UCL

Visibility Techniques

V1.2.1

Anthony Steed

Based on slides from Celine Loscos (v1.1), Anthony Steed (v0.1), Several Others

UCL

Goals: The Visibility Problem

- The average number of polygons visible from a view point is much smaller than the model size
  - Select the (exact?) set of polygons from the model which are visible from a given viewpoint
- Review common techniques to do this
- Examine the suitability of different visibility algorithms for different problem domains

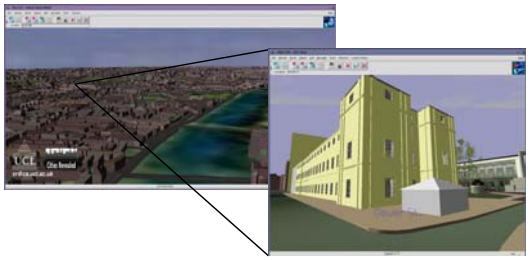
UCL

Overview

1. Motivation & Introduction
   - Definitions
   - Examples
2. View Frustum Culling
3. Occlusion Culling
   1. Image Space
   2. Object Spacej

3

UCL



4

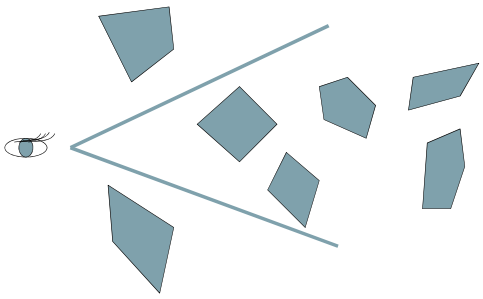
UCL

1. Visibility culling

- What is it for?
  - Avoid processing polygons which do not contribute to the rendered image
- We have three different cases of non-visible objects:
  - those outside the view volume (*view frustum/volume culling*)
  - those which are facing away from the user (*back face culling*)
  - those occluded behind other visible objects (*occlusion culling*)

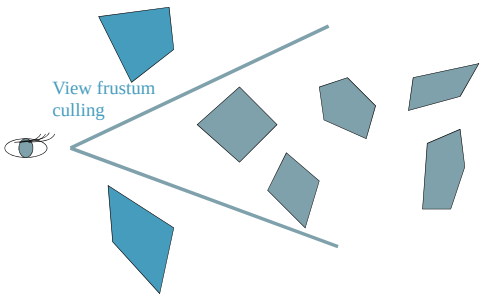
UCL

Visibility culling



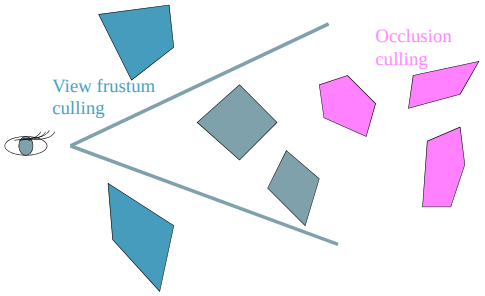
UCL

Visibility culling



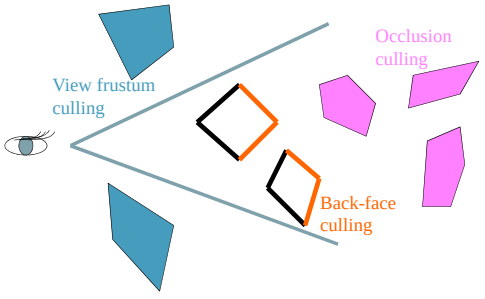
UCL

Visibility culling



UCL

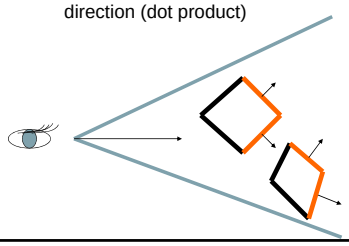
Visibility culling



UCL

Back-face culling

- Simplest version is to do it per polygon
  - Just test the normal of each polygon against the view direction (dot product)



UCL

3 types of visibility

- Exact visibility
  - Include all polygons at least partially visible and only those
- Approximate visibility
  - Include most of the visible polygons plus some hidden ones
- Conservative visibility
  - Include at least all the visible polygons plus maybe some additional hidden ones

UCL

Representation of the scene

- At pre-processing, the scene is placed in a spatial data structure that allows fast queries to the complex geometry

Space partitioning

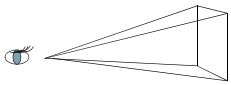
Hierarchical bounding volumes

Regular grid

UCL

2. View frustum culling

- Purpose: cull the polygons that are not inside the cone defined by
  - The viewpoint
  - The view direction
  - The two angles defining the field of view



- Easiest way
  - Test bounding box of object against the view volume (planes)

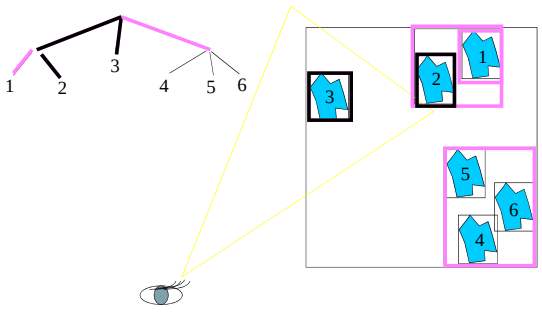
UCL

View frustum culling

- Compare the scene hierarchically against the view volume:
  - Test the root node against the view volume
  - If node is outside then stop and discard everything below it
  - If node is fully inside then render without clipping
  - Otherwise,
    - If leaf node render it,
    - Else recursively test each of its children

UCL

Example



UCL

View frustum culling

- Easy to implement
- A very fast computation
- Very effective result
- Therefore it is included in almost all current rendering systems

UCL

3. Occlusion culling

- By far the most complex of the three, both in terms of algorithmic complexity and in terms of implementation
- This is because it depends on the inter-relation of the objects
- Many different algorithms have been proposed, each one is better for different types of models

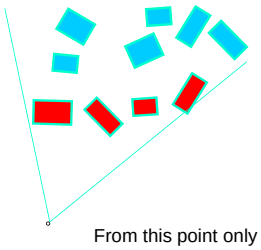
UCL

Occlusion culling

- Occlusion culling algorithms can be
  - Exact
  - Approximative
  - Conservative
- Difficulty:
  - Find as quickly as possible the ‘good’ occluders
- Different cases
  - View point / view cell /view volume
  - 2.5D /3D

UCL

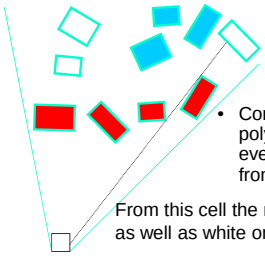
Point visibility



From this point only the red objects are visible

UCL

Cell visibility

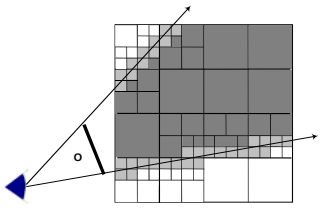


- Compute the set of all polygons visible from every possible viewpoint from a region (view-cell)

From this cell the red objects are visible as well as white ones

UCL

Hierarchical Test



UCL

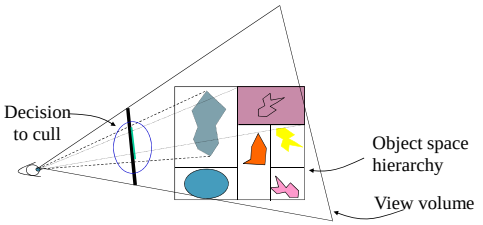
Algorithms

- Two types of approaches
  - Image space
  - Object space

UCL

3.1 Image-Space Methods

- Those where the decision to cull or render is done after projection (in image space)



UCL

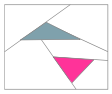
General outline of image-space methods

- During the in-order traversal of the scene hierarchy do:
  - compare each node against the view volume
  - if not culled, test node for occlusion
  - if still not culled, render objects/occluders **augmenting the image space occlusion**
- Most often done in 2 passes
  - render occluders – create occlusion structure
  - traverse hierarchy and classify/render

UCL

An image space representation of the occlusion information

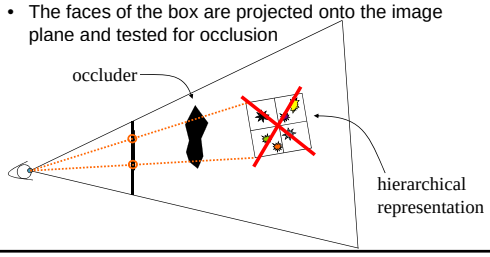
- Discrete
  - Z-hierarchy
  - Occlusion map hierarchy
- Continuous
  - BSP tree
  - Image space extends



UCL

Testing a Node for Occlusion

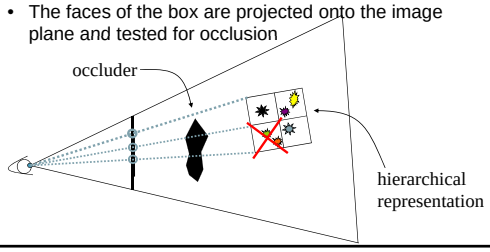
- If the box representing a node is not visible then nothing in it is either
- The faces of the box are projected onto the image plane and tested for occlusion



UCL

Testing a Node for Occlusion

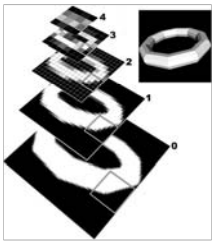
- If the box representing a node is not visible then nothing in it is either
- The faces of the box are projected onto the image plane and tested for occlusion



UCL

Visibility Culling using Hierarchical Occlusion Maps

- Idea: Building occlusion maps with different resolution
- Make use of the occluder fusion



UCL

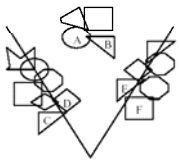
Construction of the Occlusion Map Hierarchy

- View-frustum culling
- Occluder selection
- Occluder rendering and depth estimation
- Building the hierarchical occlusion maps

UCL

Occluder Selection

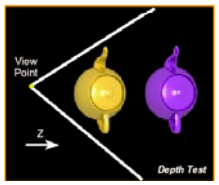
- Building the occluder database
  - Size
  - Redundancy
  - Rendering complexity
- Dynamic selection



UCL

Visibility Culling with HOM

- Consider a simple case, 2 stages:
  - Overlap test
  - Depth test



UCL

Occlusion map

- 2D array recording the opacity of occlusion

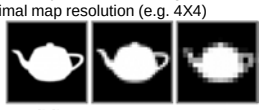
Rendered Image

Occlusion Map

UCL

Hierarchical Occlusion Maps

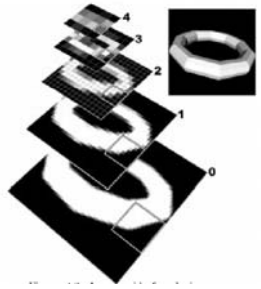
- Image pyramid
- Fast construction of the hierarchy
  - The occlusion map hierarchy is built by recursive filtering, which stops after reaching some minimal map resolution (e.g. 4X4)



- Desirable properties
  - Occluder fusion
  - Hierarchical overlap test
  - High-level opacity estimate

UCL

Image pyramid



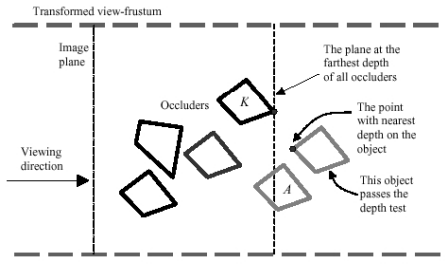
UCL

Visibility Culling with HOM

- Overlap test with occlusion maps
- Depth comparison
  - Single Z plane
  - Depth estimation buffer
    - Construction of depth estimation buffer
    - Conservative depth test

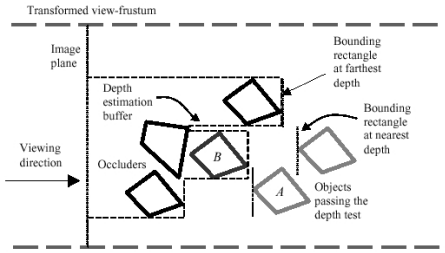
UCL

Single Z-plane



UCL

Depth estimation buffer



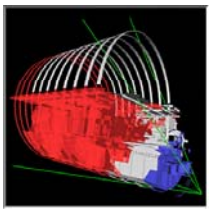
UCL

Remarks

- Approximate Visibility Culling
- Dynamic Environments
- Can use graphics hardware to generate occlusion map hierarchy, using texture map filtering.

UCL

Example of results



Blue – Objects selected as occulders

Gray – Objects not culled

Red – Objects culled

87% of model culled

UCL

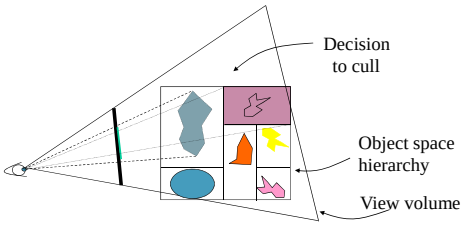
Discussion on image-space methods

- Advantages (not for all methods)
  - hardware acceleration
  - generality (anything that can be rendered can be used as an occluder)
  - robustness, ease of programming
  - option of approximate culling
- Disadvantages
  - hardware requirements
  - overheads

UCL

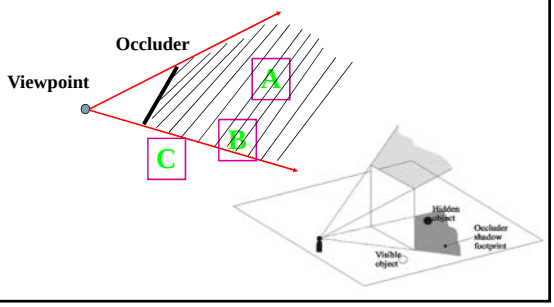
3.2 Object space methods

- The decision to cull is made in the object space



UCL

Occlusion Using Shadow Frustum



UCL

Assuming we can find good occluders

- For each frame
  - form shadow volumes from likely occluders
  - do view-volume cull and shadow-volume occlusion test in one pass across the spatial subdivision of the scene
  - each cell of the subdivision is tested for inclusion in view-volume and non-inclusion in each shadow volume

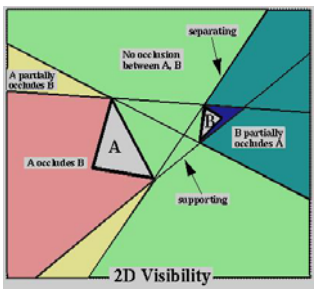
UCL

Temporally Coherent Visibility *(Coorg and Teller, SoCG 96)*

- Nice *theoretical* method
- Based on the idea that visibility changes when the view plane crosses specific planes (visual event)
- These planes partition space into regions of constant visibility – subset of an aspect graph
- Compute the critical planes dynamically using hierarchical structures, no need to pre-compute and store the entire arrangement

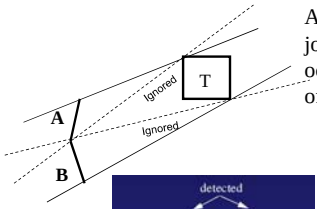
UCL

When does A occludes B ?

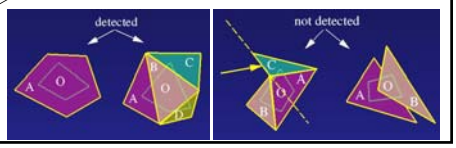


UCL

Occluder Fusion *(Coorg and Teller, I3D 97)*



Added the capability to join the effect of connected occluders, that is, a form of occluder fusion



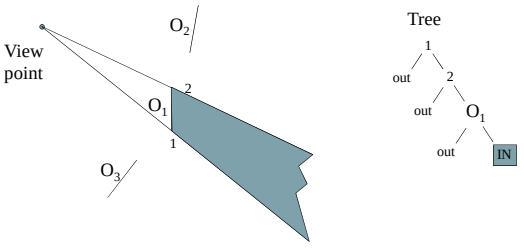
UCL

Occlusion Trees *(Bittner et al, CGI 98)*

- Just as before
  - scene represented by a hierarchy (kd-tree)
  - for each viewpoint
    - select a set of potential occluders
    - compare the scene hierarchy for occlusion
- However, unlike the previous method
  - the occlusion is accumulated into a binary tree
  - the scene hierarchy is compared for occlusion against the tree

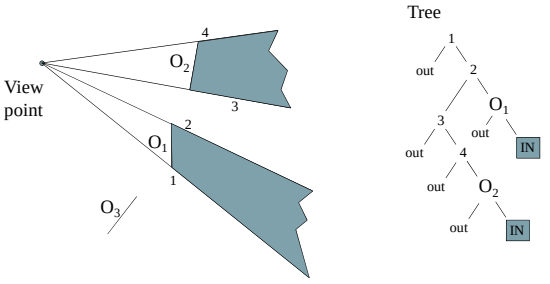
UCL

Create shadow volume of occluder 1



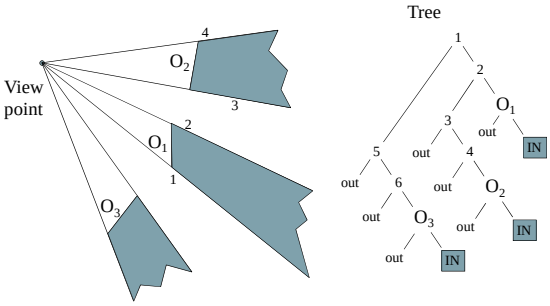
UCL

Insert occluder 2 and augment tree with its shadow volume



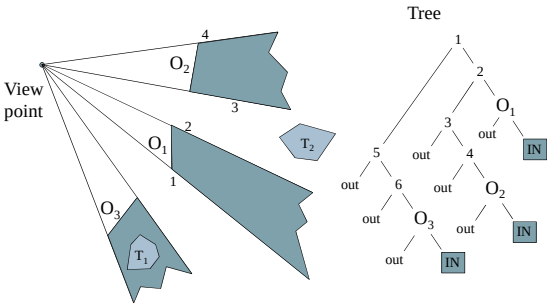
UCL

And so on until all occluders are added



UCL

Check occlusion of objects T_1 and T_2 by inserting them in tree



UCL

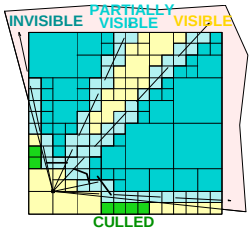
Occluder selection

- This is a big issue relevant to most occlusion culling algorithms but particularly to the last two
- At pre-processing
  - Identify likely occluders for a cell
    - they subtend a large solid-angle
  - Test likely occluders
    - use a sample of viewpoints and compute actual shadow volumes resulting
- At run time
  - locate the viewpoint in the hierarchy and use the occluders associated with that node

UCL

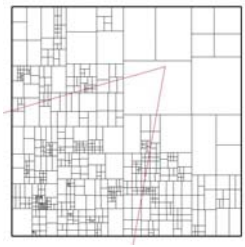
Using the occlusion tree

- Aggregation of occlusion
- Nodes classified as
  - Visible
  - Occluded
  - Partially visible
- Refinement of partially visible regions



UCL

Example

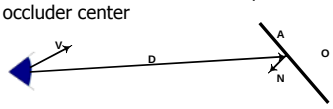


UCL

Metric for Comparing Occluder Quality

Occluder quality: $(-A * (N \cdot V)) / ||D||^2$

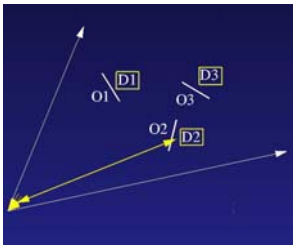
- A : the occluder's area
- N : normal
- V : viewing direction
- D : the distance between the viewpoint and the occluder center



UCL

Detail Occluders

- Smaller occluders that are close to objects could also be chosen



UCL

Cells and Portals*(Teller and Sequin, SIG 91)*

- Decompose space into convex cells
- For each cell, identify its boundary edges into two sets: opaque or portal
- Precompute visibility among cells
- During viewing (eg, walkthrough phase), use the precomputed potentially visible polygon set (PVS) of each cell to speed-up rendering

UCL

Three basic steps:

- 1 - The scene space is subdivided along its major opaque features
- 2 – cell to cell visibility is computed
- 3 – Culling is computed

- Steps 1 and 2 are pre-computations
- Step 3 is done during the simulation

UCL

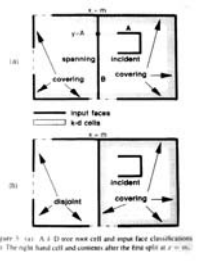
Assumptions

- The input scene has:
  - All faces are axial (orthogonal)
  - On a uniform grid (comparison of areas, lengths)
  - Convex cells

UCL

Step 1: Scene partitioning

- Cells are subdivided along wall faces
- Scene data stored in tree structure
  - Wall faces classified as:
    - Disjoint:
      if outside the cell, then discard
    - Spanning:
      if it fully partitions the cell
    - Covering:
      if on the cell boundary
    - Incident:
      otherwise

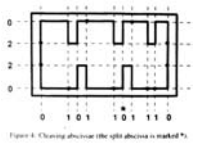


UCL

Algorithm

- At each step in subdivision:

1. Split cell on median spanning face (if it exists)
2. Otherwise split on face which cleaves minimal set of orthogonal faces
3. If cell has no interior faces, then STOP. This is a 'leaf node' of the tree.



UCL

Cell to cell visibility

- Unobstructed sight line between cells is any line which 'stabs' a portal sequence
- Portals are the transparent section of cell divisions

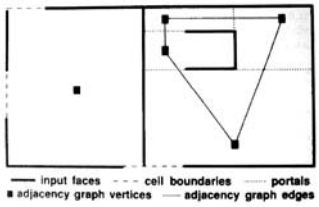


Figure 5: Subdivision, with portals and adjacency graph.

UCL

Choosing the portal sequences

- Candidates are chosen from neighbours in the subdivision

1. Divide endpoints of portal lists **L** & **R** depending on orientation
2. Sight line S exists if sets are linearly seperable

- $S \cdot L \geq 0 \quad \forall L \in \mathbf{L}$
- $S \cdot R \leq 0 \quad \forall R \in \mathbf{R}$

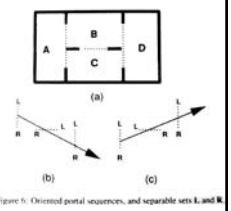


Figure 6: Oriented portal sequences, and separable sets L and R.

UCL

Recursive algorithm

Find_Visible_Cells (cell C, portal sequence P, visible cell set V)
V = V ∪ C
for each neighbour N of C
for each portal p connecting C and N
orient p from C to N
P' = P concatenate p
if *Stabbing_Line* (P') exists then
Find_Visible_Cells (N, P', V)

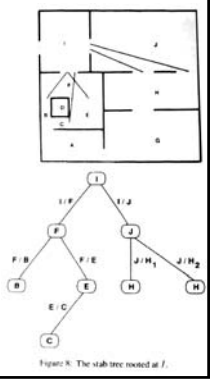


Figure 8: The stab tree rooted at I.

UCL

Eye to cell visibility

- Now we have the cells that an unconstrained observer can see from each particular cell.
- When the model is displayed interactively the view cone is known, and can be used to further reduce the number of visible cells, to those that can be seen from the current viewpoint
- Let C be the view cone, O the cell containing the observer, S the stab tree for that cell, and V the set of cells visible from O.
- We want to cull S & V against C

UCL

Overestimating

– Disjoint cell:
  - All cells intersecting viewcone.
  - Fails in Fig. a.

– Connected Component:
  - "In the cell adjacency graph, the visible cells must form a single connected component, each cell of which has a non-empty intersection with C".
  - Depth-first search from O in S. Every cell traversed must intersect interior of C.
  - Fails in Fig. b.

– Incident portals:
  - Refined: searching only through cells reachable via portals that intersect C's interior.
  - Fails in Fig. c.

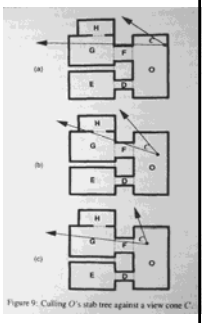


Figure 9: Culling O's stab tree against a view cone C.

UCL

Exact

- "For a cell to be visible, some portal sequence to that cell must admit a sightline that lies inside C and contains the view position".
- Using S, this can be solved relatively quickly. We search in S for a stabbing ray containing the observers view point, and lying within C.
- Works well for both Fig. a, b, c.

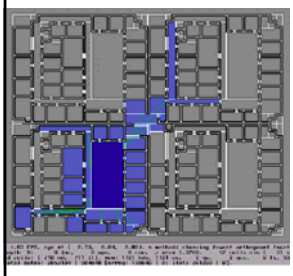
UCL

Consider frame-to-frame coherence during walkthrough

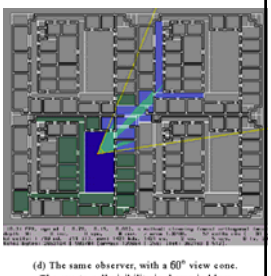
- Stab tree for current cell can be cached for additional speed up.
- Possible to prefetch polygons for cells adjacent to current one, since user must exit to one of them. Gives additional speedup, but with cost of memory.

UCL

Cells and Portals



(a) A source cell (dark blue), its cell-to-cell visibility (light blue), and stabbing lines (green).



(d) The same observer, with a 60° view cone. The eye-to-cell visibility is shown in blue; the exact visible area is shown in blue-green. The green cells have been dynamically culled.

UCL

Discussion on Object Space

- Visibility culling with large occluders
  - good for outdoor urban scenes where occluders are large and depth complexity can be very high
  - not good for general scenes with small occluders
- Cells and portals
  - gives excellent results IF you can find the cells and portals
  - good for interior scenes
  - identifying cells and portals is often done by hand
    - General polygons models "leak"

UCL

Conclusion

- 3 ways of reducing the number of polygons using visibility, with different level of precision
- Backface culing and view frustum culling are the easiest to implement
- Occlusion culling improves efficiency but is more complex to implement and use

UCL

Occlusion culling

- Properties
  - Depends on the choice of the occluders
  - The technique used needs to adapt with the model
  - Often from-point methods
  - Image-based method attractive
  - Attention needs to be taken to control the time spent on the computation so that the chosen technique IS an accelerating method

UCL

Occlusion culling

- Classifications
  - Point vs area visibility
  - Image-based vs object based
  - Exact vs conservative
  - online vs precomputed