COMP3421

Global Lighting Part1: Ray tracing

Global Lighting

The lighting equation we looked at earlier only handled direct lighting from sources:

$$I = \boxed{I_a \rho_a} + \sum_{l \in \text{lights}} I_l \left(\rho_d (\hat{\mathbf{s}}_\mathbf{l} \cdot \hat{\mathbf{m}}) + \rho_{sp} \left(\hat{\mathbf{r}}_\mathbf{l} \cdot \hat{\mathbf{v}} \right)^f \right)$$

We added an ambient fudge term to account for all other light in the scene.

Without this term, surfaces not facing a light source are black.

Story so far…

Global lighting

In reality, the light falling on a surface comes from everywhere. Light from one surface is reflected onto another surface and then another, and another, and...

Methods that take this kind of multi-bounce lighting into account are called global lighting methods.

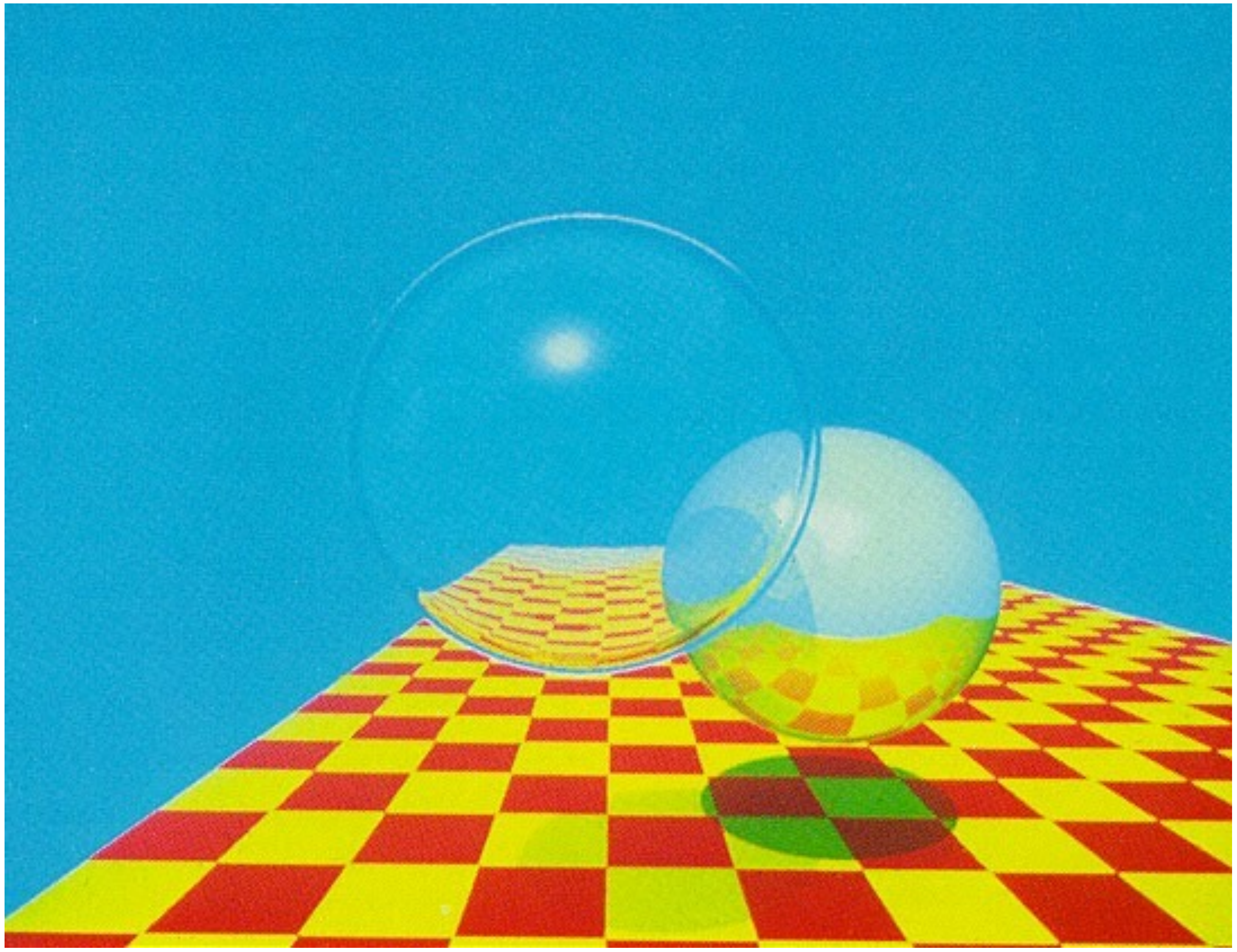
Raytracing and Radiosity

There are two main methods for global lighting:

- Raytracing models specular reflection and refraction.
- Radiosity models diffuse reflection.

Both methods are computationally expensive and are rarely suitable for real-time rendering.

Ray Tracing – 1980s



Ray tracing - 2006

Ray tracing - 2016

https://www.youtube.com/watch?v=uxE2SYDHFtQ

Ray tracing

Ray tracing is a different approach to rendering than the pipeline we have seen so far.

In the OpenGL pipeline we model objects as meshes of polygons which we convert into fragments and then display (or not).

In ray tracing, we model objects as implicit forms and compute each pixel by casting a ray and seeing which models it intersects.

Projective Methods vs RayTracing

- Projective Methods:

  For each **object**

  Find and update each

  pixel it influences

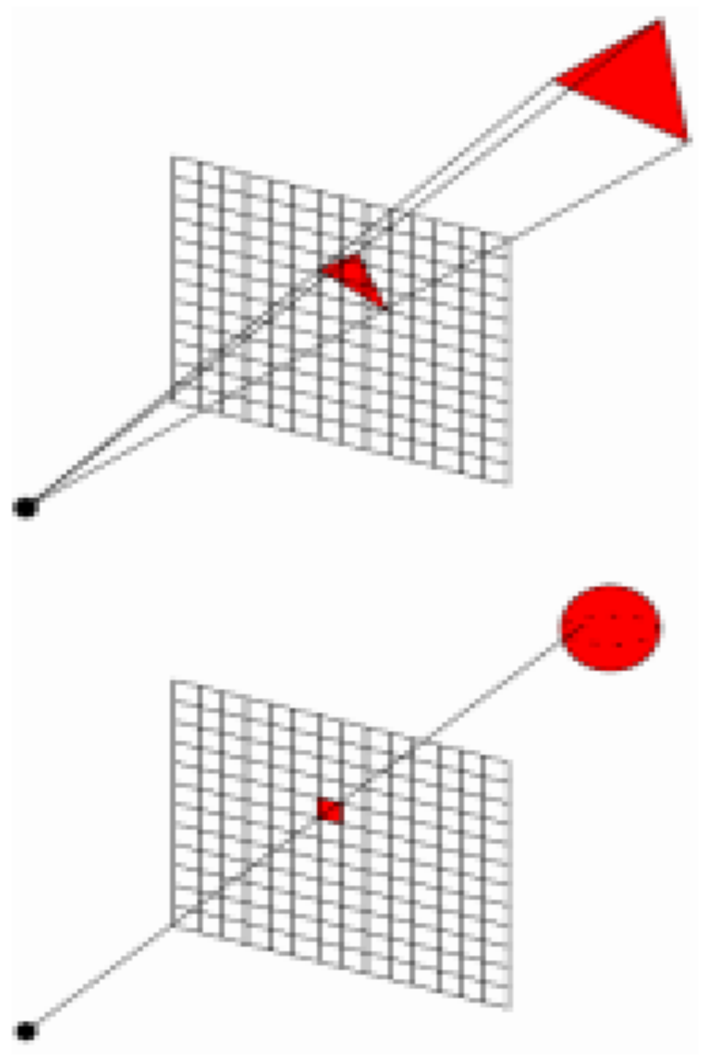
- Ray Tracing:

  For each **pixel**

  Find each object that

  influences it and update

  accordingly



Projective Methods vs RayTracing

They share lots of techniques:

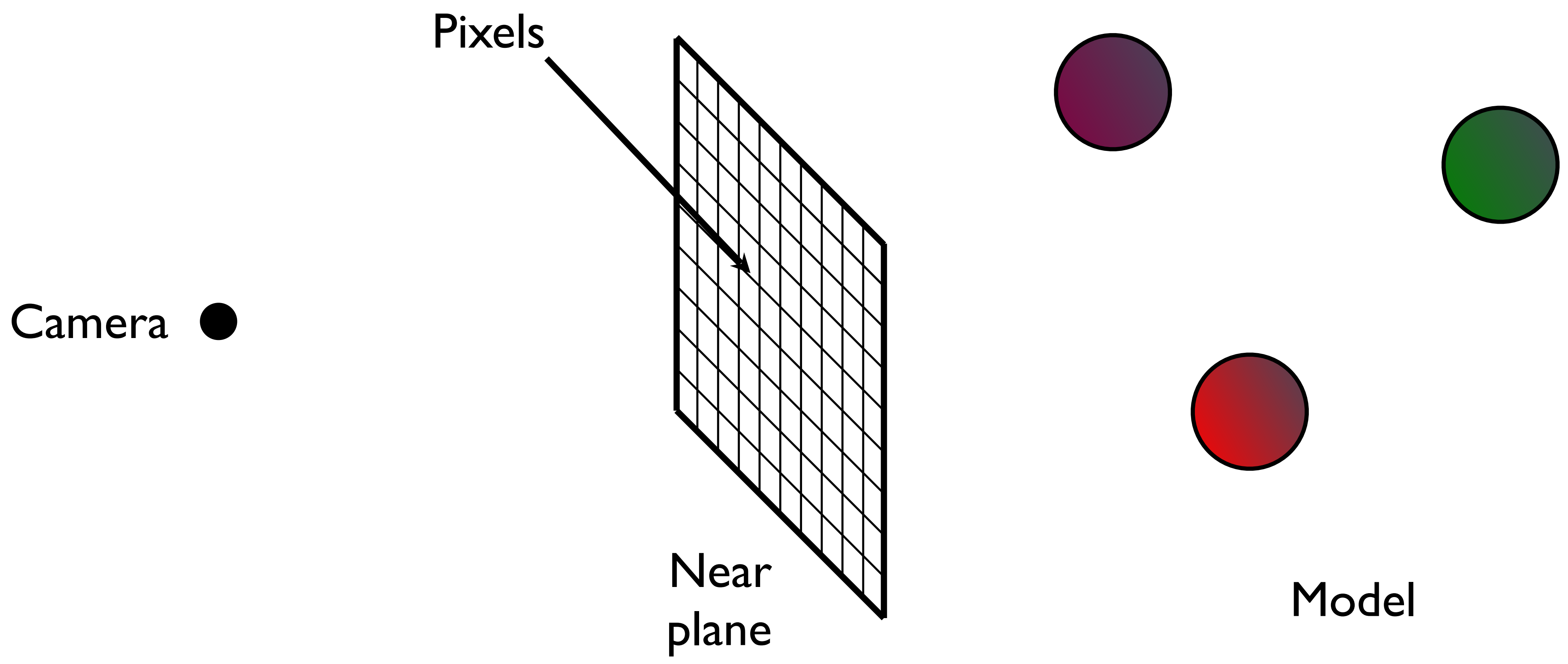
shading models,

calculation of intersections,

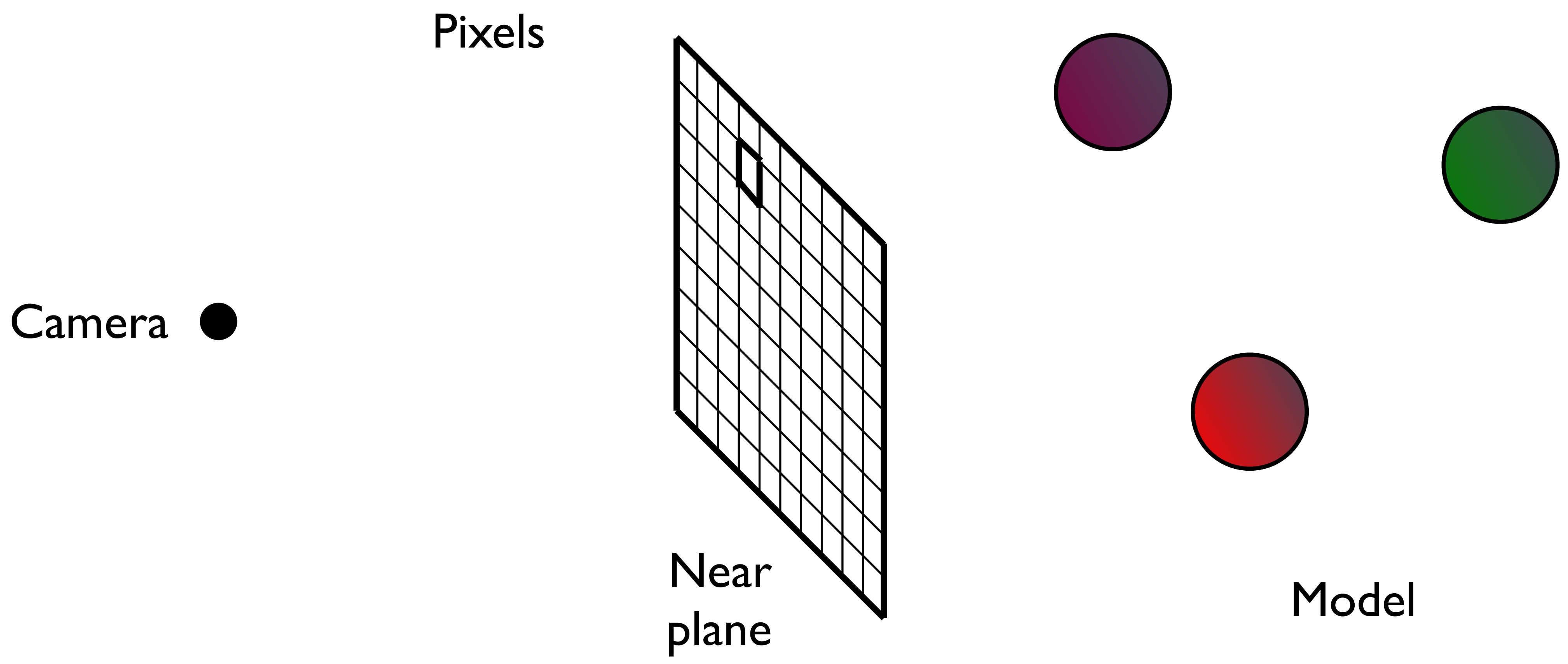
They also have differences:

projection and hidden surface removal come for 'free' in ray tracing

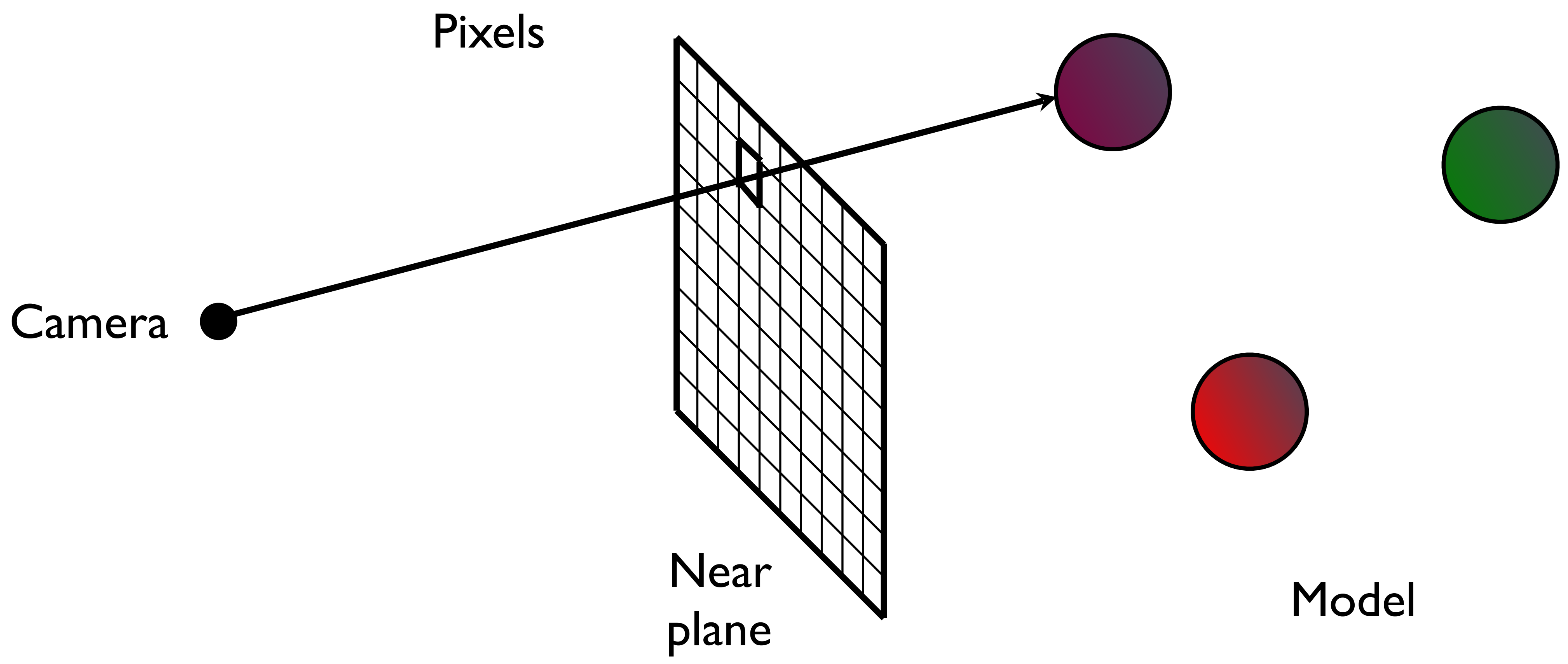
Rays



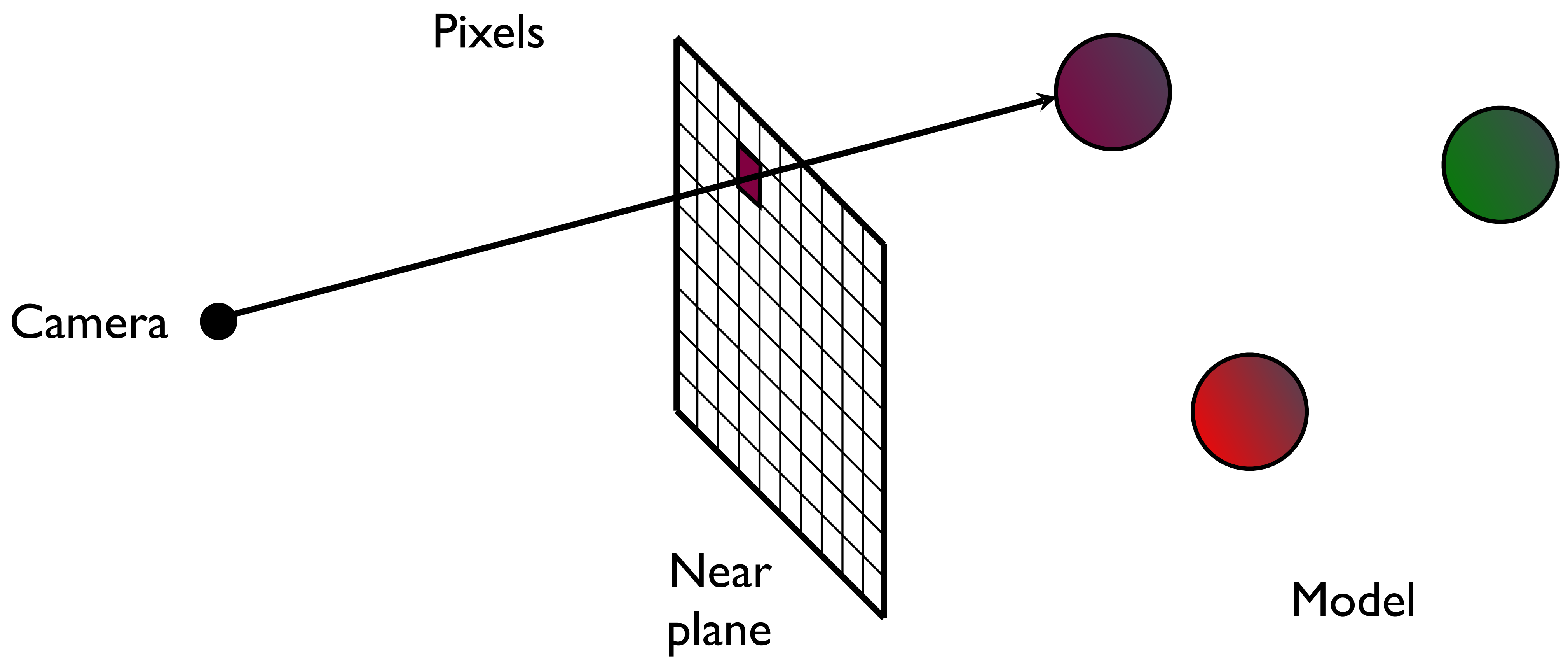
Rays

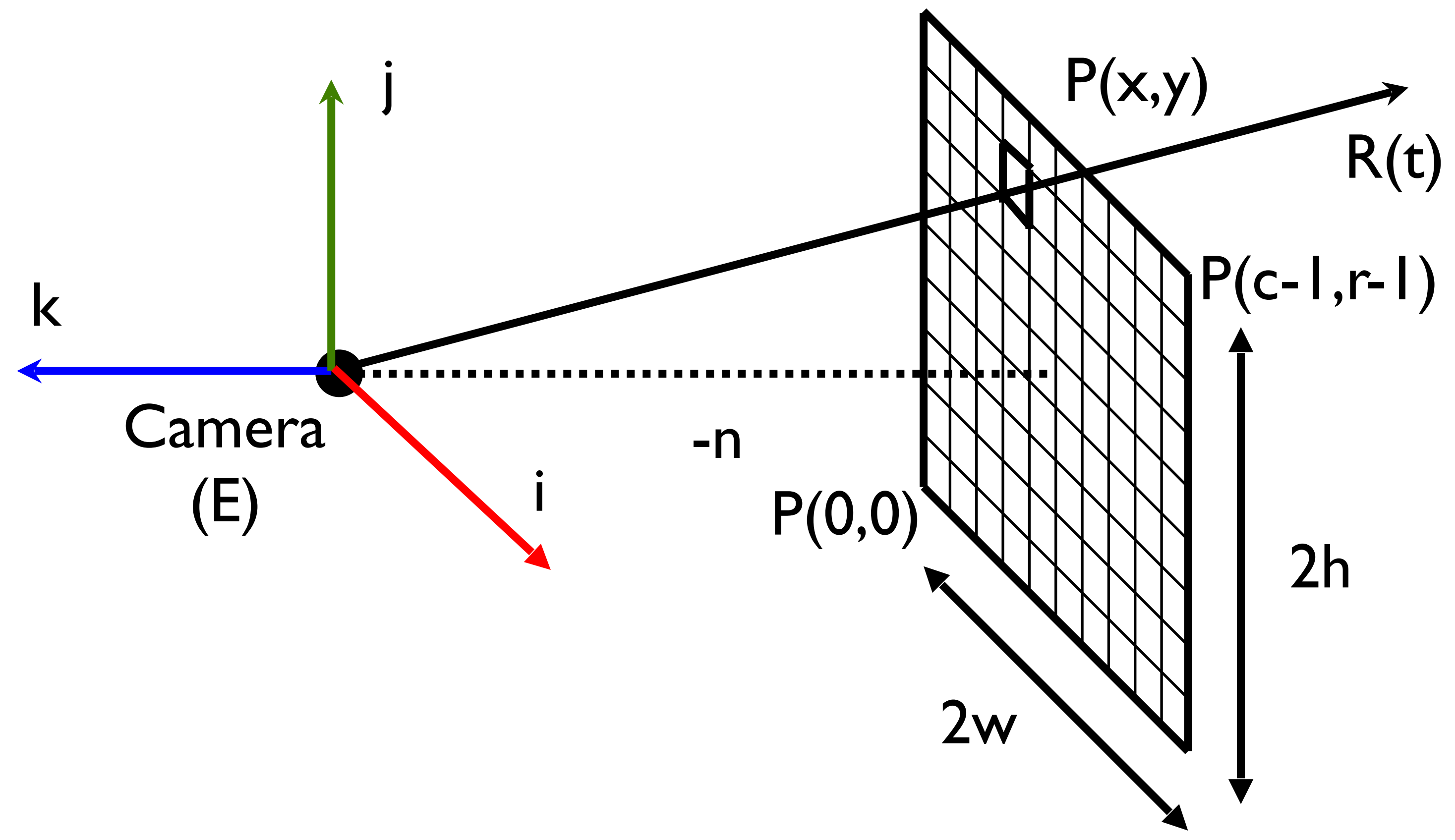


Rays



Rays





- camera coordinate frame (i, j, k, E)
- near plane distance n
- world window 2w by 2h
- viewport (0, 0) to (c-1, r-1) pixels

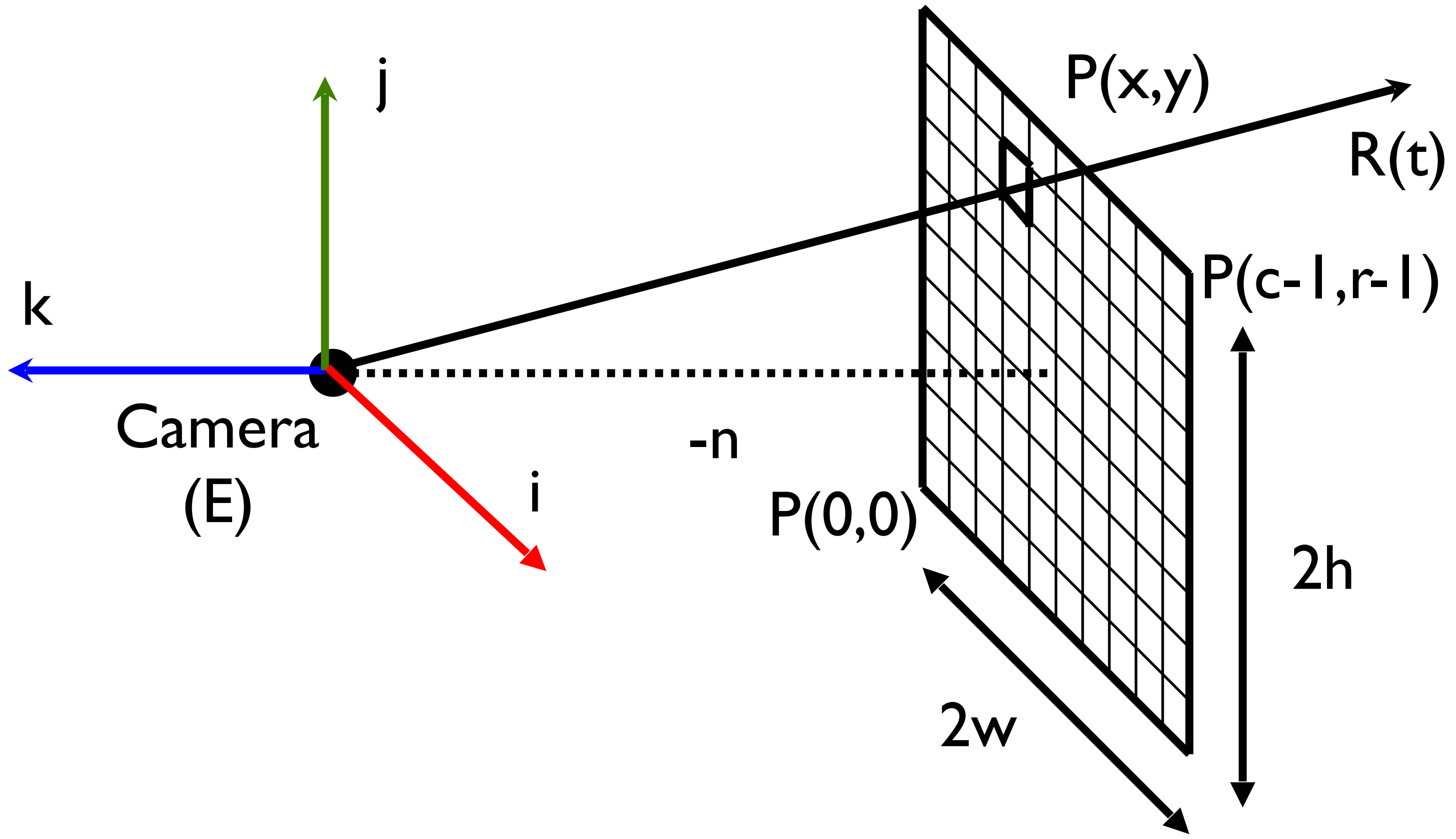
Location of Pixels

Where on the near plane does a given pixel (x,y) appear? (Lower left corners of pixels)

$$pixelWidth = \frac{2w}{c}$$

$$i_c = -w + x\left(\frac{2w}{c}\right)$$

$$= w\left(\frac{2x}{c} - 1\right)$$

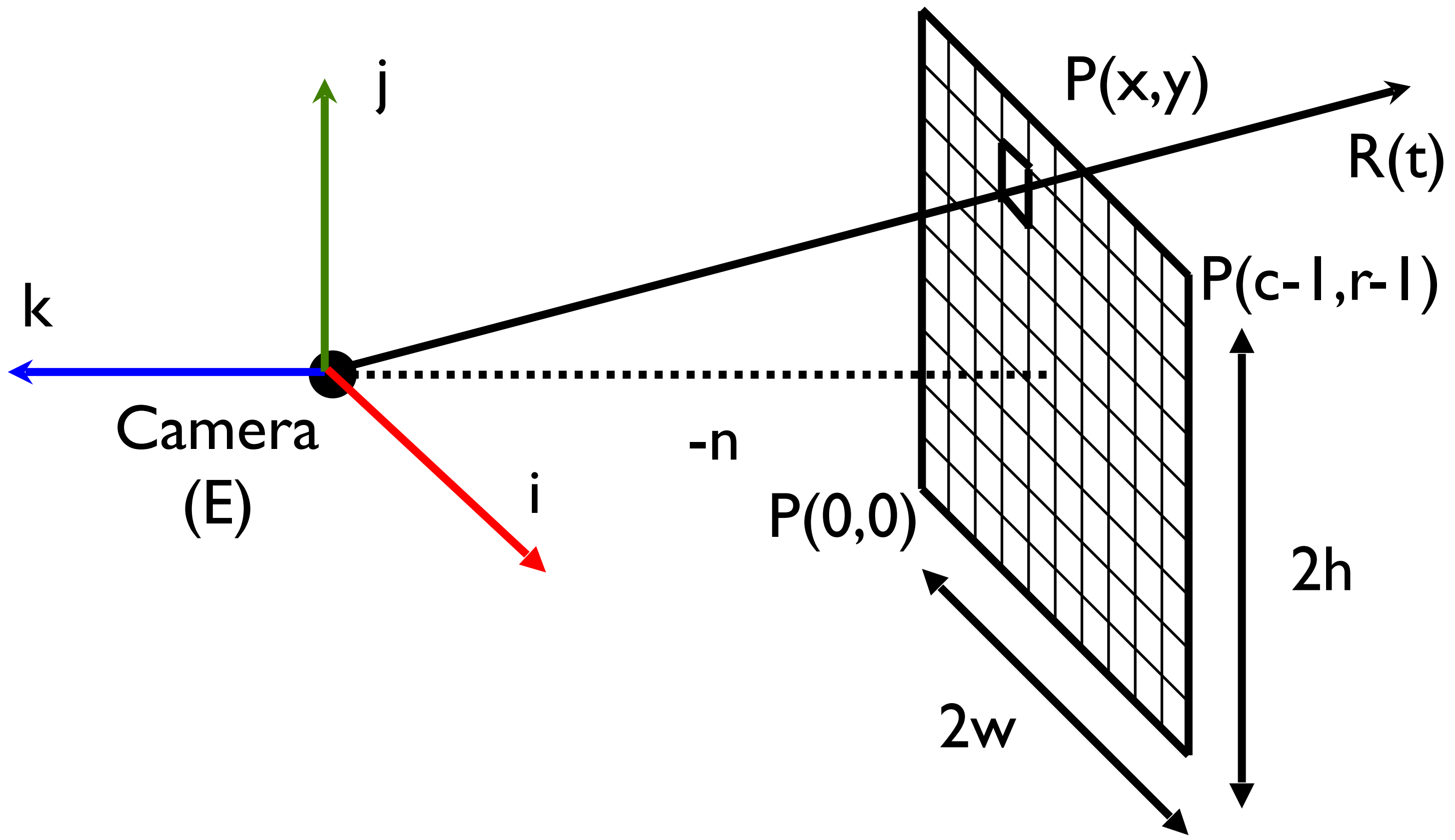


Location of Pixels

Where on the near plane does a given pixel (x,y) appear? (Lower left corners of pixels)

$$pixelHeight = \frac{2y}{r}$$

$$j_r = h\left(\frac{2y}{r} - 1\right)$$



Rays

The point P(x,y) of pixel (x,y) is given by:

$$P(x,y) \quad = \quad E + w(\tfrac{2x}{c} - 1)\mathbf{i} + h(\tfrac{2y}{r} - 1)\mathbf{j} - n\mathbf{k}$$

A ray from the camera through P(x,y) is given by:

$$\begin{aligned} R(t) &= E + t(P(x,y) - E) \\ &= E + t\mathbf{v} \\ \mathbf{v} &= w(\tfrac{2x}{c} - 1)\mathbf{i} + h(\tfrac{2y}{r} - 1)\mathbf{j} - n\mathbf{k} \end{aligned}$$

Rays

$$R(t) = E + t(P(x,y) - E)$$

$$= E + t\mathbf{v}$$

When:

t = 0, we get E (Eye/Camera)

t = 1, we get P(x,y) – the point on the near plane

t > 1 point in the world

t < 0 point behind the camera – not on ray

Intersections

We want to compute where this ray intersects with objects in the scene.

For basic shapes, we can do this with the equation of the shape in implicit form:

$$F(x, y, z) = 0$$

which we can also write as:

$$F(P) = 0$$

We substitute the formula for the ray into F and solve for t.

Intersecting a generic sphere

For example, a unit sphere at the origin has implicit form:

$$F(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$$

or:

$$F(P) = |P|^2 - 1 = 0$$

Intersecting a generic sphere

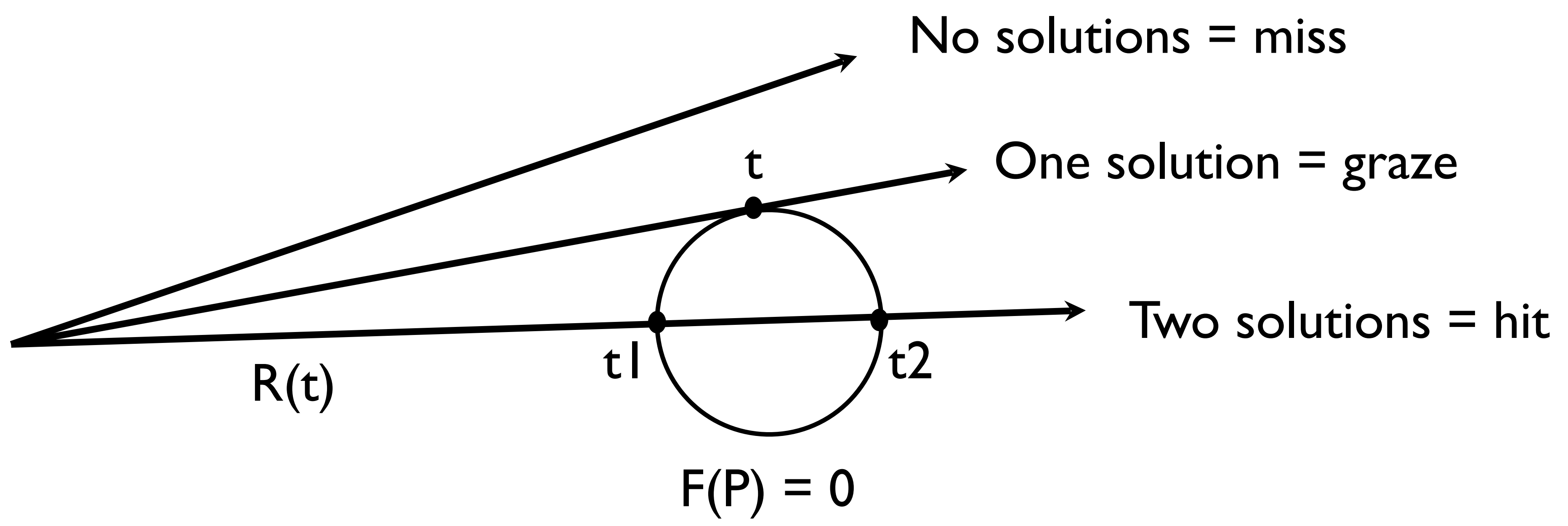
We substitute the ray equation into F and solve for t:

$$\begin{aligned} F(R(t)) &= 0 \\ |R(t)|^2 - 1 &= 0 \\ |E + \mathbf{v}t|^2 - 1 &= 0 \\ |\mathbf{v}|^2 t^2 + 2(E \cdot \mathbf{v})t + (|E|^2 - 1) &= 0 \end{aligned}$$

which we can solve for t (as a quadratic).

Intersecting a generic sphere

We will get zero, one or two solutions:



$$t_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Exercise

Where is the intersection of

$$r(t) = (3,2,3) + (-3,-2,-3)t$$

With the generic sphere?

$$a = |\mathbf{v}|^2 = |(-3,-2,-3)|^2 = 22$$
$$b = 2(E \cdot \mathbf{v}) = 2((3,2,3) \cdot (-3,-2,-3)) = -44$$
$$c = |E|^2 - 1 = |(3,2,3)|^2 - 1 = 21$$

Exercise…

$$t = \frac{44 \pm \sqrt{-44^2 - 4 \times 22 \times 21}}{44}$$

$$= 1 \pm 0.2132$$

$$t_1 = 0.7868$$

$$t_2 = 1.2132$$

Points are:

$$(3,2,3) + 0.768(-3,-2,-3) = (0.64,0.43,0.64)$$

$$(3,2,3) + 1.2132(-3,-2,-3) = (-0.64,-0.43,-0.64)$$

Intersecting a generic plane

The x-y plane has implicit form:

$$F(x, y, z) = z = 0$$
$$F(P) = p_z = 0$$

Intersecting with the ray:

$$\begin{aligned} F(R(t)) &= 0 \\ E_z + t\mathbf{v}_z &= 0 \\ t &= -\frac{E_z}{\mathbf{v}_z} \end{aligned}$$

Intersecting a generic cube

To compute intersections with the generic cube (-1,-1,-1) to (1,1,1) we apply the Cyrus-Beck clipping algorithm encountered in week 3. Extending the algorithm to 3D is straightforward.

The same algorithm can be used to compute intersections with arbitrary convex polyhedral and meshes of convex faces.

Non-generic solids

We can avoid writing special-purpose code to calculate intersections with non-generic spheres, boxes, planes, etc.

Instead we can transform the ray and test it against the generic version of the shape.

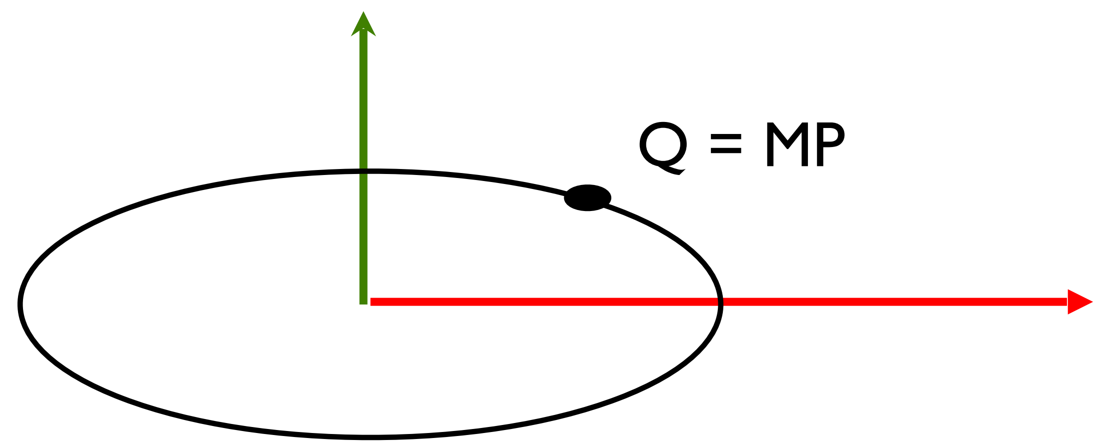
Transformed spheres

We can transform a sphere by applying affine transformations

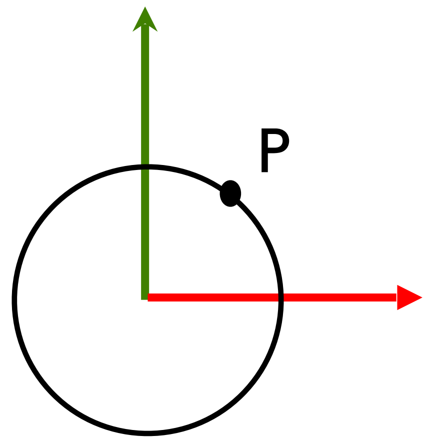
Let P be a point on the generic sphere.

We can create an arbitrary ellipsoid by transforming P to a new coordinate frame given by a matrix M.

2D example



Transformed circle
$F(M^{-1}Q) = 0$



Generic circle
F(P) = 0

$$\begin{aligned} F(P) &= 0 \\ Q &= \mathbf{M}P \\ P &= \mathbf{M}^{-1}Q \\ F(\mathbf{M}^{-1}Q) &= 0 \end{aligned}$$

Non-generic solids

So in general if we apply a coordinate transformation M to a generic solid with implicit equation F(P) = 0 we get:

$$\begin{aligned} F(\mathbf{M}^{-1}Q) &= 0 \\ F(\mathbf{M}^{-1}R(t)) &= 0 \\ F(\mathbf{M}^{-1}E + t\mathbf{M}^{-1}\mathbf{v}) &= 0 \end{aligned}$$

Non-generic Solids

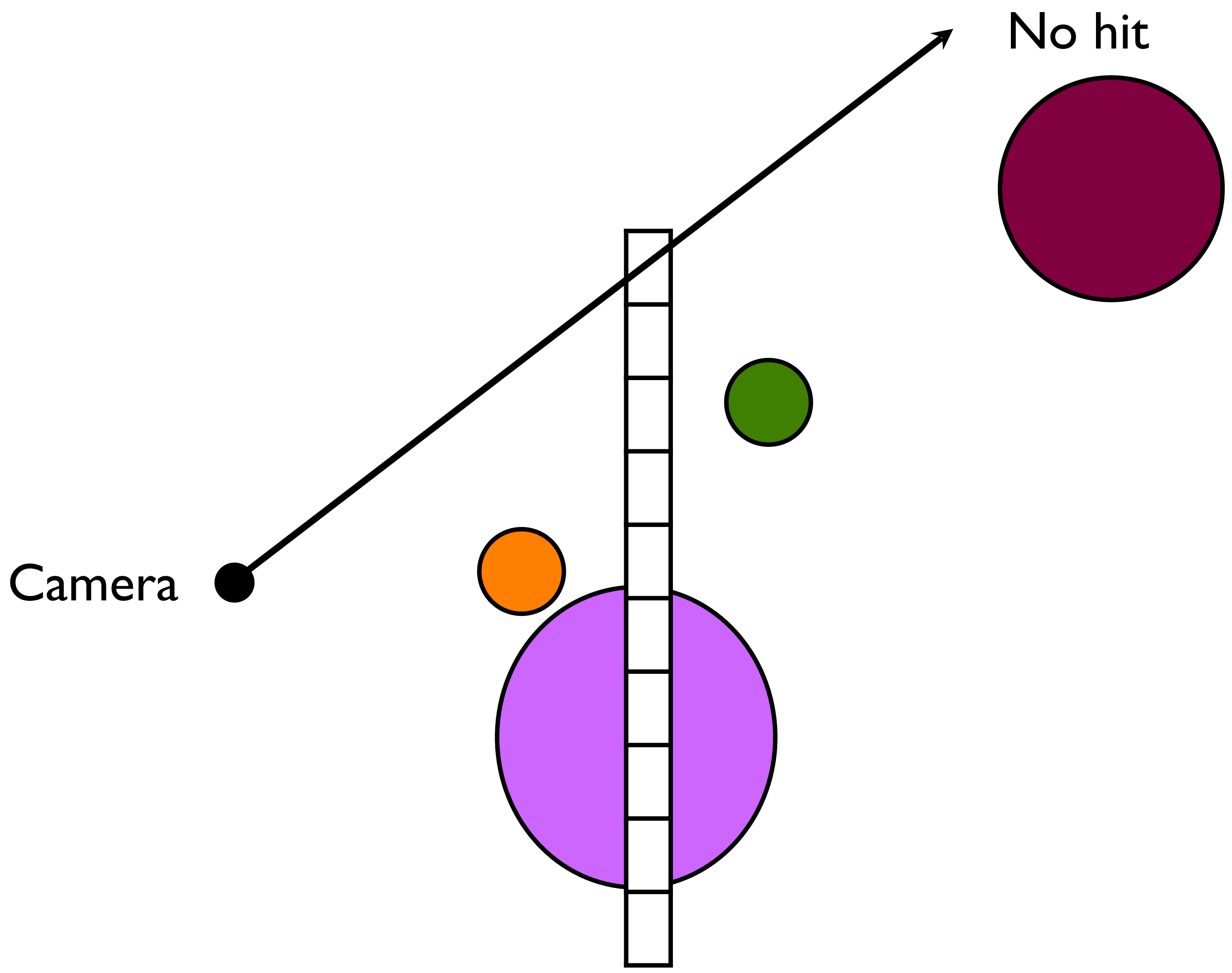
In other words:

- Apply the inverse transformation to the ray.
- Do standard intersection with the generic form of the object.
- Affine transformations preserve relative distances so values of t will be valid.

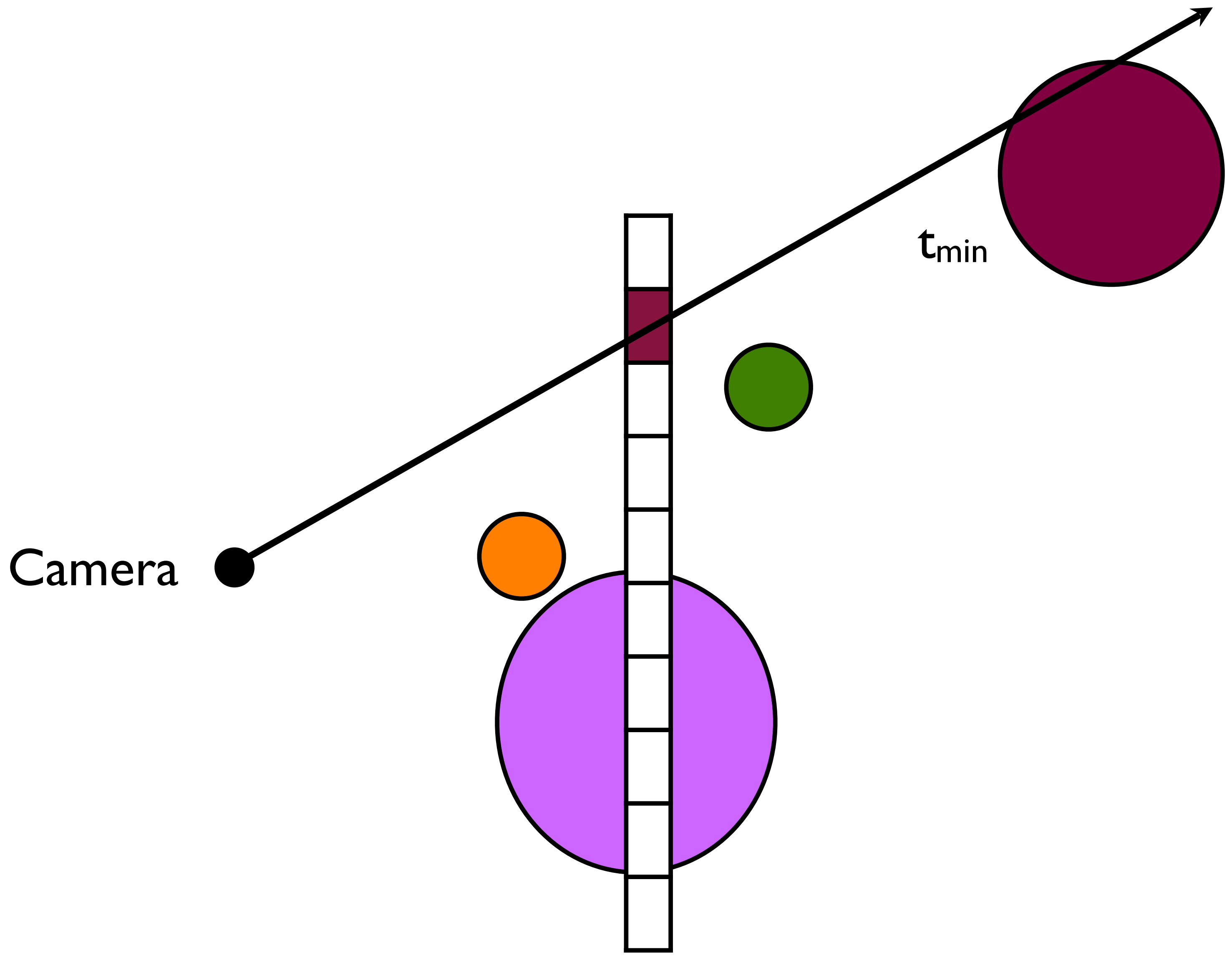
Ray Tracing Pseudocode

```
for each pixel (x,y):

   v = P(x,y) - E
   hits = {};

   for each object obj in the scene:

      E' = $M^{-1}$ * E
      v' = $M^{-1}$ * v

      hits.add(obj.hit(E', v'))

   hit = h in hits with min time > 1

   if (hit is null)
      set (x,y) to background
   else
      set (x,y) to hit.obj.colour(R(hit.time))
```

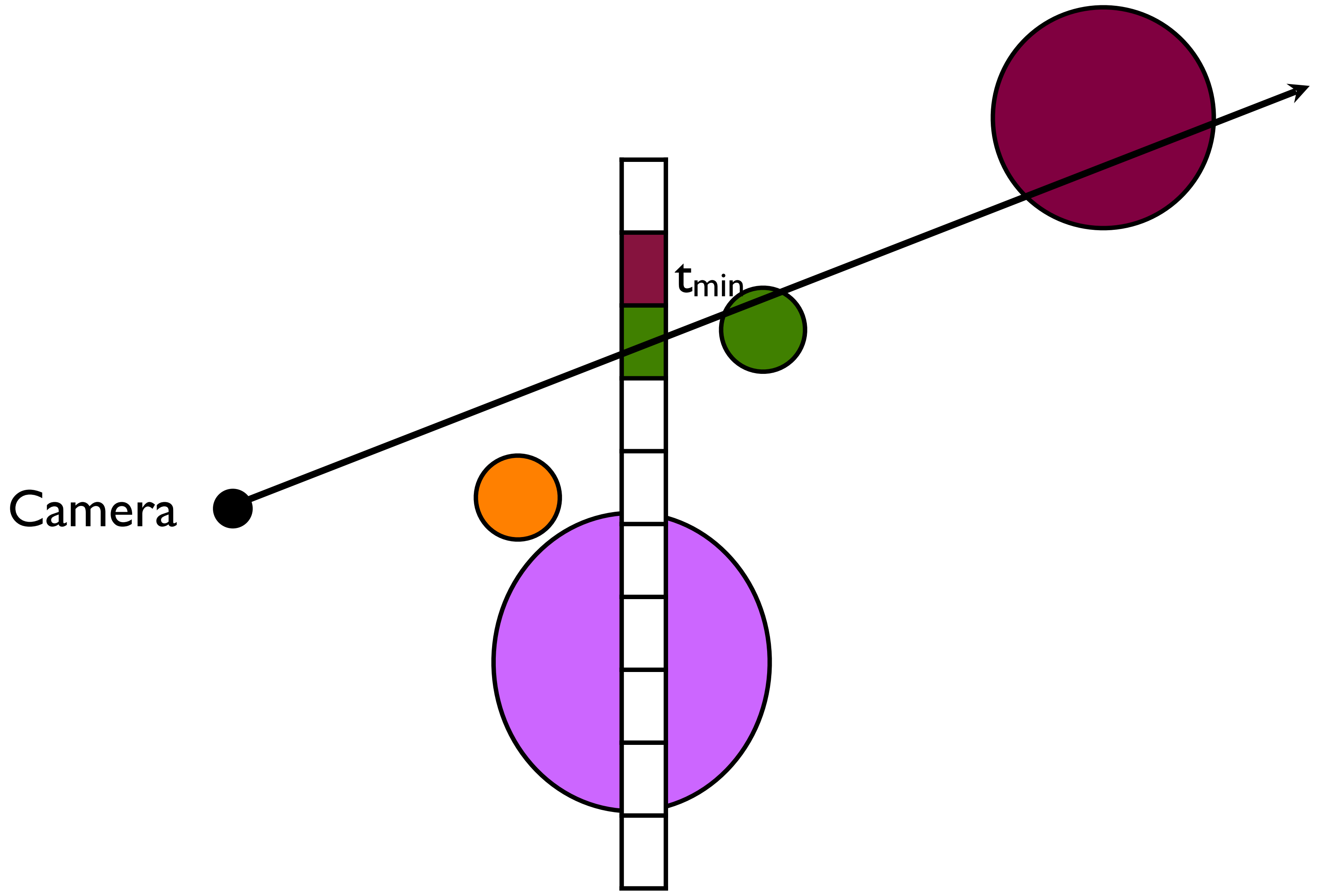
2D Example



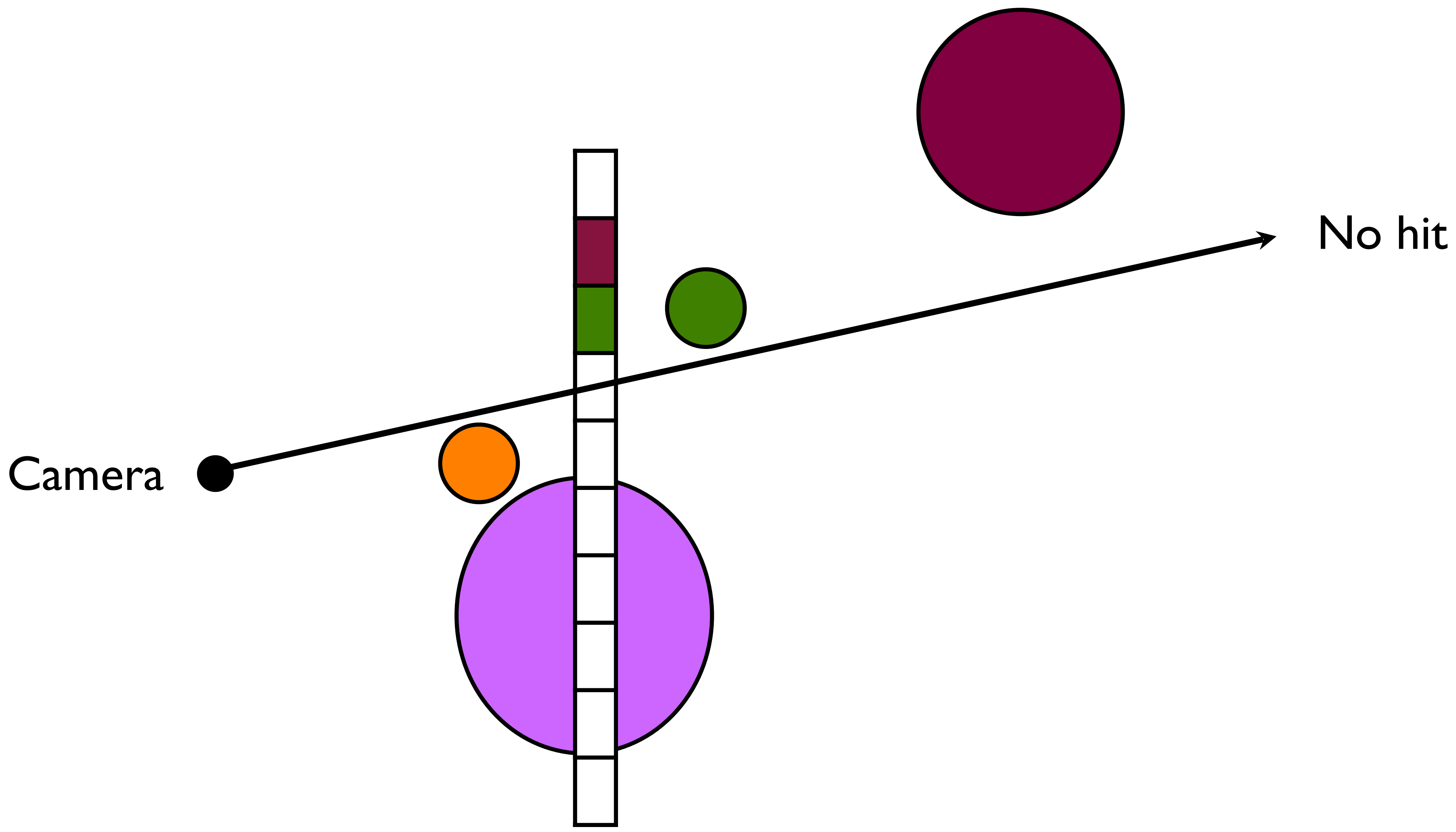
2D Example



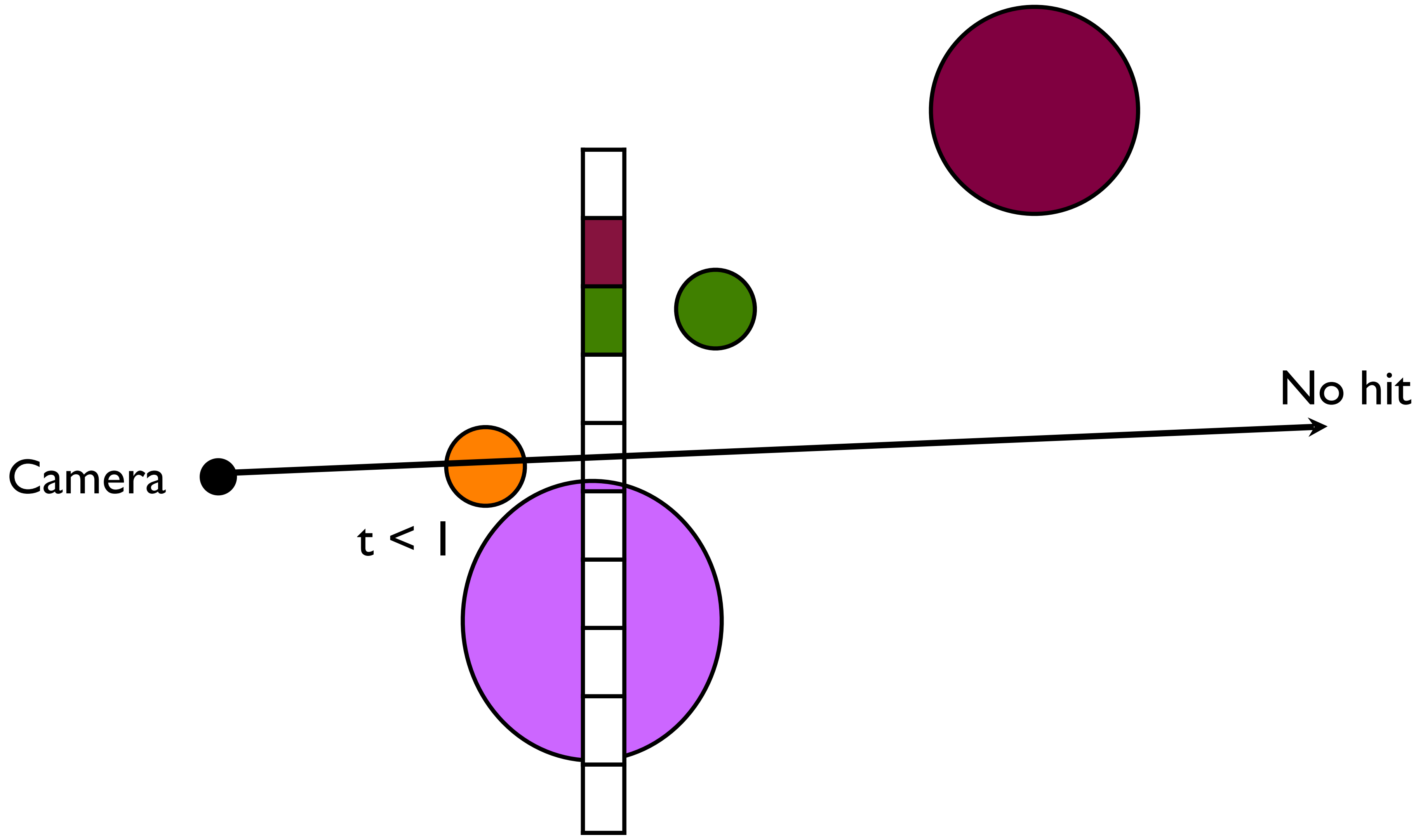
2D Example



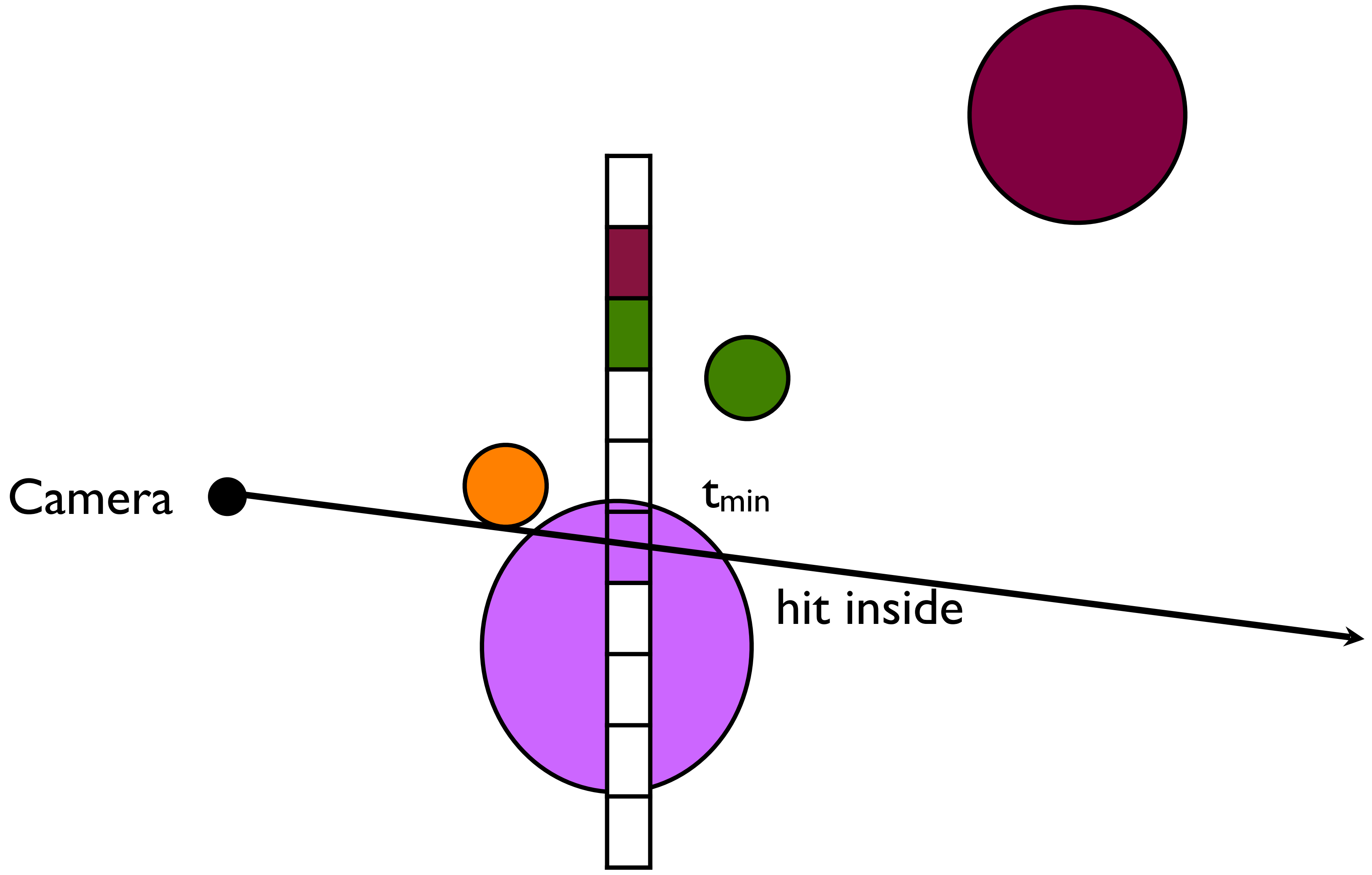
2D Example



2D Example



2D Example



Shading & Texturing

When we know the object we hit and the point at which the hit occurs, we can compute the lighting equation to get the illumination.

Likewise if the object has a texture we can compute the texture coordinates for the hit point to calculate its colour.

We combine these as usual to compute the pixel colour.

Antialiasing

We can smooth out aliasing artefacts in our image by supersampling.

For each pixel we cast multiple rays with slight offsets and average the results.

Adaptive sampling is also appropriate here.

Optimisation

Testing collisions for more complex shapes (such as meshes) can be very time consuming.

In a large scene, most rays will not hit the object, so performing multiple expensive collision tests is wasteful.

We want fast ways to rule out objects which will not be hit.

Extents

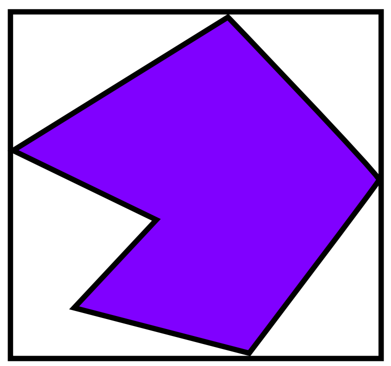
Extents are bounding boxes or spheres which enclose an object.

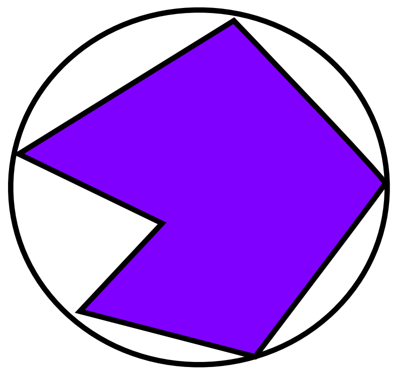
Testing against a box or sphere is fast.

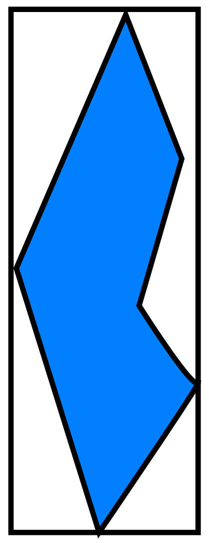
If this test succeeds, then we proceed to test against the object.

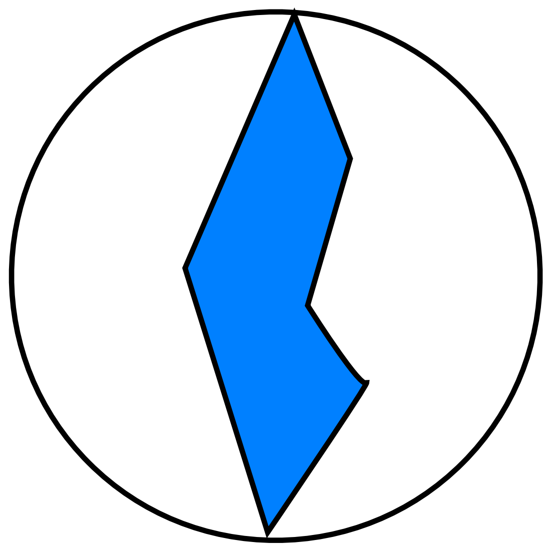
We want tight fitting extents to minimise false positives.

Extents


Good fit


Better fit


Good fit


Poor fit

Computing extents

To compute a box extent for a mesh we simply take the min and max x, y and z coordinates over all the points.

To compute a sphere extent we find the centroid of all the vertices by averaging their coordinates. This is the centre of the sphere.

The radius is the distance to the vertex farthest from this point.

Projection extents

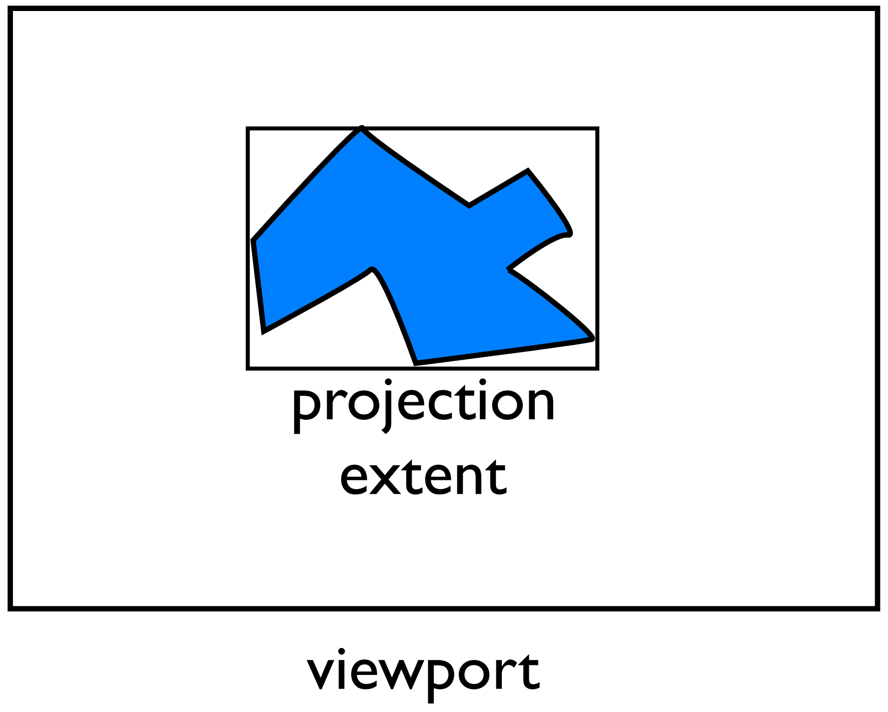
Alternatively, we can build extents in screen space rather than world space.

A projection extent of an object is a bounding box which encloses all the pixels which would be in the image of the object (ignoring occlusions).

Pixels outside this box can ignore the object.

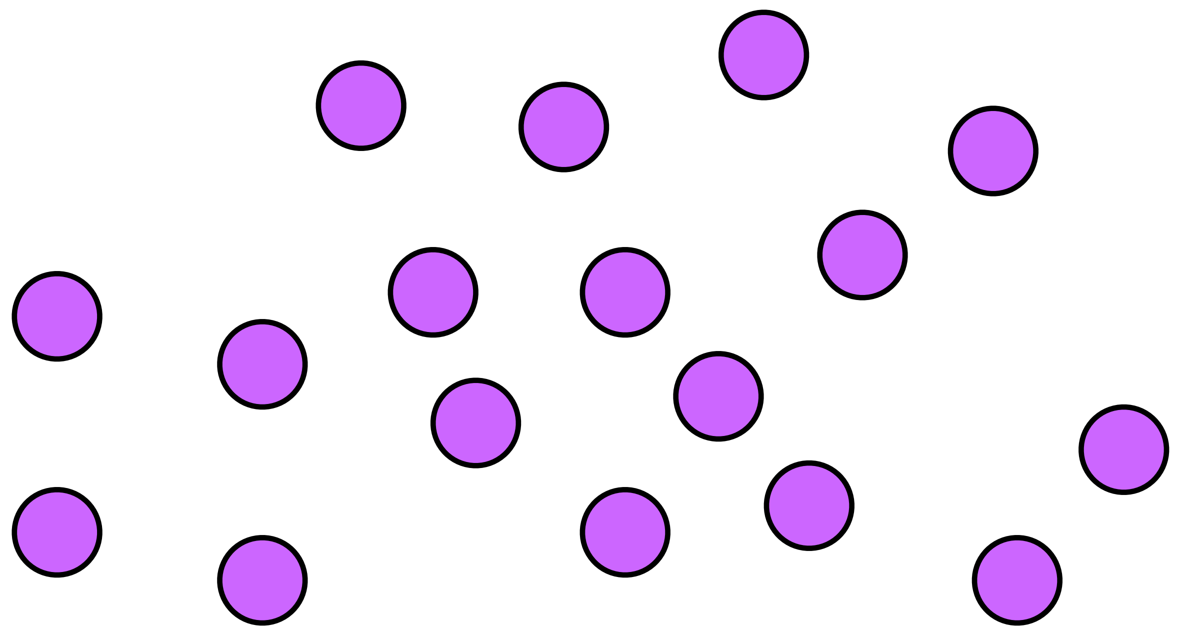
Projection extents

We can compute a projection extent of a mesh by projecting all the vertices into screen space and finding the min and max x and y values.



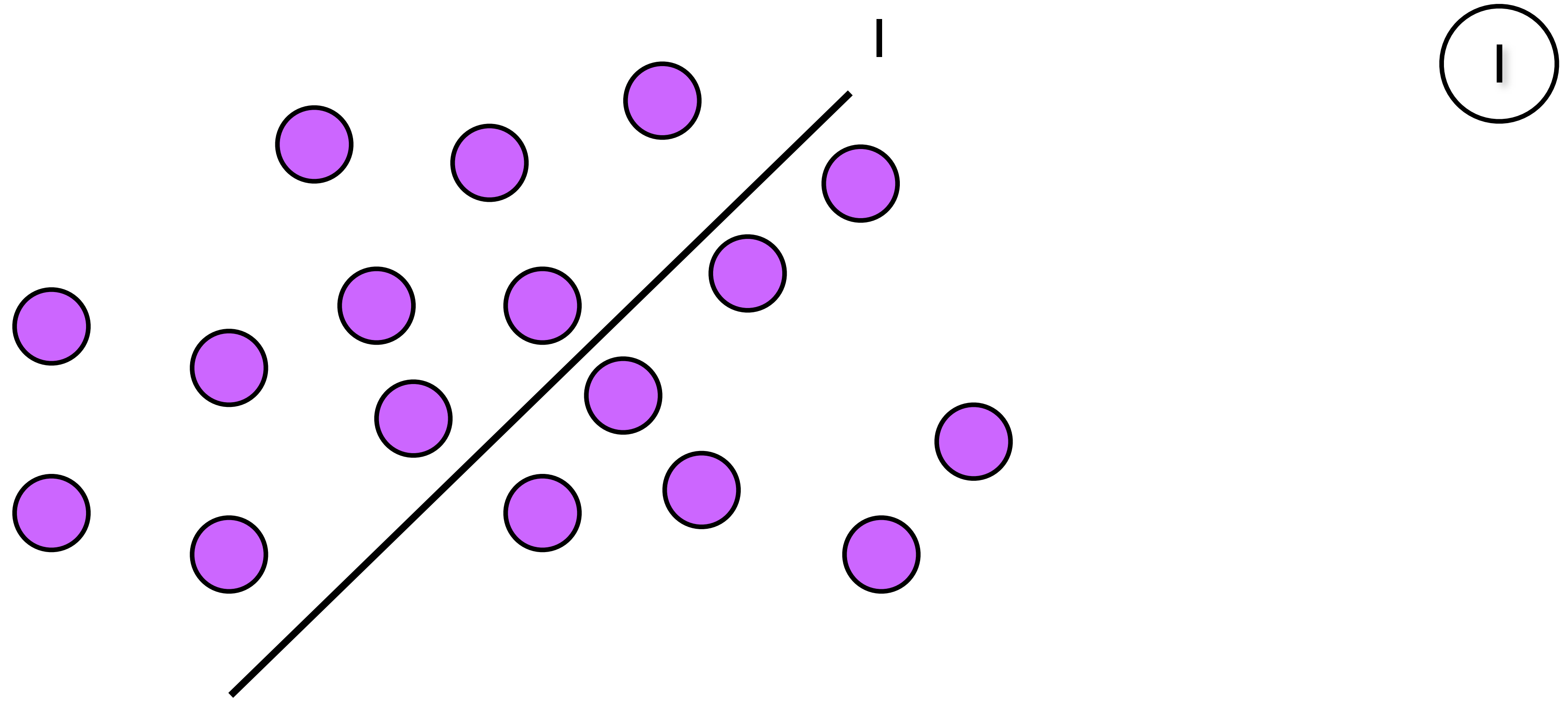
BSPs

Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.



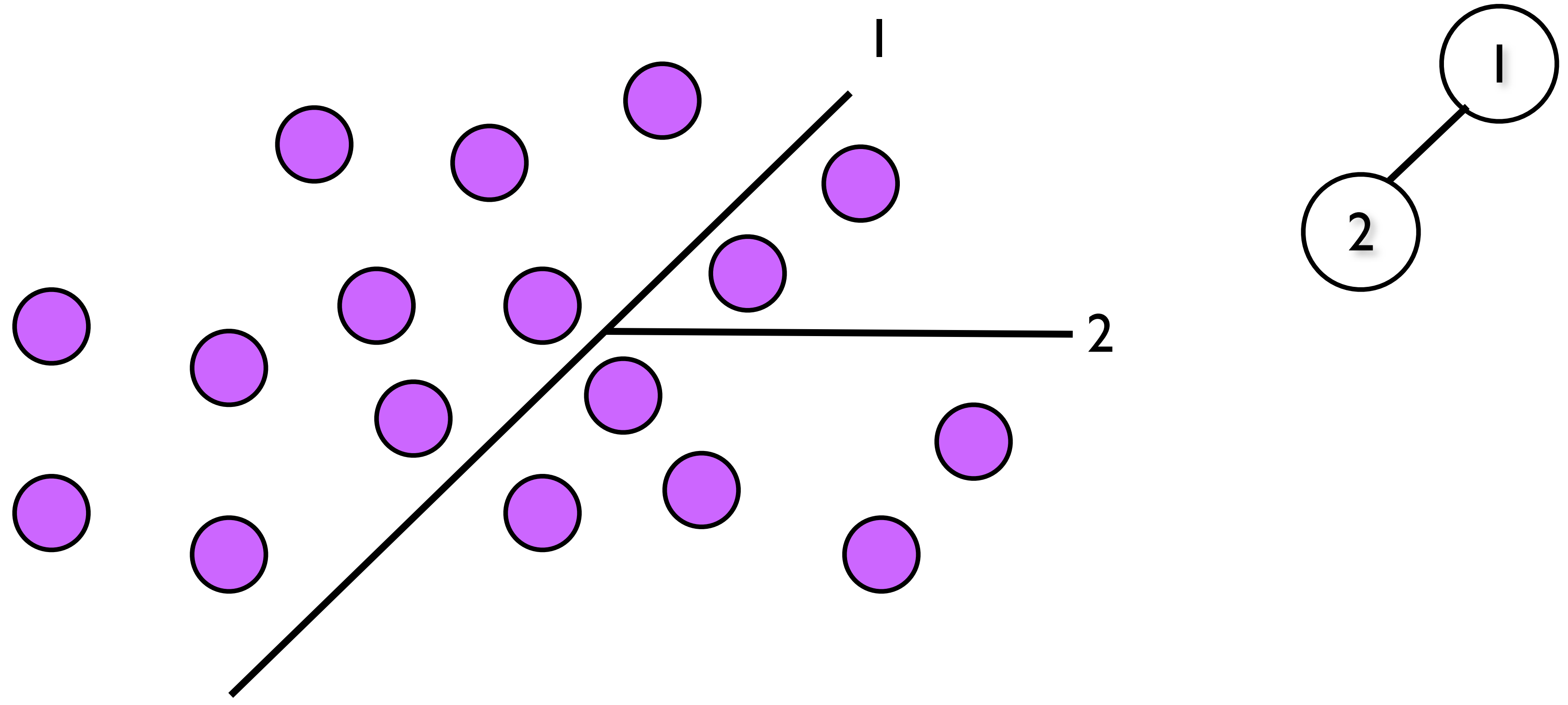
BSPs

Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.



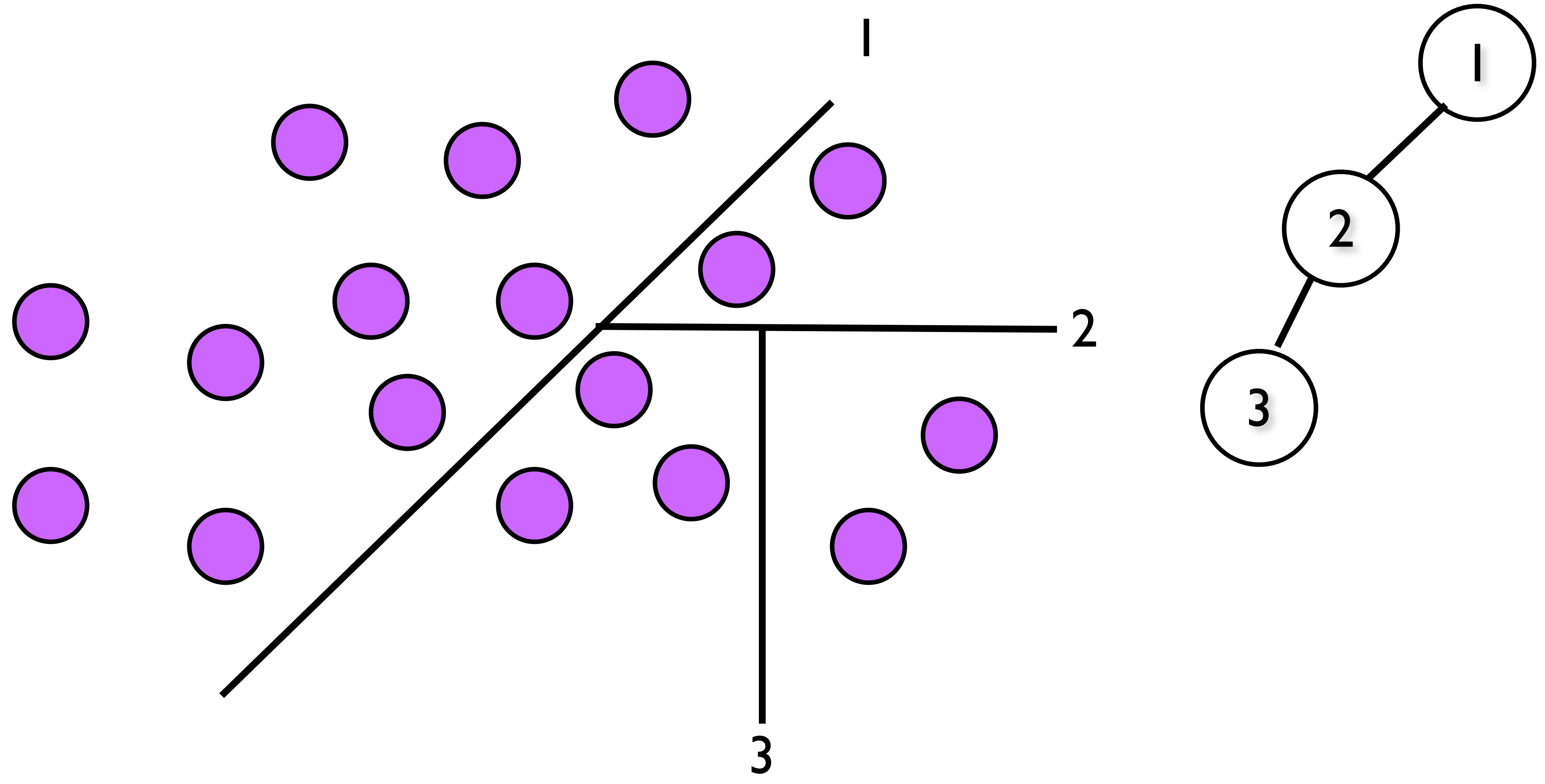
BSPs

Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.



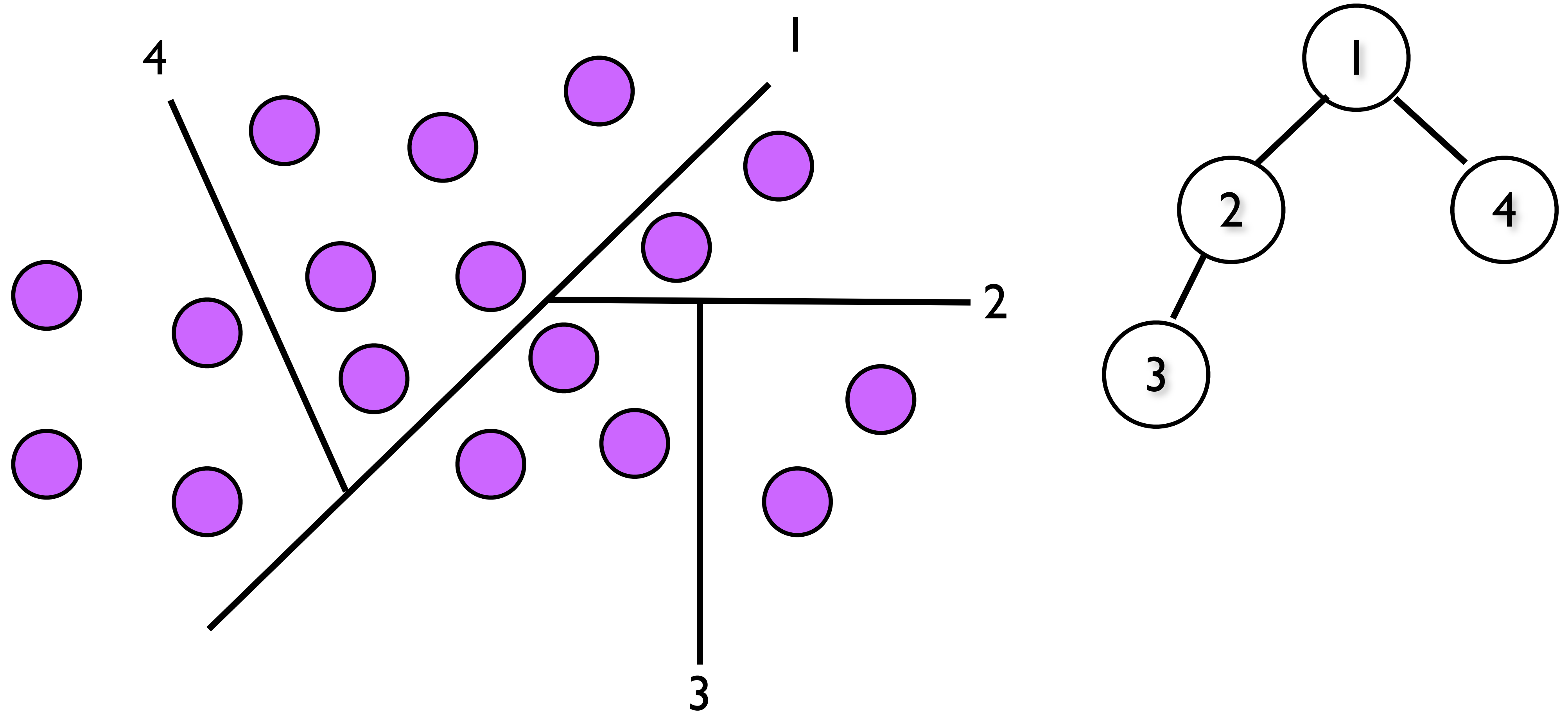
BSPs

Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.



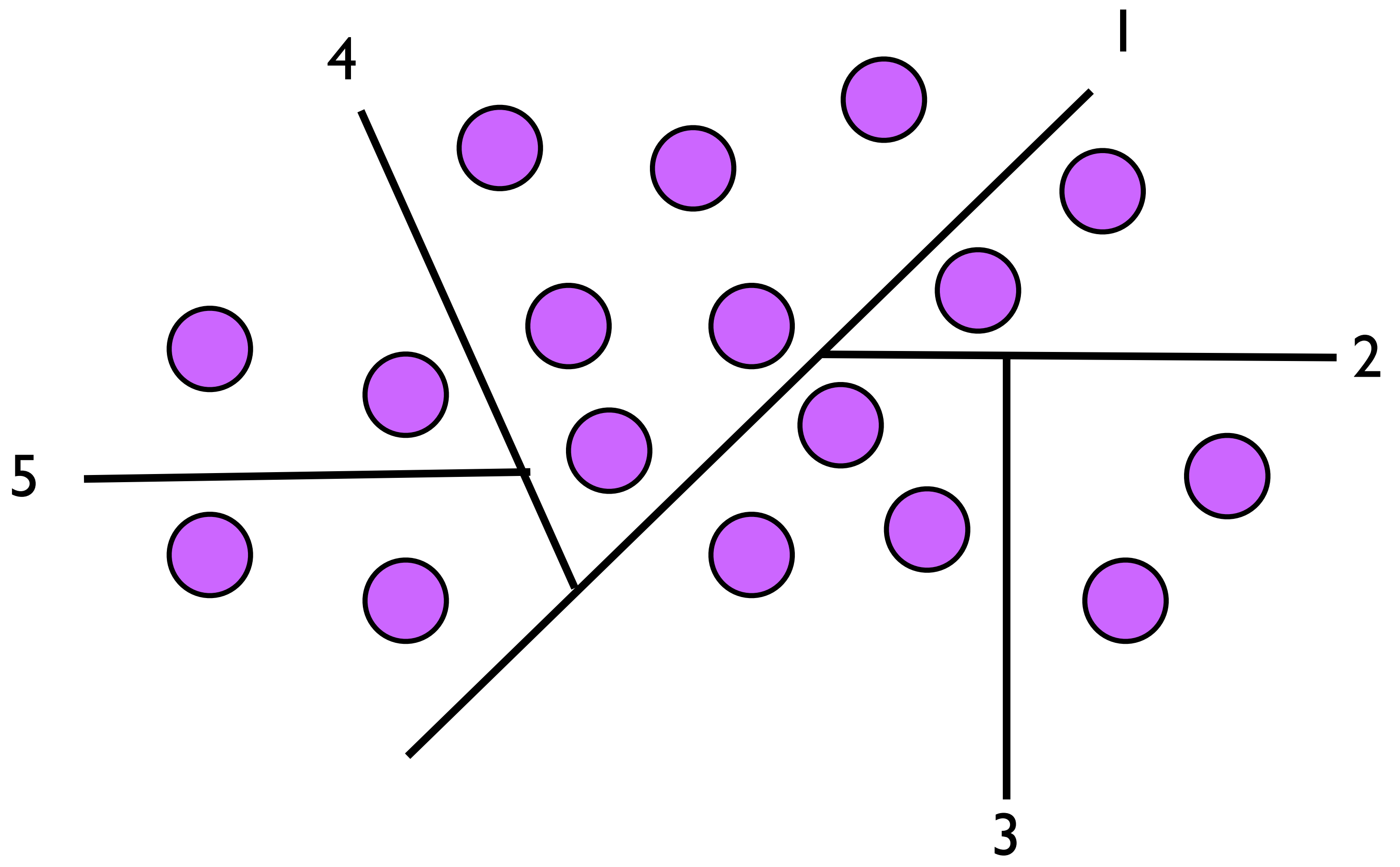
BSPs

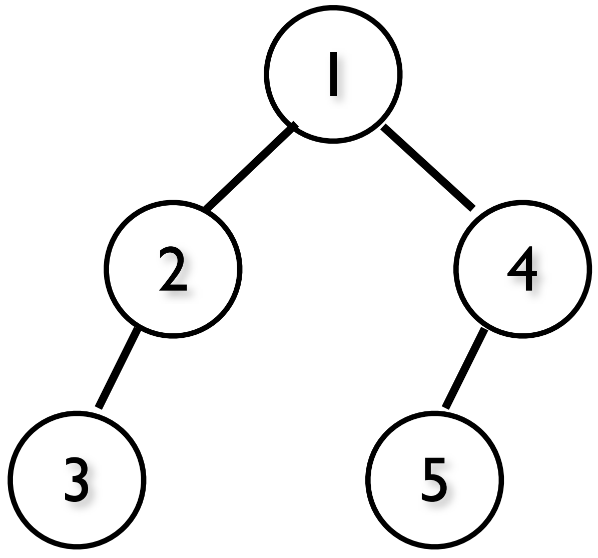
Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.



BSPs

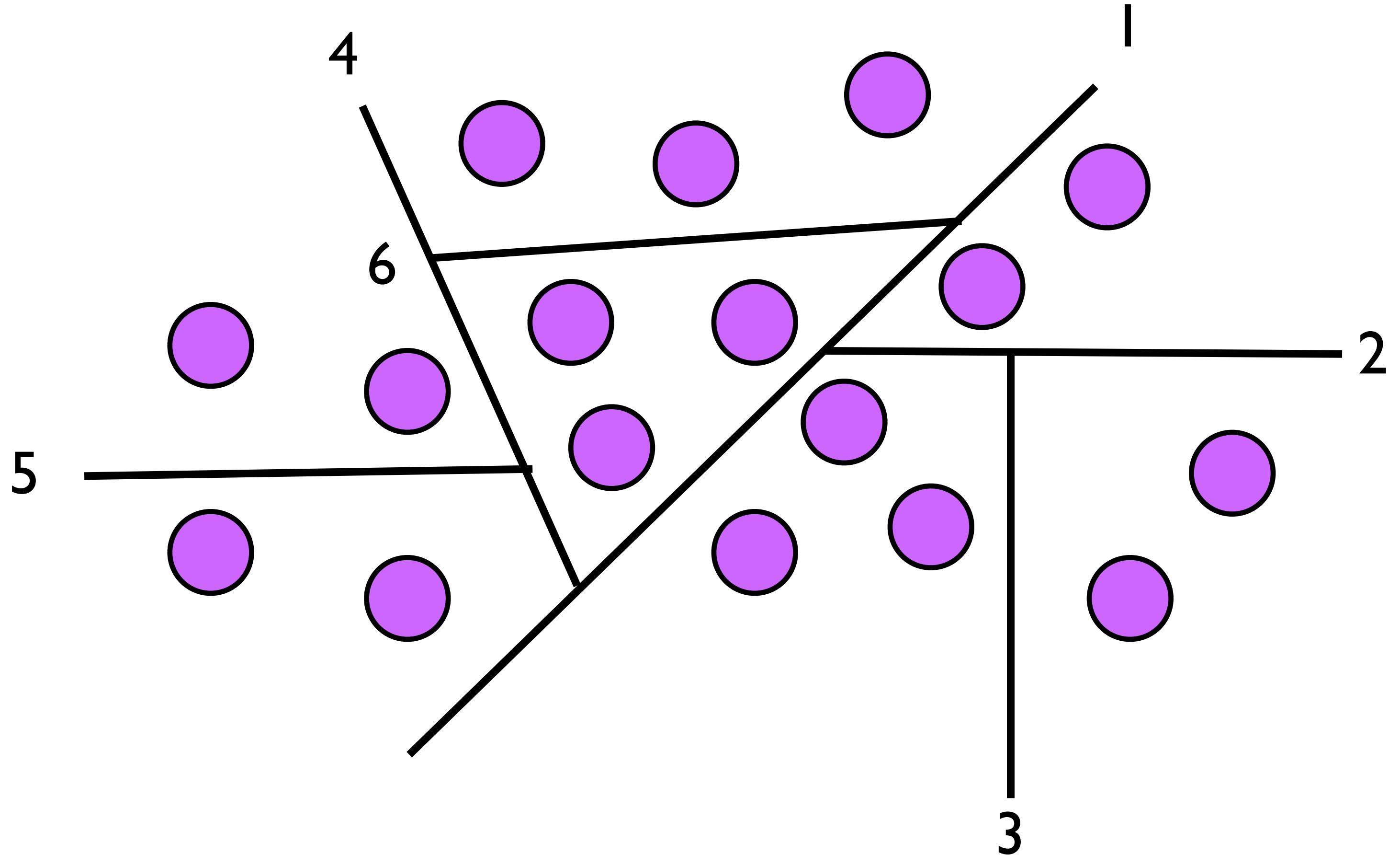
Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.

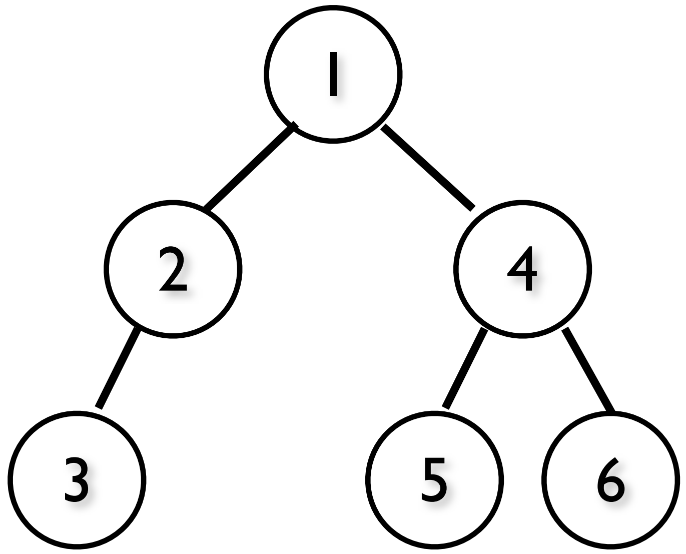




BSPs

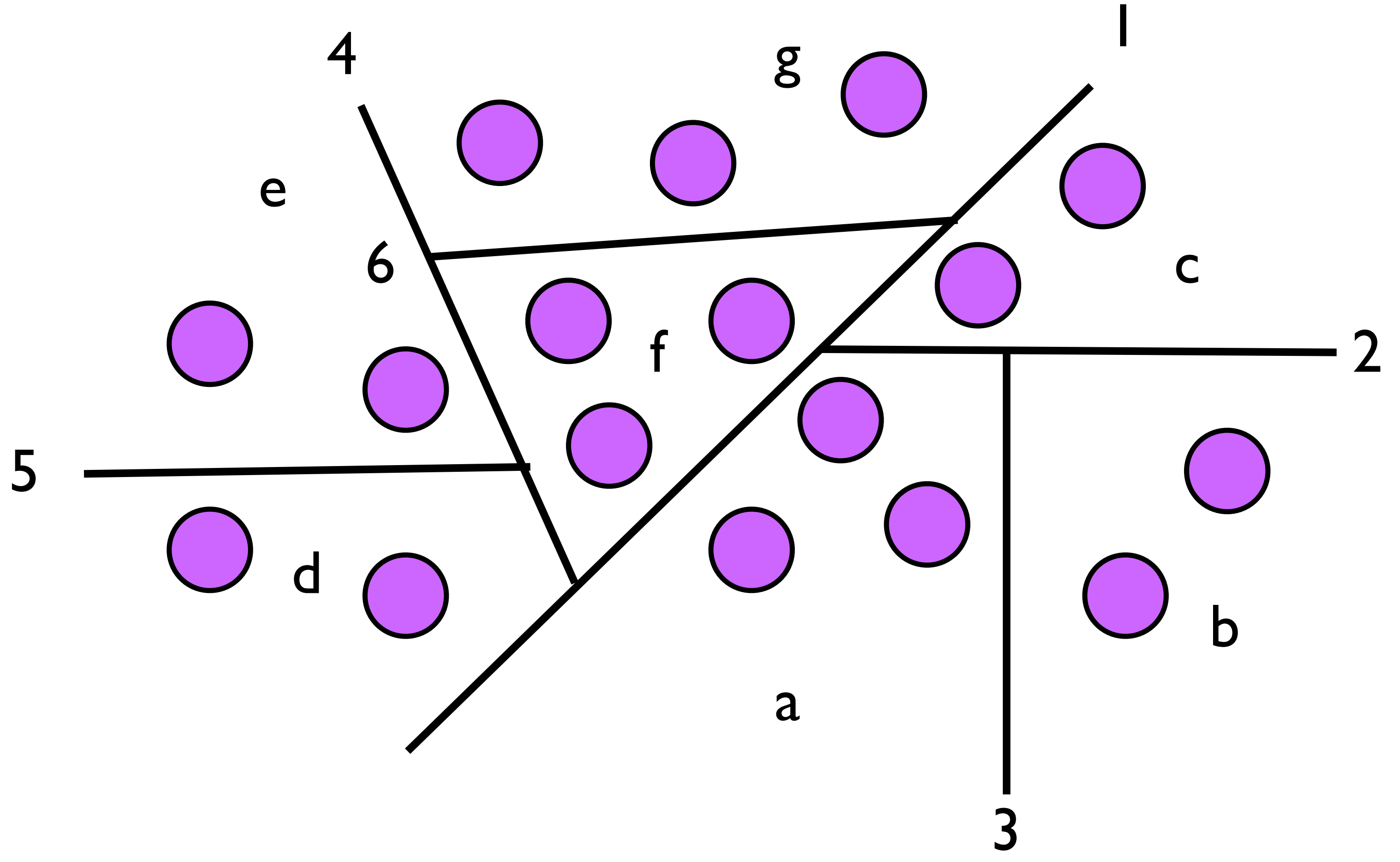
Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.

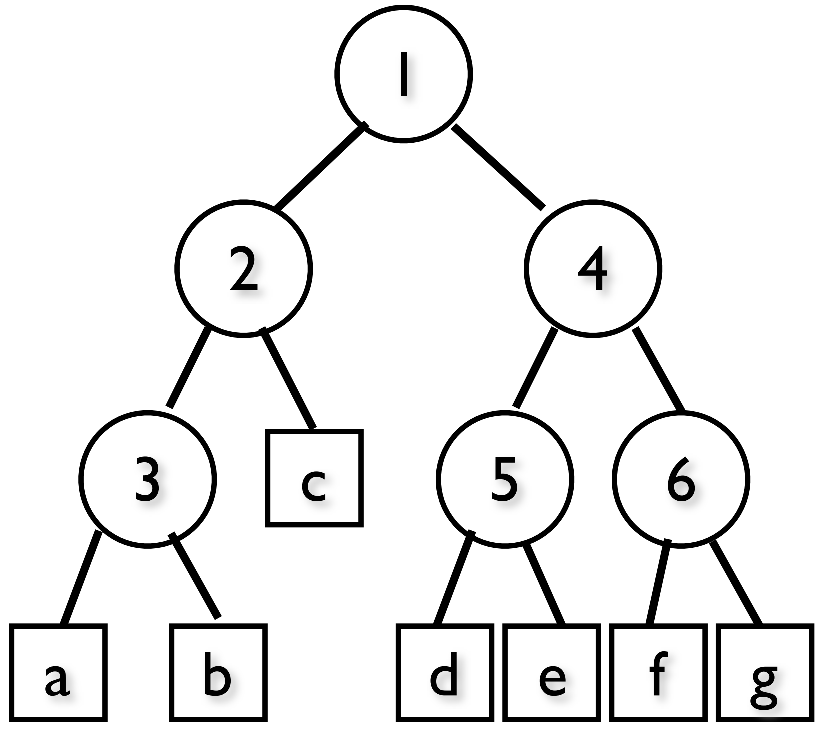




BSPs

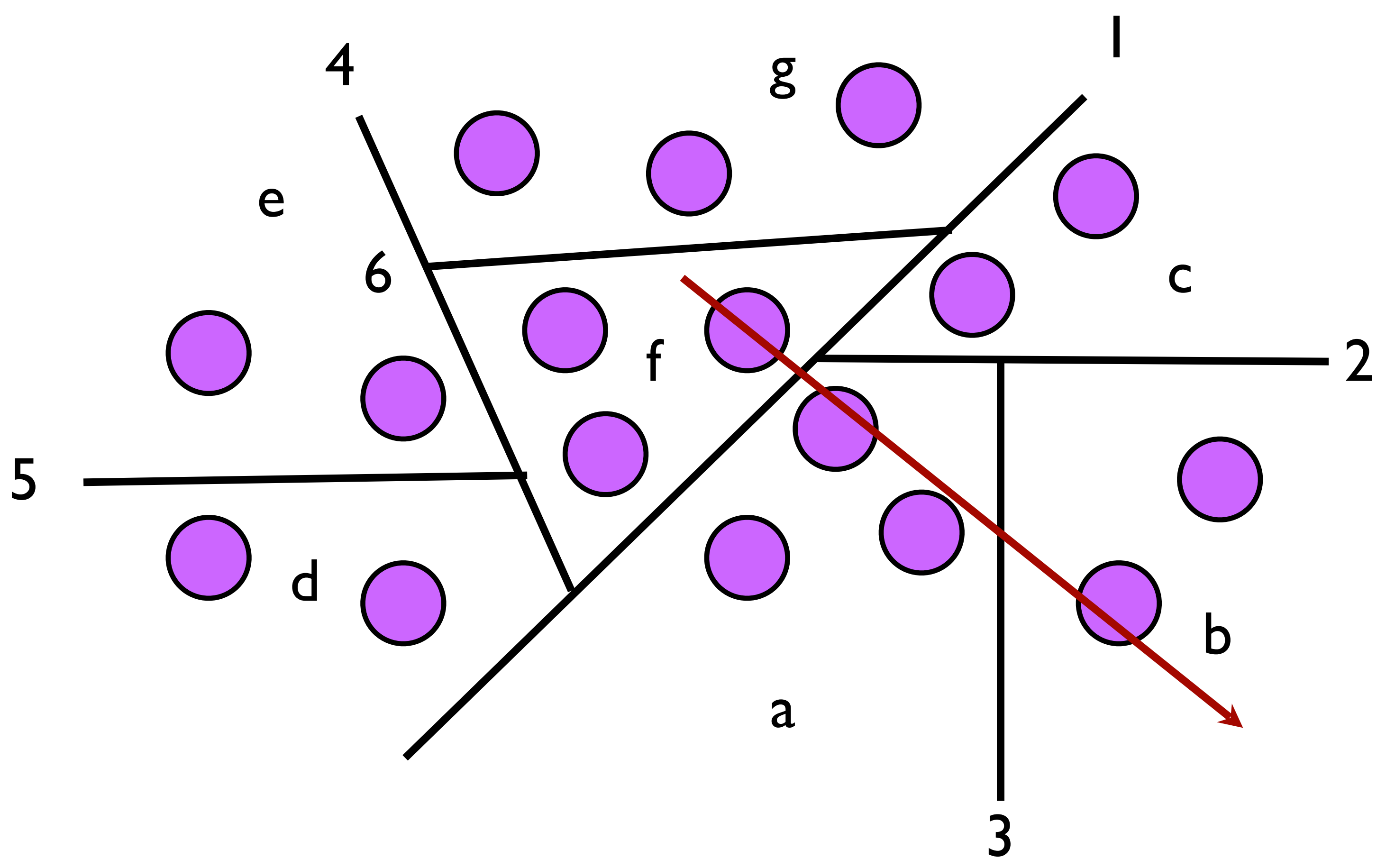
Another approach to optimisation is to build a BSP tree dividing the world into cells, where each cell contains a small number of objects.

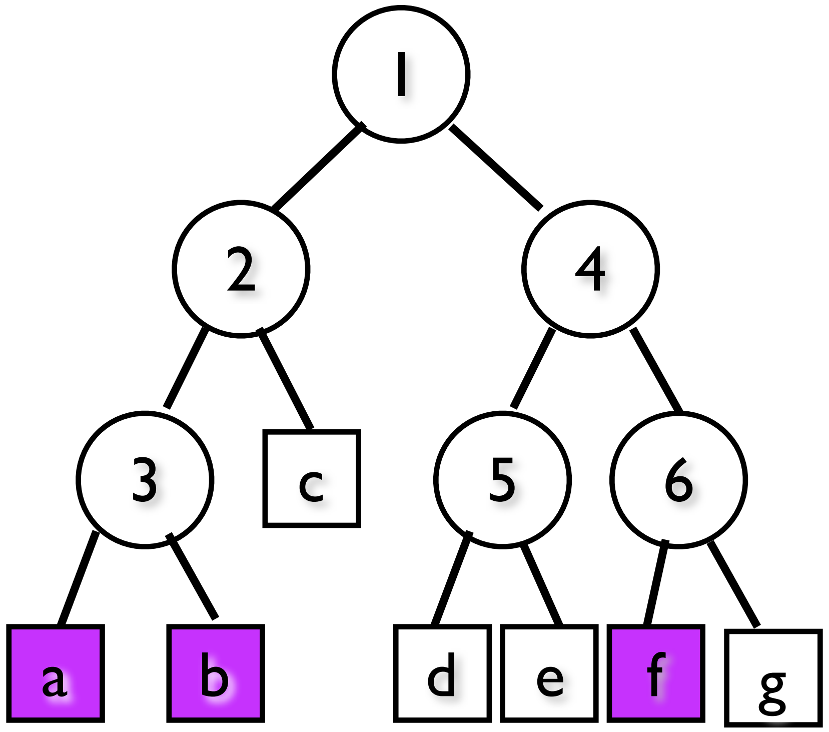




Traversing the tree

In this case we do not want to traverse the entire tree. We only want to visit the leaves the ray passes through.

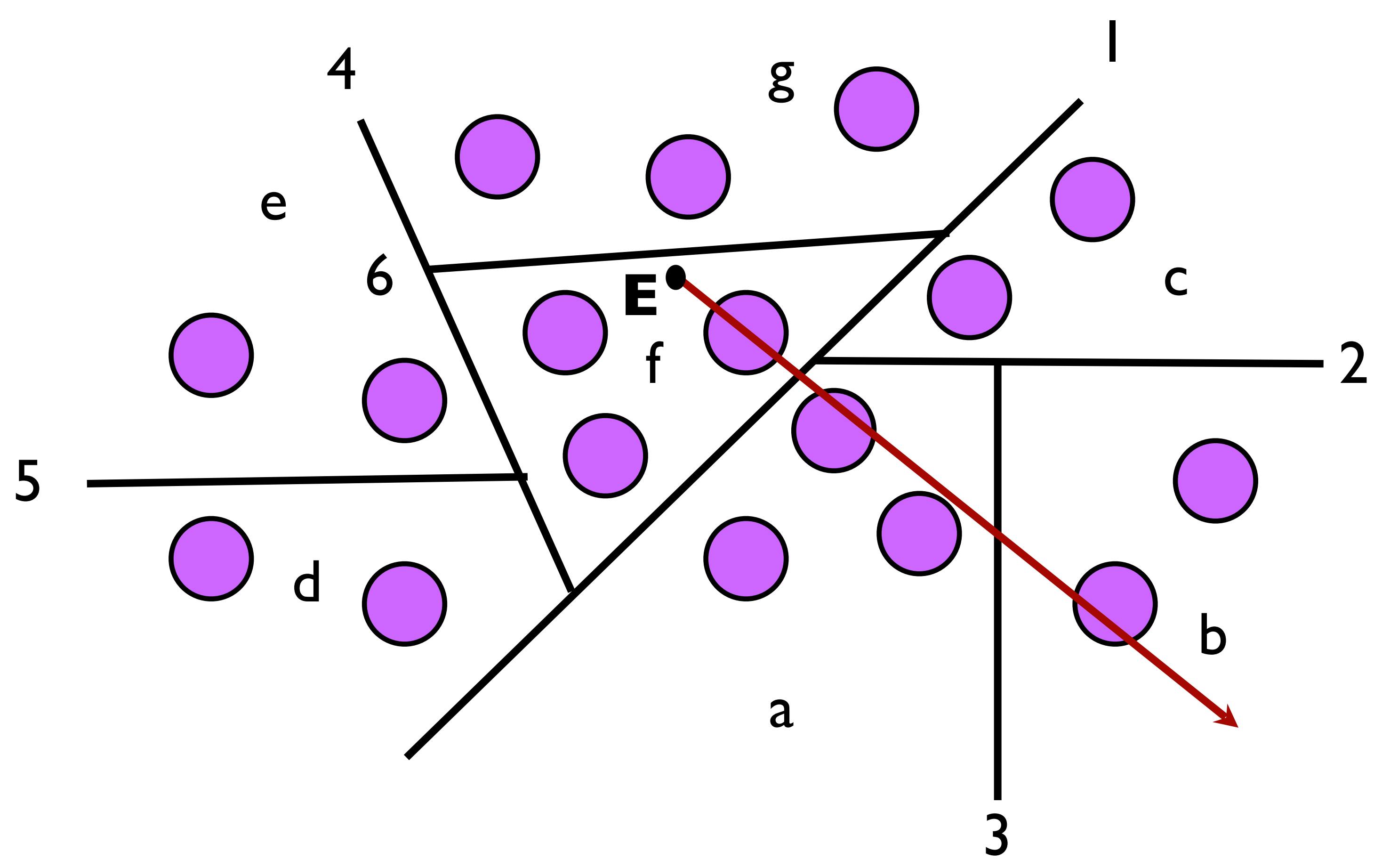


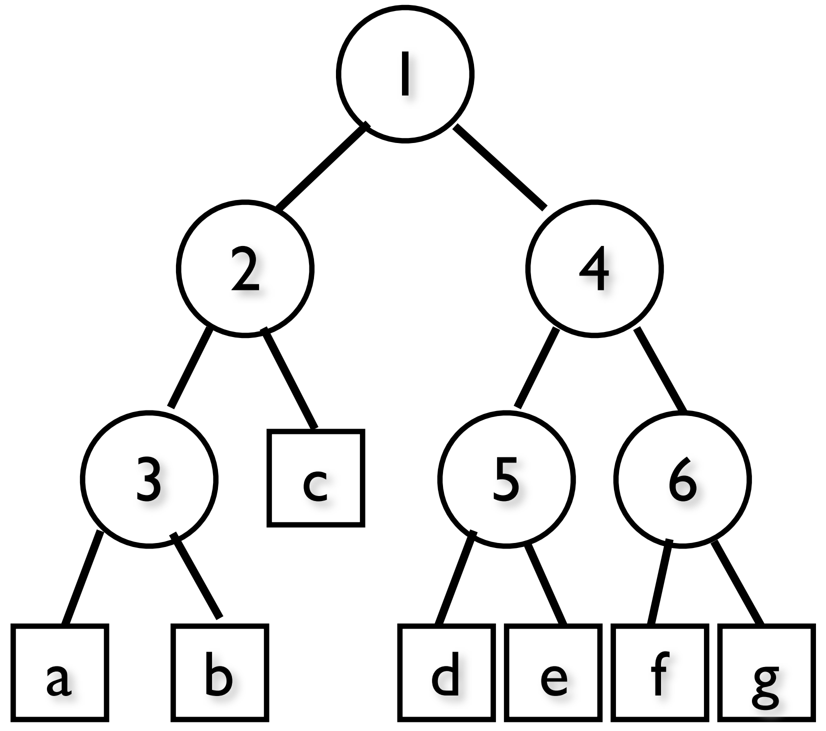


Traversal algorithm

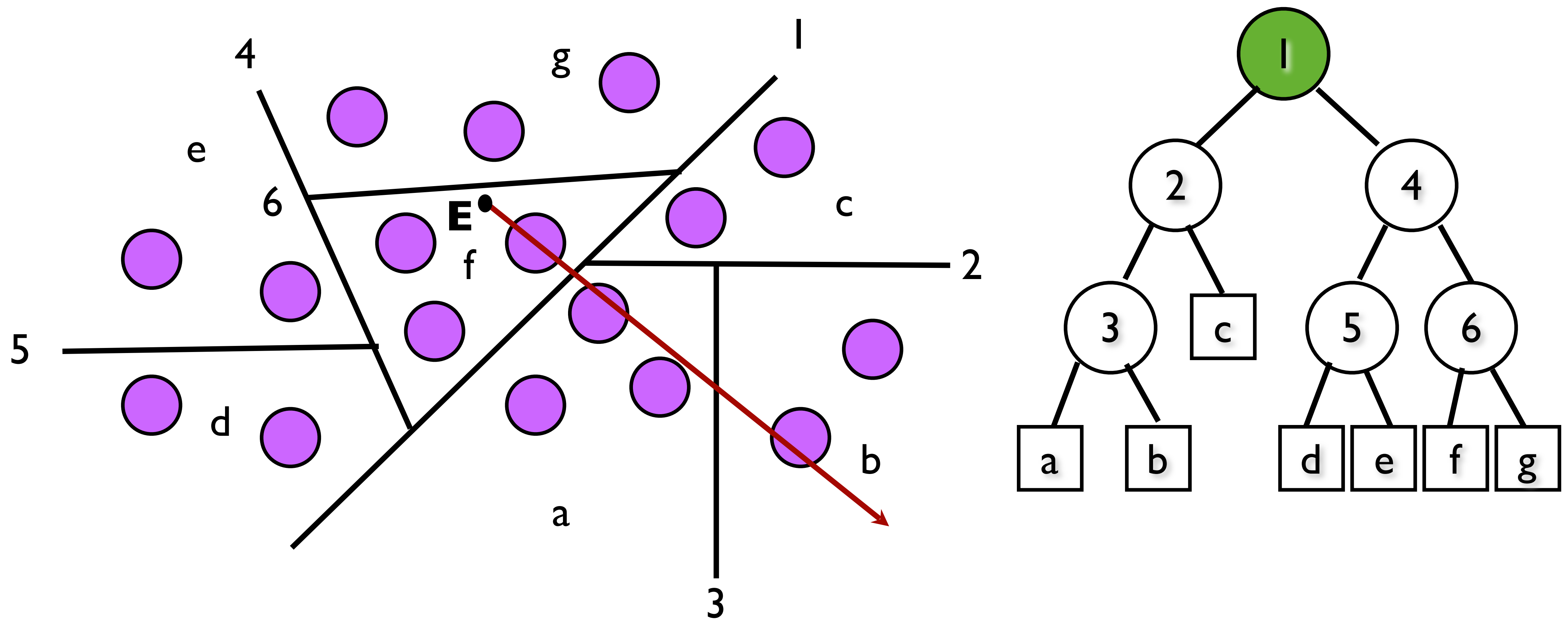
```
visit(E, v, node): (E eye)

  if (node is leaf):
     intersect ray with objs in leaf
  else:
     if (E on left):
         visit(E, v, left)
         other = right;
     else:
         visit(E, v, right)
         other = left
     endif

     if (ray crosses boundary):
         E' = intersect(E, v, boundary)
         visit(E', v, other)
     endif
  endif
```

Traversing the tree

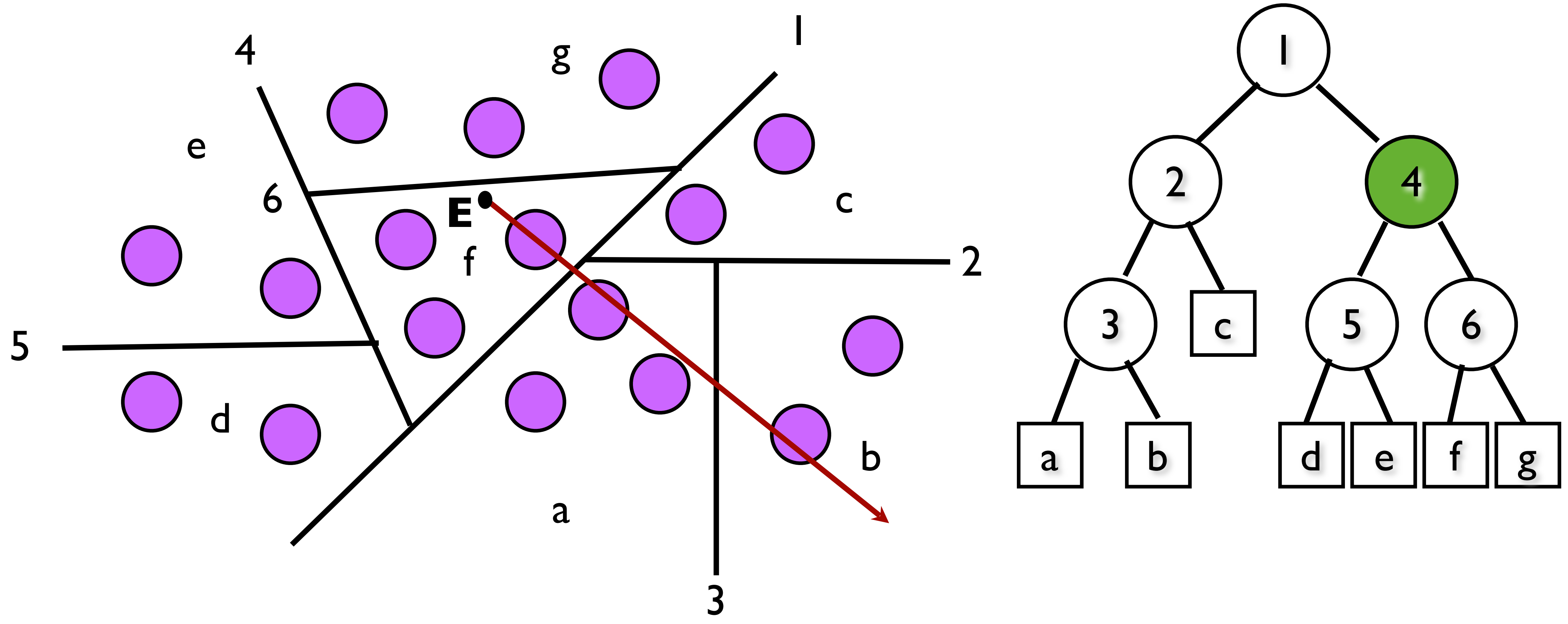




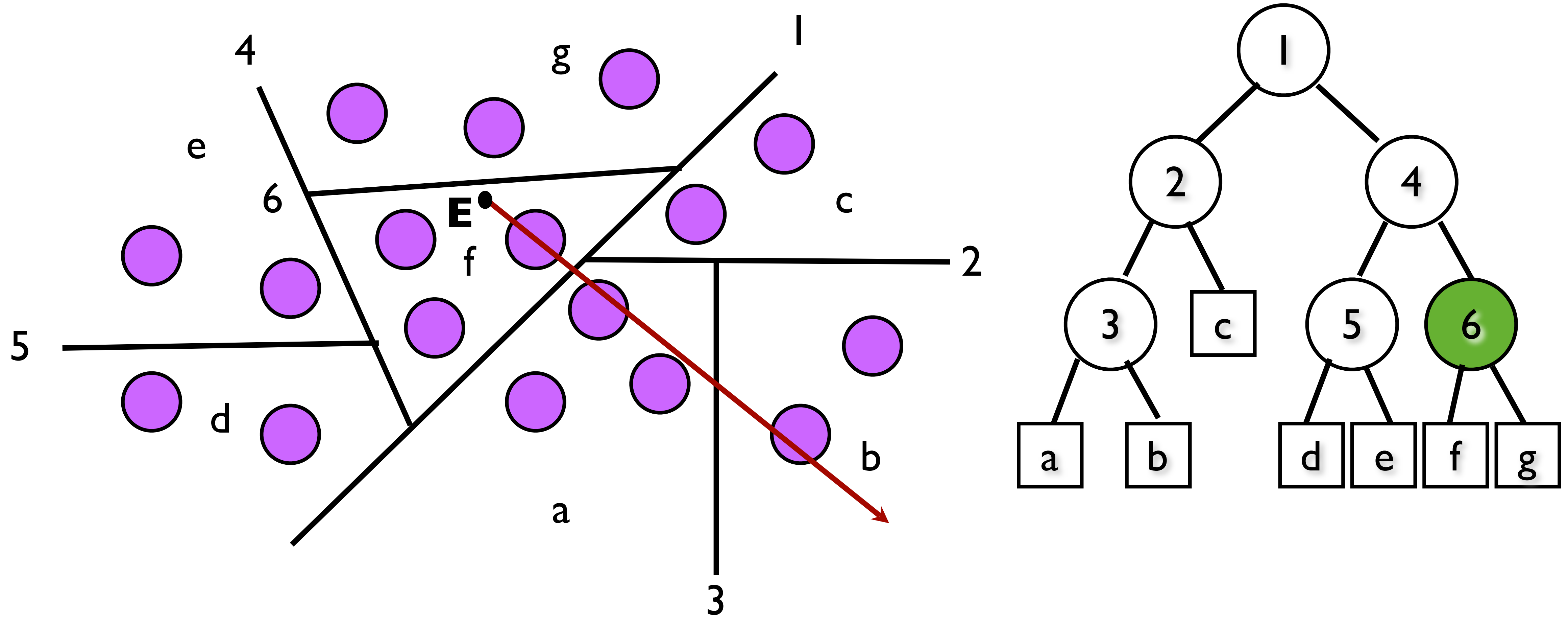
Traversing the tree



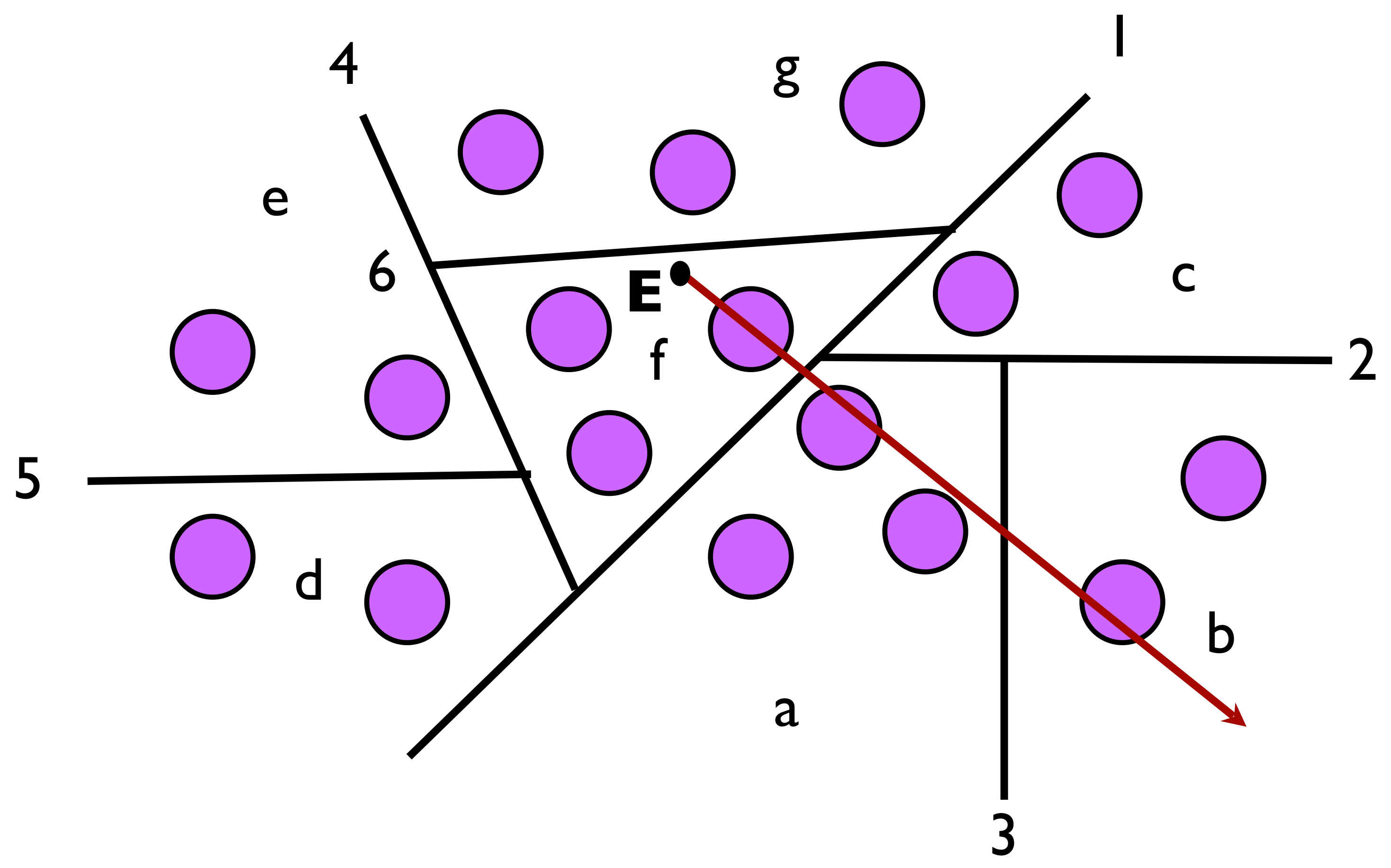
Traversing the tree

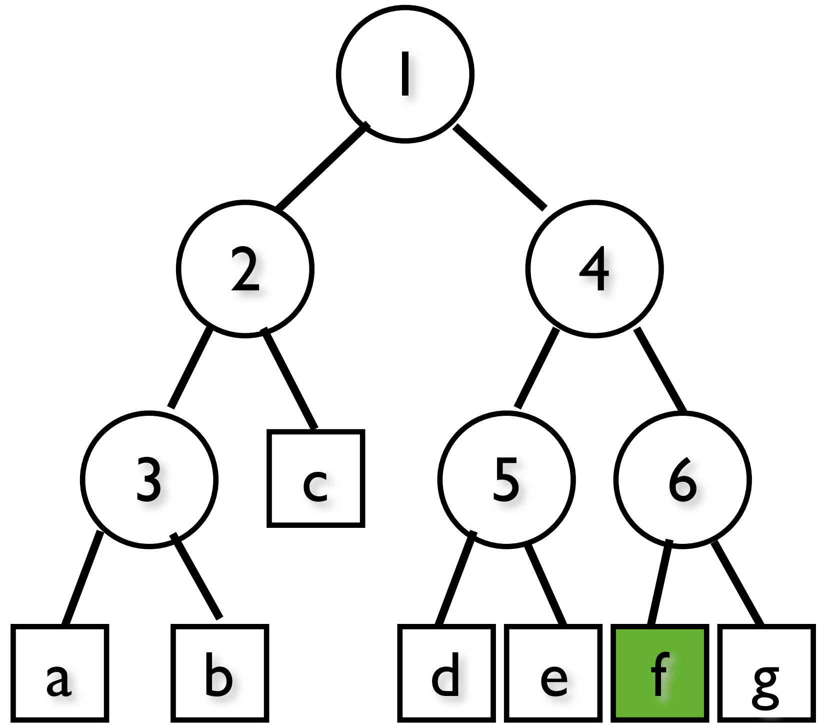


Traversing the tree

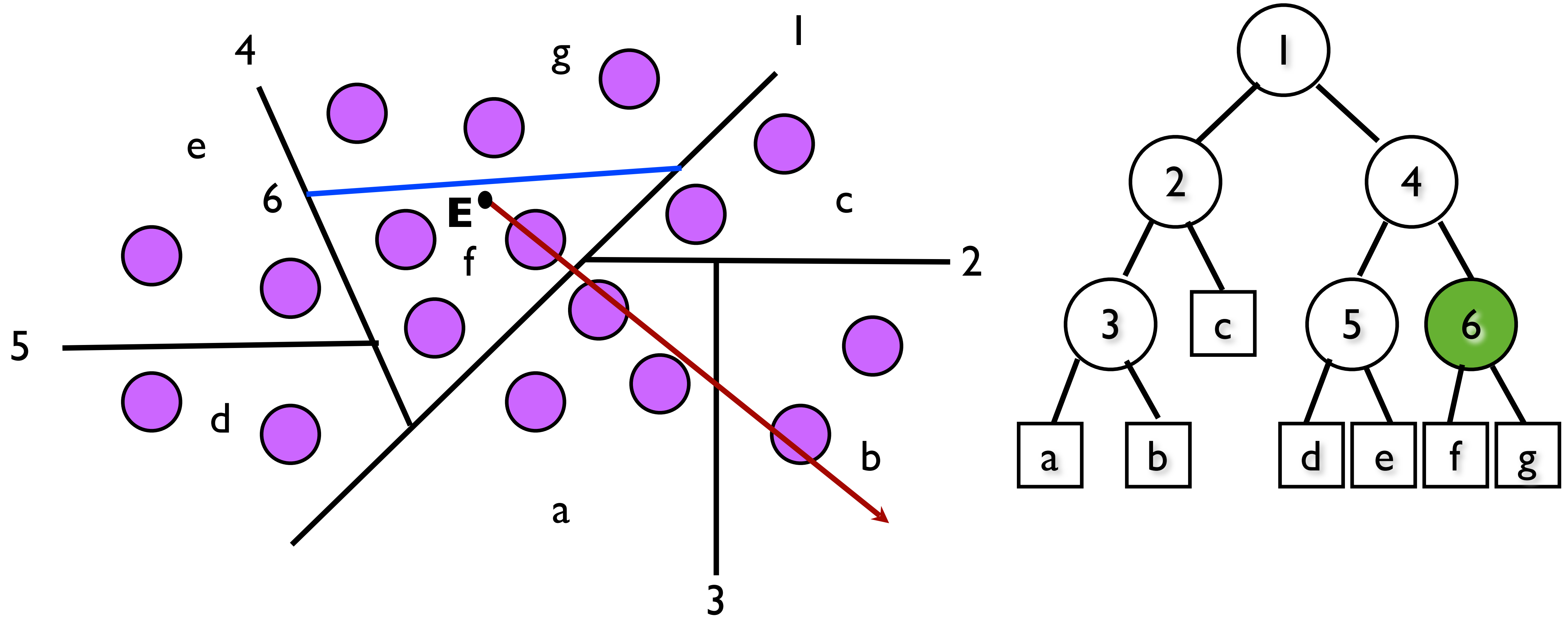


Traversing the tree

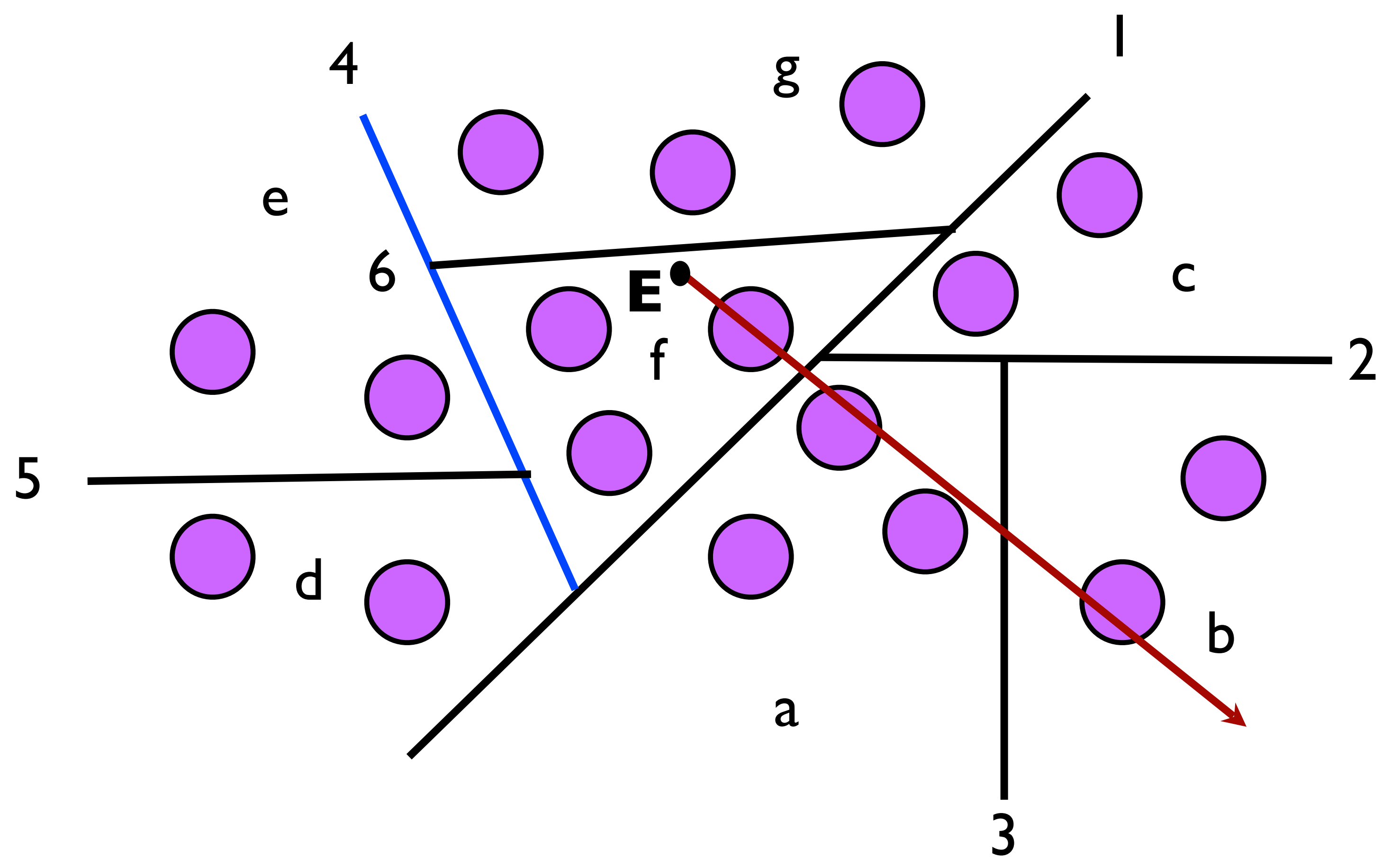


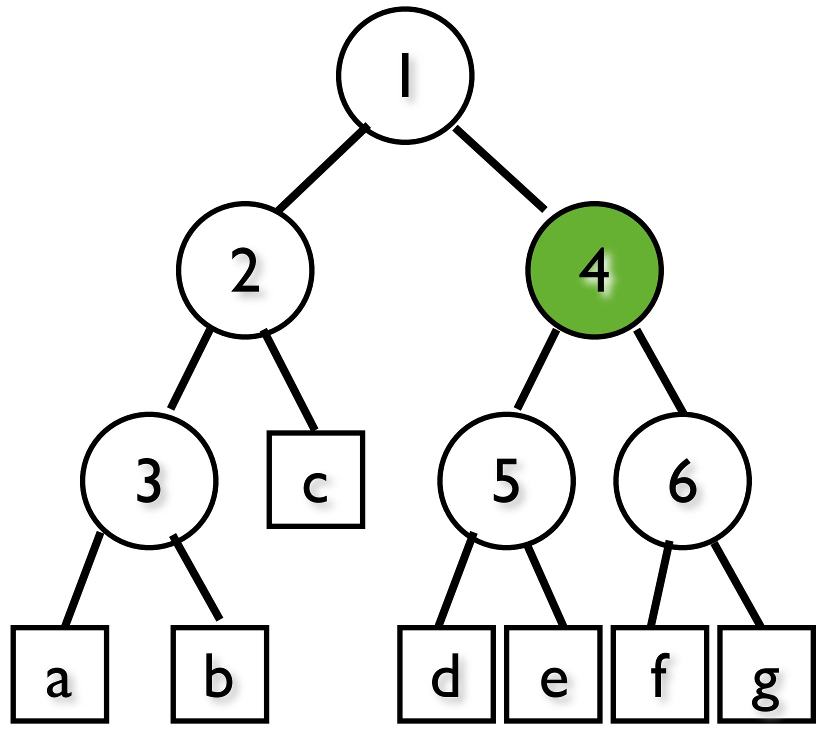


Traversing the tree

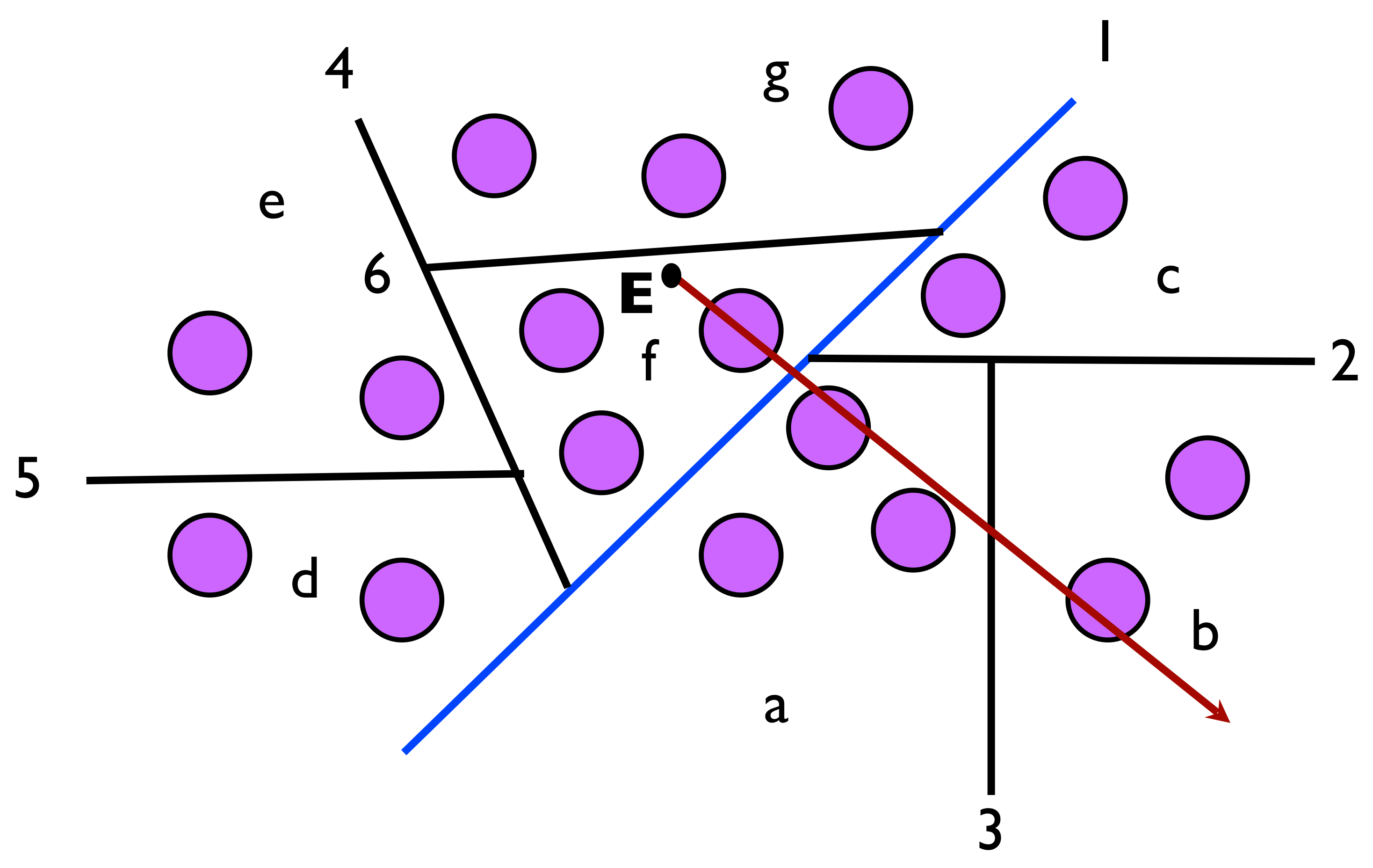


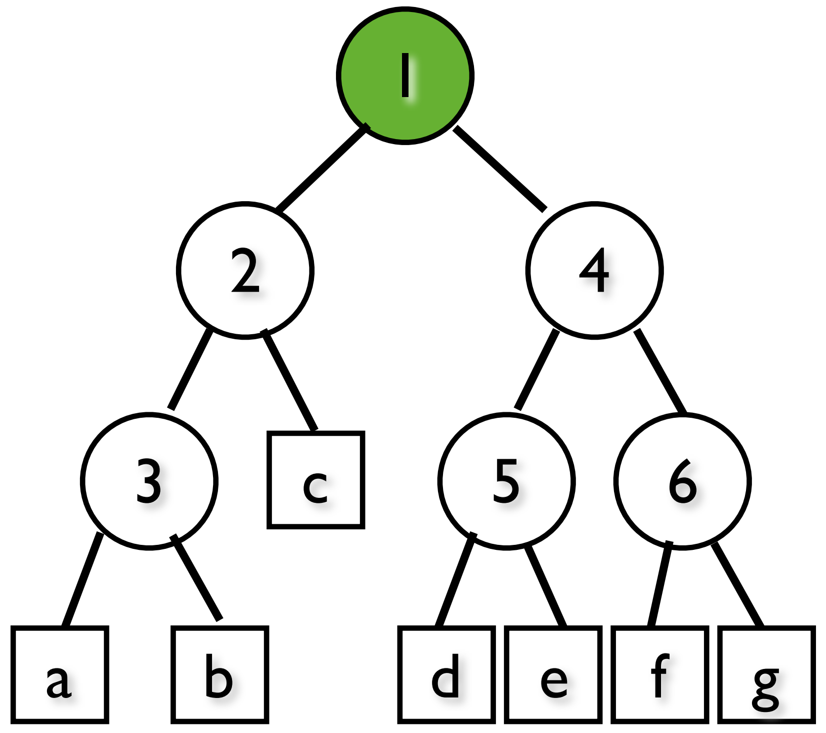
Traversing the tree



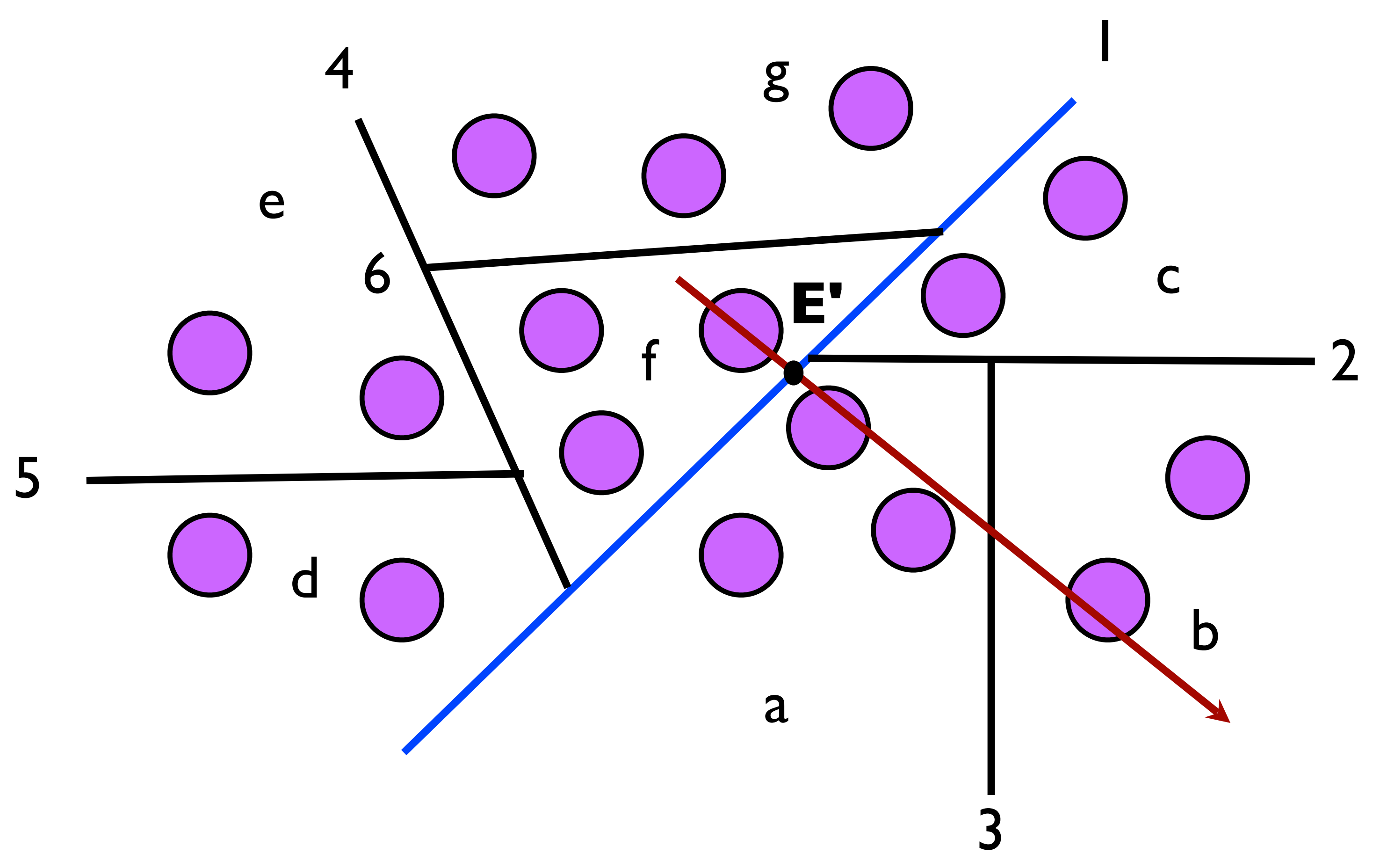


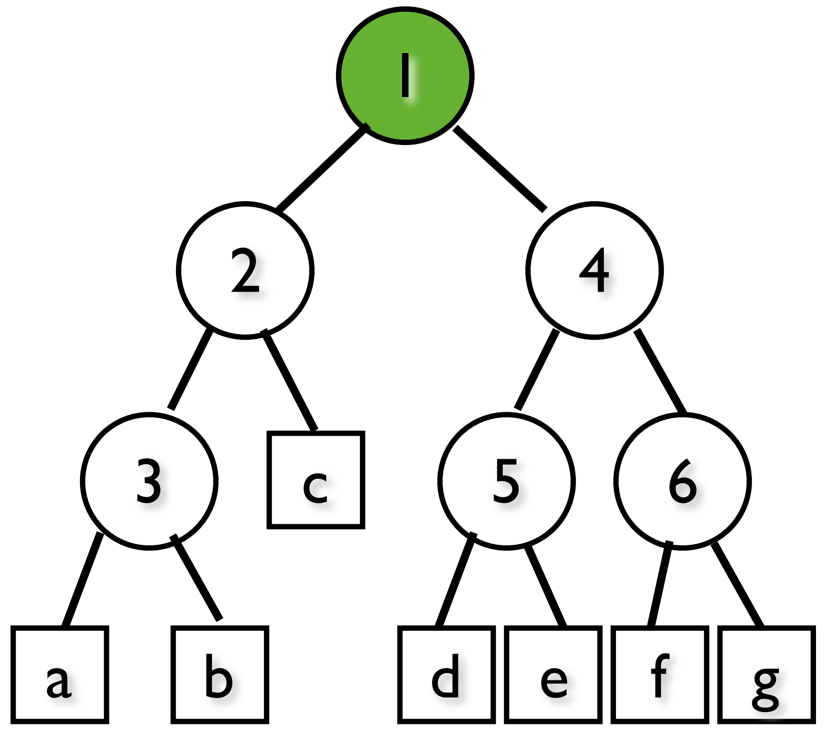
Traversing the tree



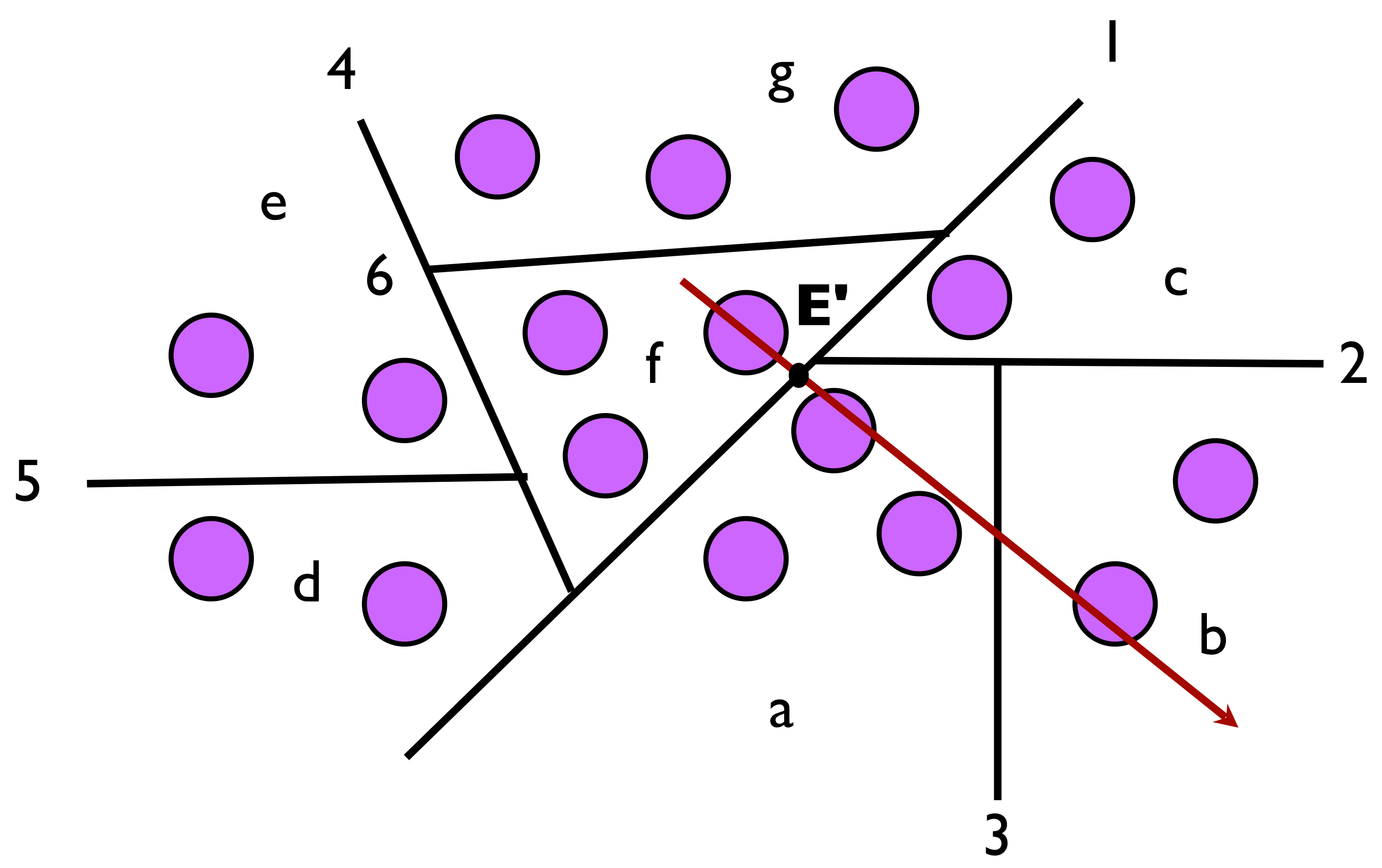


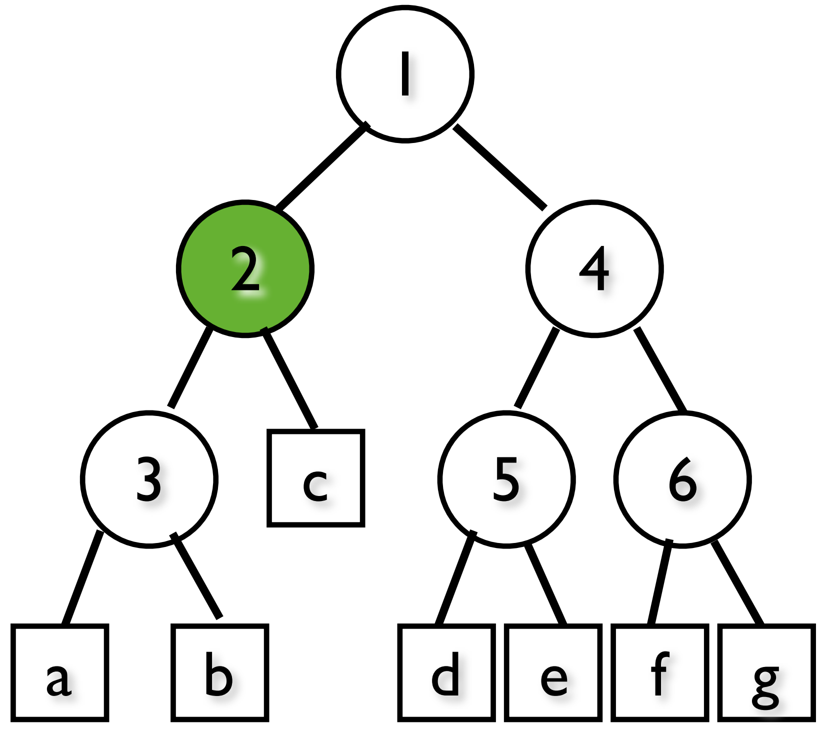
Traversing the tree



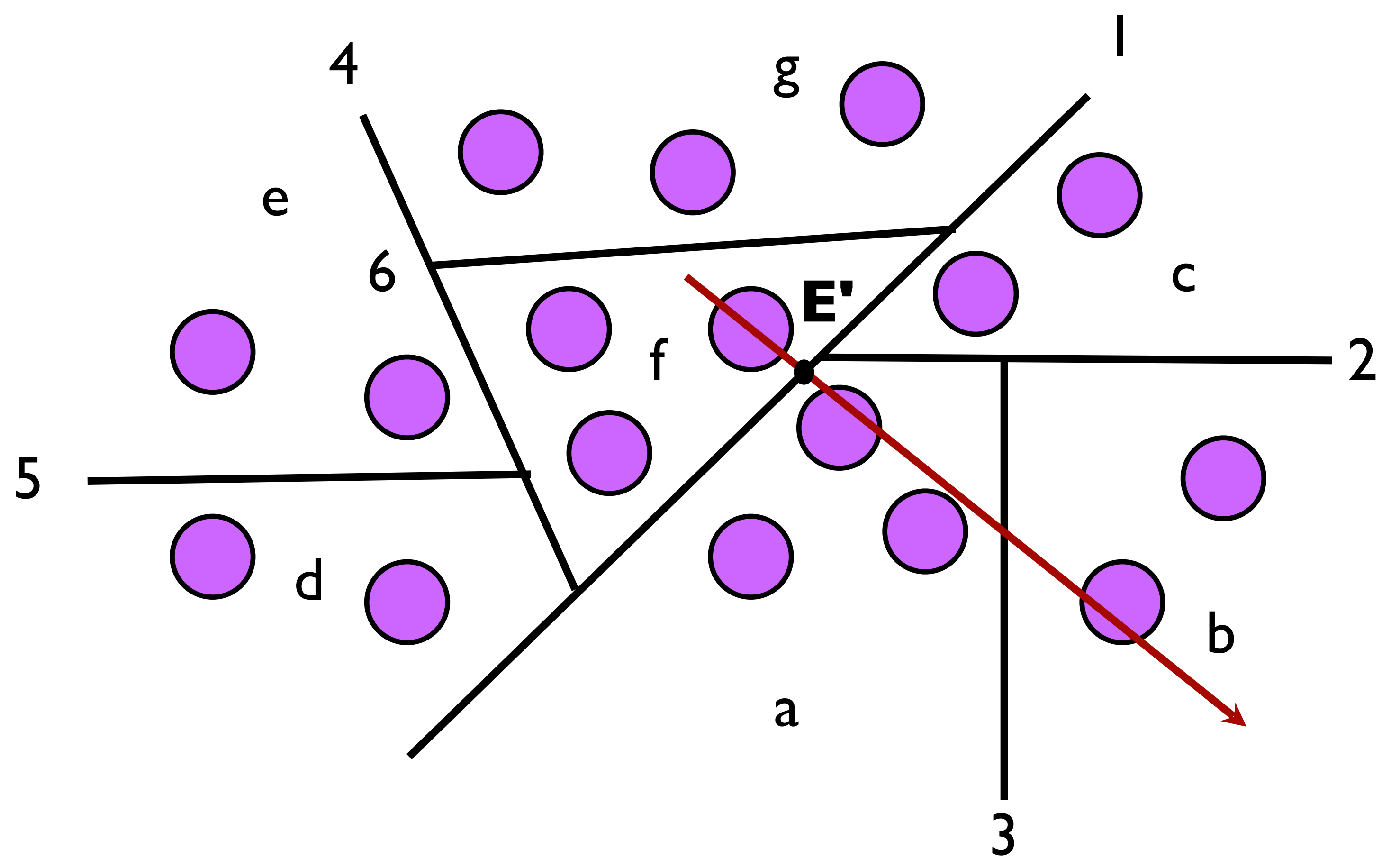


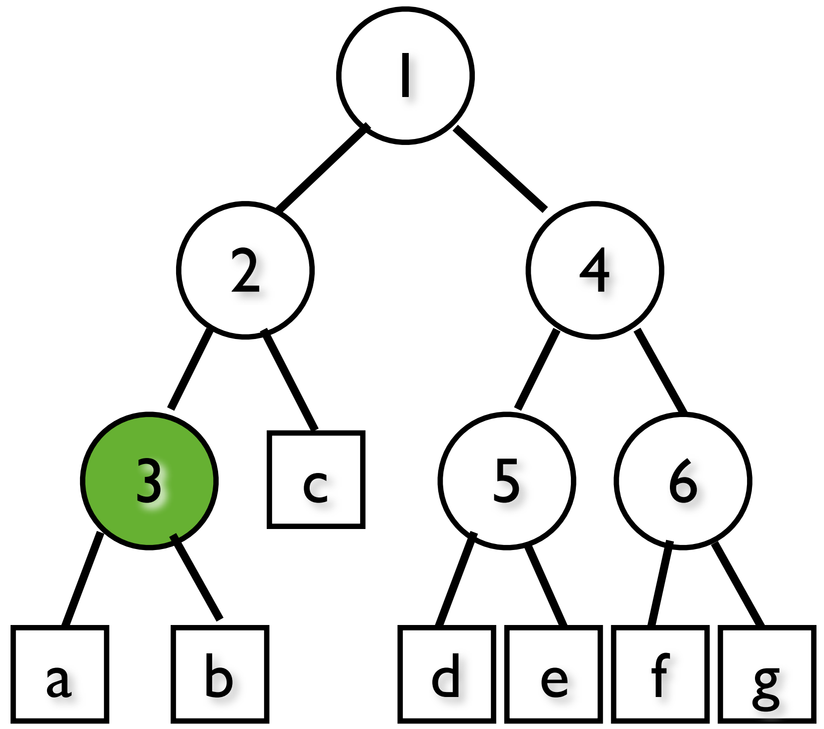
Traversing the tree



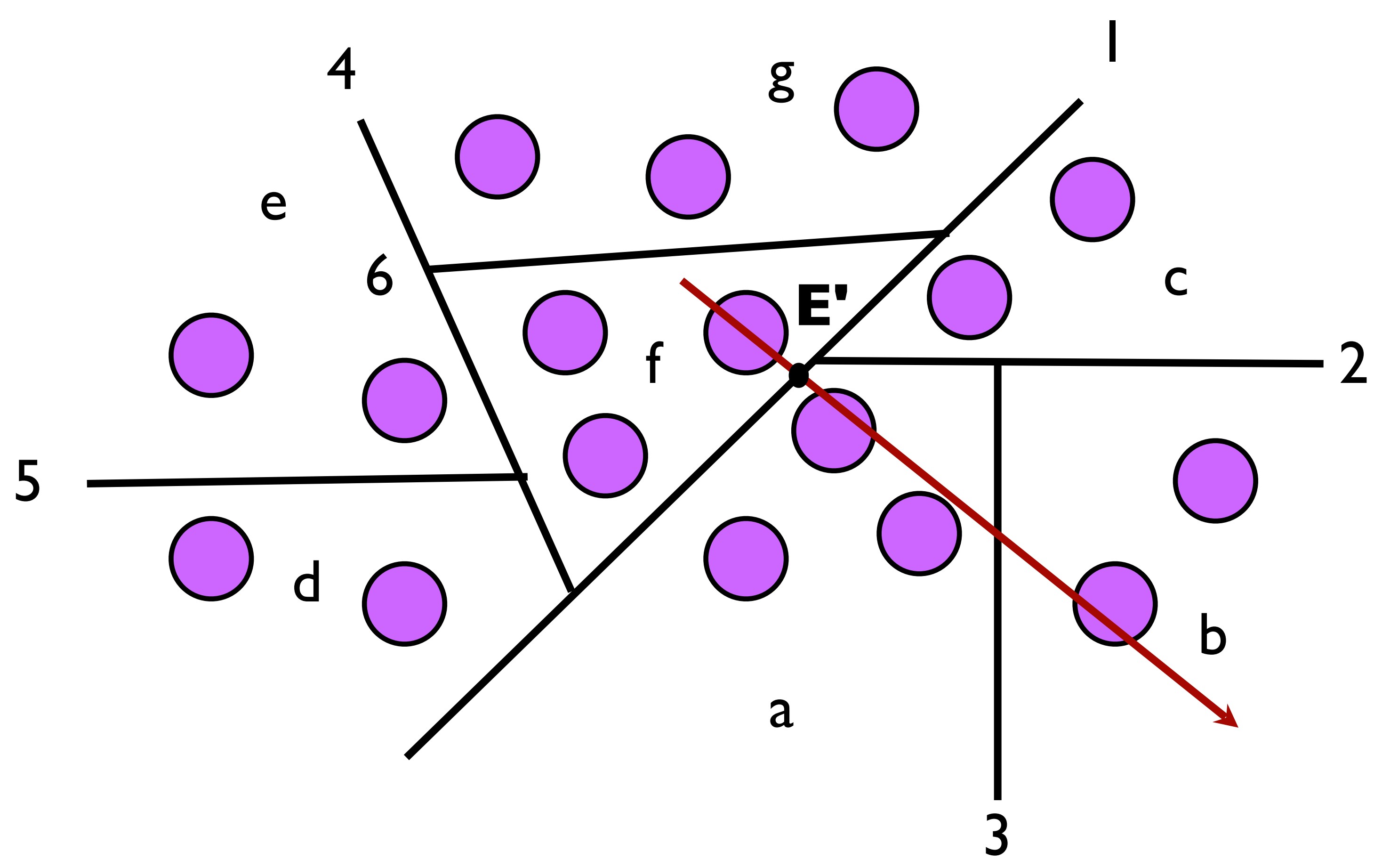


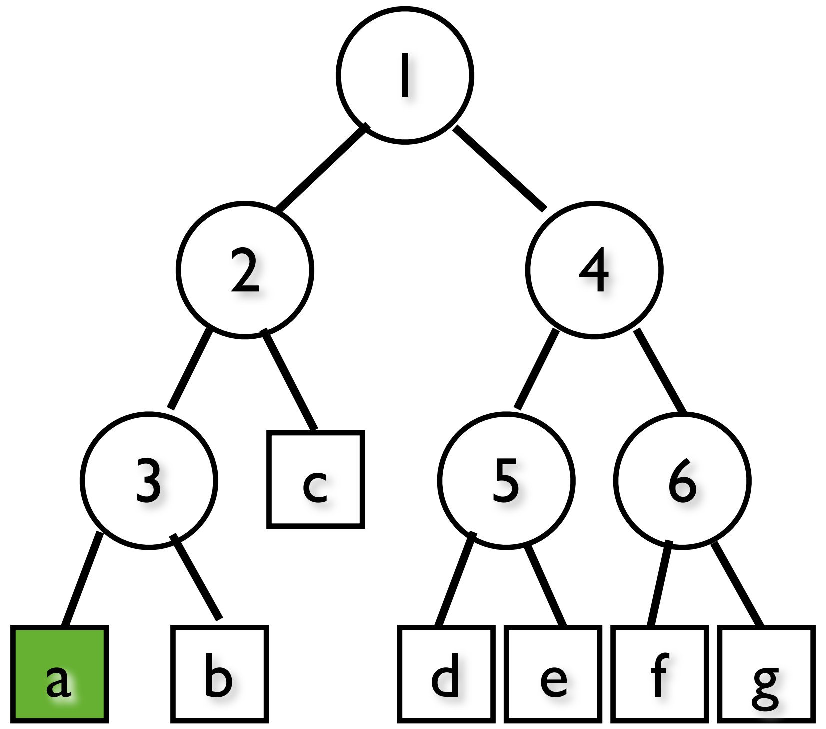
Traversing the tree



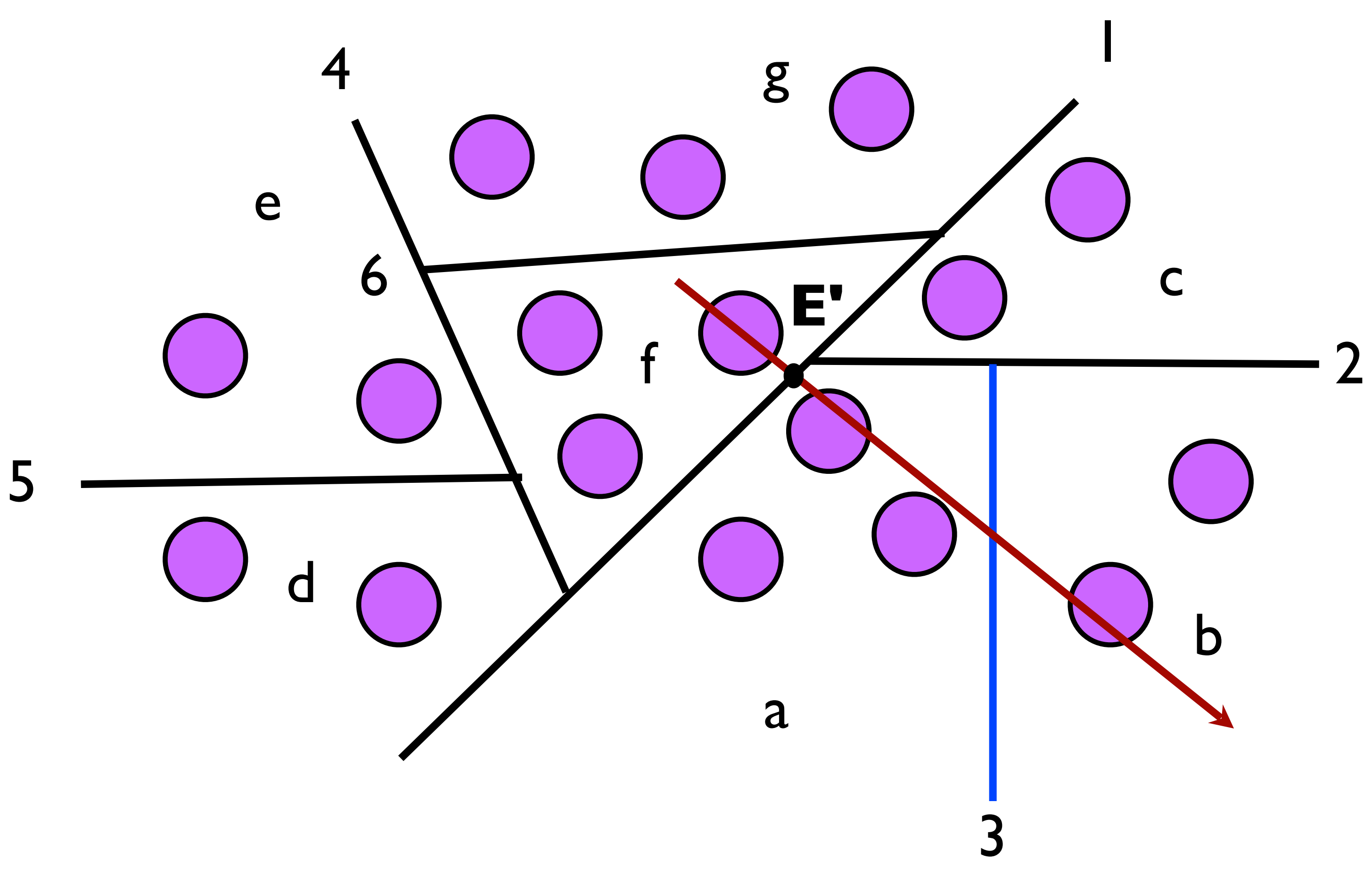


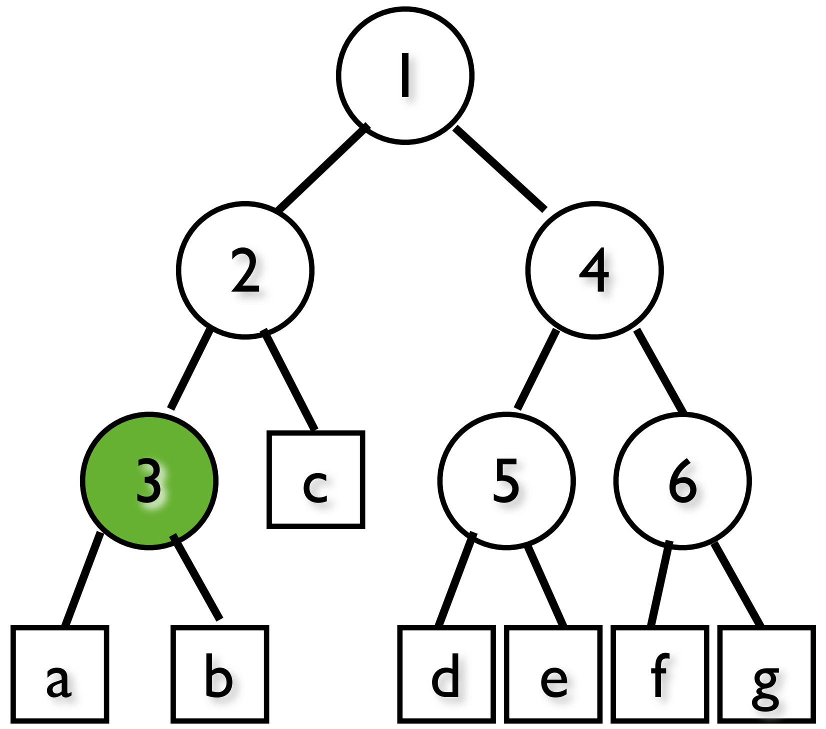
Traversing the tree



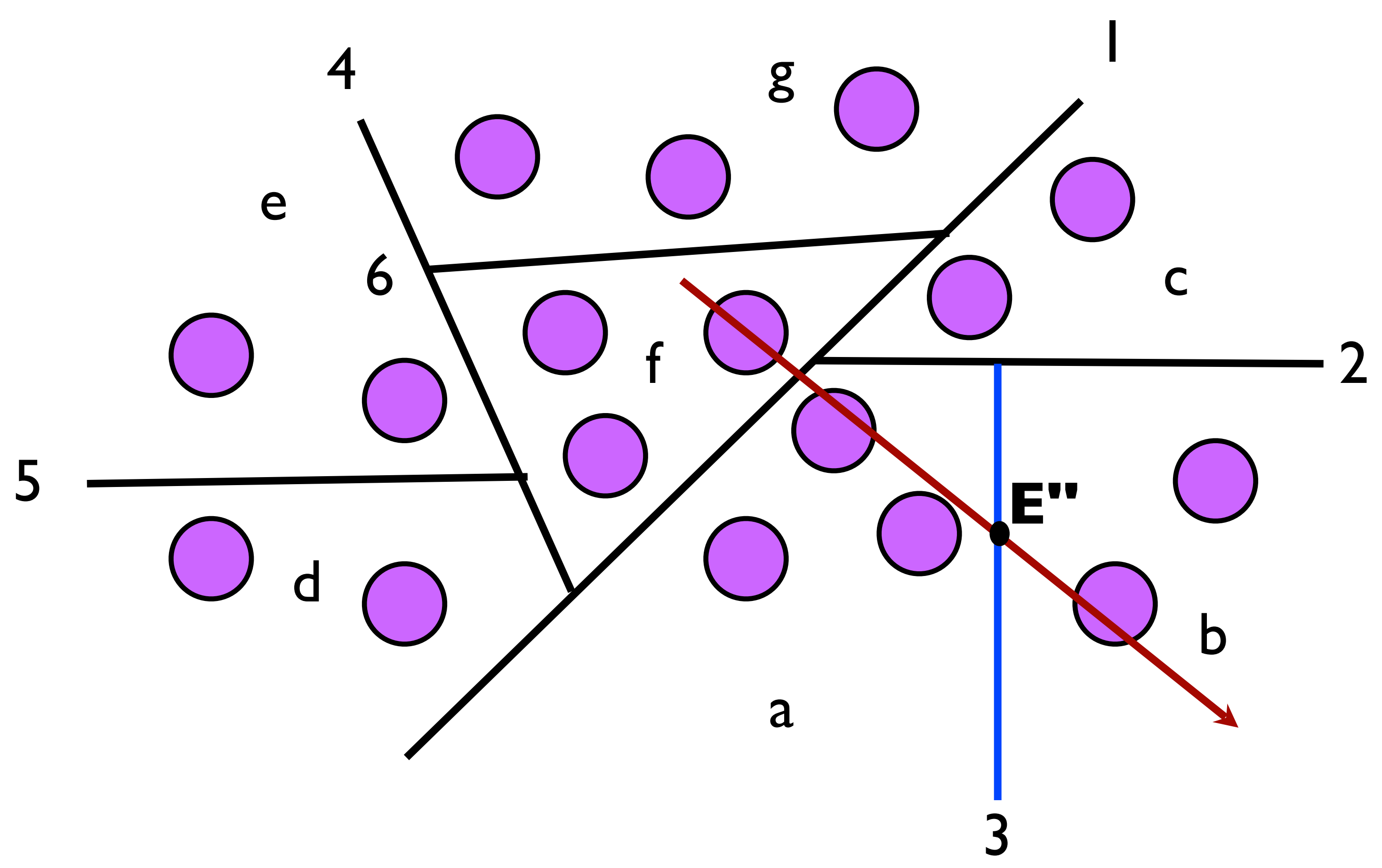


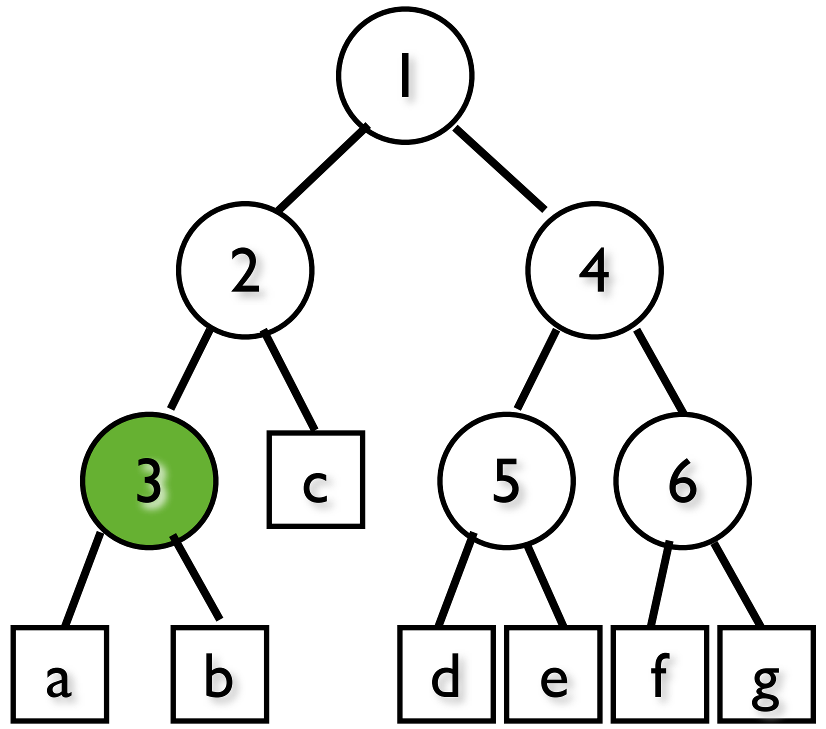
Traversing the tree



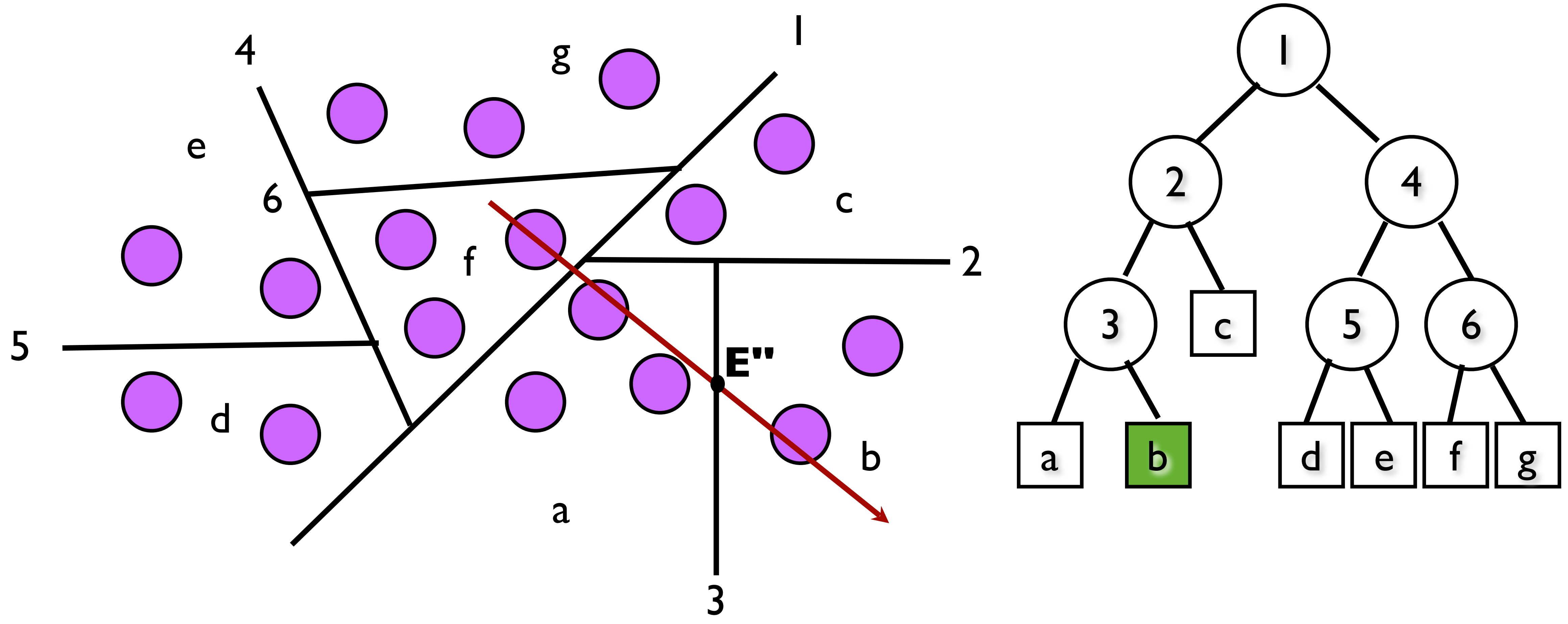


Traversing the tree

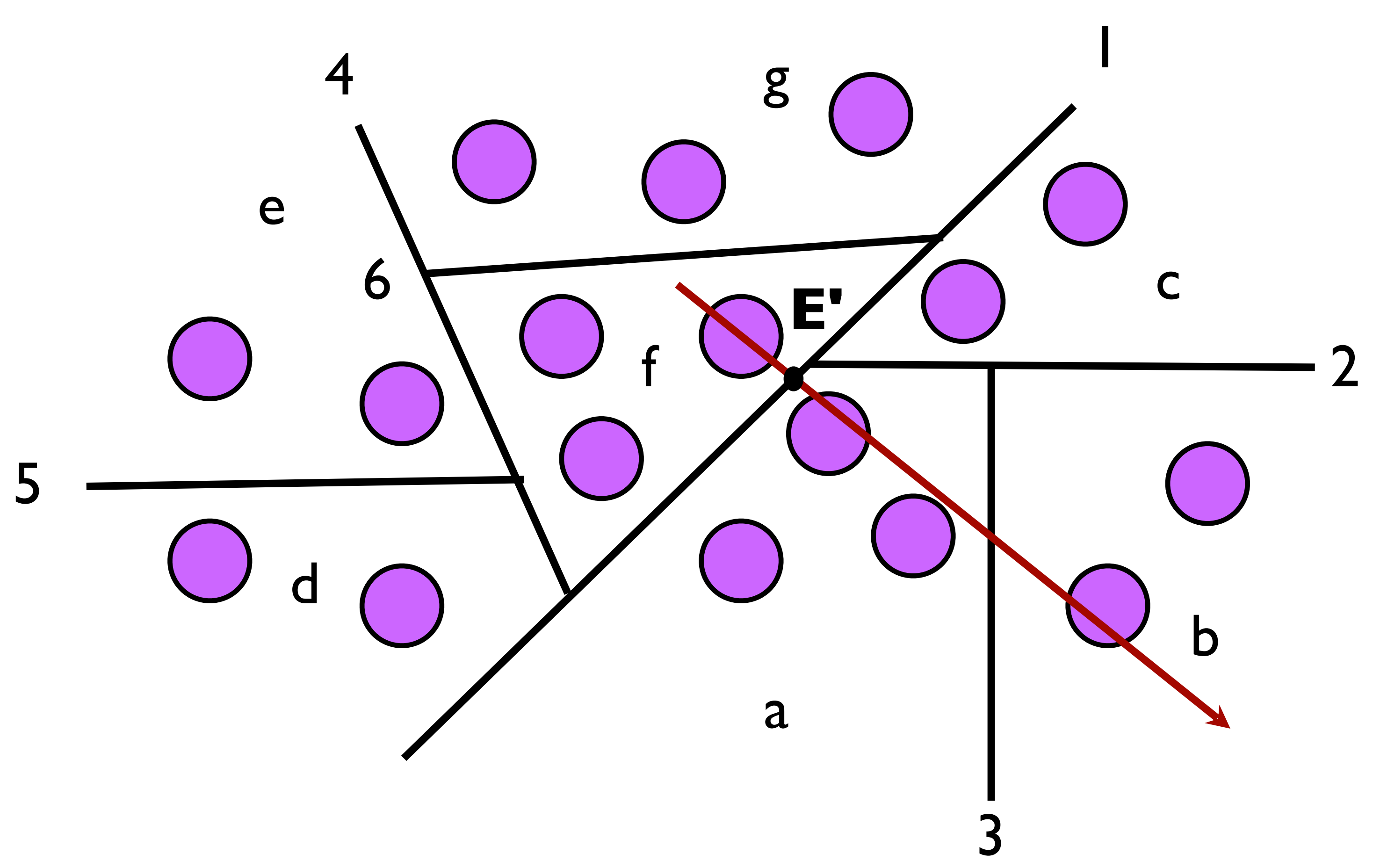


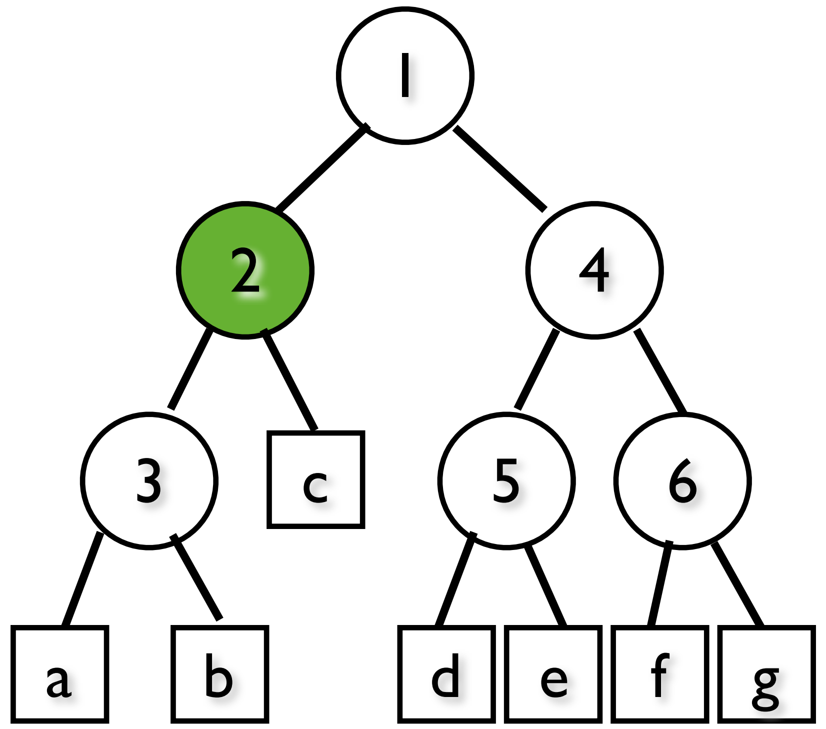


Traversing the tree



Traversing the tree





Shadows

We can add shadows very simply.

At each hit point we cast a new ray towards each light source. These rays are called shadow feelers.

If a shadow feeler intersects an object before it reaches the source, then omit that source from the illumination equation for the point.

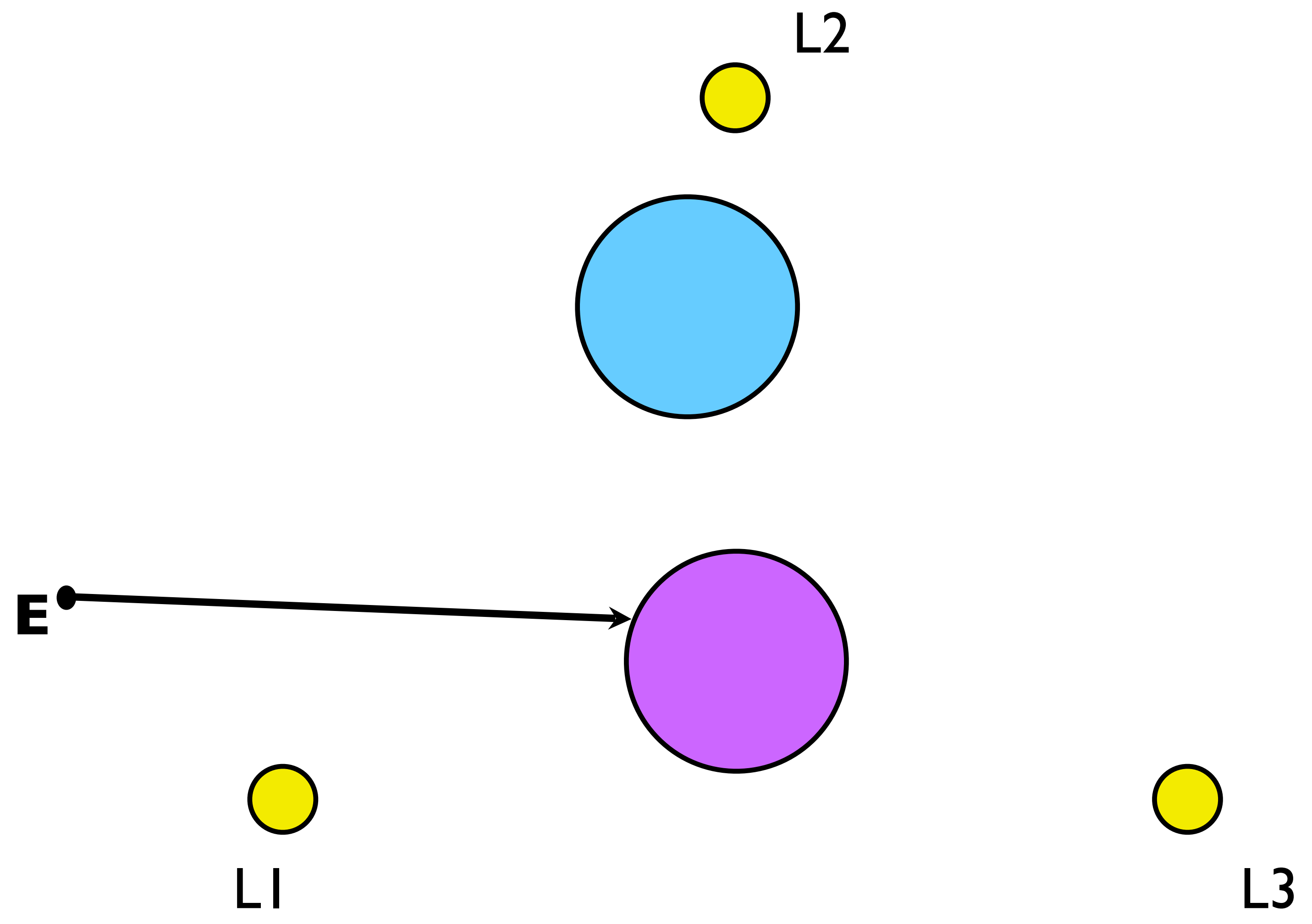
Self-shadows

We need to take care when the shadow is cast by the hit object itself.

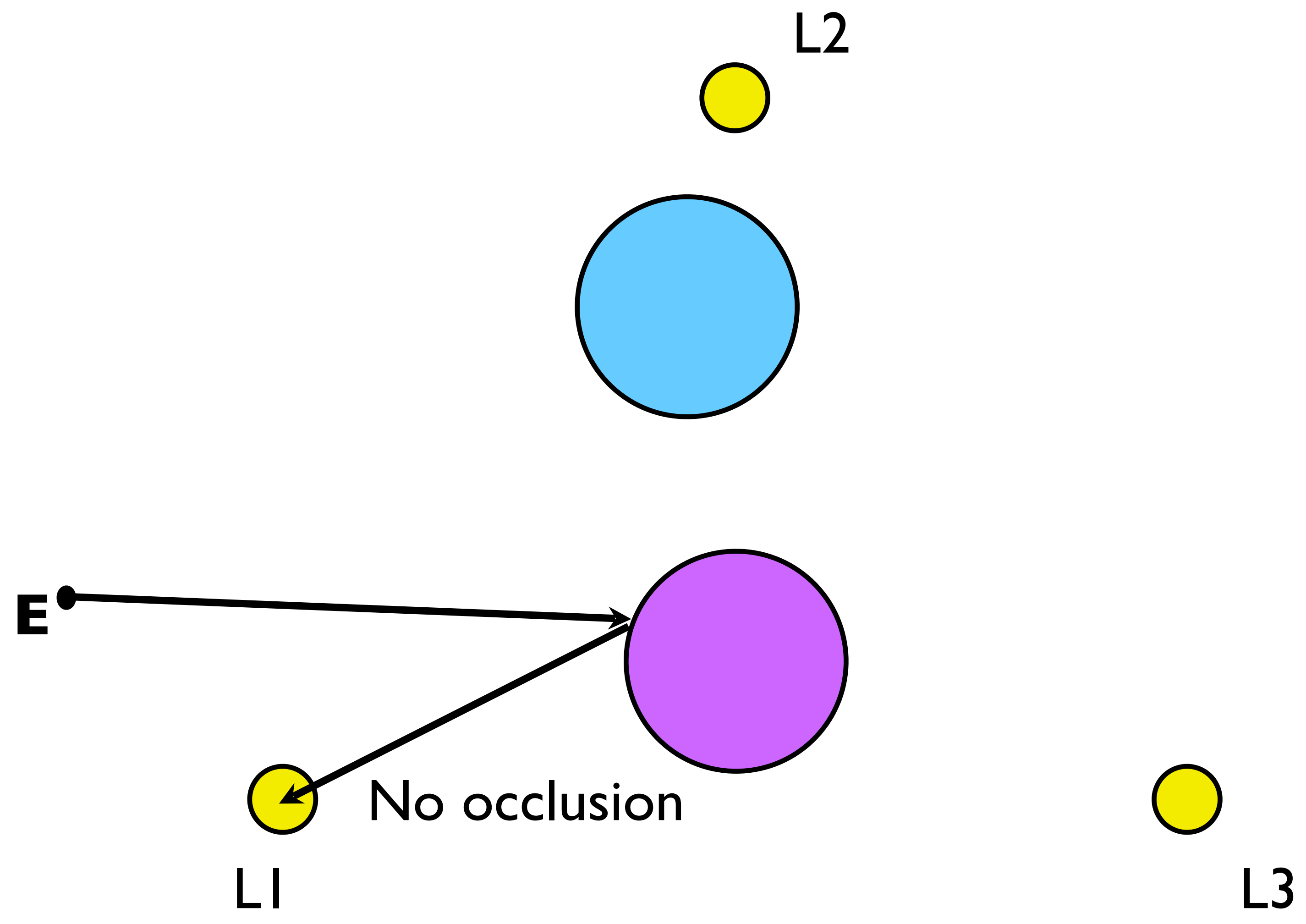
The shadow feeler will always intersect the hit object at time t=0.

This intersection is only relevant if the light is on the opposite side of the object.

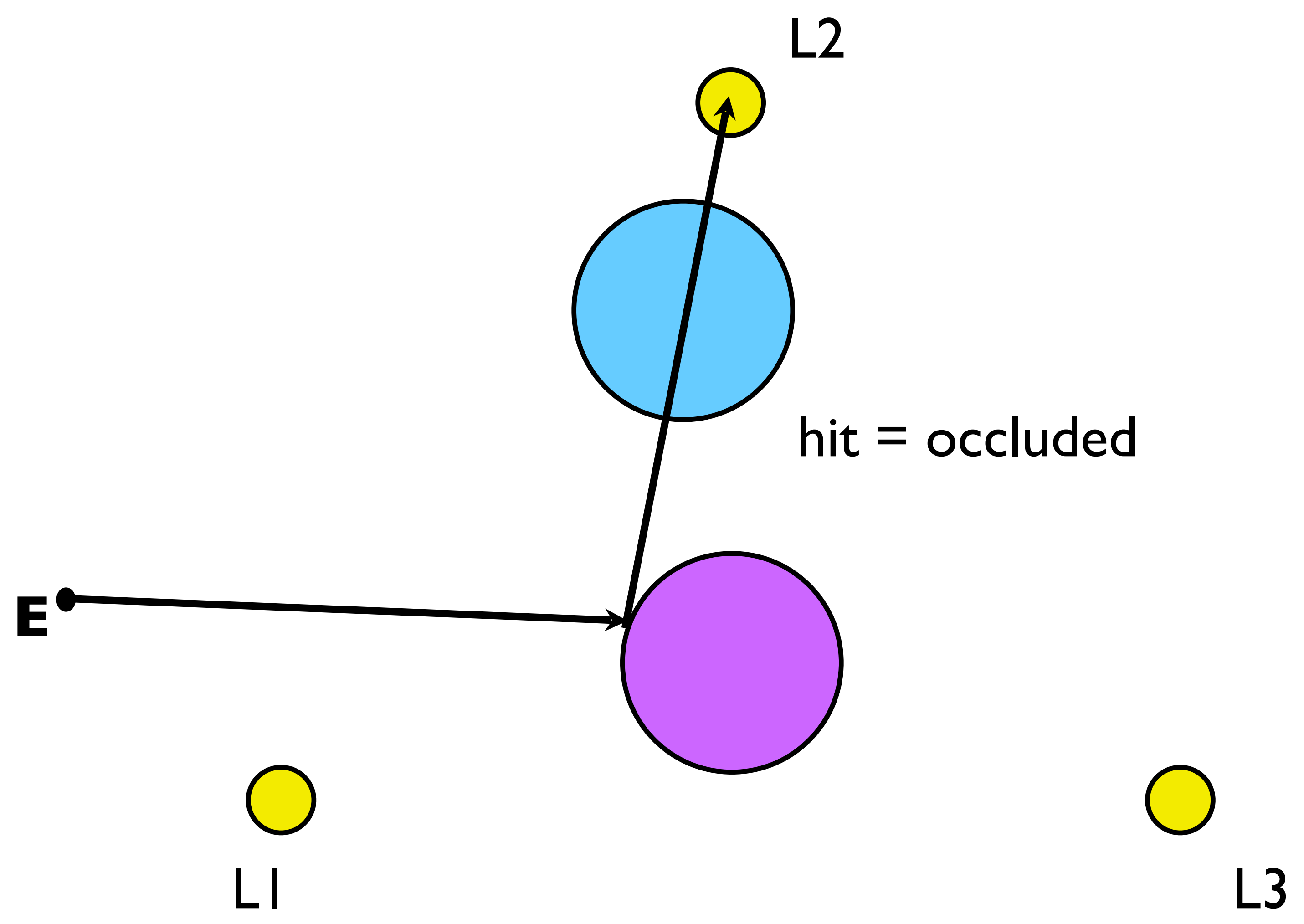
Example



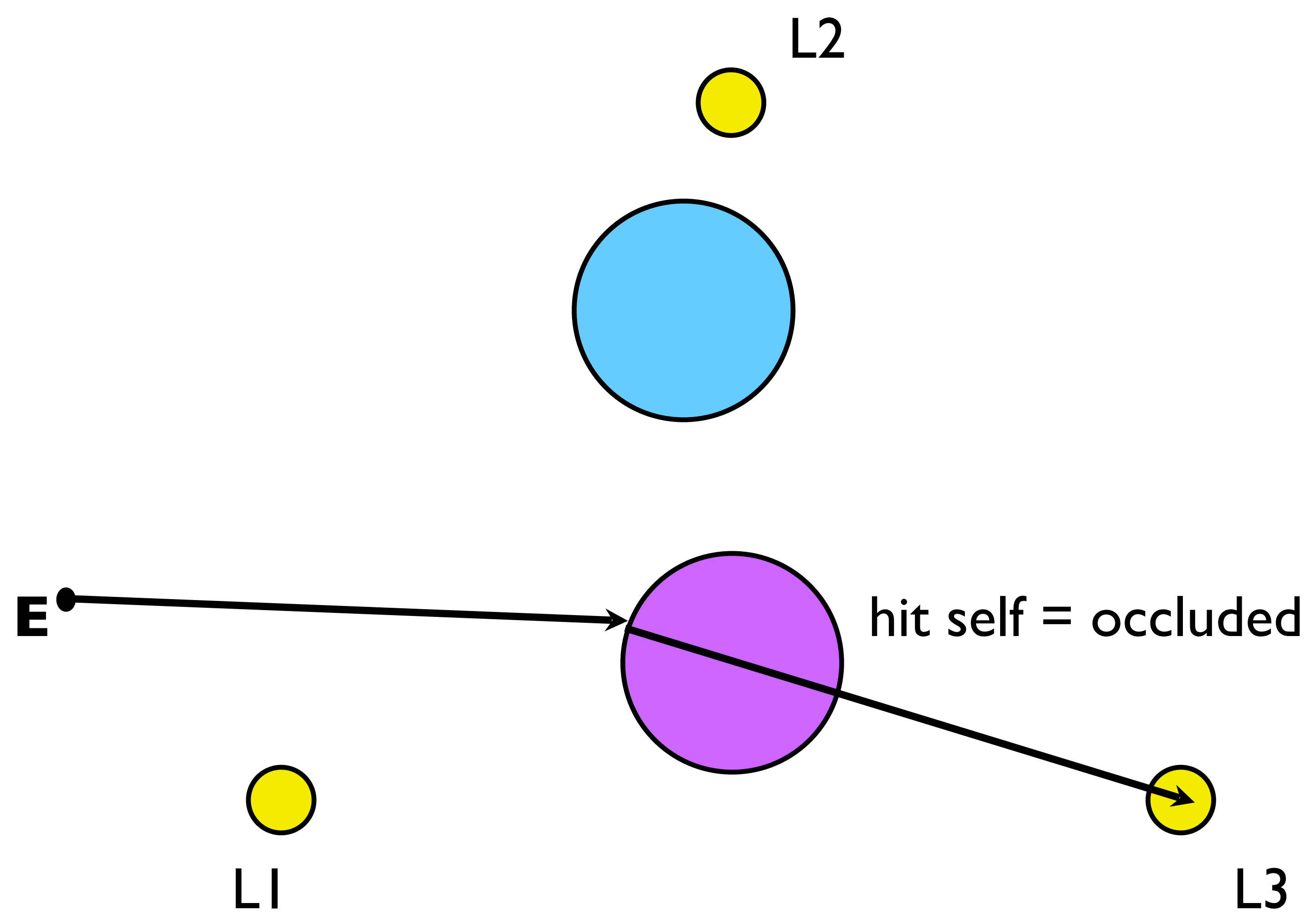
Example



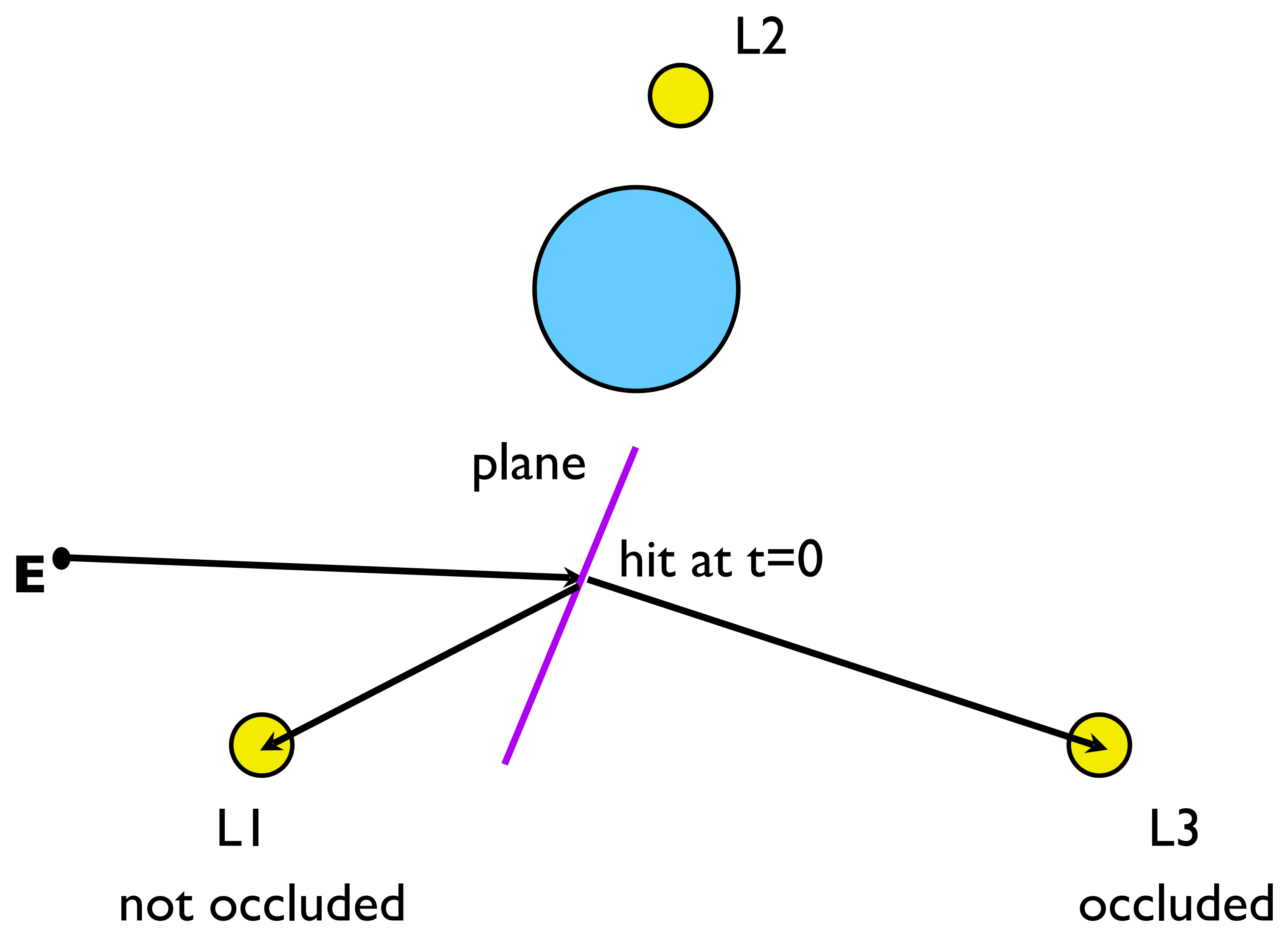
Example



Example



Example

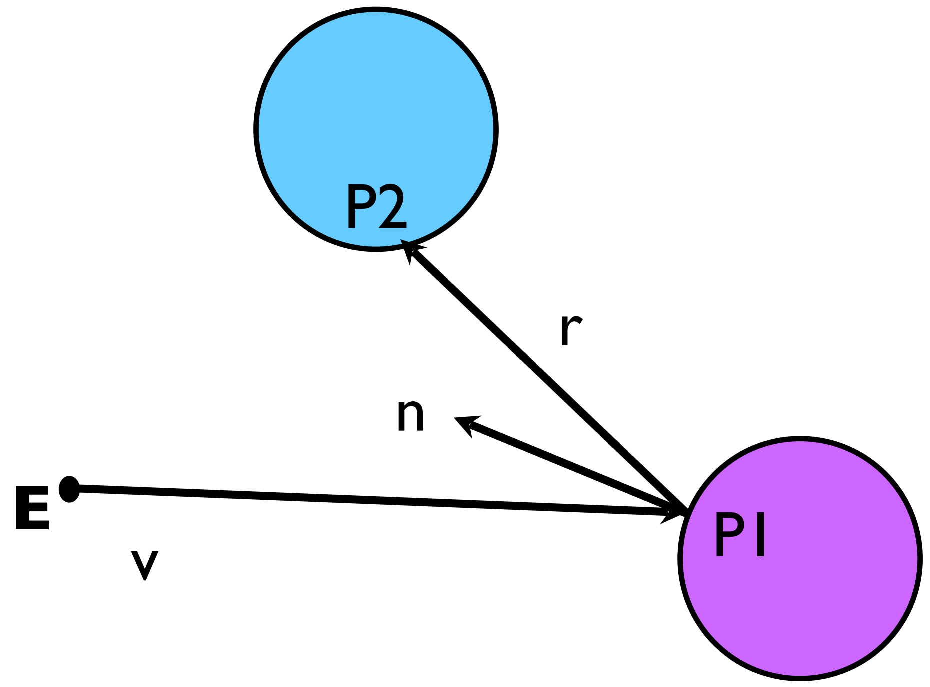


PseudoCode

```
Trace primary ray
if (hit is null)
  set (x,y) to background
else
  set (x,y) = ambient color
  Trace secondary ray to each light
    if not blocked from light
      (x,y) += contribution from light
```

Reflections

We can now implement realistic reflections by casting further reflected rays.

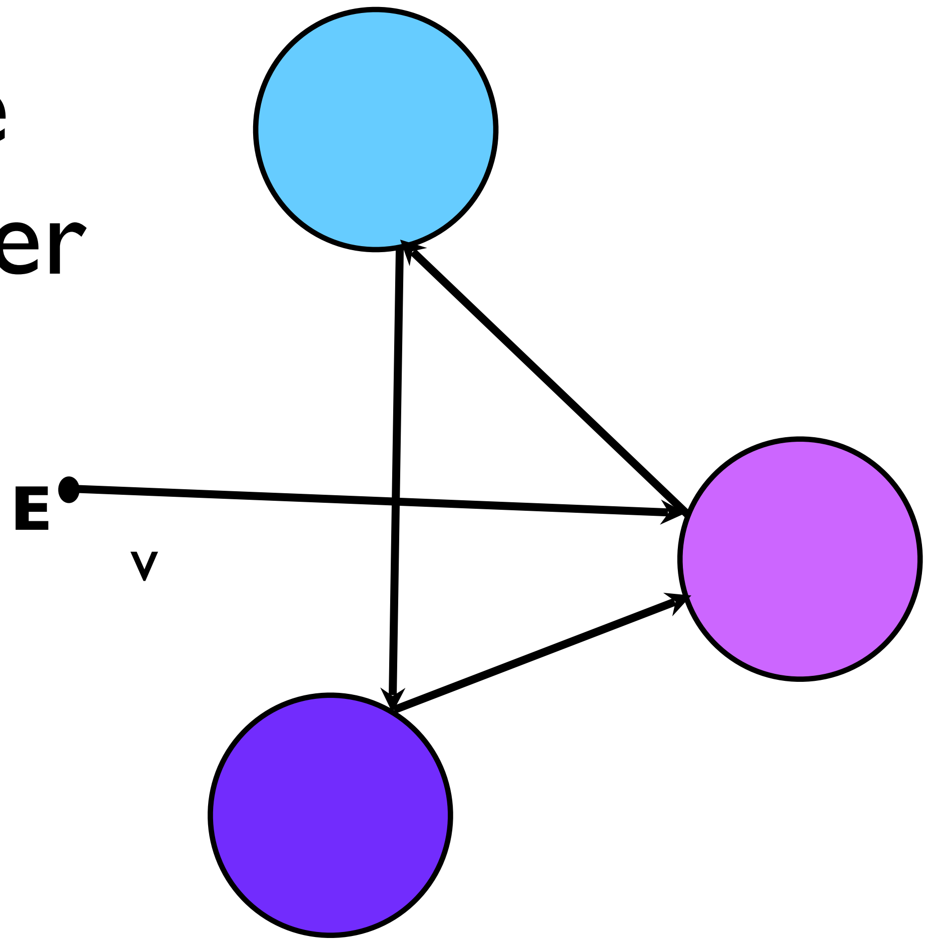


$$\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

Reflections

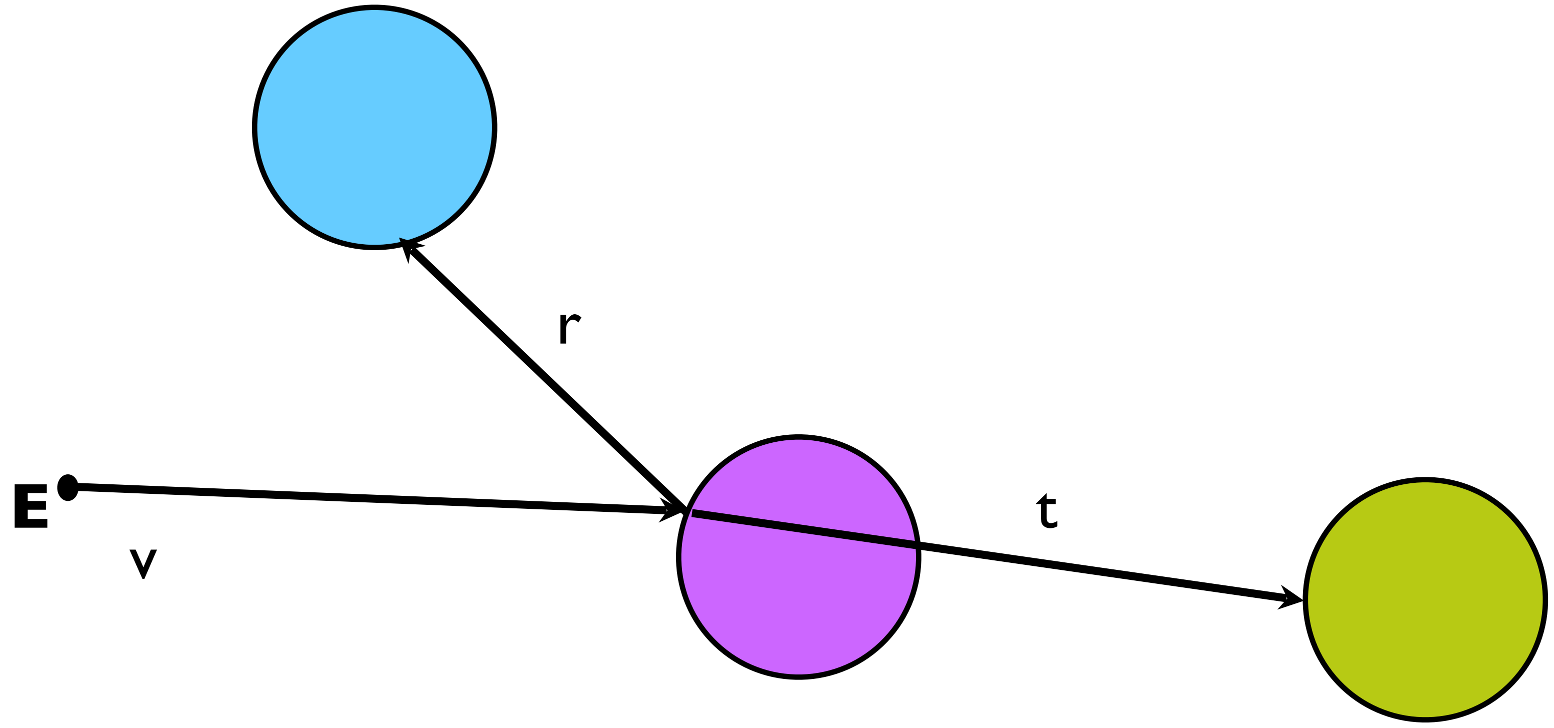
Reflected rays can in turn be reflected off another object and another.

We usually write out code to stop after a fixed number of reflections to avoid infinite recursion.



Transparency

We can also model transparent objects by casting a second ray that continues through the object.



Transparency

Transparency can also be applied reflexively, yielding a tree of rays.

$$\begin{tikzcd}
 & & & \text{Eye} \arrow[d, "\mathbf{v}"] & & & \\
 & & & \text{Hit} \arrow[dll, "\mathbf{t1}"'] \arrow[drr, "\mathbf{r1}"] & & & \\
 & \text{Hit} \arrow[dl, "\mathbf{t2}"'] \arrow[dr, "\mathbf{r2}"] & & & & \text{Hit} \arrow[dl, "\mathbf{t3}"'] \arrow[dr, "\mathbf{r3}"] & \\
\text{Hit} & & \text{Hit} & & \text{Hit} & & \text{Hit}
\end{tikzcd}$$

Illumination

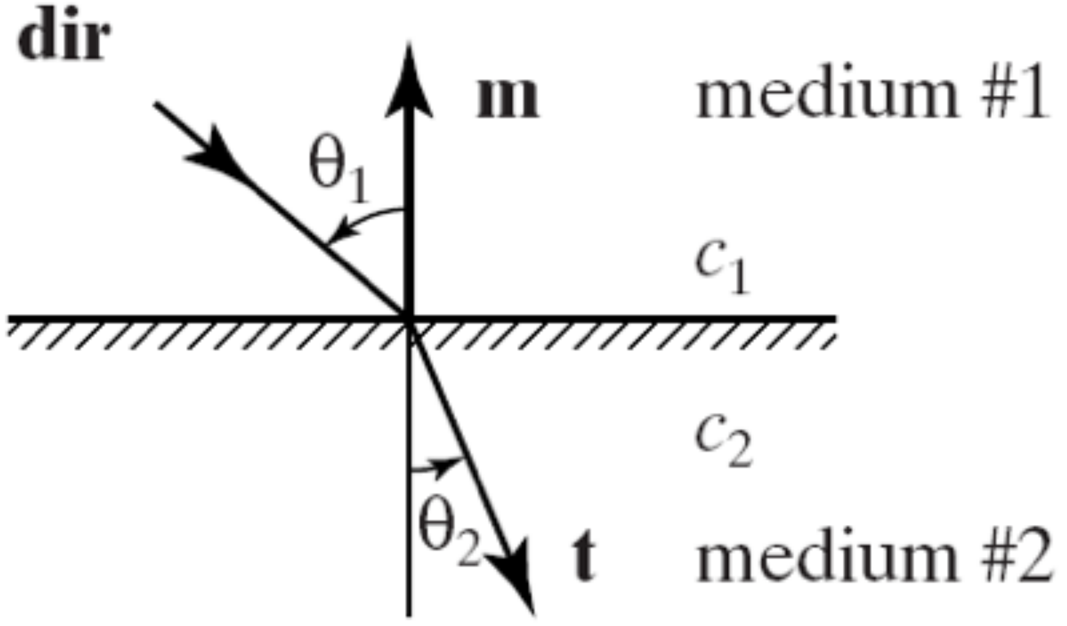
The illumination equation is extended to include reflected and transmitted components, which are computed recursively:

$$I(P) = I_{amb} + I_{dif} + I_{spe} + \boxed{I(P_{ref}) + I(P_{tra})}$$

We will need material coefficients to attenuate the reflected and transmitted components appropriately.

Refraction of Light

When a light ray strikes a transparent object, a portion of the ray penetrates the object. The ray will change direction from **dir** to **t** if the speed of light is different in medium 1 and medium 2. Vector **t** lies in the same plane as **dir** and the normal **m**.



Refraction

To handle transparency appropriately we need to take into account the refraction of light.

Light bends as it moves from one medium to another. The change is described by Snell's Law:

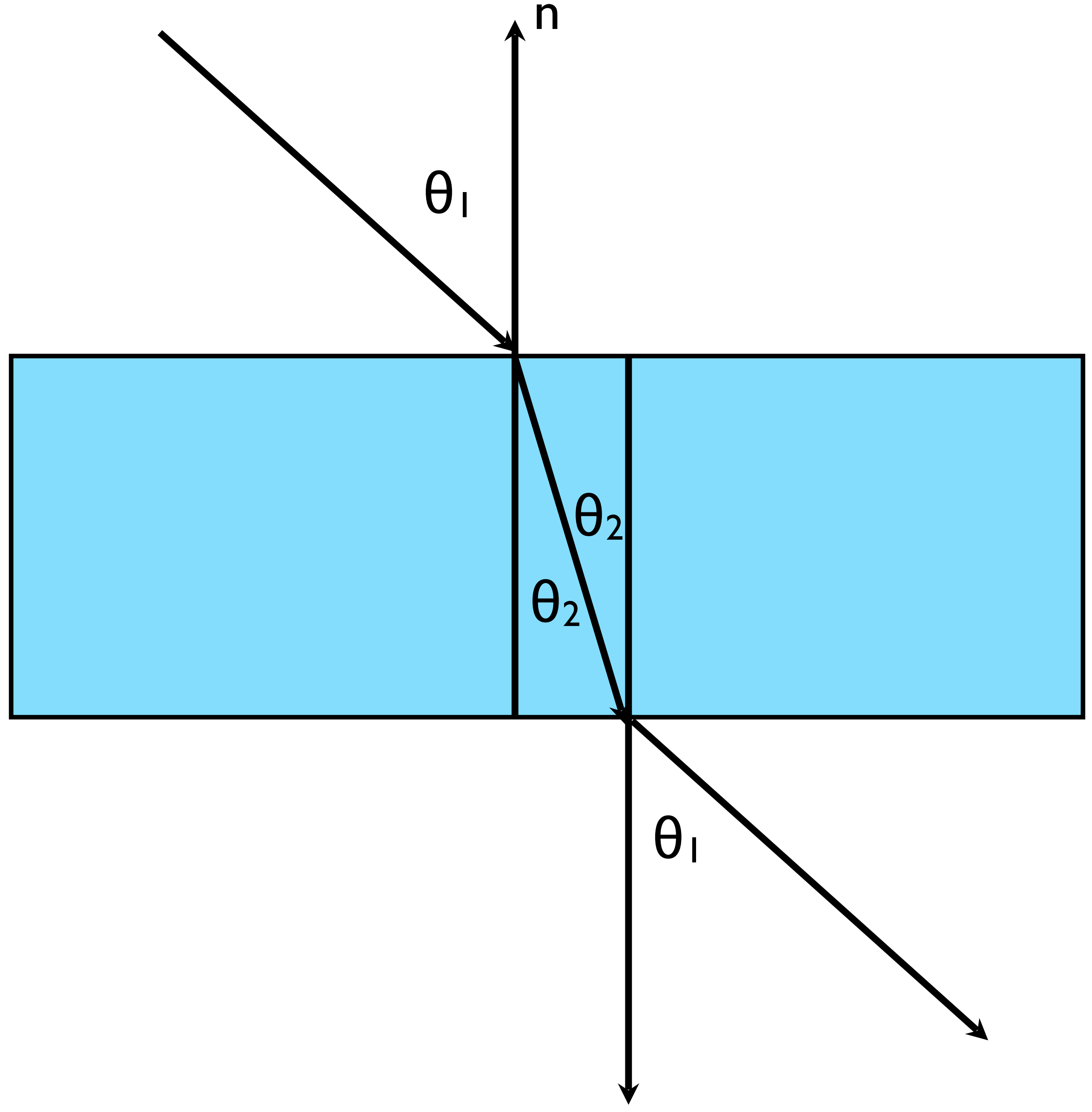
$$\frac{\sin\theta_1}{c_1} = \frac{\sin\theta_2}{c_2}$$

where c1 and c2 are the speeds of light in each medium.

Example Snell's law

Air:
c1 = 99.97% c

Glass:
c2 ~= 55% c



Example

Suppose medium 2 is some form of glass in which light only travels 55% as fast as in medium 1 which is the air. Suppose the angle of incidence of the light is 60 degrees from the normal. What is the angle of the transmitted light?

c2/c1 = 0.55

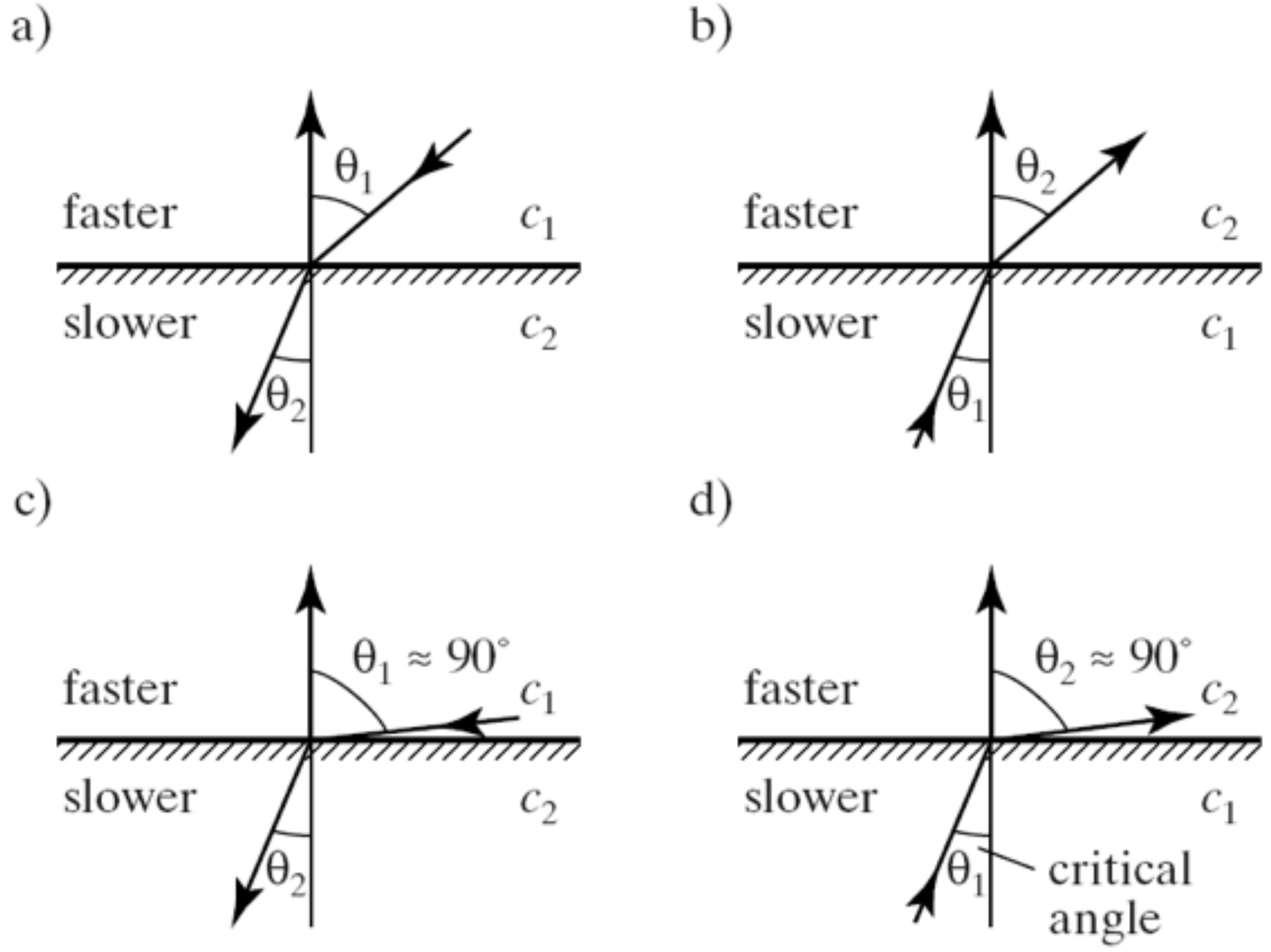
sin(theta2) = 0.55 * sin(60)

theta2 = 28.44 degrees

Refraction

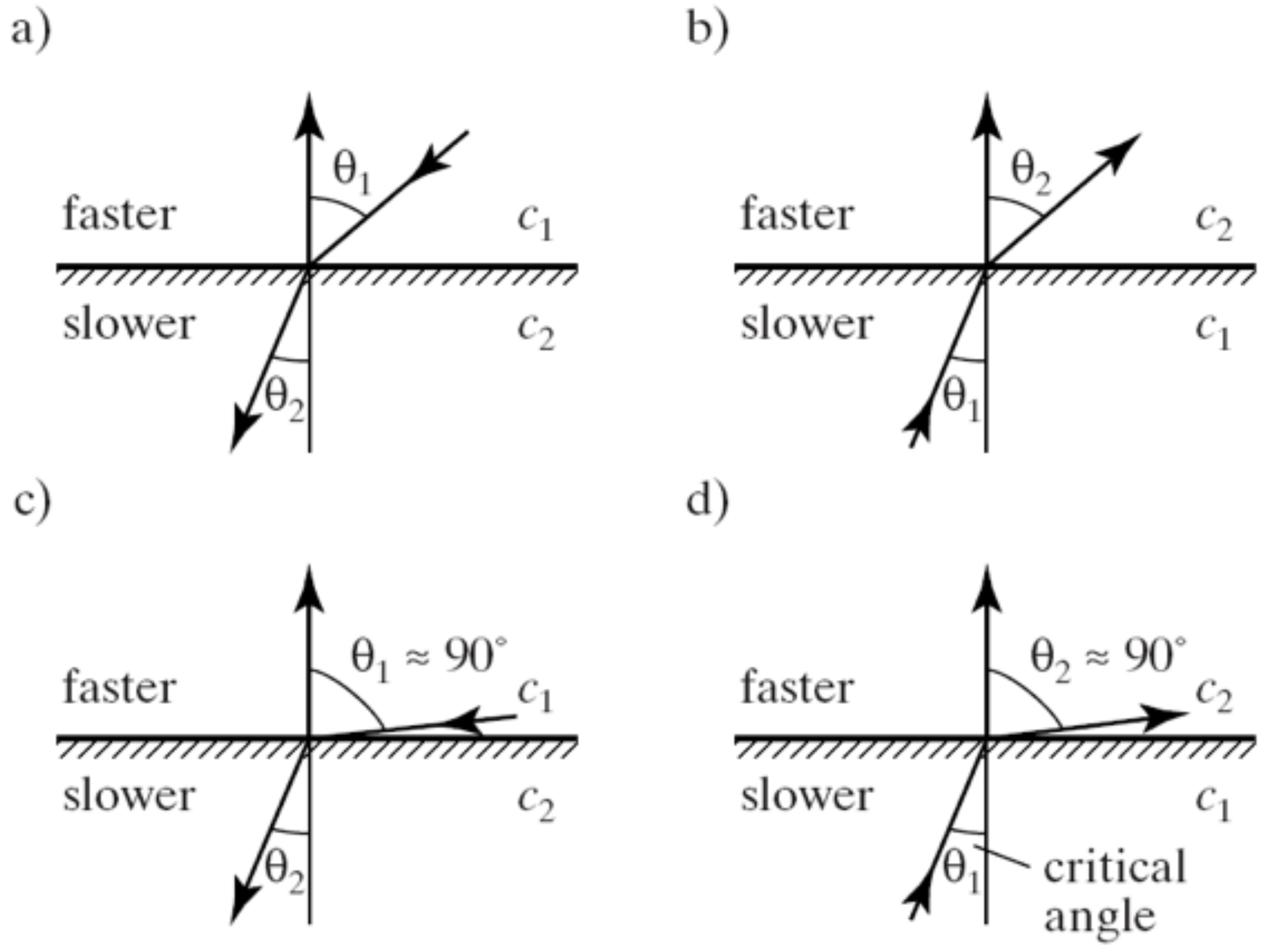
The figure (a) shows light moving from the faster medium to the slower, and (b) shows light moving from the slower to the faster medium.

The angles pair together in the same way in both cases; only the names change.



Refraction

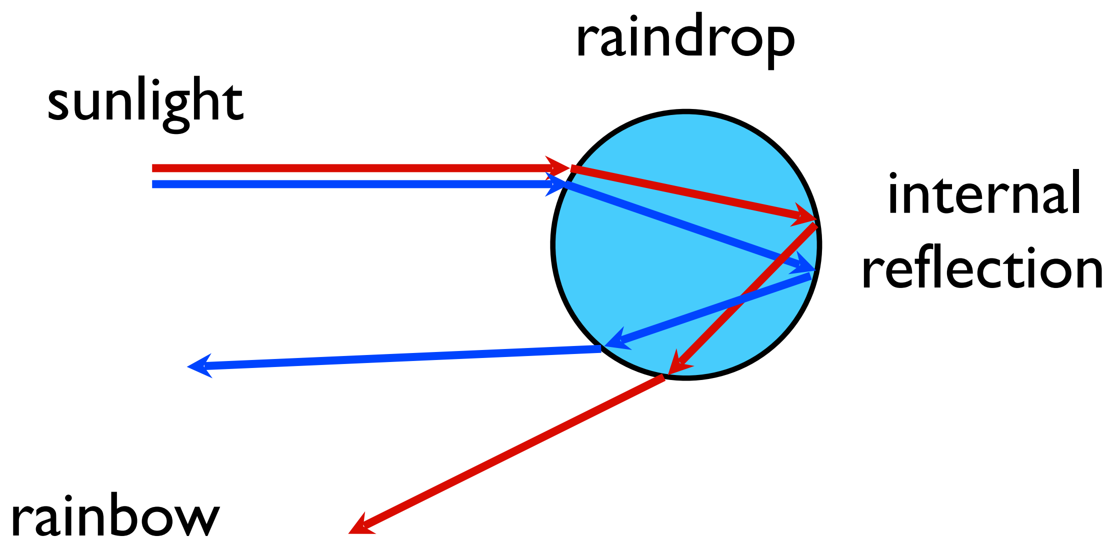
In (c) and (d), the larger angle has become nearly 90^0. The smaller angle is near the **critical angle**: when the smaller angle (of the slower medium) gets large enough, it forces the larger angle to 90^0. A larger value is impossible, so no light is transmitted into the second medium. This is called **total internal reflection**.



Refraction

Different wavelengths of light move at different speeds (except in a vacuum).

So for maximum realism, we should calculate different paths for different colours.



Refraction of Light

Simplest to model transparent objects so that their index of refraction does not depend on wavelength.

To do otherwise would require tracing separate rays for each of the color components, as they would refract in somewhat different directions.

This would be expensive computationally, and would still provide only an approximation, because an accurate model of refraction should take into account a large number of colors, not just the three primaries.

Exercise

How does milk look different to white paint?

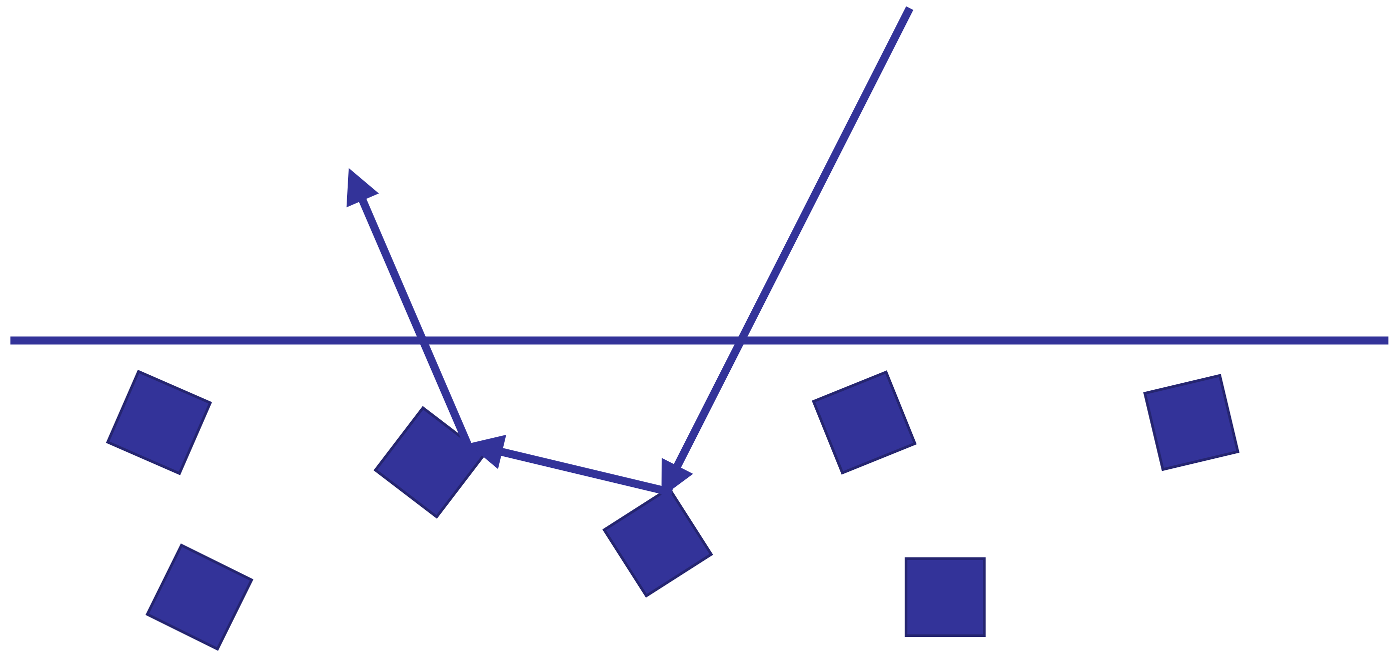
Both are opaque.

Both are essentially pure white.

Milk is an example of scattering

Scattering

Scattering (or subsurface scattering) is when light refracts into an object that is non-uniform in its density and is reflected out at a different angle and position.

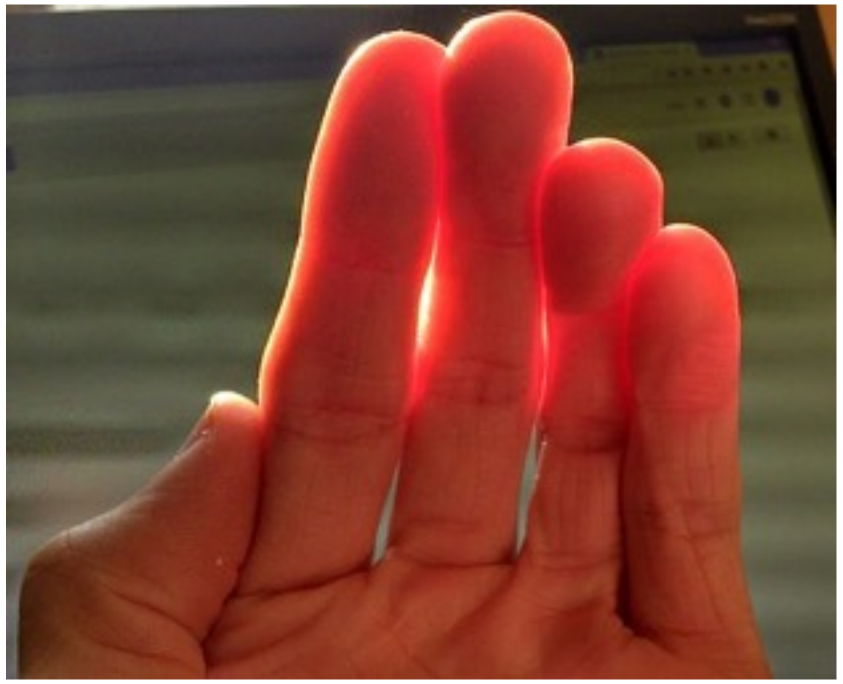


Scattering

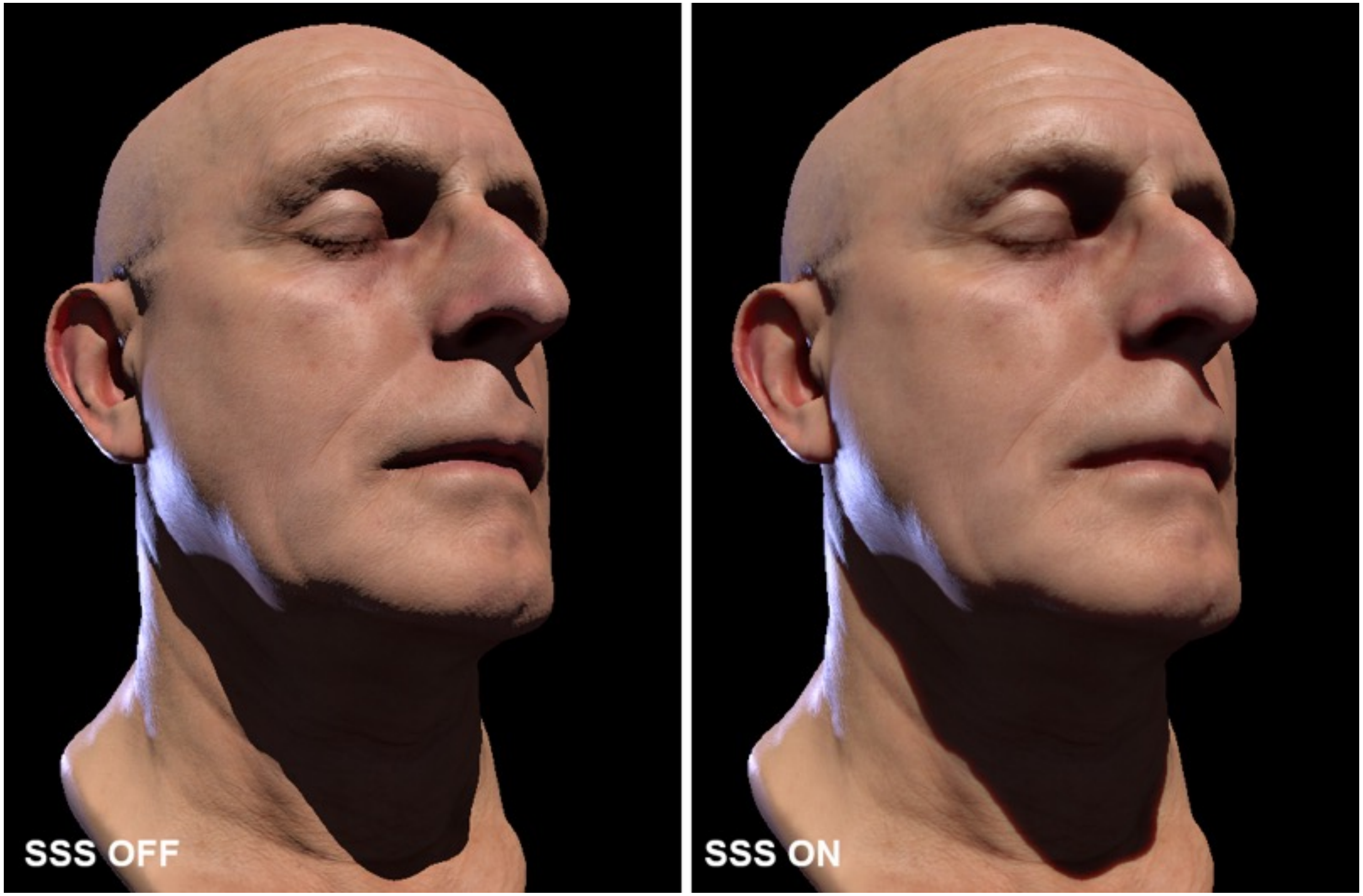
Milk is a substance that has this property.

As is skin, leaves and wax.

Typically they are hard to render.



Scattering



Scattering

We don't really have time to cover this in more depth in this course. Read this if you want to know more (NOT EXAMINABLE).

http://graphics.ucsd.edu/~henrik/images/subsurf.html

Raytracing Can’t Do

Basic recursive raytracing cannot do:

- Light bouncing off a shiny surface like a mirror and illuminating a diffuse surface
- Light bouncing off one diffuse surface to illuminate others
- Light transmitting then diffusing internally

Also a problem for rough specular reflection

- Fuzzy reflections in rough shiny objects

Raytracing Examples

https://www.youtube.com/watch?v=h5mRRElXy-w

https://www.youtube.com/watch?v=pm85W-f7xuk

https://www.youtube.com/watch?v=XVZDH15TRro

https://www.youtube.com/watch?v=zx48ntkDai0

Volumetric ray tracing

We can also apply ray tracing to volumetric objects like smoke or fog or fire.

Such objects are transparent, but have different intensity and transparency throughout the volume.

Volumetric Ray Tracing

We represent the volume as two functions:

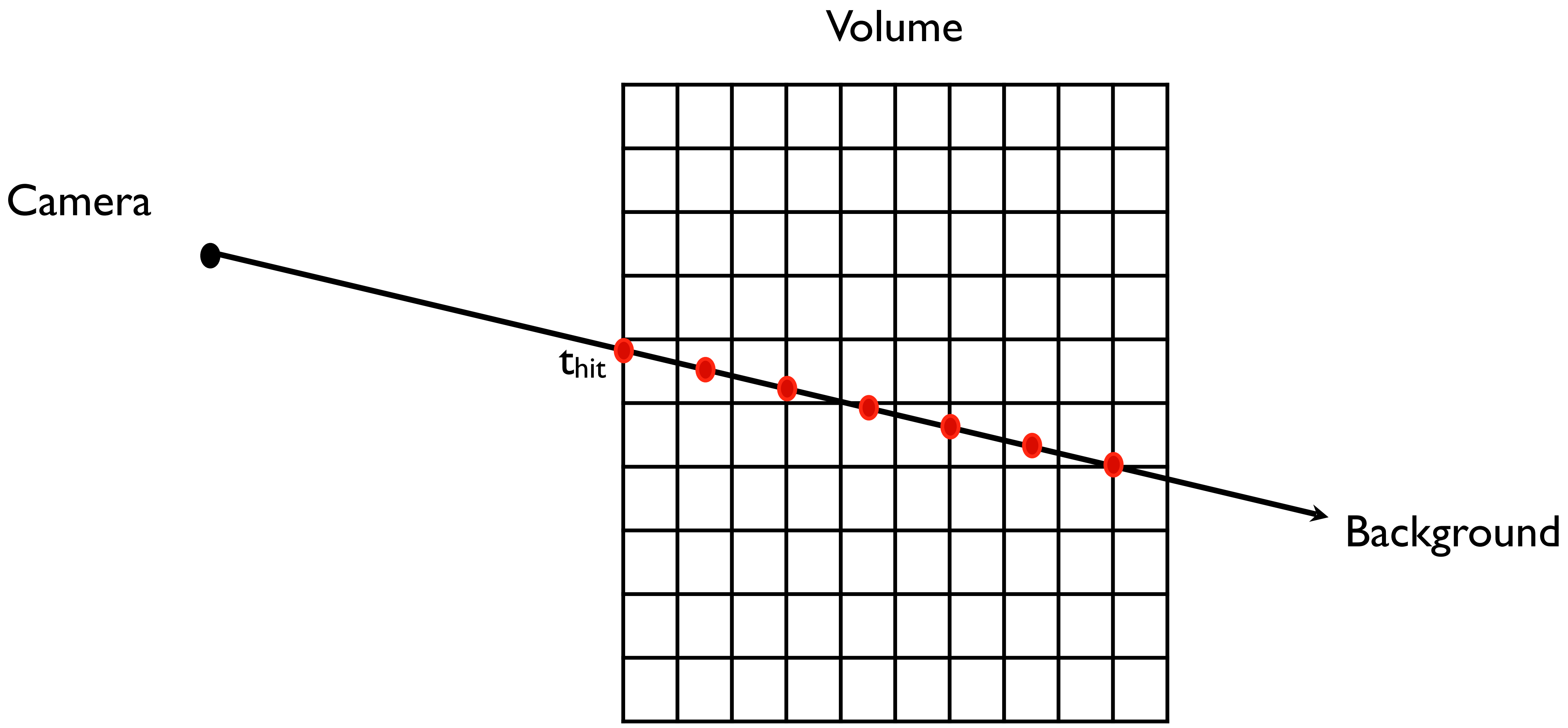
$$\begin{aligned} C(P) &= \text{colour at point } P \\ \alpha(P) &= \text{transparency at point } P \end{aligned}$$

Typically these are represented as values in a 3D array. Interpolation is used to find values at intermediate points.

These functions may in turn be computed based on density, lighting or other physical properties.

Sampling

We cast a ray from the camera through the volume and take samples at fixed intervals along the ray.



Sampling

We end up with (N+1) samples:

$$\begin{aligned}
P_i &= R(t_{hit} + i\Delta t) \\
C_i &= C(P_i) \\
\alpha_i &= \alpha(P_i) \\
C_N &= (r, g, b)_{background} \\
\alpha_N &= 1
\end{aligned}$$

Alpha compositing

We now combine these values into a single colour by applying the alpha-blending equation.

$$C_N^N = C_N$$

$$\underset{\boxed{\text{Total colour at i}}}{C_N^i} = \alpha_i \underset{\boxed{\text{Local colour at i}}}{C_i} + (1 - \alpha_i) \underset{\boxed{\text{Total colour at i+1}}}{C_N^{i+1}}$$

Example

We have a background color of (0,1,0)

And 2 other samples that both have color (1,0.5,0.5) and alpha_0 is 0.2 and alpha_1 is 0.1.

$$C_2 = (0,1,0)$$
$$C_1 = 0.1(1,0.5,0.5) + 0.9(0,1,0) = (0.1,0.95,0.05)$$
$$C_0 = 0.2(1,0.5,0.5) + 0.8(0.1,0.95,0.05) = (0.28,0.86,0.14)$$

Alpha compositing

We can write a closed formula for the colour from a to b as:

$$C_b^a = \sum_{i=a}^{b} \alpha_i C_i \prod_{j=a}^{i-1} (1 - \alpha_j)$$

We can compute this function from front to back, stopping early if the transparency term gets small enough that nothing more can be seen.

In OpenGL

Volumetric ray tracing (aka ray marching) does not require a full ray tracing engine.

It can be implemented in OpenGL as a fragment shader applied to a cube with a 3D texture.

https://www.shadertoy.com/view/XslGRr

See http://shadertoy.com/ for more examples.

Sources

http://en.wikipedia.org/wiki/Volume_ray_casting

http://graphics.ethz.ch/teaching/former/scivis_07/Notes/Slides/03-raycasting.pdf

http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html