Collision Detection on the GPU

Mike Donovan
CIS 665
Summer 2009

Overview

- Quick Background
- CPU Methods
- CULLIDE
- RCULLIDE
- QCULLIDE
- CUDA Methods

Background

- Need to find collisions for lots of reasons
    - Physics engines
    - Seeing if a projectile hits an object
    - Ray casting
    - Game engines
    - Etc…

Background

- Broad phase:
  - Looks at entire scene
  - Looks at proxy geometry (bounding shapes)
  - Determines if two objects *may* intersect
  - Needs to be very fast

Background

- Narrow phase:
    - Looks at pairs of objects flagged by broad phase
    - Looks at the actual geometry of an object
    - Determines if objects are truly intersecting
    - Generally slower

Background

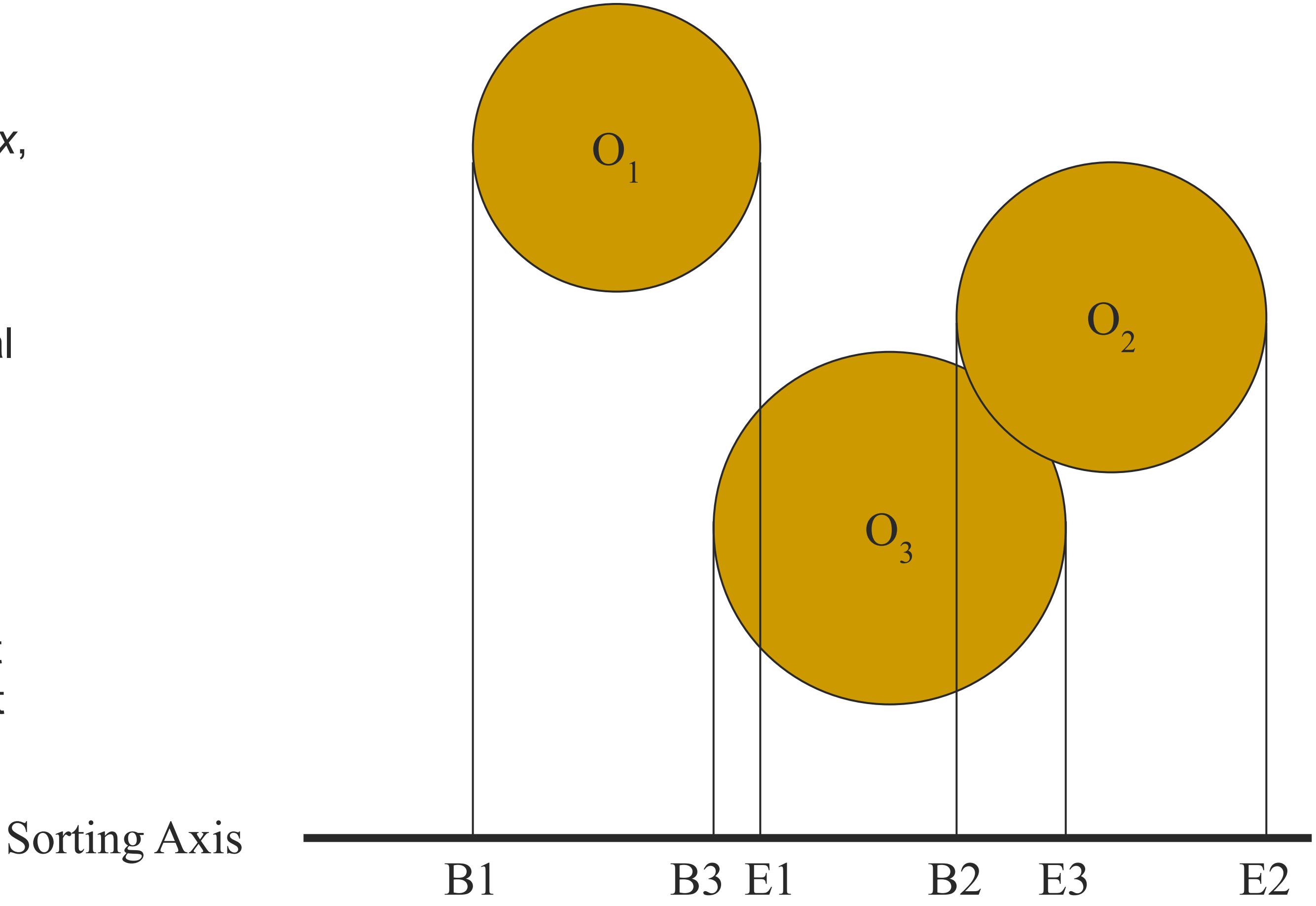
- Resolution
  - Compute forces according to the contact points returned from the narrow phase
  - Can be non trivial if there are multiple contact points
  - Returns resulting forces to be added to each body

CPU Methods

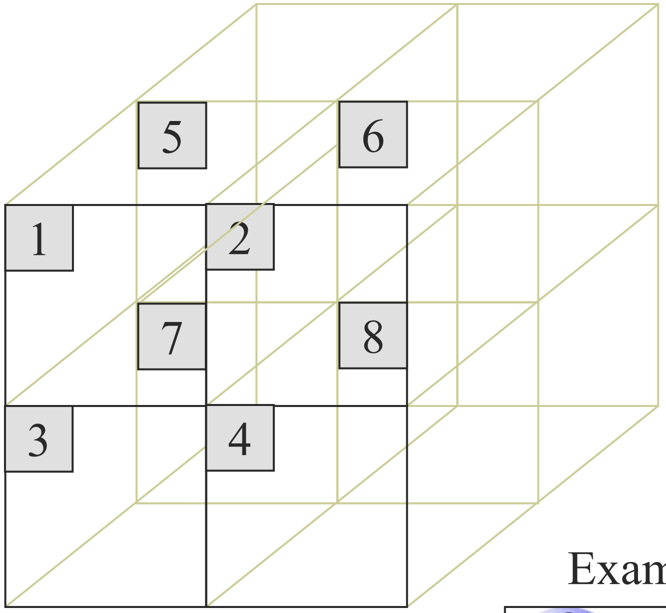
- Brute Force
  - Check every object against every other
    - N(N-1)/2 tests $O(N^2)$
- Sweep and Prune
    - Average case: $O(N \log N)$
    - Worst case: $O(N^2)$
- Spatial Subdivisions
    - Average case: $O(N \log N)$
    - Worst case: $O(N^2)$

Sweep and Prune

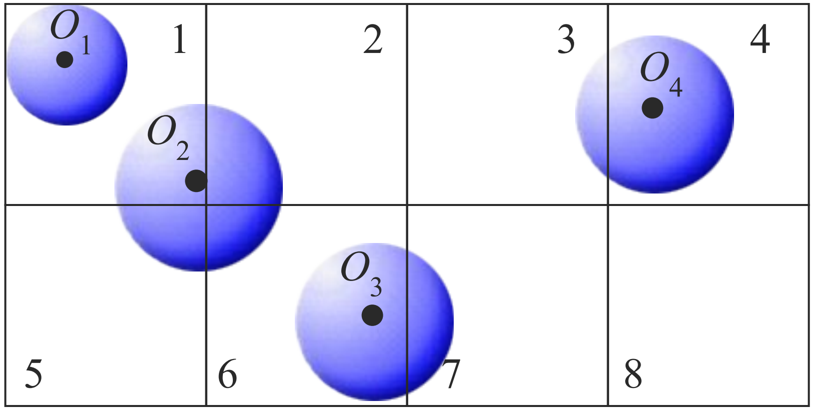
- Bounding volume is projected onto *x*, *y*, *z* axis
- Determine collision interval for each object $[b_i, e_i]$
- Two objects who's collision intervals do not overlap can not collide



Spatial Subdivisions



Example



Images from pg 699, 700 GPU Gems III

CULLIDE

- Came out of Dinesh’s group at UNC in 2003
- Uses graphics hardware to do a broad-narrow phase hybrid
- No shader languages

Outline

- Overview
- Pruning Algorithm
- Implementation and Results
- Conclusions and Future Work

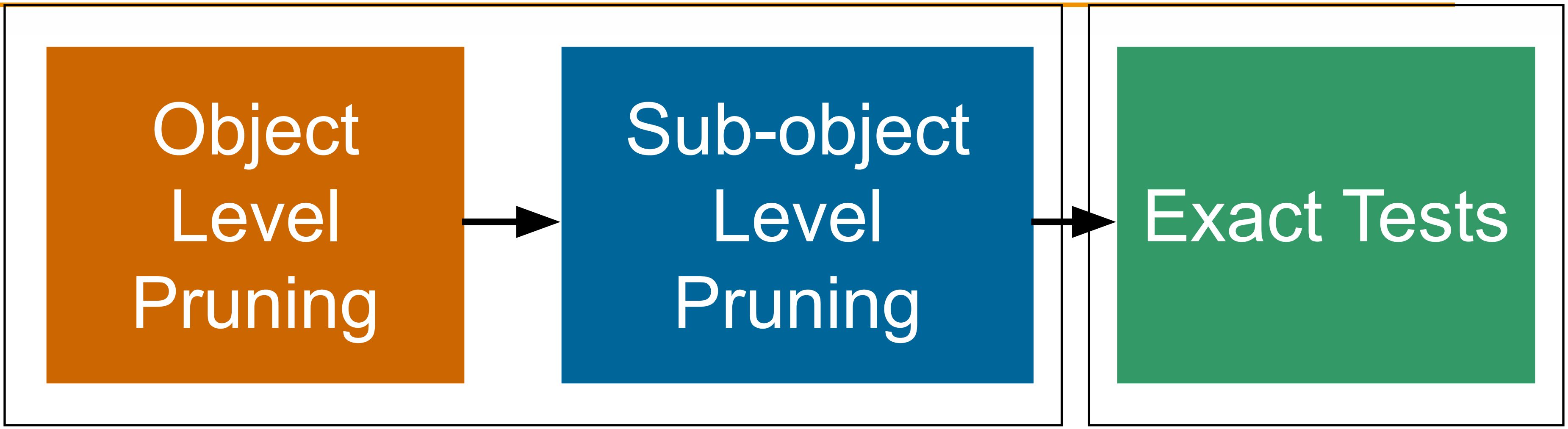
Outline

- **Overview**
- Pruning Algorithm
- Implementation and Results
- Conclusions and Future Work

Overview

- Potentially Colliding Set (PCS) computation
- Exact collision tests on the PCS

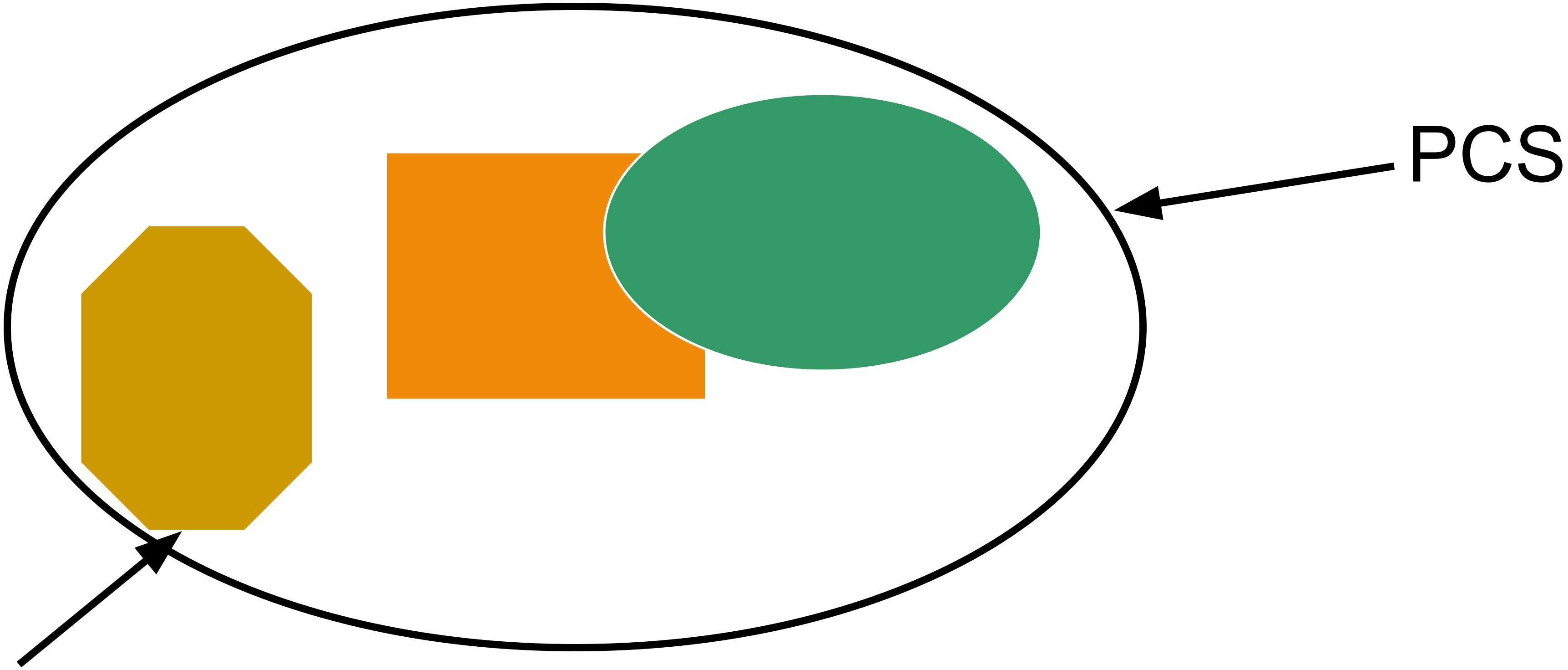
Algorithm



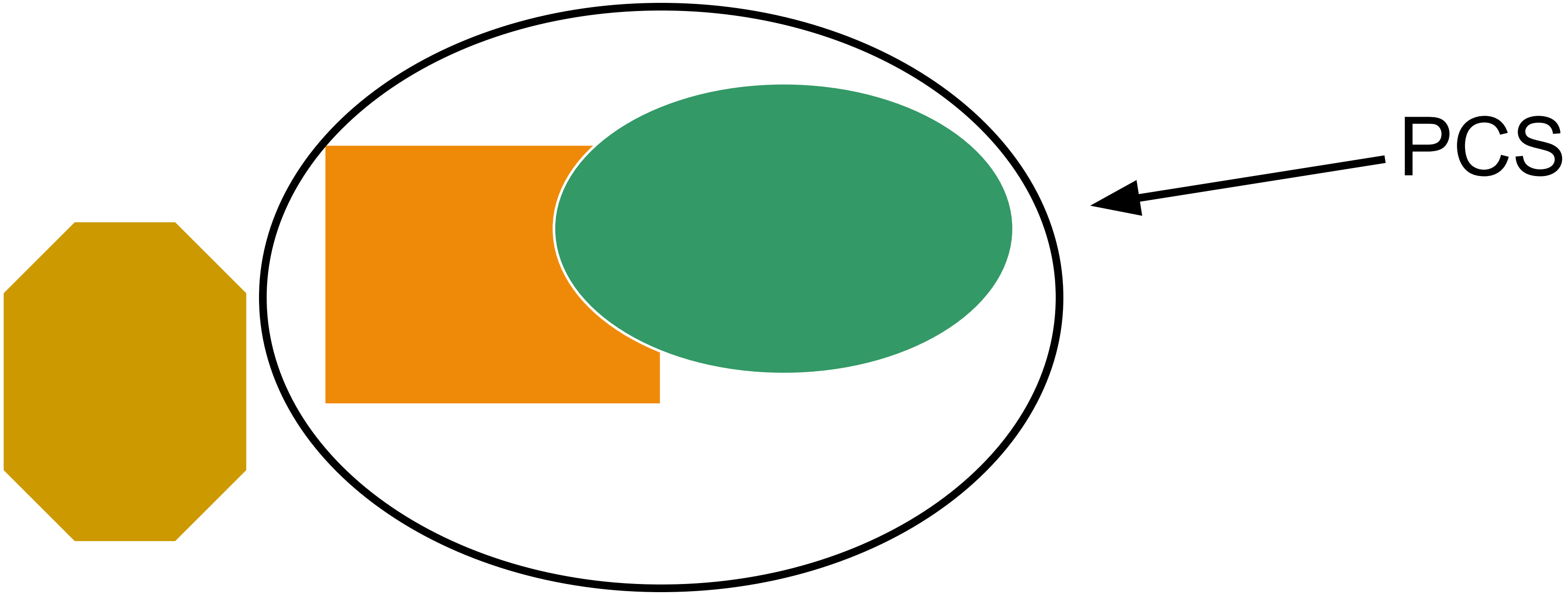
GPU based PCS computation

Using CPU

Potentially Colliding Set (PCS)



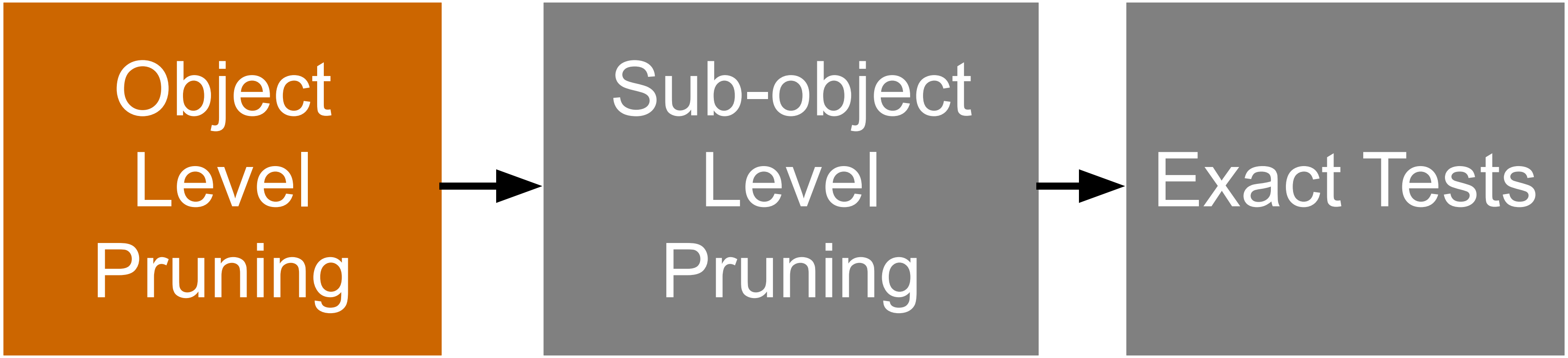
Potentially Colliding Set (PCS)



Outline

- Problem Overview
- Overview
- **Pruning Algorithm**
- Implementation and Results
- Conclusions and Future Work

Algorithm



Object Level Pruning

Sub-object Level Pruning

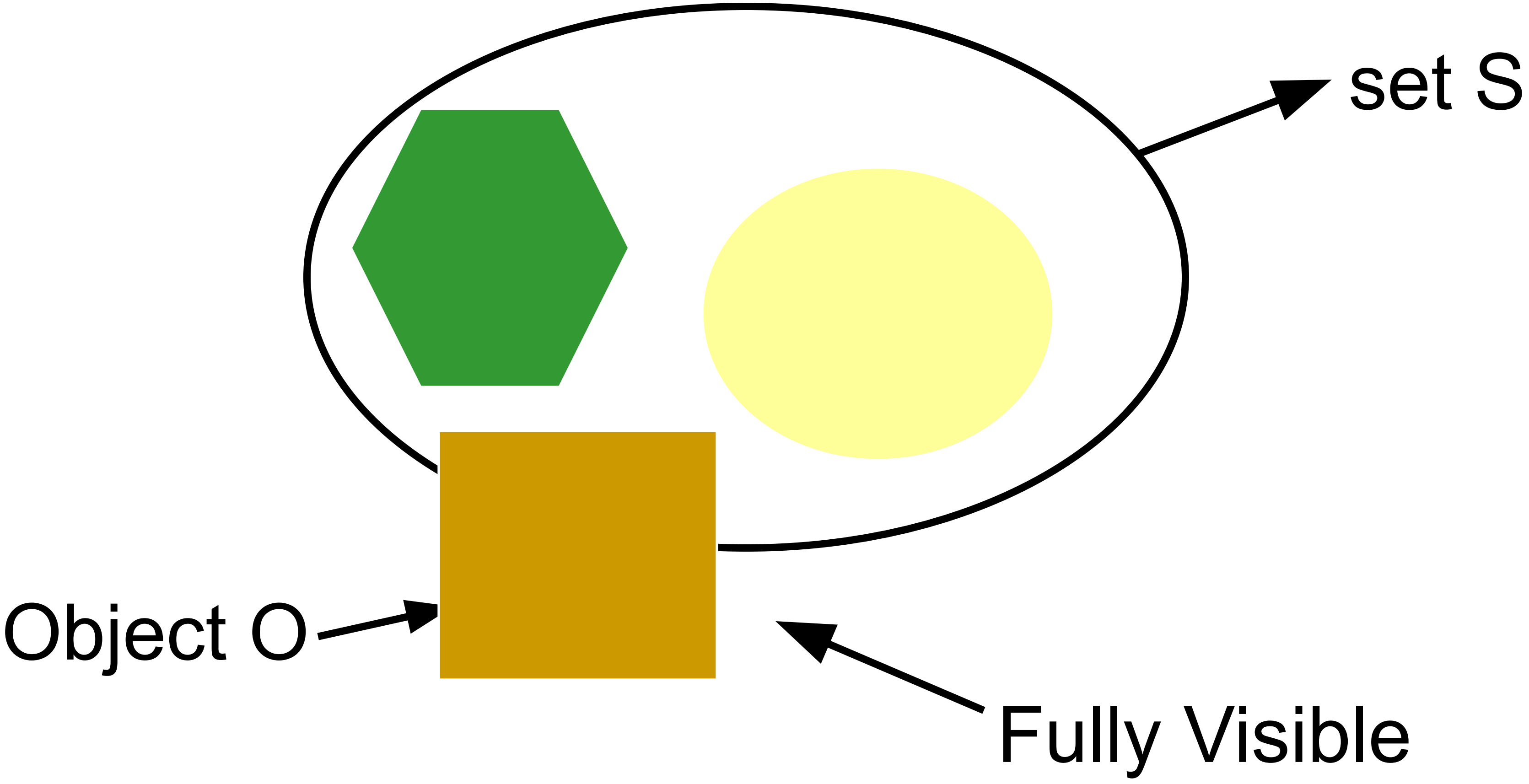
Exact Tests

Visibility Computations

Lemma 1: *An object O does not collide with a set of objects S if O is fully visible with respect to S*

- Utilize visibility for PCS computation

Collision Detection using Visibility Computations



PCS Pruning

Lemma 2: *Given n objects $O_1, O_2, \ldots, O_n$, an object O_i does not belong to PCS if it does not collide with $O_1, \ldots, O_{i-1}, O_{i+1}, \ldots, O_n$*

- Prune objects that do not collide

PCS Pruning

$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n$

PCS Pruning

$o_1 \quad o_2 \quad \ldots \quad o_{i-1} \; o_i$

PCS Pruning

$O_i \; O_{i+1} \; \ldots \; O_{n-1} \; O_n$

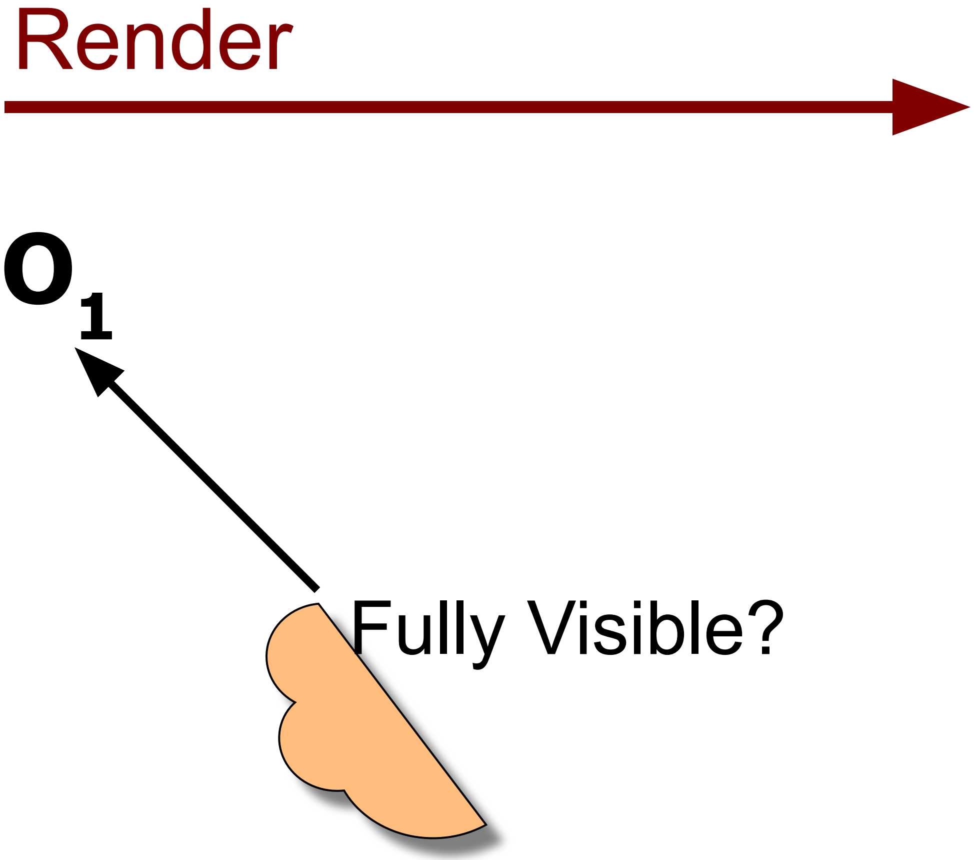
PCS Computation

- Each object tested against all objects but itself
- Naive algorithm is $O(n^2)$
- Linear time algorithm
  - Uses two pass rendering approach
  - Conservative solution

PCS Computation: First Pass

$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n$

PCS Computation: First Pass

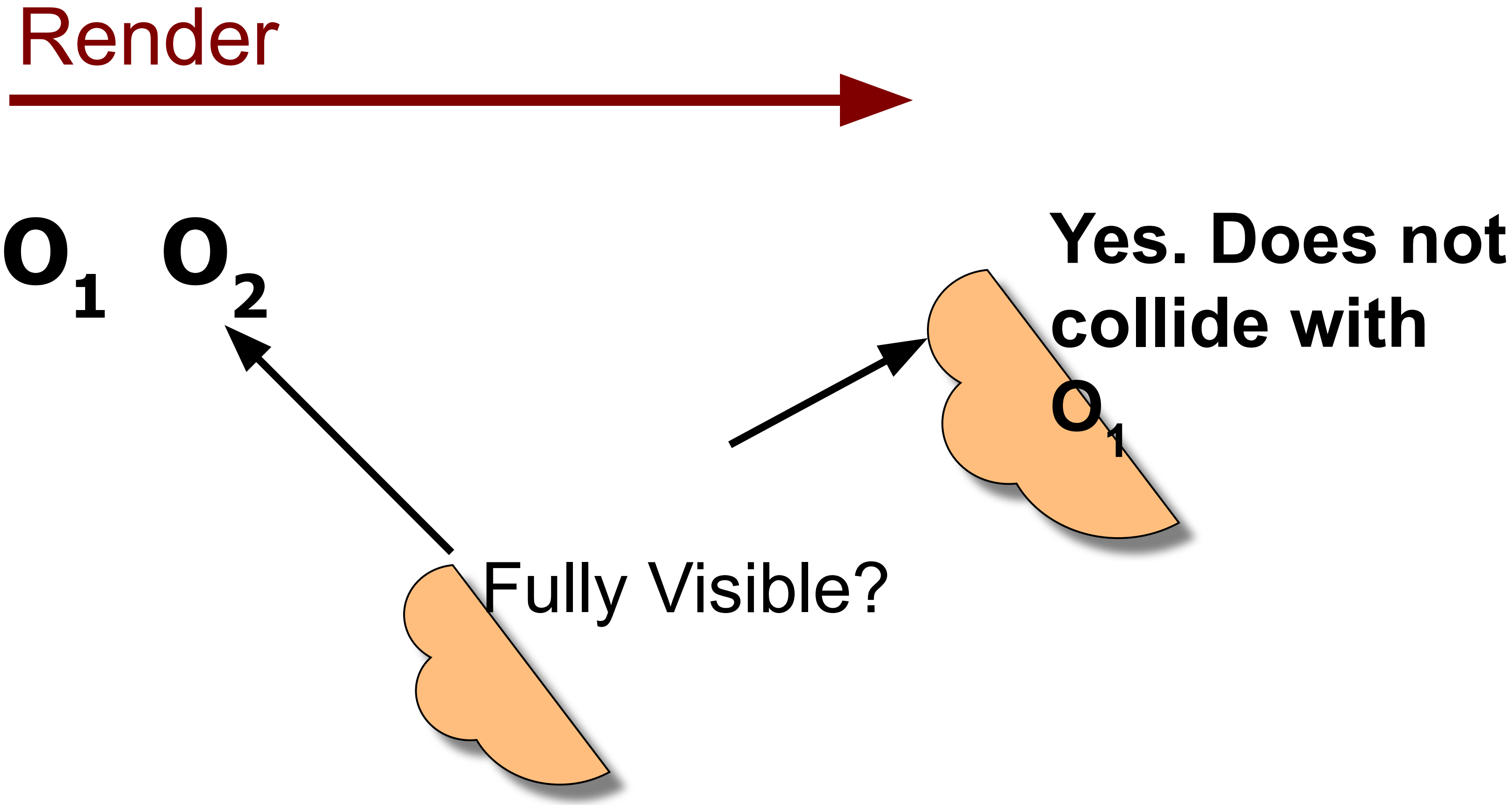


Render

O_1

Fully Visible?

PCS Computation: First Pass



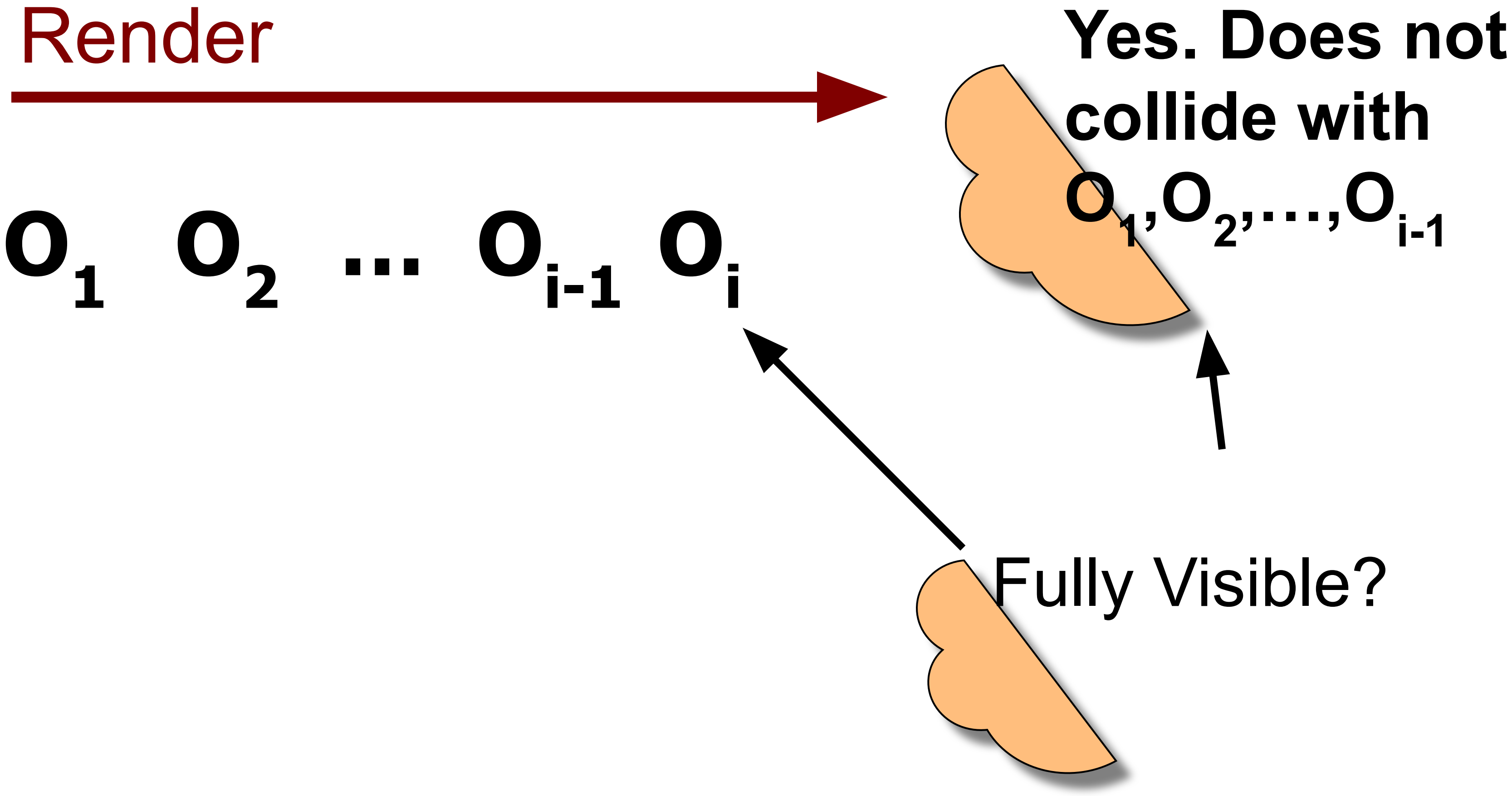
Render

O_1 O_2

Fully Visible?

Yes. Does not collide with O_1

PCS Computation: First Pass



Render

O_1 O_2 … O_{i-1} O_i

Yes. Does not collide with $O_1, O_2, \ldots, O_{i-1}$

Fully Visible?

PCS Computation: First Pass

Render

O_1 O_2 … O_{i-1} O_i O_{i+1} … O_{n-1} O_n

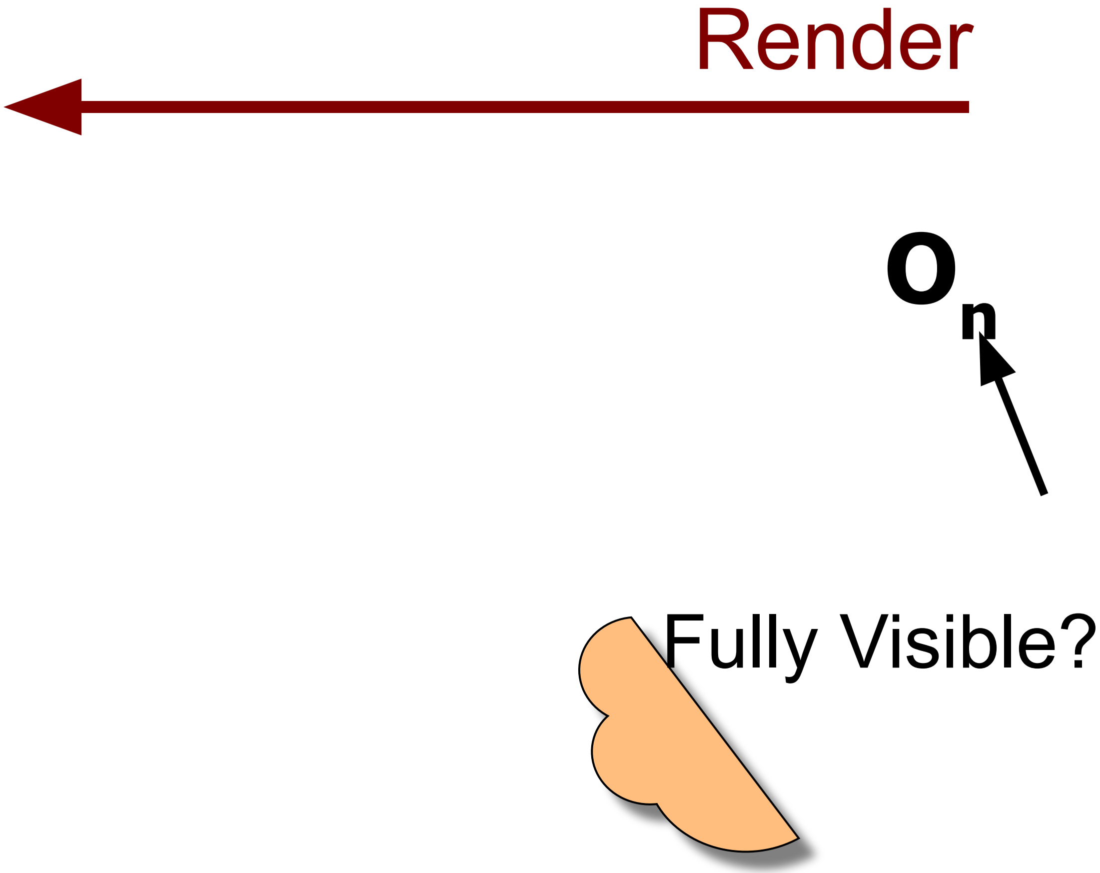
Fully Visible?

PCS Computation: Second Pass


Render

$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n$

PCS Computation: Second Pass

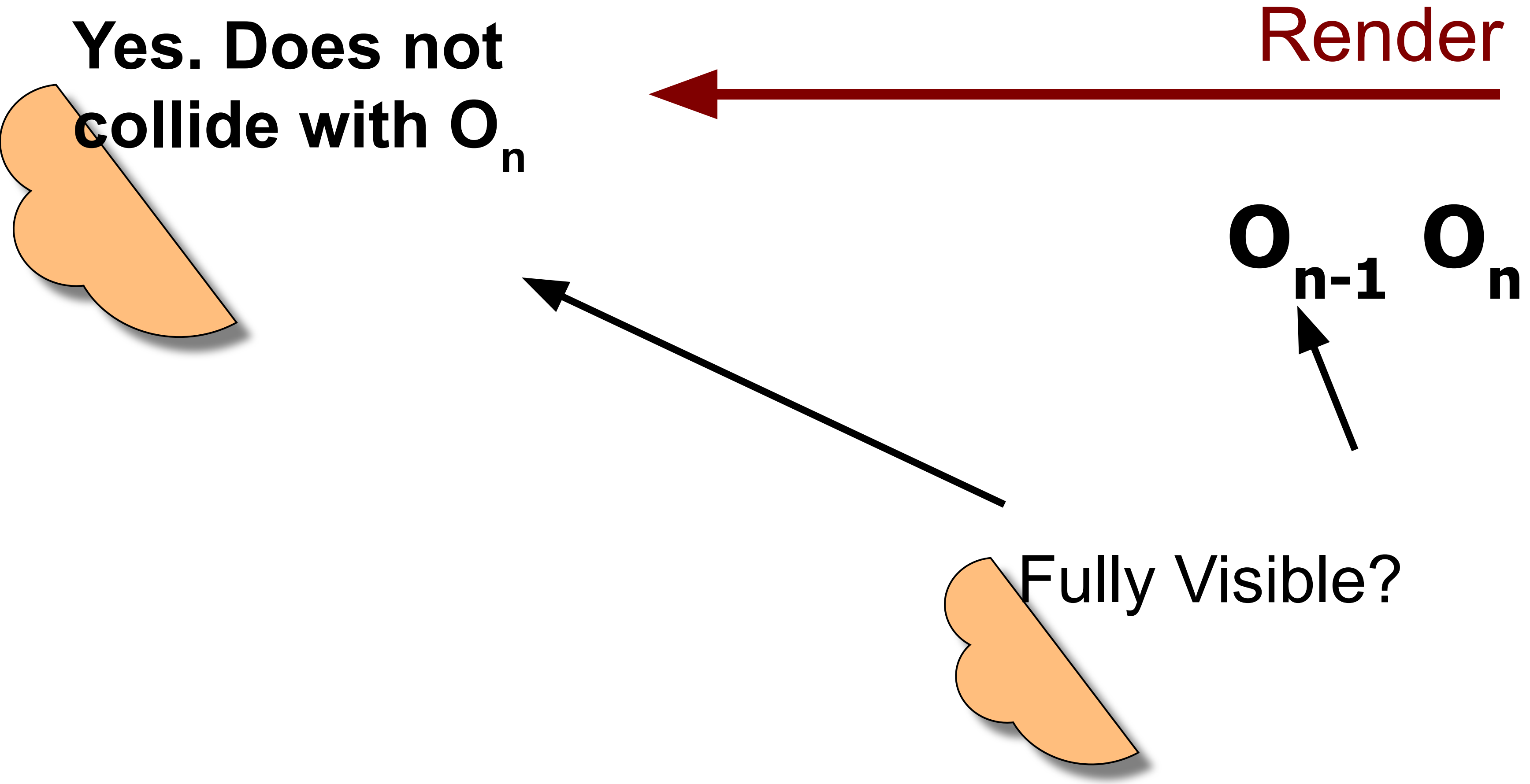


Render

O_n

Fully Visible?

PCS Computation: Second Pass



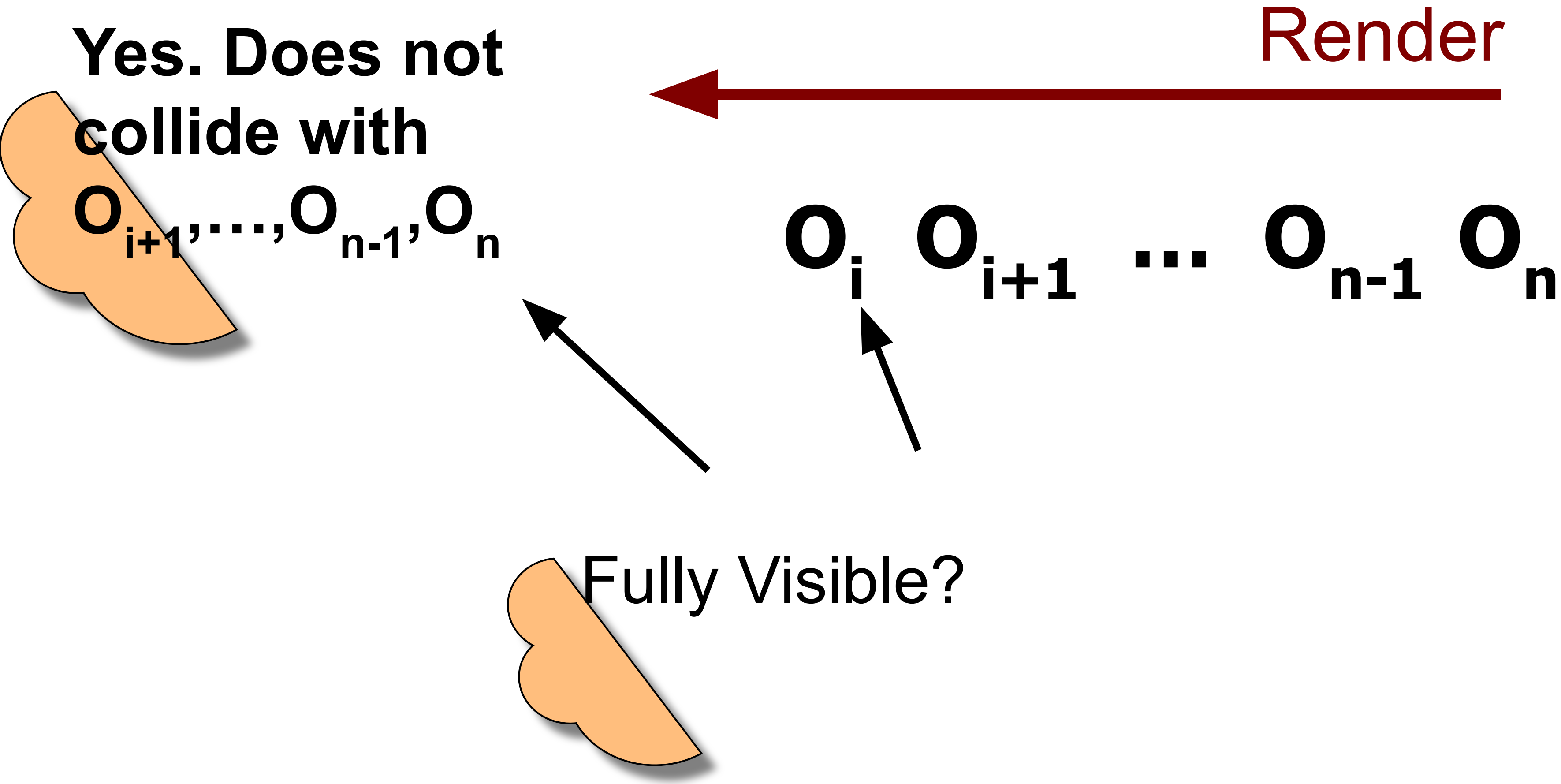
Yes. Does not collide with O_n

Render

O_{n-1} O_n

Fully Visible?

PCS Computation: Second Pass



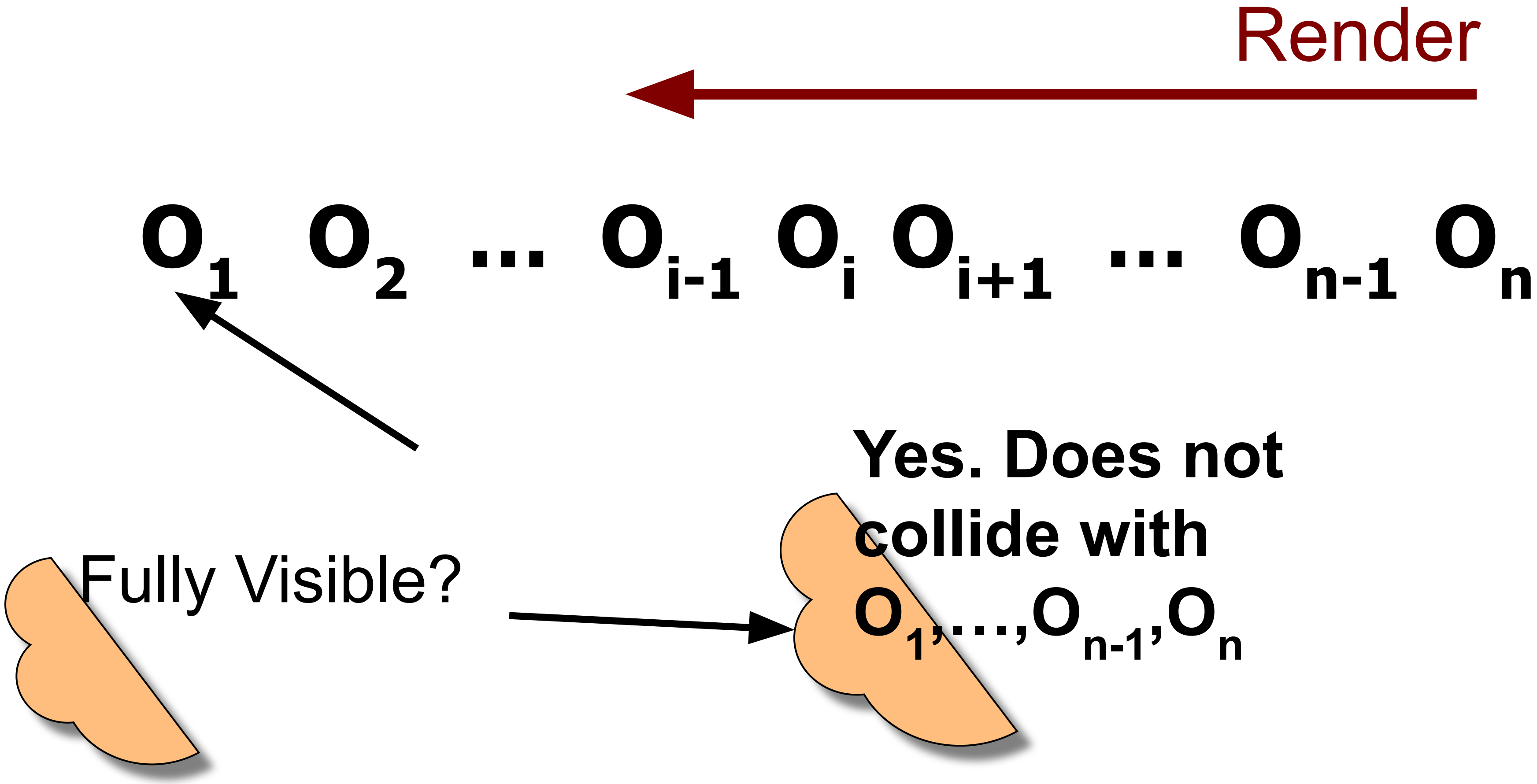
Render

Yes. Does not collide with $O_{i+1},\ldots,O_{n-1},O_n$

$O_i \quad O_{i+1} \quad \ldots \quad O_{n-1} \quad O_n$

Fully Visible?

PCS Computation: Second Pass



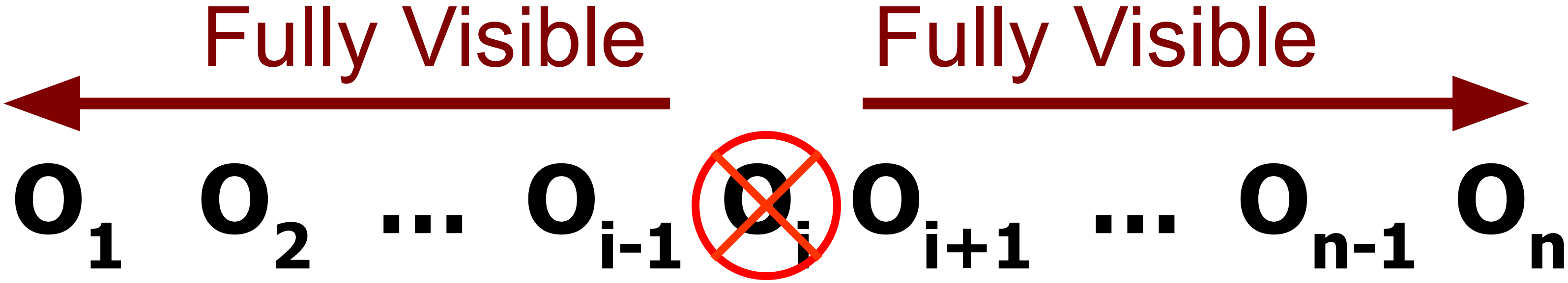
Render

O_1 O_2 … O_{i-1} O_i O_{i+1} … O_{n-1} O_n

Fully Visible?

Yes. Does not collide with $O_1, \ldots, O_{n-1}, O_n$

PCS Computation



Fully Visible Fully Visible

O_1 O_2 … O_{i-1} $\xcancel{O_i}$ O_{i+1} … O_{n-1} O_n

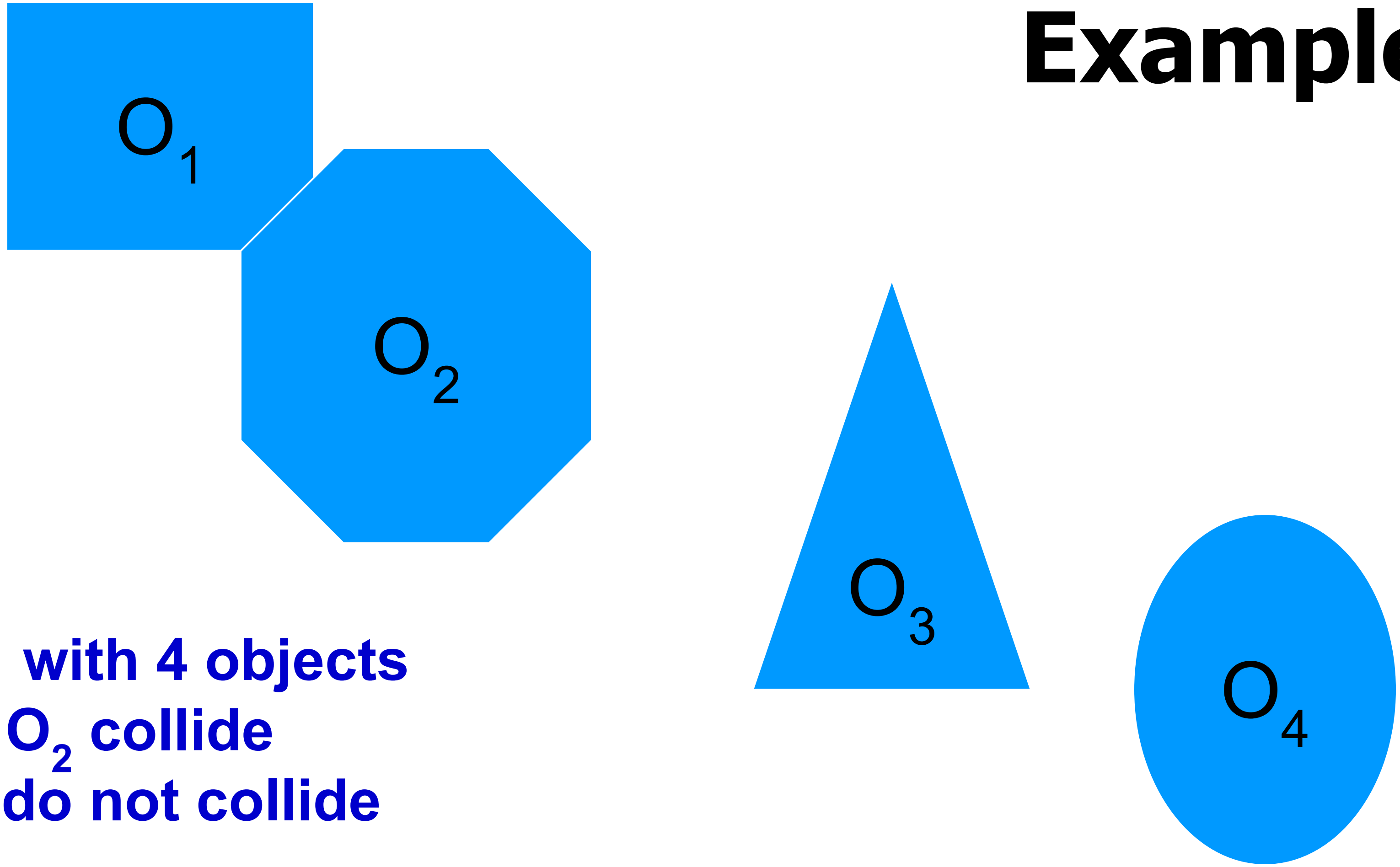
PCS Computation

$$O_1\ \cancel{O_2}\ O_3 \ldots O_{i-1}\ \cancel{O_i}\ O_{i+1} \ldots O_{n-2}\ O_{n-1}\ \cancel{O_n}$$

$$\downarrow$$

$$O_1\quad O_3\quad \ldots\quad O_{i-1}\ O_{i+1}\quad \ldots\quad O_{n-1}$$

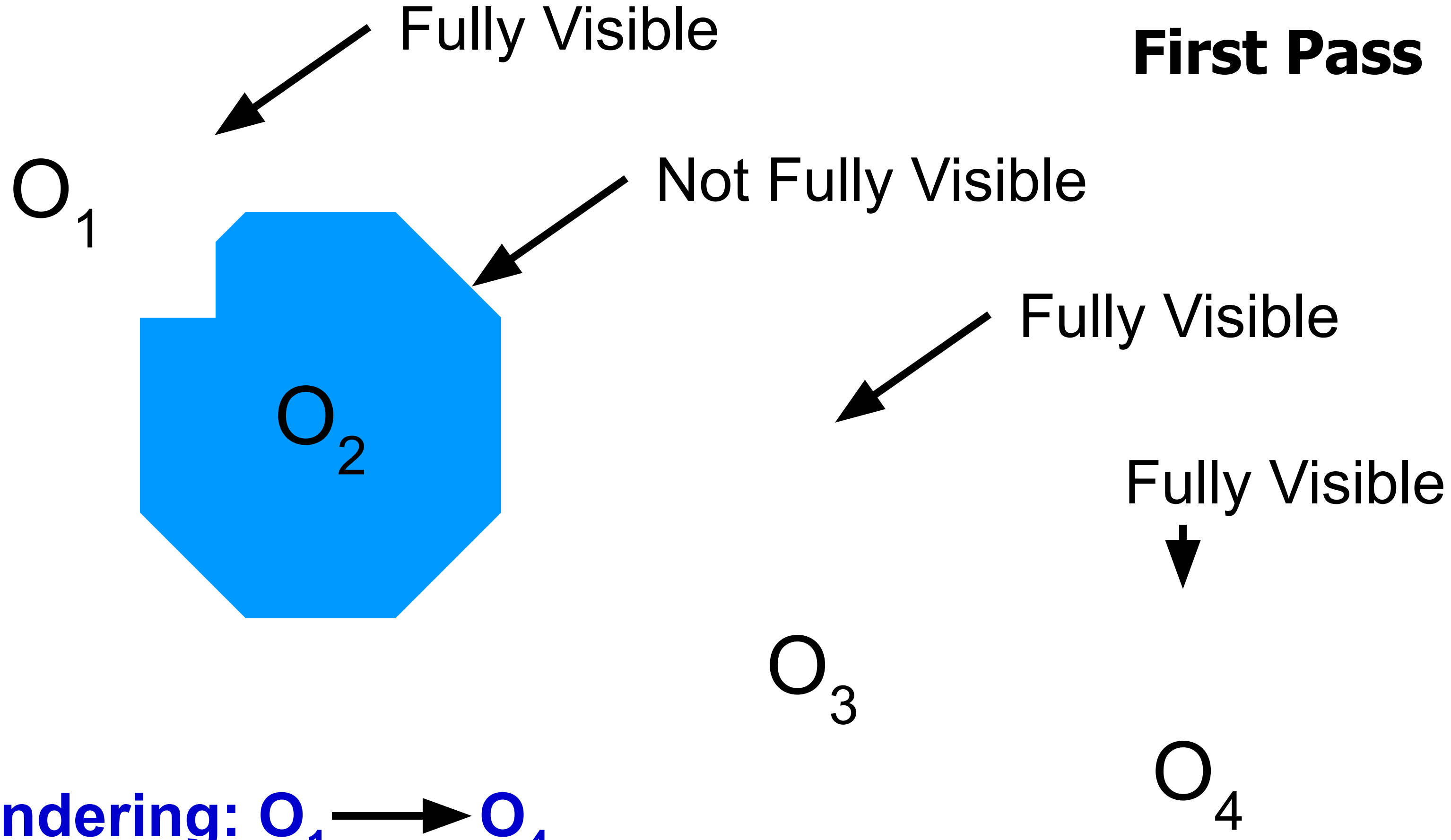
Example



Scene with 4 objects
O_1 and O_2 collide
O_3, O_4 do not collide

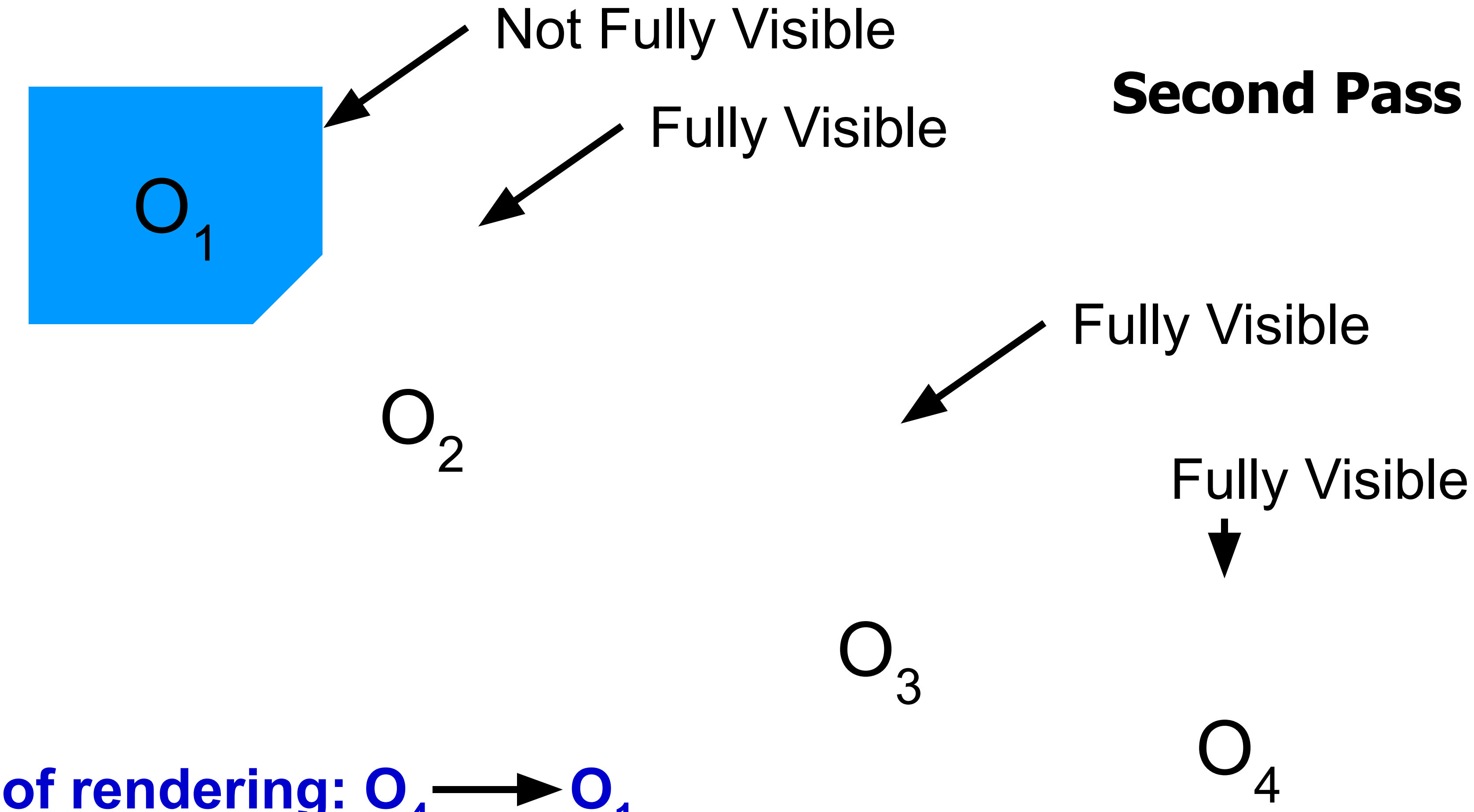
Initial PCS = { O_1, O_2, O_3, O_4 }

First Pass



Order of rendering: $O_1 \longrightarrow O_4$

Second Pass



Not Fully Visible

Fully Visible

Fully Visible

Fully Visible

O_1

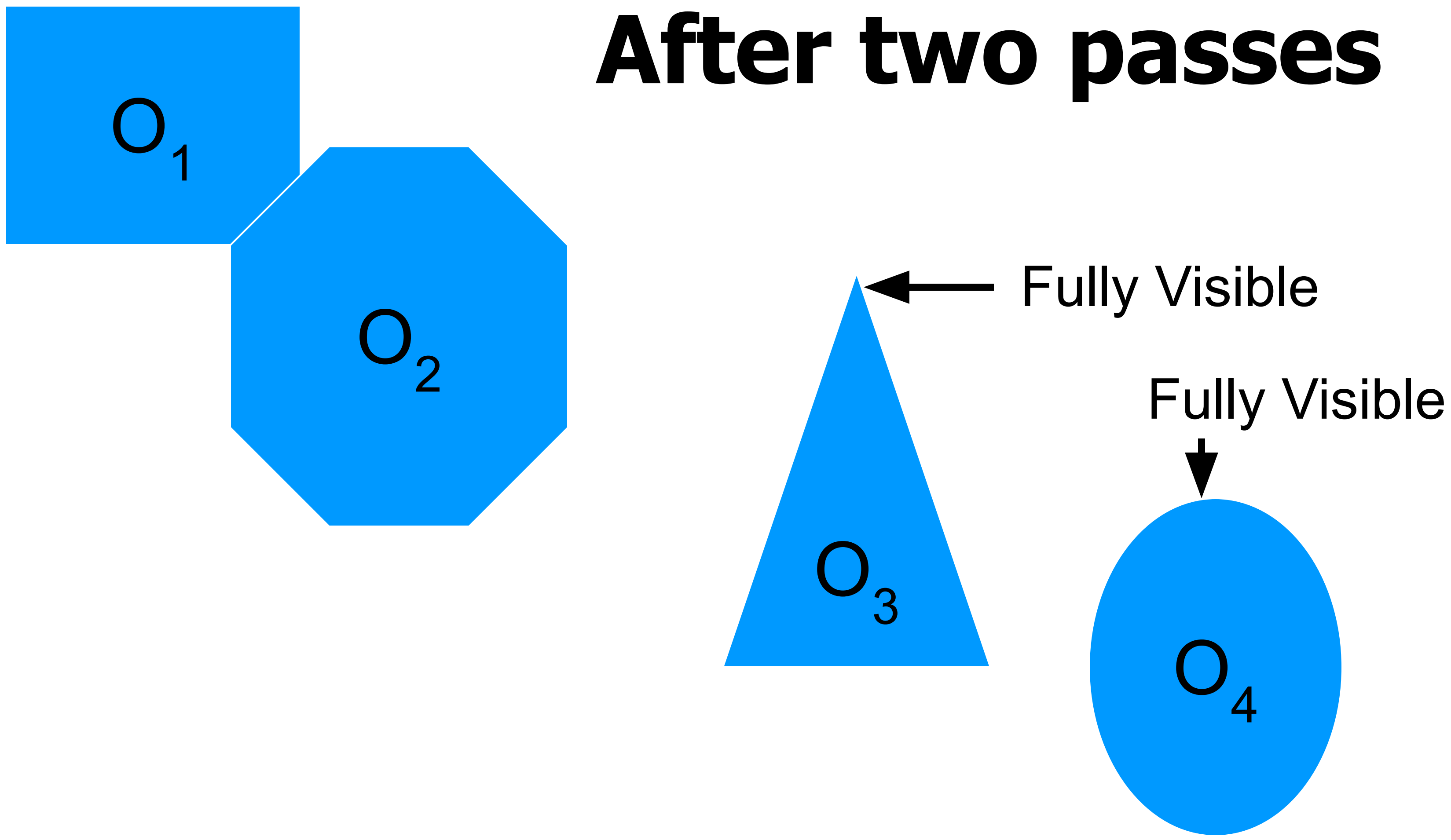
O_2

O_3

O_4

Order of rendering: $O_4 \rightarrow O_1$

After two passes



O_1

O_2

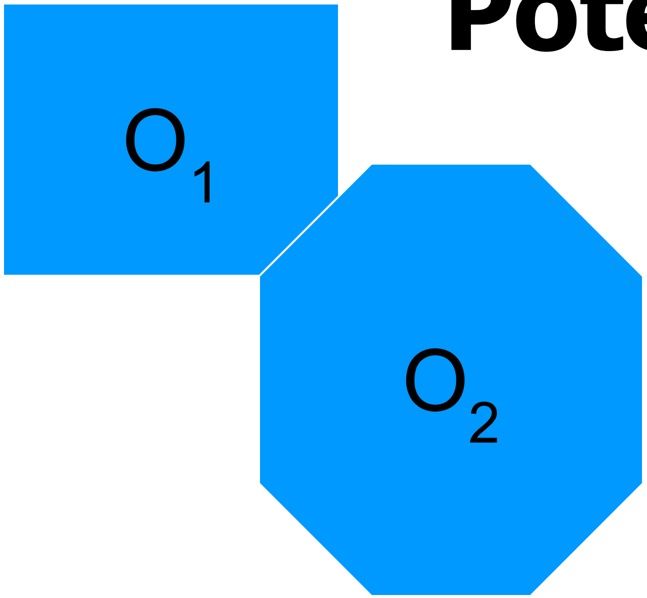
O_3

O_4

Fully Visible

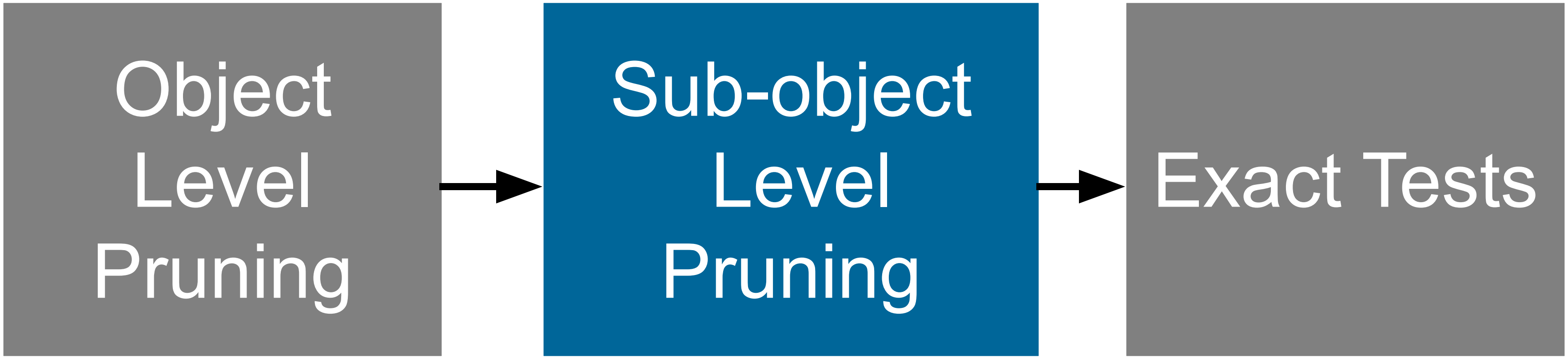
Fully Visible

Potential Colliding Set



PCS =$\{O_1,O_2\}$

Algorithm



Object Level Pruning

Sub-object Level Pruning

Exact Tests

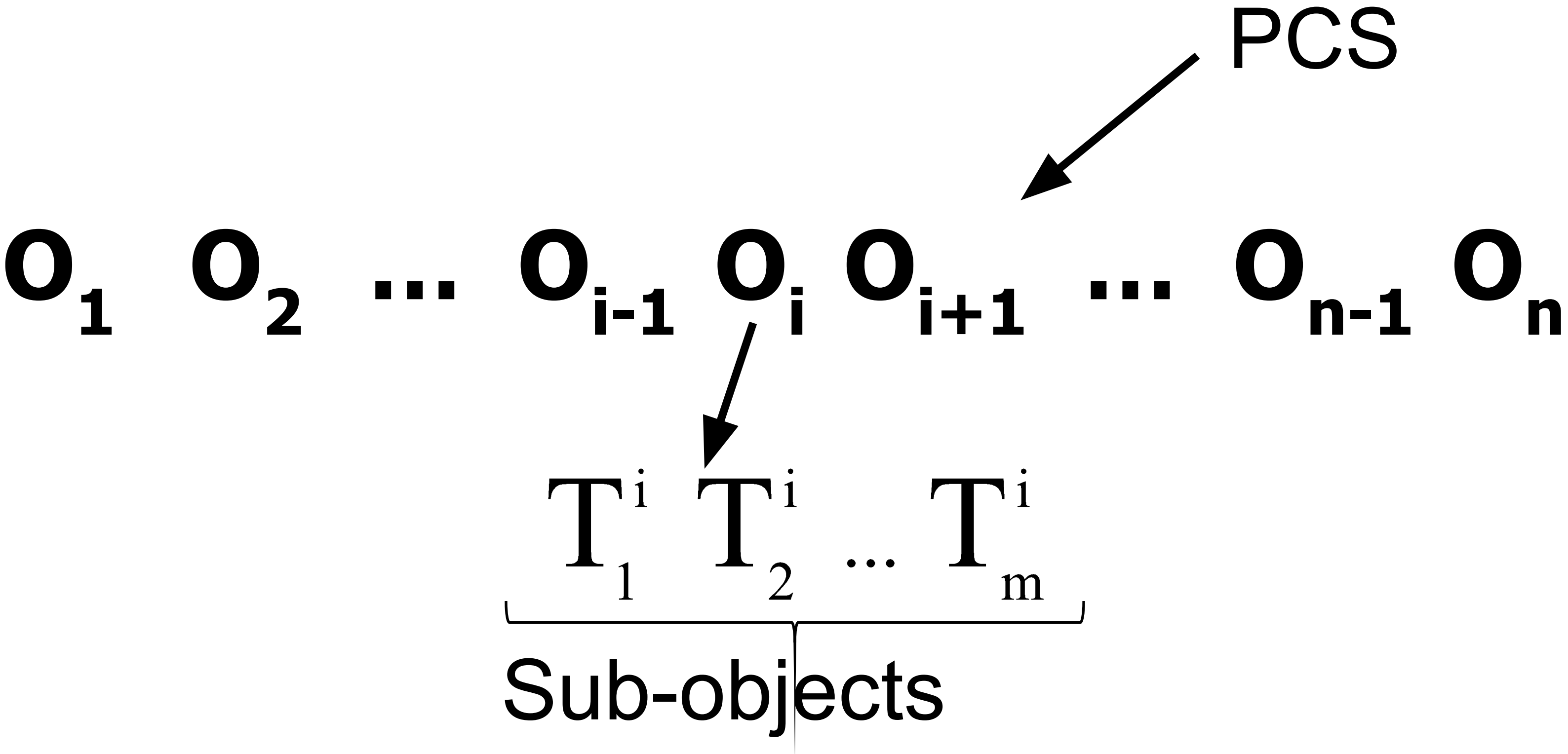
Overlap Localization

- Each object is composed of sub-objects
- We are given n objects $O_1,\ldots,O_n$
- Compute sub-objects of an object O_i that overlap with sub-objects of other objects

Overlap Localization

- **Our solution**
  - **Test if each sub-object of O_i overlaps with sub-objects of $O_1,..O_{i-1}$**
  - **Test if each sub-object of O_i overlaps with sub-objects of $O_{i+1},...,O_n$**
- **Linear time algorithm**
- **Extend the two pass approach**

Overlap Localization



PCS

$$O_1 \quad O_2 \quad \dots \quad O_{i-1} \; O_i \; O_{i+1} \quad \dots \quad O_{n-1} \; O_n$$

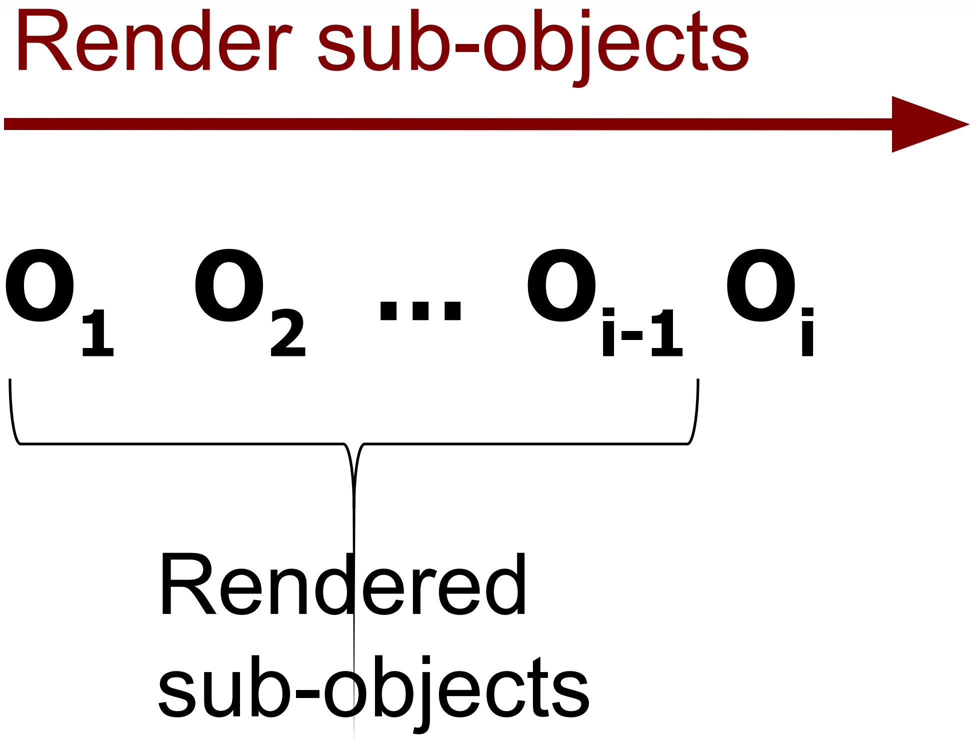
$$\underbrace{T_1^i \; T_2^i \; \dots \; T_m^i}_{\text{Sub-objects}}$$

Overlap Localization: First Pass

Render sub-objects

O_1 O_2 … O_{i-1} O_i O_{i+1} … O_{n-1} O_n

Overlap Localization: First Pass

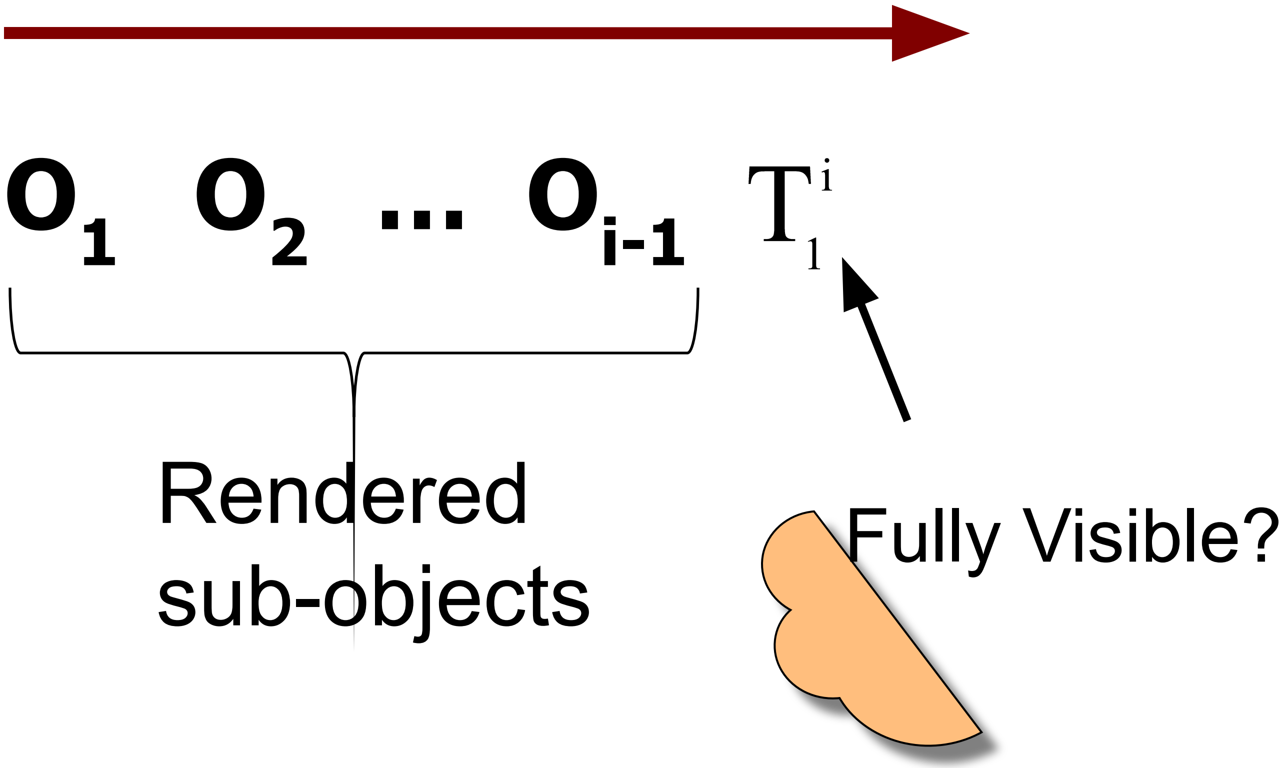


Render sub-objects

O_1 O_2 … O_{i-1} O_i

Rendered sub-objects

Overlap Localization: First Pass

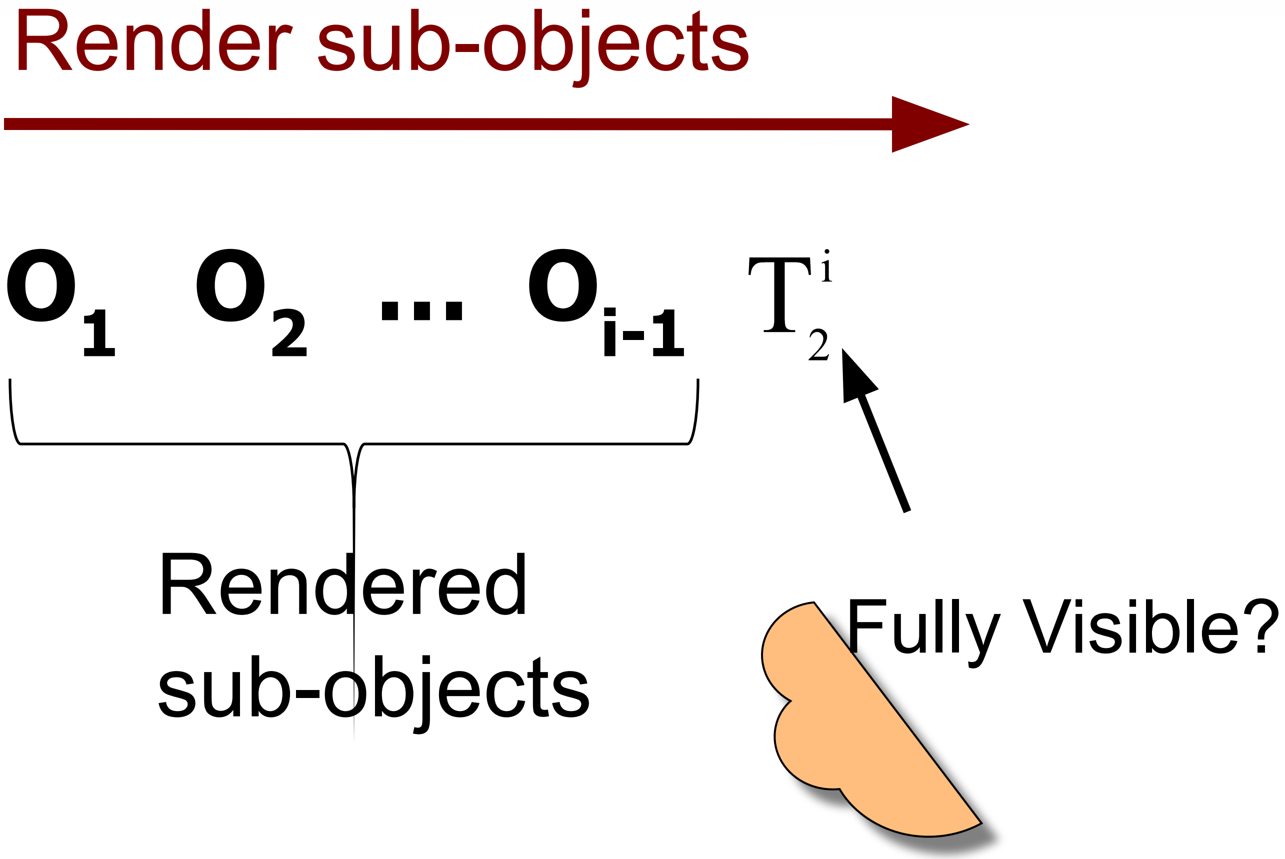


Render sub-objects

$$\underbrace{O_1 \quad O_2 \quad \ldots \quad O_{i-1}}_{\text{Rendered sub-objects}} \quad T_1^i$$

Fully Visible?

Overlap Localization: First Pass



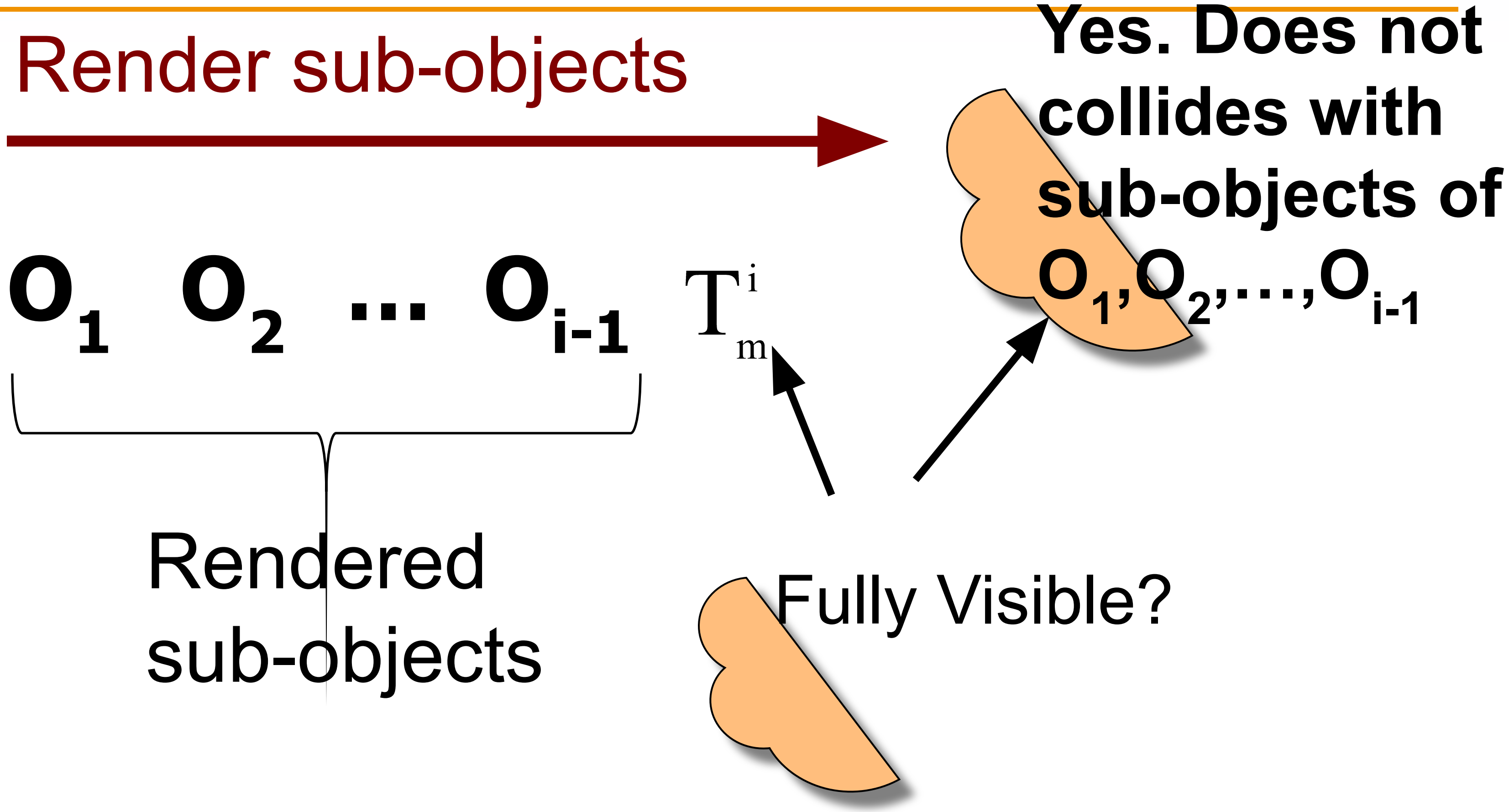
Render sub-objects

O_1 O_2 … O_{i-1} T_2^i

Rendered sub-objects

Fully Visible?

Overlap Localization: First Pass



Render sub-objects

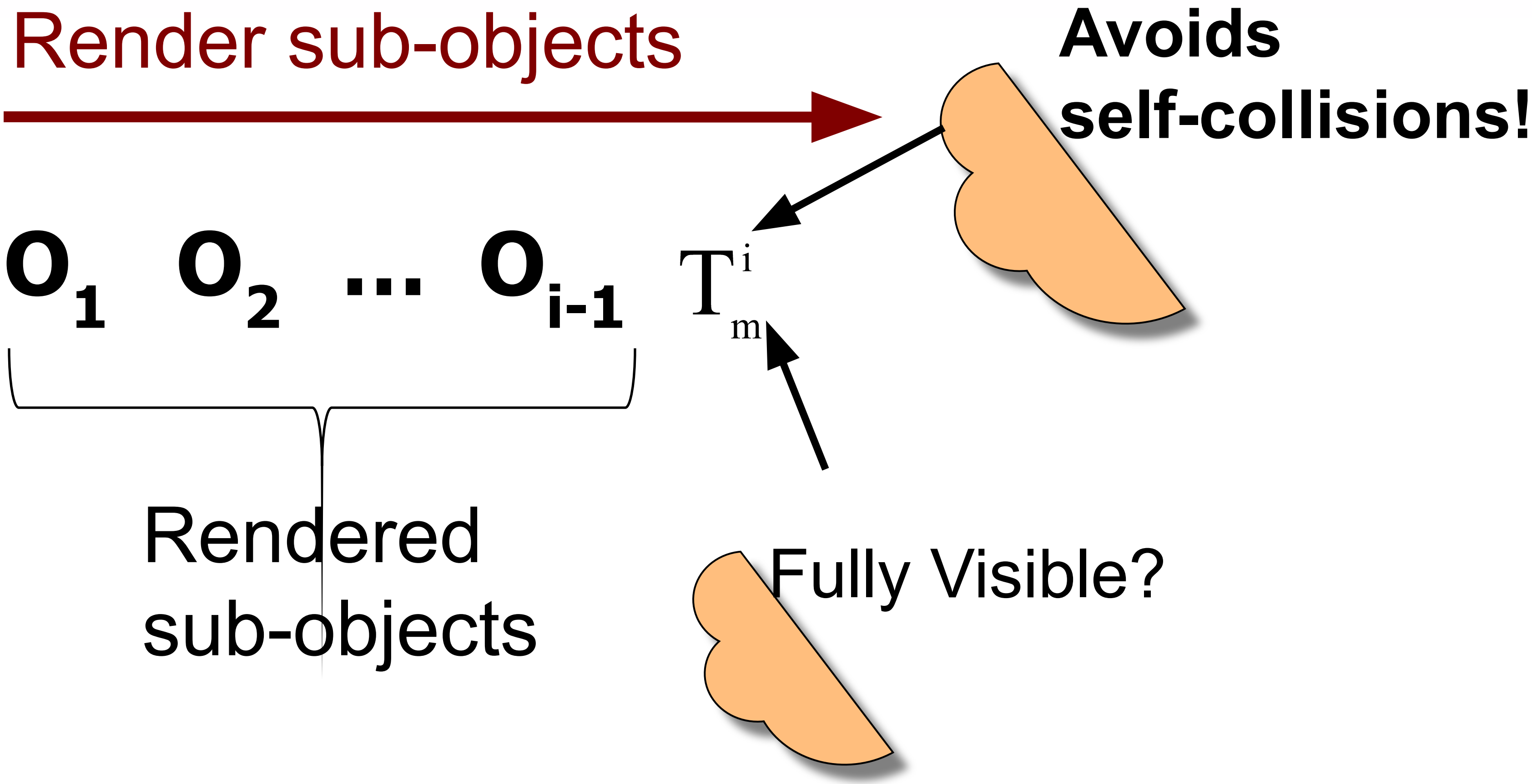
$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \quad T^i_m$

Rendered sub-objects

Fully Visible?

Yes. Does not collides with sub-objects of $O_1, O_2, \ldots, O_{i-1}$

Overlap Localization: First Pass



Render sub-objects

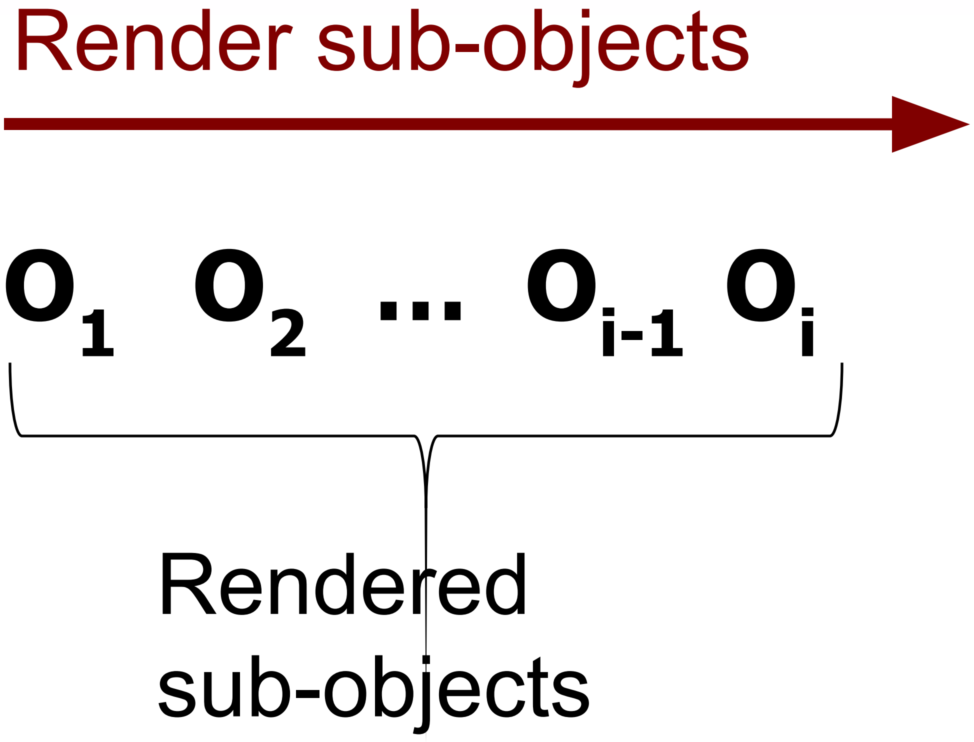
O_1 O_2 … O_{i-1} T^i_m

Rendered sub-objects

Avoids self-collisions!

Fully Visible?

Overlap Localization: First Pass



Render sub-objects

$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i$

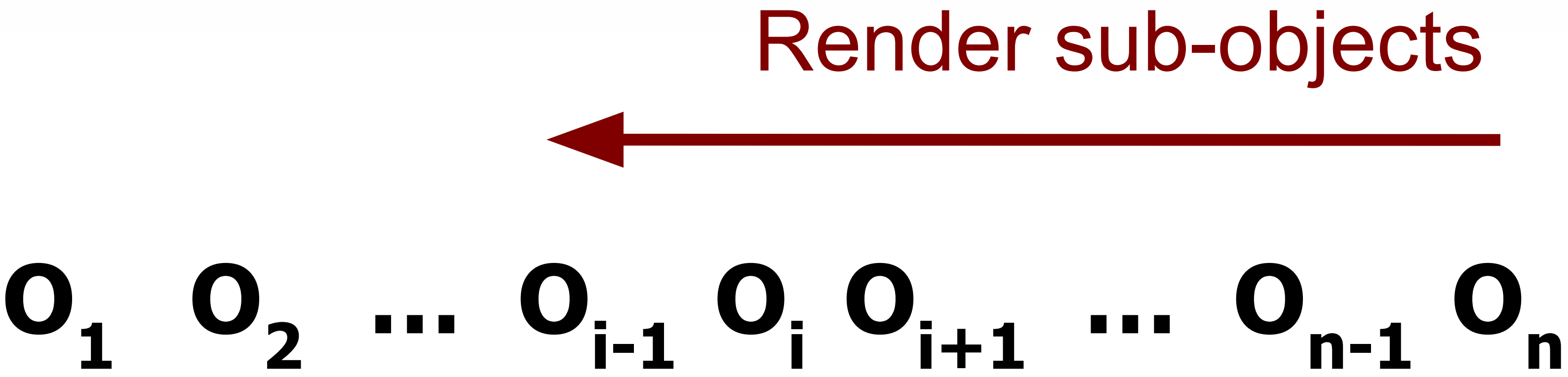
Rendered sub-objects

Overlap Localization: First Pass

$$\underbrace{O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n}$$

Rendered
sub-objects

Overlap Localization: Second Pass

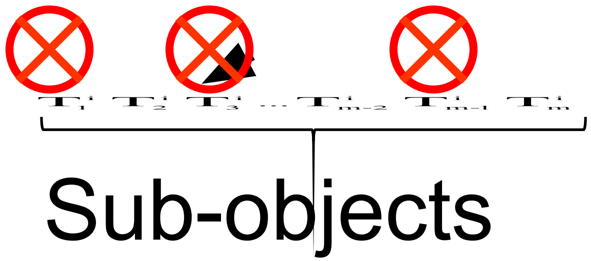


Render sub-objects

$$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n$$

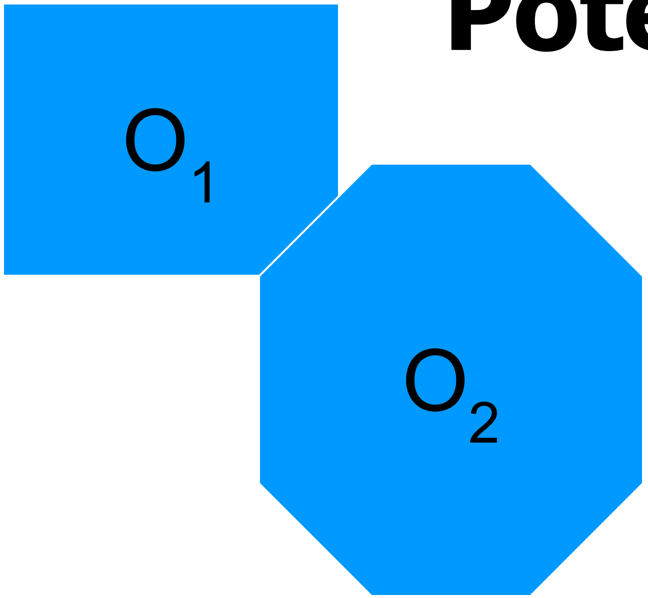
Overlap Localization

$$O_1 \quad O_2 \quad \ldots \quad O_{i-1} \; O_i \; O_{i+1} \quad \ldots \quad O_{n-1} \; O_n$$



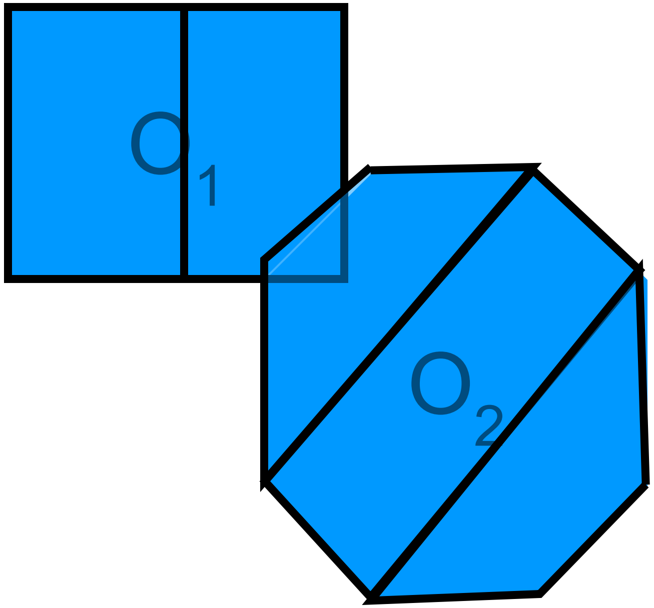
$$\underbrace{T_1^i \; T_2^i \; T_3^i \; \ldots \; T_{m-2}^i \; T_{m-1}^i \; T_m^i}_{\text{Sub-objects}}$$

Potential Colliding Set



PCS = $\{O_1,O_2\}$

Sub-objects

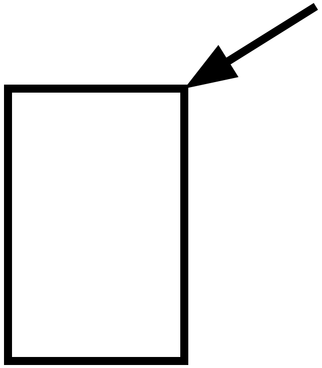


PCS = sub-objects of $\{O_1,O_2\}$

First Pass

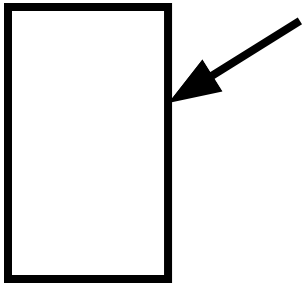
Rendering order: Sub-objects of $O_1 \longrightarrow O_2$

First Pass



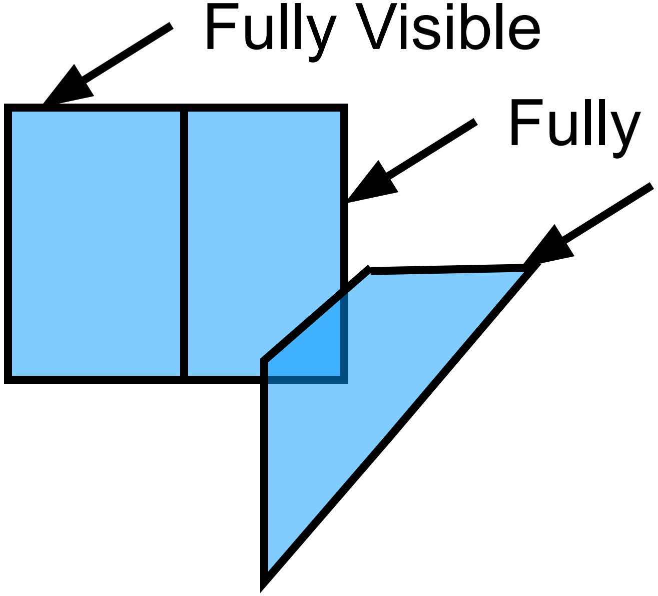
Fully Visible

First Pass



Fully Visible

First Pass

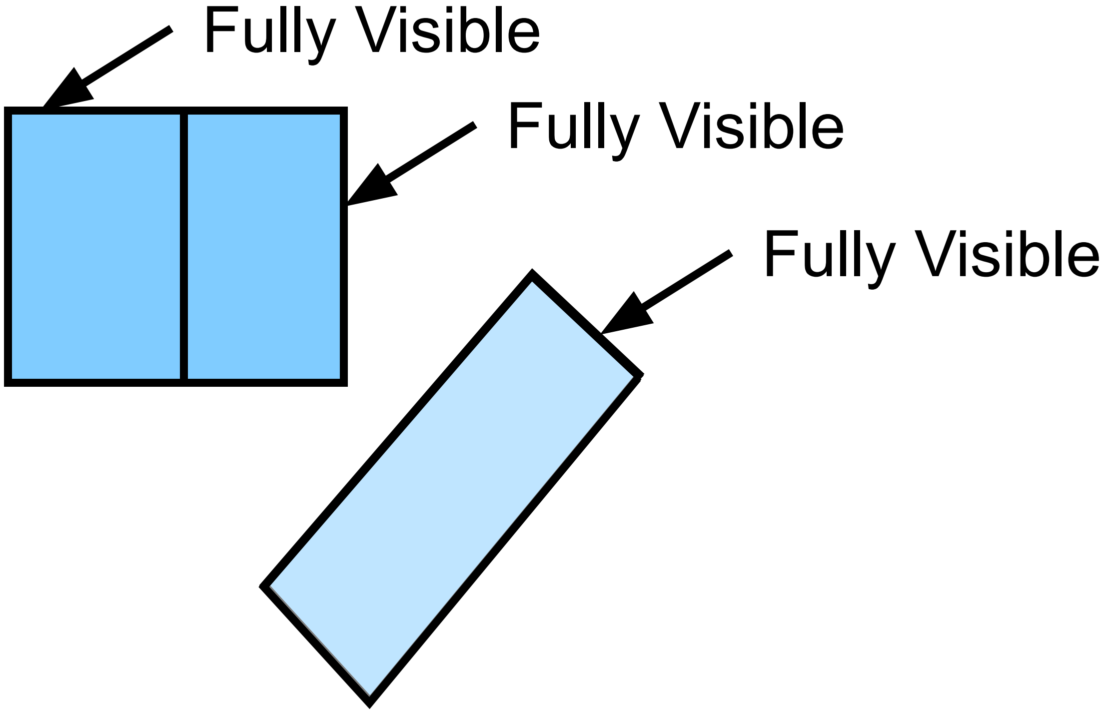


Fully Visible

Fully Visible

Not Fully Visible

First Pass

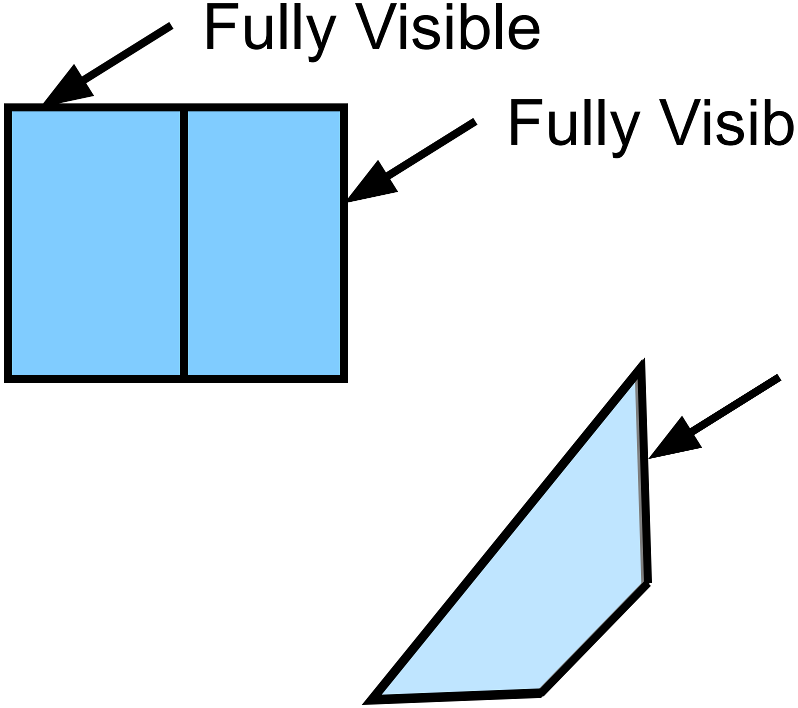


Fully Visible

Fully Visible

Fully Visible

First Pass

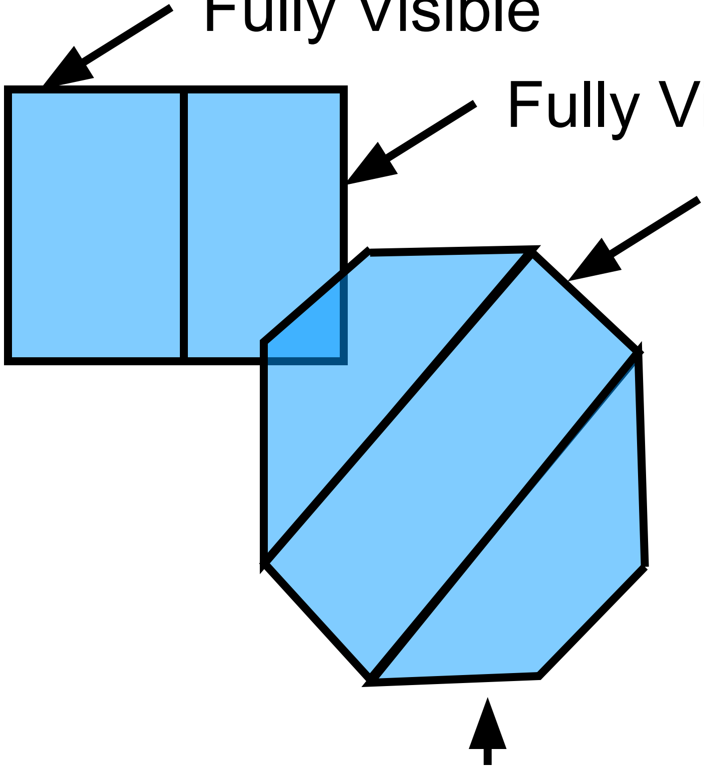


Fully Visible

Fully Visible

Fully Visible

First Pass



Fully Visible

Fully Visible

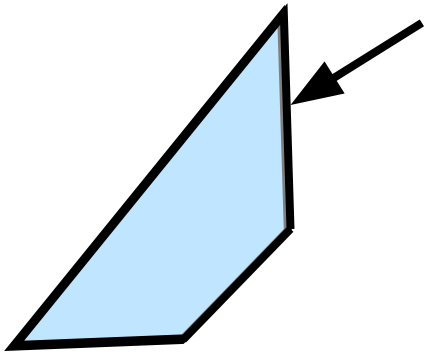
Fully Visible

Fully Visible

Second Pass

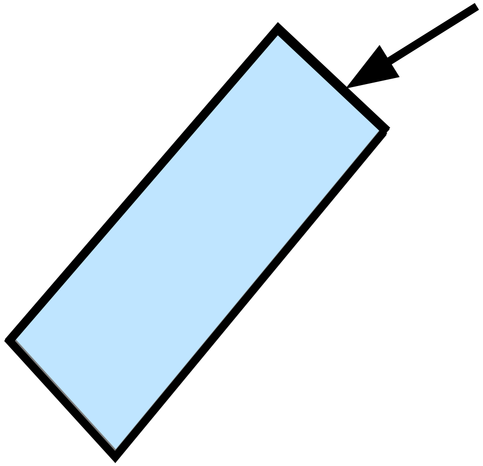
Rendering order: Sub-objects of $O_2 \longrightarrow O_1$

Second Pass



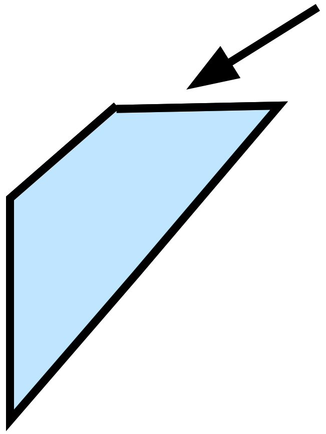
Fully Visible

Second Pass



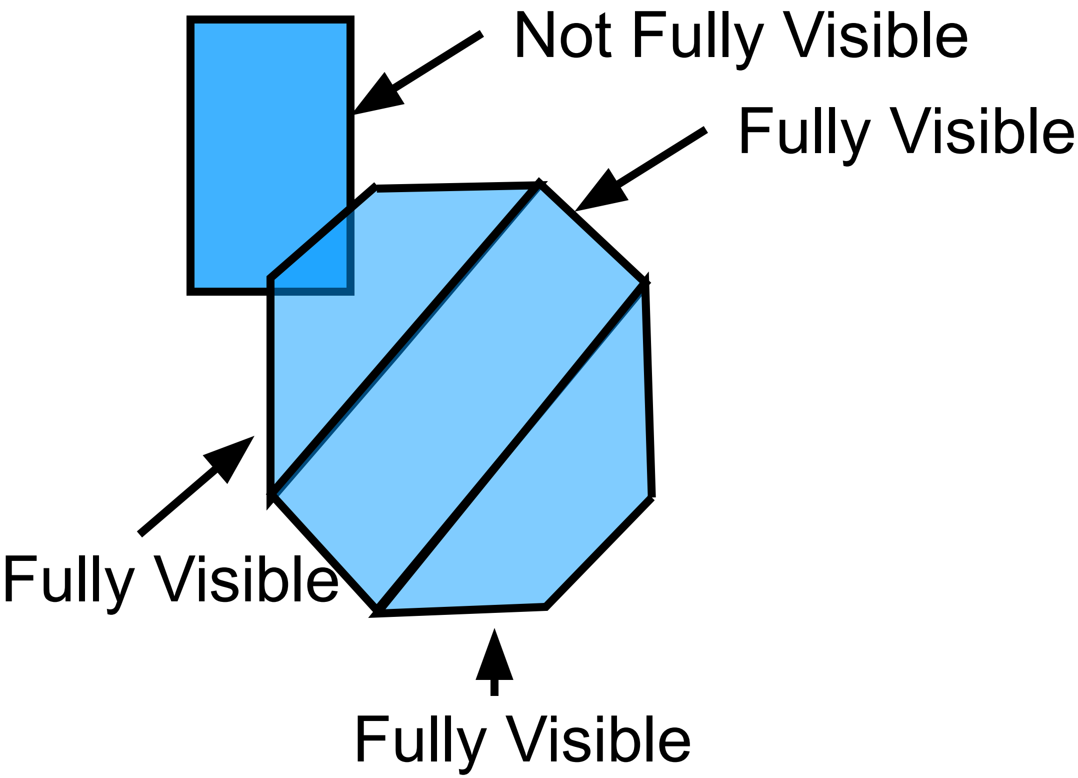
Fully Visible

Second Pass



Fully Visible

Second Pass



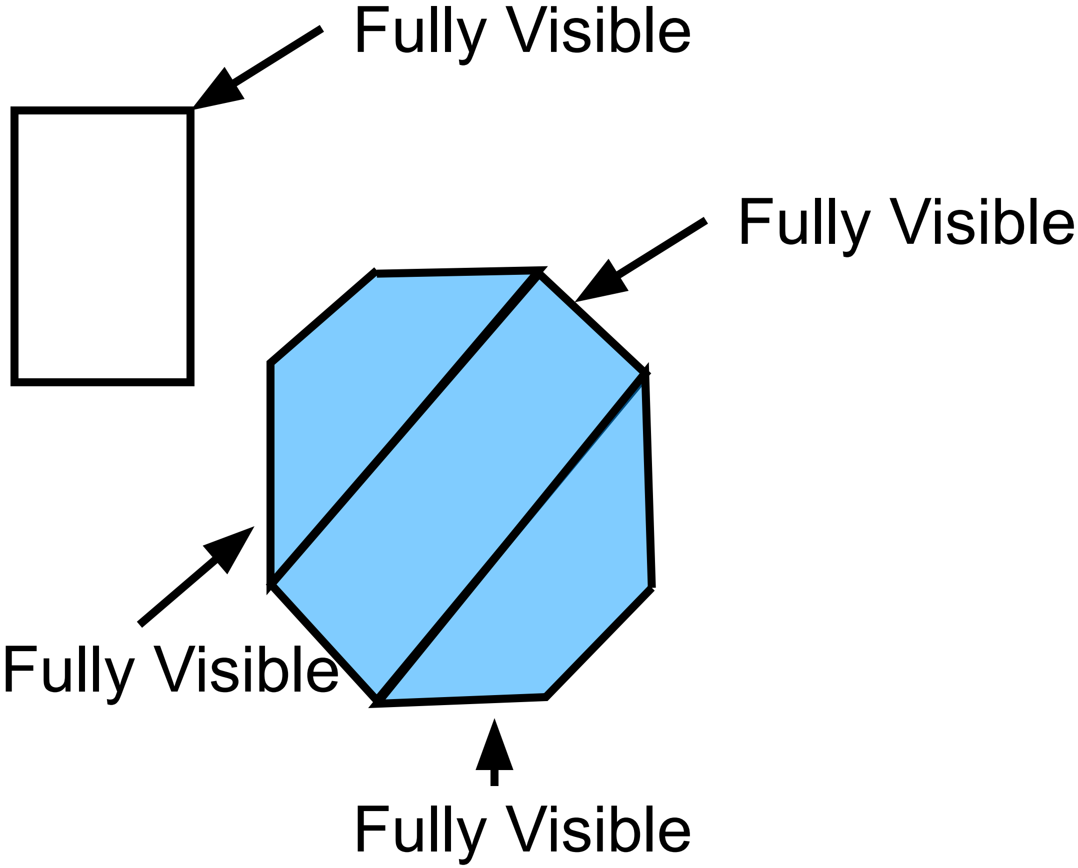
Not Fully Visible

Fully Visible

Fully Visible

Fully Visible

Second Pass



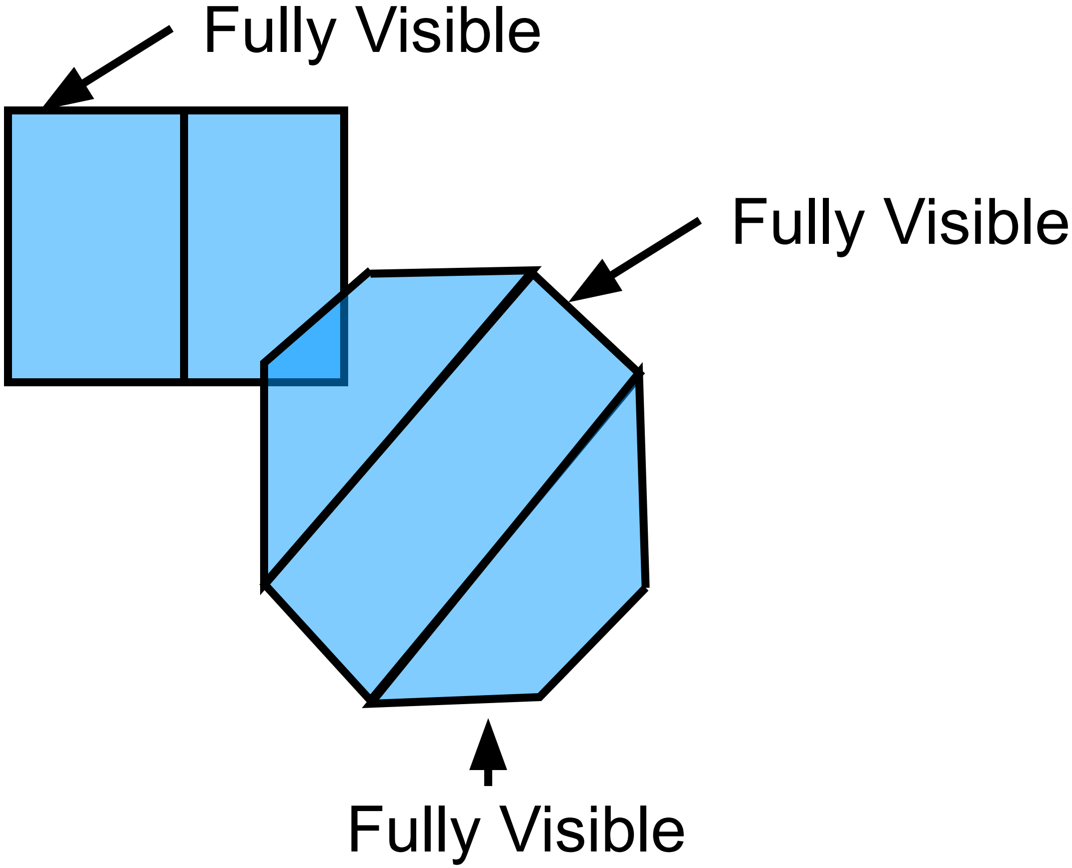
Fully Visible

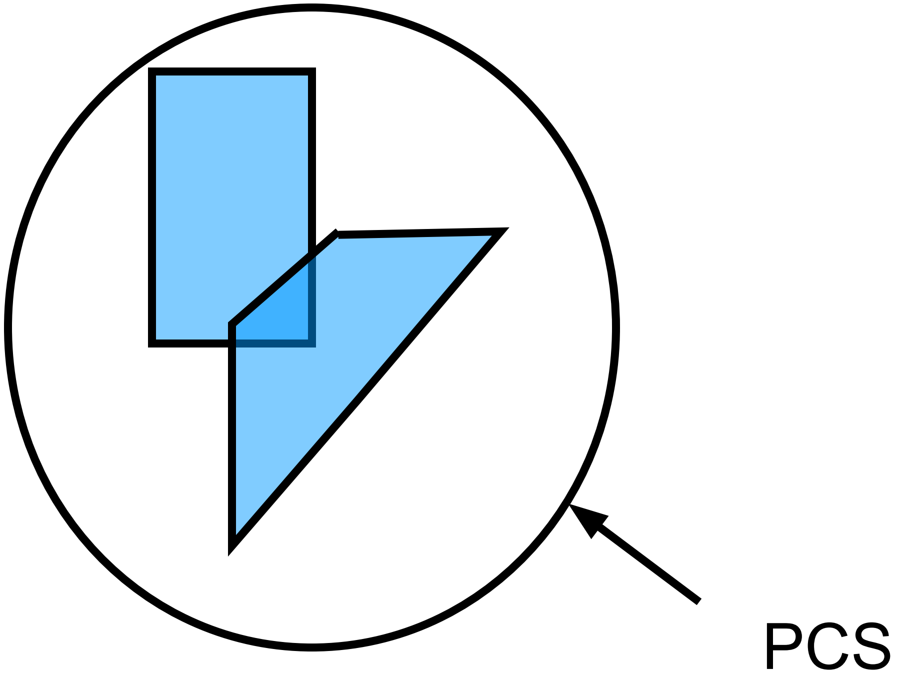
Fully Visible

Fully Visible

Fully Visible

After two passes




PCS

Algorithm

Object Level Pruning → Sub-object level Pruning → Exact Tests

Exact Overlap tests using CPU

Visibility Queries

- We require a query
  - Tests if a primitive is fully visible or not
- Current hardware supports occlusion queries
  - Test if a primitive is visible or not
- Our solution
  - Change the sign of depth function

Visibility Queries

	Depth function	
	GEQUAL	LESS
All fragments	Pass	Fail
	Fail	Pass

Occlusion query

Query not supported

- Examples - HP_Occlusion_test, NV_occlusion_query

Bandwidth Analysis

- Read back only integer identifiers
  - Independent of screen resolution

Optimizations

- First use AABBs as object bounding volume
- Use orthographic views for pruning
- Prune using original objects

Advantages

- No coherence
- No assumptions on motion of objects
- Works on generic models
- A fast pruning algorithm
- No frame-buffer readbacks

Limitations

- No distance or penetration depth information
- Resolution issues
- No self-collisions
- Culling performance varies with relative configurations

Assumptions

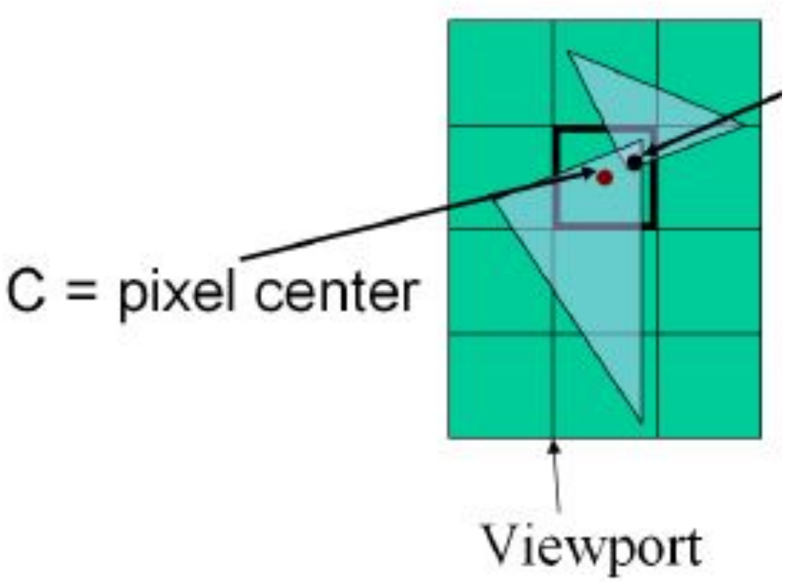
- Makes assumptions that their algorithm will get faster as hardware improves.
- Luckily they were right

RCULLIDE

- An improvement on CULLIDE in 2004
- Resolves issue of screen resolution precision

Overview

- A main issue with CULLIDE was the fact that it wasn't reliable
- Collisions could easily be missed due to screen resolution



Overview

- 3 kinds of error associated with visibility based overlap
  - Perspective error
    - Strange shapes from the transformation
  - Sampling error
    - Pixel resolution isn’t high enough
  - Depth buffer precision error
    - If distance between primitives is less than the depth buffer resolution, we will get incorrect results from our visibility query

Reliable Queries

- The three errors cause the following:
  - A fragment to not be rasterized
  - A fragment is generated but not sampled where interference occurs
  - A fragment is generated and sampled where the interference occurs but the precision of the buffer is not sufficient

Reliable Queries

- Use “fat” triangles
  - Generate 2 fragments for each pixel touched by a triangle (no matter how little it is in the pixel)
  - For each pixel touched by the triangle, the depth of the 2 fragments must bound the depth of all points of the triangle in that pixel
- Causes method to become more conservative (read: slower) but much more accurate

Minkowski Sum

- Scary name…easy math

$$A + B = \{\mathbf{a} + \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}.$$

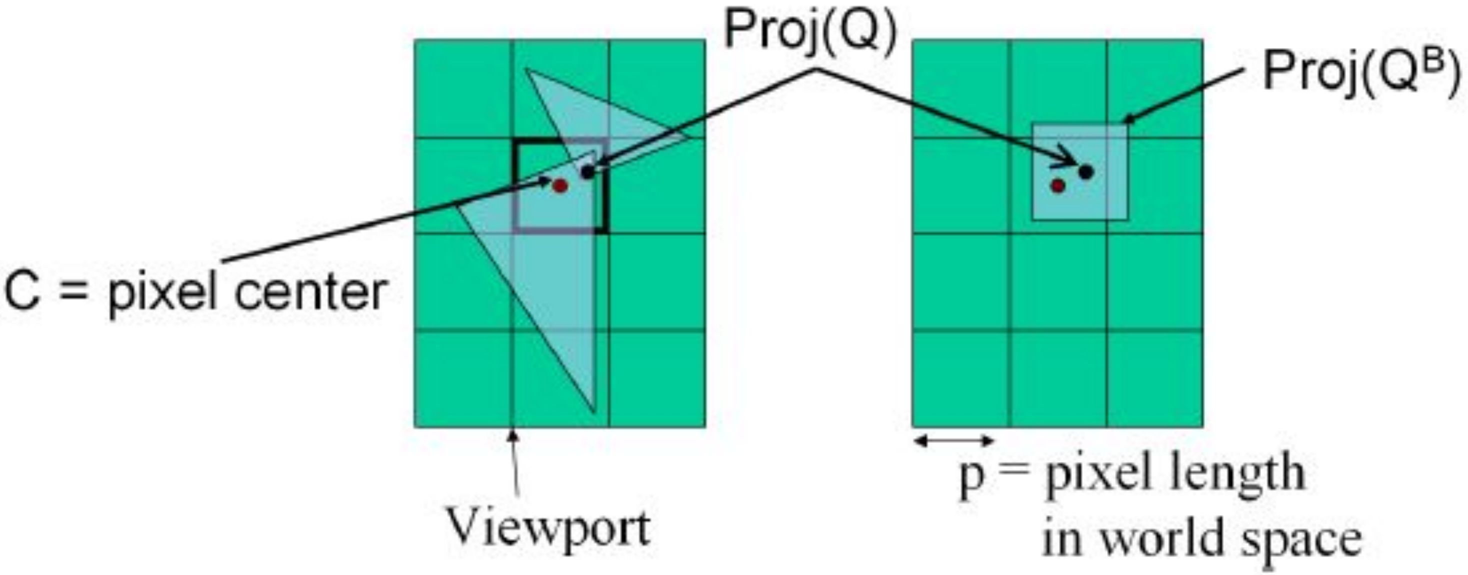
$A = \{ (1, 0), (0, 1), (0, -1)\}$
$B = \{ (0, 0), (1, 1), (1, -1)\}$

$A + B = \{ (1, 0), (2, 1), (2, -1), (0, 1), (1, 2), (1, 0), (0, -1), (1, 0), (1, -2)\}$

Reliable Queries

- In practice, we use the Minkowski sum of a bounding cube B and the triangle T
- B = max(2dx, 2dy, 2dz) where dx,y,z are pixel dimensions
- If uniform supersampling is known to occur on the card, we can reduce the size of B
  - We need B to cover at least 1 sampling point for the triangle it bounds

Reliable Queries

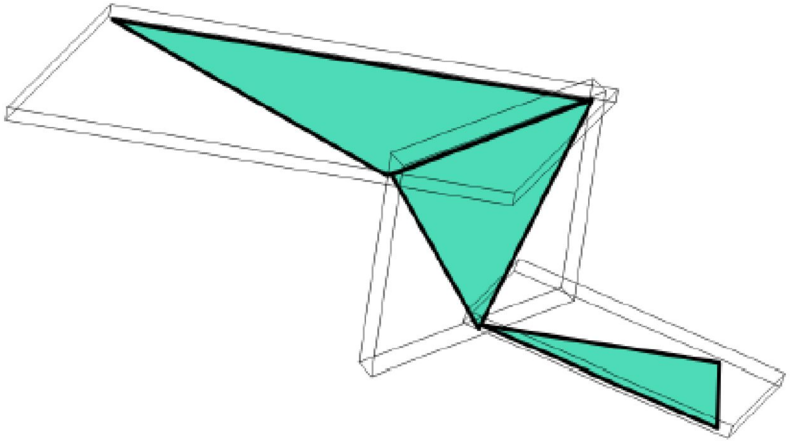


- Cubes only work for z-axis projections so in practice use a bounding sphere of radius sqrt(3)p/2

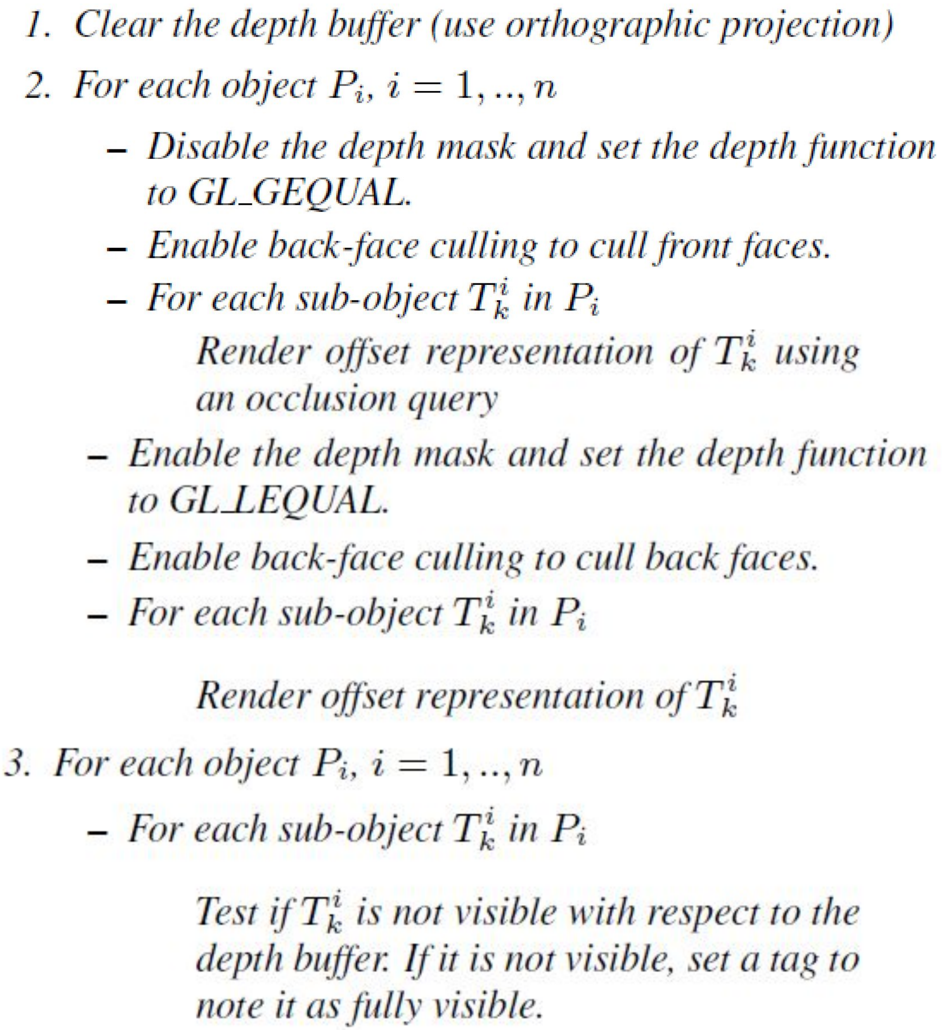
Bounding Offset

- So far we've just dealt with single triangles but we need whole objects

- This is done using a Union of Object-oriented Bounding Boxes(UOBB)



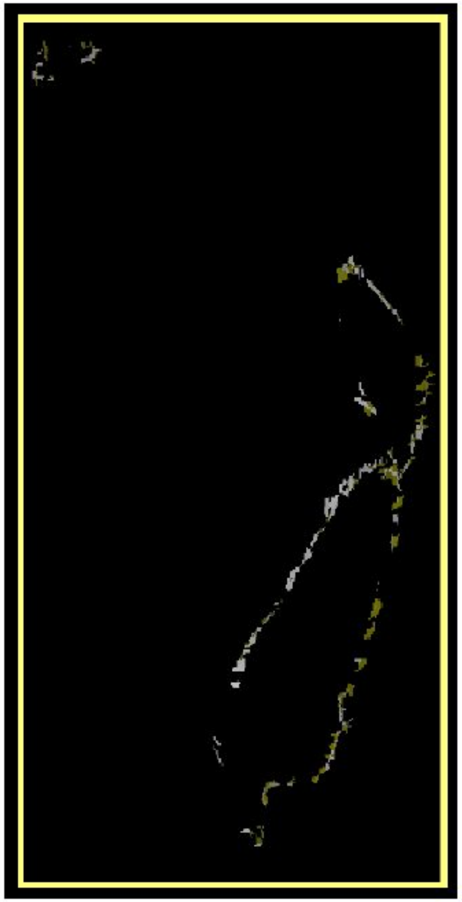
Algorithm



```
1. Clear the depth buffer (use orthographic projection)
2. For each object $P_i$, $i = 1, .., n$
    – Disable the depth mask and set the depth function to GL_GEQUAL.
    – Enable back-face culling to cull front faces.
    – For each sub-object $T_k^i$ in $P_i$
          Render offset representation of $T_k^i$ using an occlusion query
    – Enable the depth mask and set the depth function to GL_LEQUAL.
    – Enable back-face culling to cull back faces.
    – For each sub-object $T_k^i$ in $P_i$
          Render offset representation of $T_k^i$
3. For each object $P_i$, $i = 1, .., n$
    – For each sub-object $T_k^i$ in $P_i$
          Test if $T_k^i$ is not visible with respect to the depth buffer. If it is not visible, set a tag to note it as fully visible.
```

Improvement over CULLIDE

(a) *Interference computation on two bunnies*



(b) *Intersection curve computed by CULLIDE*

(c) *Intersection curve computed by FAR*

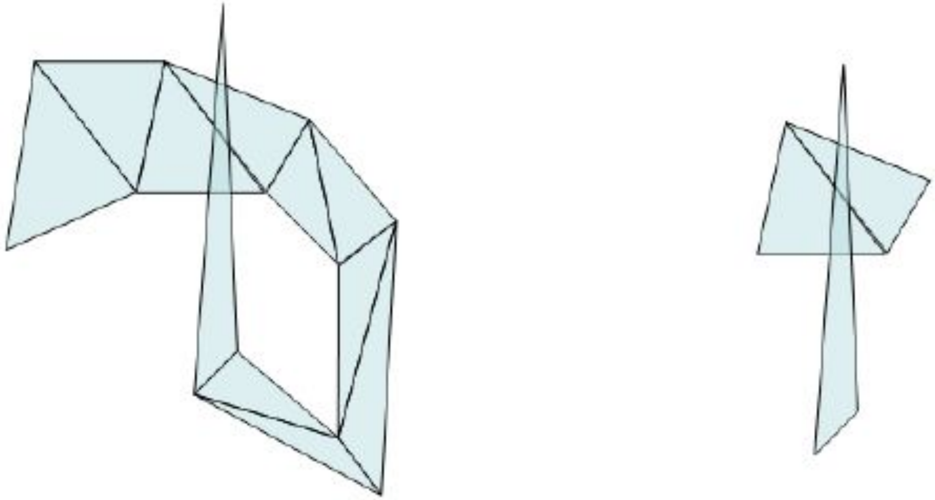
Performance

- Still runs faster than CPU implementations
- 3x slower than CULLIDE due to bounding box rasterization vs triangle rasterization

QCULLIDE

- Extends CULLIDE to handle self collisions in complex meshes
- All running in real time

Self Collision Culling



- Note that only intersecting triangles that don’t share a vertex or edge are considered colliding

Self Collision Culling

- Algorithm
  - Include all potentially colliding primitives and PCS where each primitive is a triangle
  - Perform the visibility test to see if a triangle is penetrating any other
  - If completely visible, the object is not colliding

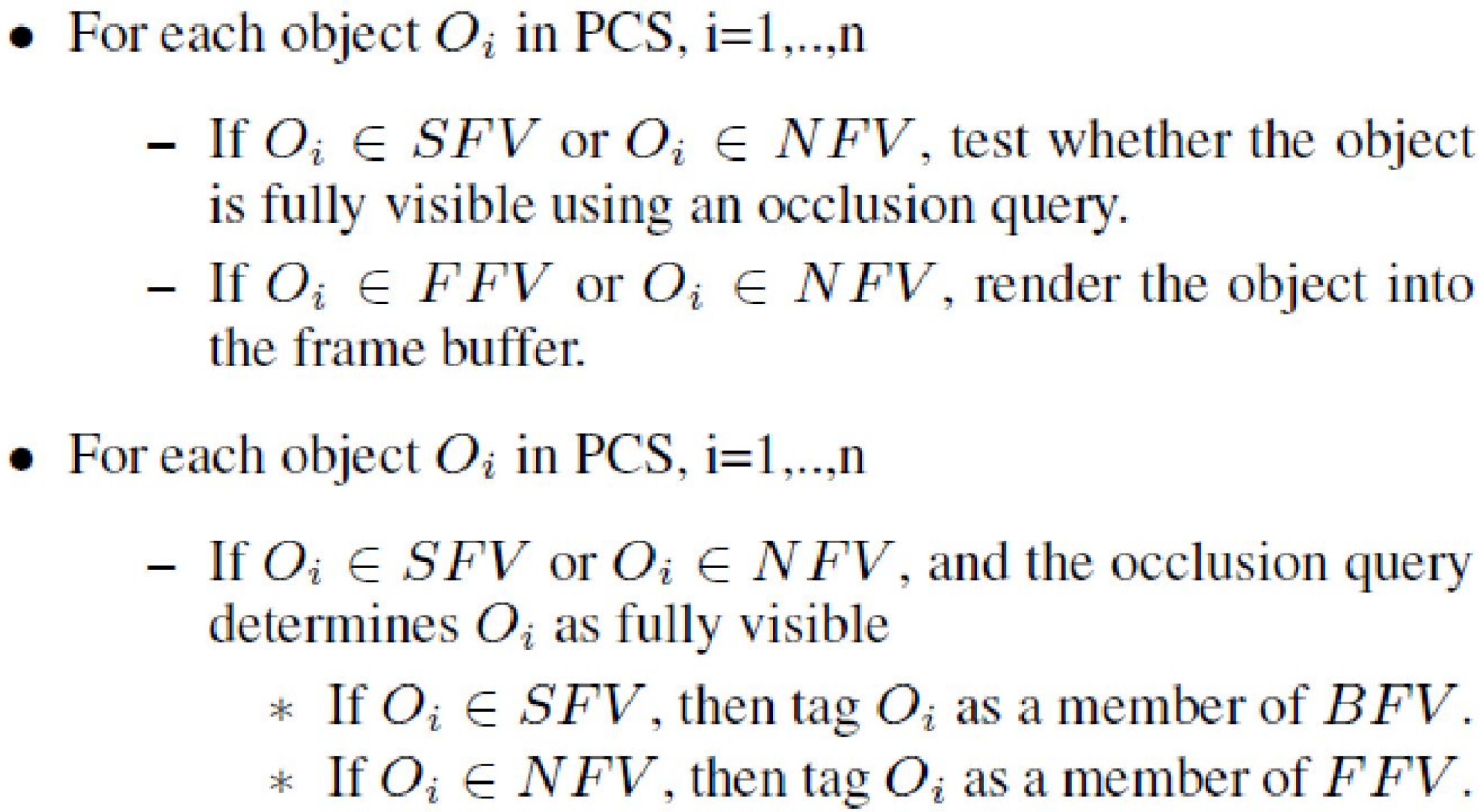
Q-CULLIDE

- Sets
  - BFV – Objects fully visible in both passes and are pruned from the PCS
  - FFV – Fully visible in only the first pass
  - SFV – Fully visible in only the second pass
  - NFV – Not fully visible in both passes

Q-CULLIDE

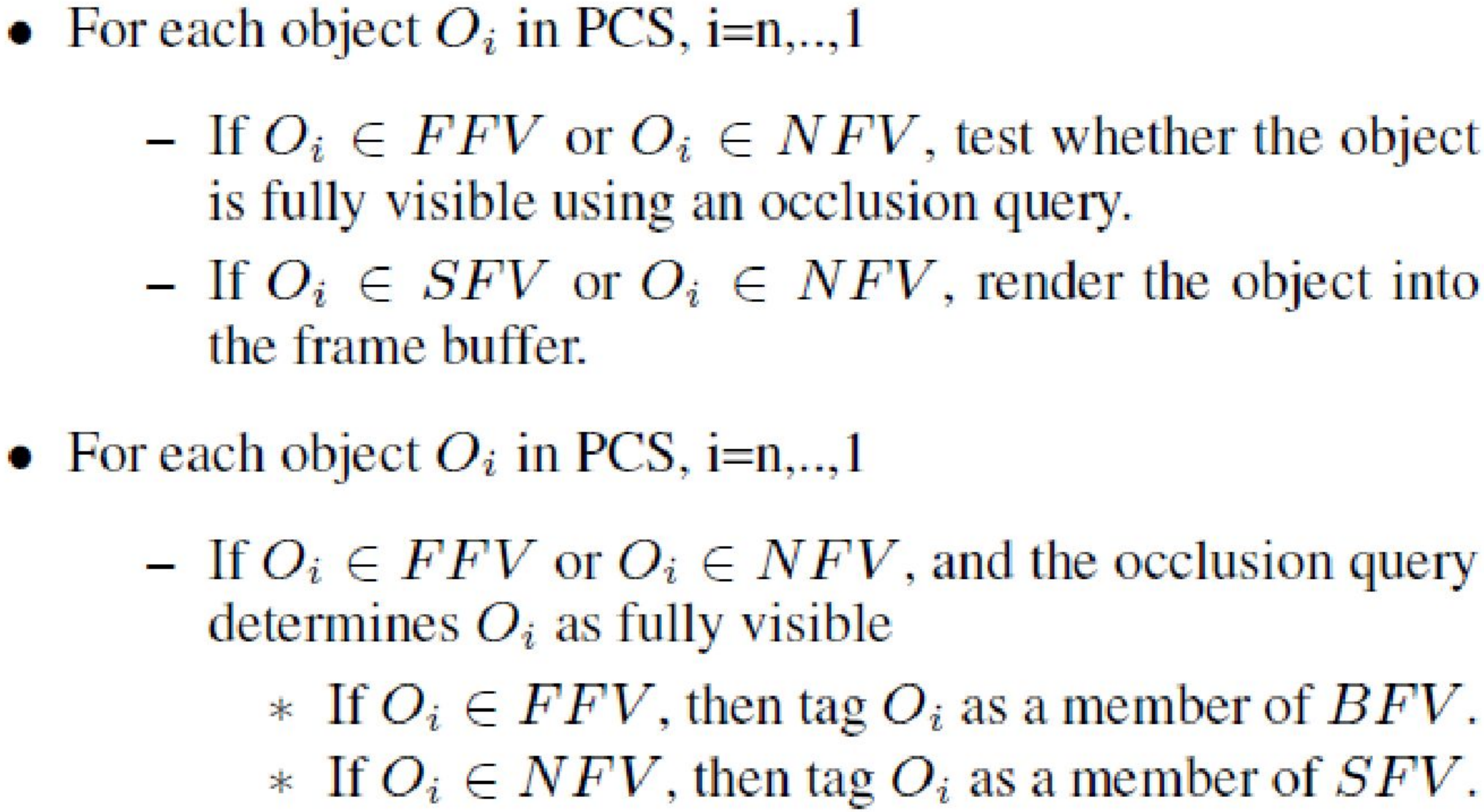
- Properties of sets
  - FFV and SFV are collision free
    - No object in FFV collides with any other in FFV…same for SFV
  - If an object is in FFV and is fully visible in the 2nd pass of the algorithm, we can prune it and vice versa

Algorithm



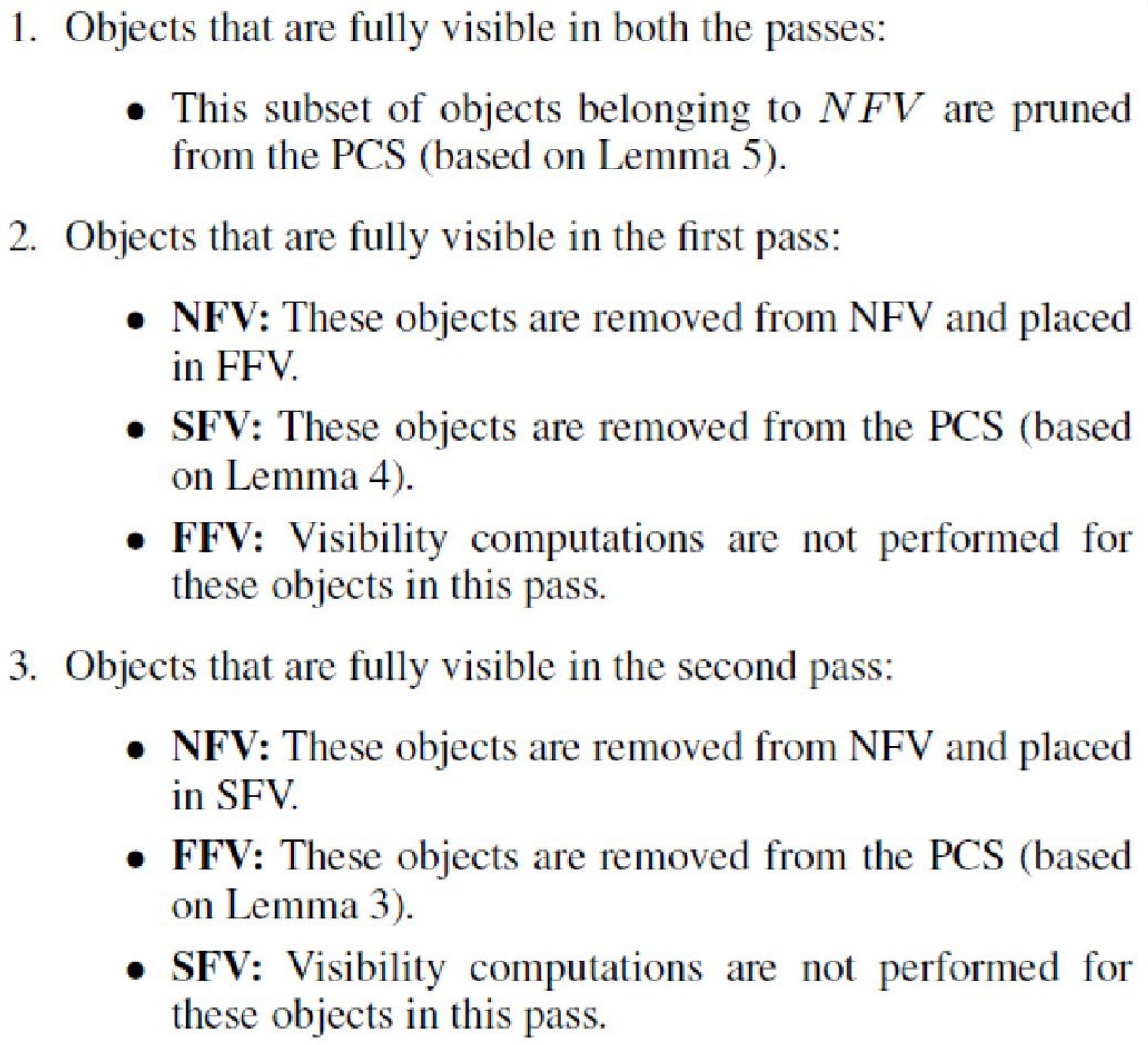
- For each object O_i in PCS, i=1,..,n
  - If $O_i \in SFV$ or $O_i \in NFV$, test whether the object is fully visible using an occlusion query.
  - If $O_i \in FFV$ or $O_i \in NFV$, render the object into the frame buffer.
- For each object O_i in PCS, i=1,..,n
  - If $O_i \in SFV$ or $O_i \in NFV$, and the occlusion query determines O_i as fully visible
    - $*$ If $O_i \in SFV$, then tag O_i as a member of BFV.
    - $*$ If $O_i \in NFV$, then tag O_i as a member of FFV.

Algorithm



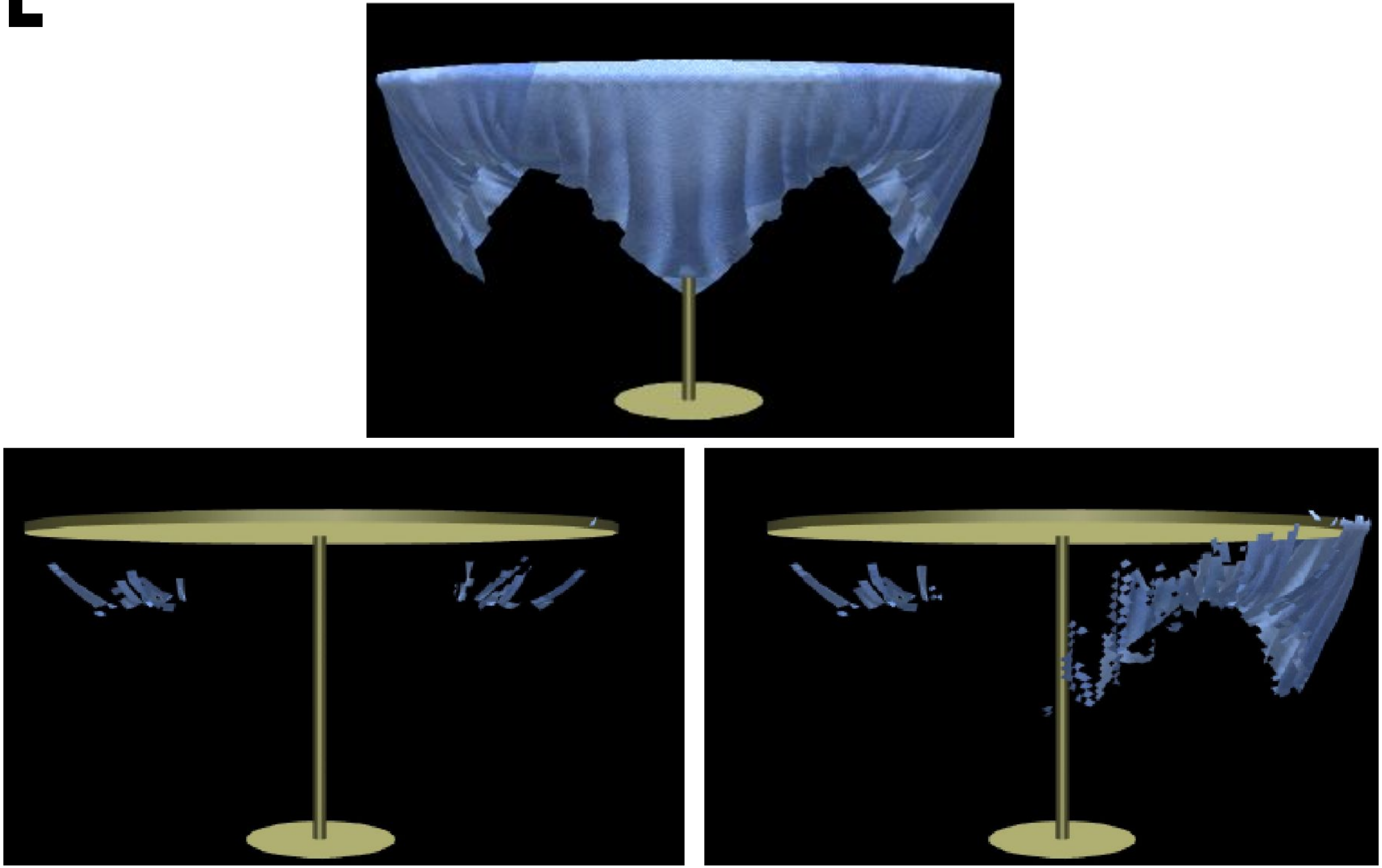
- For each object O_i in PCS, i=n,..,1
  - If $O_i \in FFV$ or $O_i \in NFV$, test whether the object is fully visible using an occlusion query.
  - If $O_i \in SFV$ or $O_i \in NFV$, render the object into the frame buffer.
- For each object O_i in PCS, i=n,..,1
  - If $O_i \in FFV$ or $O_i \in NFV$, and the occlusion query determines O_i as fully visible
    - If $O_i \in FFV$, then tag O_i as a member of BFV.
    - If $O_i \in NFV$, then tag O_i as a member of SFV.

What’s Happening



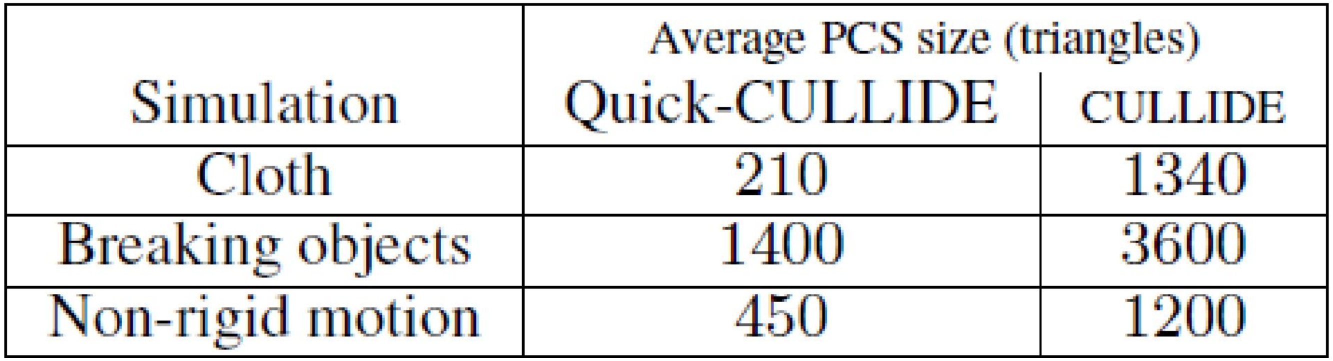
1. Objects that are fully visible in both the passes:
   - This subset of objects belonging to NFV are pruned from the PCS (based on Lemma 5).
2. Objects that are fully visible in the first pass:
   - **NFV:** These objects are removed from NFV and placed in FFV.
   - **SFV:** These objects are removed from the PCS (based on Lemma 4).
   - **FFV:** Visibility computations are not performed for these objects in this pass.
3. Objects that are fully visible in the second pass:
   - **NFV:** These objects are removed from NFV and placed in SFV.
   - **FFV:** These objects are removed from the PCS (based on Lemma 3).
   - **SFV:** Visibility computations are not performed for these objects in this pass.

Improvement Over CULLIDE



Improvements Over CULLIDE

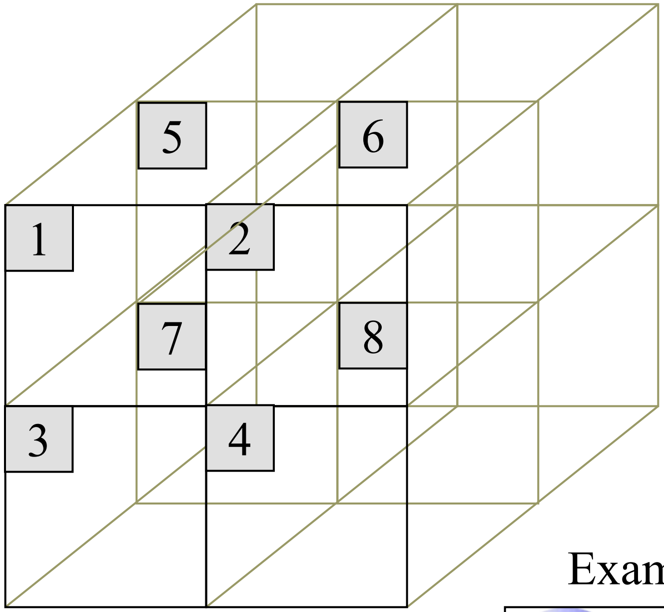
- Sends an order of magnitude less collisions to the CPU than CULLIDE



Simulation	Average PCS size (triangles) Quick-CULLIDE	 CULLIDE
Cloth	210	1340
Breaking objects	1400	3600
Non-rigid motion	450	1200

Spatial Subdivision

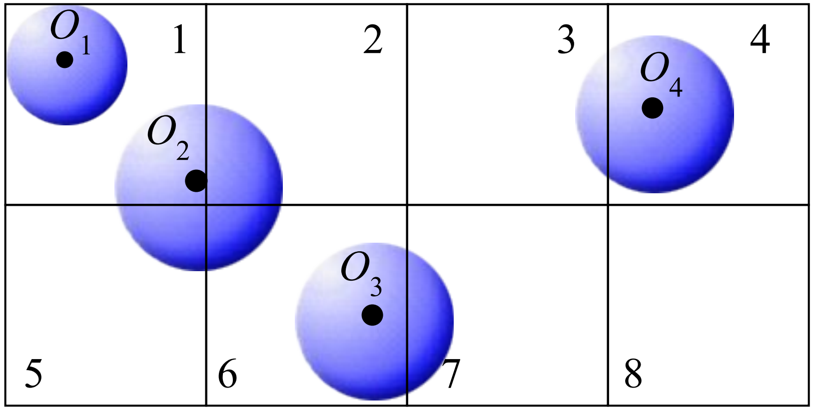
- Partition space into uniform grid
- Grid cell is at least as large as largest object
- Each cell contains list of each object whose centroid is in the cell
- Collision tests are performed between objects who are in same cell or adjacent cells



Implementation:

1. Create list of object IDs along with hashing of cell IDs in which they reside
2. Sort list by cell ID
3. Traverse swaths of identical cell IDs
4. Perform collision tests on all objects that share same cell ID

Example



Images from pg 699, 700 GPU Gems III

Parallel Spatial Subdivision

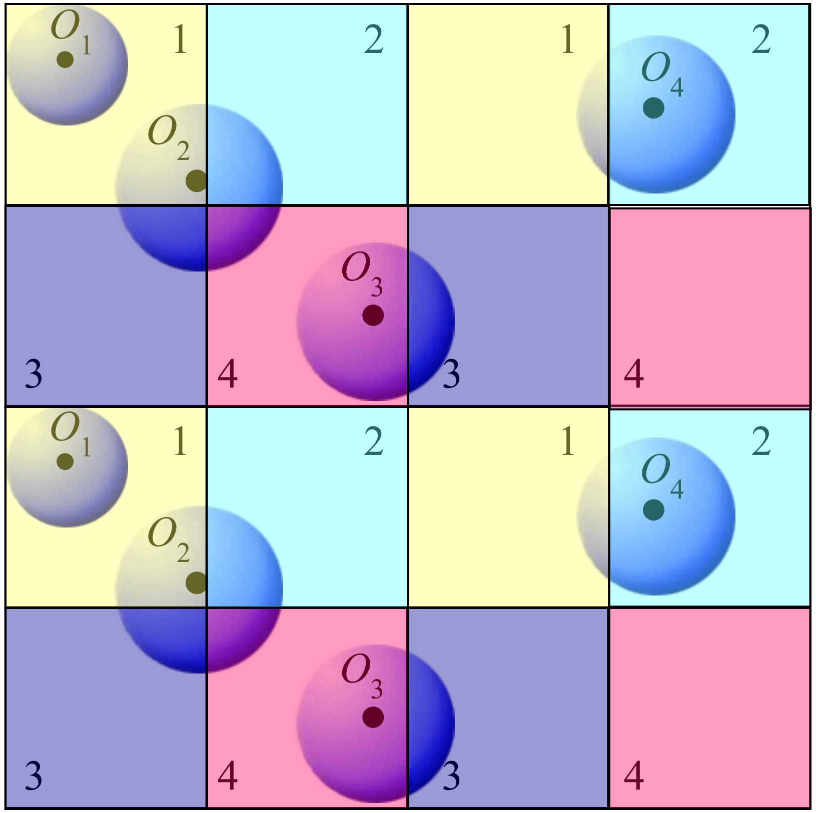
- Complications:
  1. Single object can be involved in multiple collision tests
  2. Need to prevent multiple threads updating the state of an object at the same time

Ways to solve this?

Guaranteed Individual Collision Tests

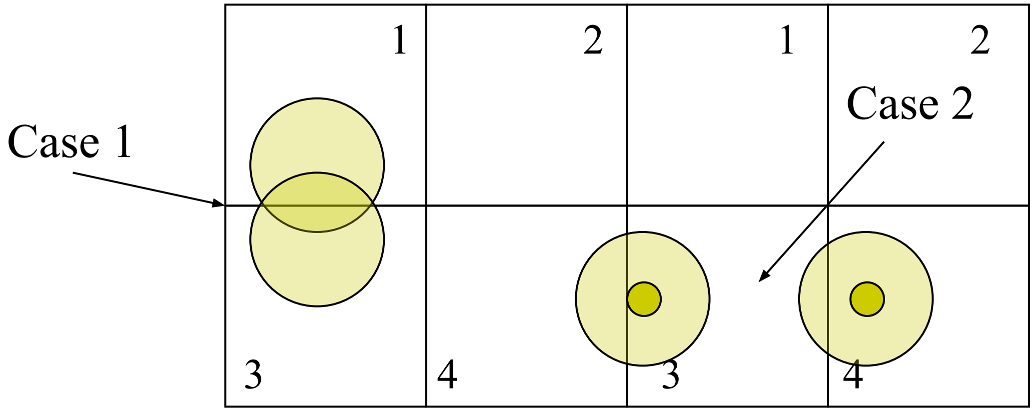
- Prove: No two cells updated in parallel may contain the same object that is being updated
    - Constraints
    1. Each cell is as large as the bounding volume of the largest object
    2. Each cell processed in parallel must be separated by each other cell by at least one intervening cell
    - In 2d this takes ___4___ number of passes
    - In 3d this takes ___8___ number of passes

Example of Parallel Spatial Subdivision



Avoiding Extra Collision Testing

1. Associate each object a set of control bits to test where its centroid resides
2. Scale the bounding sphere of each object by sqrt(2) to ensure the grid cell is at least 1.5 times larger than the largest object



Implementing in CUDA

- Store list of object IDs, cell IDs in device memory
- Build the list of cell IDs from object's bounding boxes
- Sorting list from previous step
- Build an index table to traverse the sorted list
- Schedule pairs of objects for narrow phase collision detection

Initialization

Cell ID Array

OBJ 1 Cell ID 1
OBJ 1 Cell ID 2
OBJ 1 Cell ID 3
OBJ 1 Cell ID 4
OBJ 2 Cell ID 1
OBJ 2 Cell ID 2
OBJ 2 Cell ID 3
OBJ 2 Cell ID 4
.
.
.

Object ID Array

OBJ 1 ID, Control Bits
OBJ 1 ID, Control Bits
OBJ 1 ID, Control Bits
OBJ 1 ID, Control Bits
OBJ 2 ID, Control Bits
OBJ 2 ID, Control Bits
OBJ 2 ID, Control Bits
OBJ 2 ID, Control Bits
.
.
.

Construct the Cell ID Array

Host Cells (H – Cells)

Contain the centroid of the object

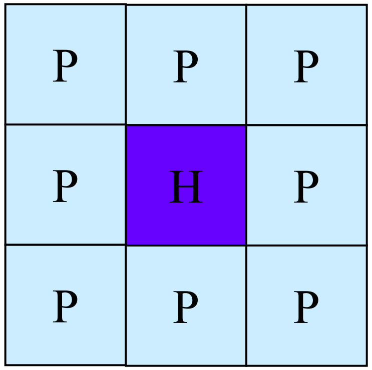
H-Cell Hash = (pos.x / CELLSIZE) << XSHIFT) |
(pos.y / CELLSIZE) << YSHIFT) |
(pos.z / CELLSIZE) << ZSHIFT)

Phantom Cells (P-Cells)

Overlap with bounding volume but do not contain the centroid

P-Cells – Test the 3^d-1 cells surrounding the H cell

There can be as many as 2^d-1 P cells

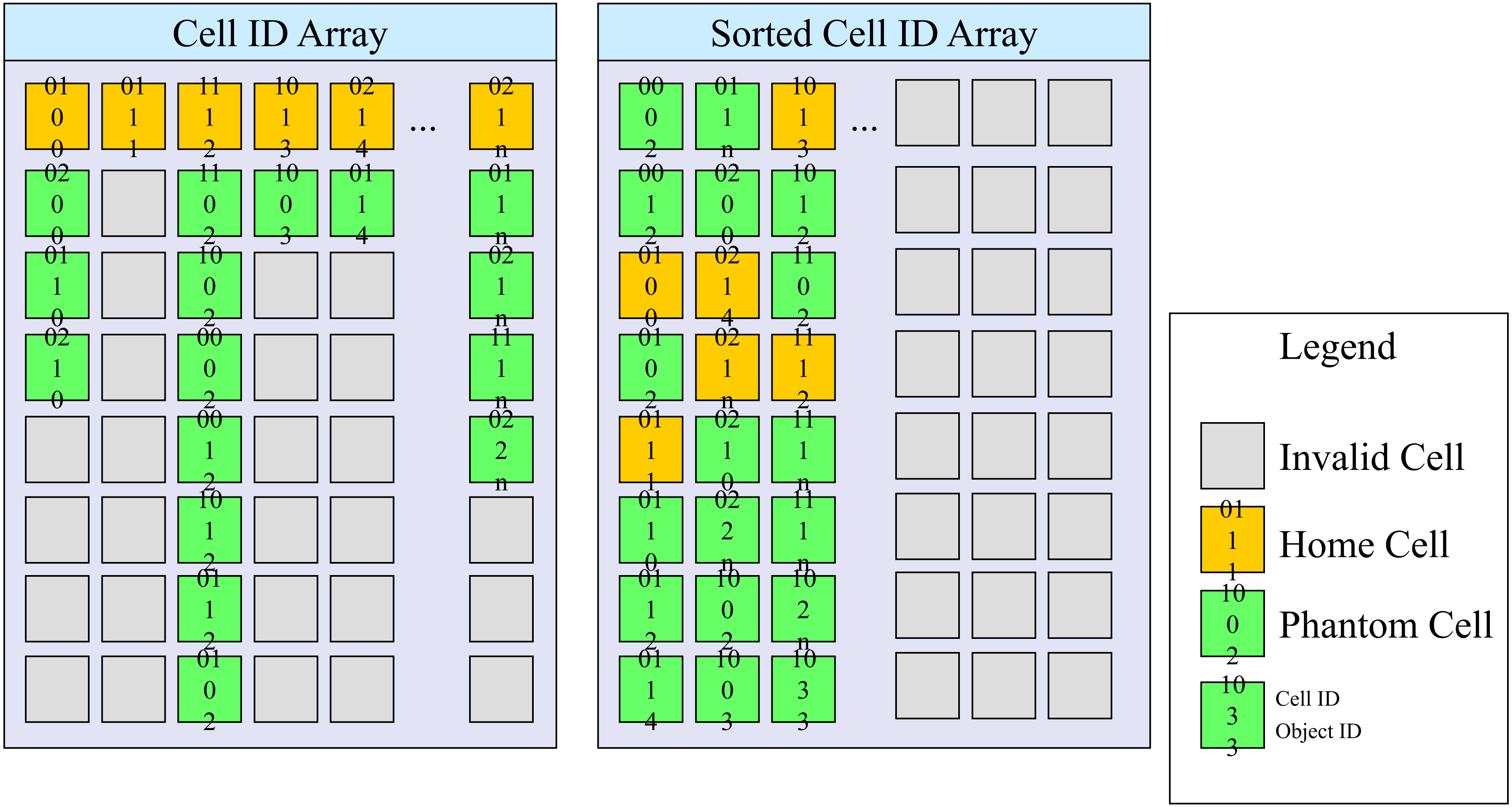


P	P	P
P	H	P
P	P	P

Sorting the Cell ID Array

- What we want:
  - Sorted by Cell ID
  - H cells of an ID occur before P cells of an ID
- Starting with a partial sort
  - H cells are before P cells, but array is not sorted by Cell ID
- Solution:
  - Radix Sort
  - Radix Sort ensures identical cell IDs remain in the same order as before sorting.

Sorting Cell Array



Cell ID Array

🟨 01 0 0	🟨 01 1 1	🟨 11 1 2	🟨 10 1 3	🟨 02 1 4	…	🟨 02 1 n
🟩 02 0 0	⬜	🟩 11 0 2	🟩 10 0 3	🟩 01 1 4		🟩 01 1 n
🟩 01 1 0	⬜	🟩 10 0 2	⬜	⬜		🟩 02 1 n
🟩 02 1 0	⬜	🟩 00 0 2	⬜	⬜		🟩 11 1 n
⬜	⬜	🟩 00 1 2	⬜	⬜		🟩 02 2 n
⬜	⬜	🟩 10 1 2	⬜	⬜		⬜
⬜	⬜	🟩 01 1 2	⬜	⬜		⬜
⬜	⬜	🟩 01 0 2	⬜	⬜		⬜

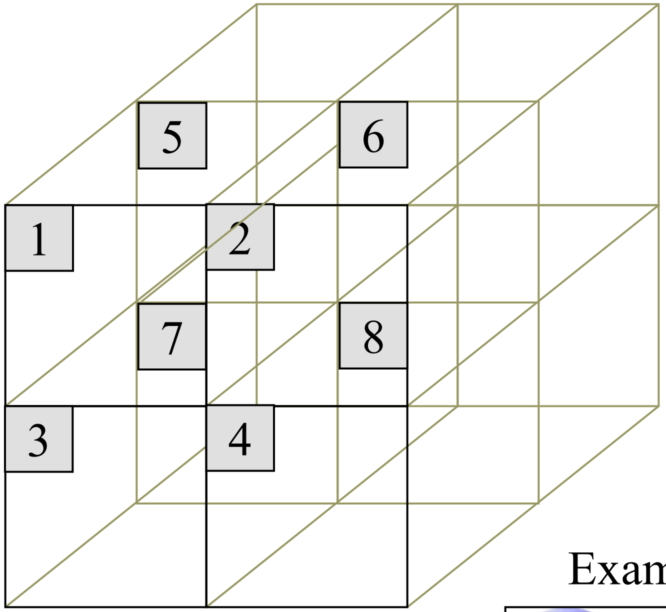
Sorted Cell ID Array

🟩 00 0 2	🟩 01 1 n	🟨 10 1 3	…	⬜	⬜	⬜
🟩 00 1 2	🟩 02 0 0	🟩 10 1 2		⬜	⬜	⬜
🟨 01 0 0	🟨 02 1 4	🟩 11 0 2		⬜	⬜	⬜
🟩 01 0 2	🟨 02 1 n	🟨 11 1 2		⬜	⬜	⬜
🟨 01 1 1	🟩 02 1 0	🟩 11 1 n		⬜	⬜	⬜
🟩 01 1 0	🟩 02 2 n	🟩 11 1 n		⬜	⬜	⬜
🟩 01 1 2	🟩 10 0 2	🟩 10 2 n		⬜	⬜	⬜
🟩 01 1 4	🟩 10 0 3	🟩 10 3 3		⬜	⬜	⬜

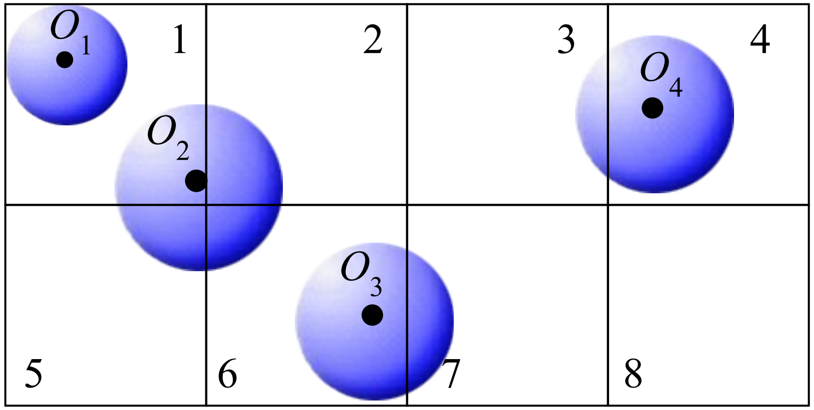
Legend

⬜	Invalid Cell
🟨 01 1 1	Home Cell
🟩 10 0 2	Phantom Cell
🟩 10 3 3	Cell ID Object ID

Spatial Subdivision



Example



1. Assign to each cell the list of bounding volumes whose objects intersect with the cell
2. Perform Collision test only if both objects are in the cell and one has a centroid in the cell

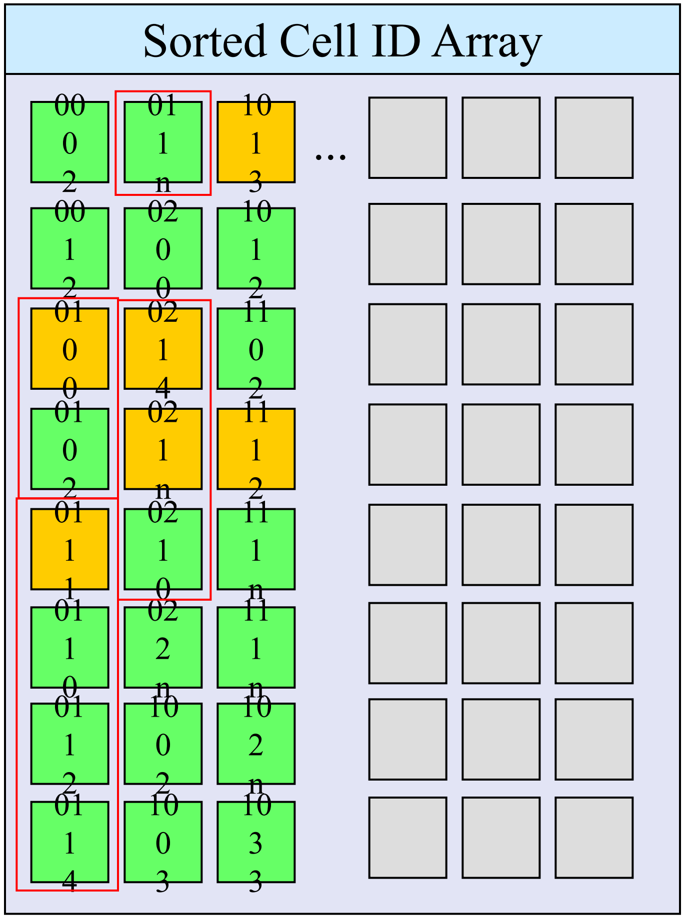
Images from pg 699, 700 GPU Gems III

Create the Collision Cell List

- Scan sorted cell ID array for changes of cell ID
  - Mark by end of the list of occupants of one cell and beginning of another

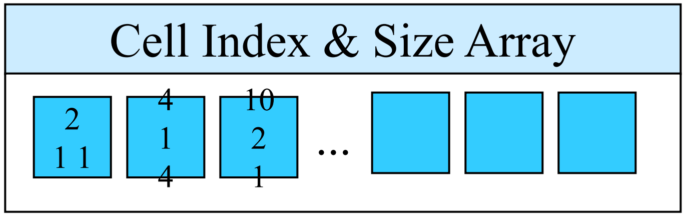
1. Count number of objects each collision cell contains and convert them into offsets using scan
2. Create entries for each collision cell in new array
   1. Start
   2. Number of H occupants
   3. Number of P occupants

Create Collision Cell List



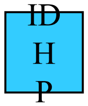
Sorted Cell ID Array

00 0 2	01 1 n	10 1 3	…
00 1 2	02 0 0	10 1 2	
01 0 0	02 1 4	11 0 2	
01 0 2	02 1 n	11 1 2	
01 1 1	02 1 0	11 1 n	
01 1 0	02 2 n	11 1 n	
01 1 2	10 0 2	10 2 n	
01 1 4	10 0 3	10 3 3	



Cell Index & Size Array

2 1 1	4 1 4	10 2 1	…

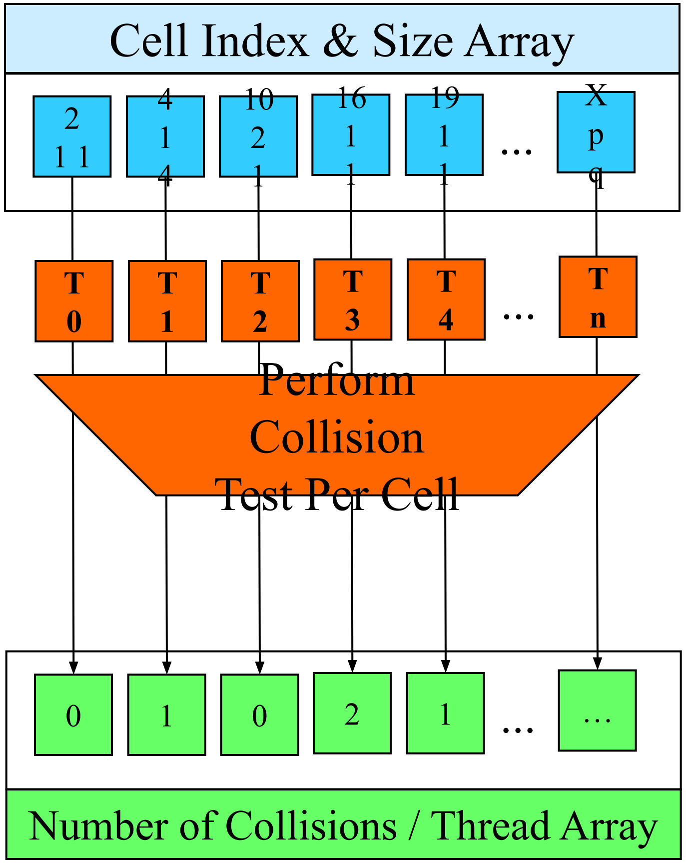


ID = Cell index in sorted Cell ID Array

H = Number of Home Cell IDs

P = Number of Phantom Cell IDs

Traverse Collision Cell List



Cell Index & Size Array

2
1 1

4
1
4

10
2
1

16
1
1

19
1
1

…

X
p
q

T 0

T 1

T 2

T 3

T 4

…

T n

Perform Collision Test Per Cell

0

1

0

2

1

…

…

Number of Collisions / Thread Array