Eurographics 2013

May 6-10, Girona (Spain)

# Lazy Visibility Evaluation for Exact Soft Shadows

Frédéric Mora, Lilian Aveneau,

Oana Apostu, Djamchid Ghazanfarpour


Soft shadows

- underlying problem: area light source visibility
- visibility accuracy / soft shadows quality

A common solution: sampling

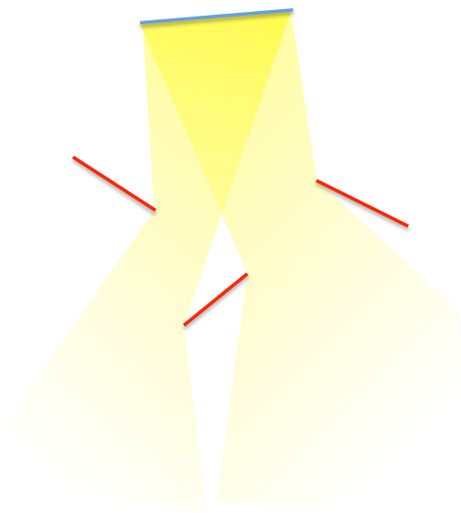
- noise sensitive
- can become expensive

Our context is high quality soft shadows computation. This implies to determine the visibility of an area light source from any point in a scene. And the soft shadows quality highly depends on the visibility accuracy.

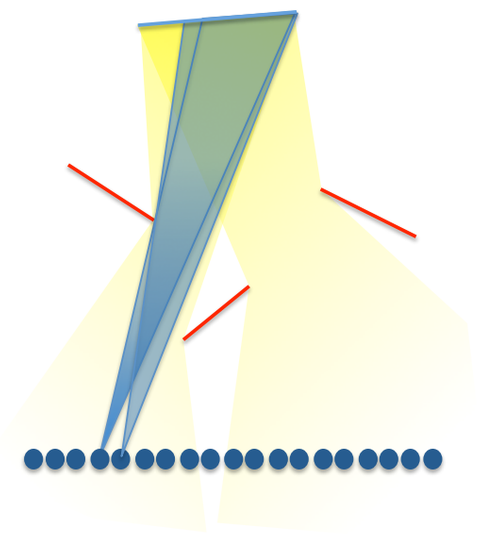
A common solution is to sample the area light source and cast shadow rays. It is easy and robust. However, as any sampling approach, it is sensitive to noise. Many samples may be required to avoid the noise, increasing the computation time.  Of course, many works on this topic address this issue.

# Introduction > our approach, main ideas

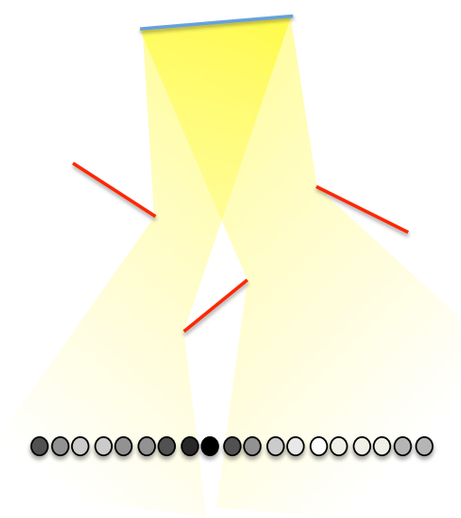
- coherent visibility representation from/to an area light source



- contains all the possible light views



- extract light visibility to compute analytic direct illumination



In this work, we propose to explore a completly different approach.

We compute analytic visibility from the area light source. This allows to find the visible parts of the light source from any point, taking advantage of the visibility coherence between neighboring points. The visible parts of the light are then used to compute analytic direct illumination.

Exact from-polygon visibility

- a 4D problem, complex
- 2002 Nirenstein / Bittner (target scenario: exact PVS)
- CSG computations (substractions of 5D convex polyhedrons)

Common drawbacks

- very expensive (preprocess)
- robustness issues
- implementation is complicated

So, exact from-polygon visibility is the underlying problem. However this is a challenging one because it is 4D and thus very complex.

The two first practical solutions were introduced by Nirenstein and Bittner in two thousand two.

All the previous works on this topic rely on CSG computations in five dimensions.

This explains several drawbacks, because this is very expensive and prone to numerical inaccuracy. In addition, this is complicated to implement properly.

Analytic from-polygon visibility

- starts over from the theory
- no 5D CSG computations ⇨ easy, robust, efficient
- no preprocess



Clearly, 5D CSG is the weak point.

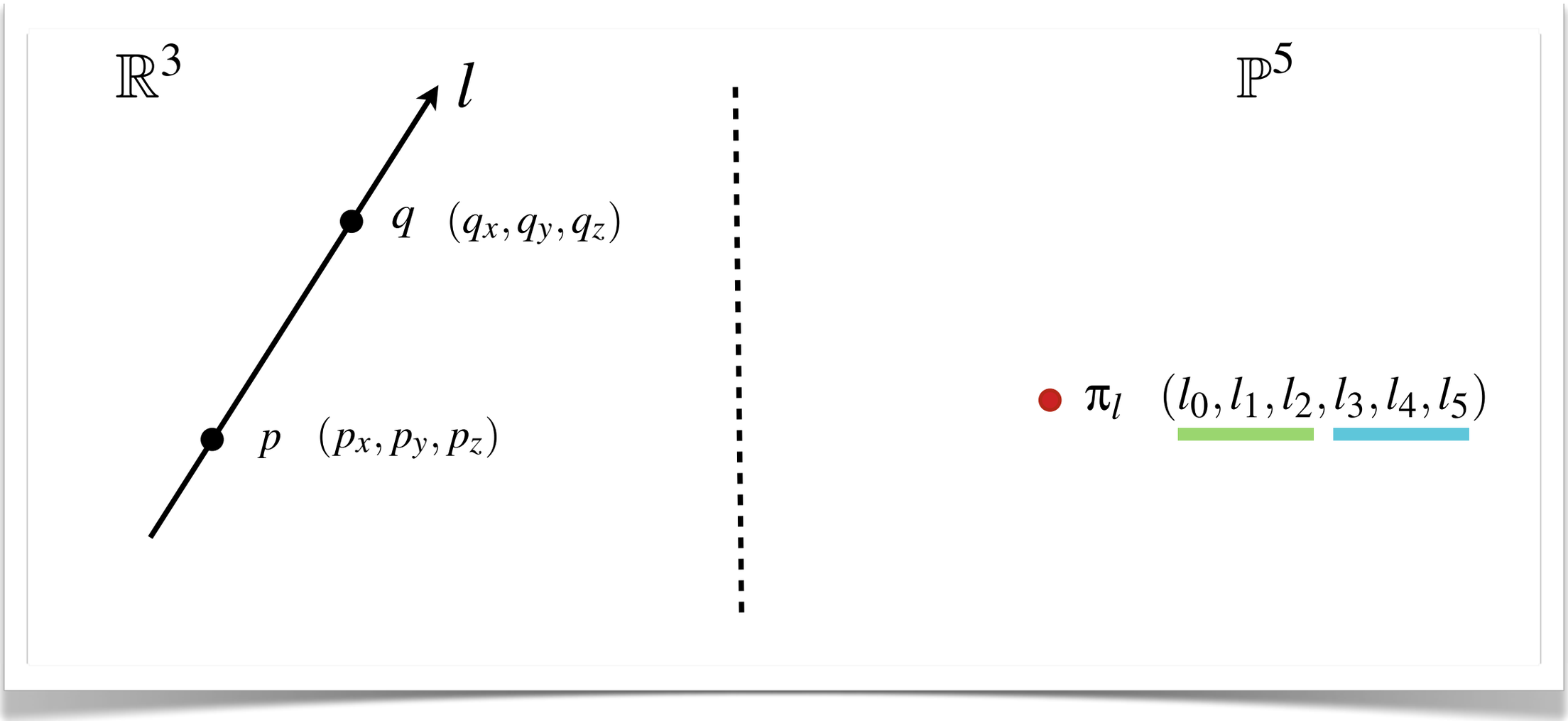
Therefore we propose to start over from the theory in order to implement an approach which does not use any 5D CSG computations. We want a method which is easy to implement, robust and efficient, without any preprocess.

# Outline >

- Geometric basis
- Visibility algorithm
- Results

I will have to present or recall some geometric basis, mainly the Plucker Coordinates.
Next I will explain our visibility algorithm, followed by some results.

# Geometrical basis > plucker coordinates of an oriented line

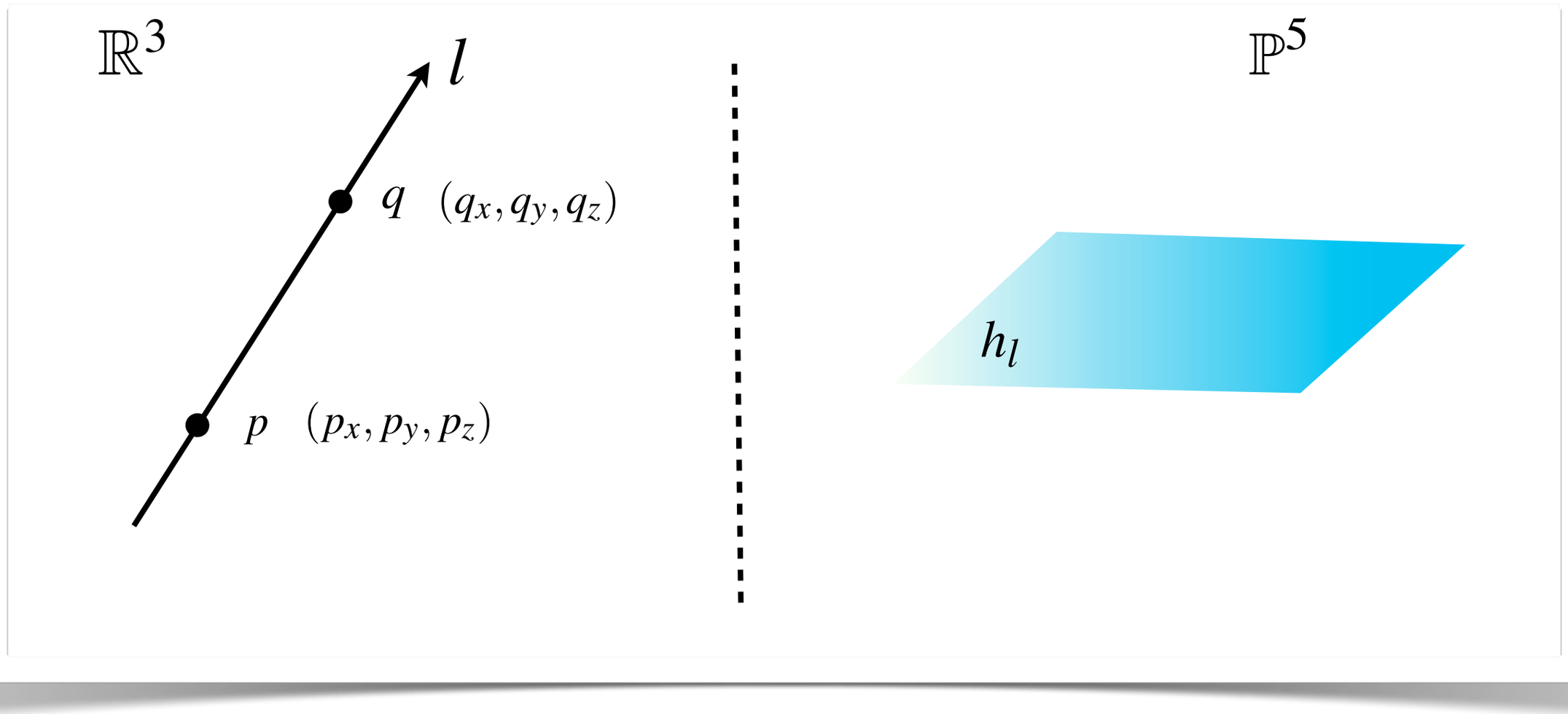



$$\overrightarrow{pq} \quad \begin{vmatrix} l_0 = q_x - p_x \\ l_1 = q_y - p_y \\ l_2 = q_z - p_z \end{vmatrix} \quad \begin{vmatrix} l_3 = q_z p_y - q_y p_z \\ l_4 = q_x p_z - q_z p_x \\ l_5 = q_y p_x - q_x p_y \end{vmatrix} \quad \overrightarrow{op} \times \overrightarrow{oq}$$

So the Plucker space is five dimensional projective space of lines.

A 3D line maps to a point in the plucker space.
The six coordinates are not very intuitive but if we look closely, we can see that the three first coordinates are the line direction while the three last ones encode the line location.

# Geometrical basis > point - hyperplane duality

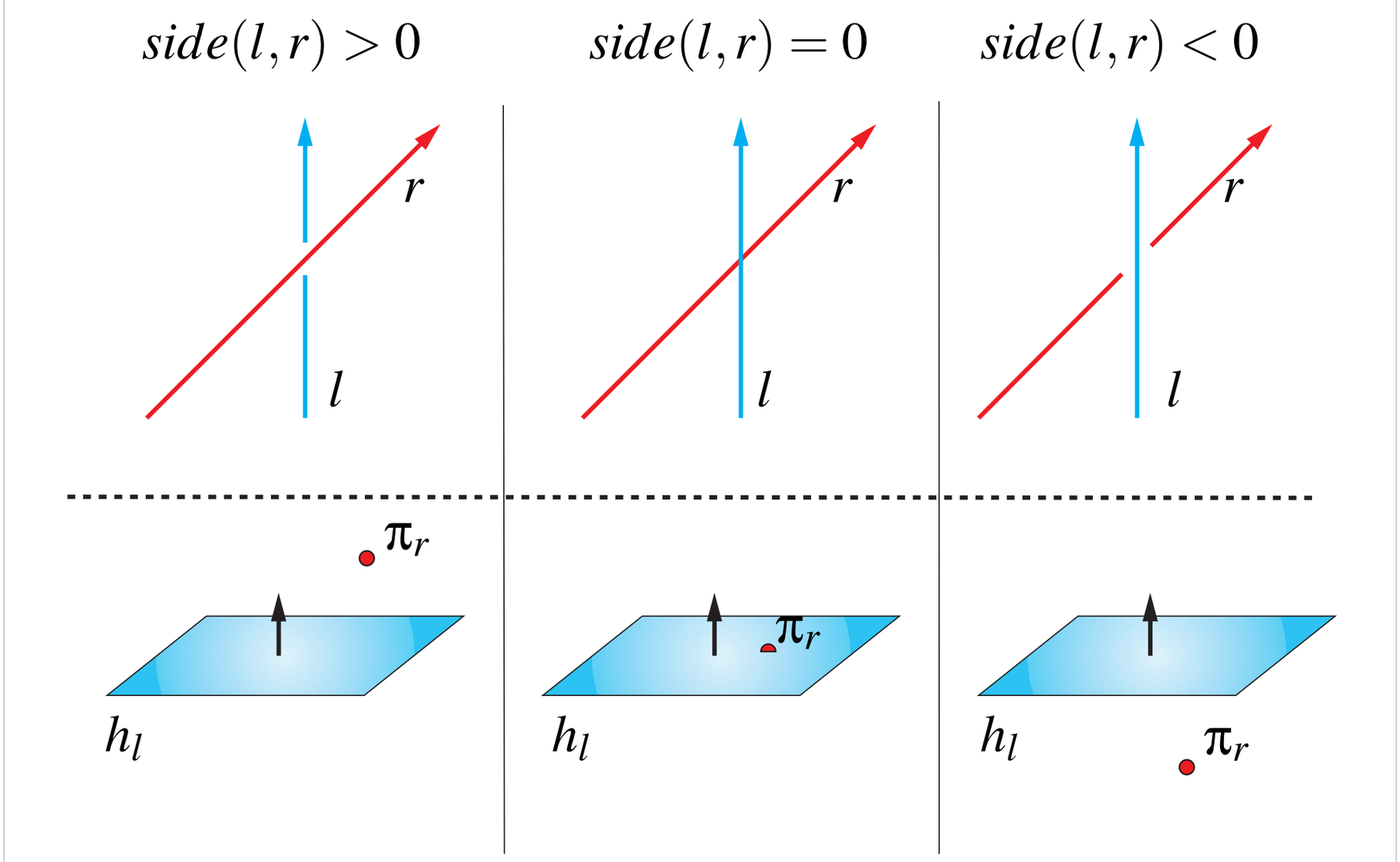



- $\pi_l \quad (l_0, l_1, l_2, l_3, l_4, l_5)$

$$h_l(x) = l_3x_0 + l_4x_1 + l_5x_2 + l_0x_3 + l_1x_4 + l_2x_5 = 0$$

In the plucker space, any line can also be represented by a hyperplane.
This defines a dual representation as a point or as a hyperplane.

# Geometrical basis > relative orientation of lines




$$side(l,r) = l_3r_0 + l_4r_1 + l_5r_2 + l_0r_3 + l_1r_4 + l_2r_5$$

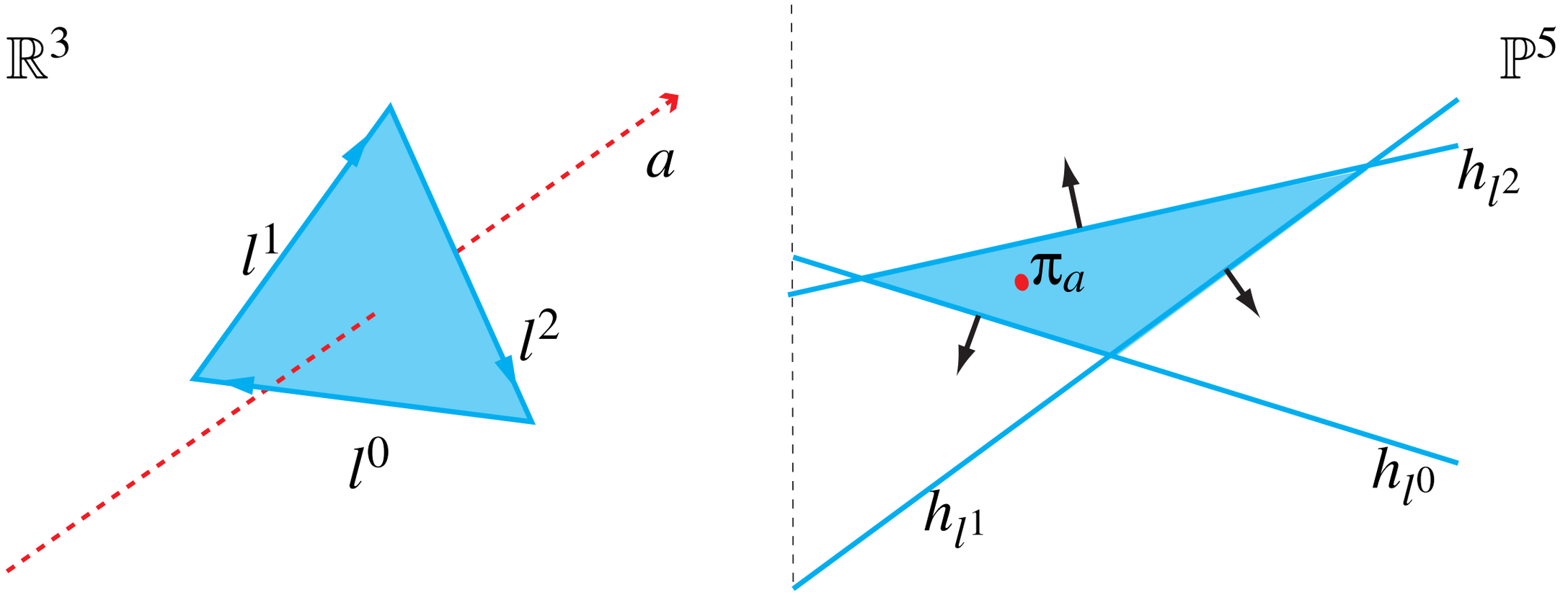
$$= h_l(\pi_r)$$

Plucker coordinates are often used for the so called «side operator». This allows to compute the relative orientation of two lines. In other words, it describes how a line «turns» around another one.

This is not an intuitive notion in 3D. To simplify, let's consider two lines and let's imagine we are walking along the first one. At a time, we will left the second line on our right or on our left. Eventually we will encounter the other line if the two lines are incident. The side operator helps to distinguish between these tree possibilities: if it is positive, the two lines have a positive orientation, if it is null they are incident, if it is negative they have a negative orientation.

In the plucker space, the side operator has a much simpler geometric interpretation. indeed, the side operator is equivalent to test the plucker point of one line against the hyperplane of the other one. It is clear that three cases can occur: either the point is in the positive halfspace, either it is incident to the hyperplane, either it is in the negative halfspace.

# Geometrical basis > line-triangle intersection test

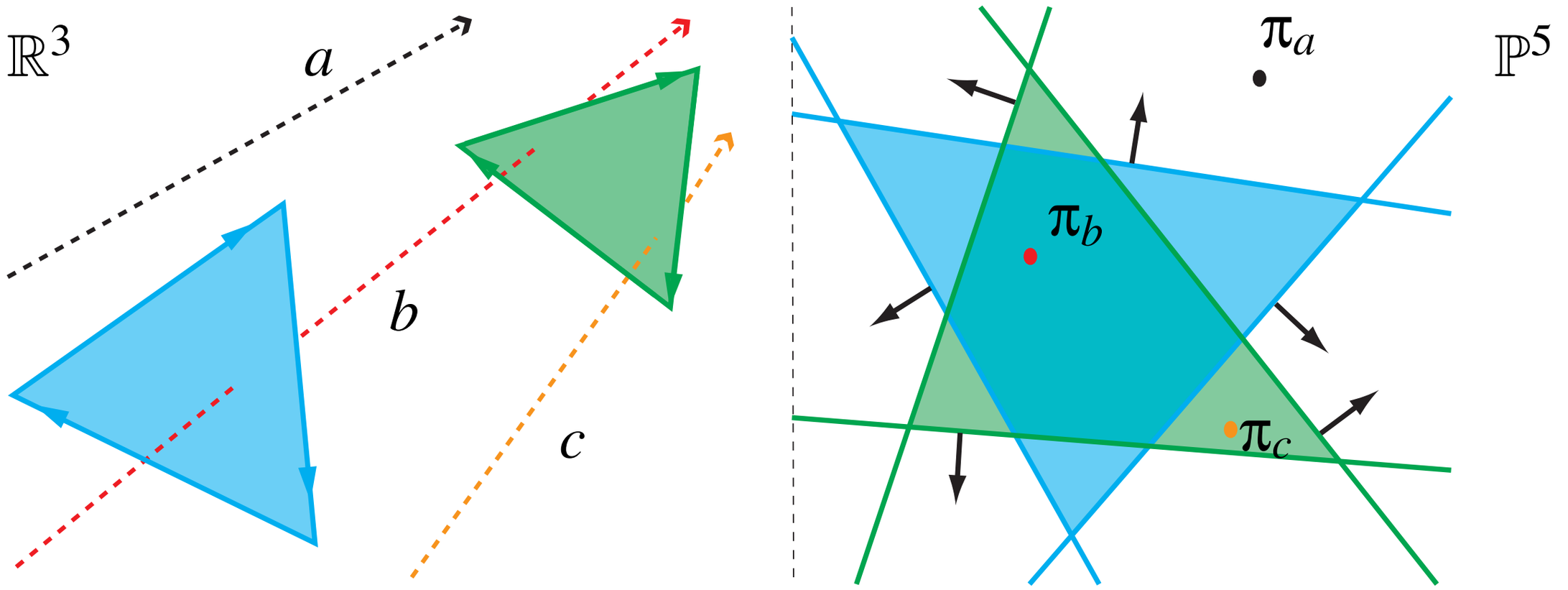



- lines stabbing a triangle...
- ... have their plucker point inside the convex polyhedron...
- ... whose hyperplanes are the lines spanning the triangle edges

The plucker coordinates can be used to make a robust triangle-line intersection test.
Once again, we can imagine a nice walk along a triangle edges. All the lines stabbing the triangle will be on the same side.

Here is the geometric interpretation in the plucker space:
Each line spanning a triangle edge can be mapped to a hyperplane.
And this three hyperplanes define a convex polyhedron.
All the lines stabbing the triangle have a consistent orientation with respect to the three hyperplanes. This means that their plucker points are inside the polyhedron bounded by the hyperplanes

# Geometrical basis > lines stabbing triangles




## Pellegrini

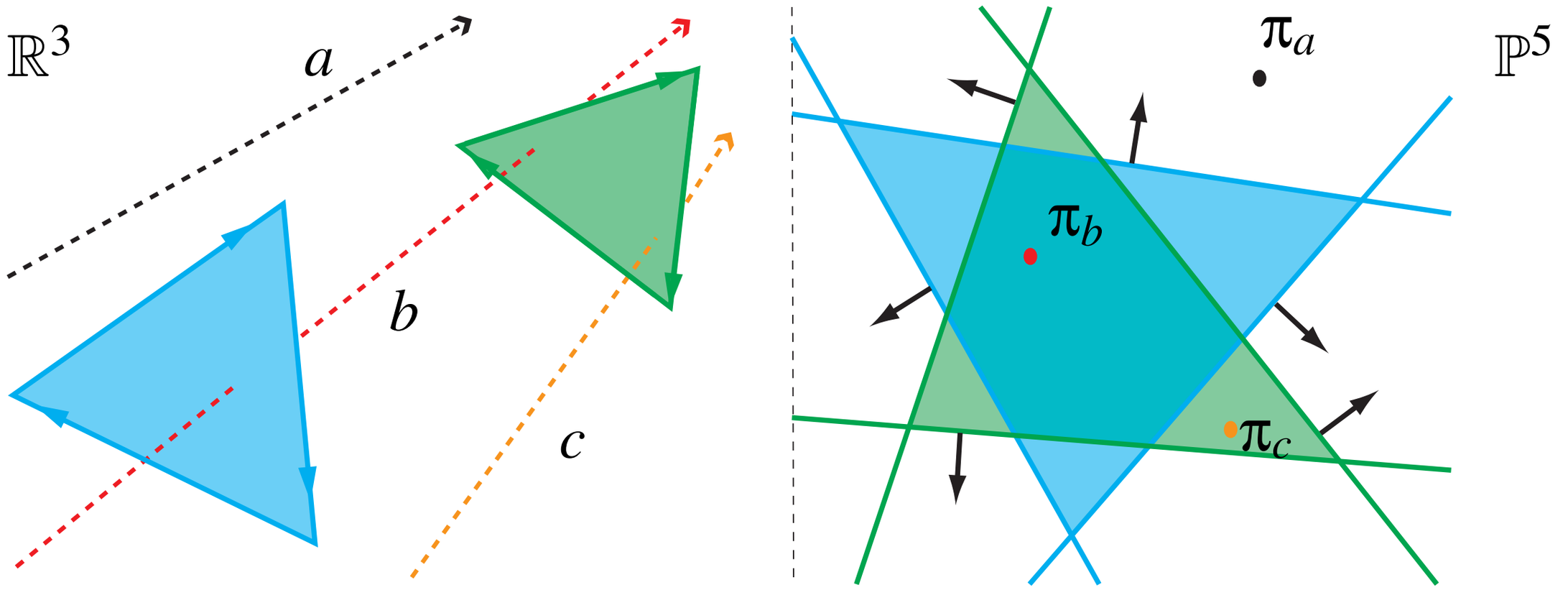
- lines spanning triangle edges ➡ arrangement of hyperplanes
- lines in a same cell intersect the same subset of triangles

This line-triangle intersection test extends to a set of triangles or convex polygons. This is well explained by Pellegrini.

If all the lines spanning the triangle edges are mapped to hyperplanes, they define an arrangement of hyperplanes which subdvide the plucker space into convex cells. And there is an important property: points in a given cell are lines stabbing a same subset of triangles. And if this subset is empty, they are lines missing all the triangles.

For example, any line stabbing the blue and the green triangles belongs to the same cell than the red line b.

# Geometrical basis > lines stabbing triangles



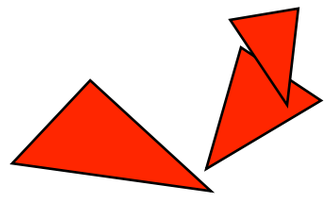

- equivalence relation on lines
- lines are grouped according to the geometry they intersect

The arrangement of hyperplanes induces an equivalence relation on lines.

Using this equivalence relation we can group the lines according to the geometry they intersect.

# Geometrical basis > in short...

- take some triangles

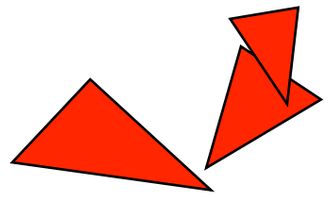




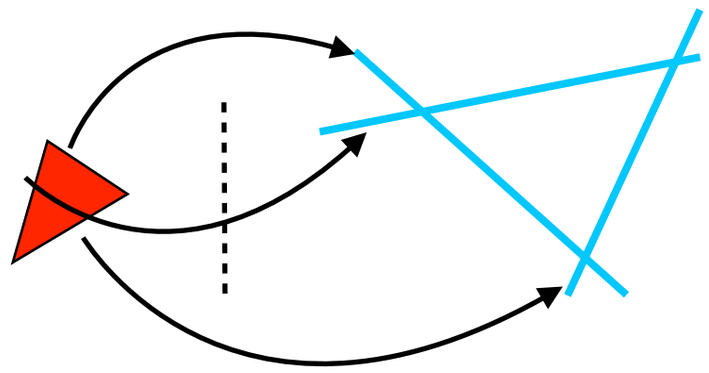
To sum-up:
Given a set of triangles,
map the line spanning their edges to hyperplanes in the plucker space
each cell of the related arrangement contains lines that hit or miss the same subset of triangles

# Geometrical basis > in short...

- take some triangles



- map the lines spanning their edges to plucker hyperplanes

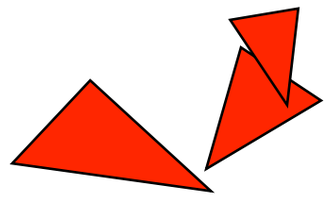




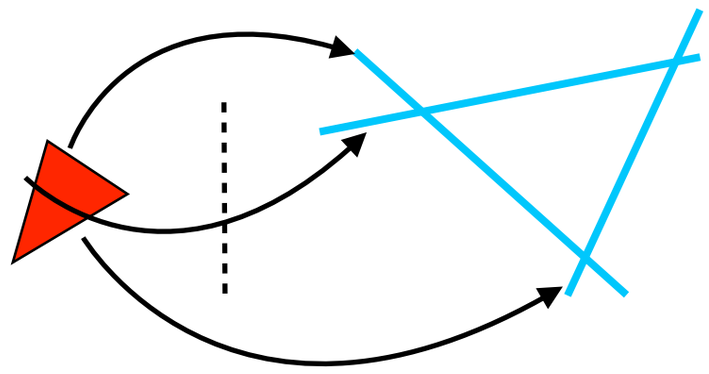
To sum-up:
Given a set of triangles,
map the line spanning their edges to hyperplanes in the plucker space
each cell of the related arrangement contains lines that hit or miss the same subset of triangles

# Geometrical basis > in short...

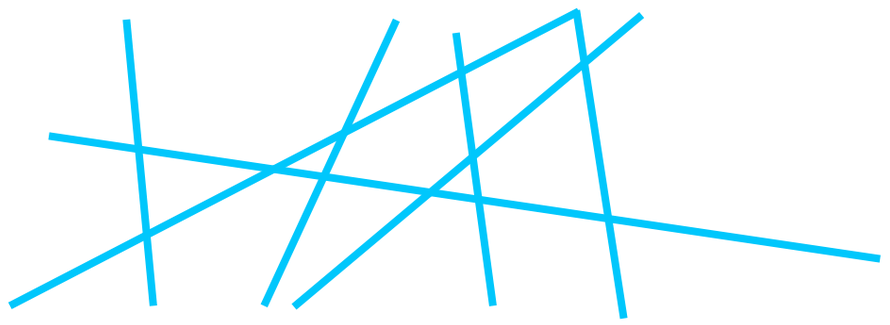
- take some triangles



- map the lines spanning their edges to plucker hyperplanes



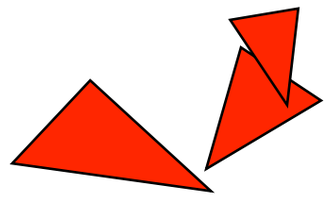
- this defines an arrangement of hyperplanes in Plucker space



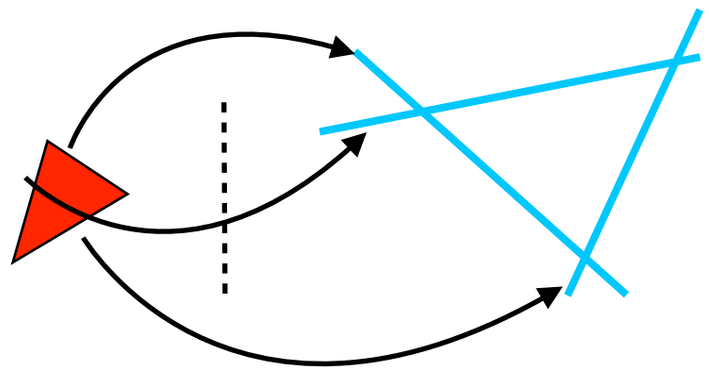
To sum-up:
Given a set of triangles,
map the line spanning their edges to hyperplanes in the plucker space
each cell of the related arrangement contains lines that hit or miss the same subset of triangles

# Geometrical basis > in short...

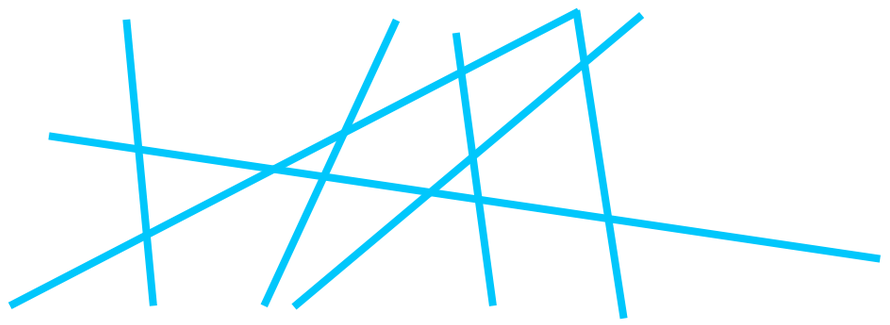
- take some triangles



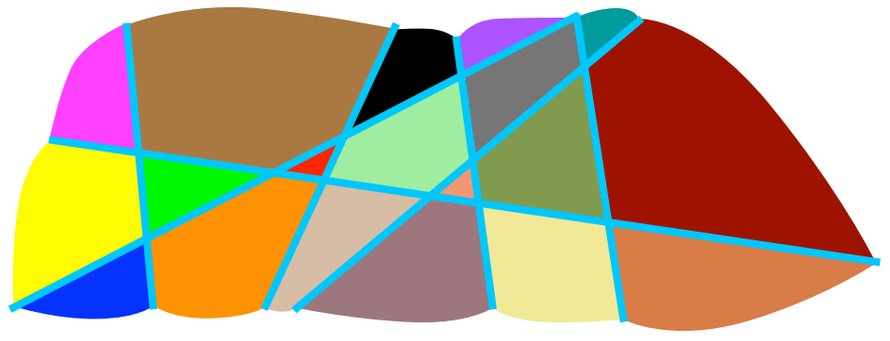
- map the lines spanning their edges to plucker hyperplanes



- this defines an arrangement of hyperplanes in Plucker space



- each cell contains lines that hit/miss the same subset of triangles



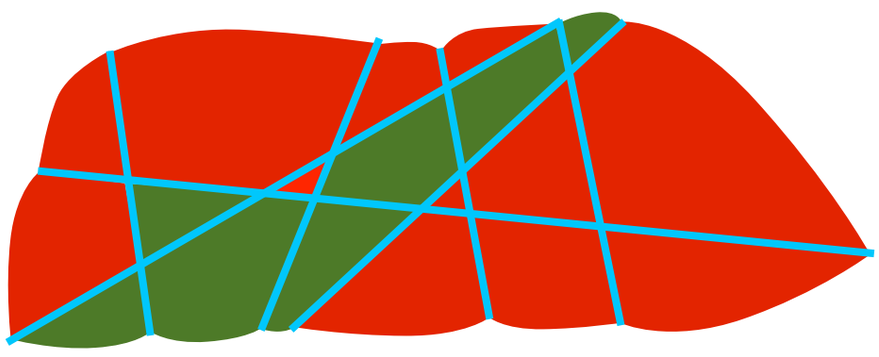
To sum-up:
Given a set of triangles,
map the line spanning their edges to hyperplanes in the plucker space
each cell of the related arrangement contains lines that hit or miss the same subset of triangles

# Geometrical basis > in short...

Equivalence relation on lines

- one cell ⇨ one equivalence class
- equivalent lines are coherent paths through the triangles

We want to compute equivalence classes representing coherent paths from/to an area light source through its occluders



One cell corresponds to one equivalence class and thus equivalent lines In a visibility context, equivalent lines are coherent paths through the triangles.

We want to compute the paths from the area light source which are not blocked by any geometry. This means computing the equivalence classes which represent sets of line not stabbing any triangle and sets of lines stabbing at least one triangle.

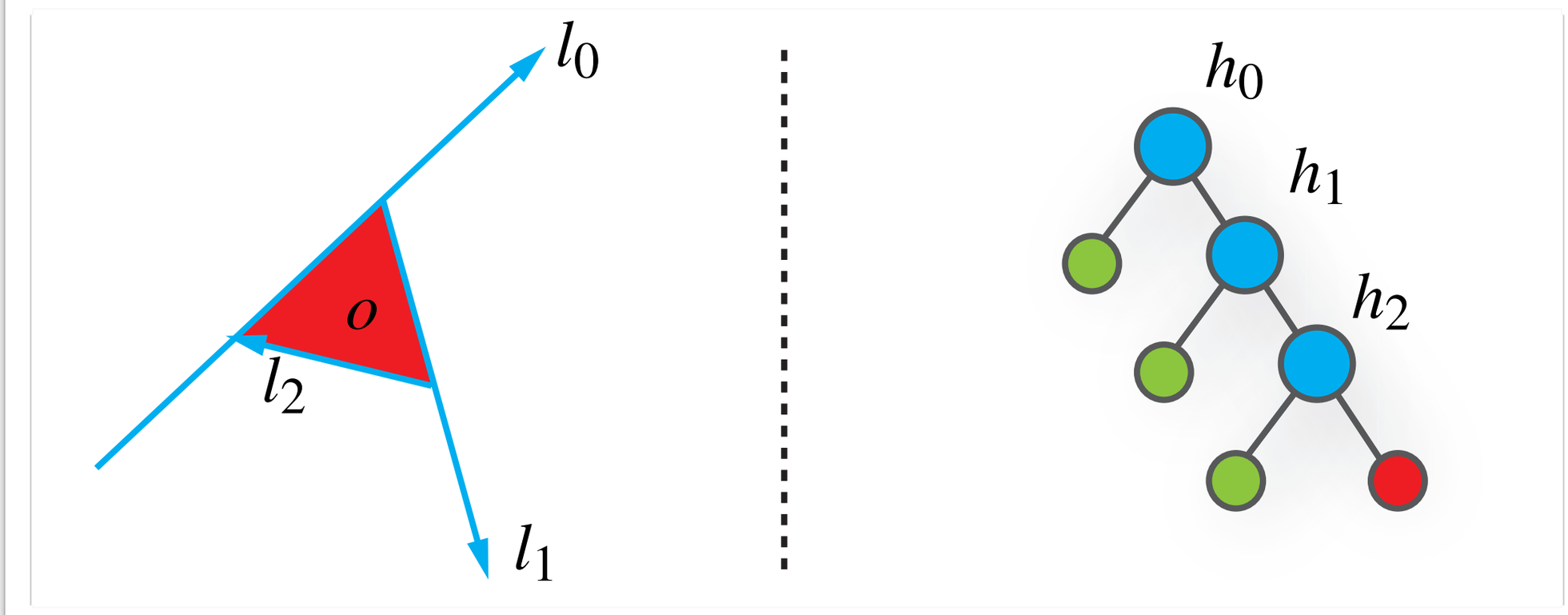
# Outline >

- Geometric basis
- Visibility algorithm
- Results



We can now detail our visibility algorithm

# Visibility algorithm > BSP representation of an occluder




hyperplanes (inner nodes)

visible classes (leaves)

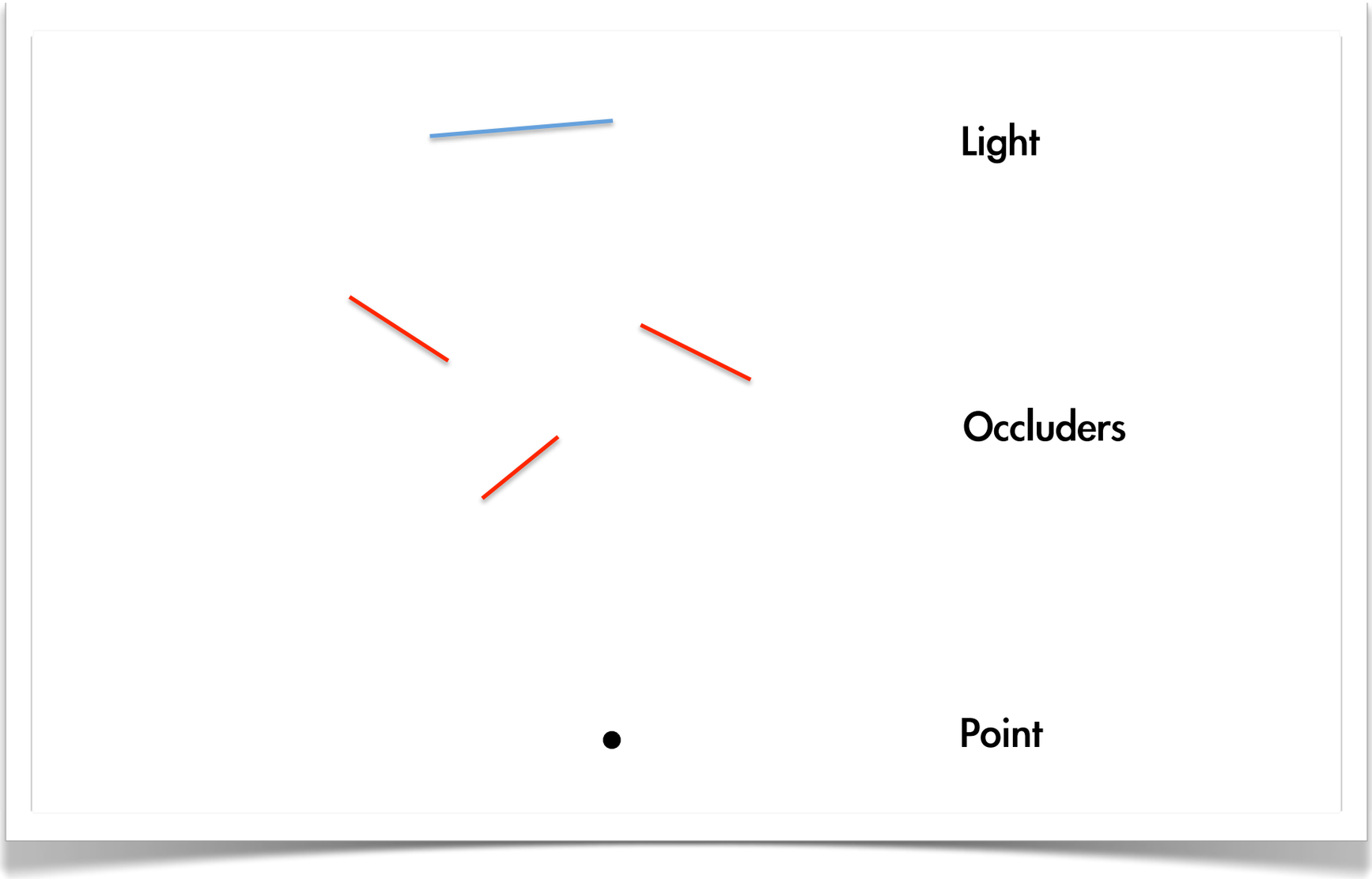
invisible classes (leaves)

First, let’s consider a single occluder. We can use a Binary Space Partionning tree to represent the equivalence classes induced by this occluder.

Each node stores one hyperplane from an occluder edge.

Each leaf correspond to one equivalence class. The left leaves are visible classes, they represent the lines missing the occluder. The right leaf is an invisible class, it represents the lines blocked by the occluder.

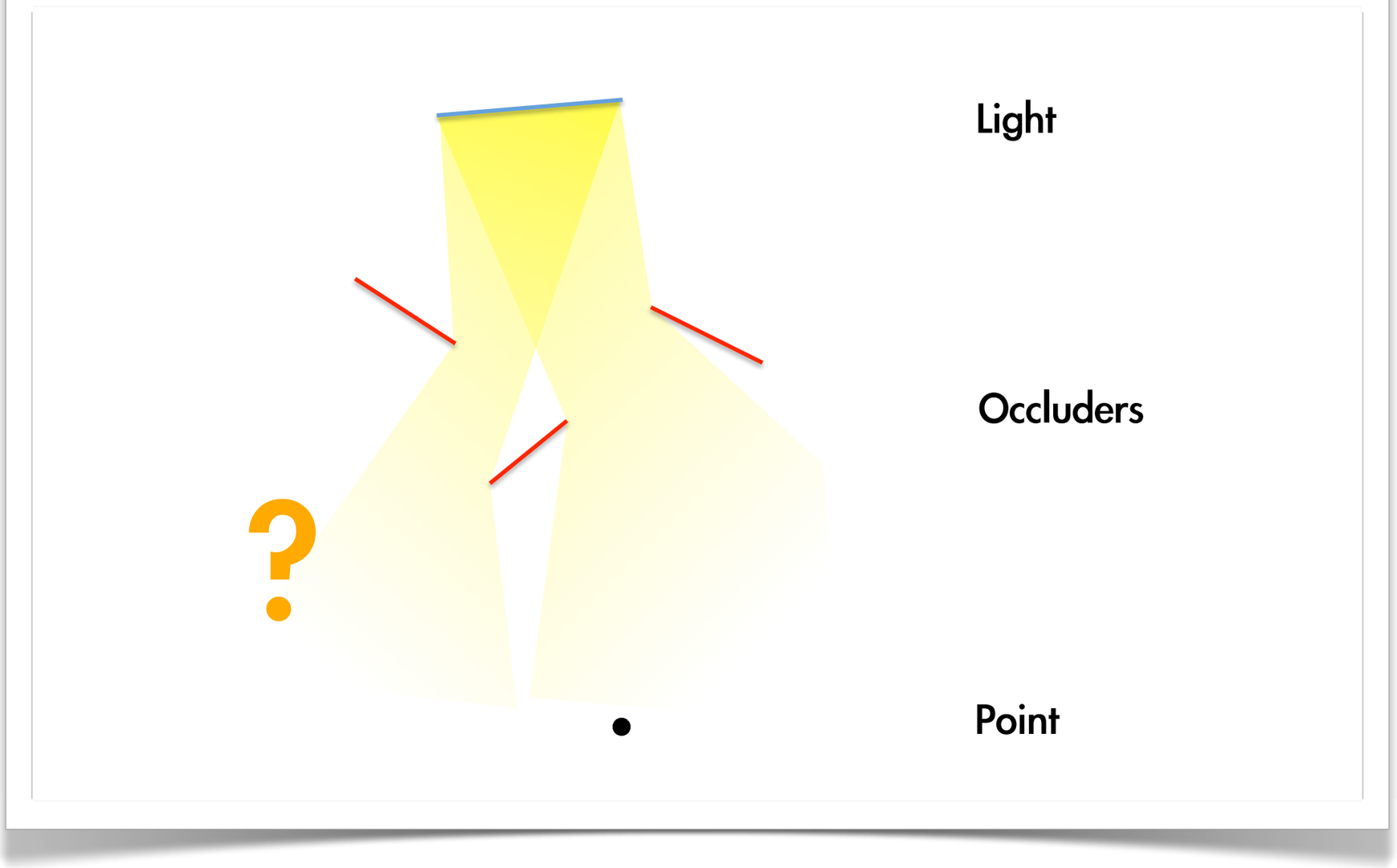
# Visibility algorithm > principle




- visibility from the area light source ?
- light visibility from the point ?

Here is an example:
We have an area light source, a set of occluders and a point.

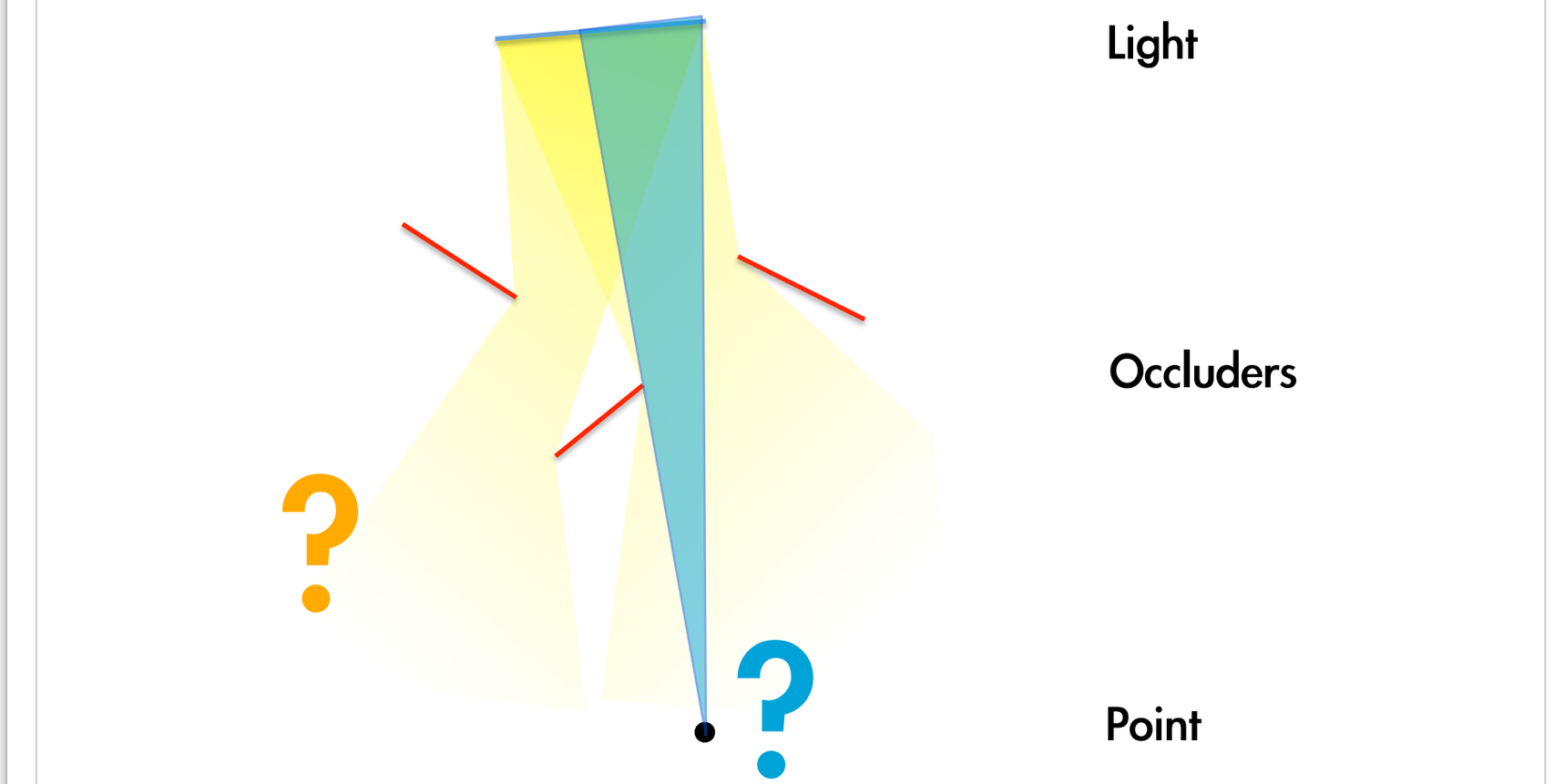
# Visibility algorithm > principle




- visibility from the area light source ?
- light visibility from the point ?

Two questions:
 - what are the visible or invisible classes corresponding to the visibility from the area light source ?

# Visibility algorithm > principle

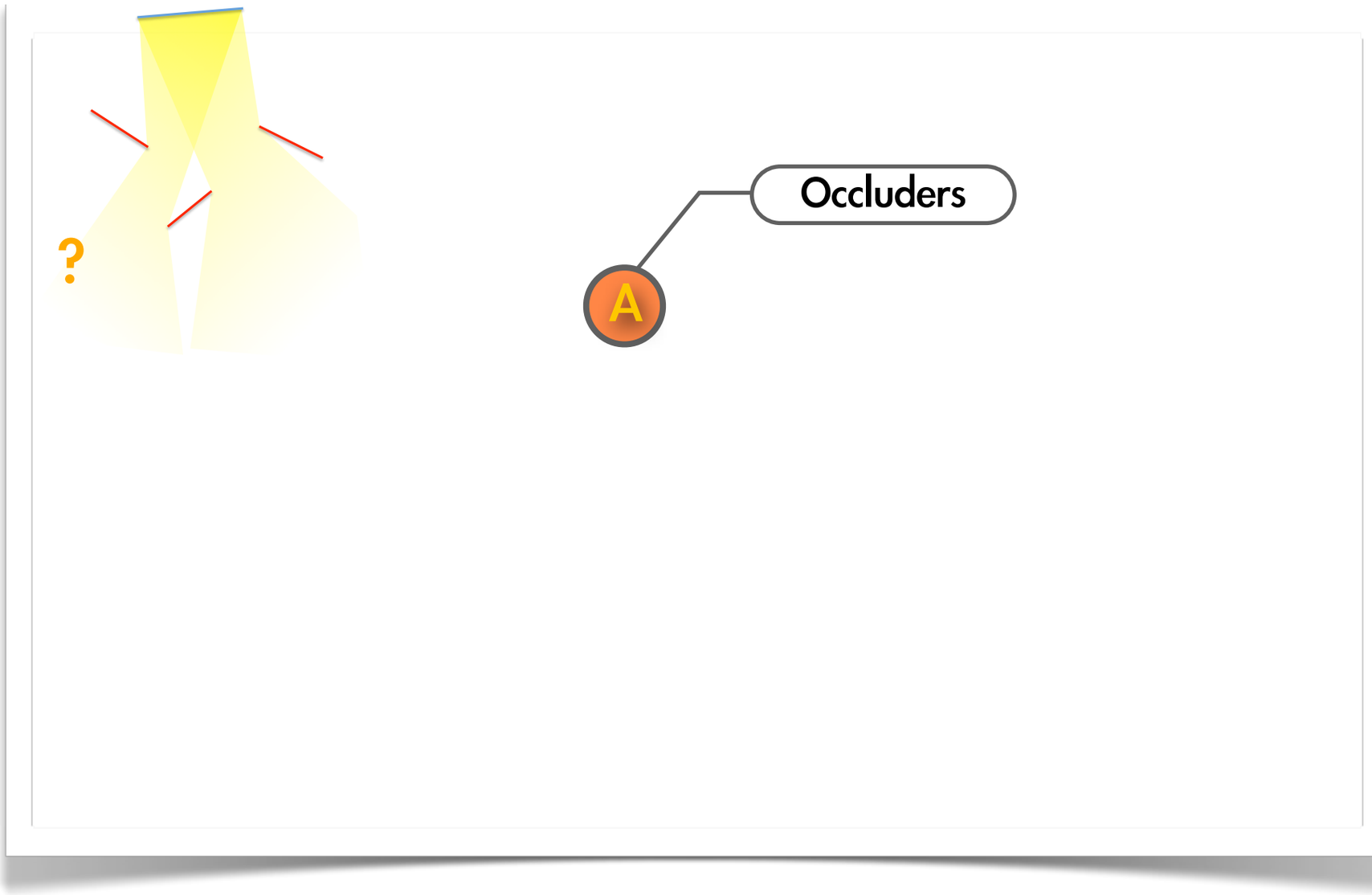



- visibility from the area light source ?
- light visibility from the point ?

– what parts of the light are visible from the point ?

Our solution solves both these two problems.

# Visibility algorithm > principle



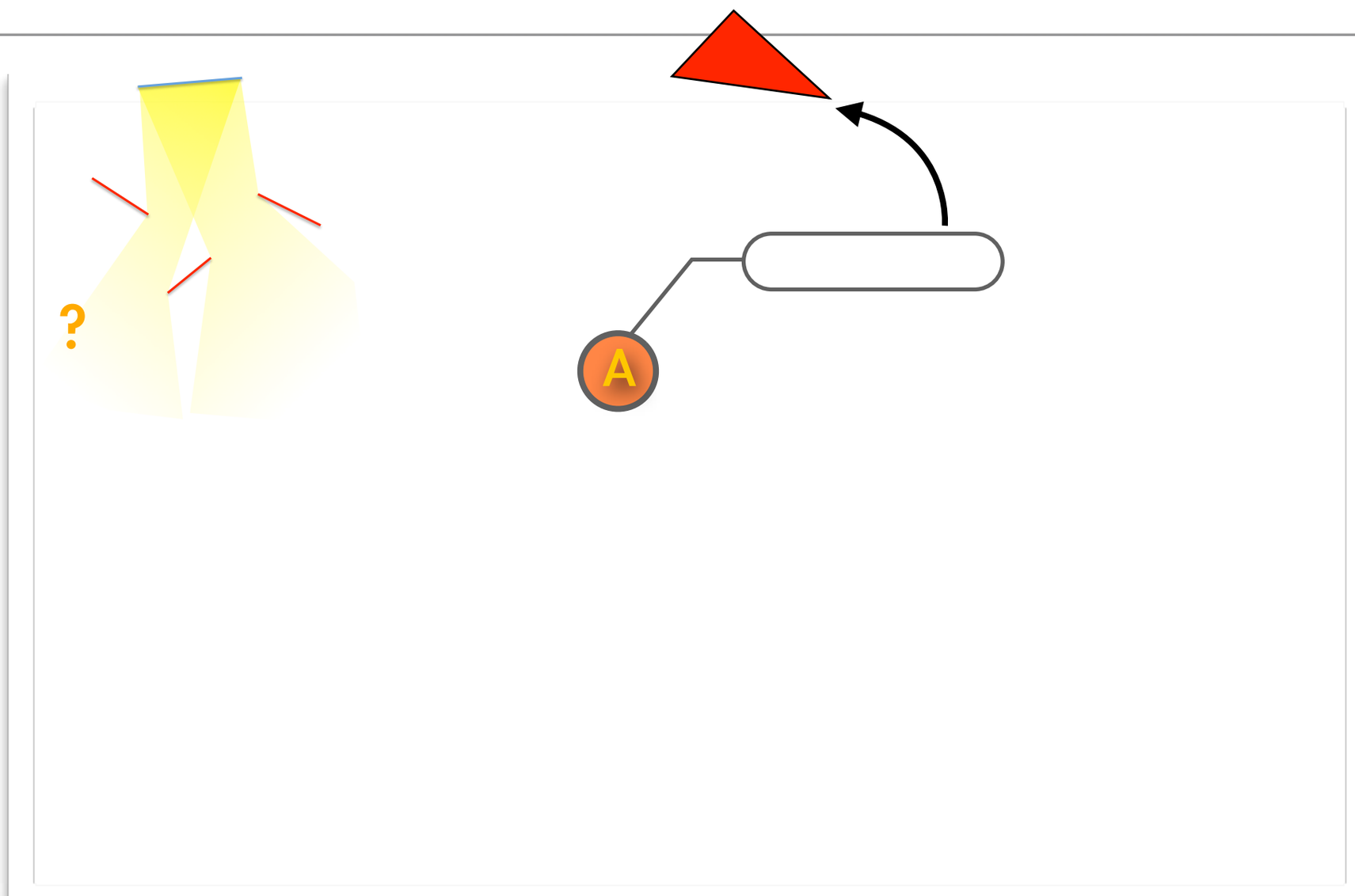

- start with an undefined class and all the occluders

At first we focus on the first problem by looking for the classes representing the visibility from the light.

This is done by lazily growing a BSP tree in the plucker space.

The algorithm starts with a single leaf associated with all the occluders. Since occluders remain, the visibility from the light is undefined.

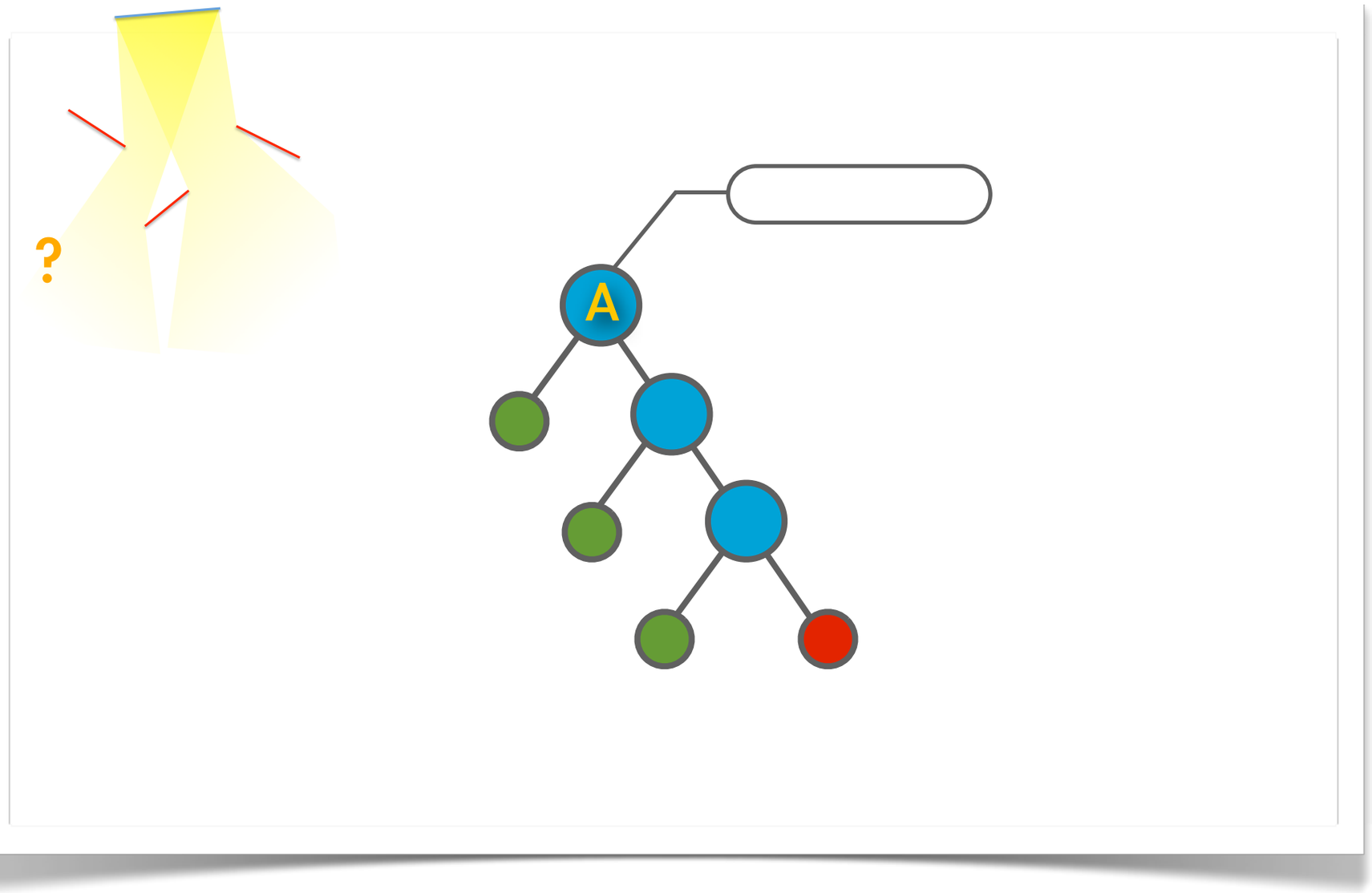
# Visibility algorithm > principle




- select a random occluder

And since the visibility is undefined, we have to grow the BSP tree.
We select randomly an occluder...

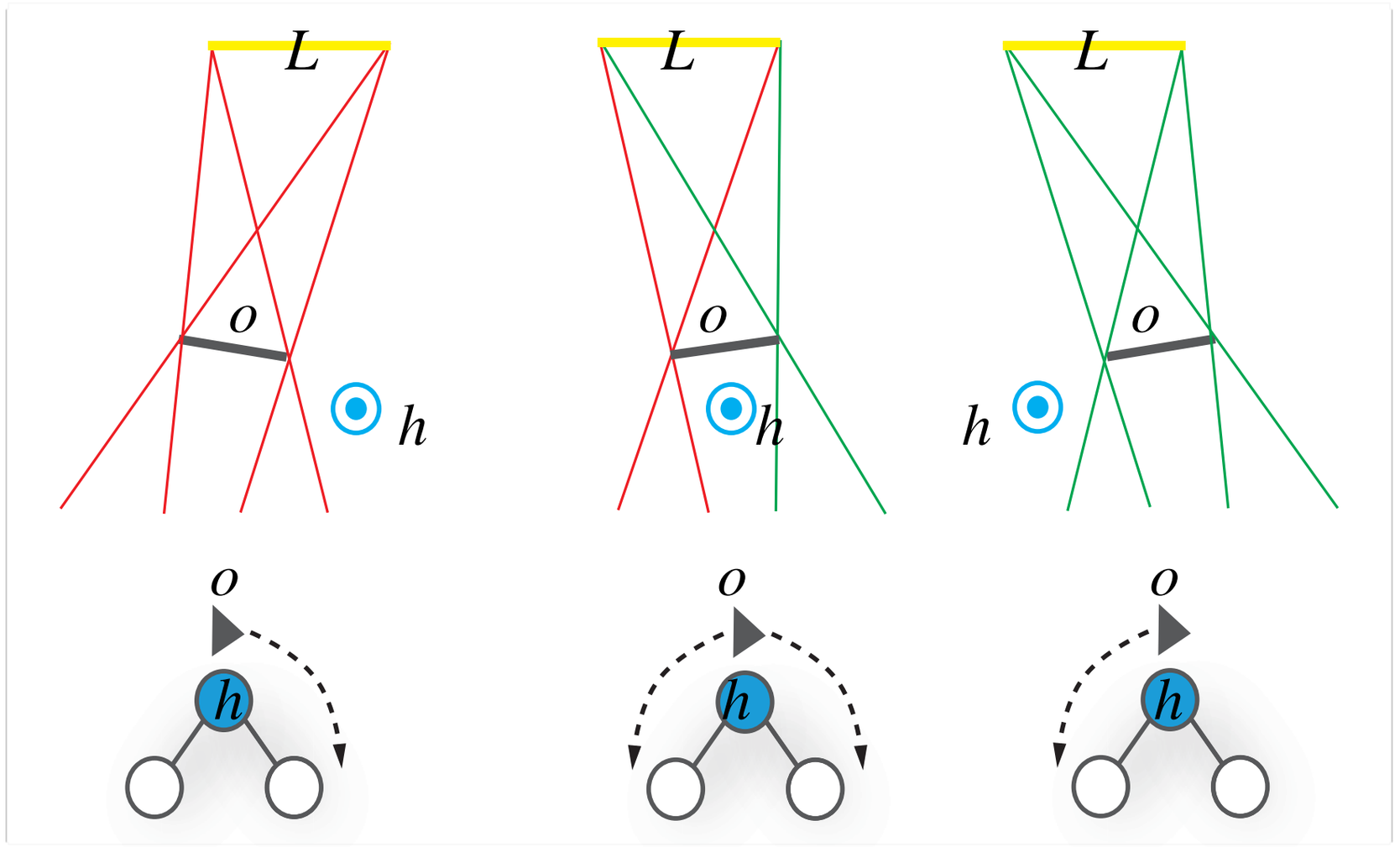
# Visibility algorithm > principle




- replace the leaf with the occluder BSP representation
- remaining occluders can affect the visible classes

... and replace the leaf with its BSP representation, inserting the related classes.
Then the remaining occluders have to be inserted into the tree to check wether they affect the newly added equivalence classes.

# Visibility algorithm > occluded lines orientation




- orientation of the occluded lines with respect to h ?
- computing the vertex-to-vertex lines orientation is sufficient

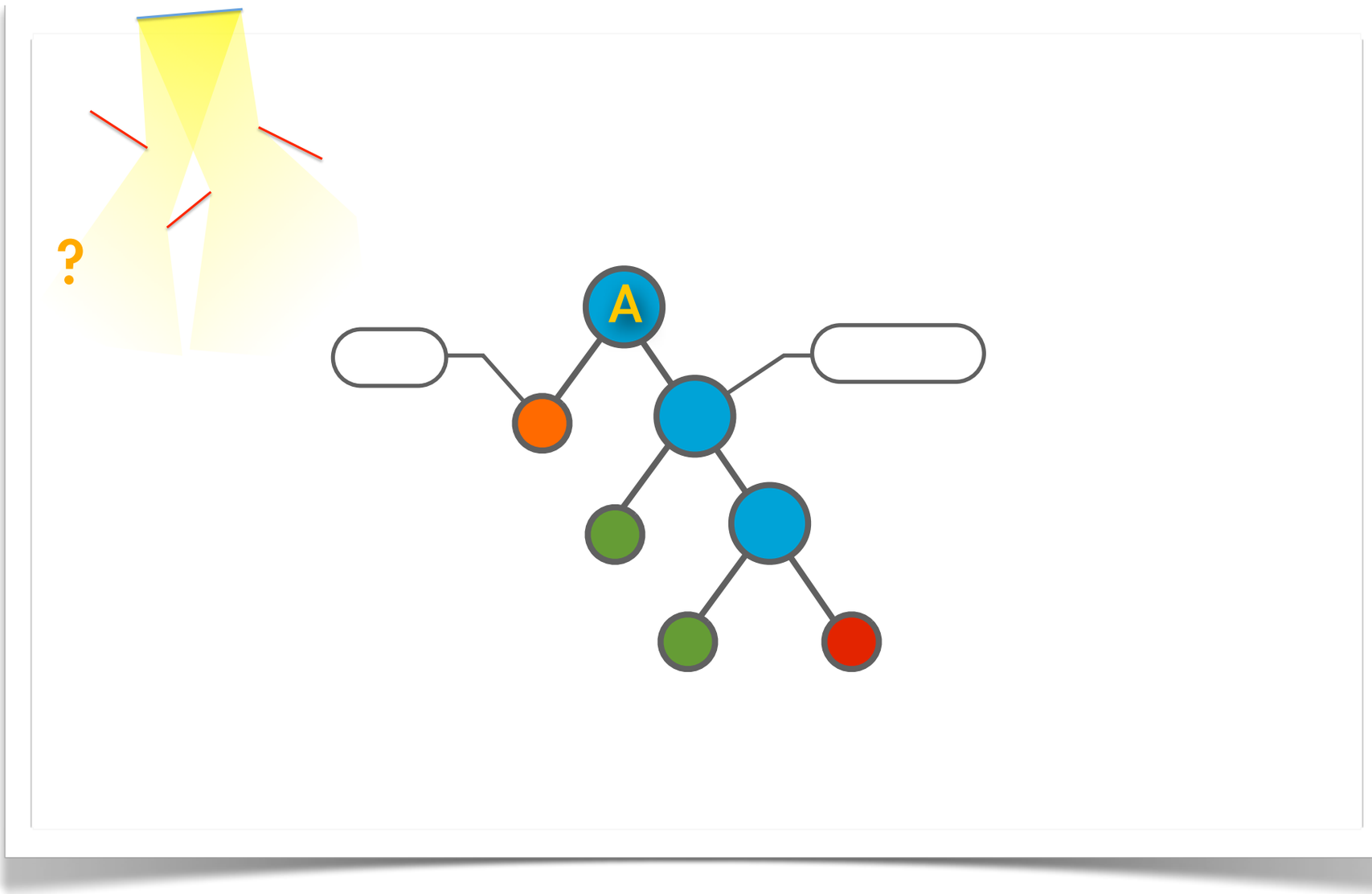
This requires to test the orientation of the lines stabbing the light source and an occluder with the hyperplane in the nodes.

Fortunately, it is sufficient to test the vertex-to-vertex lines orientation with respect to a hyperplane. The details are in the paper.

If all the vertex-to-vertex lines have a consistent orientation, so does any line stabbing the light and the occluder. In this case we can insert the occluder in the left or in the right subtree.

if the orientation is not consistent, the occluder may affect both the subtree. Thus we conservatively insert the occluder in the both subtrees. Previous works would have computed CSG operations in such a case.

# Visibility algorithm > principle

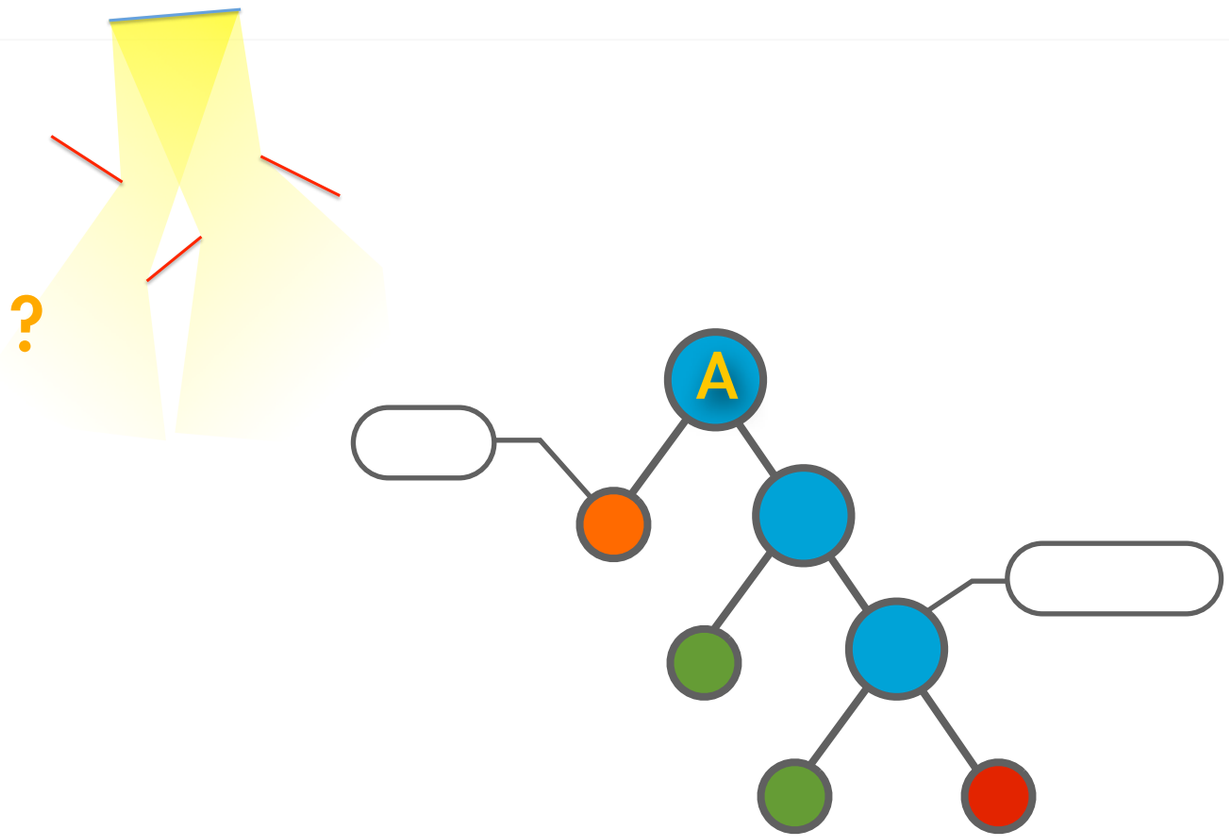



- insert the remaining occluders in the tree
- visible classes reached by occluders become undefined

So we insert the occluders into the tree.

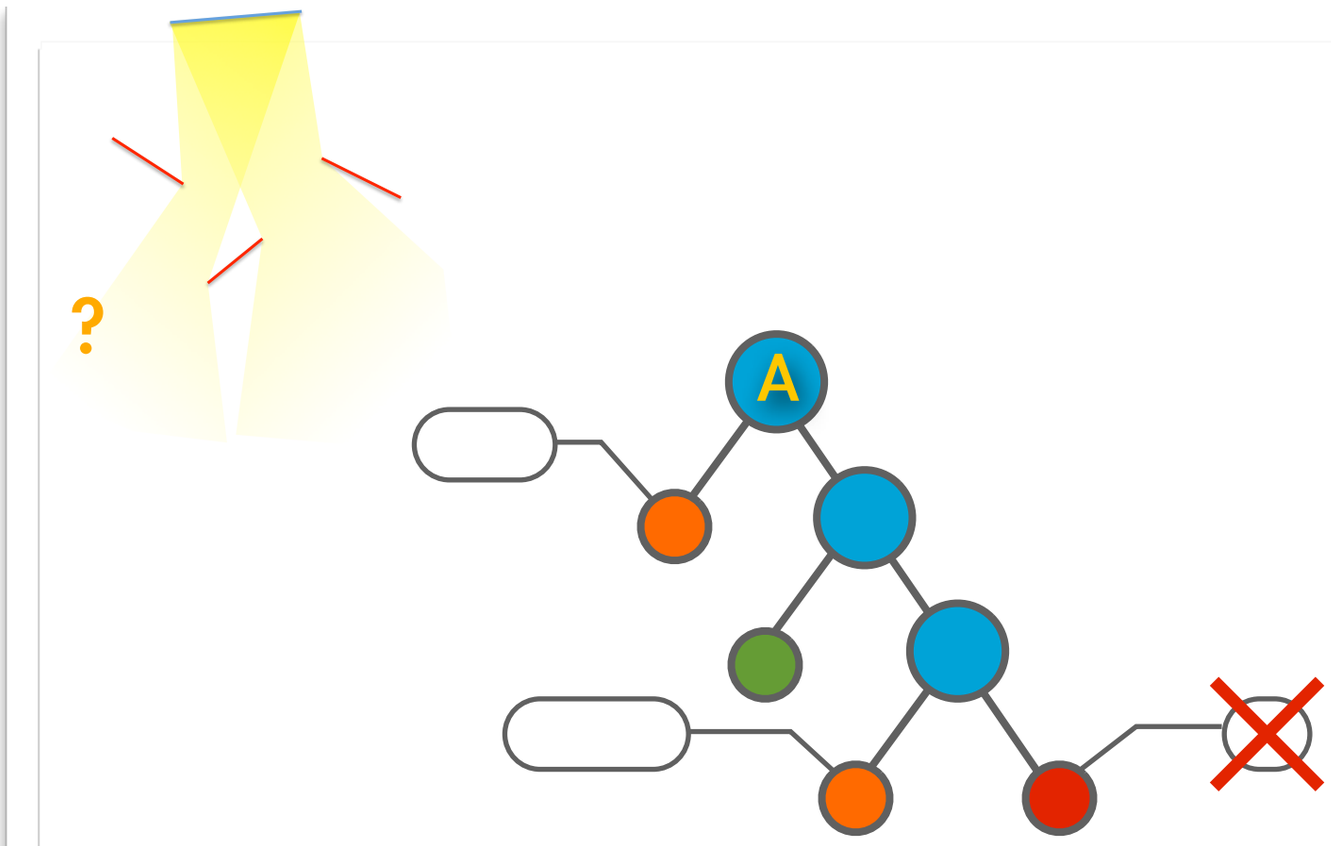
if they reach a visible class, they are associated to its leaf which becomes undefined

# Visibility algorithm > principle




The process continues until all the occluders reach a leaf

# Visibility algorithm > principle

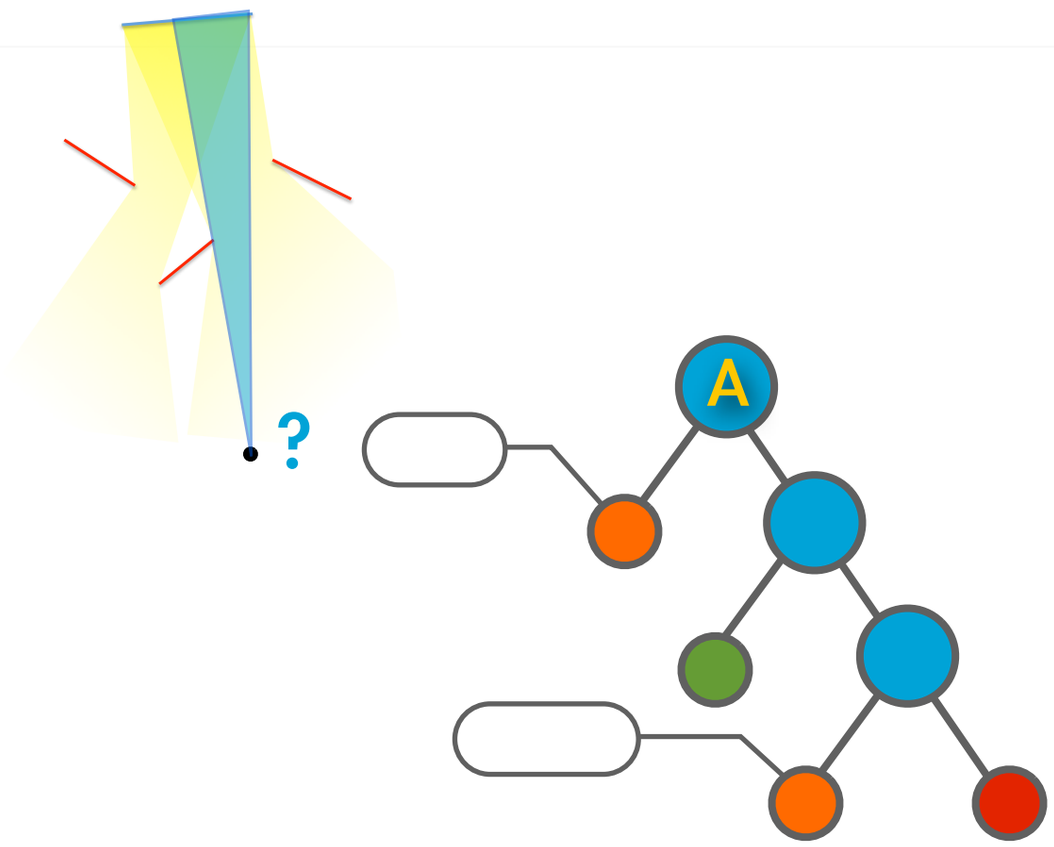



- occluders reaching invisible classes are discarded
- first problem is partially solved

If it is an invisible class, the occluders are discarded since this already represents blocked lines.

Now we have partially solved our first problem : to find the visible and invisble classes representing the visibility from the light

# Visibility algorithm > principle




- light visibility from the point ?

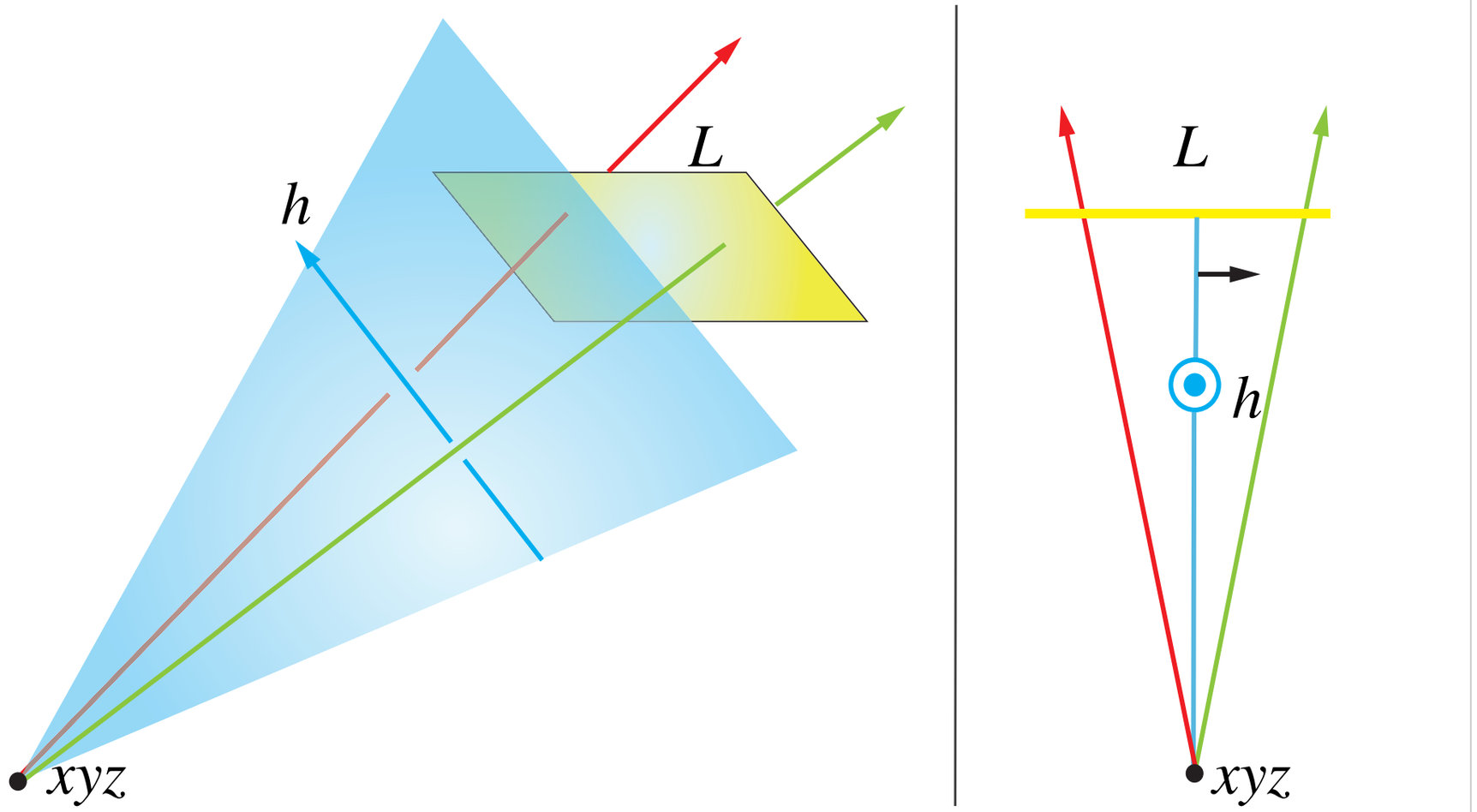
We now focus on our second problem : to compute the light source visibility from a given point.

Is this view is already included in the tree ?

To answer, we have to filter the lines in the view beam into the tree to find the classes they belong to.

Thus we need to test the lines orientation in the view beam against each hyperplane in the nodes.

# Visibility algorithm > view beam orientation



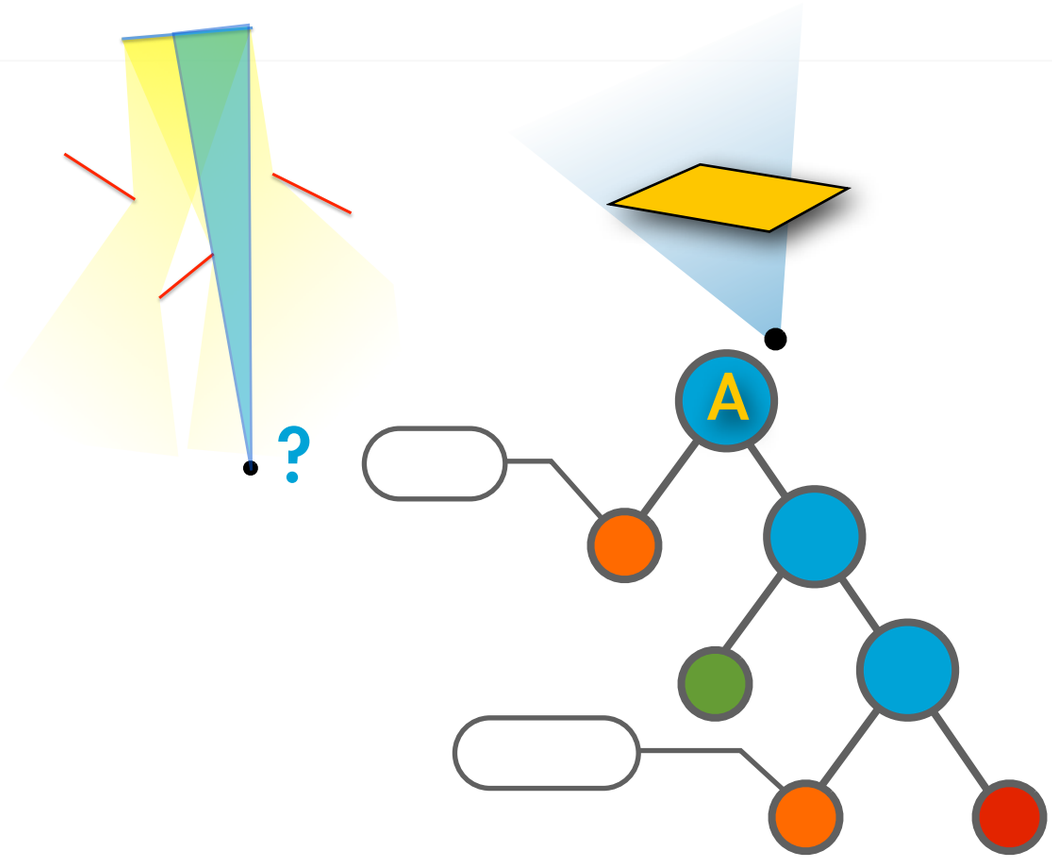

- view beam orientation with respect to a given line h
- not consistent ? Then split the light L !

Such a test is equivalent to compute the light position with respect to the plane defined by the viewpoint and the line corresponding to the hyperplane in the node.

If the light is in the positive or negative halfspace of the plane, any line in the view beam has a negative or positive orientation with the hyperplane.

And if the plane intersect the light source, lines in the view beam have different orientations. However, we can split the light by the plane. This defines two view beams, one with all the lines having a positive orientation, one with all the lines having a negative orientation.

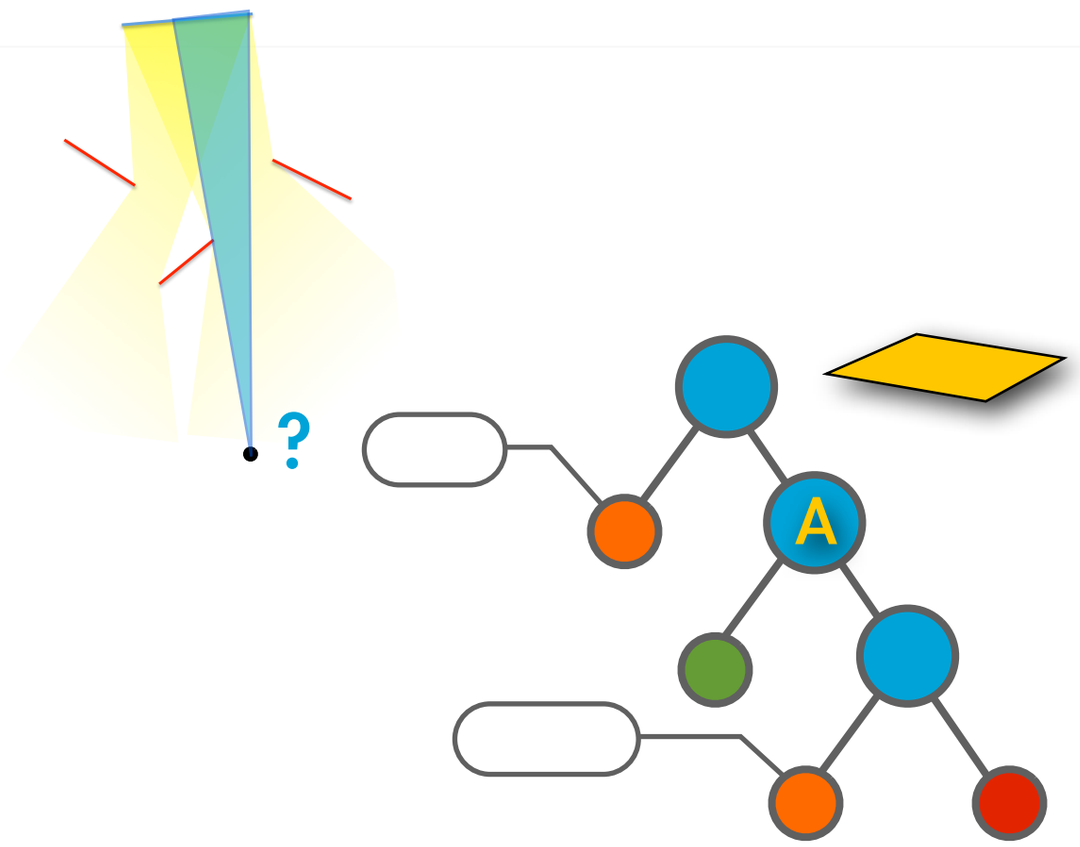
# Visibility algorithm > principle



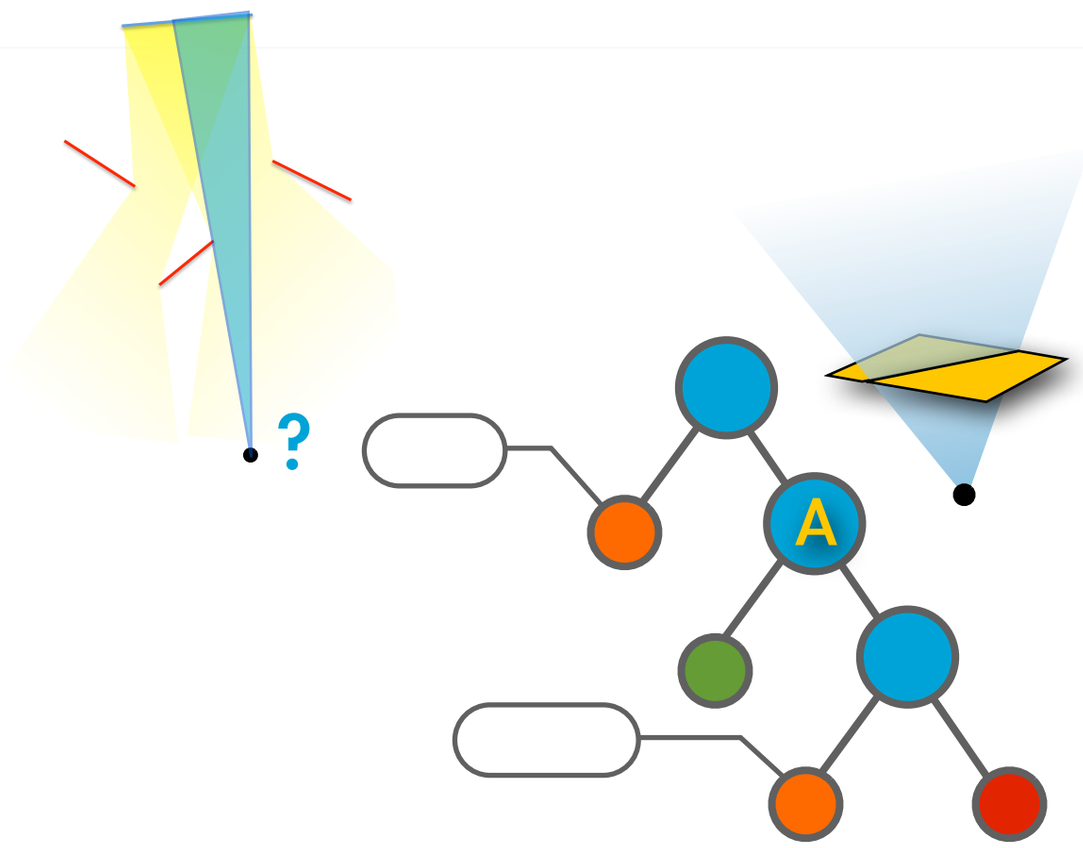

- negative orientation ⇨ right child

For example, we start from the root node and find that the orientation is negative. So we have to search in the right subtree, which contains all the lines having a negative orientation with respect to the root node.

# Visibility algorithm > principle



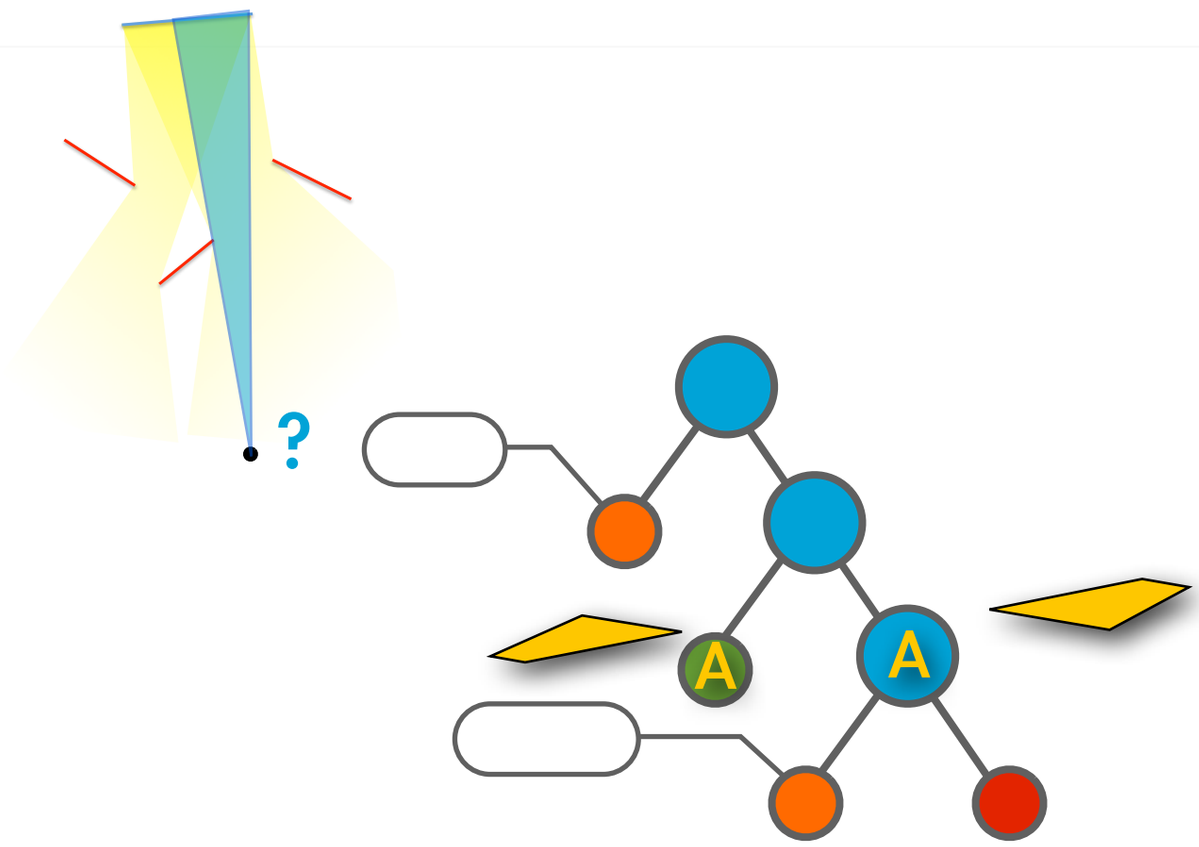
# Visibility algorithm > principle




- heterogeneous orientation ⇨ split the light

If the light is split by the plane, the orientation is not consistent, so we compute the light fragments

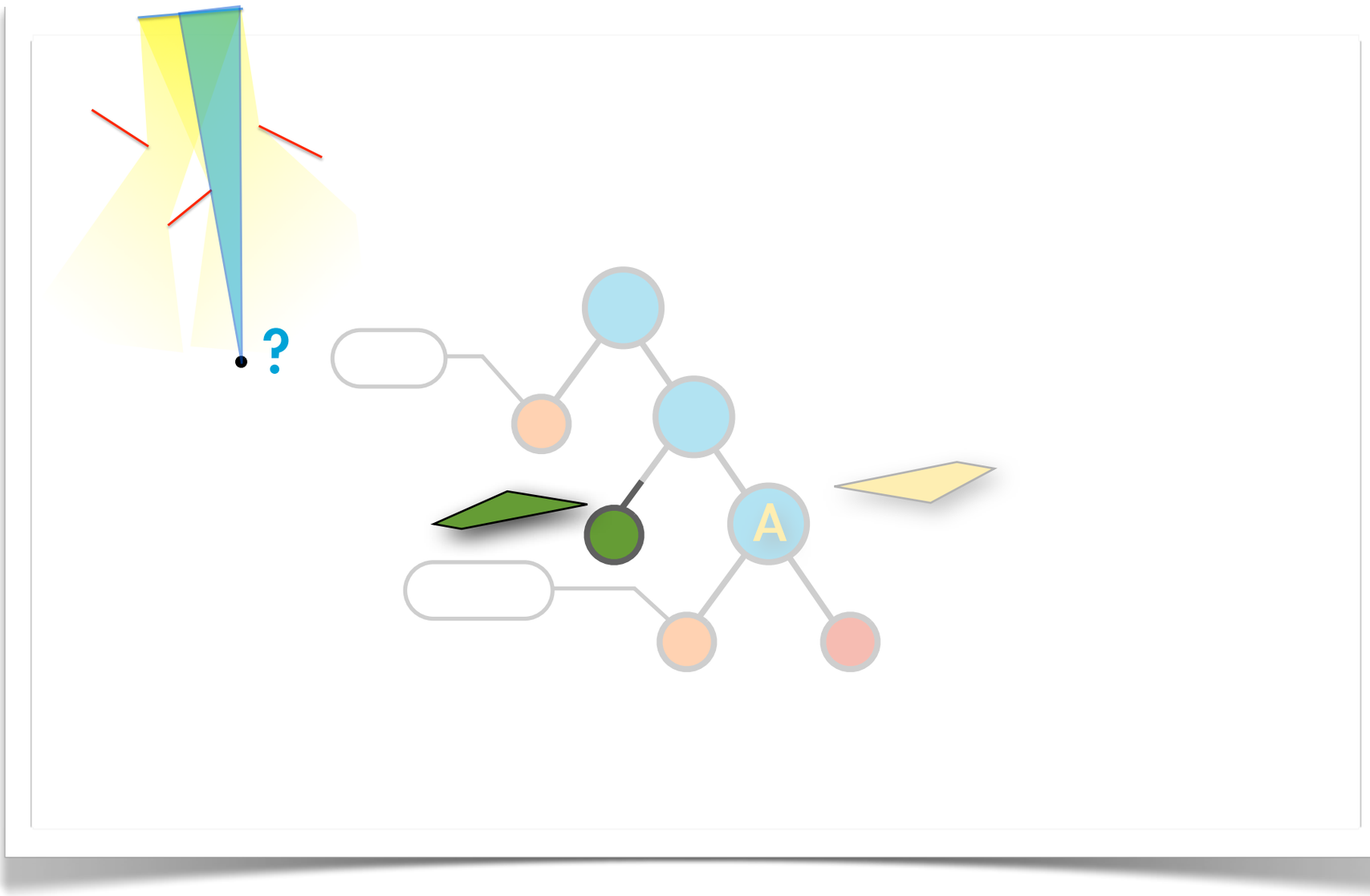
# Visibility algorithm > principle




- insert light fragments in the relevant child

and insert them in the relevant child

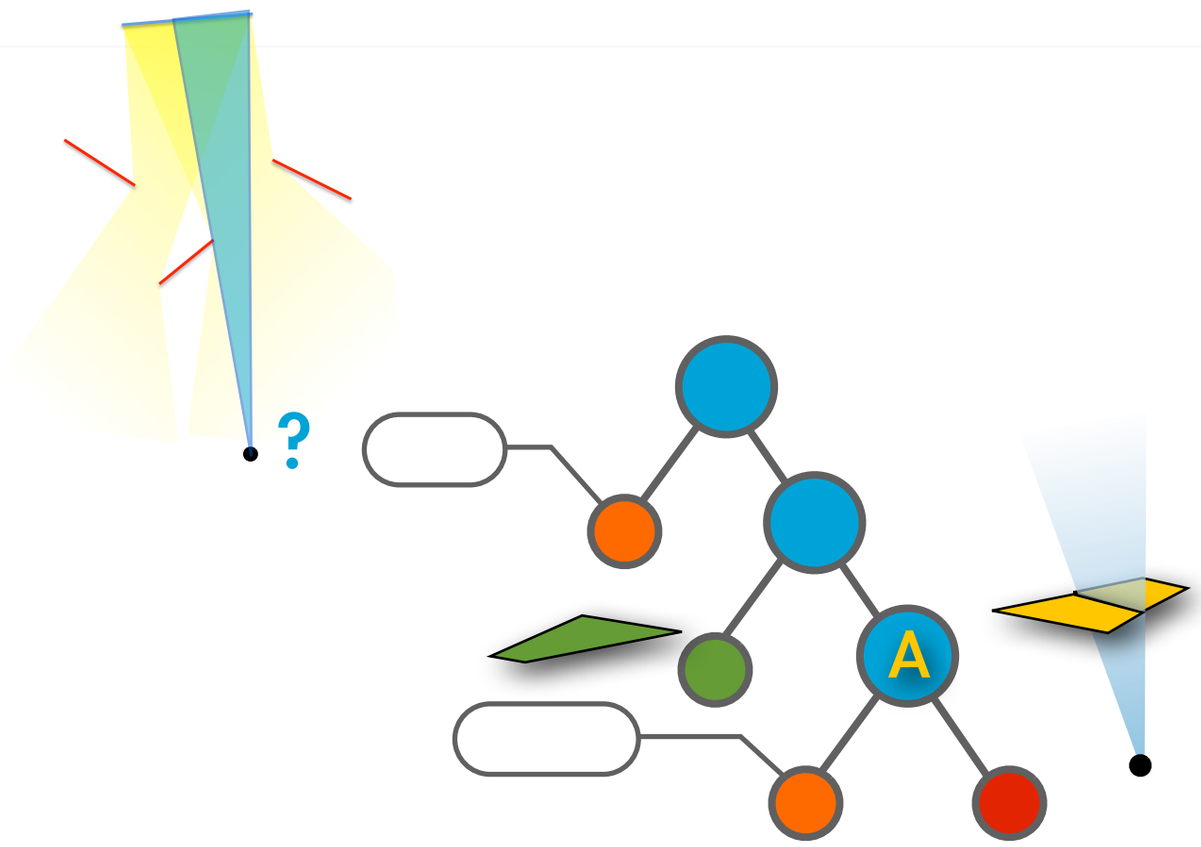
# Visibility algorithm > principle




- light fragments reaching a visible class are visible

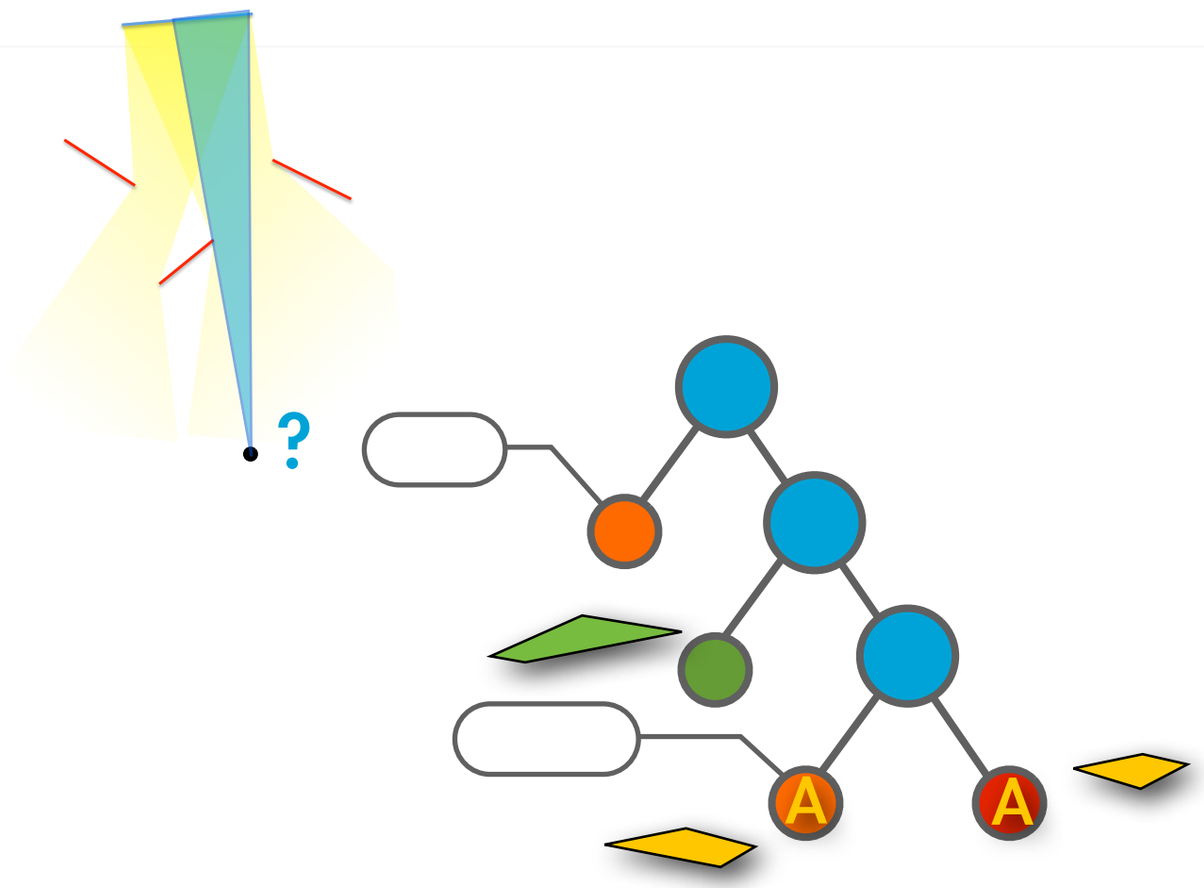
if a fragment reaches a visible class, the fragment is visible from the point. Because it belongs to a set of lines that does not intersect any occluder.

# Visibility algorithm > principle

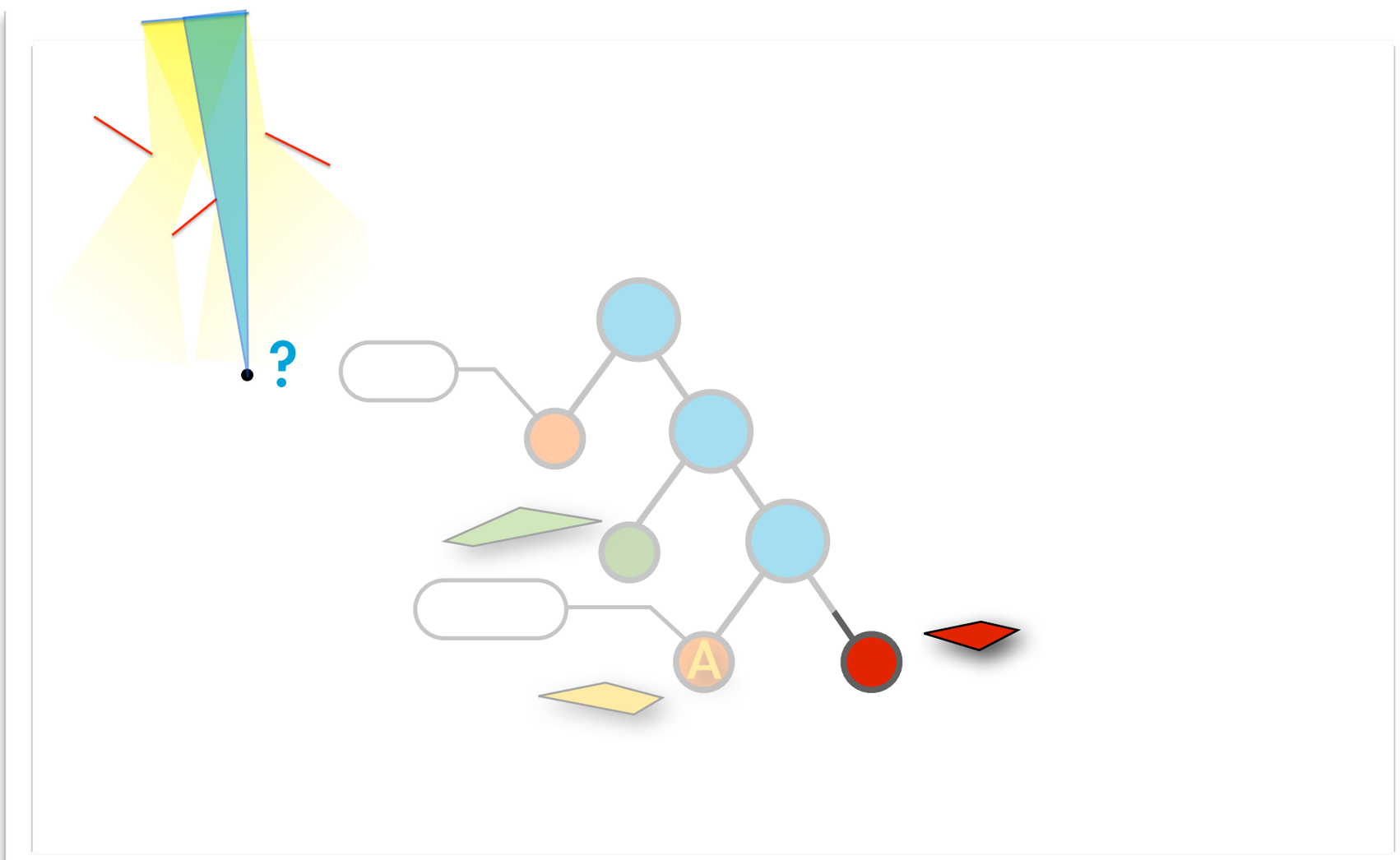



The process continues until every fragments reach a leaf

# Visibility algorithm > principle



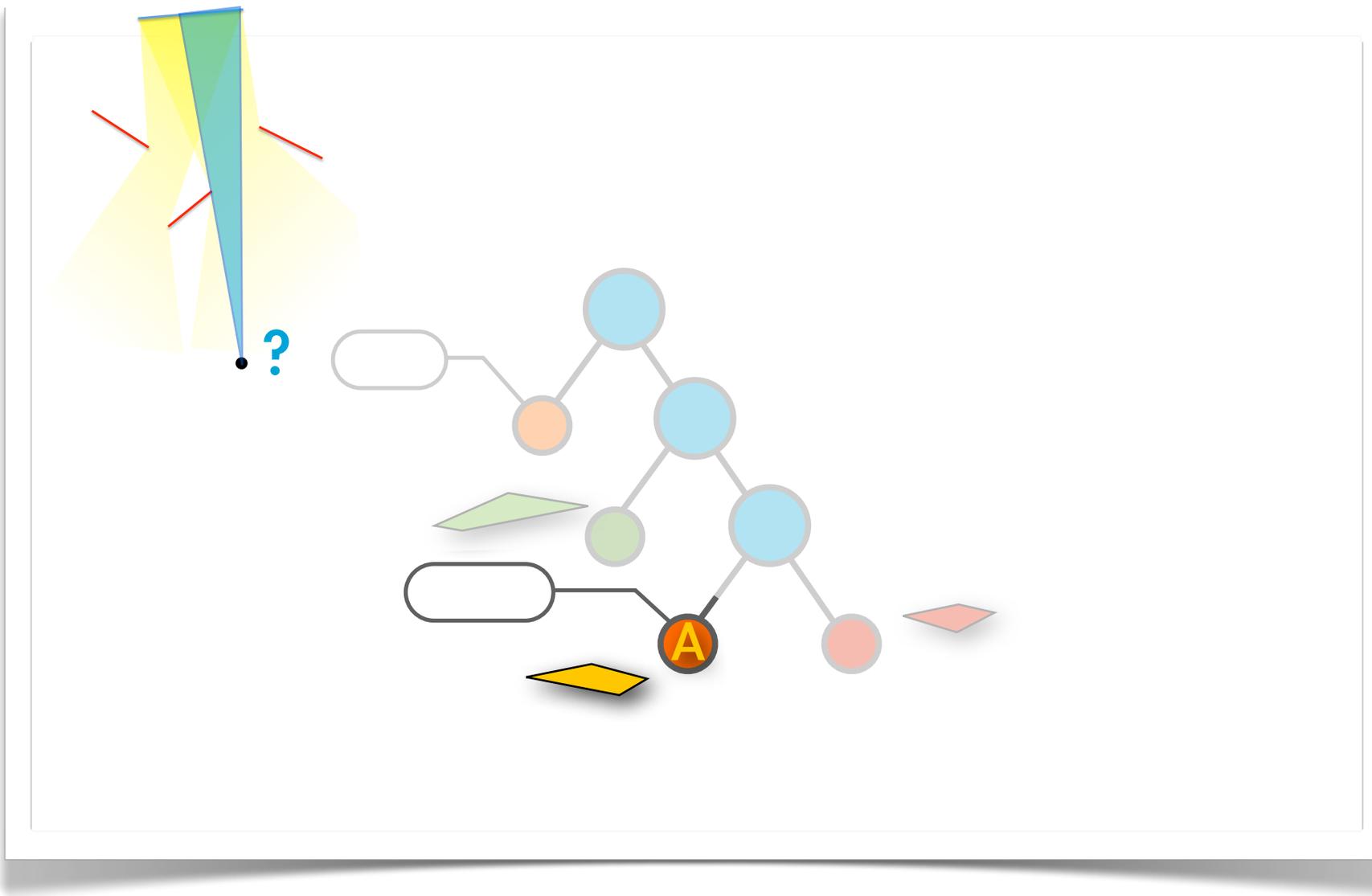
# Visibility algorithm > principle




- Light fragments reaching invisible class are invisible

If a fragment reaches an invisible class, it is not visible from the point since it belongs to a set of lines that intersect at least one occluder.

# Visibility algorithm > principle




- a light fragment has reached an undefined class
- recurse !
- BSP tree grows on demand

If a fragment reaches an undefined class, we don’t know if it is visible or not, because occluders remain and may affect the visibility.

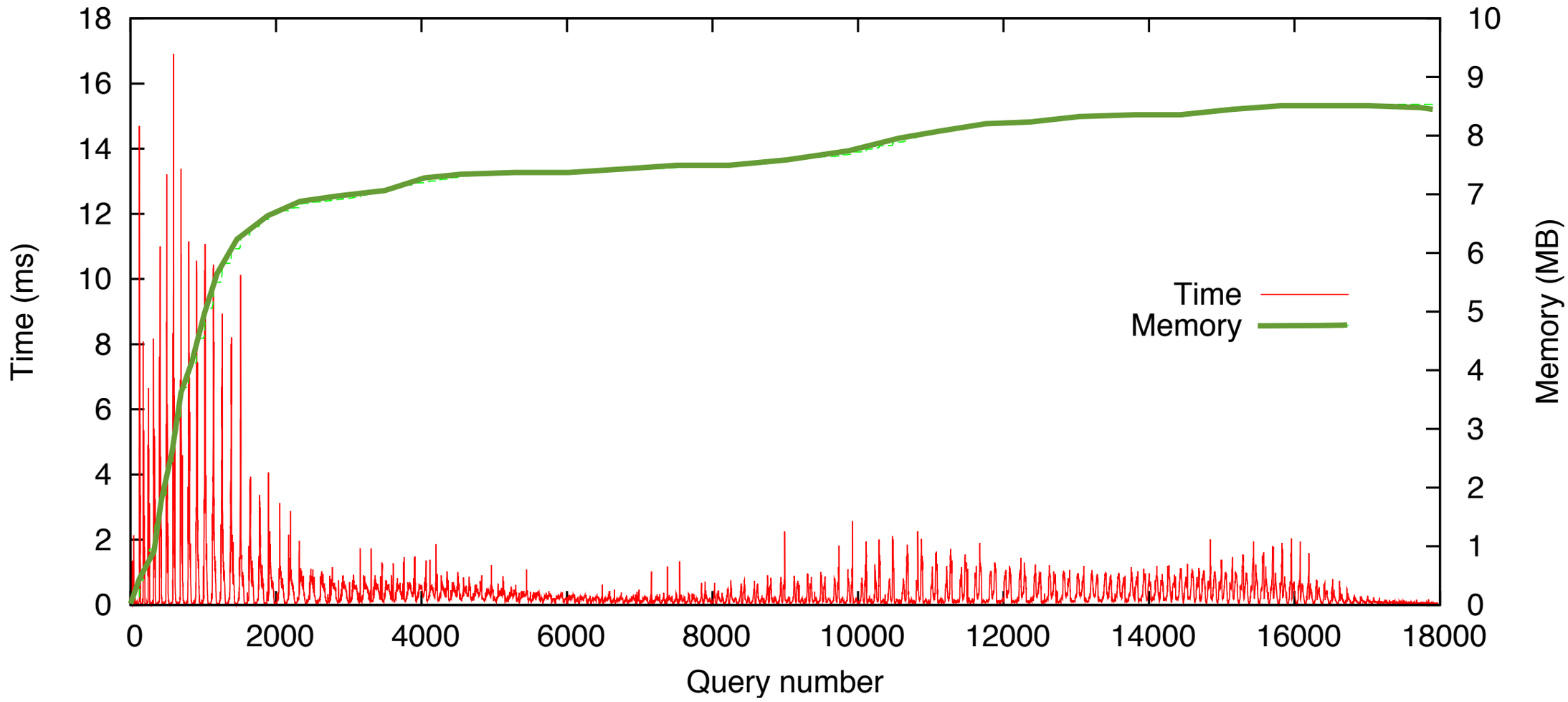
This is the same situation as starting. We just have to recurse until all fragments are found visible or invisible.

Finally the BSP tree grows only when and where it is needed.

# Visibility algorithm > key points

## Visibility coherence

- first queries grow the tree
- next queries take advantage of the first ones



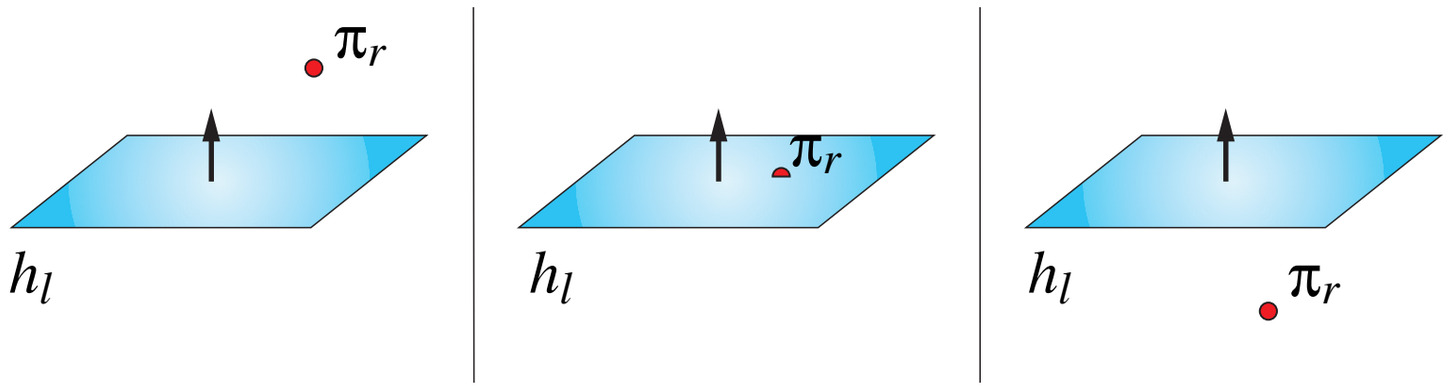

We gave an example with one point-light query, but this algorithm is designed to answer many point-light queries.
This graph show the memory growth of the tree in green, as well as the cost per point-light query in red. About eighteen thousand queries were performed

The first queries have an extra cost because they grow the tree.
However the following queries are cheaper, taking advantage of the previous ones. This is because neigbouring points with similar view are likely to depend on the same equivalence classes.

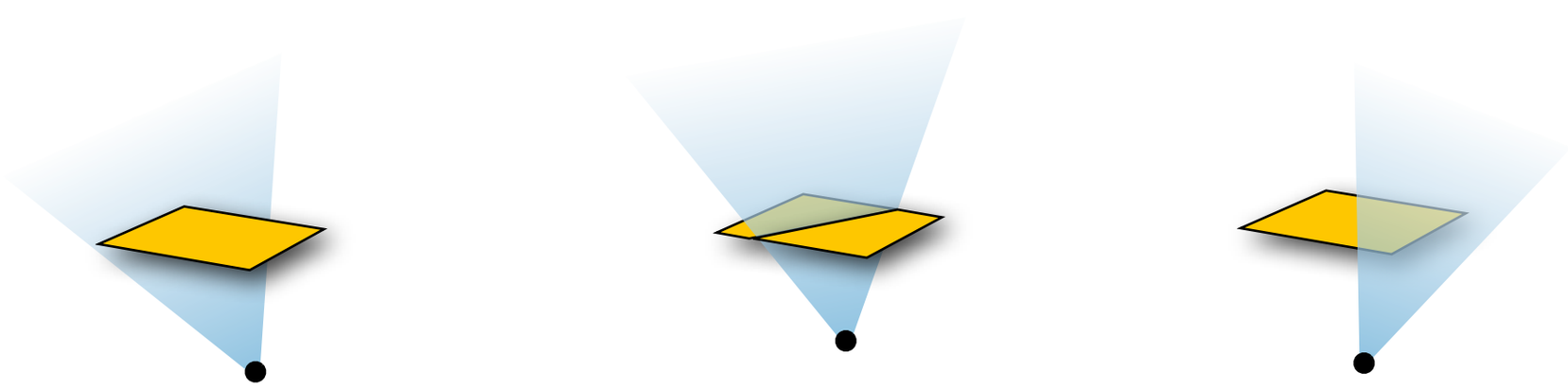
# Visibility algorithm > key points

## Robustness/efficiency

- BSP tree growth : point/hyperplane sign tests




- view beam filtering : at worst, plane-polygon intersection



The process is robust because it relies on simple operations.
To grow the tree we only need to perform point/hyperplane tests in the plucker space
An to filter a view beam, we need, at worst, to split a convex polygon by a plane.

# Outline >

- Geometric basis
- Visibility algorithm
- Results

I will now present some results

# Visibility algorithm > application to soft shadows

- group all image points per visible triangles
  - for each group
    - get the occluders and initialize a BSP tree
    - for each image point
      - run a point-light query
      - compute analytic direct illumination

To test our approach, we plug our algorithm in a ray tracer to compute soft shadows.

From the primary rays, we group all the image points per visible triangles.

Next, for each group we select the occluders using a shaft culling approach, and initialize an empty BSP tree.

At last, for each image point in a group, we query the tree to find the visible parts of the light and compute analytic direct illumination (assuming a uniform light source)

# Results > comparison method

Ray-traced shadows

- optmized (CPU) ray-tracer (SAH kd-tree, SIMD, multithreading)
- 4 shadow rays at a time
- uncorrelated stratified sampling

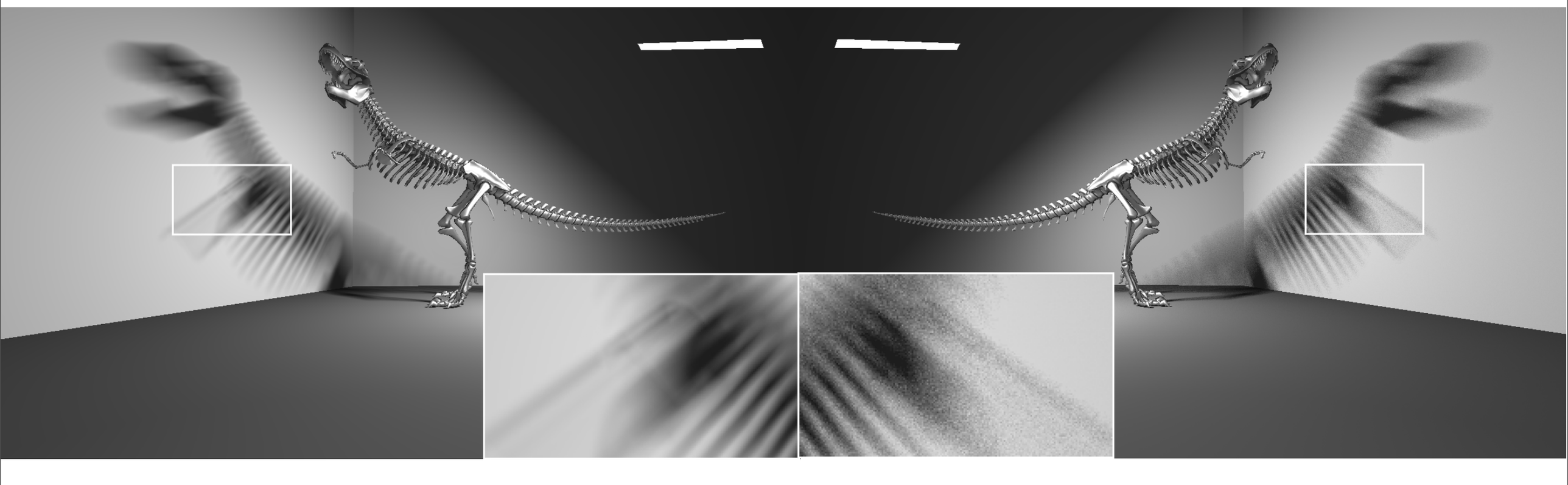
Configuration

- 2.67 GHz Intel Core i7 920, 4GB of memory
- all picture at 1280 x 720, 1 primary ray per pixel
- all tests use 4 threads

We compare our approach with ray traced soft shadows using an optimized ray tracer.

All picture are rendered at the same resolution with one primary ray per pixel for comparison purpose.

# Results > comparison on equivalent time



T-Rex (26K triangles)

Time : 6.5s
Memory : 19 MB

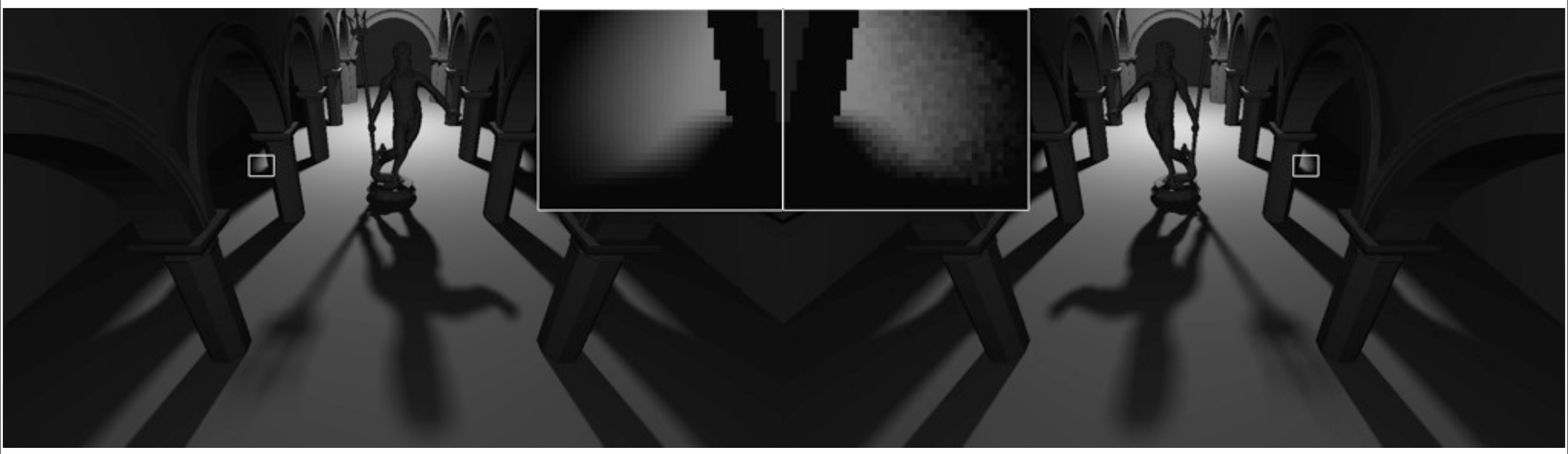
Time : 7s
32 samples

At first we present results at equivalent time. This is only the time spent in the soft shadows computations.

The first scene, T-Rex is moderate but has complex shadows with a long area light source. It is a difficult configuration for our method because complex shadows means more equivalence classes to compute.
During the process, the memory consumption can vary because the trees are built one after another and each one grows lazily. So, here we report the maximum memory load reach during the computations.

We produce accurate and noise free shadows within seven seconds. In comparison, the ray traced shadows remain very noisy because of the low sample number.

# Results > comparison on equivalent time



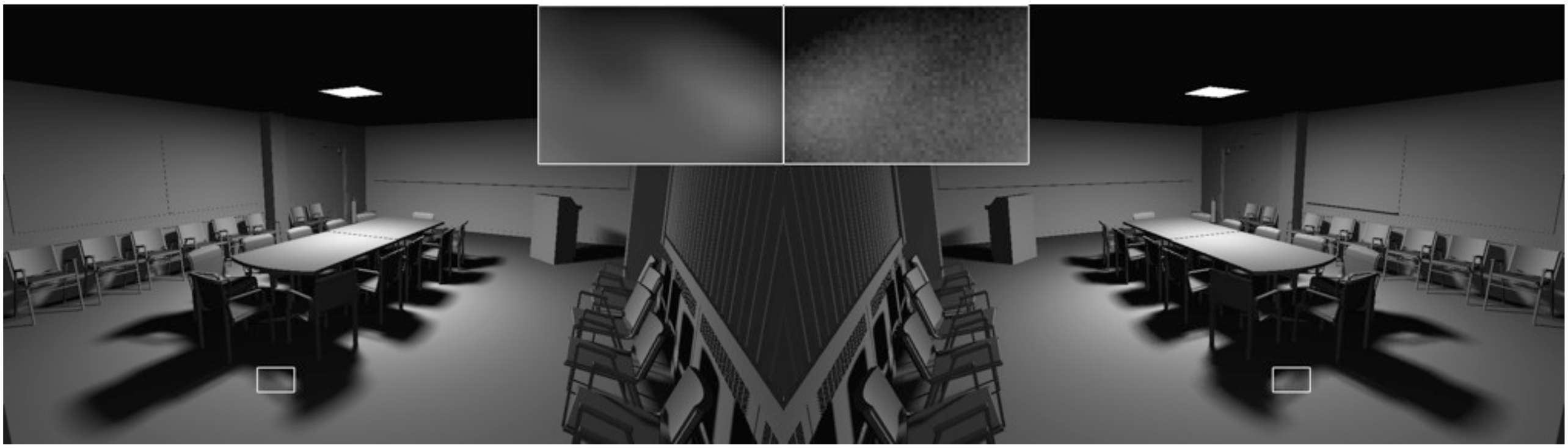
Sponza with Neptune (115K triangles)

Time : 7s
Memory : 16 MB

Time : 7s
32 samples

This is a more important model with more than one hundred K triangles, Sponza with the Neptune statue.

# Results > comparison on equivalent time



Conference (282K triangles)

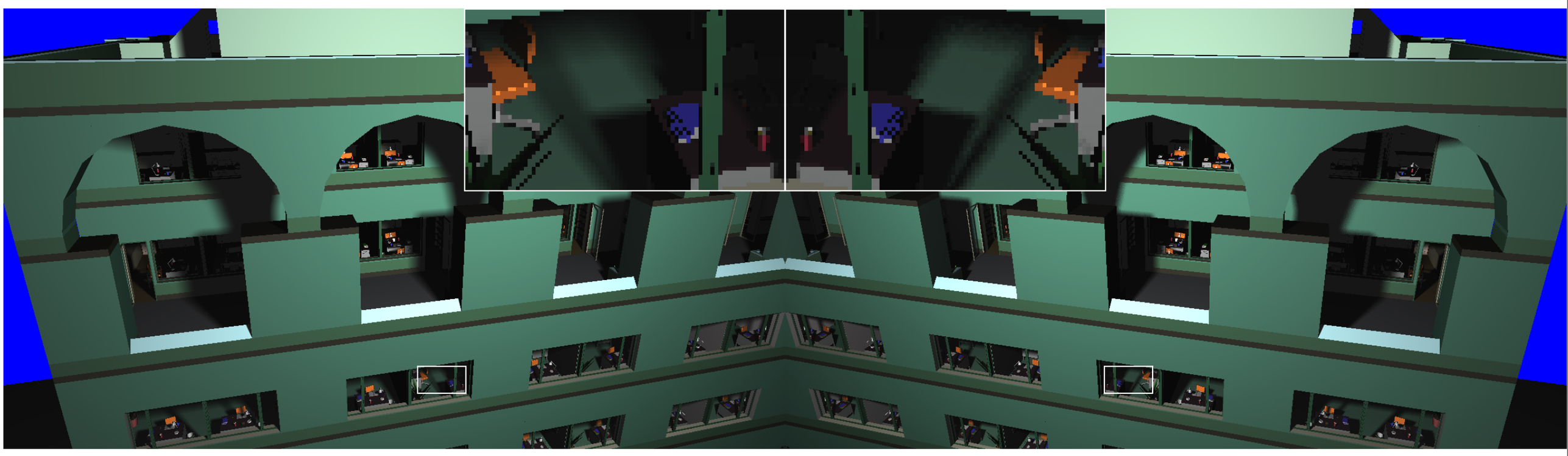
Time : 6s
Memory : 20 MB

Time : 6s
32 samples

The well knonw conference model, with almost three hundred triangles.

These results show that our method has a consistent behaviour in different configurations. The shadow complexity has the main impact because it is directly related to the number of equivalence classes we have to compute.

# Results > comparison on equivalent time



Soda Hall (2147K triangles)

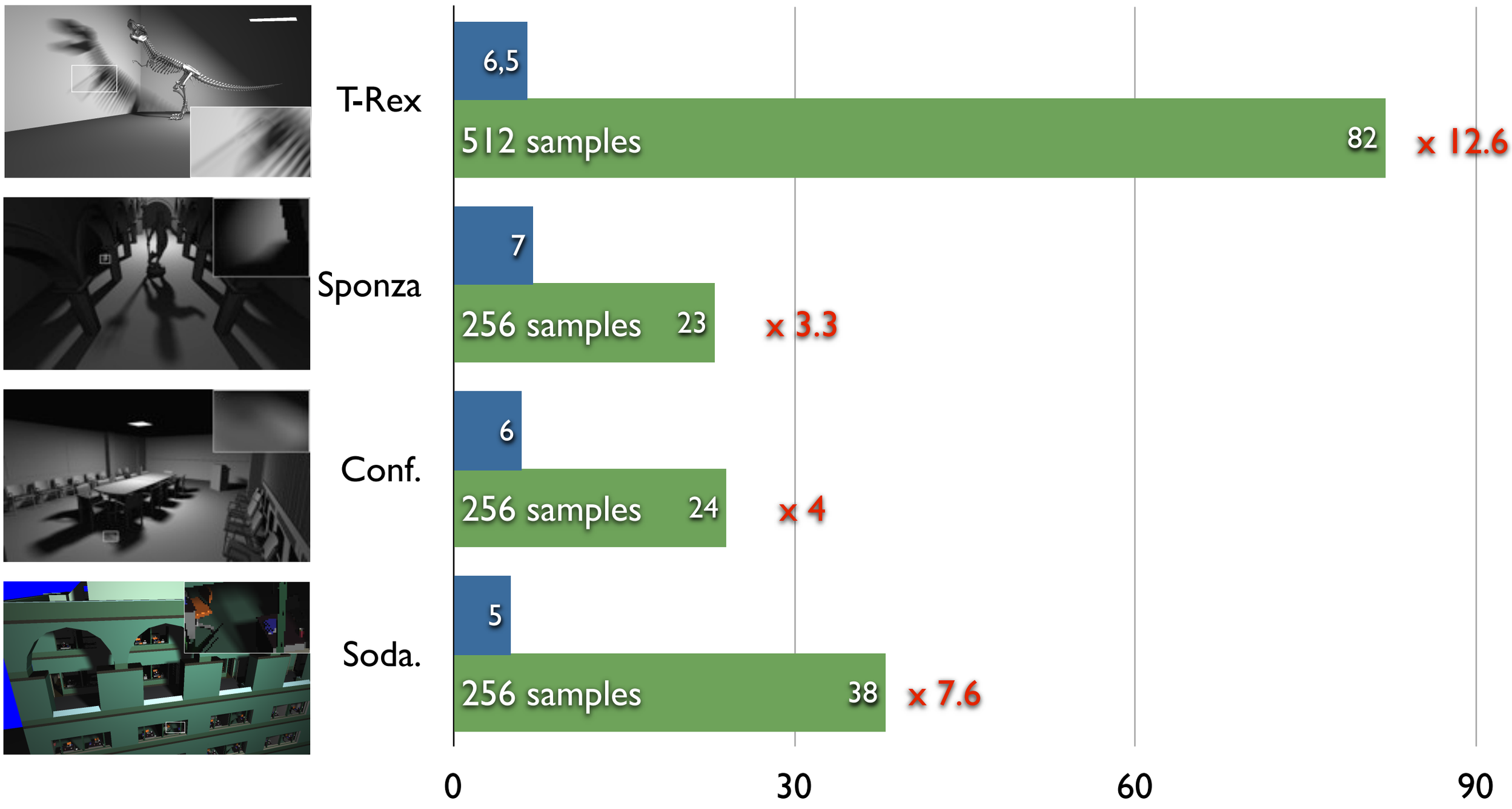
Time : 5s
Memory : 20 MB

Time : 8s
32 samples

And at last, the soda hall model.

This is not a very interesting scene for shadows, but this is to illustrate that the whole process is rather independant from the number of triangles in the scene, because we only deal with visible triangles from the camera.

# Results > comparison on equivalent quality




Using the same scenes, this is the timings at equivalent quality.

Two hundred and fifty six samples are generally sufficient to avoid the noise in ray traced shadows. Except for the T-Rex where the double is required.

At equivalent quality, the ray traced shadows are clearly more expensive.

Detailed timings and more tests can be found in the paper.

# Conclusion >

What we have shown:

- coherent from-polygon visibility representation
- exact and analytic soft shadows

What we hope:

- analytic (from polygon) visibility can be robust and efficient
- may be useful for many other problems

What we plane

- occlusion ⇨ visibility ? (done)

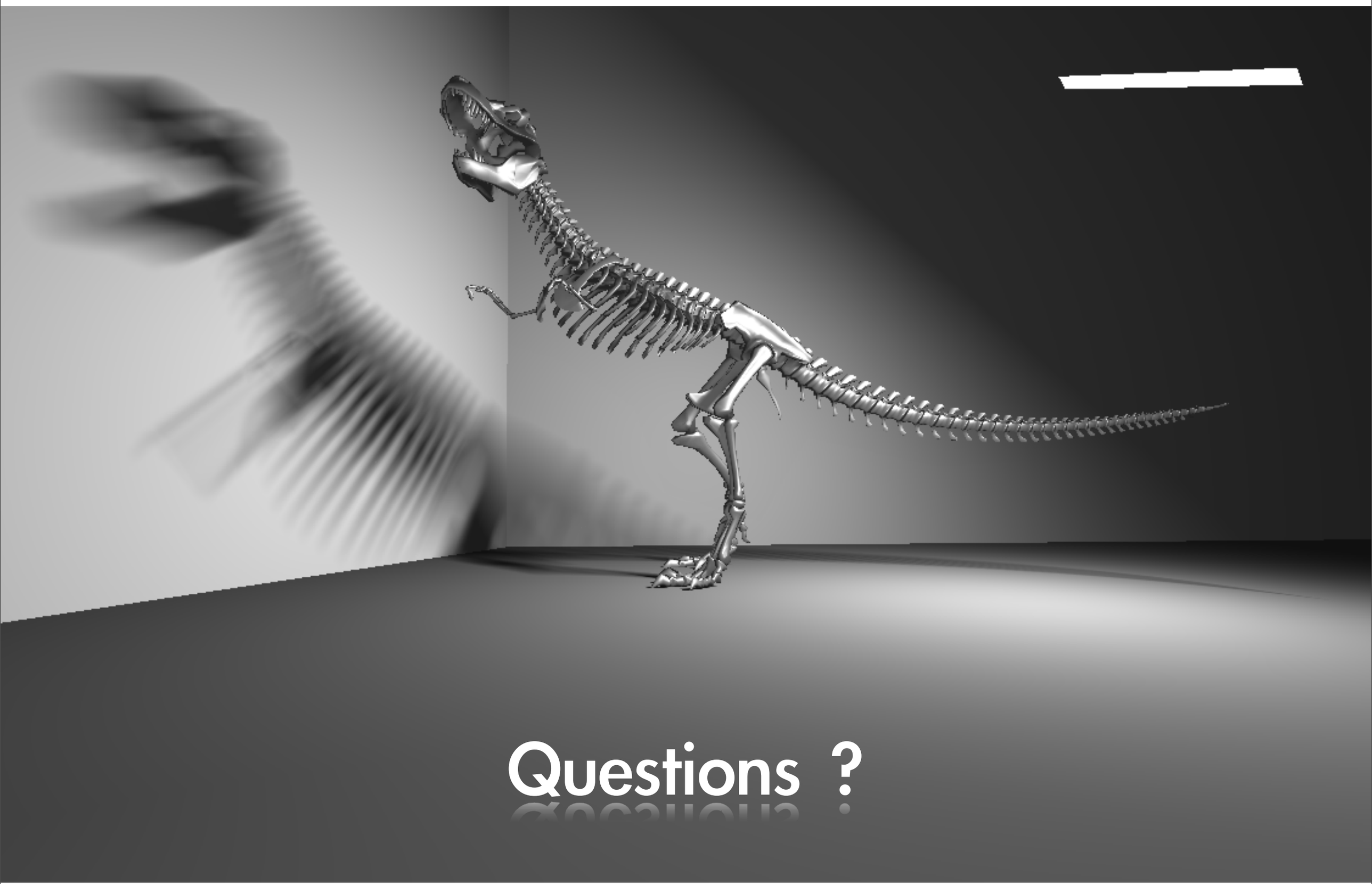


To conclude, we have presented a new from polygon visibility algorithm which was used to efficiently compute analytic soft shadows.

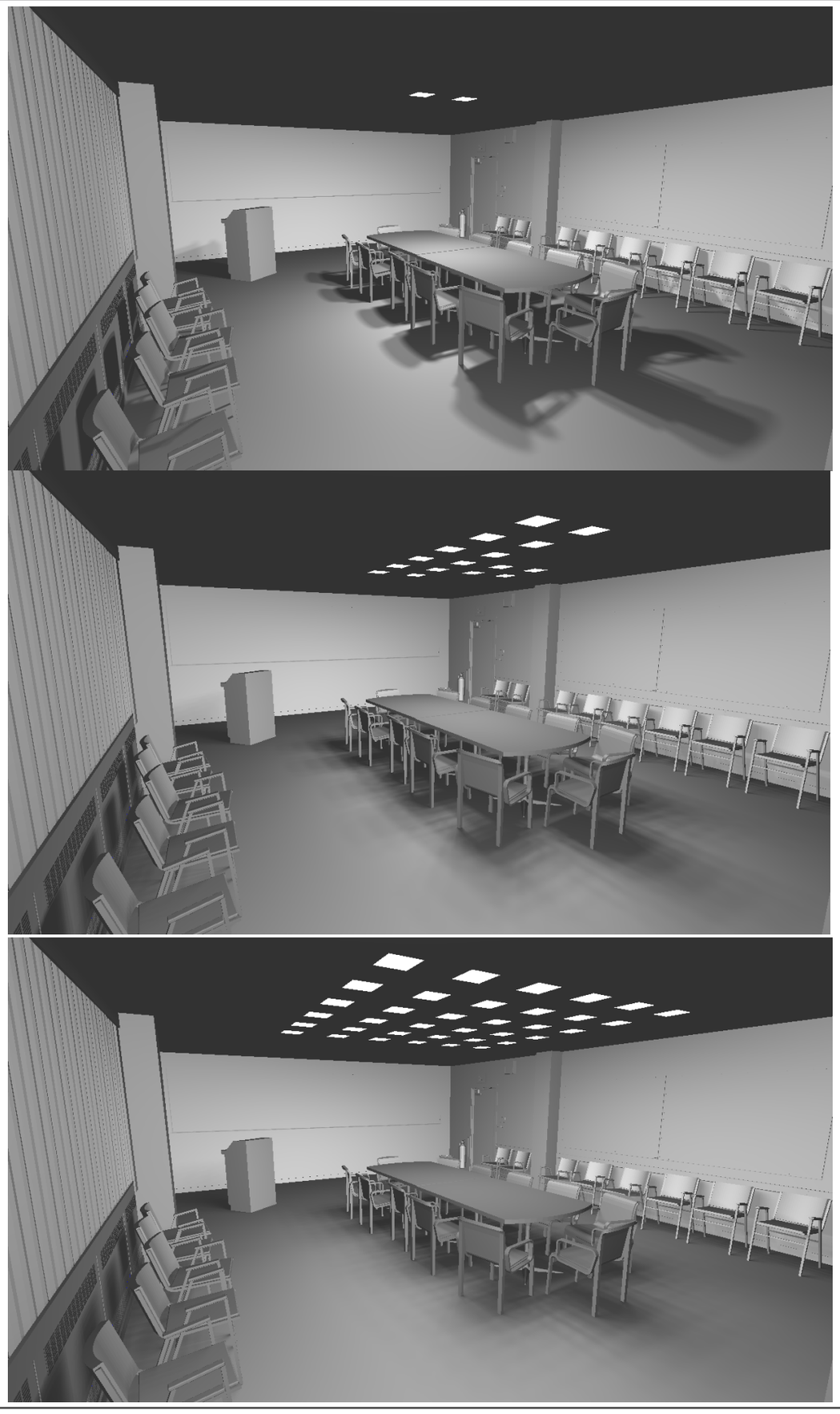
We hope this work shows that from-polygon visibility can be made robust and efficient. This can be a solution to take advantage of the visibility coherence. Thus we think it may be usefull for many other problems.

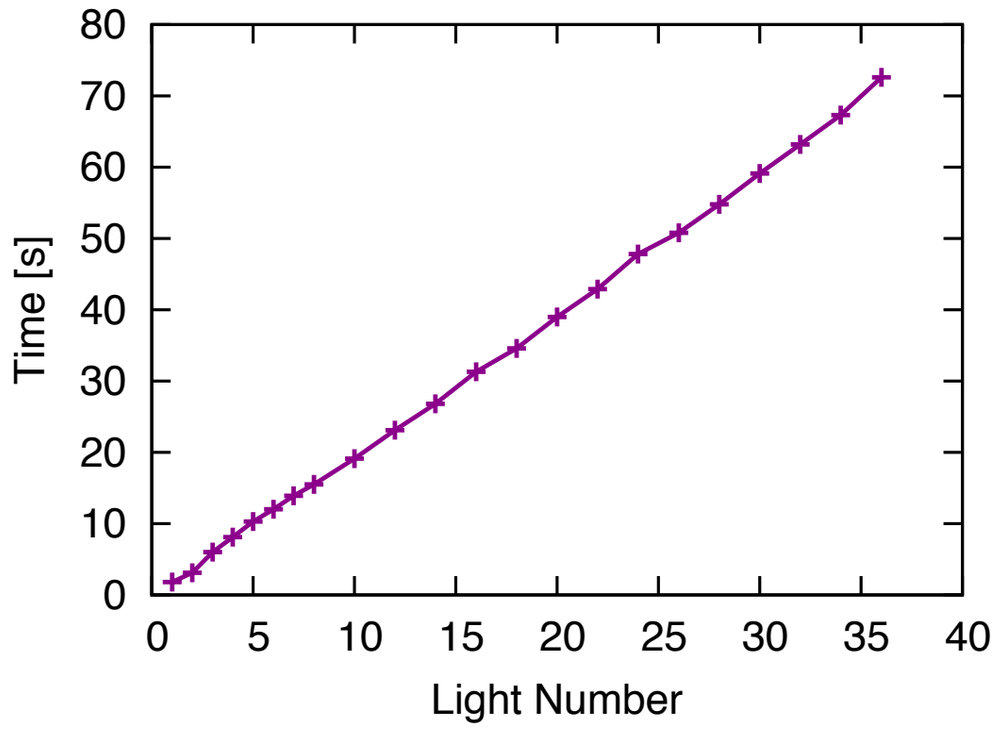
Actually our algorithm encodes occlusion rather than visibility. Therefore, the next step was to add a depth information in order to really compute the visibility from a polygon. Since last year we have done and published this work in a companion paper with an application to ambiant occlusion.

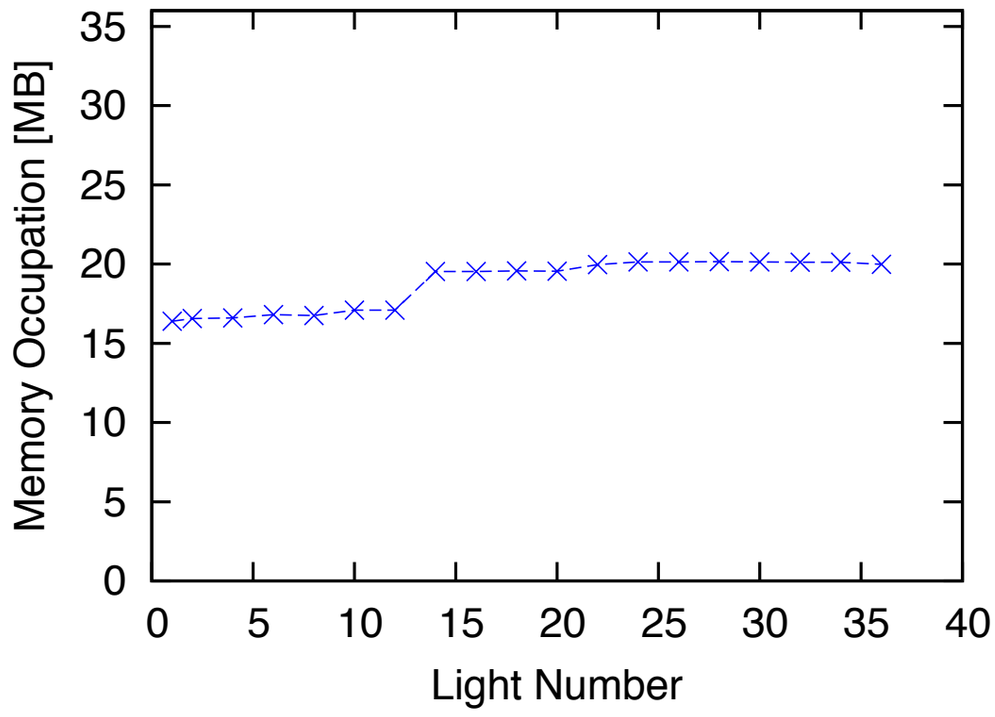
We are now working on a GPU friendly version of this algorithm. Not only to be faster but mainly because we think this would create new perspectives.


Questions ?

# Results > increasing the number of lights





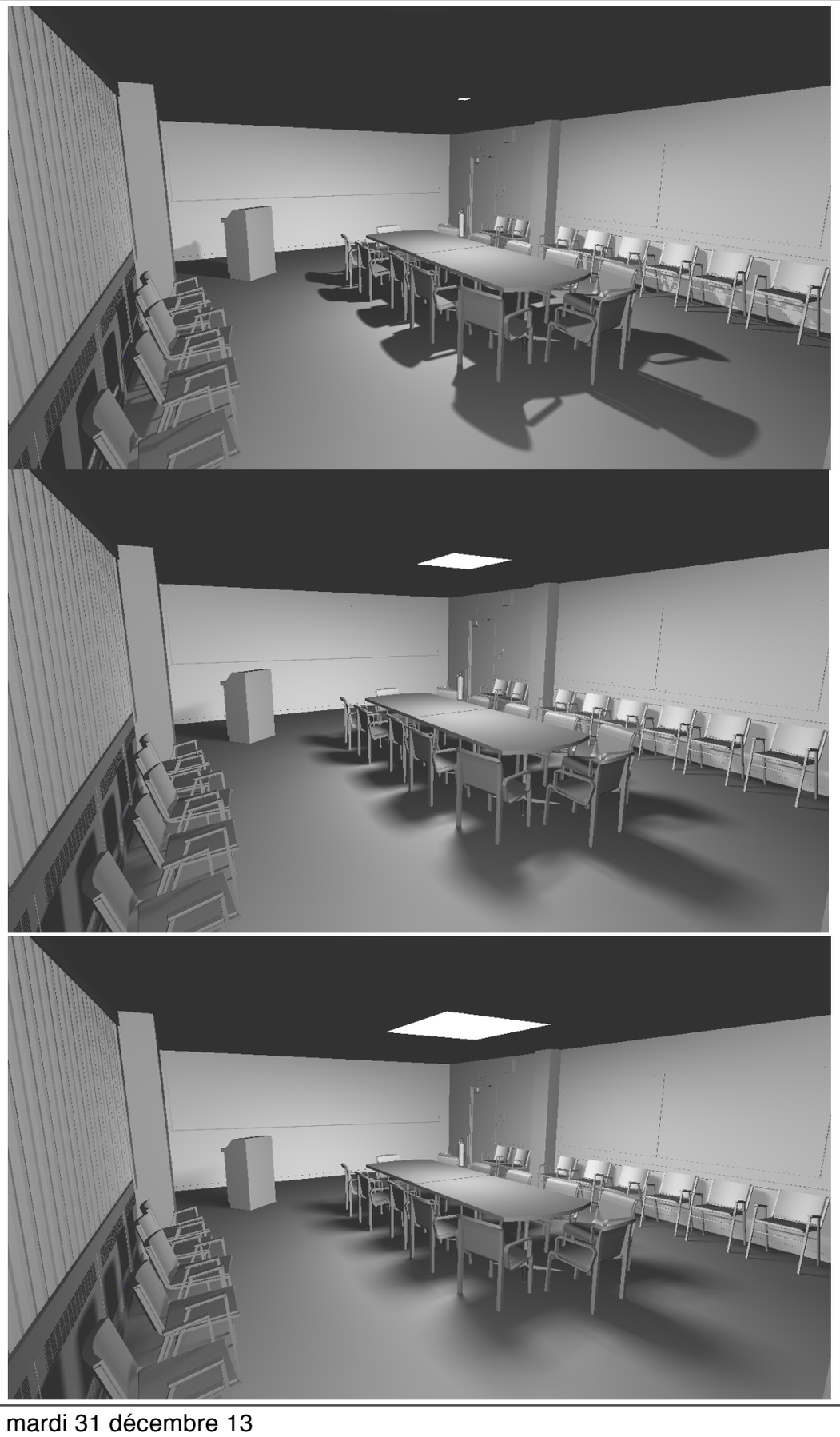


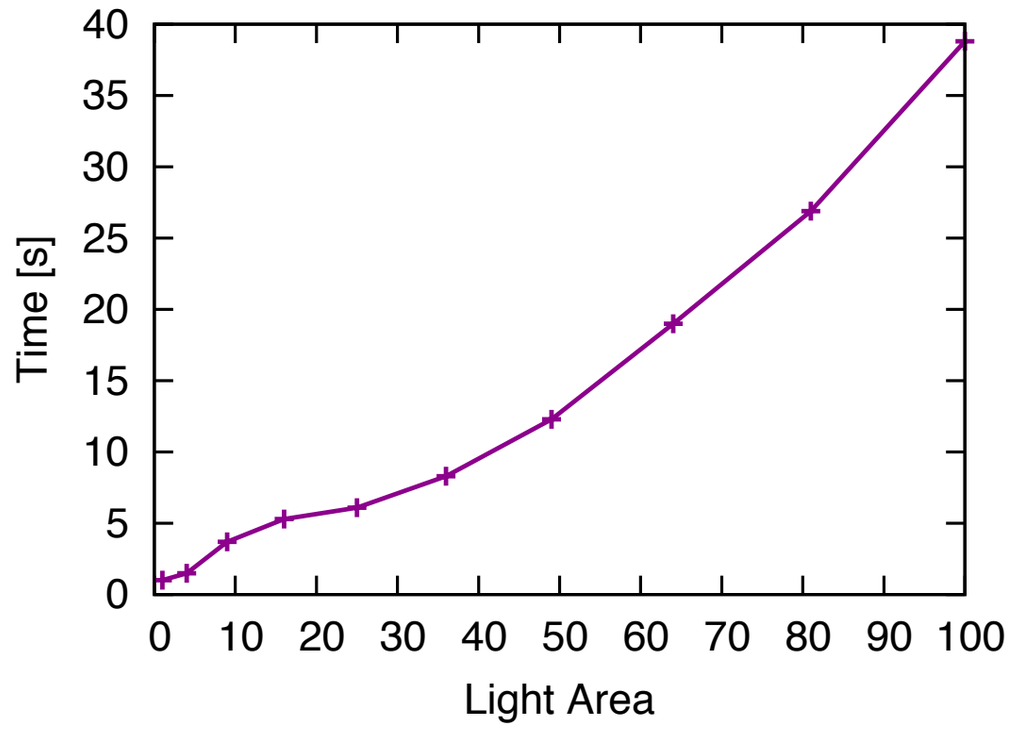

In this test we increase the light number to see how our method behave.

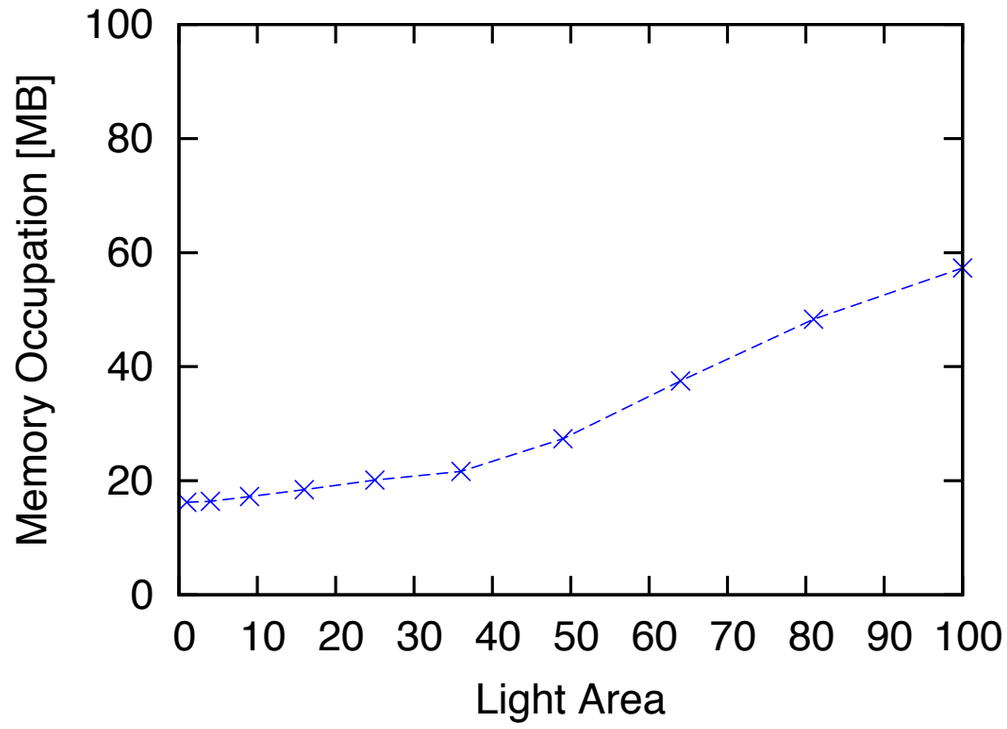
Since lights are handled one after another, the computation time increase linearly, an the memory consumption remain stable.

We can notice a small increase from the twelth lights. This is because the twelve first light are above the conference table while the following are abovethe chairs, casting more complex shadows.

# Results > increasing the area light source






Here we test how our method behave when the area light source increases in size.
Obviously, time and memory consumption increase because more and more equivalence classes are computed.
However, the method remains practicable even for the biggest light source.