# COMP3421

Hidden surface removal
Local Illumination

The fixed function graphics pipeline
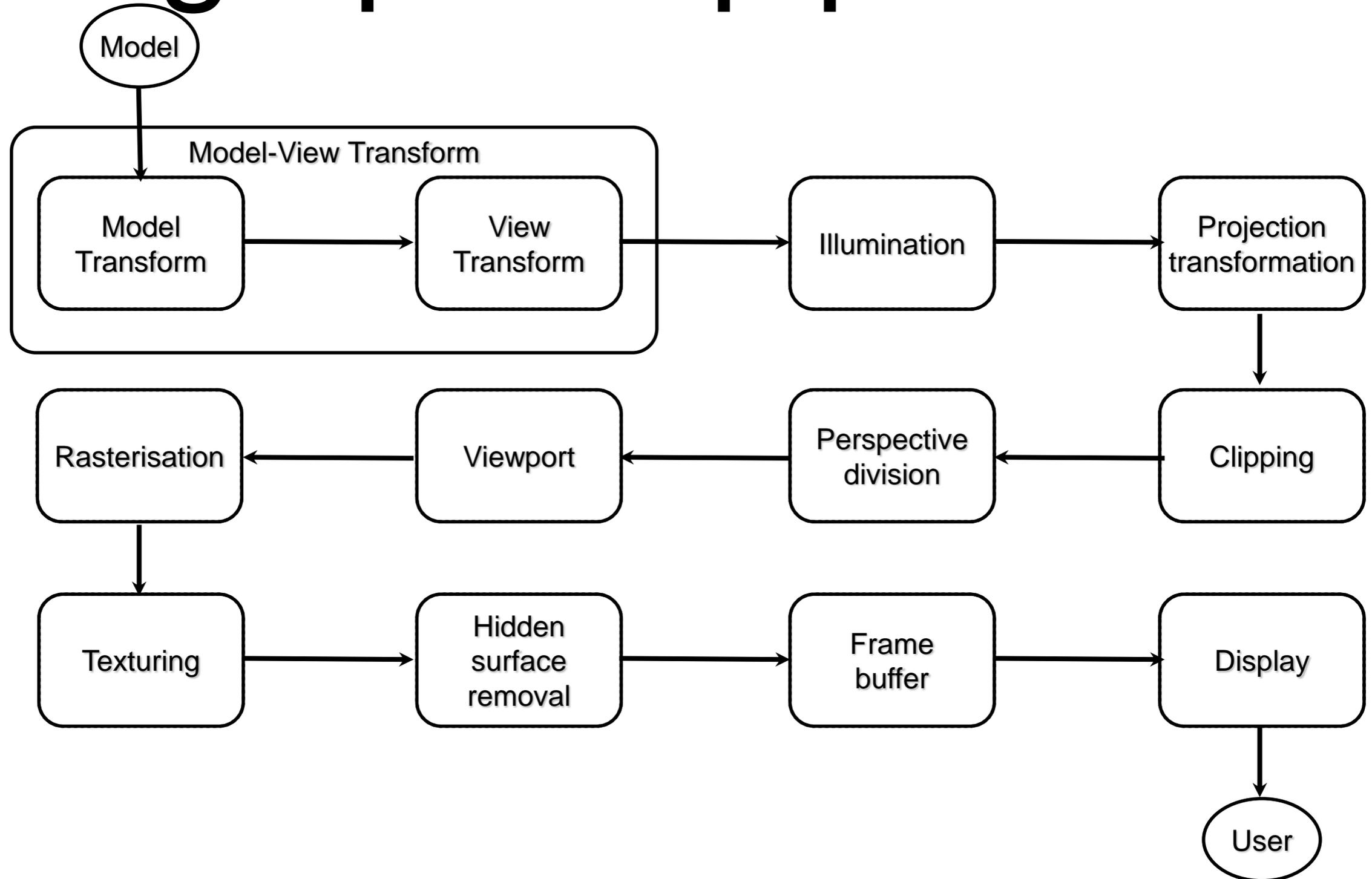
Fragments

Rasterisation converts polygons into collections of fragments.

A fragment corresponds to a single image pixel.

A fragment may not make it to the final image if it is discarded by the depth testing etc

There are many more fragments that vertices!

Fragments

Every fragment that is created has:

```
float[3] pos;    // pixel coords and
                 // depth info
float[4] color;  // rgba colour
```

And possibly other values.

Back face culling

An optimisation called face culling allows non-visible triangles of closed surfaces (back faces) to be culled in the rasteriser.

This avoids unnecessary fragments from being created.

Calculate the signed area in window coordinates. If it is negative, ignore the polygon.

This is based on the winding order of the polygon – front faces are CCW by default.

Back face culling

```
// Disabled by default

// To turn on culling:

gl.glEnable(GL2.GL_CULL_FACE);

gl.glCullFace(GL2.GL_BACK);
```

Back Face Culling Example

SphereInBoxMyLighting

Turn box around and show they are not back faces

Then open box

Cull front faces

Then cull on shere.

Check our solution from before the exercise.

Teapot oengl culling.

Scaling with/without normalise

Hidden surface removal

We now have a list of fragments expressed in screen coordinates and pseudodepth.

For any particular pixel on the screen, we need to know what we can see at that pixel.

Some fragments may be behind other fragments and should not

be seen
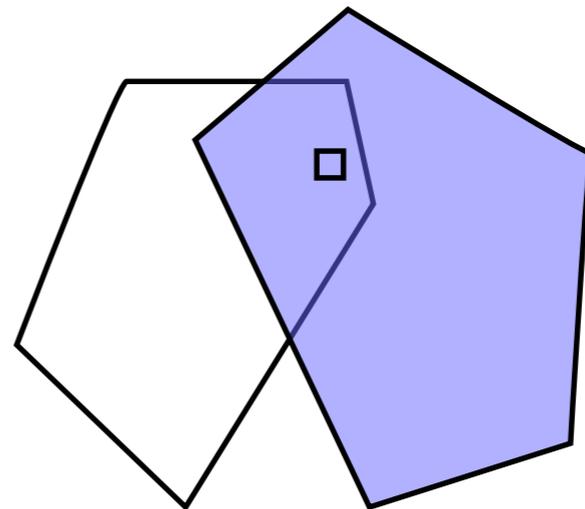
Hidden Surface Removal

We will look at 2 approaches to this problem.

1. Make sure all polygons (and therefore fragments) are drawn in the correct order in terms of depth using BSP trees. This is done at the model level. This is not built into OpenGL.

2. Use the depth buffer. This is done at the fragment level and is built into OpenGL.

Painter's algorithm

The naive approach to hidden surface removal:

- Sort polygons by depth

- Draw in order from back to front

This is known as the Painter's algorithm
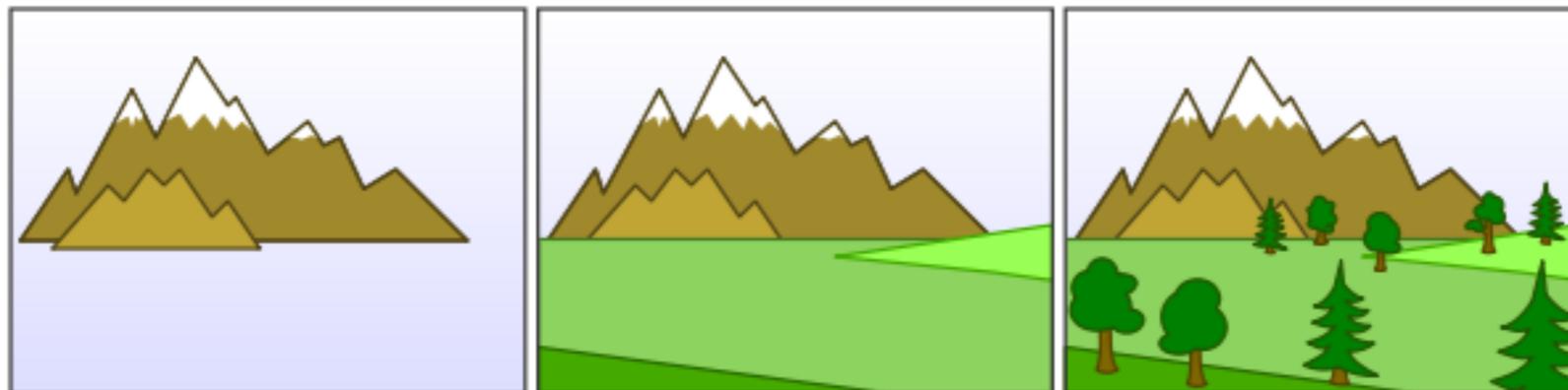
Problem

Which polygon to paint first?

We need to split them into pieces.

If you decide to split,

what key do you sort on?

The center of the polygon?

The nearest vertex? The furthest?
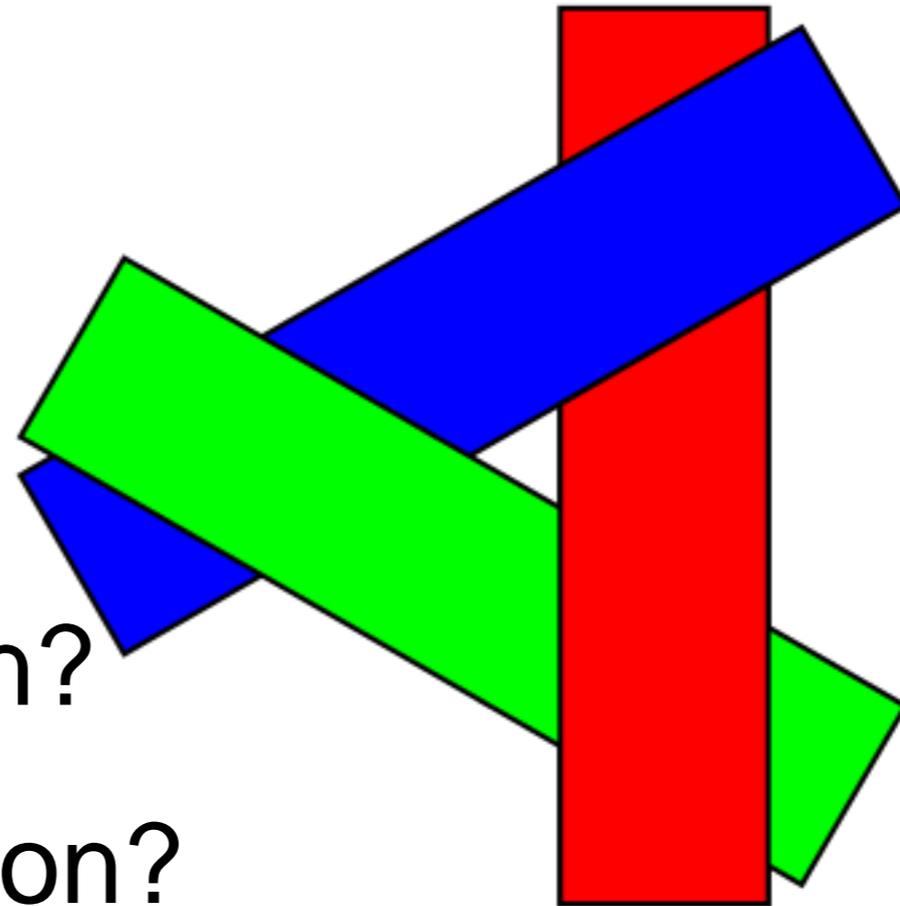
BSP Trees

Binary Space Partitioning trees

Not implemented in OpenGL

Each polygon splits the world half

- things behind of the polygon

- things in front of the polygon

We can divide all the other polygons into two sets.

BSP Trees

BSP Trees

6

4

Back
polygons

5

1

1

front   back

2,3   4,5,6

Front
polygons

2

3
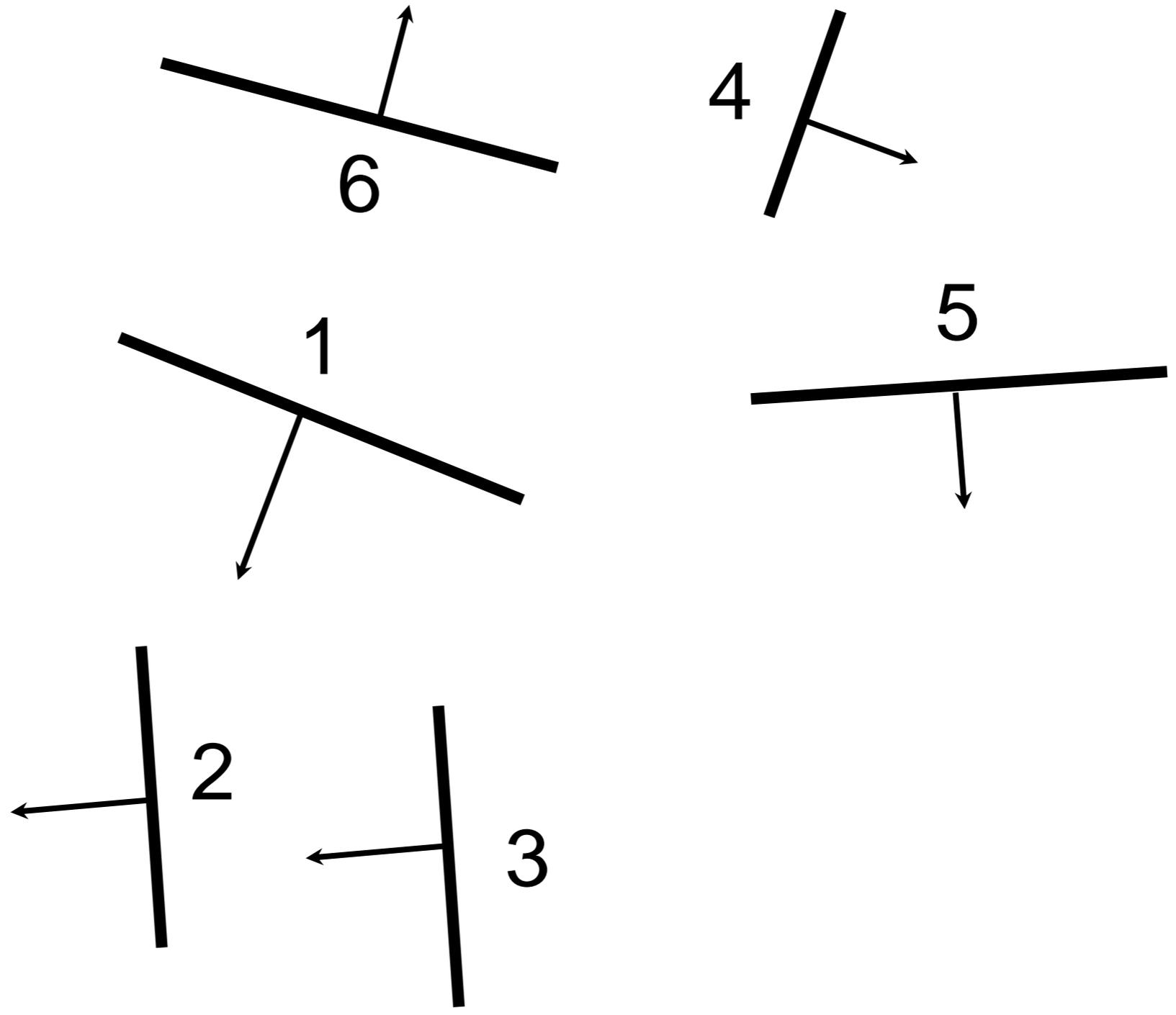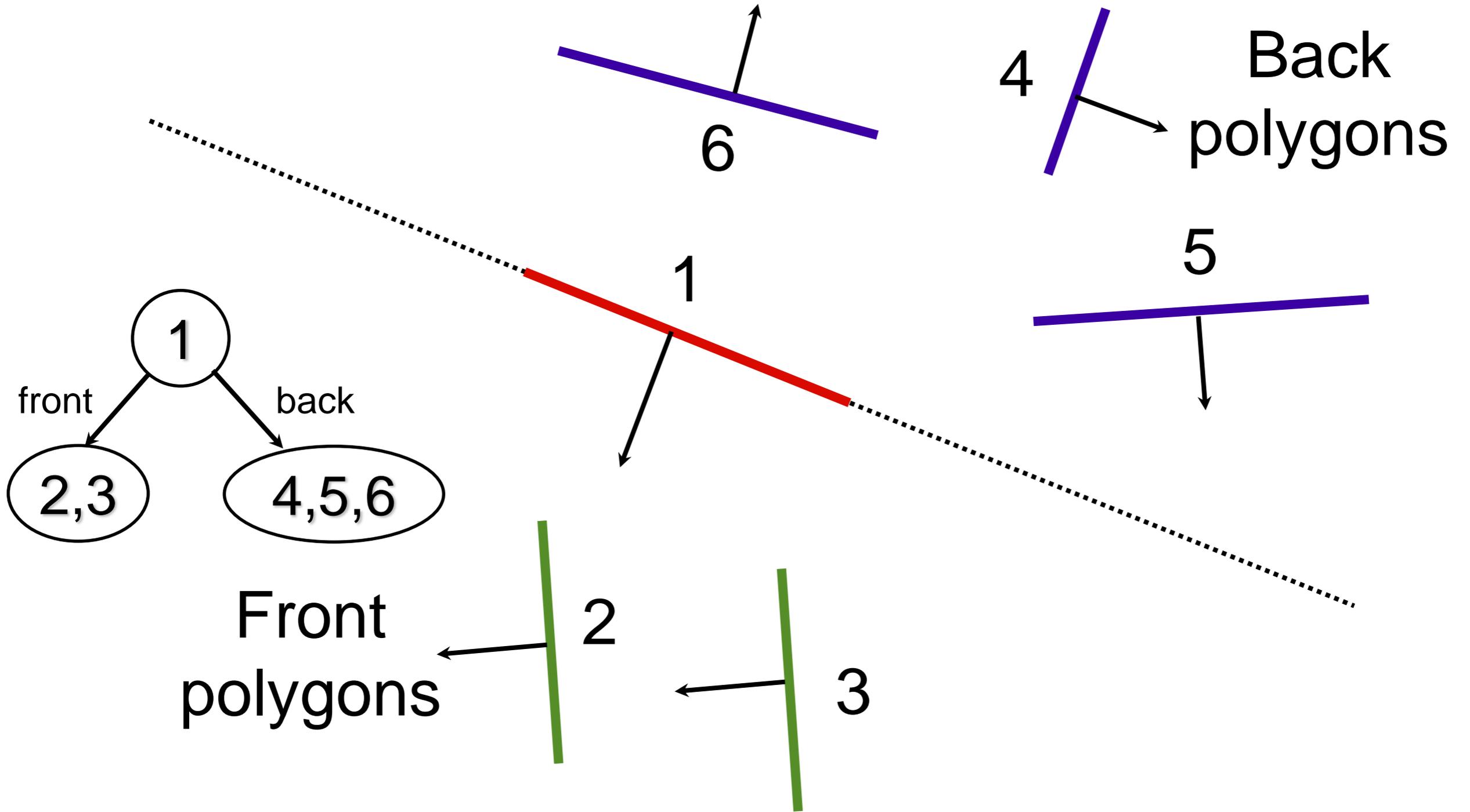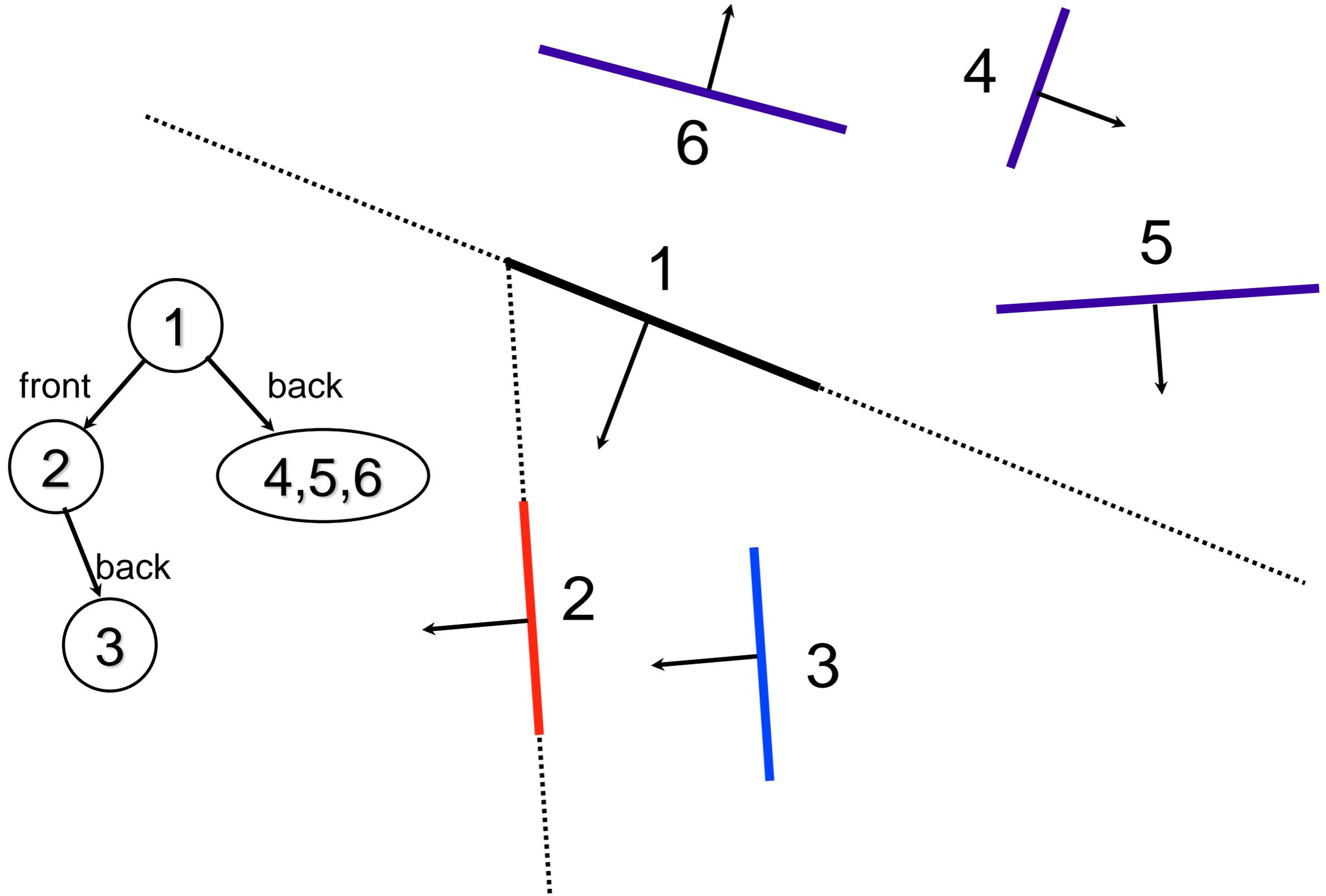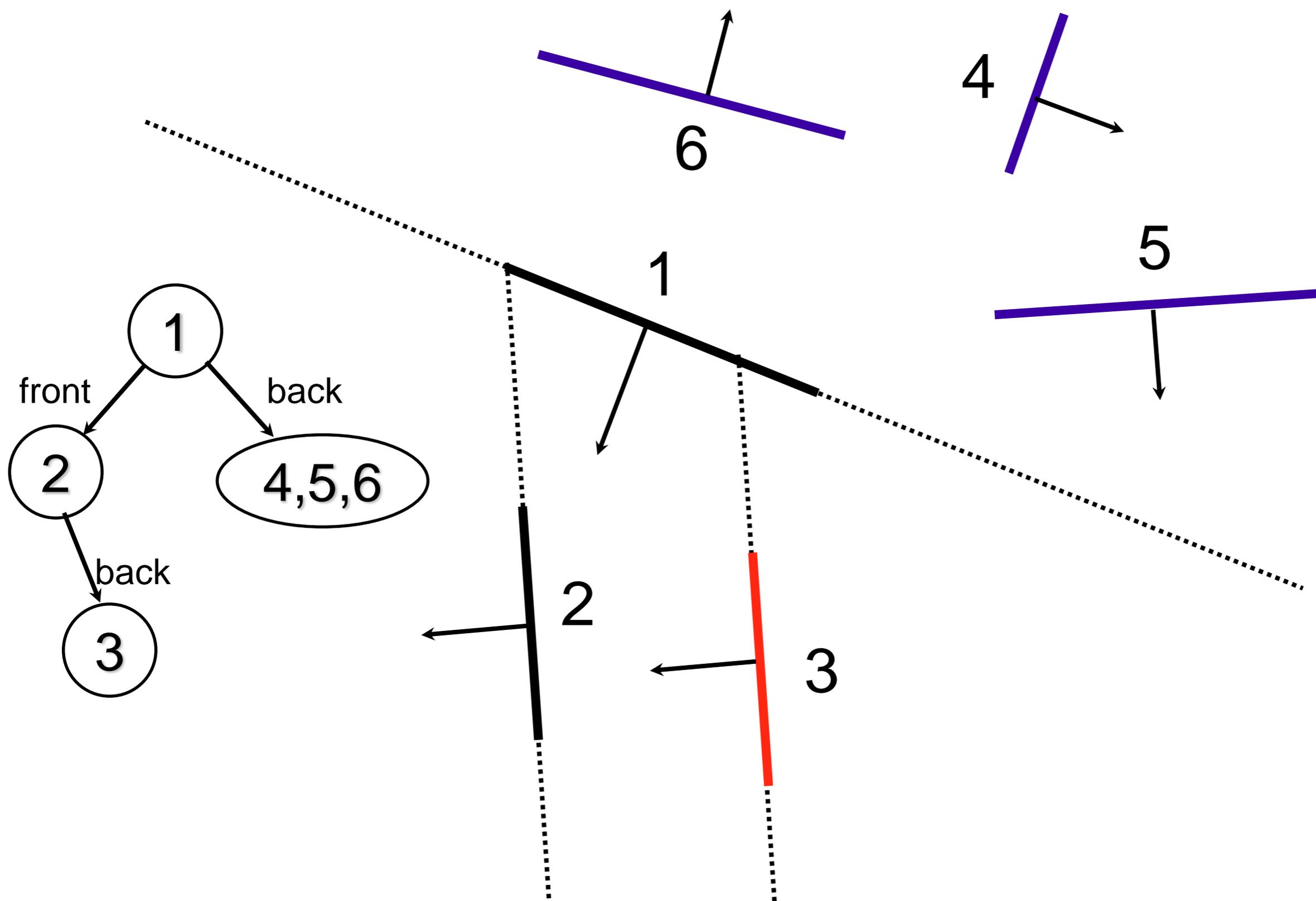
# BSP Trees
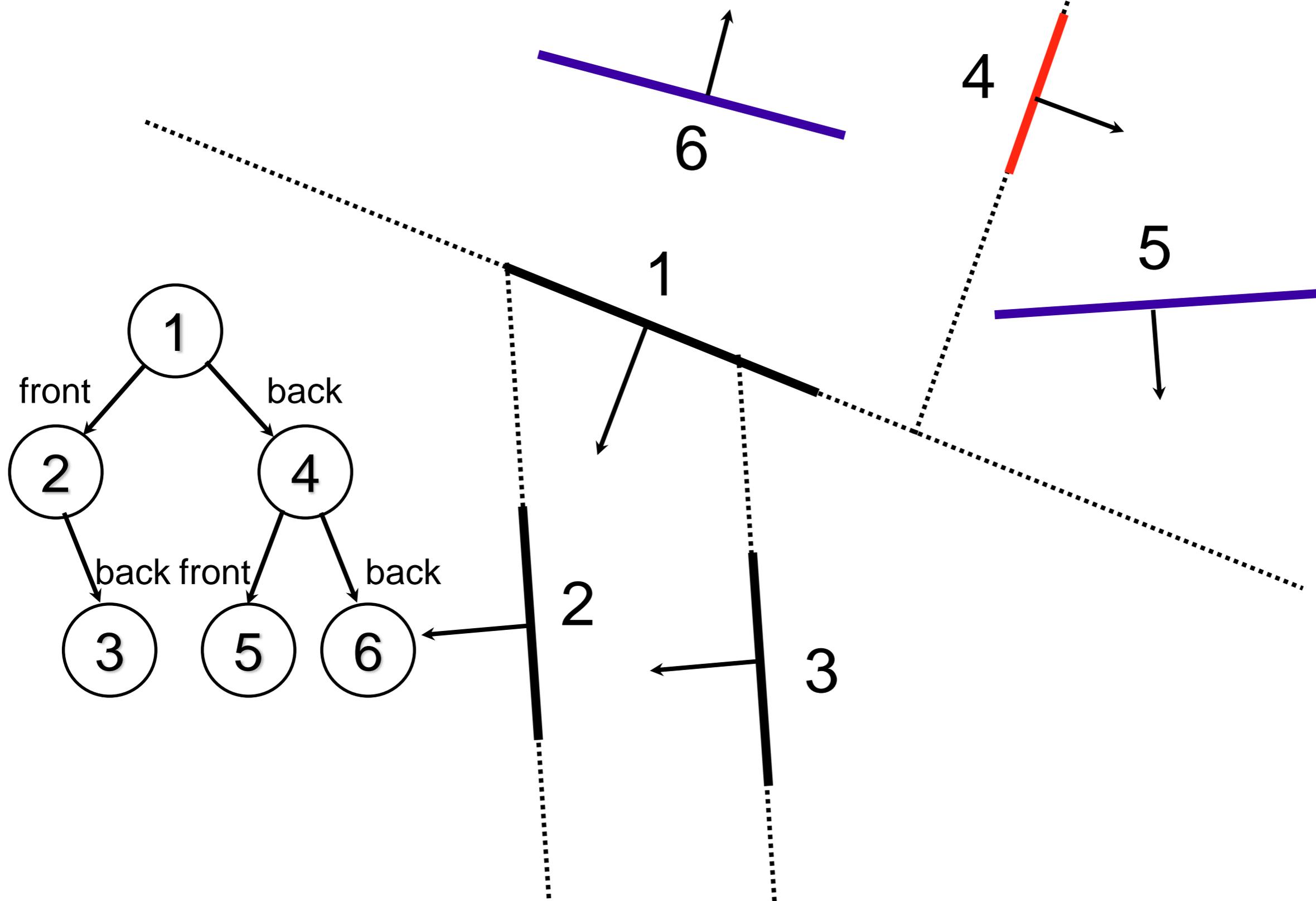
BSP Trees

BSP Trees

BSP Trees
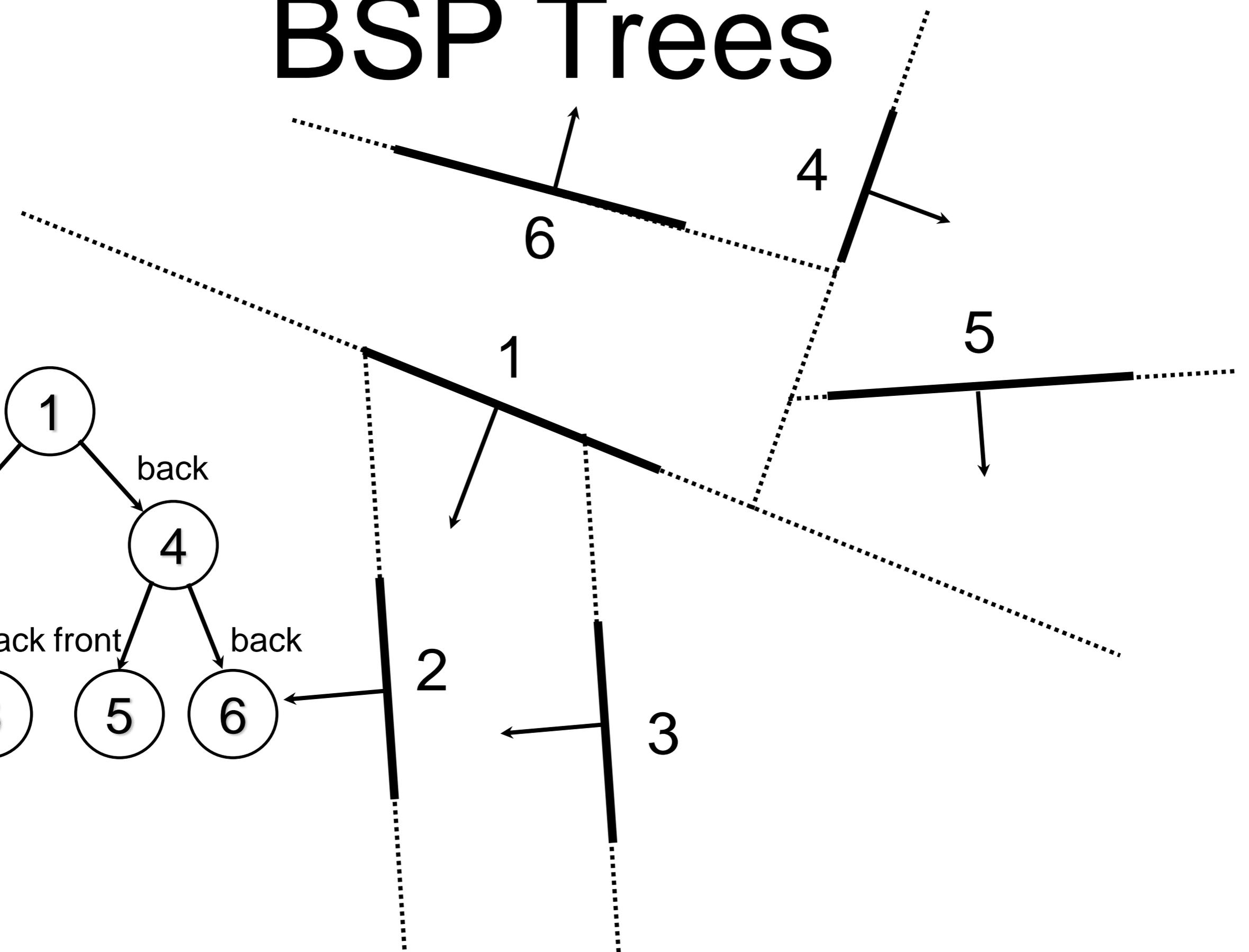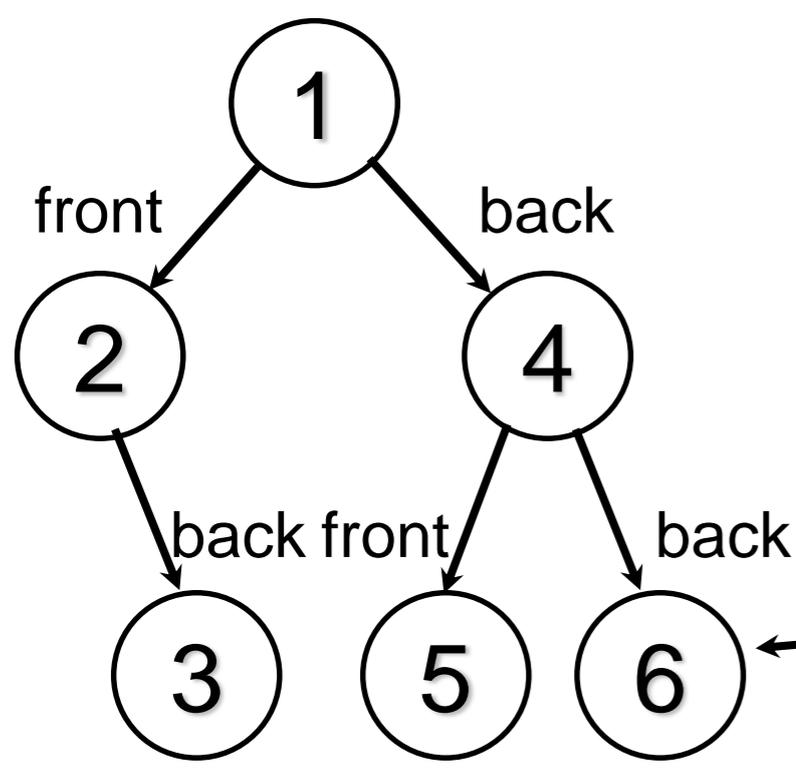
To traverse

If camera is in front of the polygon:

  draw what is behind the polygon
  draw the polygon
  draw what is in front of the polygon

If camera is behind the polygon:

  draw what is in front of the polygon
  draw the polygon
  draw what is behind the polygon

BSP Trees
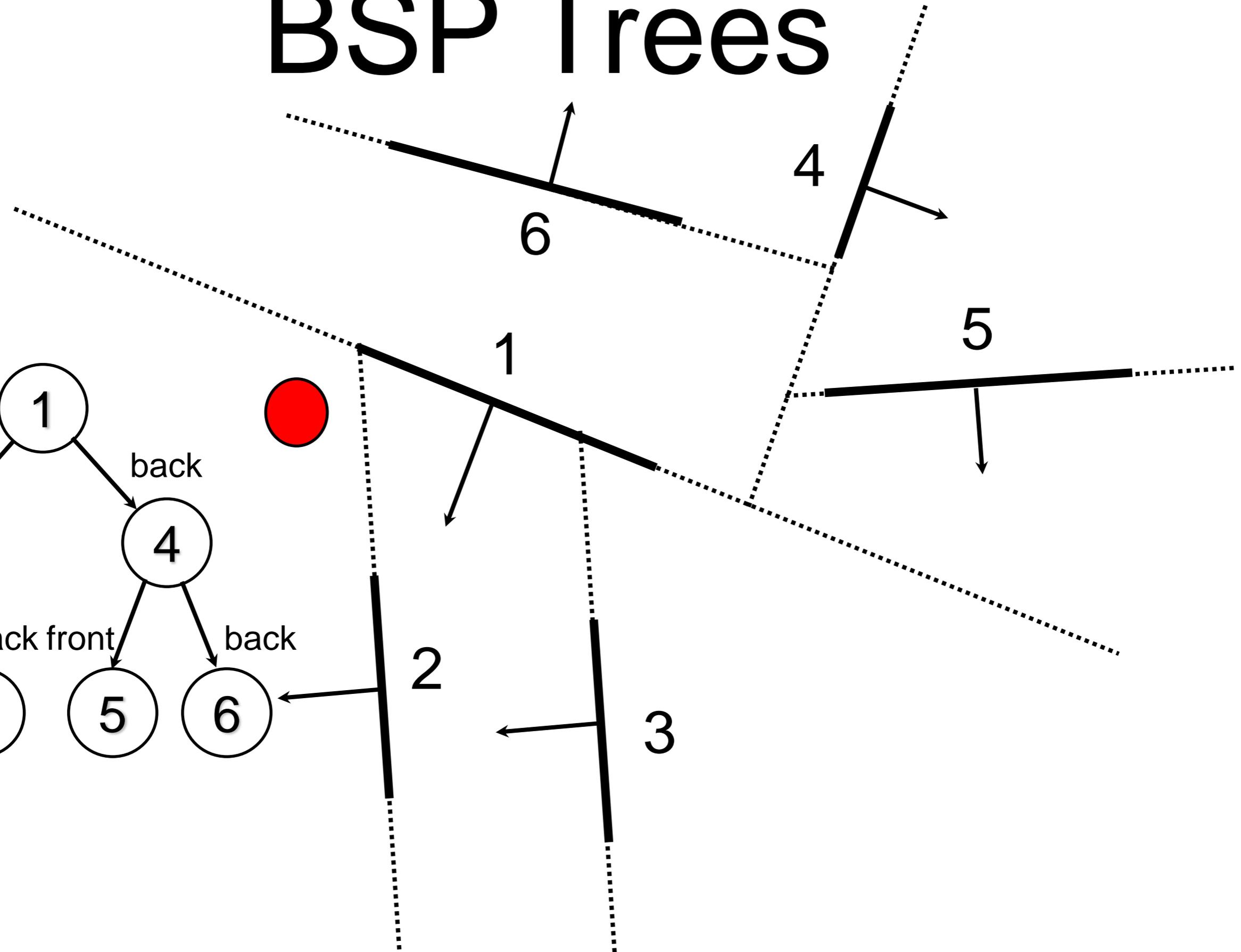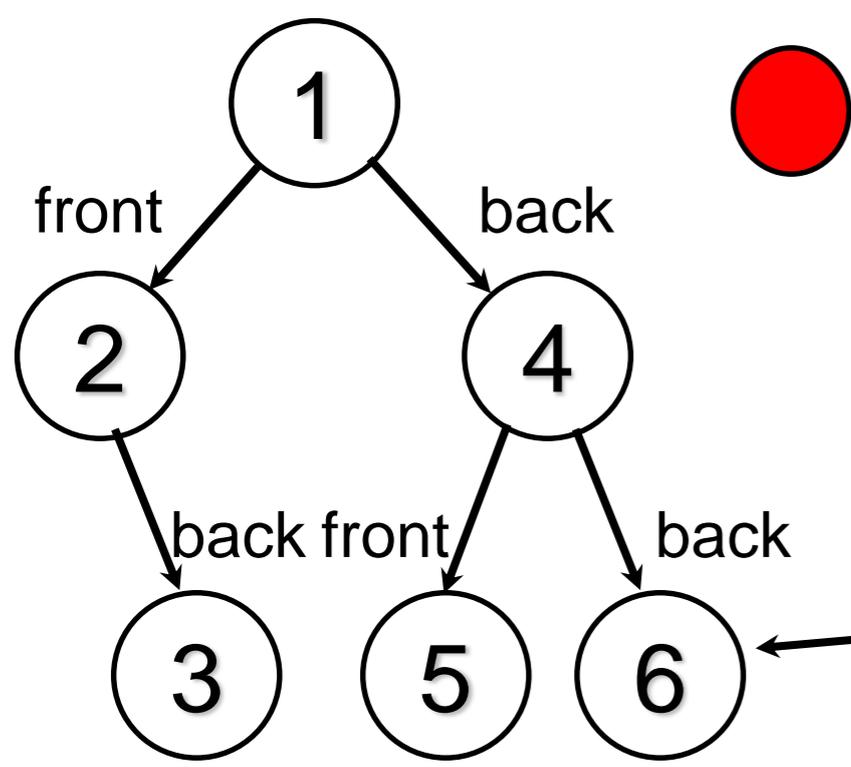
Example Traversal

The algorithm is first applied to the root node of the tree, node *1*. *Camera* is in **front** of node *1*, so we recursively traverse the BSP sub-tree containing polygons behind *1*.

- This tree has root node *4*. *Camera* is behind *4* so we traverse the BSP sub-tree containing polygons in front of *4*:

  - 5 is a leaf node so we draw it

- Then we draw 4

Example Traversal

- We then traverse the BSP sub-tree containing polygons behind 4, this is just 6 so we draw 6.

We then draw 1 and apply traversal to all nodes in front of node 1

- Camera is in front of 2 so the nodes behind 2 are traversed. This is just node 3 which is drawn

- Finally node 2 is drawn

Example Traversal

So the order that we draw the polygons for the given camera position is

- 5 4 6 1 3 2

Splitting Polygons

4

Back polygons

1

2

3

Splitting Polygons

4

Back polygons

1

3b

front   back

1

2,3 a   4,3 b

3a

2
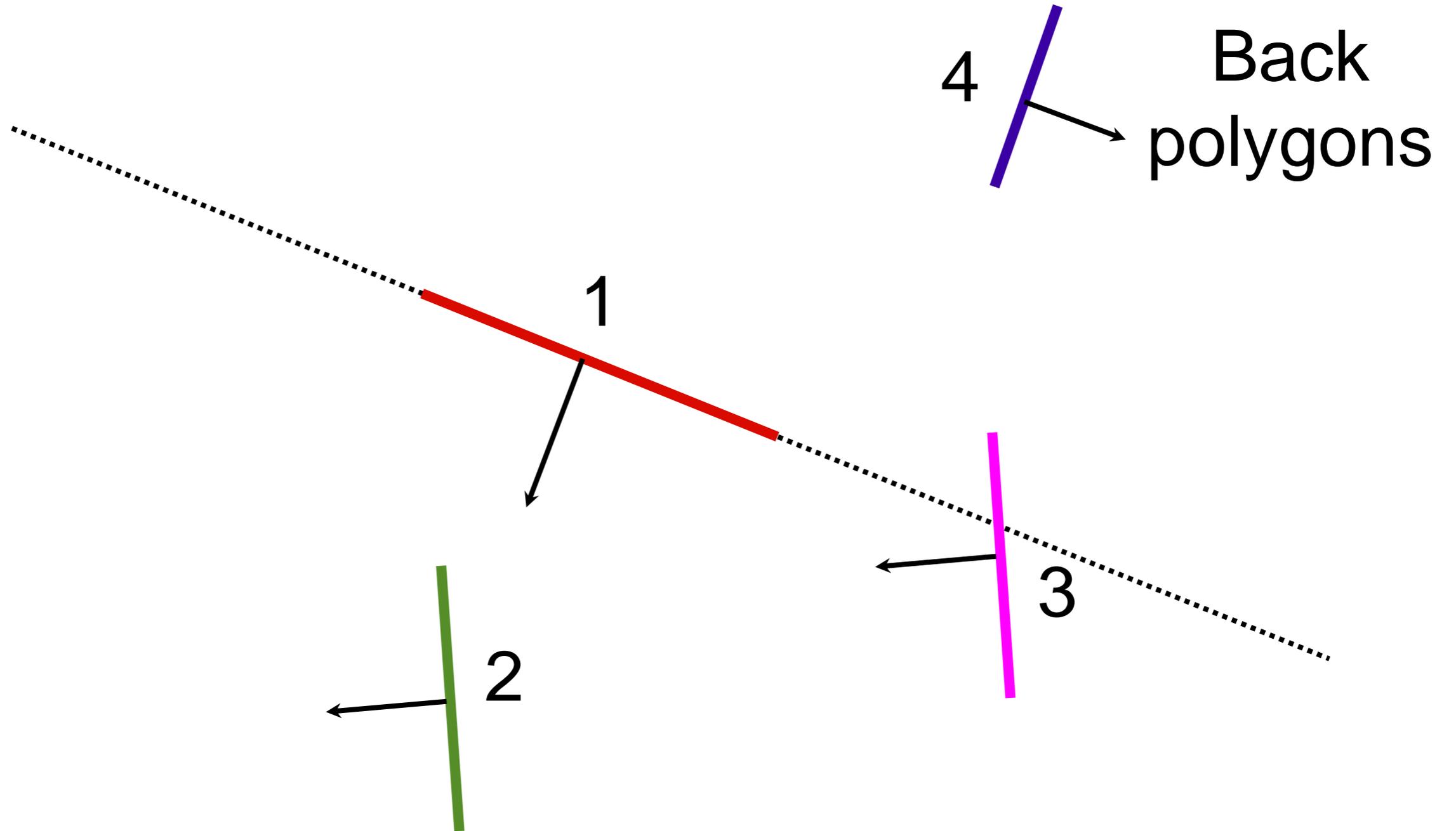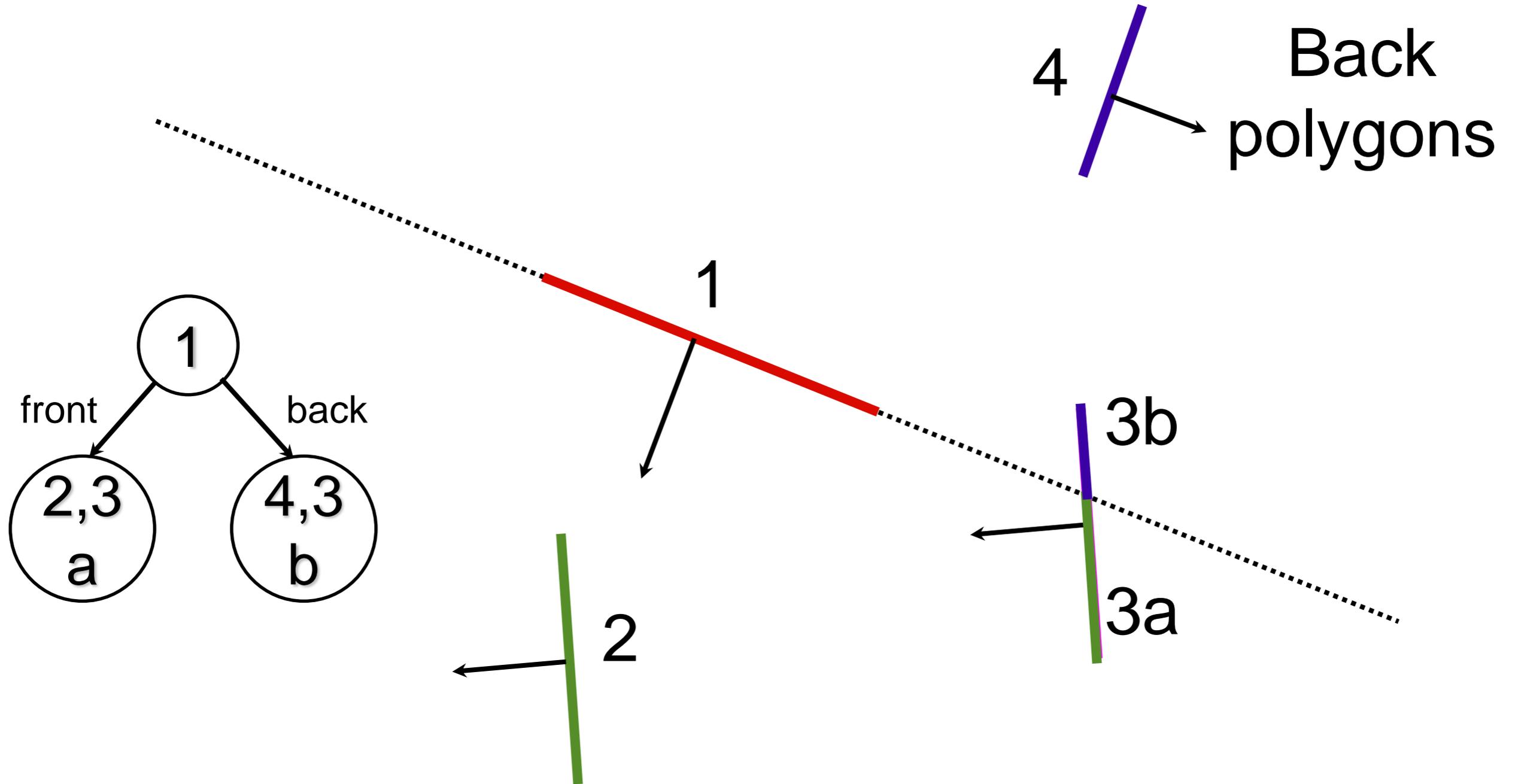
Good/bad trees

The order of polygons chosen for spliiting can greatly affect the size of the tree.

Bad choices lead to many splits = big tree.

Optimal tree is hard to find.

Randomly try (for example) 5 or 6 times and keep the best.

Pros + Cons

Good: If the camera/viewpoint moves the tree does not need to be recalculated. Can be pre-calculated and loaded in from a file.

Good: Testing is fast once the tree is built

Bad: Building the tree is slow.

Bad: If the geometry changes, the tree must be rebuilt.

Useful for rendering static geometry (eg walls).

Depth buffer

Another approach to hidden surface removal is to keep per-pixel depth information.

This is what OpenGL uses.

This is stored in a block of memory called the depth buffer (or z-buffer).

d[x][y] = pseudo-depth of pixel (x,y)

OpenGL

```
// in init()

gl.glEnable(GL2.GL_DEPTH_TEST);


// in display()

gl.glClear(
    GL.GL_COLOR_BUFFER_BIT |
    GL.GL_DEPTH_BUFFER_BIT);
```

Depth buffer

Initially the depth buffer is initialised to the far plane depth.

We draw each polygon fragment by fragment.

For each fragment we calculate its pseudodepth and compare it to the value in the depth buffer.

If it is closer, we update the pixel in the colour buffer and update the buffer value to the new pseudodepth. If not we discard it.

Pseudocode

```
Initialise db[x][y] = max for
all x,y

For each polygon:

  For each fragment (px,py):

    d = pseudodepth of (px,py)

    if (d < db[px][py]):

      draw fragment

      db[x][y] = d
```

Example

Polygon

Buffer

Example

Polygon

Buffer

Example

Polygon

				.7	.7	.7	.7	.7	.7
				.6	.6	.6	.6	.6	.6
				.5	.5	.5	.5	.5	.5
				.4	.4	.4	.4	.4	.4
				.3	.3	.3	.3	.3	.3
				.2	.2	.2	.2	.2	.2

Buffer

.1	1	1	1	1	1	1	1	1	1
.1	.1	1	1	1	1	1	1	1	1
.2	.2	.2	1	1	1	1	1	1	1
.2	.2	.2	.2	1	1	1	1	1	1
.3	.3	.3	.3	.3	1	1	1	1	1
.3	.3	.3	.3	.3	.3	1	1	1	1
.4	.4	.4	.4	.4	.4	.4	1	1	1
.4	.4	.4	.4	.4	.4	.4	.4	1	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	1
.5	.5	.5	.5	.5	.5	.5	.5	.5	.5

Example

Polygon

Buffer

Computing pseudodepth

We know how to compute the pseudo depth of a vertex.

How do we compute the depth of a pixel?

We use bilinear interpolation based on the depth values for the polygon vertices.

Bilinear interpolation is lerping in 2 dimensions.

Bilinear interpolation
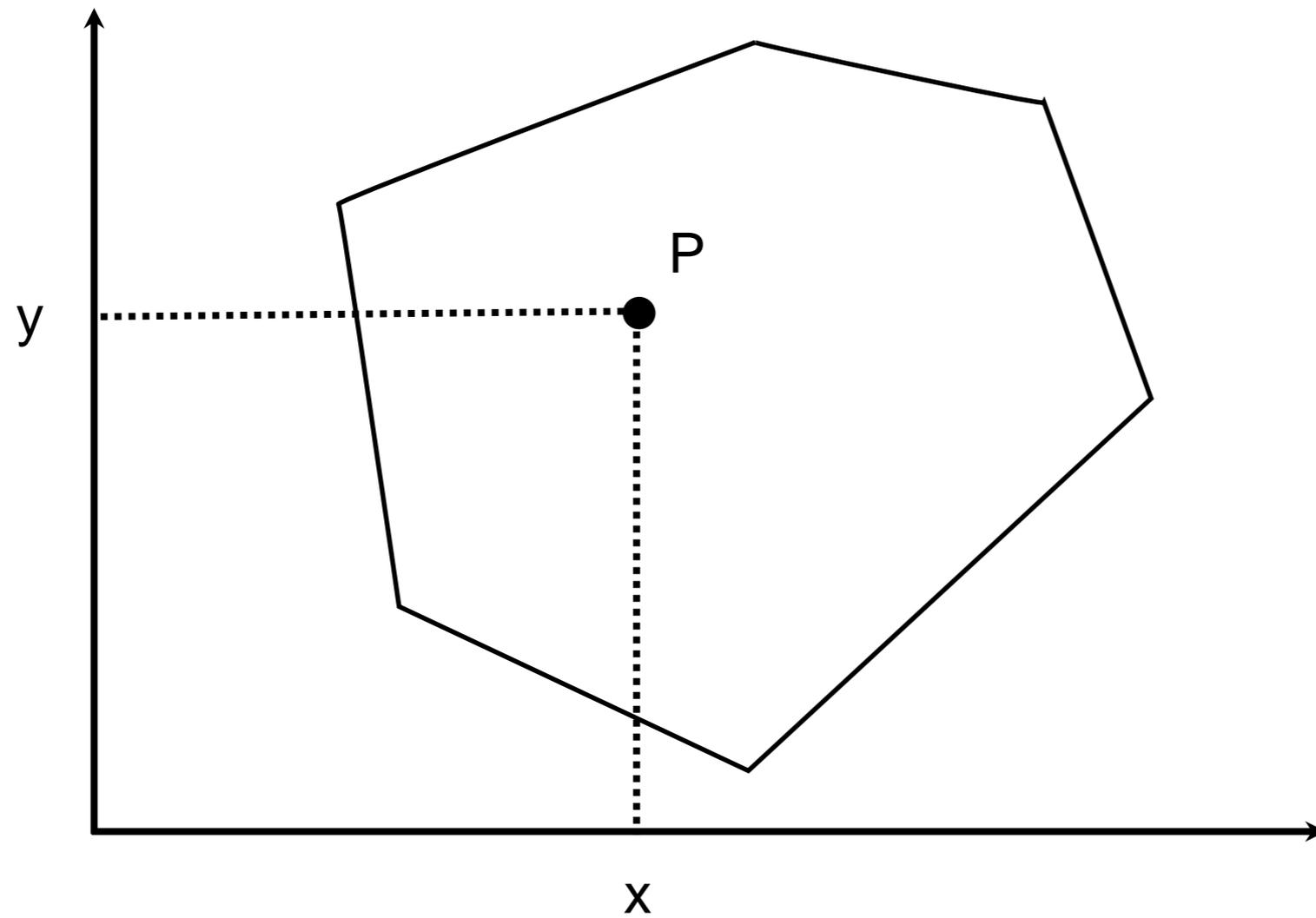
Bilinear interpolation
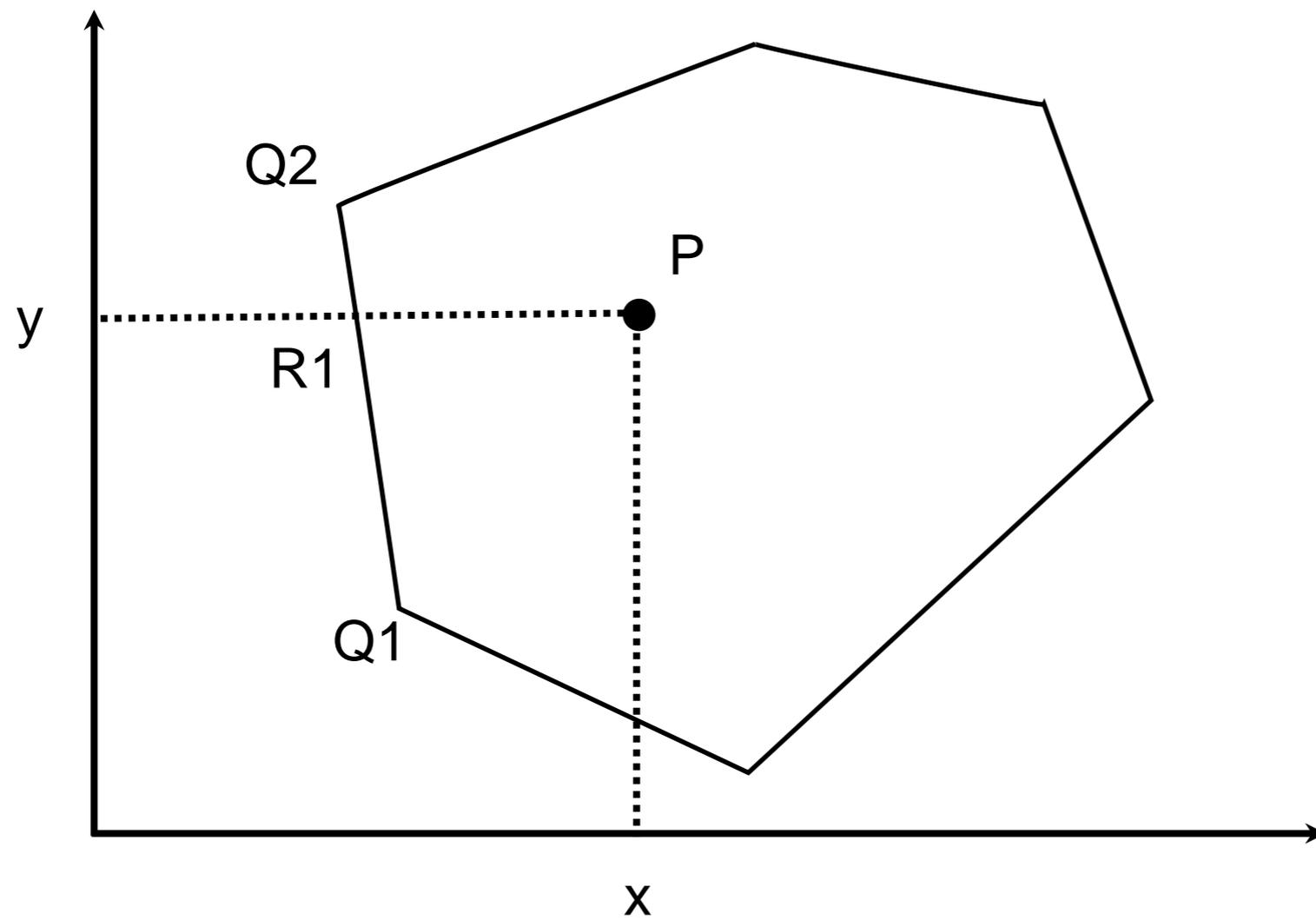
Bilinear interpolation

$$f(R_1) = \frac{y - y_1}{y_2 - y_1} f(Q_2) + \frac{y_2 - y}{y_2 - y_1} f(Q_1)$$
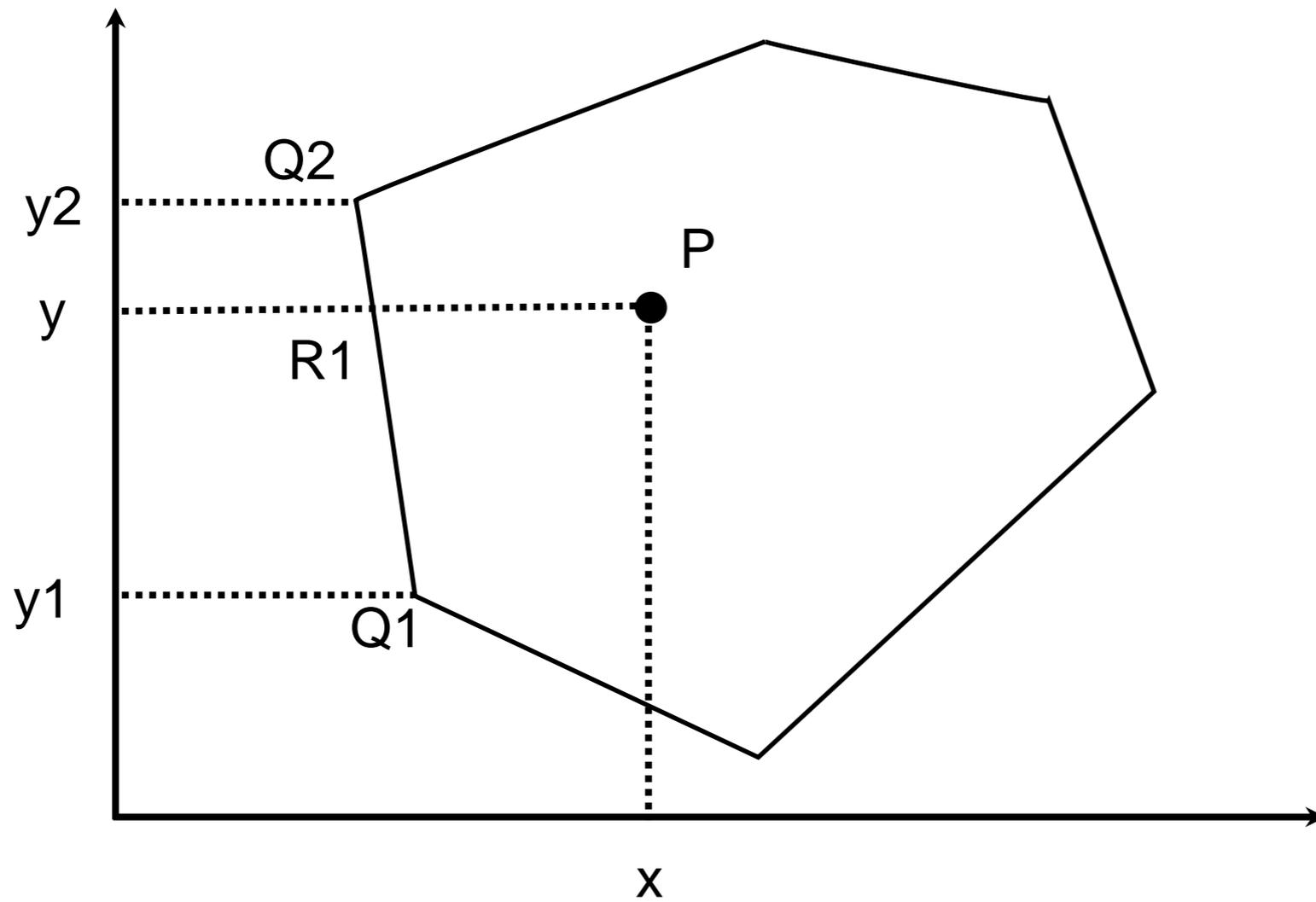
Bilinear interpolation

$$f(R_2) = \frac{y - y_3}{y_4 - y_3} f(Q_4) + \frac{y_4 - y}{y_4 - y_3} f(Q_3)$$
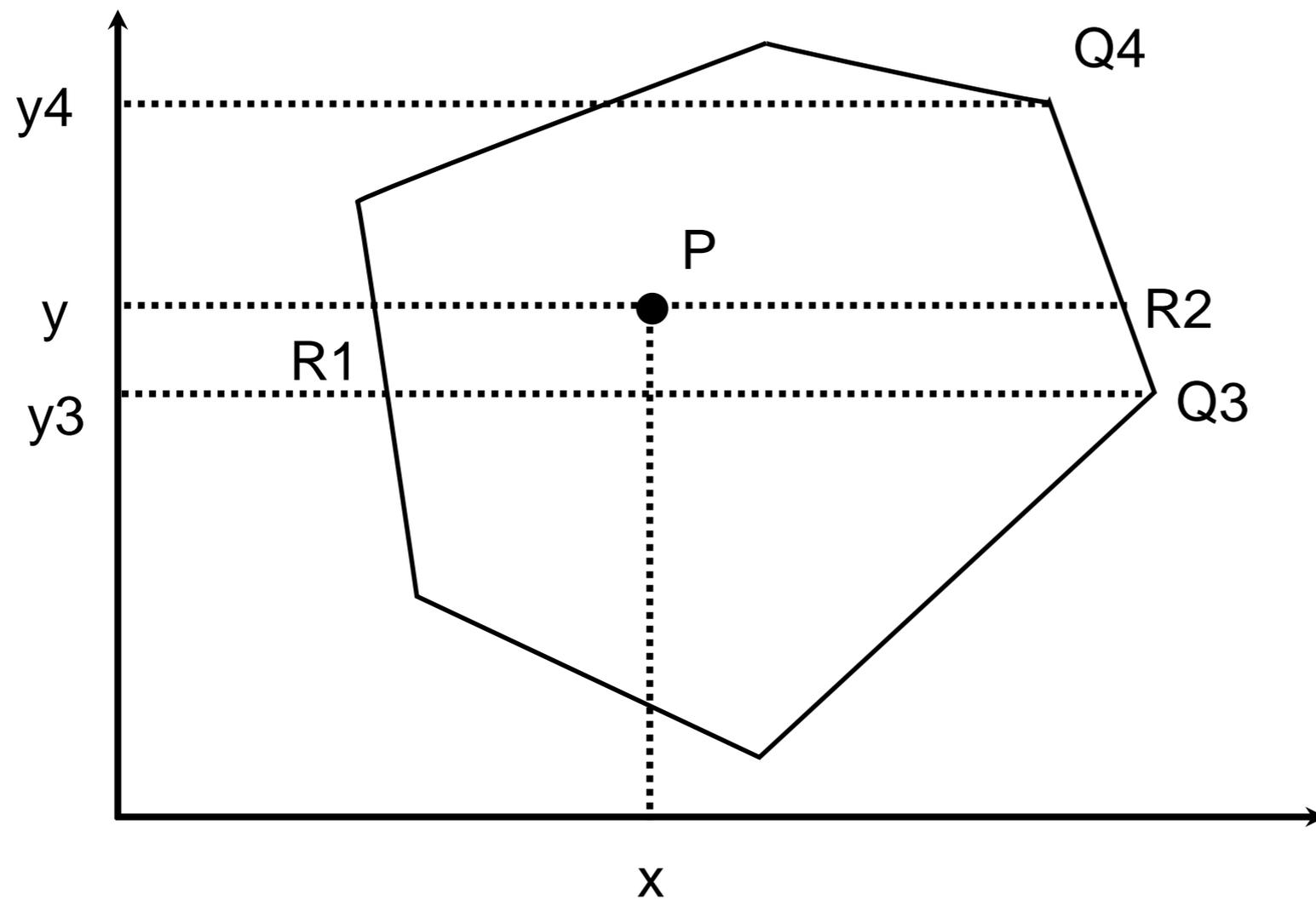
Bilinear interpolation

$$f(P) = \frac{x - x_1}{x_2 - x_1} f(R_2) + \frac{x_2 - x}{x_2 - x_1} f(R_1)$$

(1,7,1)

Depth?

(7,4,0)

(4,2,0.5)

(1,7,1)

(6,5,DR)

(2,5,DL)

Depth?

(7,4,0)
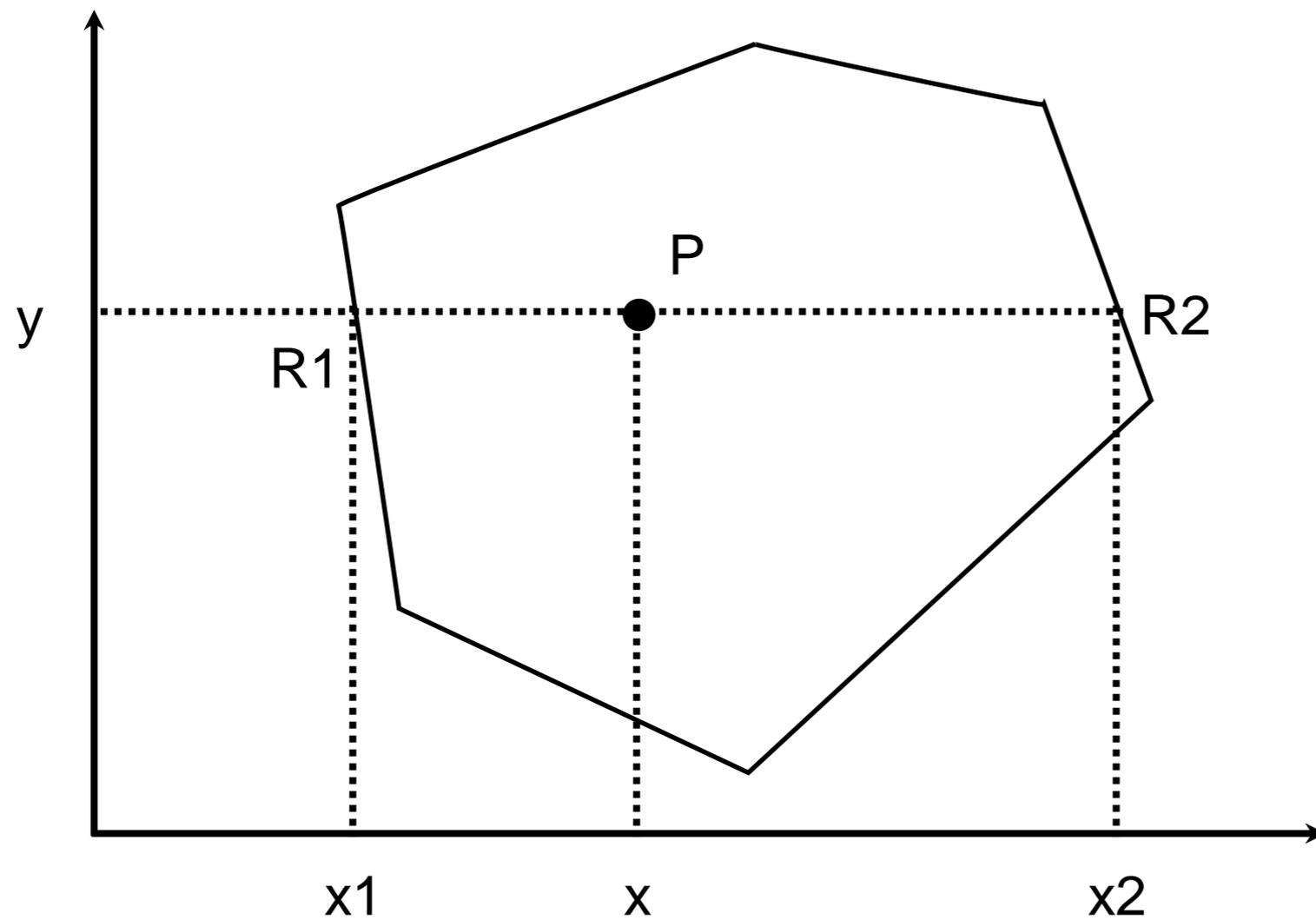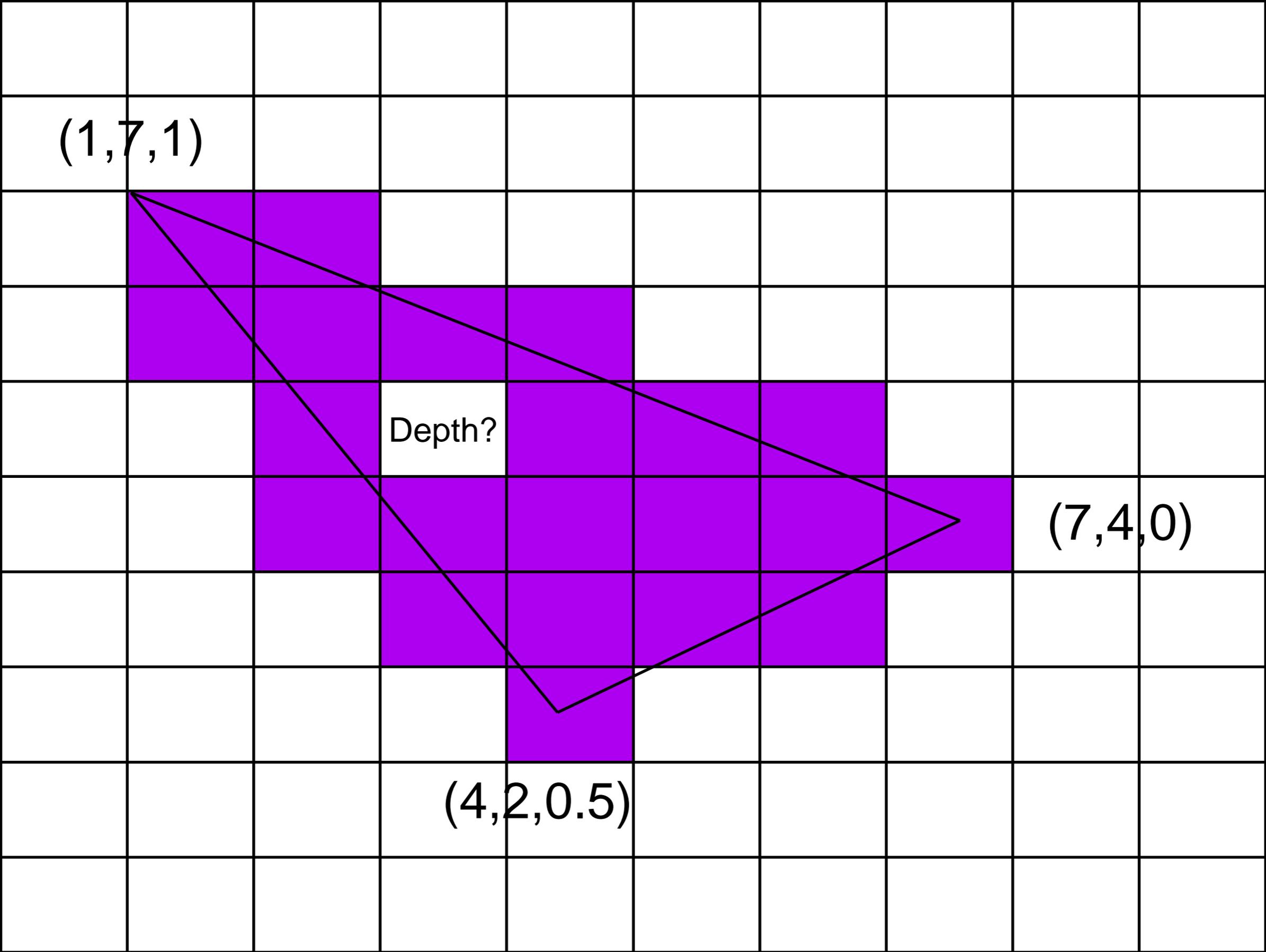
(4,2,0.5)

Interpolation - Y

$$f(R_1) = \frac{y - y_1}{y_2 - y_1} f(Q_2) + \frac{y_2 - y}{y_2 - y_1} f(Q_1)$$

(2,5,DL) Q1(4,2,0.5) Q2(1,7,1)

DL = (5 − 2)/(7-2)*1 + (7 − 5)/(7-2)*0.5

$\quad$ = (3/5)*1 + 2/5*0.5 = 0.8

(6,5,DR) Q1(7,4,0) Q2(1,7,1)

DR = (5-4)/(7-4)*1 + (7-5)/(7-4)*0

$\quad$ = (1/3)*1 + (2/3)*0 = 0.3

(1,7,1)

(6,5,0.3)

(2,5,0.8)   Depth?

(7,4,0)

(4,2,0.5)

Interpolation - X

$$f(P) = \frac{x - x_1}{x_2 - x_1} f(R_2) + \frac{x_2 - x}{x_2 - x_1} f(R_1)$$

(3,5,Depth) R1(2,5,0.8) R2(6,5,0.3)

Depth = (3 − 2)/(6-2)*0.3 + (6-3)/(6-2)*0.8

=1/4*0.3 + 3/4*0.8

= 0.675

(1,7,1)

(6,5,0.3)

(2,5,0.8)

0.675

(7,4,0)

(4,2,0.5)

Z-fighting

The depth buffer has limited precision (usually 16 bits).

If two polygons are (almost) parallel small rounding errors will cause them to "fight" for which one will be in front, creating strange effects.
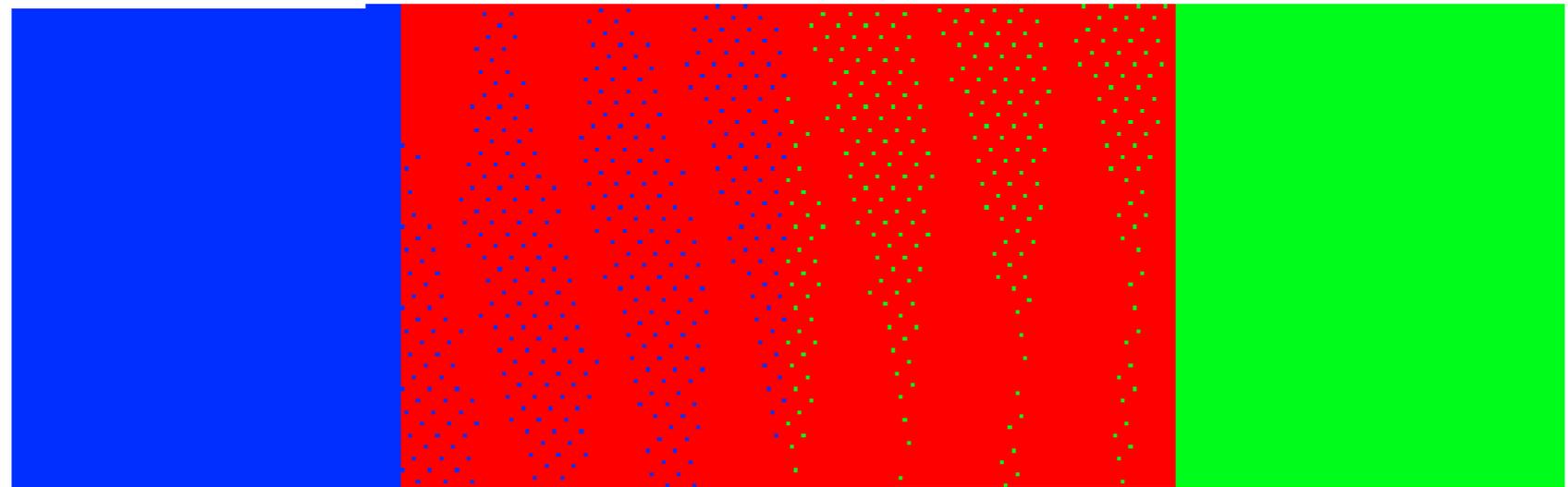
glPolygonOffset

When you have two overlapping polygons you can get Z-fighting.

To prevent this, you can offset one of the two polygons using glPolygonOffset().

This method adds a small offset to the pseudodepth of any vertices added after the call. You can use this to move a polygon slightly closer or further away from the camera.

glPolygonOffset

To use glPolygonOffset you must first enable it. You can enable offsetting for points, lines and filled areas separately:

```
gl.glEnable(
    GL2.GL_POLYGON_OFFSET_POINT);
gl.glEnable(
    GL2.GL_POLYGON_OFFSET_LINE);
gl.glEnable(
    GL2.GL_POLYGON_OFFSET_FILL);
```

glPolygonOffset

Usually you will call this as either:

```
//Push polygon back a bit
gl.glPolygonOffset(1.0, 1.0);
//Push polygon forward a bit
gl.glPolygonOffset(-1.0, -1.0);
```

If this does not give you the results you need play around with values or check the (not very clear) documentation

Z-filling

We can combine BSP-trees and Z-buffering.

We use BSP-trees to represents the static geometry, so the tree only needs to be built once (at compile time).

As the landscape is drawn, we write depth values into the Z-buffer, but we don't do any testing (as it is unnecessary).

Dynamic objects are drawn in a second pass, with normal Z-buffer testing.

Transparency

A transparent (or translucent) object lets some of the light through from the object behind it.
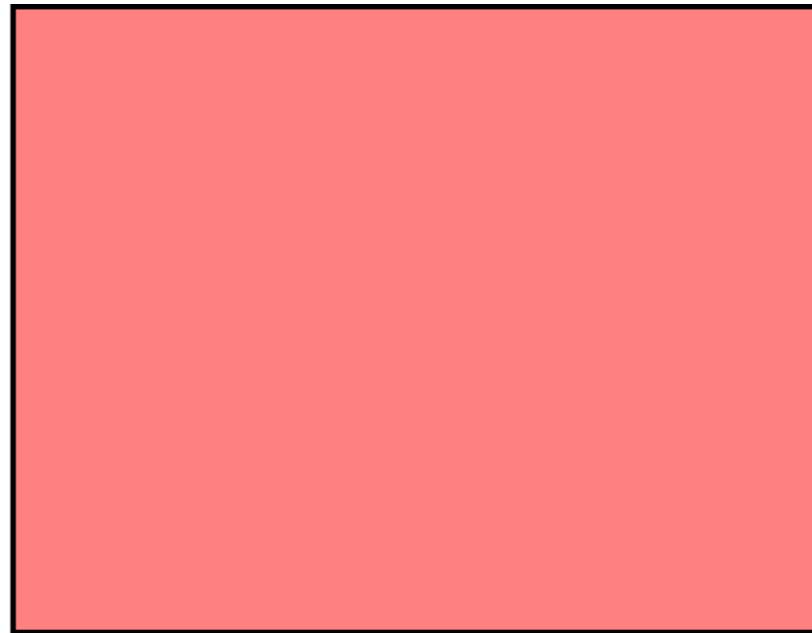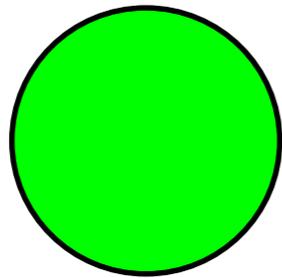
Transparency

A transparent (or translucent) object lets some of the light through from the object behind it.
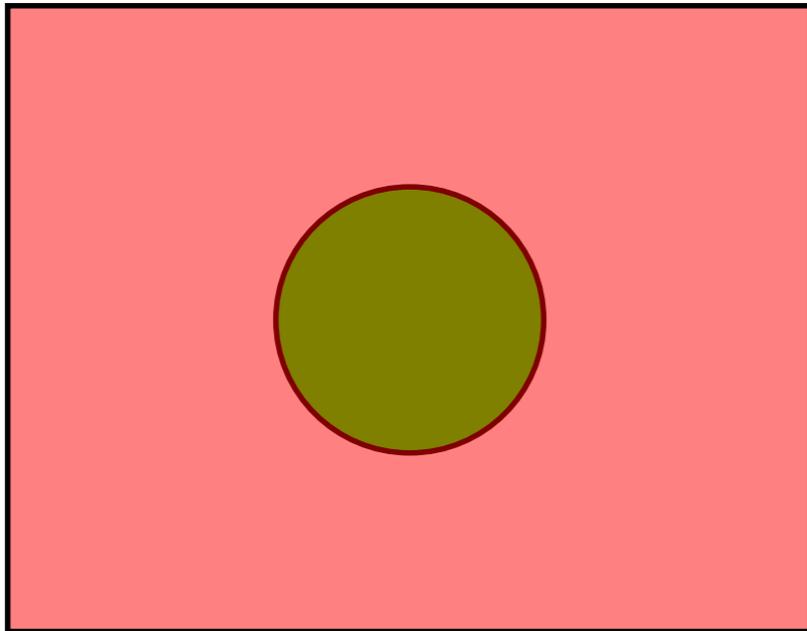
The alpha channel

When we specify colours we have used 4 components:

- red/green/blue

- alpha - the opacity of the colour

alpha = 1 means the object is opaque

alpha = 0 means the object is completely transparent (invisible)

Alpha blending

When we draw one object over another, we can blend their colours according to the alpha value.

There are many blending equations, but the usual one is linear interpolation:

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} \leftarrow \alpha \begin{pmatrix} r_{image} \\ g_{image} \\ b_{image} \end{pmatrix} + (1 - \alpha) \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

Example

If the pixel on the screen is currently green,
and we draw over it with a red pixel,
with alpha = 0.25

$$p = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad p_{image} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0.25 \end{pmatrix}$$

Example

Then the result is a mix of red and green.

$$p = 0.25 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0.25 \end{pmatrix} + 0.75 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0.25 \\ 0.75 \\ 0 \\ 0.8125 \end{pmatrix}$$

OpenGL

```java
// Alpha blending is disabled by
// default. To turn it on:

gl.glEnable(GL2.GL_BLEND);
// other blend functions are
// also available

gl.glBlendFunc(
    GL2.GL_SRC_ALPHA,
    GL2.GL_ONE_MINUS_SRC_ALPHA);
```

Transparency Example

Compare alpha 1 and 0.75 and 0.25 and show it blending with background.

What about green triangle?

Need ordering here!

Now try with different ordering and different values.

Problems

Alpha blending depends on the order that pixels are drawn.

You need to draw transparent polygons after the polygons behind them.

If you are using the depth buffer and you try to draw the transparent polygon before the objects behind it, the later objects will not be drawn.

Back-to-front

Back Polygon

Buffer

Back-to-front

Back Polygon

Buffer

Back-to-front

Front Polygon

.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1

Buffer

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	.5	.5	.5	.5	.5	.5	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Back-to-front

Front Polygon

Buffer

Correct

Front-to-back

Front Polygon

.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1
.1	.1	.1	.1	.1

Buffer

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Front-to-back

Front Polygon

.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					

Buffer

.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1
.1	.1	.1	.1	.1	1	1	1	1	1

Front-to-back

Front Polygon

Buffer

Front-to-back

Front Polygon

Buffer

Wrong

Transparency

Transparent polygons should be drawn in back-to-front order after your opaque ones.

BSP-tree is one solution for this problem.

Other fudges are to draw your transparent polygons after the opaque ones but in any order, but turning off the depth buffer writing for the transparent polygons. This will not result in correct blending, but may be ok.

gl.glDepthMask(false).

Illumination

In this section we will be considering how much light reaches the camera from the surface of an object.

In the OpenGL fixed function pipeline these calculations are performed on vertices and and then interpolated to determine values for fragments/pixels.

If we write our own shaders we can do calculations at the fragment/pixel level.

Achromatic Light

To start with we will consider lighting equations for achromatic light which has no colour, but simply a brightness.

We will then extend this to include coloured lights and coloured objects. The computations will be identical but will have separate intensities of red, blue and green calculated.

Local Illumination

Interreflections: In real life light reflects from a light off one object onto another object etc

So objects with no direct light are not completely in darkness

This is very costly to model.

In local illumination we only model reflections directly from a light source….

(and then add a fudge factor)

Illumination

The colour of an object in a scene depends on:

- The colour and amount of light that falls on it.

- The colour and reflectivity of the object

There are two kinds of reflection we need to deal with: diffuse and specular.

Diffuse reflection

Dull or matte surfaces exhibit diffuse reflection. Light falling on the surface is reflected uniformly in all directions. It does not depend on the viewpoint.

incident
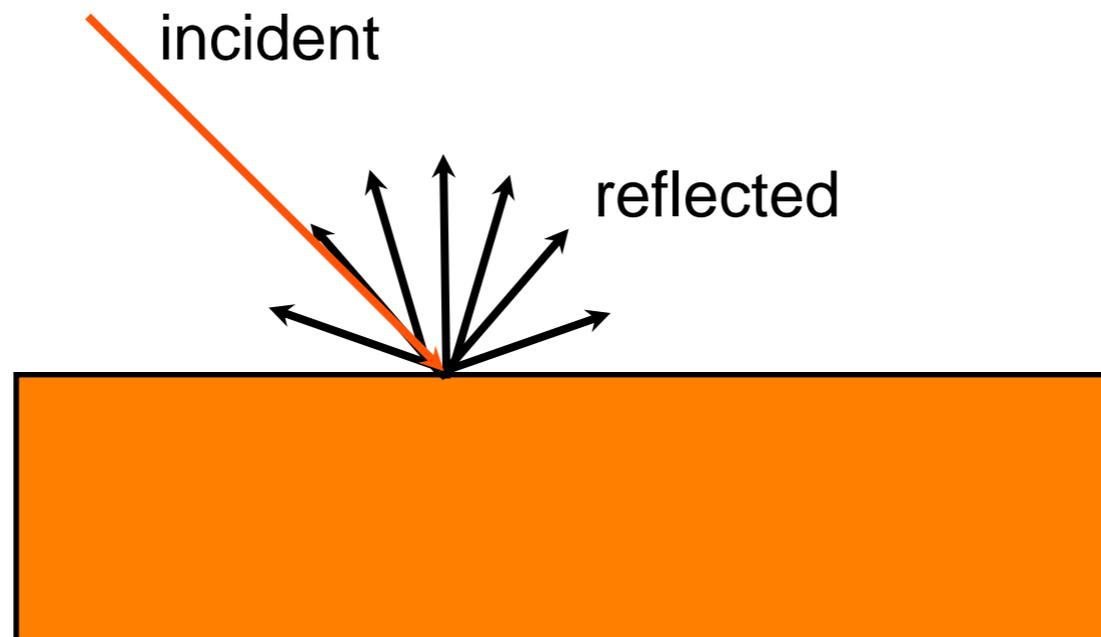
reflected

Specular reflection

Polished surfaces exhibit specular reflection. Light falling on the surface is reflected at the same angle. Reflections will look different from different view points.
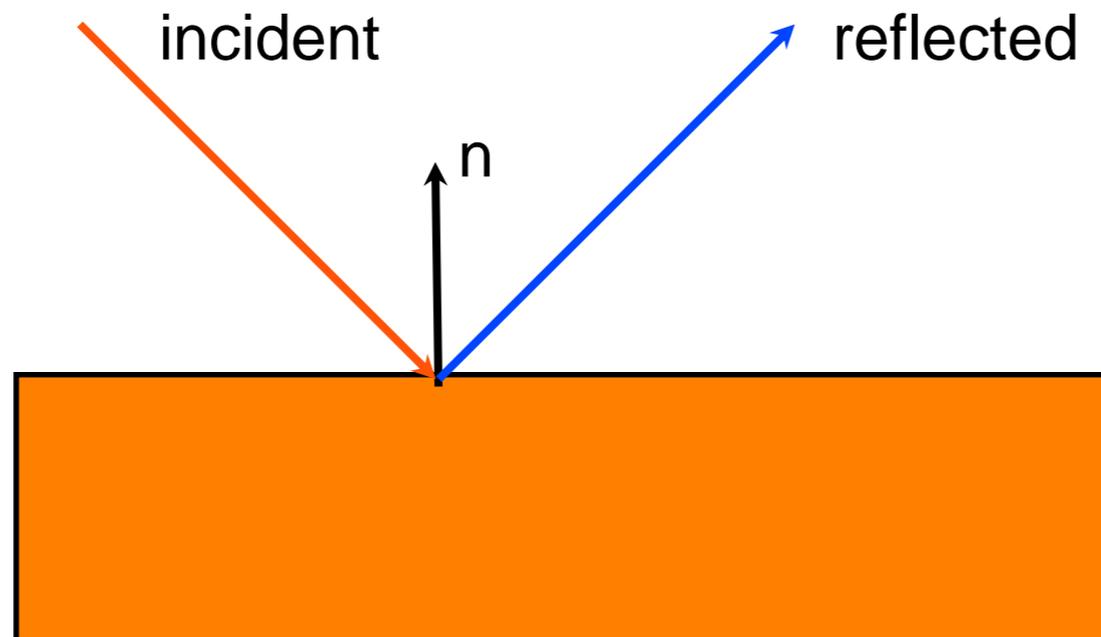
Components

Most objects will have both a diffuse and a specular component to their illumination.

We will also include an ambient component to cover lighting from indirect sources.

We will build a lighting equation:

$$I(P) = I_{ambient}(P) + I_{diffuse}(P) + I_{specular}(P)$$

I(P) is the amount of light coming from P to the camera.

Ingredients

To calculate the lighting equation we need to know three important vectors:

- The normal vector $\mathbf{m}$ to the surface at P

- The view vector $\mathbf{v}$ from P to the camera

- The source vector $\mathbf{s}$ from P to the light source.
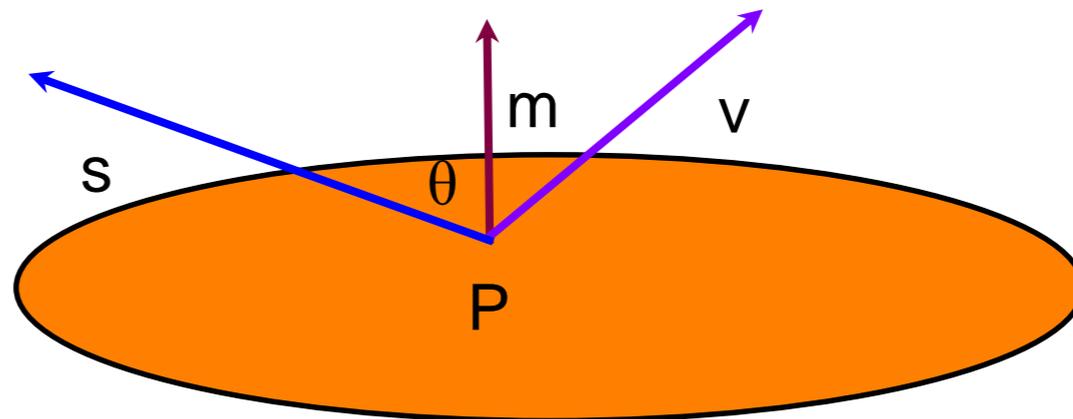
Diffuse illumination

Diffuse scattering is equal in all directions so does not depend on the viewing angle.

The amount of reflected light depends on the angle of the source of the light

Small angle
= small lit area
= higher intensity

Large angle
= larger lit area
for same light
=lower intensity
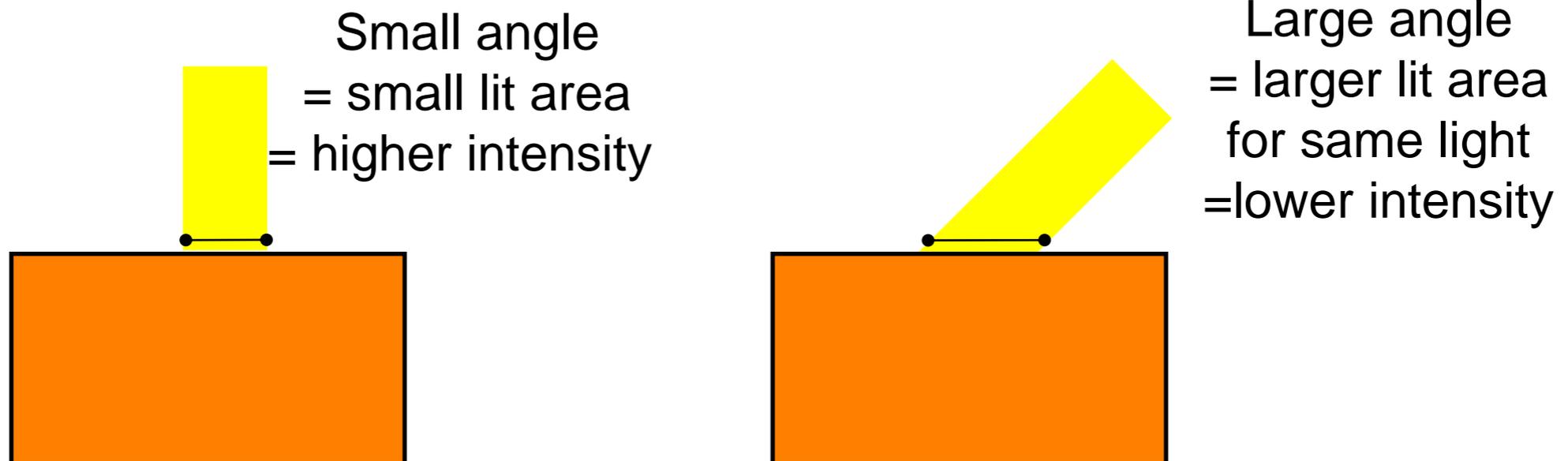
Lambert's Cosine Law

We can formalise this as Lambert's Law:

$$I_d \quad \propto \quad I_s \cos\theta$$
$$= \quad I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})$$

where:

I_s is the source intensity, and

ρ_d is the diffuse reflection coefficient in (0,1)

Note: both vectors are normalised
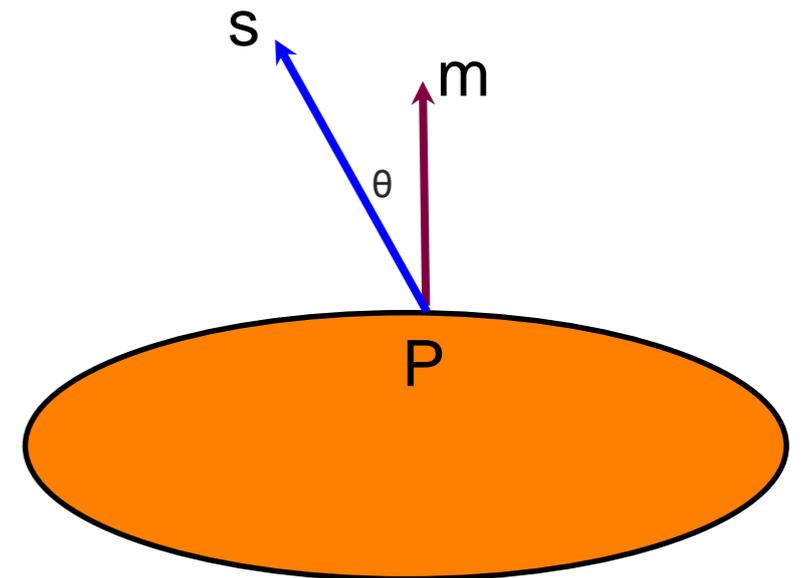
Lambert's Cosine Law

When the angle is 0 degrees the cosine is 1

You get all the reflected light back

When the angle is 90 degrees

None of the light is reflected back

When the angle is > 90 degrees

cos gives us a negative value! This is not what we want.

Lambert's Law

If the angle > 90, then the light is on the wrong side of the surface and the cosine is negative. In this case we want the illumination to be zero. So:

$$I_d = \max\left(0, I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})\right)$$
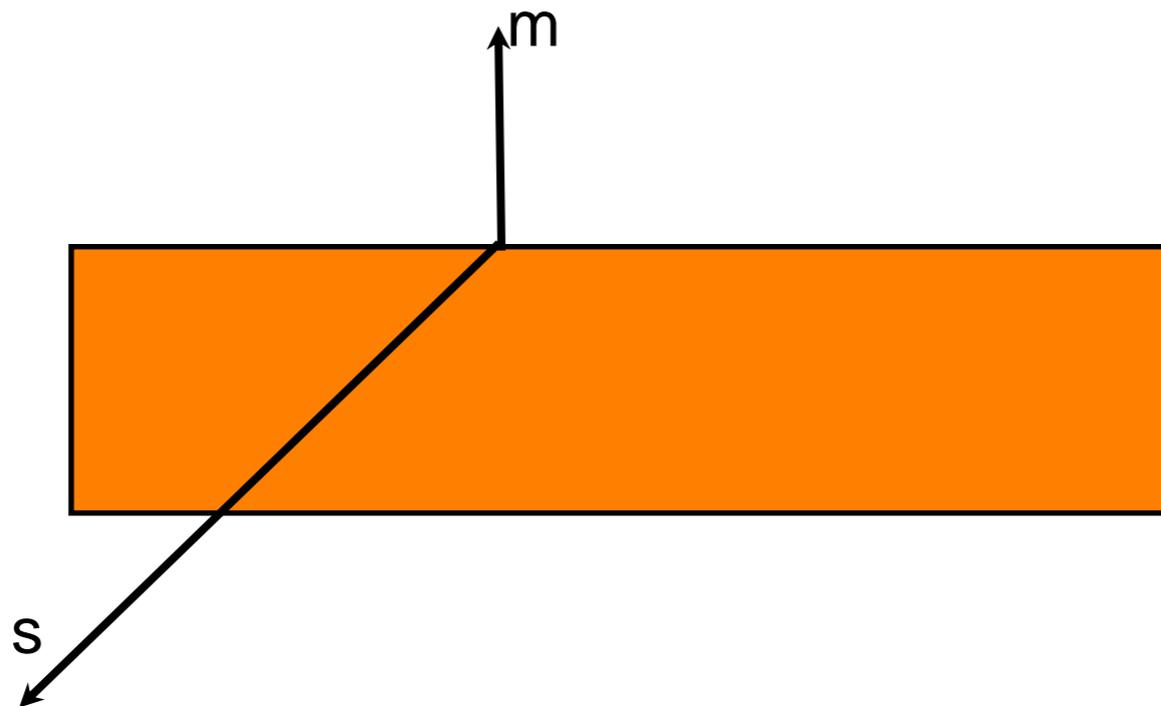
Diffuse reflection coefficient

The coefficient ρ_d is a property of the surface.

•	Light surfaces have values close to 1 as they reflect more light

•	Dark surfaces have values close to 0 as they absorb more light

In reality the coefficient varies for different wavelengths of light so we would have 3 separate values for R, G and B.

Specular reflection

Only mirrors exhibit perfect specular reflection. On other surfaces there is still some scattering.
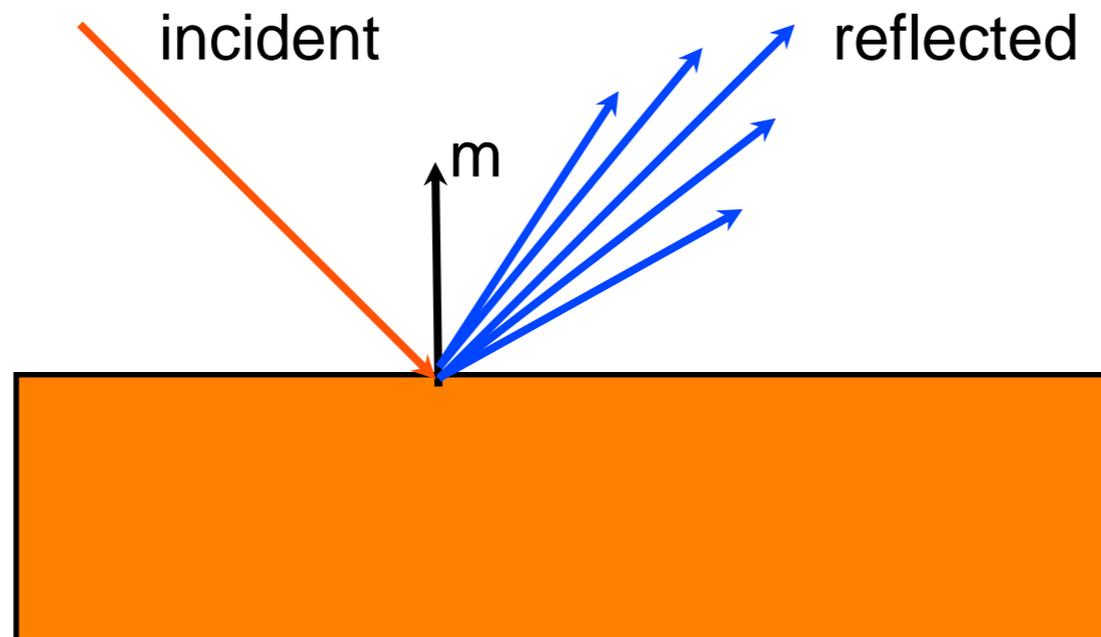
Phong model

The Phong model is an approximate model (not correct physics) of specular reflection. It allows us to add highlights to shiny surfaces.

It looks good for plastic and glass but not good for polished metal (in which real reflections are visible).

Phong model

Reflection is brightest around the reflection vector:
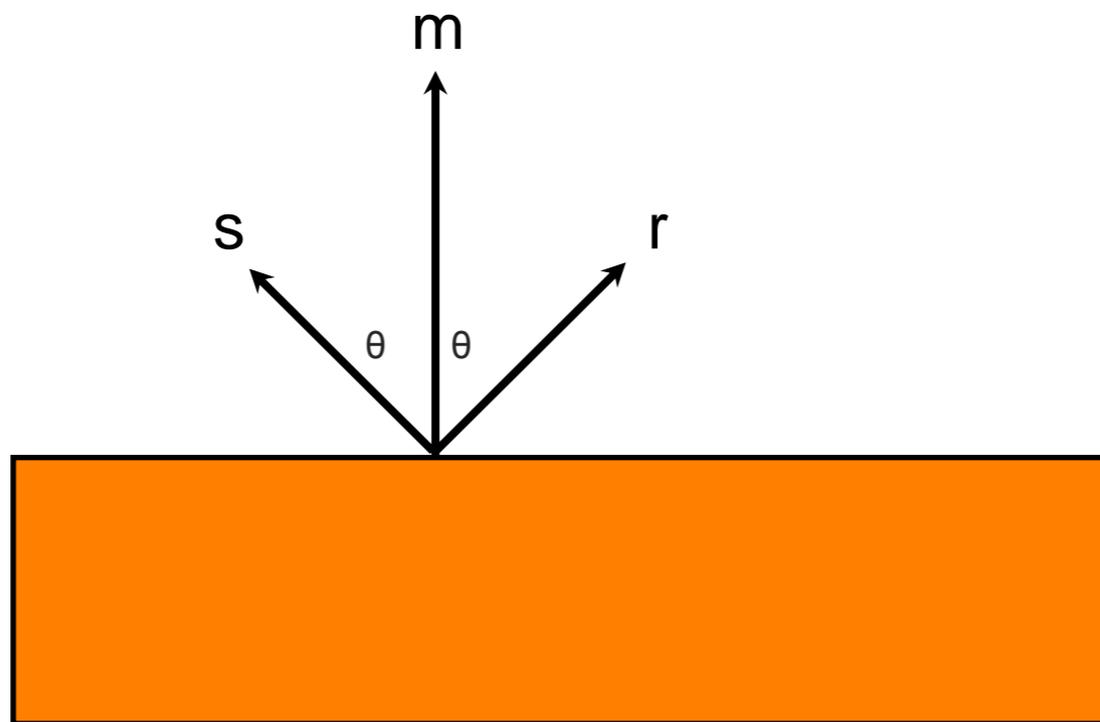
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$

Phong model

Reflection is brightest around the reflection vector :
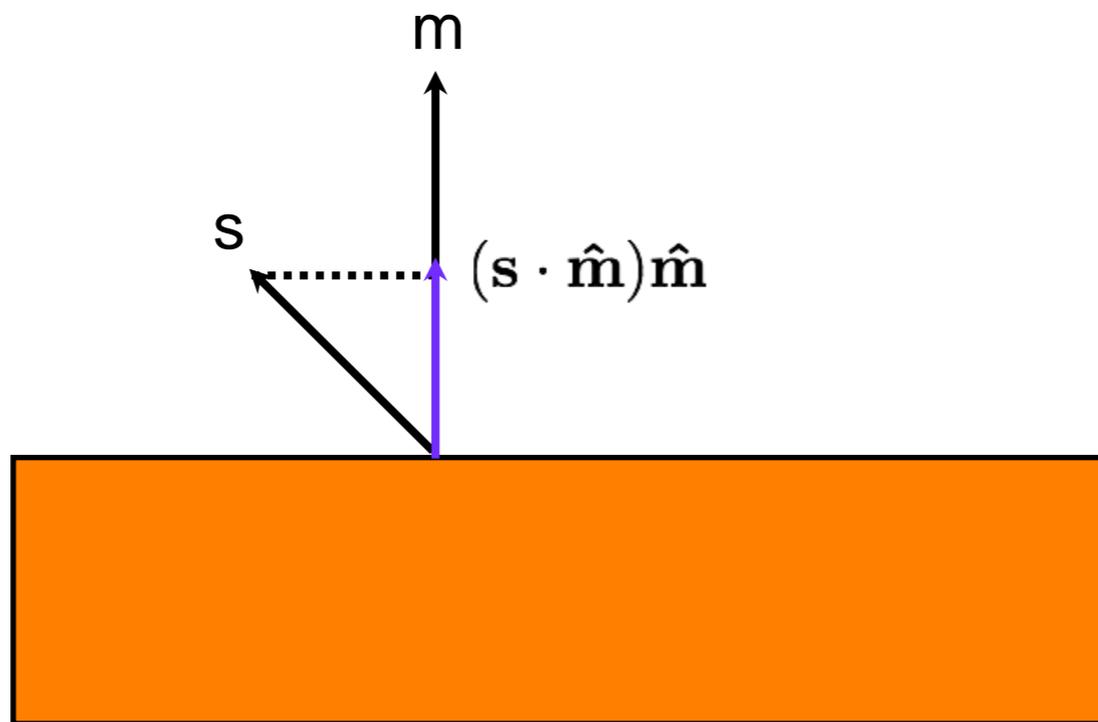
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$

Phong model

Reflection is brightest around the reflection vector:
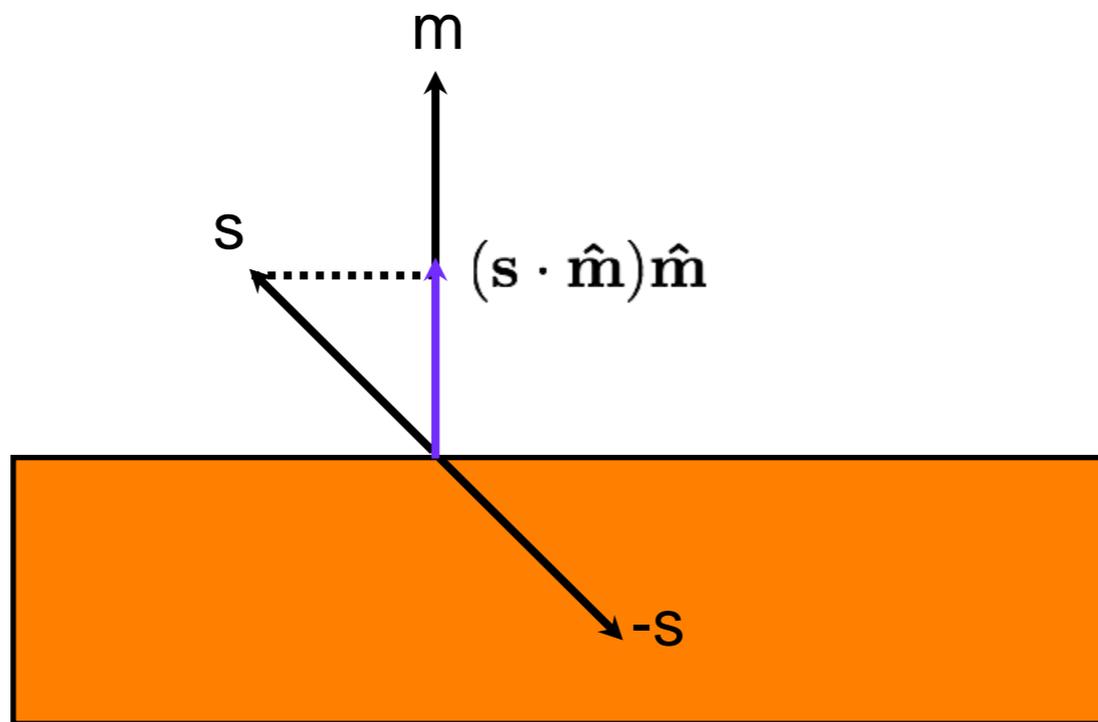
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$

Phong model

Reflection is brightest around the reflection vector:
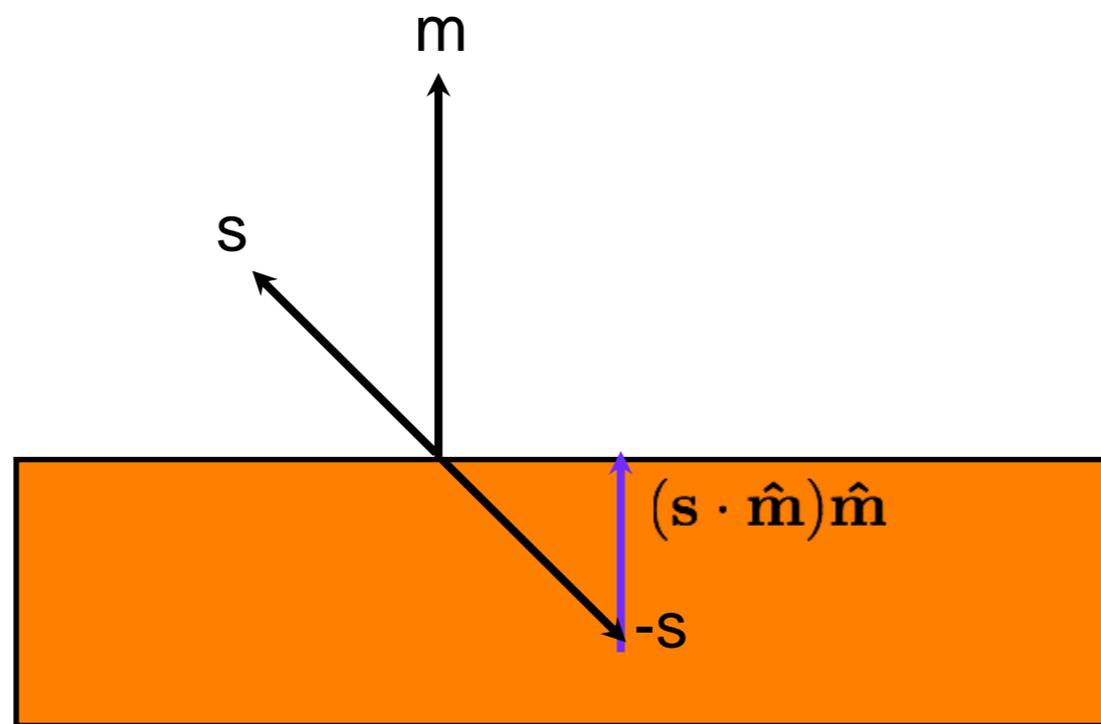
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$

Phong model

Reflection is brightest around the reflection vector:

$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$

Phong model

The intensity falls off with the angle Φ between the reflected vector and the view vector.
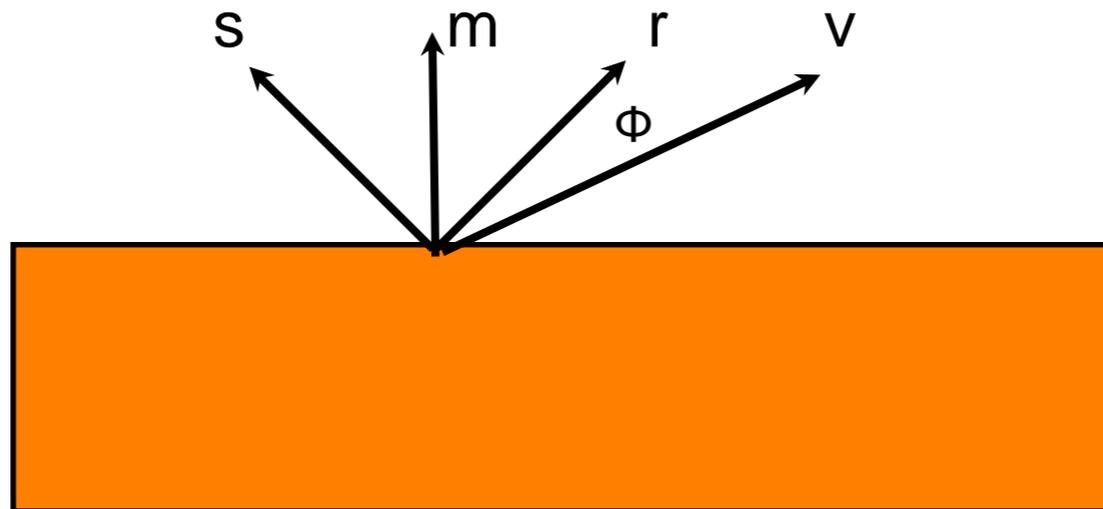
Phong model

The Phong equation is:

$$I_{sp} \propto I_s \, (\cos \Phi)^f$$

$$= \max(0, \, I_s \rho_{sp} \, (\widehat{\boldsymbol{r}} . \, \widehat{\boldsymbol{v}})^f)$$

where:

ρ_{sp} is the specular reflection coefficient
in the range (0,1)

f is the Phong exponent
in the range (0,128)

Phong exponent

Larger values of the Phong exponent f make $\cos(\Phi)^f$ smaller, produce less scattering, creating more mirror-like surfaces.
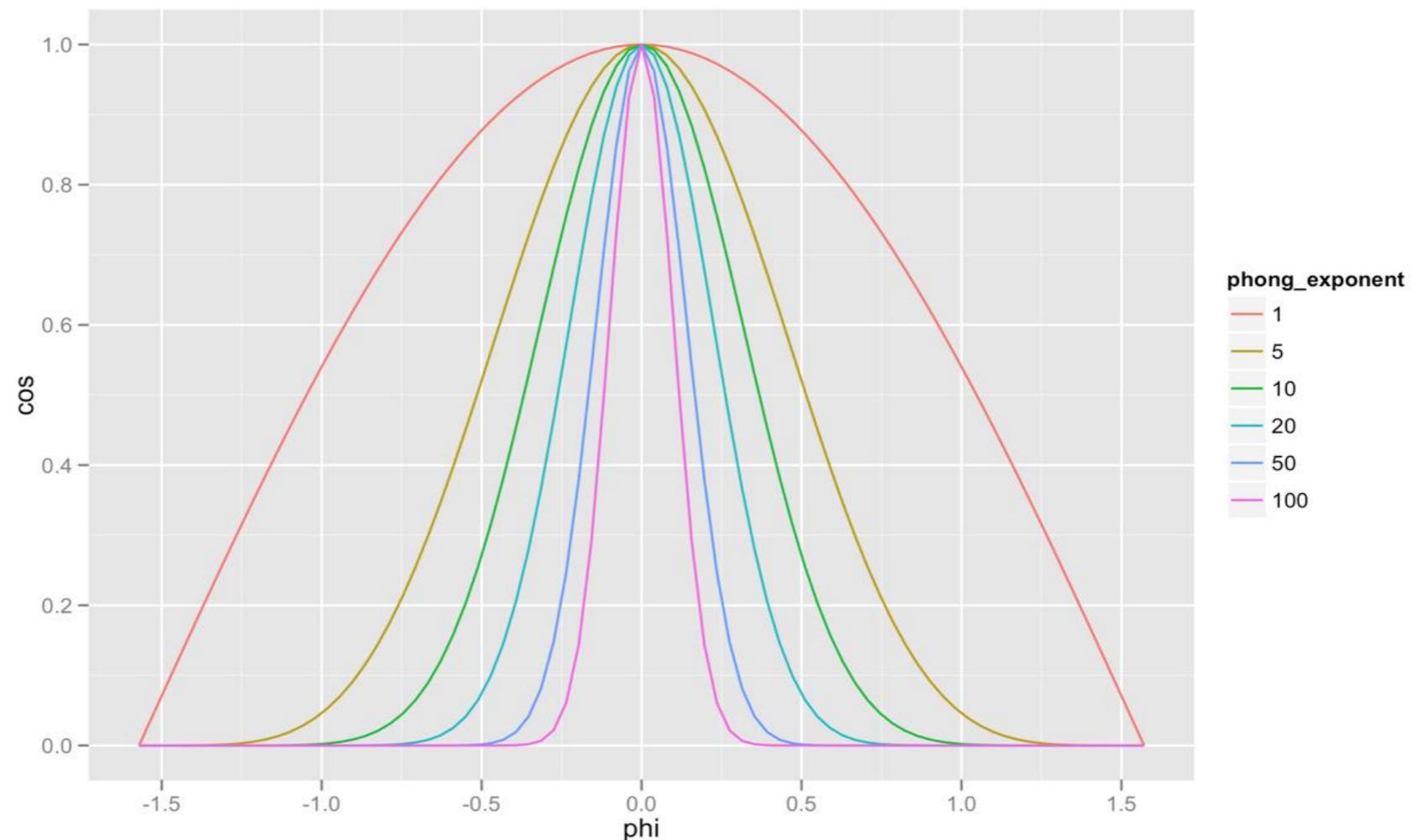
Blinn Phong Model

Opengl fixed function pipeline uses a slight variant on the Phong model for specular highlights

The Blinn Phong model uses a vector halfway between the source and the viewer instead of calculating the reflection vector.

Blinn-Phong Specular Light

Note: where s and v are normalised

Find the halfway vector $\hat{h}$ = (**s** + **v**)/|**s+v**|

Then the angle β between **h** and **m** approximately measures the falloff of intensity.

$$I_{sp} = \max(0, I_s\, \rho_s\, (\hat{h}\cdot\hat{m})^f)$$
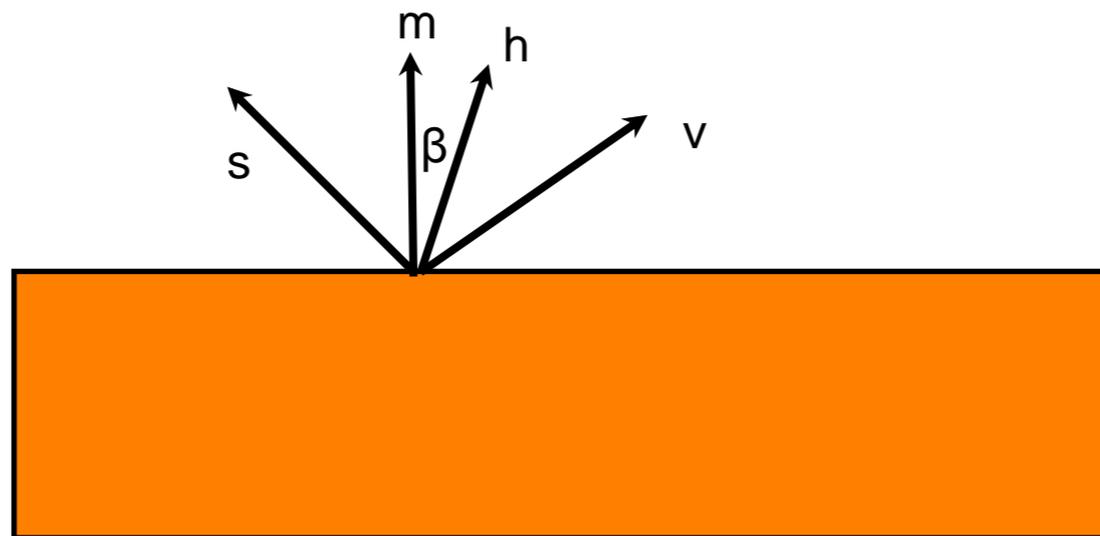
Reflection

Note that the Phong/ Blinn Phong model only reflects light sources, not the environment.

It is good for adding bright highlights but cannot create a true mirror.

Proper reflections are more complex to compute (as we'll see later).

Ambient light

Lighting with just diffuse and specular lights gives very stark shadows.

In reality shadows are not completely black.

Light is coming from all directions, reflected off other objects, not just from 'sources'

It is too computationally expensive to model this in detail.

Ambient light

The solution is to add an ambient light level to the scene for each light:

$$I_{ambient} = I_a \rho_a$$

where:

I_a is the ambient light intensity
ρ_a is the ambient reflection coefficient
in the range (0,1) (usually $\rho_a = \rho_d$)

And also to add a global ambient light level to the scene

Emissive Light

The emissive component of light from an object O is that which is unrelated to an external light source. This is similar to just giving an object a set colour.

Emissive light is perceived only by the viewer and does not illuminate other objects

Combining Light Contributions

For a particular light source at a vertex

I = ambient + diffuse + specular

$I = I_a\rho_a + I_d\rho_d * \text{lambert} + I_{sp}\rho_{sp*}\text{blinnPhong}$

Where

$\text{lambert} = \max(0, \hat{s} \cdot \hat{m})$

$\text{blinnPhong} = \max(0, \hat{h} \cdot \hat{m})$

Combining all Light Sources

We combine the emissive light from the object, global ambient light, and the ambient, diffuse and specular components from multiple light sources.

$$I = \rho_e + I_{ga}\rho_a + \sum_{l \in lights} (I_a\rho_a + I_d\rho_d * \text{lambert} + I_{sp}\rho_{sp*}\text{blinnPhong})$$

emissive light

global ambient term

Limitations

It is only a local model.

Colour at each vertex V depends only on the interaction between the light properties and the material properties at V

It does **not** take into account

- whether V is obscured from a light source by another object or shadows

- light that strikes V not having bounced off other objects (reflections and secondary lighting).

Colour

We implement colour by having separate red, green and blue components for:

- Light intensities I_a I_l

- Reflection coefficients ρ_a ρ_d ρ_{sp}

The lighting equation is applied three times, once for each colour.

Colored Light and surfaces

$$I_r = \rho_{er} + I_{gar}\rho_{ar} +$$
$$\sum_{l \in lights} (I_{ar}\rho_{ar} + I_{dr}\rho_{dr} * \text{lambert} + I_{spr}\rho_{spr*}\text{blinnPhong})$$

$$I_g = \rho_{eg} + I_{gag}\rho_{ag} +$$
$$\sum_{l \in lights} (I_{ag}\rho_{ag} + I_{dg}\rho_{dg} * \text{lambert} + I_{spg}\rho_{spg*}\text{blinnPhong})$$

$$I_b = \rho_{eb} + I_{gab}\rho_{ab} \text{ etc....}$$
$$\sum_{l \in lights} (I_{ab}\rho_{ab} + I_{db}\rho_{db} * \text{lambert} + I_{spb}\rho_{spb*}\text{blinnPhong})$$

Caution

Using too much/many lights can result with colour components becoming 1 (if they add up to more than 1 they are clamped).

This can result in things changing 'colour' and/or turning white.

OpenGL

OpenGL supports at least 8 light sources.

Refer to them by the constants:

`GL_LIGHT0, GL_LIGHT1,`
`GL_LIGHT2, ...`

Note:

`GL_LIGHT1 == GL_LIGHT0 + 1`

Default Light Settings

Default Position: `(0,0,1,0)`

Default Ambient co-efficient: `(0,0,0,1)`

`GL_LIGHT0:`

Default diffuse/specular: `(1,1,1,1)`

All other lights:

Default diffuse/specular: `(0,0,0,1)`

Enabling Lighting

```
// enable lighting
gl.glEnable(GL2.GL_LIGHTING);
// enable individual lights
gl.glEnable(GL2.GL_LIGHT0);
gl.glEnable(GL2.GL_LIGHT1);
//etc
```

Global Ambient

```
//This sets global ambient
lighting

float[] amb =
    {0.1f, 0.2f, 0.3f, 1.0f};

gl.glLightModelfv(
  GL_LIGHT_MODEL_AMBIENT, amb, 0);
```

Setting light intensities

```
float[] amb = {0.1f, 0.2f, 0.3f, 1.0f};
float[] dif = {1.0f, 0.0f, 0.1f, 1.0f};
float[] spe = {1.0f, 1.0f, 1.0f, 1.0f};

gl.glLightfv(
    GL_LIGHT0, GL_AMBIENT, amb, 0);

gl.glLightfv(
    GL_LIGHT0, GL_DIFFUSE, dif, 0);

gl.glLightfv(
    GL_LIGHT0, GL_SPECULAR,spe, 0);
```

Material Properties

```
float[] diffuseCoeff =
   {0.8f, 0.2f, 0.0f, 1.0f};

gl.glMaterialfv( GL2.GL_FRONT,
GL2.GL_DIFFUSE, diffuseCoeff,
0);
// all subsequent vertices have
//this property similarly for
// GL_AMBIENT GL_SPECULAR and
// GL_EMISSION
```

Material Properties

```
// the Phong exponent is called
// shininess.

float phong = 10f;

gl.glMaterialf( GL2.GL_FRONT,
GL2.GL_SHININESS, phong);
```

Material Properties

Material properties can be set on a vertex-by-vertex basis if desired:

```
gl.glBegin(GL2.GL_POLYGON);

  gl.glMaterialfv(GL2.GL_FRONT, GL2.GL_DIFFUSE, red, 0);

  gl.glVertex3d(0,0,0);

  gl.glMaterialfv(GL2.GL_FRONT, GL2.GL_DIFFUSE, blue, 0);

  gl.glVertex3d(1,0,0);

  // etc
gl.glEnd();
```

Alpha Values

- One confusing thing is that each of the colour components (Ambient,Diffuse, Specular and Emission) for lights and materials have an associated 'alpha' component for setting transparency.

- Only the **diffuse** colour's alpha value of the **material** actually determines the transparency of the polygon.

Point and directional lights

We have assumed so far that lights are at a point in the world, computing the source vector from this.

These are called point lights

Point light
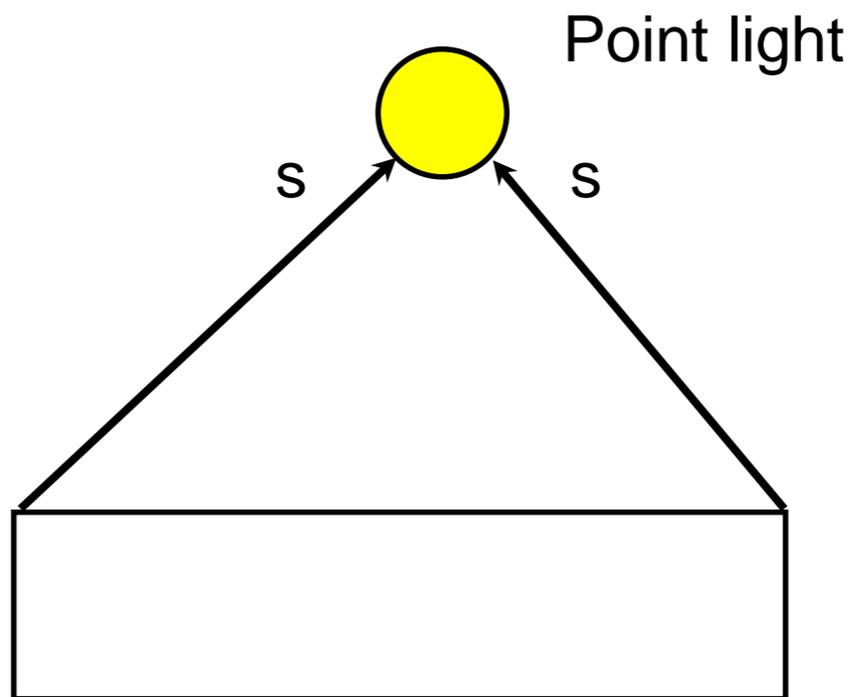
Positional lights

```
// set the position to (5,2,3) in the
// current coordinate frame
// Note: we use homogeneous coords
// By using a 1 we are specifying a
// a point or position.

float[] pos = {5,2,3,1};

gl.glLightfv(GL2.GL_LIGHT0,
             GL2.GL_POSITION,
             pos, 0);
```

Directional lights

Some lights (like the sun) are so far away that the source vector is effectively the same everywhere.

These are called directional lights.

Directional
light
s   s
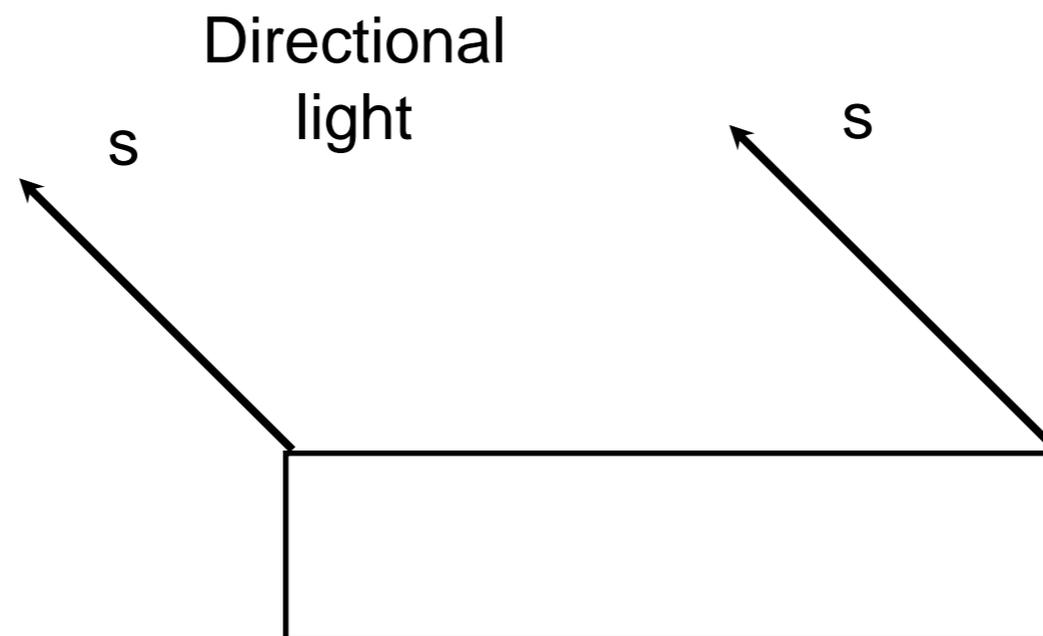
Point and directional lights

We represent a directional light by specifying its position as a vector rather than a point:

```
// The direction is TO the light
source so it is shining down
// note: the fourth component is 0

float[] dir = {0, 1, 0, 0};

gl.glLightfv(GL2.GL_LIGHT0,
          GL2.GL_POSITION, dir, 0);
```

Distance attenuation

All real light sources also lose intensity with distance. This is ignored in the default equations.

But OpenGL does support an attenuation equation:

$$I(d) = \frac{I_s}{k_c + k_l d + k_q d^2}$$

By default $k_c = 1$, $k_l = 0$, $k_q = 0$
which means no attenuation.

OpenGL

```
// kc = 2, kl = 1, kq = 0.5
gl.glLightf(GL2.GL_LIGHT0,
  GL2.GL_CONSTANT_ATTENUATION, 2);

gl.glLightf(GL2.GL_LIGHT0,
  GL2.GL_LINEAR_ATTENUATION, 1);

gl.glLightf(GL2.GL_LIGHT0,
  GL2.GL_QUADRATIC_ATTENUATION, 0.5);
```

LocalViewer

By default OpenGL does not calculate the true viewing angle and uses a default vector of v = (0,0,1).

For more accurate calculations for specular highlights with trade-off of decreased performance use the setting

```
gl.glLightModeli(GL2.GL_LIGHT_MODEL
_LOCAL_VIEWER, GL2.GL_TRUE);
```

Spotlights

Point sources emit light equally in all directions.

For sources like headlights or torches it is more appropriate to use a spotlight.

A spotlight has a direction and a cutoff angle,
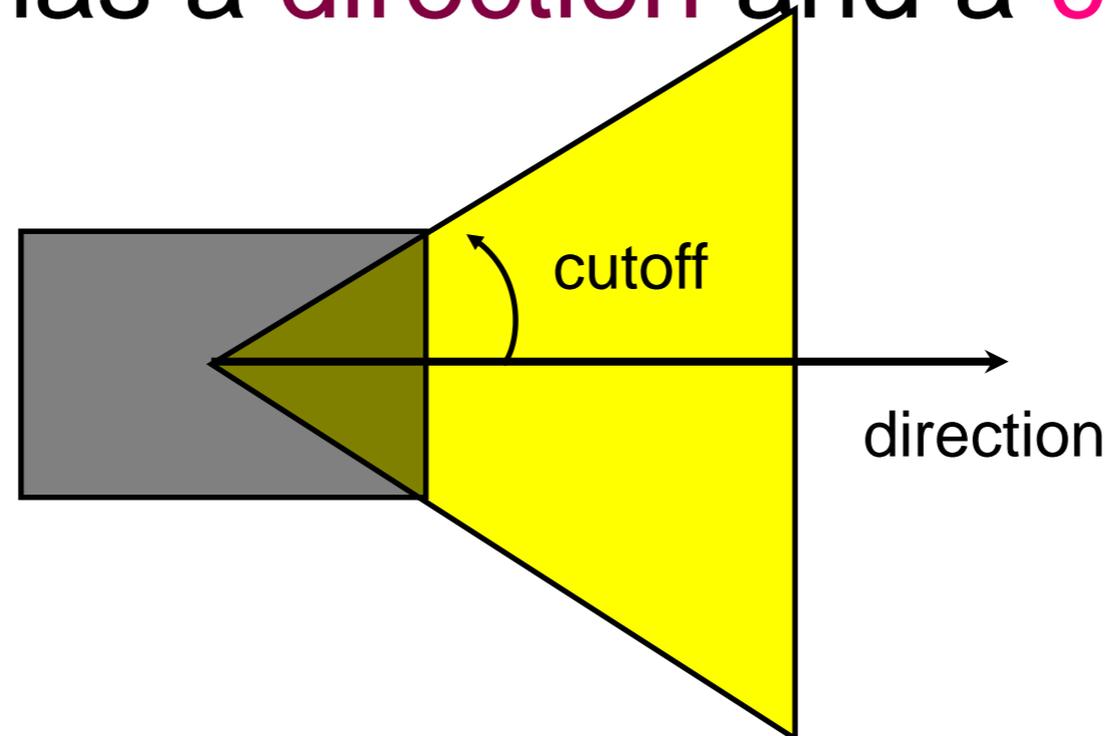
Spotlights

Spotlights are also attenuated, so the brightness falls off as you move away from the centre.

$$I = I_s(\cos(\beta))^\varepsilon$$

where ε is the attenuation factor
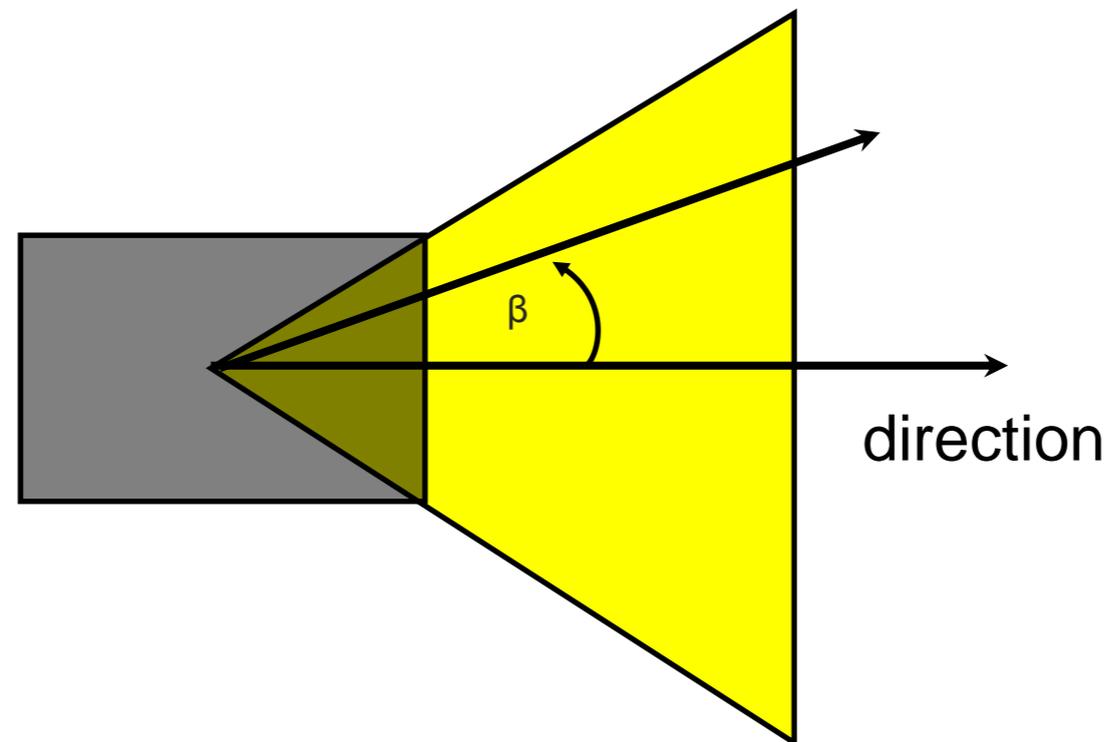
OpenGL

```
// create a spotlight
// with 45 degree cutoff

gl.glLightf(GL2.GL_LIGHT0,
        GL2.GL_SPOT_CUTOFF, 45);

gl.glLightf(GL2.GL_LIGHT0,
        GL2.GL_SPOT_EXPONENT, 4);

gl.glLightfv(GL2.GL_LIGHT0,
    GL2.GL_SPOT_DIRECTION, dir, 0);
```

Sphere in a Box

Where to set lighting position.

Moving Lights

To make the light move with the camera like a miner's light set its position to (0,0,1,0) or (0,0,0,1) (for example) while the modelview transform is the identity (before setting the camera transformations)

To make the light fixed in world co-ordinates, set the position after the viewing transform has been applied and before any modeling transform is applied.

Moving Lights

To make a light move with an object in the scene make sure it is subject to the same modelling transformation as the object

Warning: Before any geometry is rendered, all the light sources that might affect that geometry must already be configured and enabled and the lights' positions must be set be set)

Exercise 1

Suppose we have a sphere with

Diffuse and ambient (1,0,0,1)

Specular (1,1,1,1)

What color will the sphere appear to be with a light with

Specular,diffuse(1,1,1,1) and no ambient?

What color will its specular highlights be?

Exercise 2

Suppose we have a sphere with

Diffuse and ambient (1,0,0,1)

Specular (1,1,1,1)

What color will the sphere appear to be with a light with

Specular,diffuse(0,0,1,1) and no ambient?

What color will its specular highlights be?

Exercise 3

Take the sphere in a box example and make the following changes:

Make the box your favourite colour inside and outside

Make the sphere look metallic in a colour of your choice

Add another positional light somewhere in the world and make it a yellow/orange light. Make it switch on and off with a keypress.