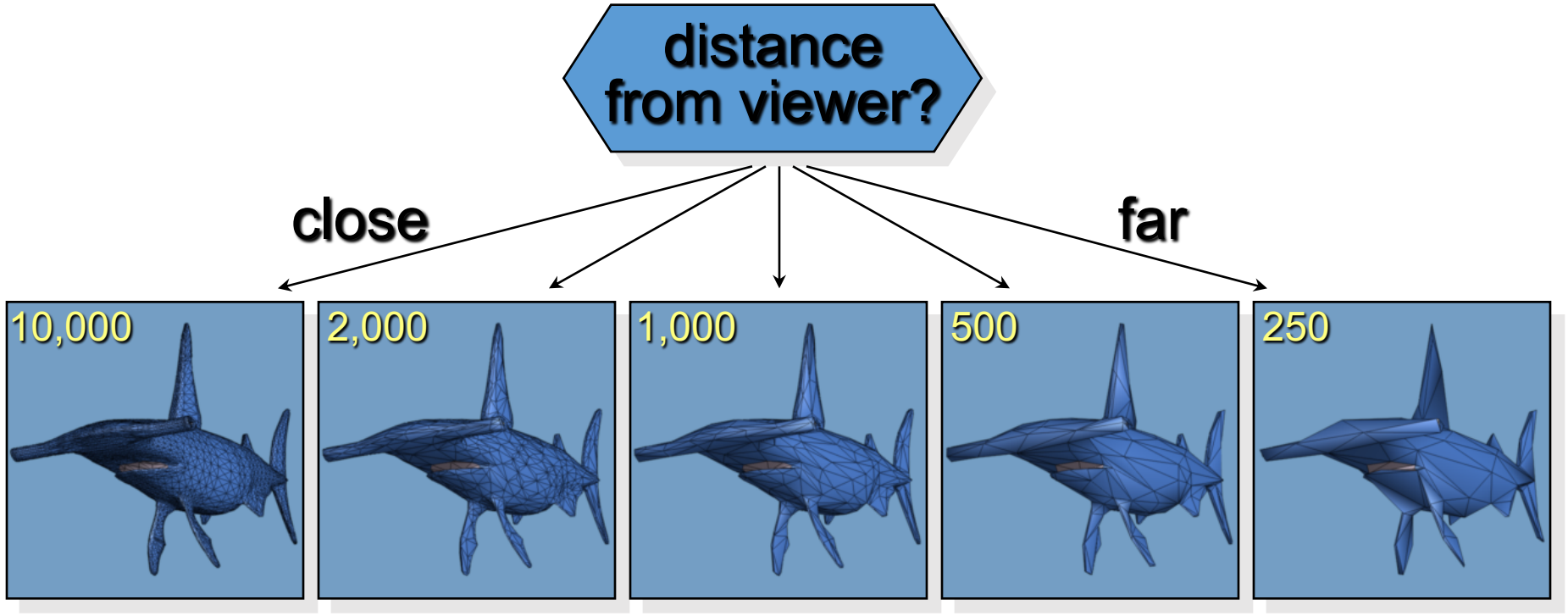
Discrete LOD Approach

Traditional LOD in a nutshell:

- Create LODs for each object separately in a preprocess
- At run-time, pick each object's LOD according to the object's distance (or similar criterion)

Since LODs are created at fixed resolutions, we call this discrete LOD

DISCRETE LEVEL-OF-DETAIL (LOD)



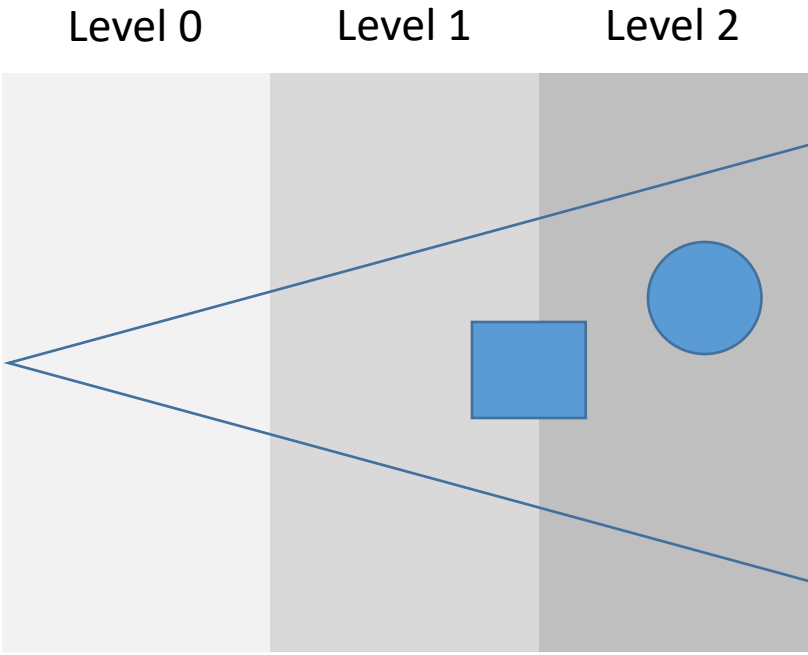
[CLARK76, FUNKHOUSER93]

Discrete LOD: Advantages

Simple programming model—decouples simplification and rendering

- LOD creation need not address real-time rendering constraints
- Run-time rendering need only pick LODs

Choosing the LOD



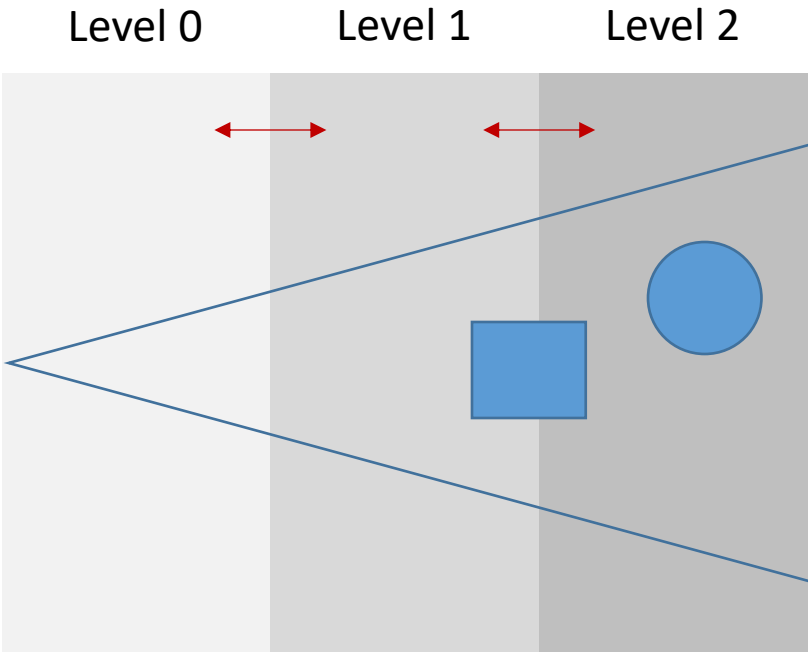
Heuristic Approach

- Assign each LOD a range of distances
- Calculate distance from viewer to object
- Use corresponding LOD

Problems:

- Visual "pop" when switching LODs can be disconcerting
- Doesn't maintain constant frame rate
- Requires someone to assign switching distances by hand
- Correct switching distance may vary with field of view, resolution, etc.

Choosing the LOD



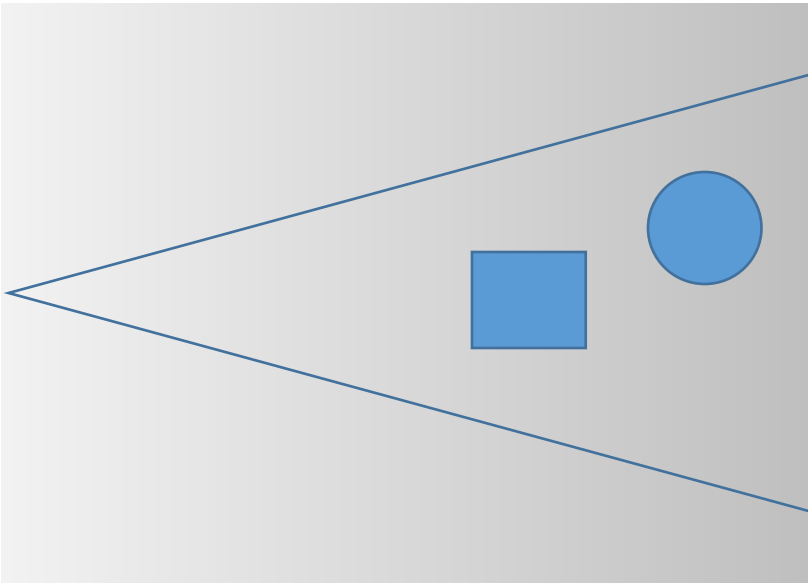
Adaptively maintaining frame rate

- Scale LOD switching distances by a "bias"
  - If last frame took too long, decrease bias
  - If last frame took too little time, increase bias

Problems:

- Oscillation caused by overly aggressive feedback
- Sudden change in rendering load can still cause overly long frame times

Choosing the LOD



Predictively maintaining frame rate

For each LOD estimate:

- Cost (rendering time)
  - # of polygons
  - Vertex & fragment processing load
- Benefit (importance to the image)
  - Size: larger objects contribute more to image
  - Accuracy: # of verts/polys, shading model, etc.
  - Priority: account for inherent importance
  - Eccentricity: peripheral objects harder to see
  - Velocity: fast-moving objects harder to see
  - Hysteresis: avoid flicker; use previous frame state

Given a fixed time budget, select LODs to maximize benefit within a cost constraint [Funkhouser & Sequin, SIGGRAPH 93]

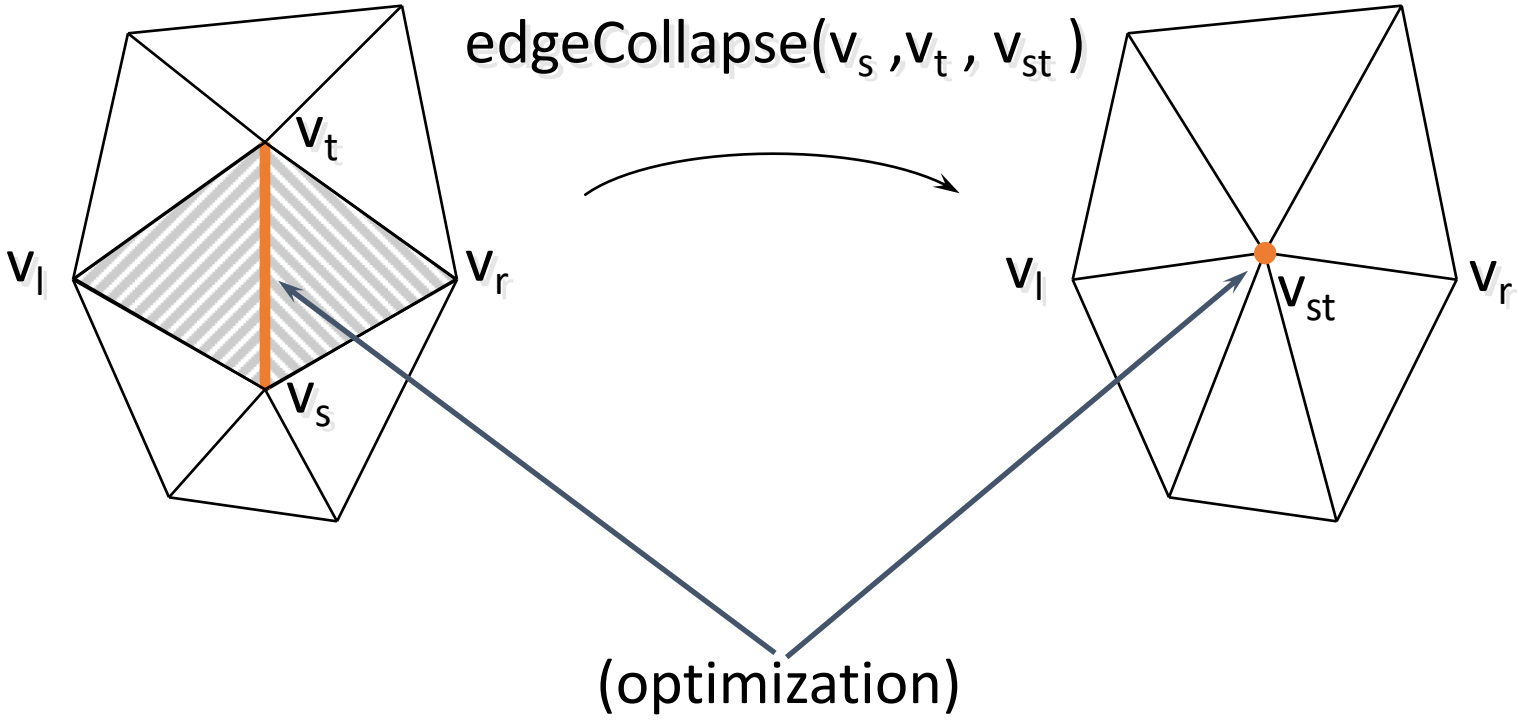
- Sort objects by benefit/cost ratio, pick in sorted order until budget is exceeded

Continuous Level of Detail

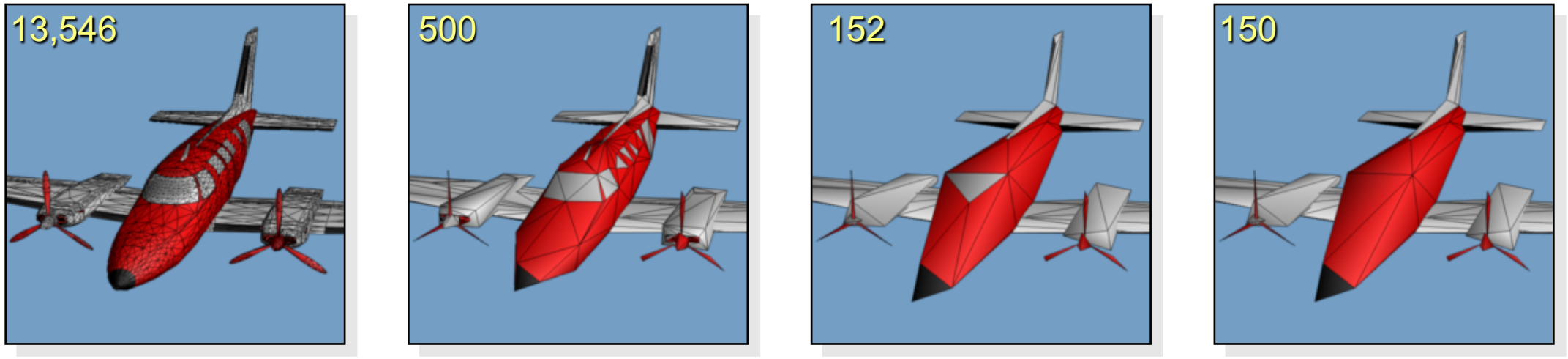
Create data structure from which a desired level of detail can be extracted *at run time*.

NEW MESH SIMPLIFICATION PROCEDURE

Apply sequence of edge collapses



SIMPLIFICATION PROCESS (ALSO INVERTIBLE)



$M' = M_n \xleftrightarrow{ecol_{n-1}} M_j \xleftrightarrow{ecol_i} M_1 \xleftrightarrow{ecol_0} M_0$

Continuous LOD: Advantages

Better granularity == better fidelity

- LOD is specified exactly, not chosen from a few pre-created options
- Thus objects use no more polygons than necessary, which frees up polygons for other objects
- Net result: better resource utilization, leading to better overall fidelity/polygon

Better granularity == smoother transitions

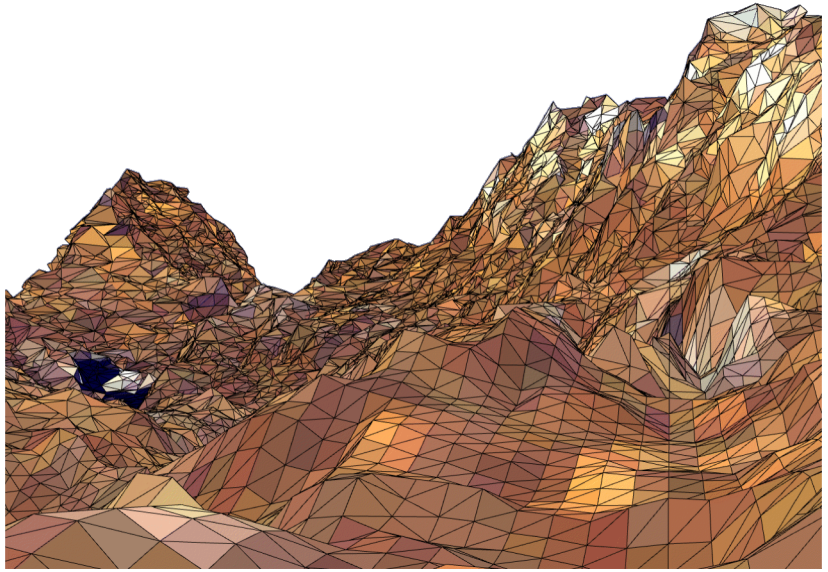
- Switching between traditional LODs can introduce visual "popping" effect
- Continuous LOD can adjust detail gradually and incrementally, reducing visual pops
  - Can even geomorph the fine-grained simplification operations over several frames to eliminate pops [Hoppe 96, 98]

Naturally leads to *view-dependent LOD*

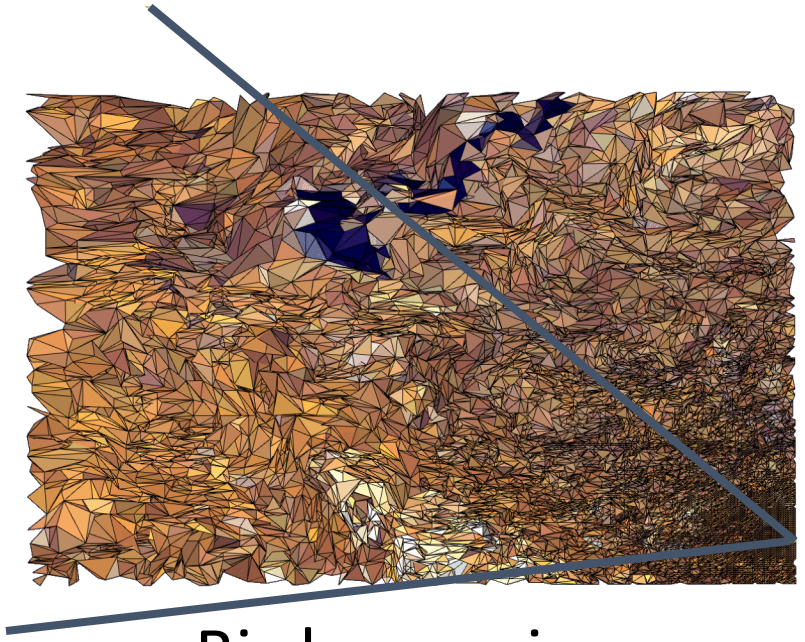
View-Dependent LOD: Examples

Use current view parameters to select best representation *for the current view*

Show nearby portions of object at higher resolution than distant portions



View from eyepoint



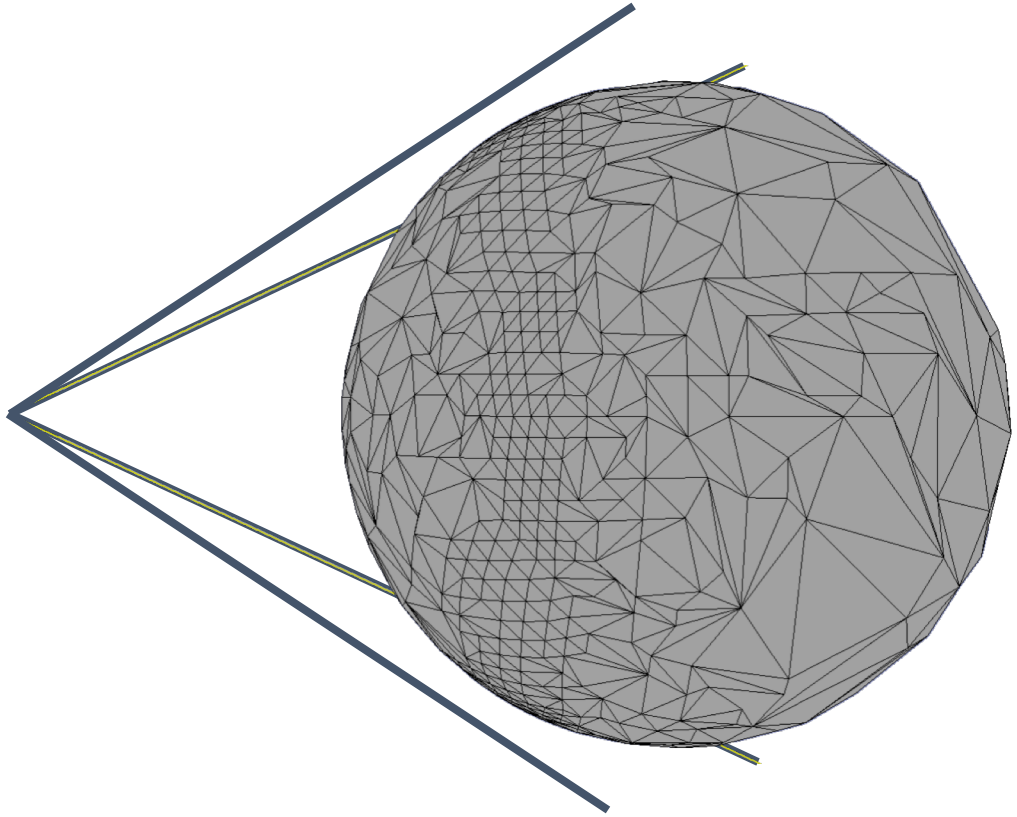
Birds-eye view

Demo Video

https://youtu.be/JBvyqu_wFeE

View-Dependent LOD: Examples

Show silhouette regions of object at higher resolution than interior regions



View-Dependent LOD

Advantages

- Enables drastic simplification of very large objects
- Better granularity
  - Allocates polygons where they are most needed
  - Enables even better overall fidelity

Disadvantages

- More complicated object data structures
- More expensive LOD selection