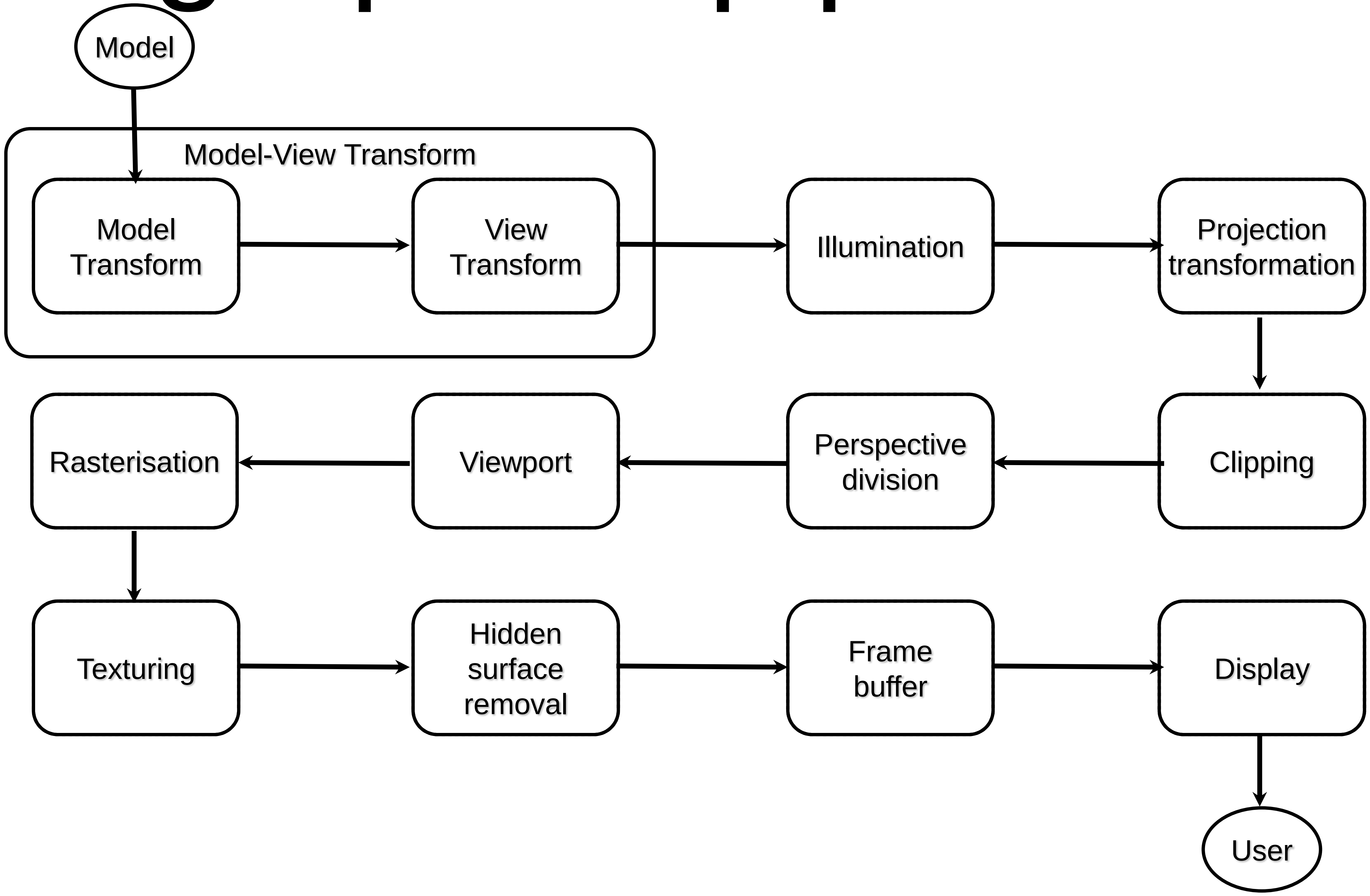
COMP3421

Hidden surface removal
Illumination

The fixed function graphics pipeline



Fragments

Rasterisation converts polygons into collections of fragments. A fragment is a single image pixel with extra information attached.

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

   // other info...

}
```

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

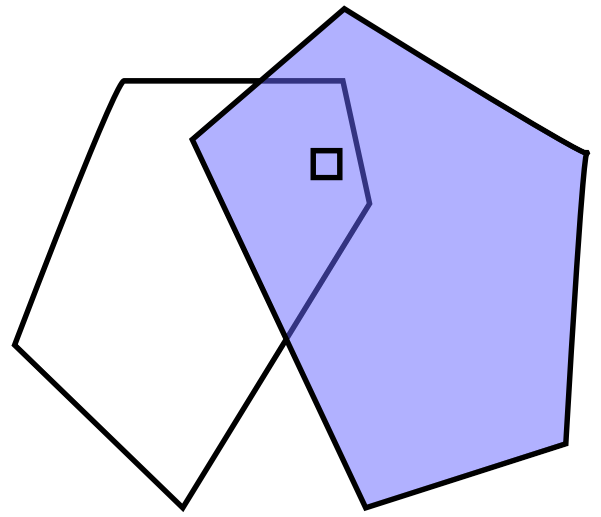
```
// Disabled by default
// To turn on culling:
gl.glEnable(GL2.GL_CULL_FACE);
gl.glCullFace(GL2.GL_BACK);
```

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.

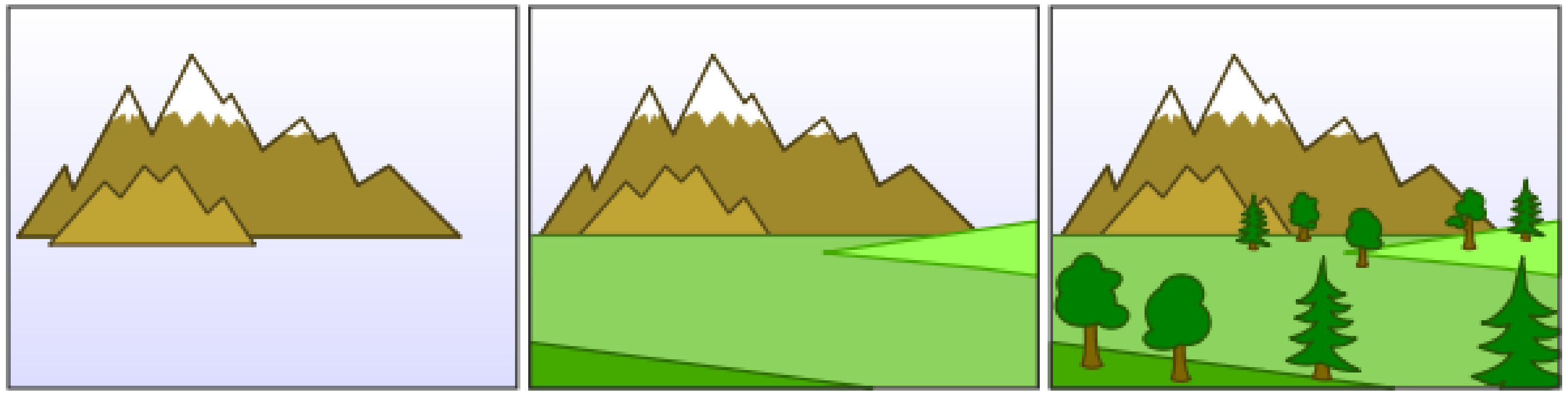
- Make sure all polygons (and therefore fragments) are drawn in the correct order in terms of depth using BSP trees. This is done at the model level. This is not built into OpenGL.
- Use the depth buffer. This is done at the fragment level and is built into OpenGL.

Painter’s algorithm

The naive approach to hidden surface removal:

- Sort polygons by depth
- Draw in order from back to front

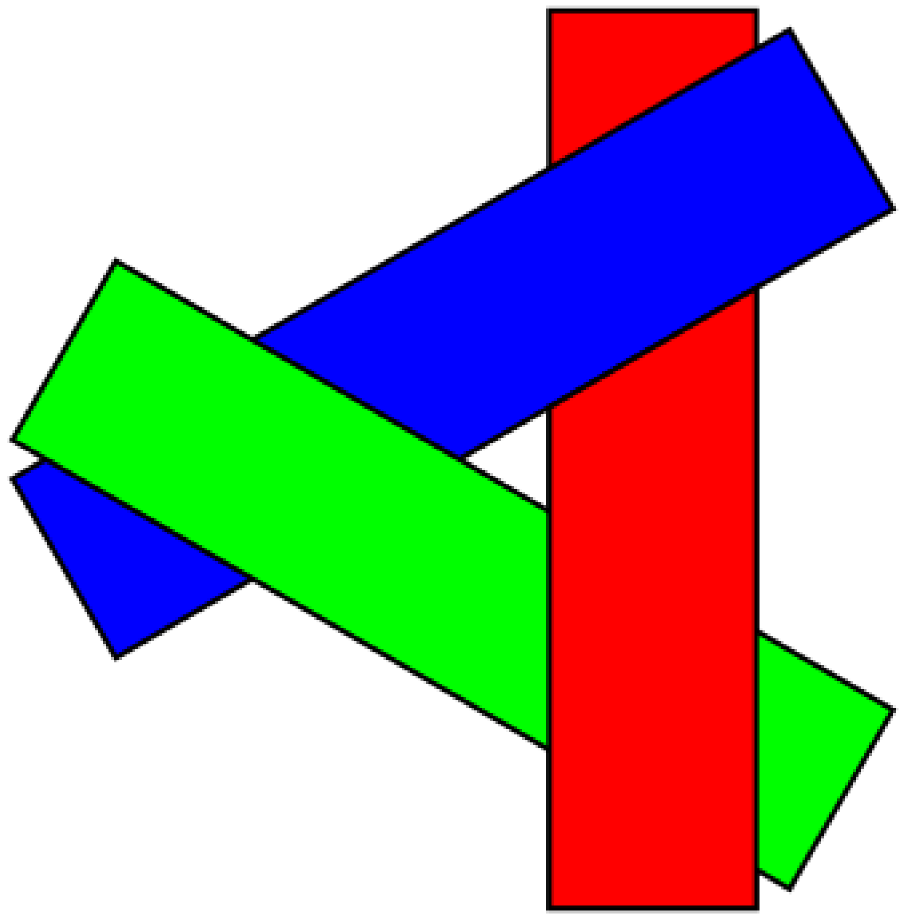
This is known as the Painter's algorithm



Problem

Which polygon to paint first?

We need to split them into pieces.



BSP Trees

Binary Space Partitioning trees

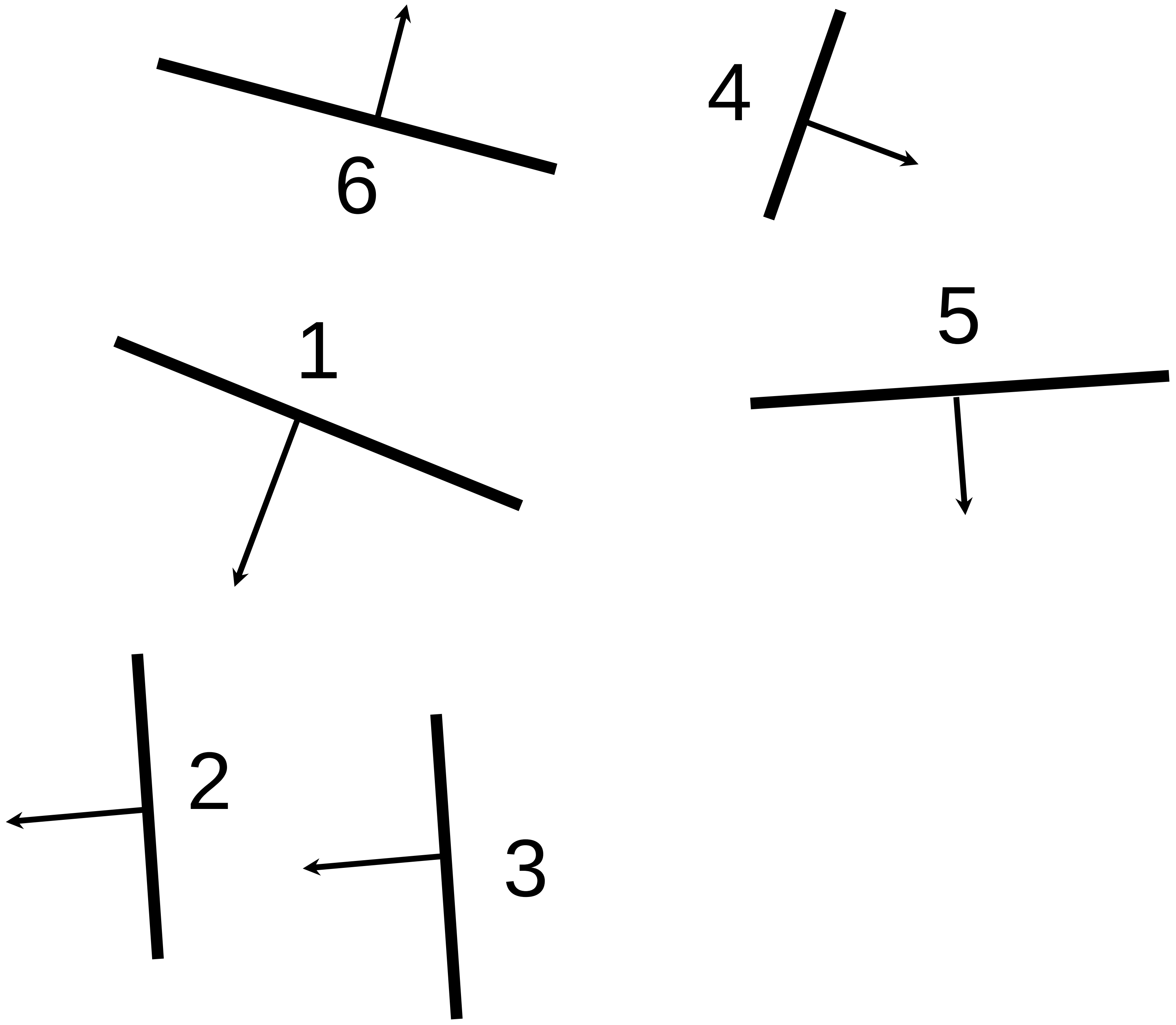
Not implemented in OpenGL

Each polygon splits the world half

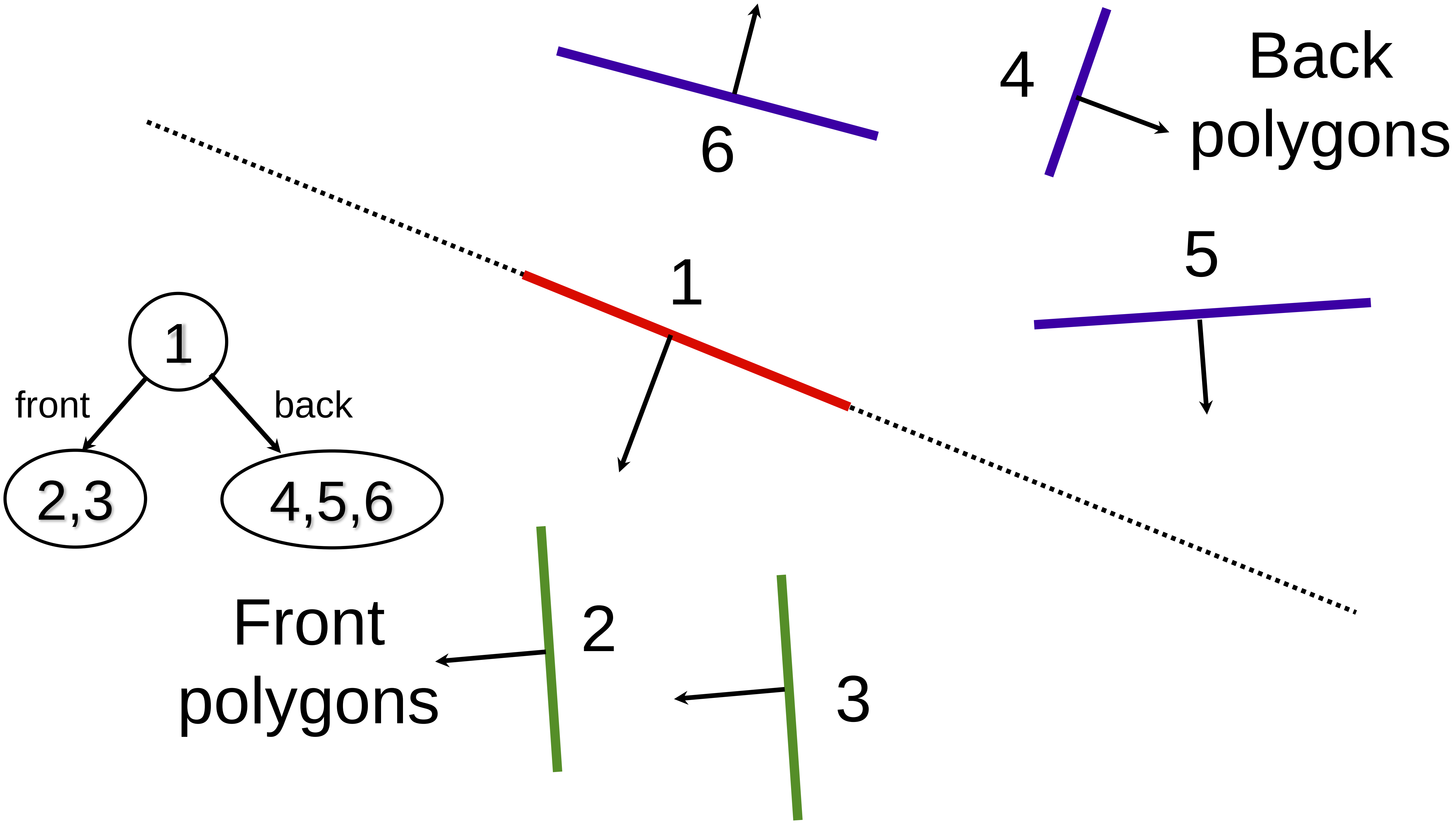
- things behind of the polygon
- things in front of the polygon

We can divide all the other polygons into two sets.

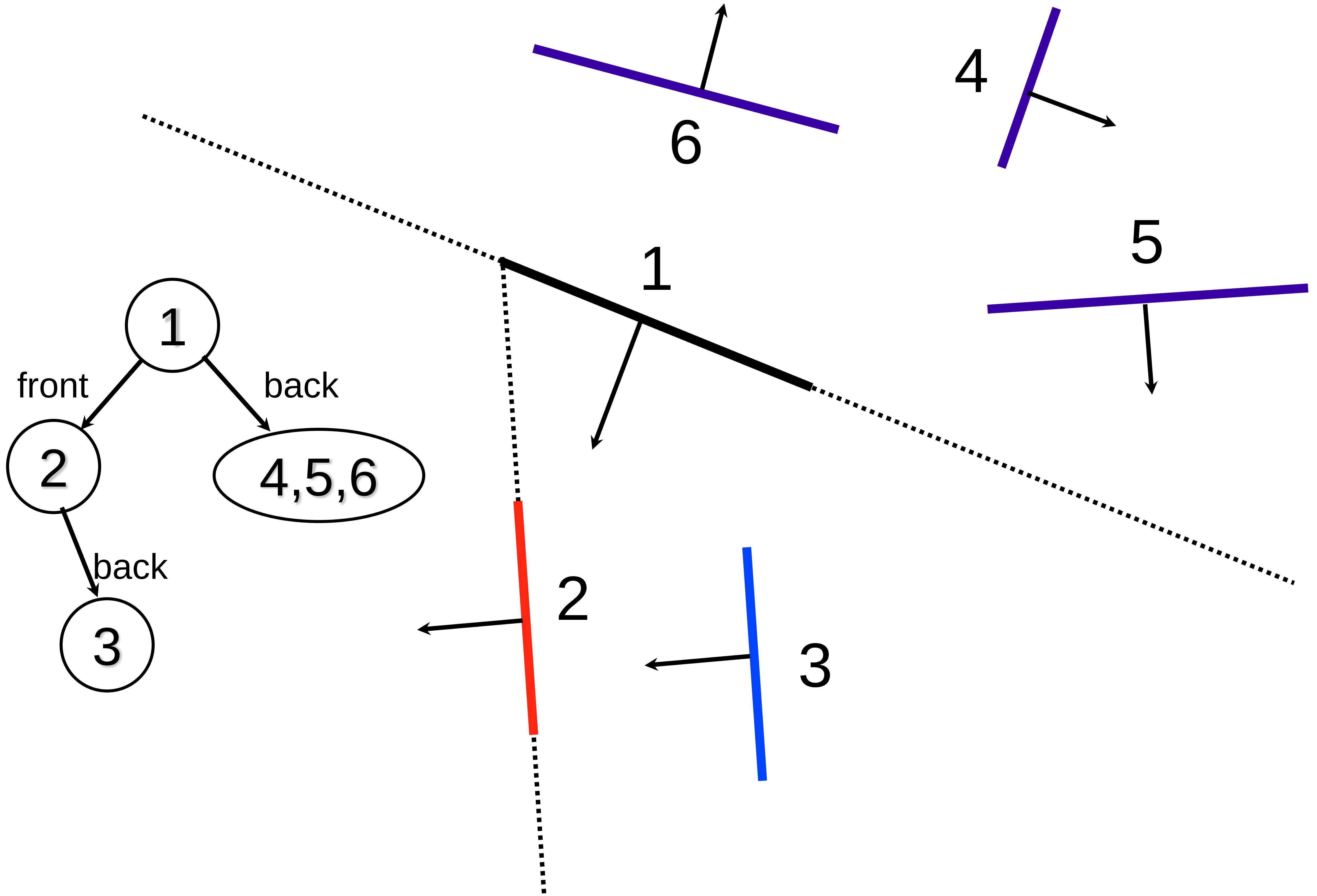
BSP Trees



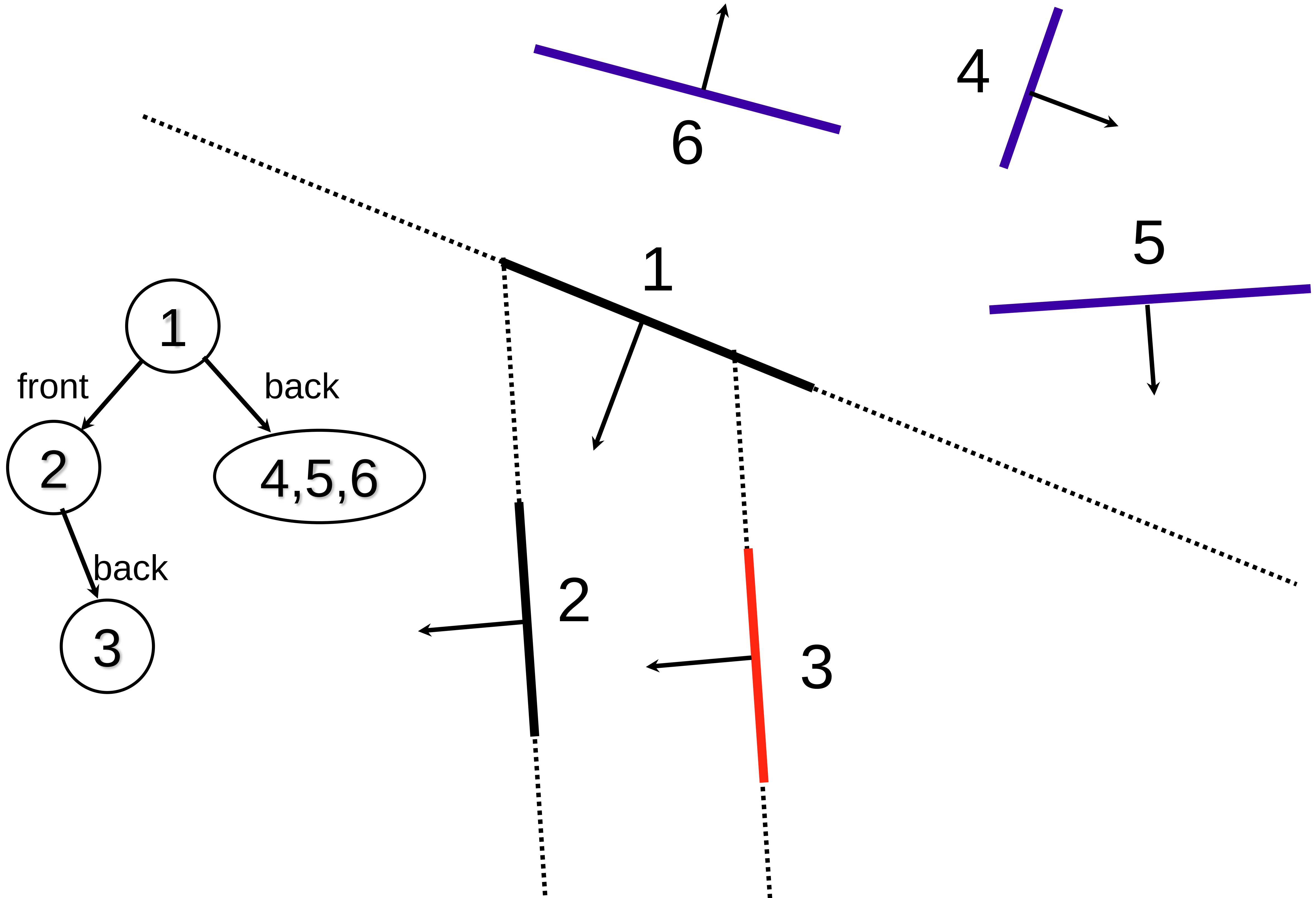
BSP Trees



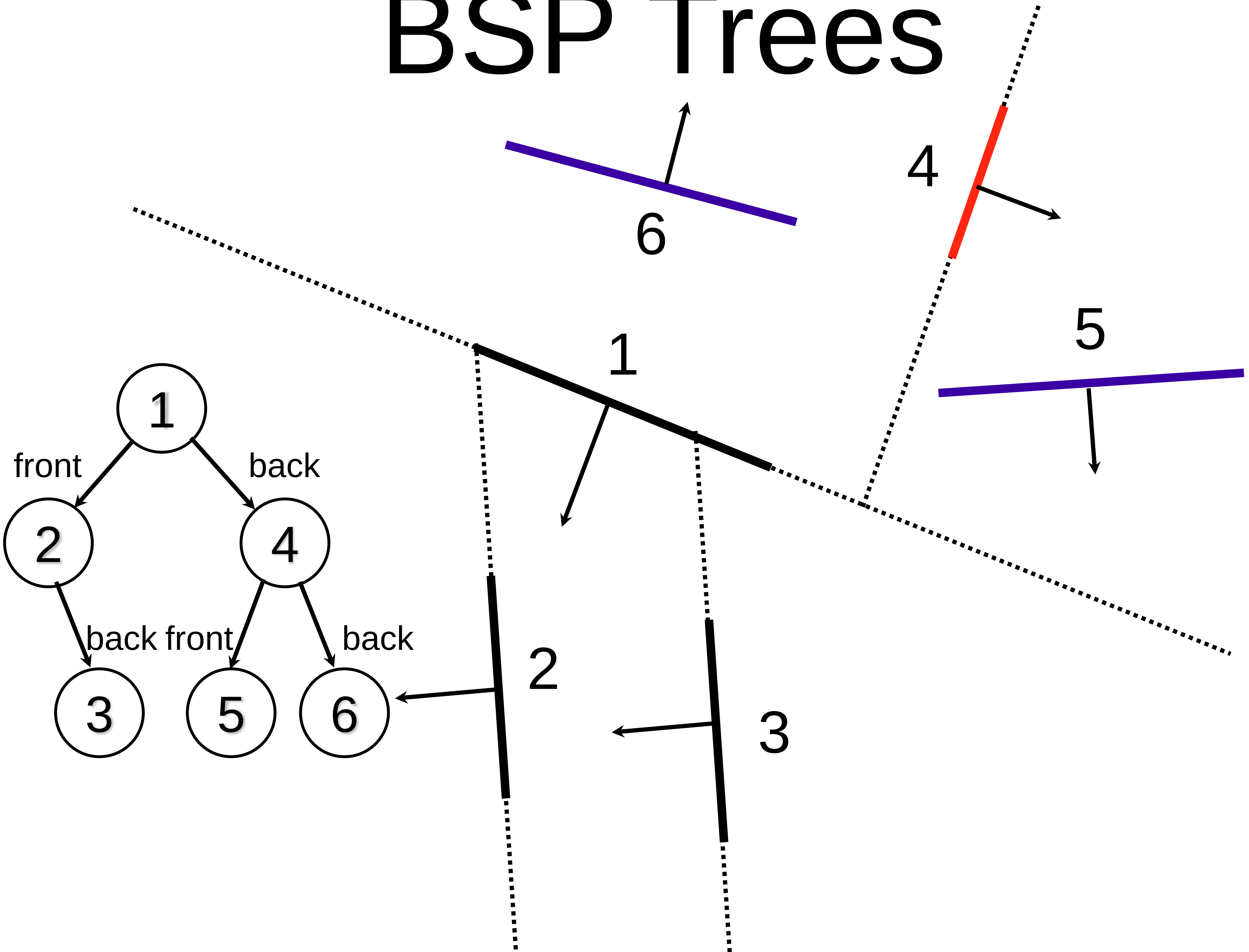
BSP Trees



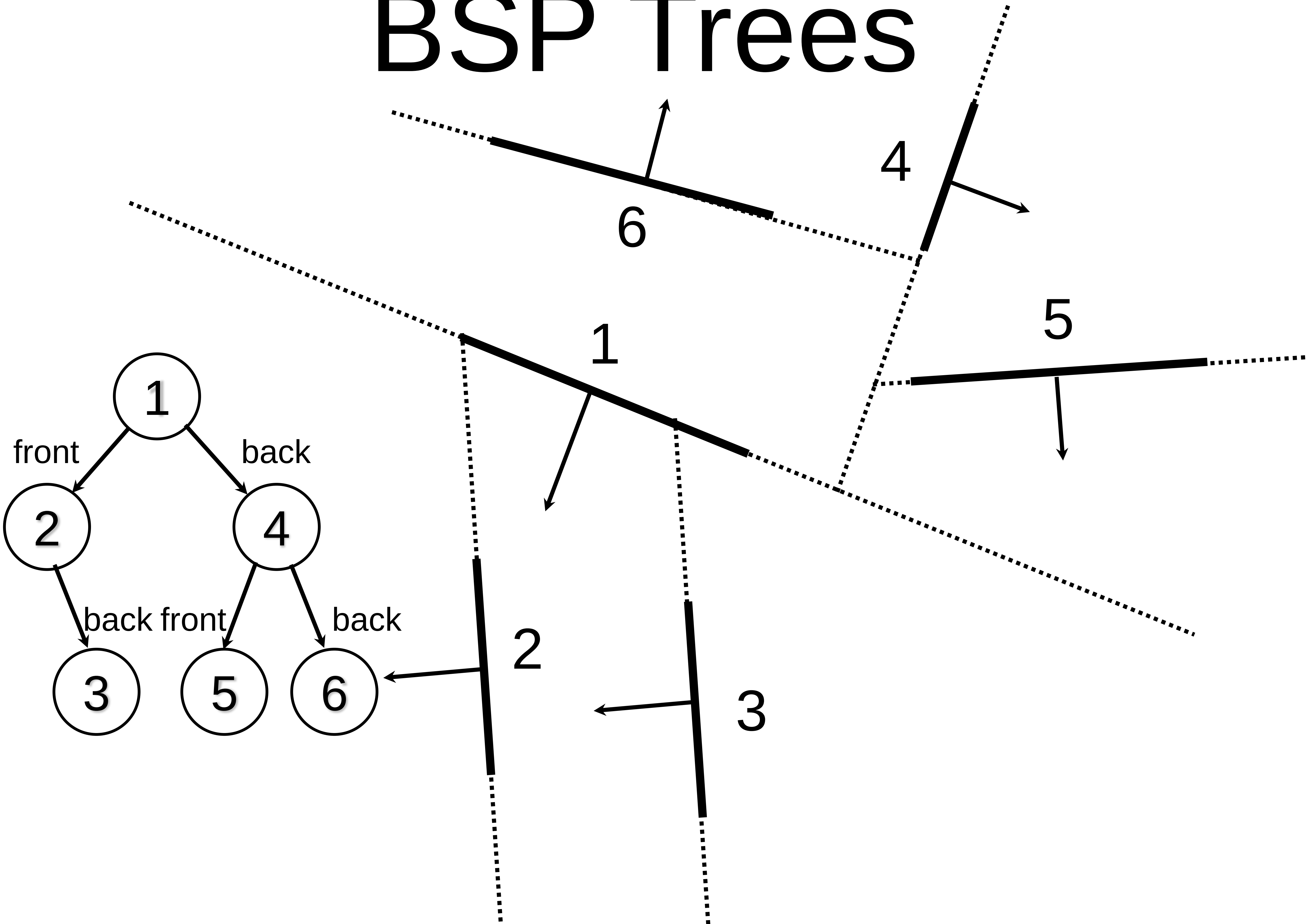
BSP Trees



BSP Trees



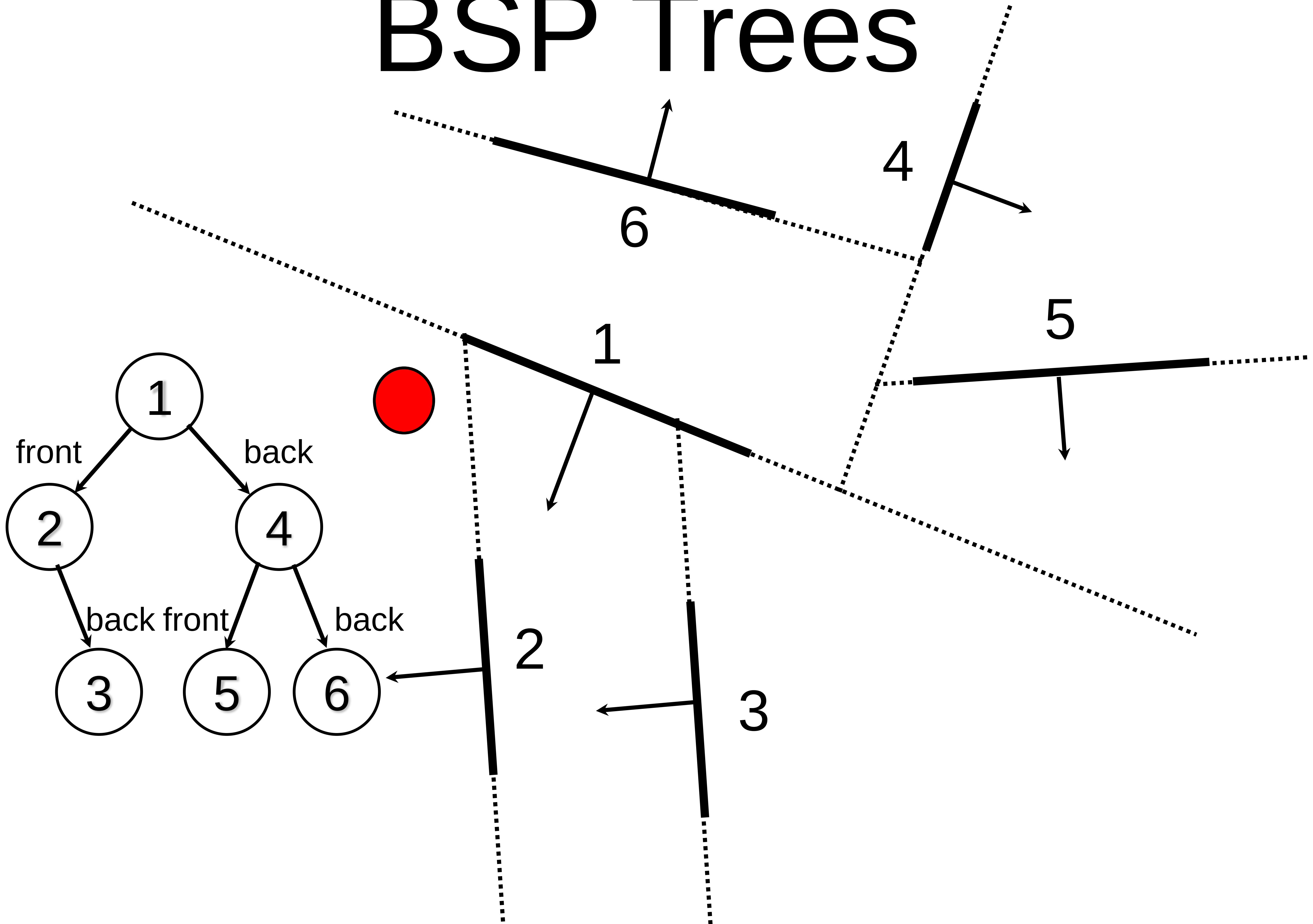
BSP Trees



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

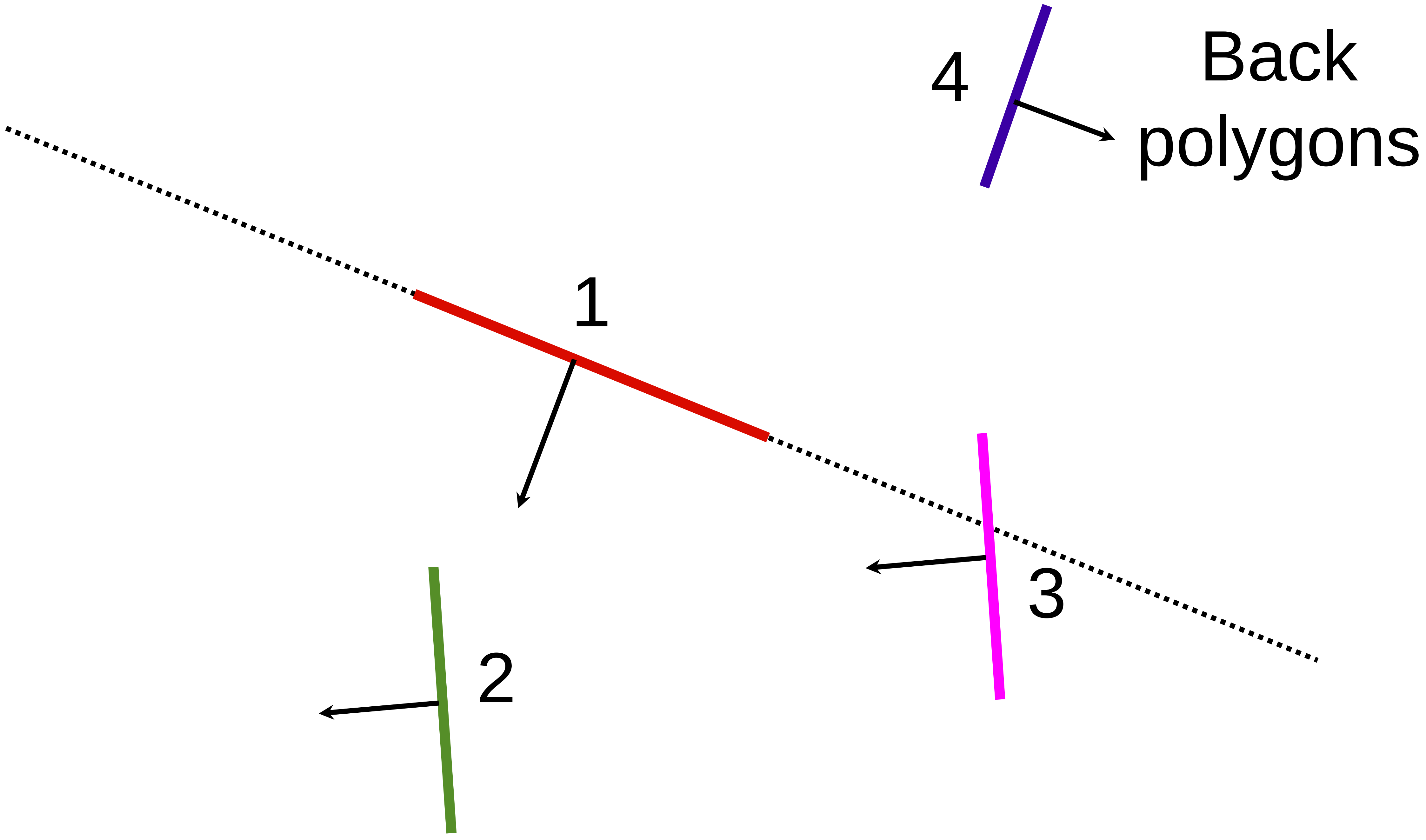
- Camera is in front of 2 so the nodes behind 2 are traversed. This is just node 3 which is drawn
- Finally node 2 is drawn

Example Traversal

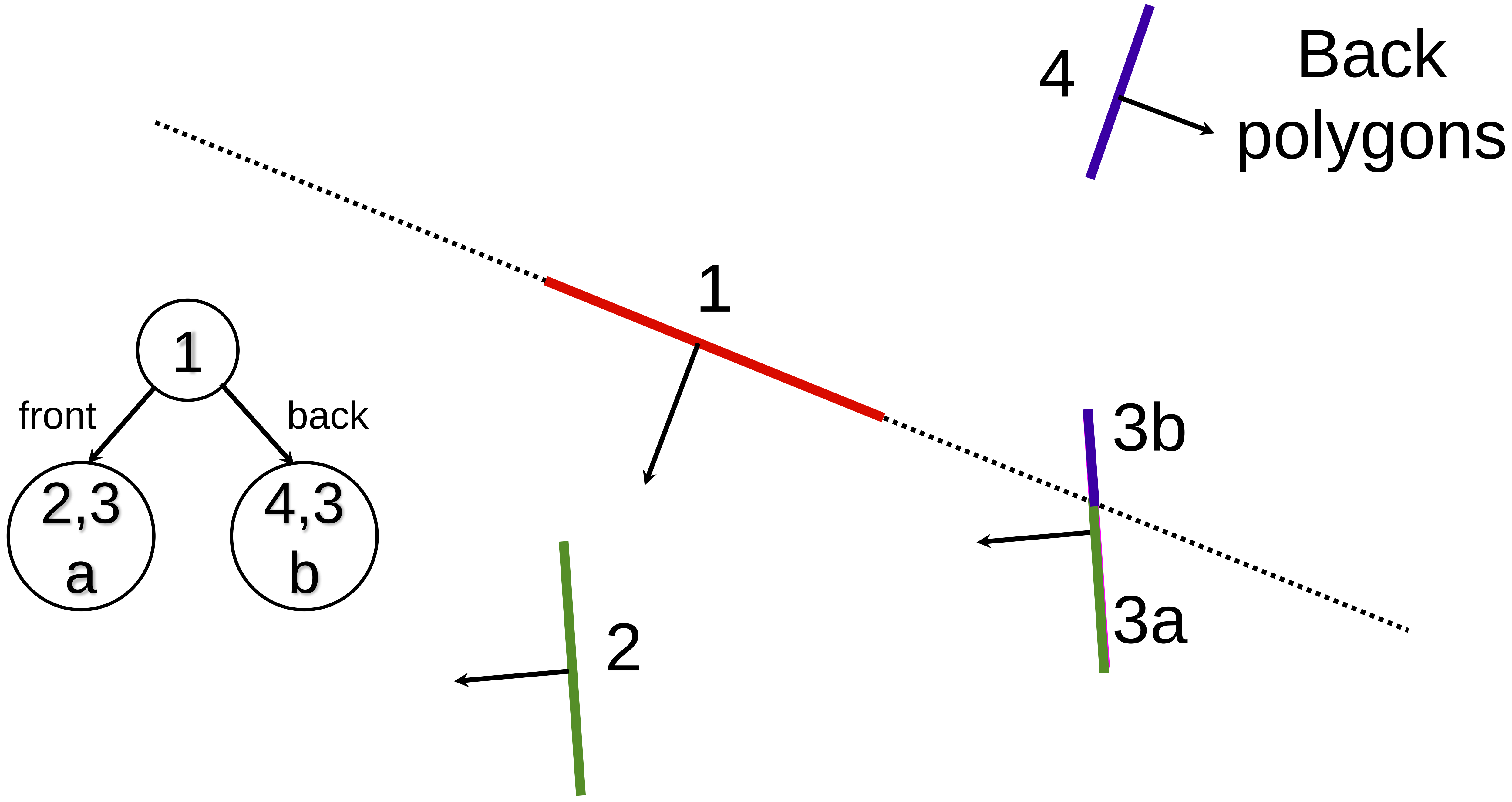
So the order that we draw the polygons for the given camera position is

- 5 4 6 1 3 2

Splitting Polygons



Splitting Polygons



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 5 or 6 times and keep the best.

Pros + Cons

Good: If the camera/viewpoint moves the tree does not need to be recalculated.

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 pixel by pixel.

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

If it is closer, we draw the pixel and update the buffer value to the new pseudodepth.

Pseudocode

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

For each polygon:

    For each pixel (px,py):

        d = pseudodepth of (px,py)

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

            draw pixel

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

Example

Polygon

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

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

Example

Polygon

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

Buffer

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

				.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

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

Computing pseudodepth

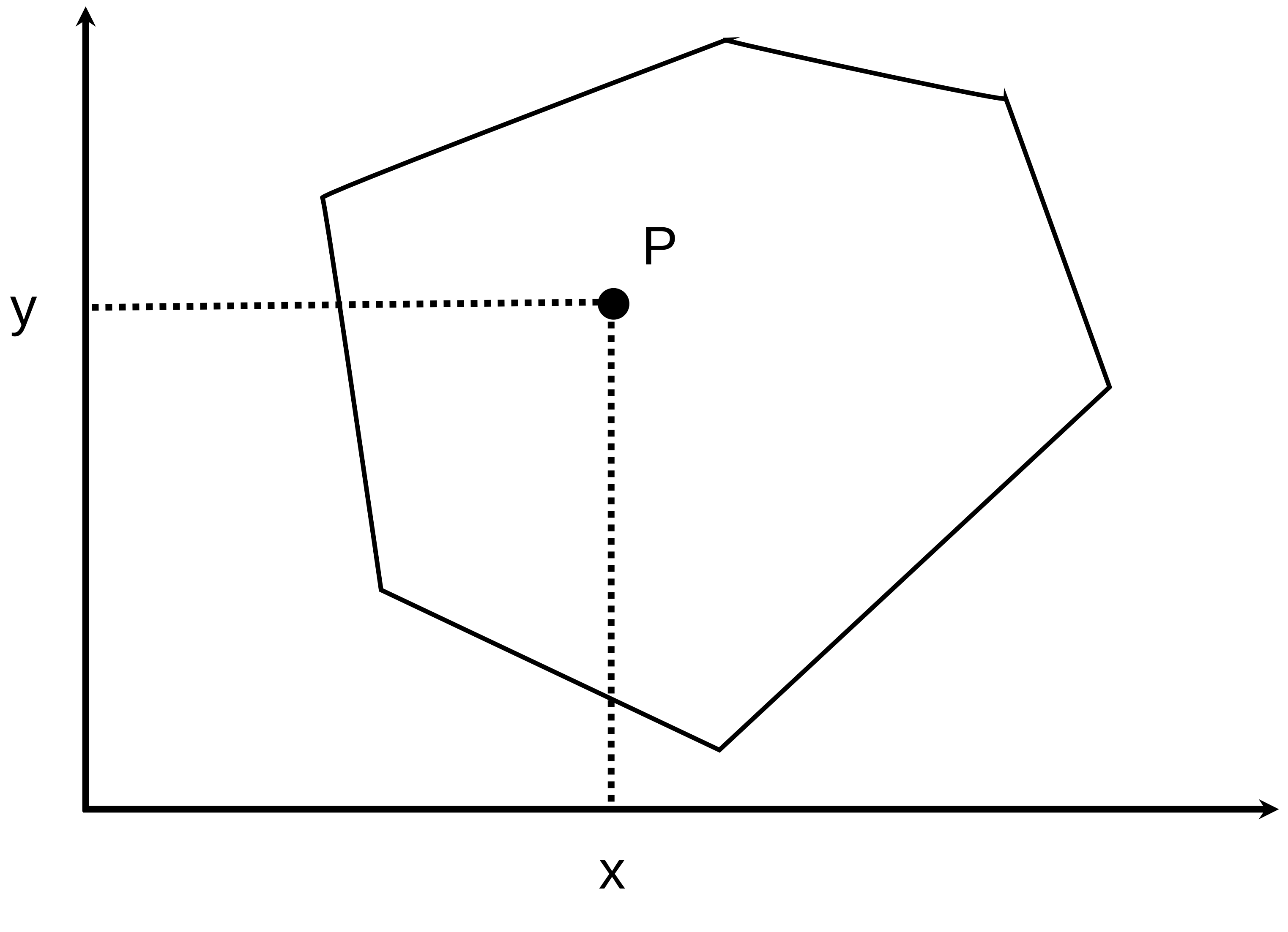
We know how to compute the pseudo depth of a vertex.

How do we compute the depth of a pixel?

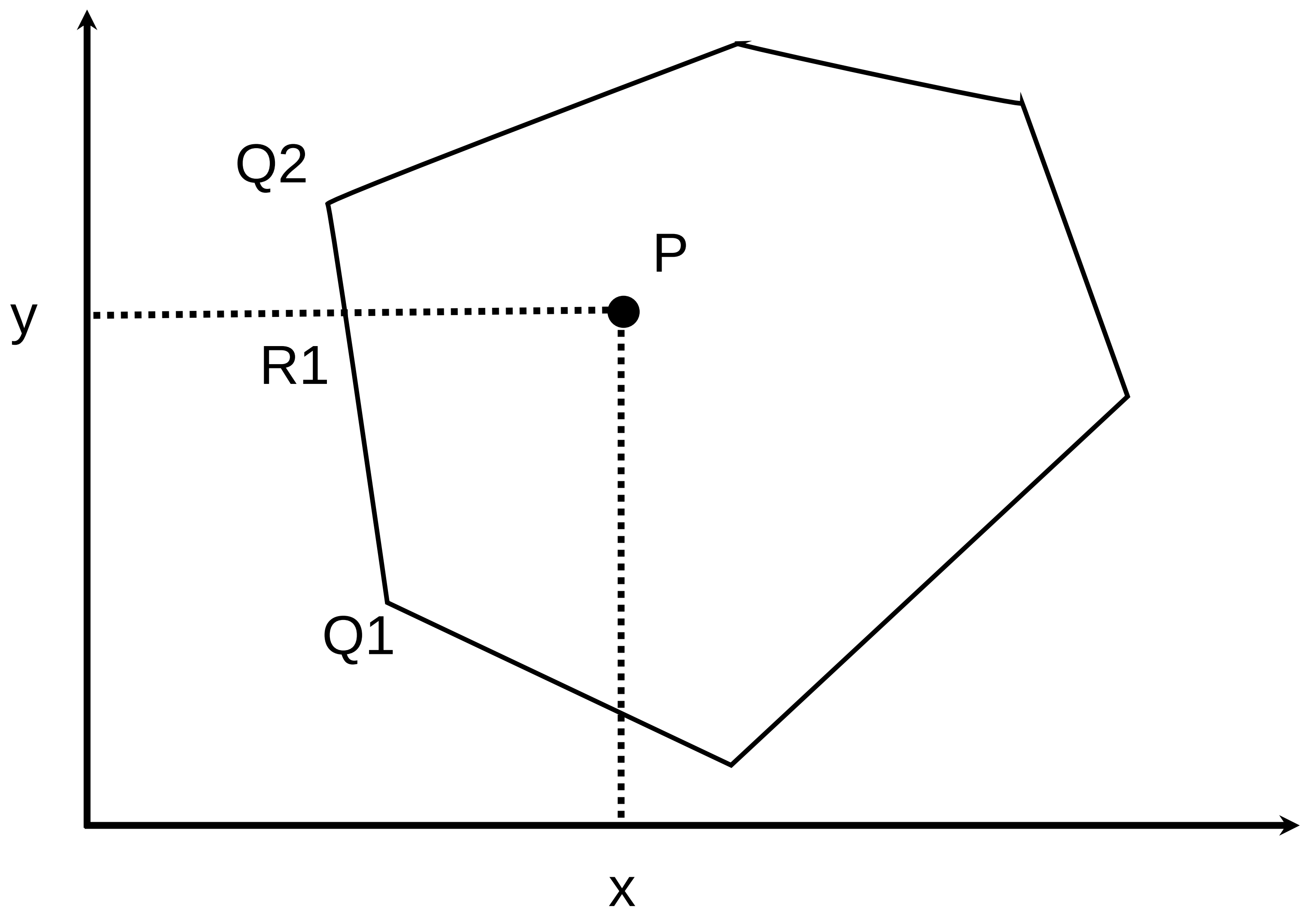
We use bilinear interpolation based on the depth values for the polygon vertices.

Bilinear interpolation is lerping in 2 dimensions.

Bilinear interpolation

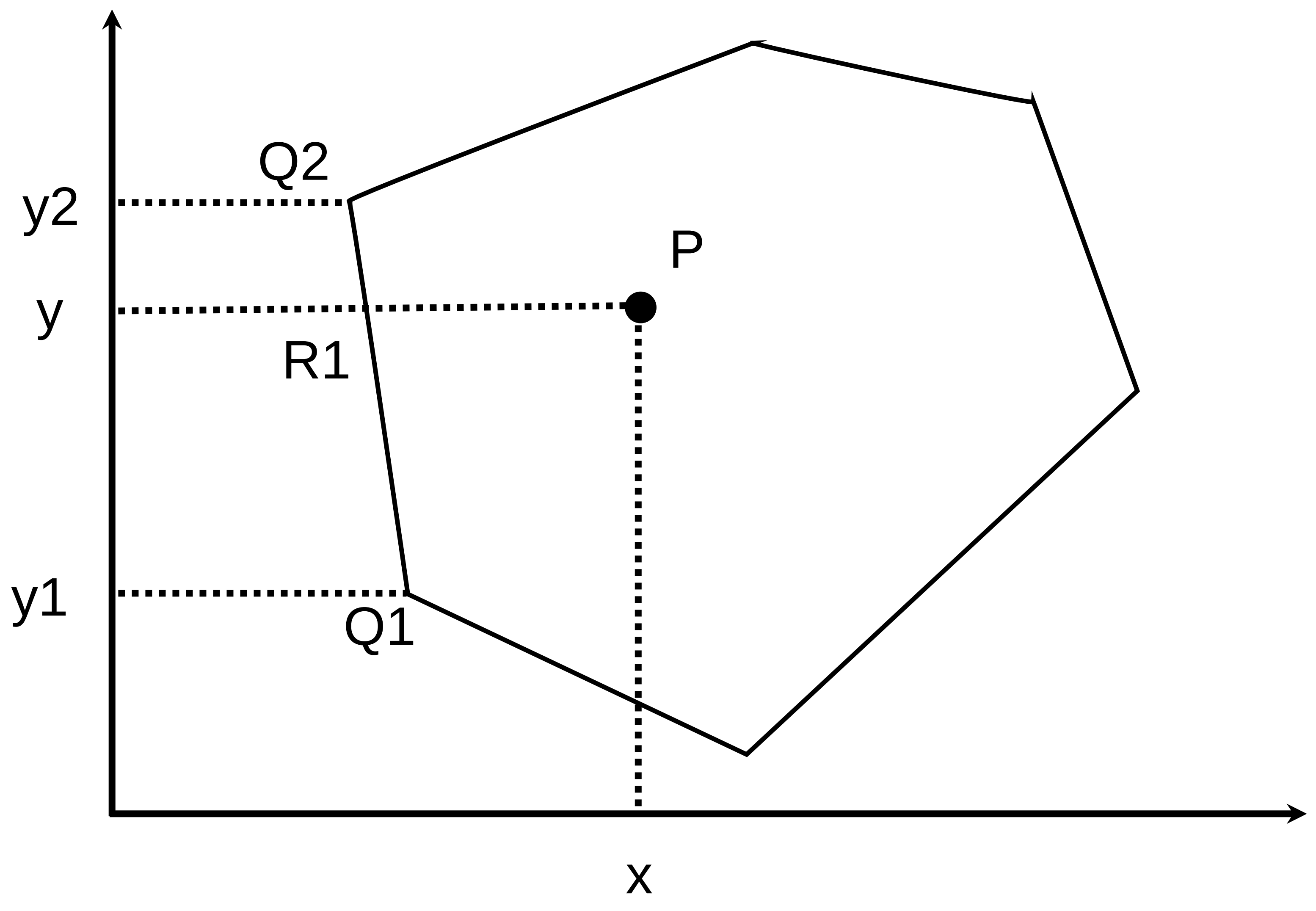


Bilinear interpolation



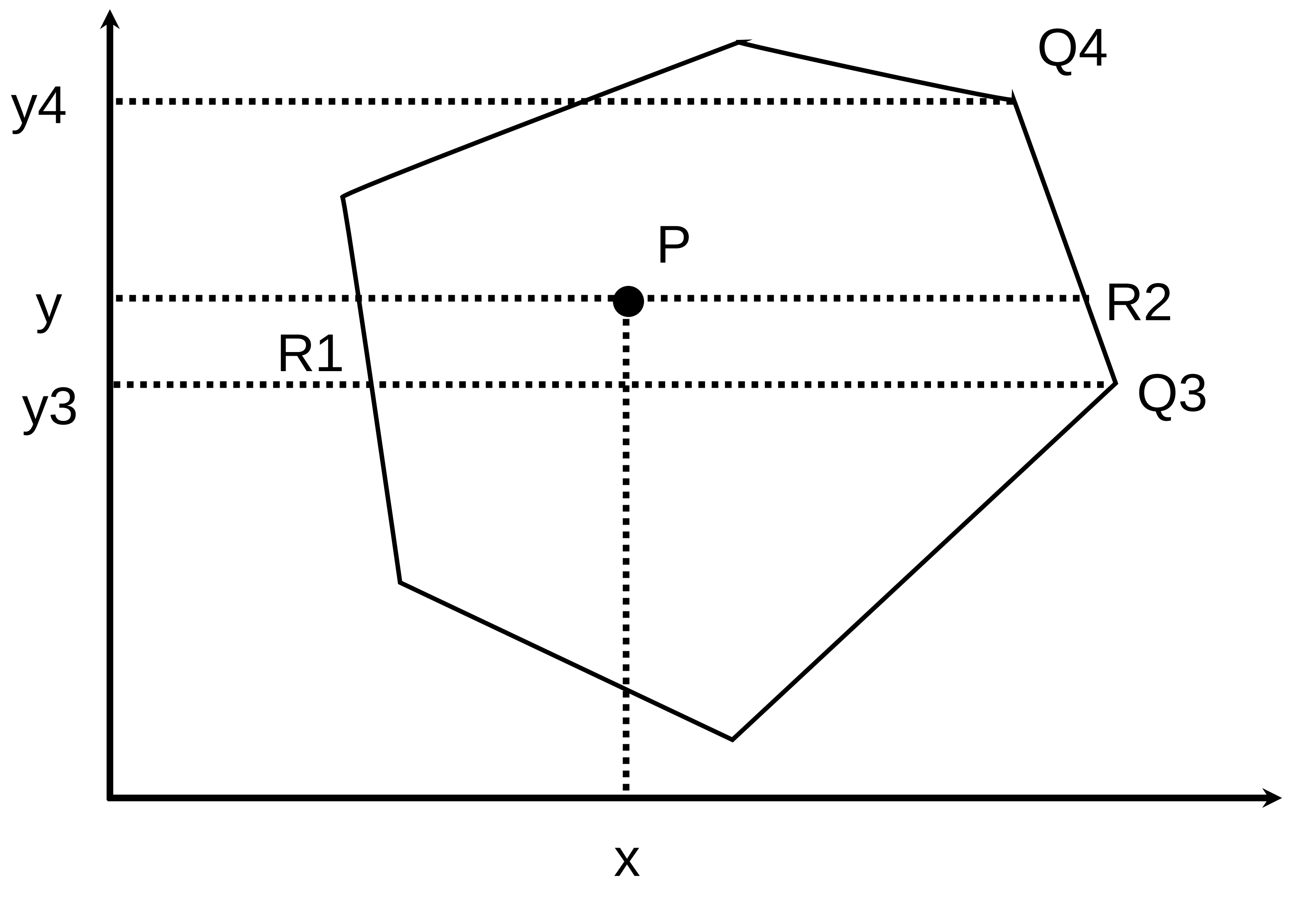
Bilinear interpolation

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



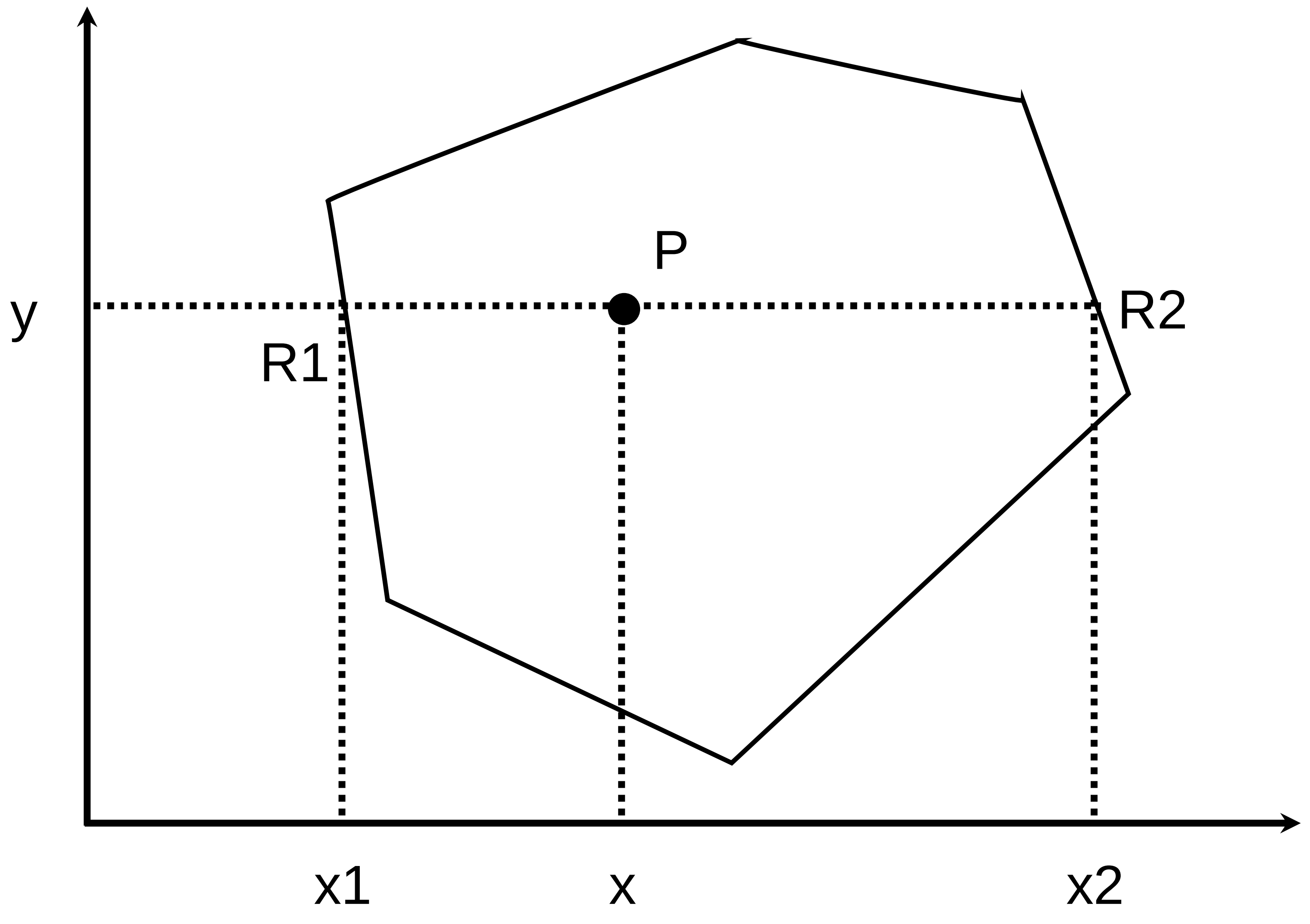
Bilinear interpolation

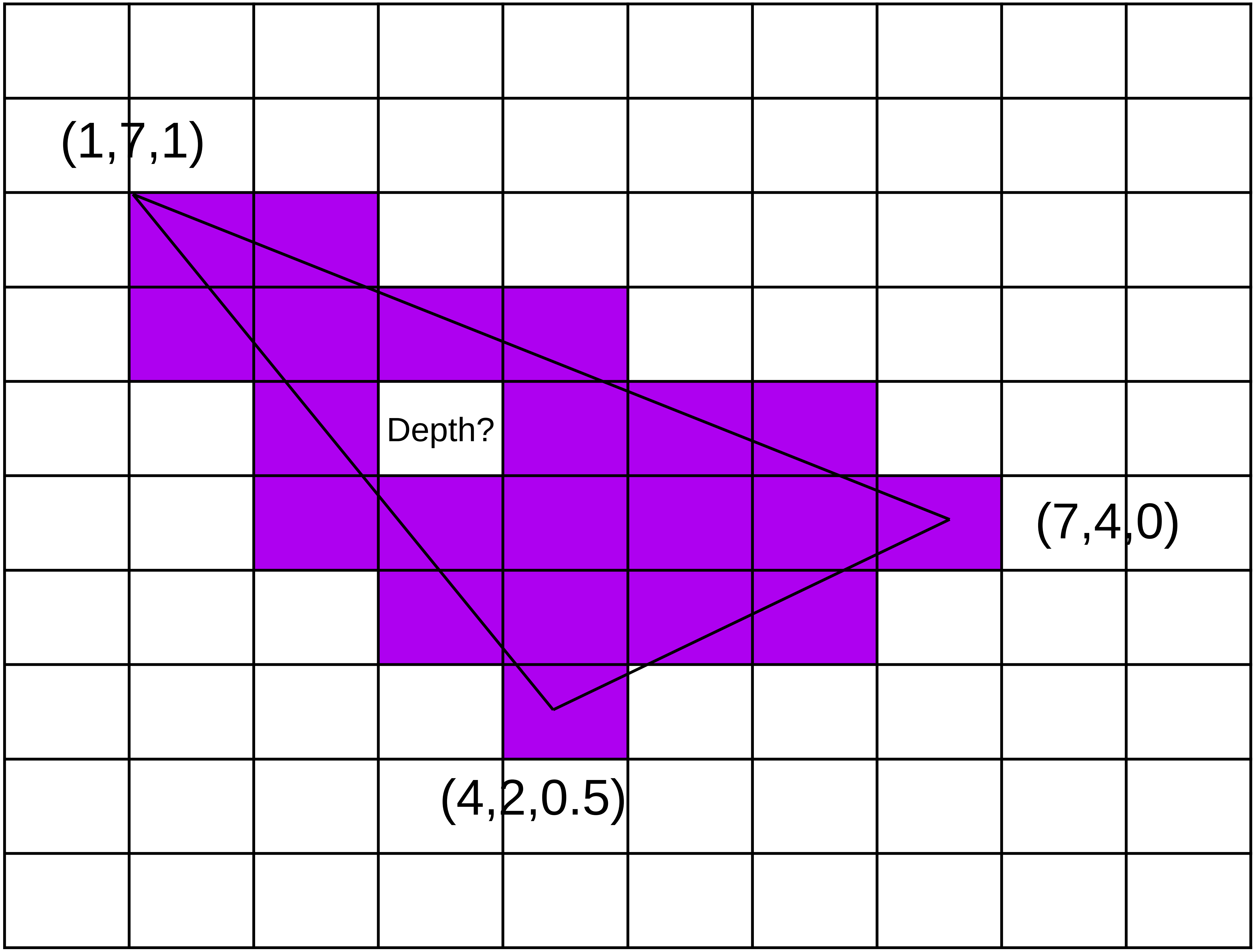
$$f(R_2) = \frac{y - y_3}{y_4 - y_3} f(Q_4) + \frac{y_4 - y}{y_4 - y_3} f(Q_3)$$

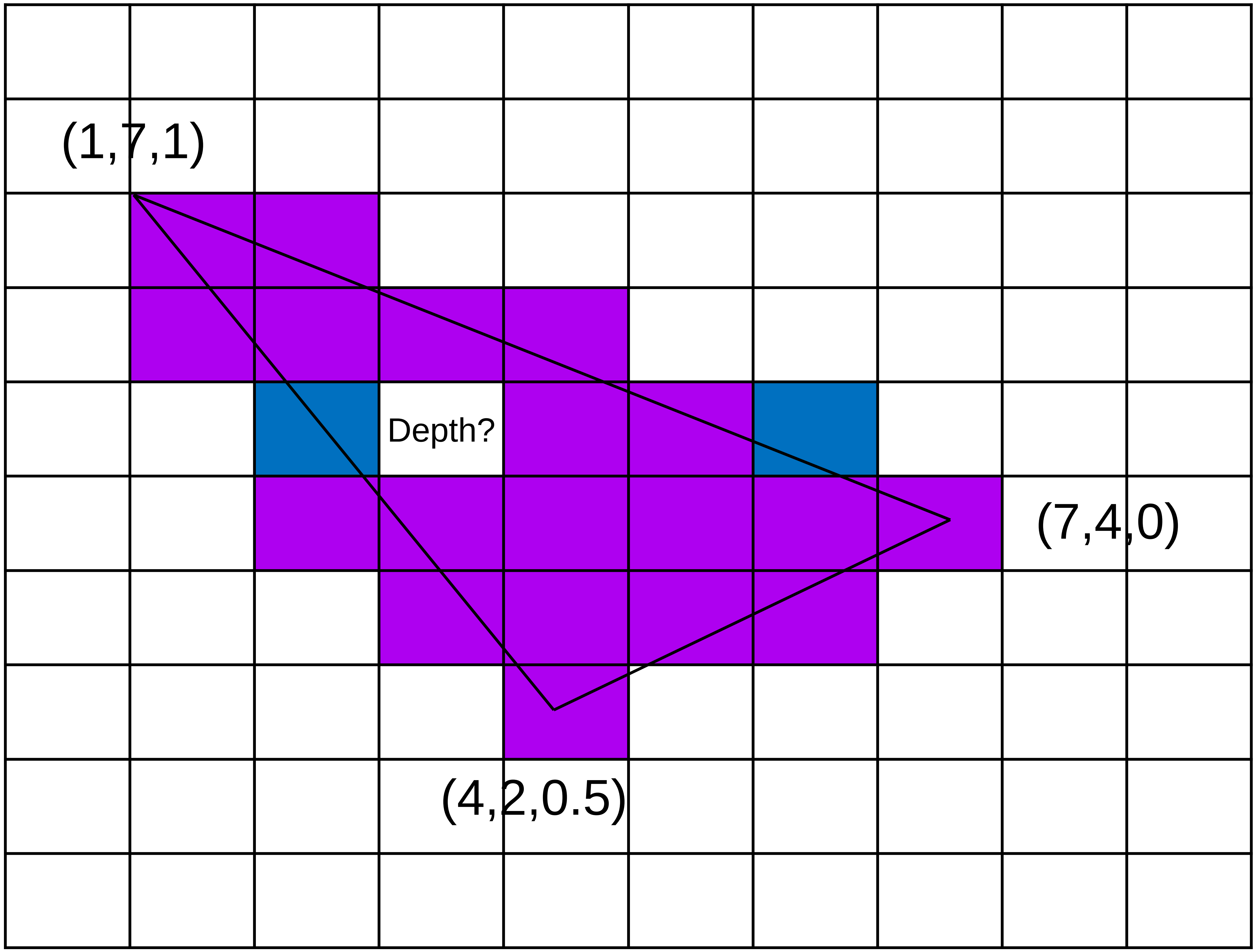


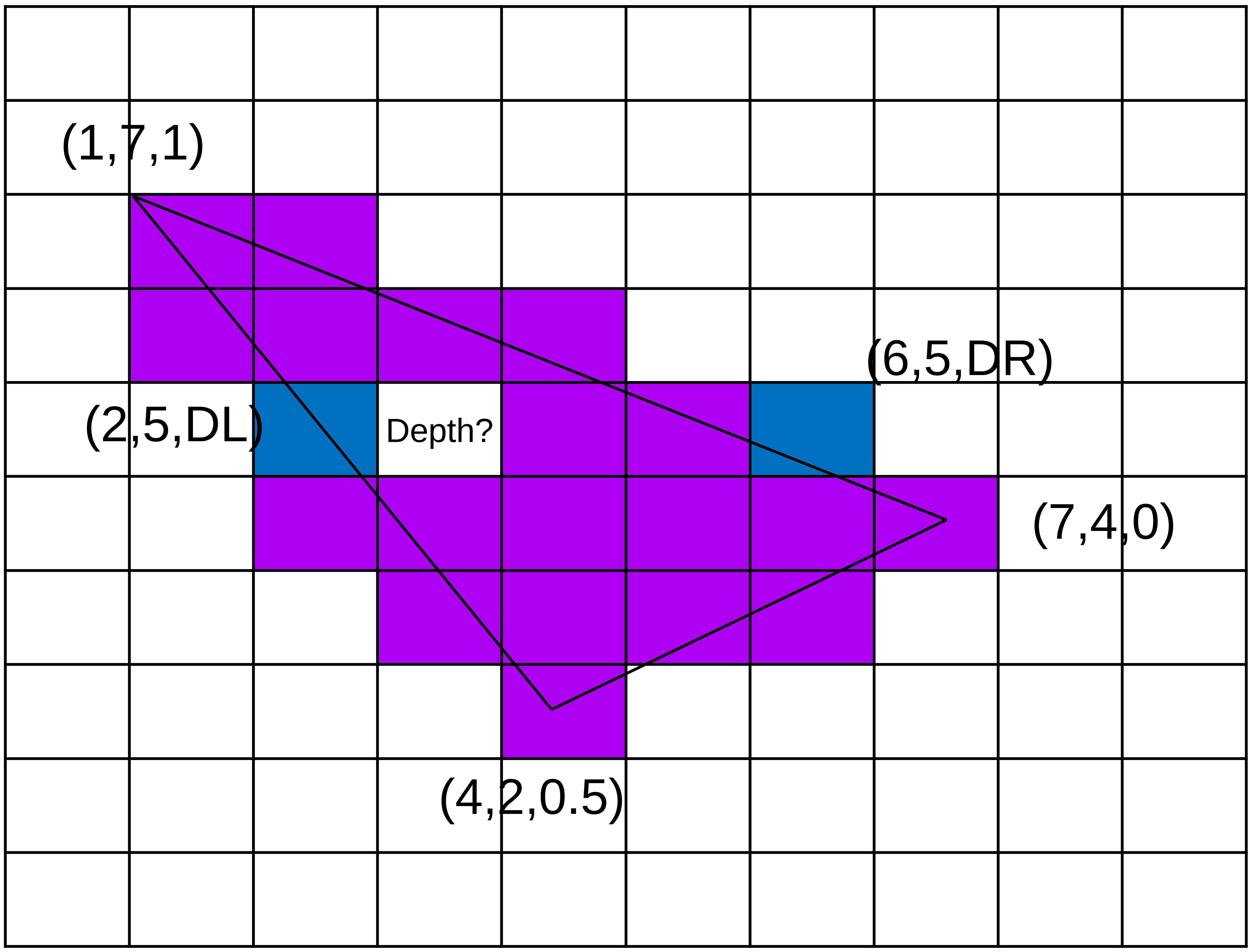
Bilinear interpolation

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




(1,7,1)
Depth?
(7,4,0)
(4,2,0.5)


(1,7,1)
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)

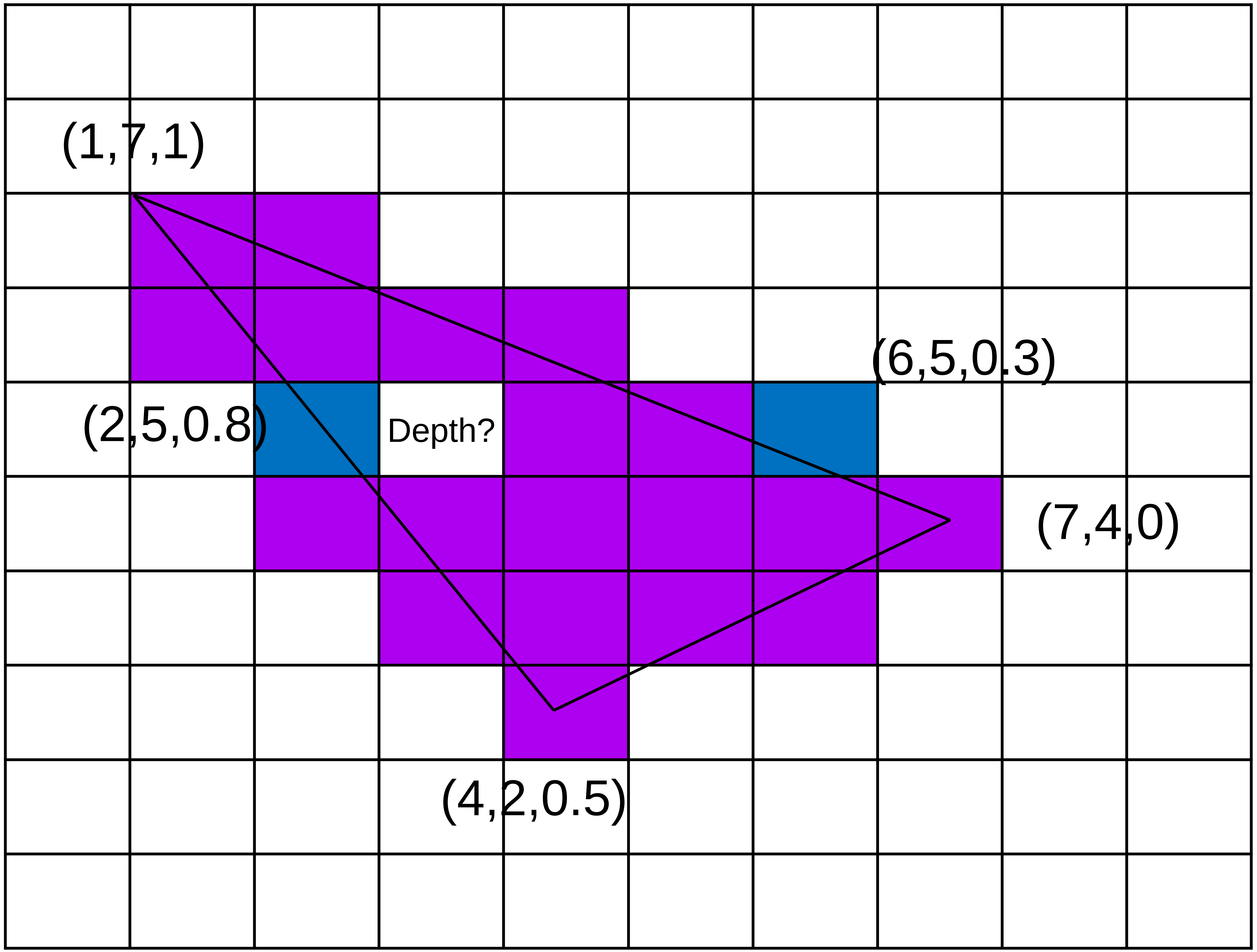
DL = (5 – 2)/(7-2)*1 + (7 – 5)/(7-2)*0.5

= (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

= (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

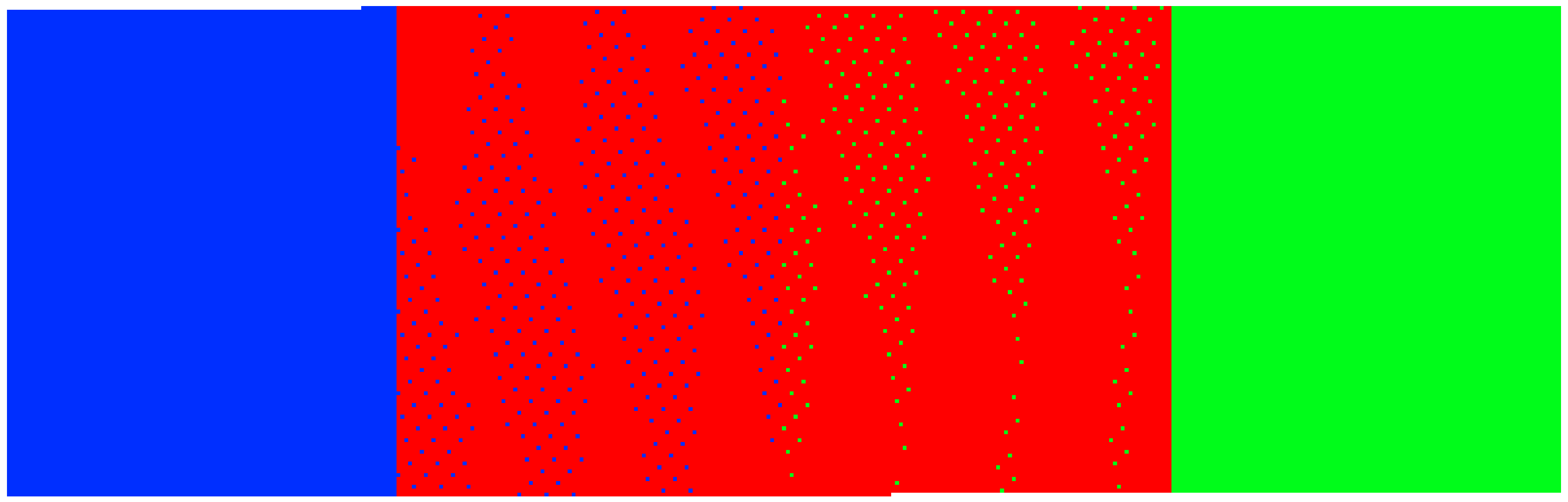
=1/4*0.3 + 3/4*0.8

= 0.675

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.

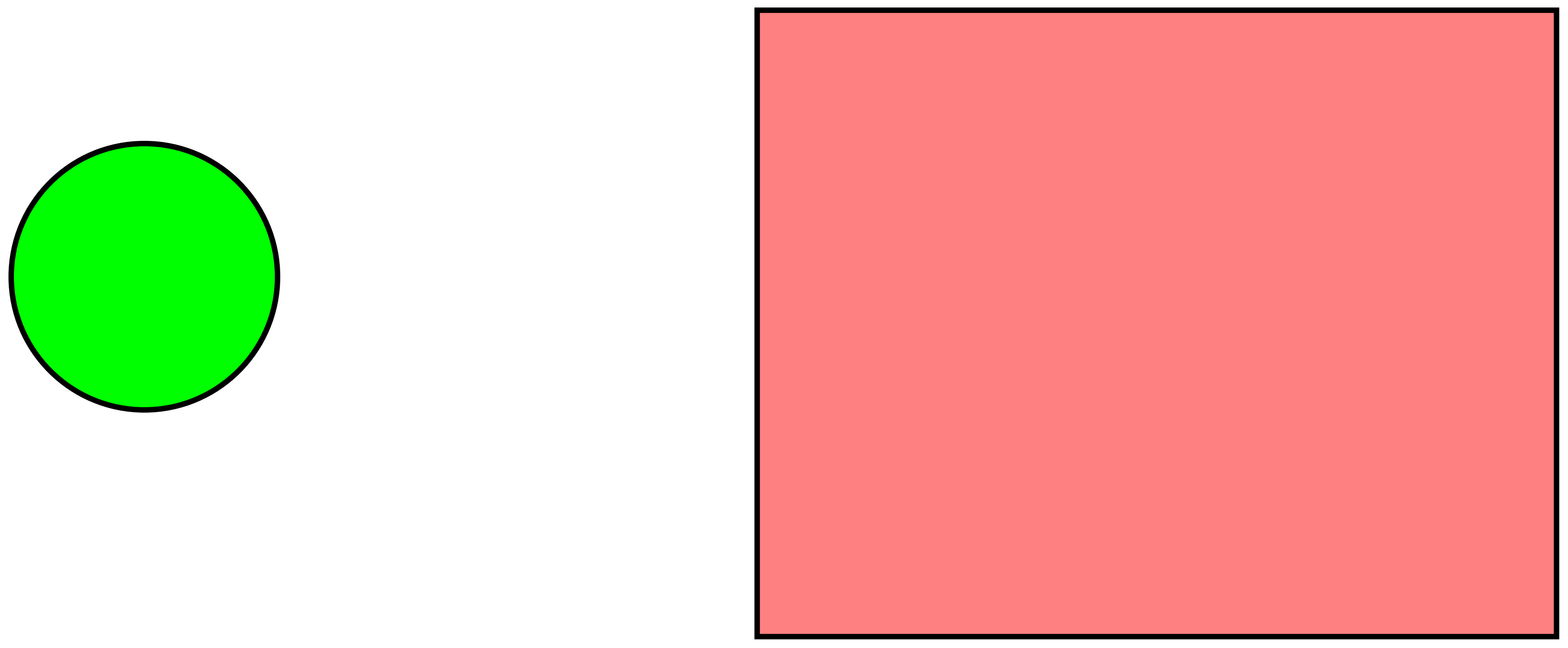
We use BSP-trees to represents the static geometry, so the tree only needs to be built once (at compile time).

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.

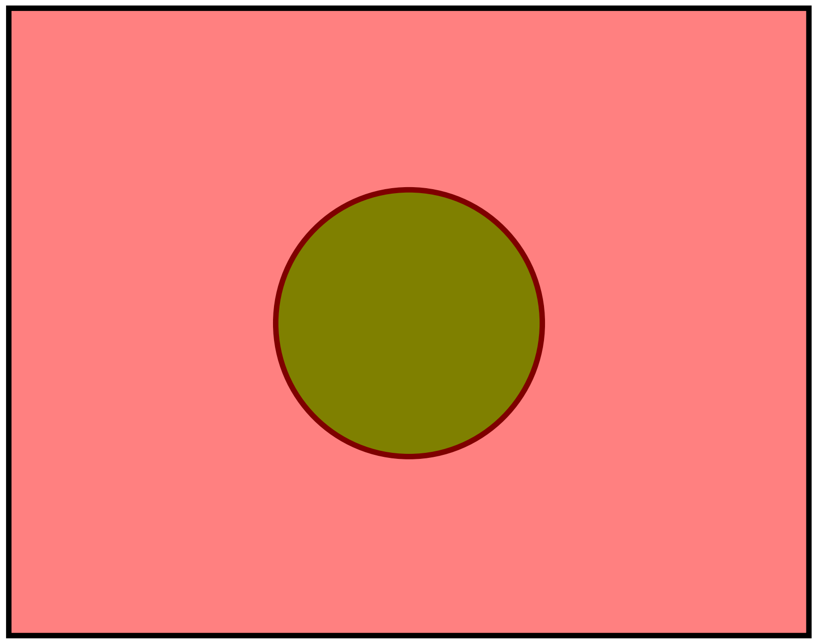
Transparency

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.

$$\begin{aligned} p &= 0.25 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0.25 \end{pmatrix} + 0.75 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 0.25 \\ 0.75 \\ 0 \\ 0.8125 \end{pmatrix} \end{aligned}$$

OpenGL

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

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

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

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

		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		

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

Back-to-front

Back Polygon

		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		

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

.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

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

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

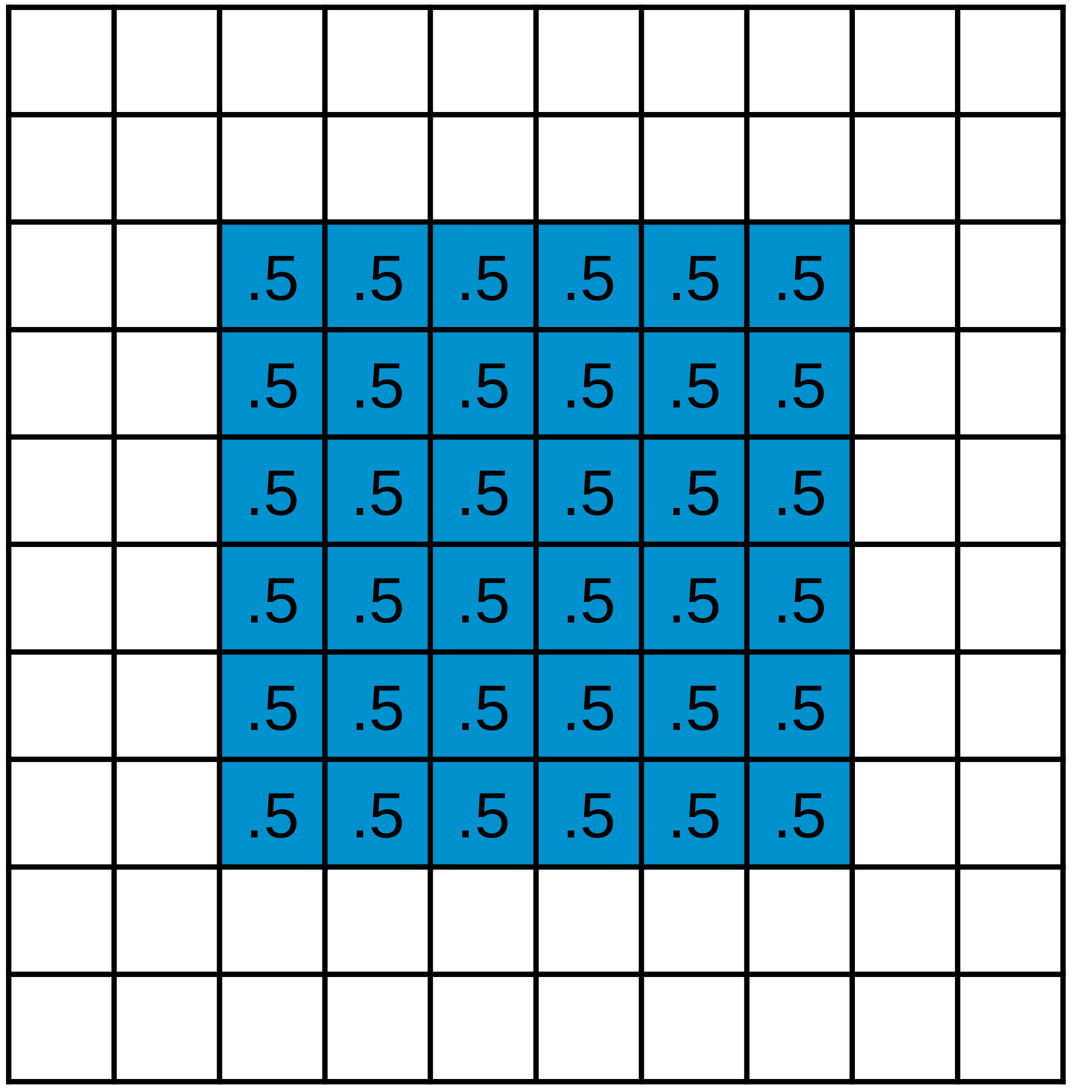
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		

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

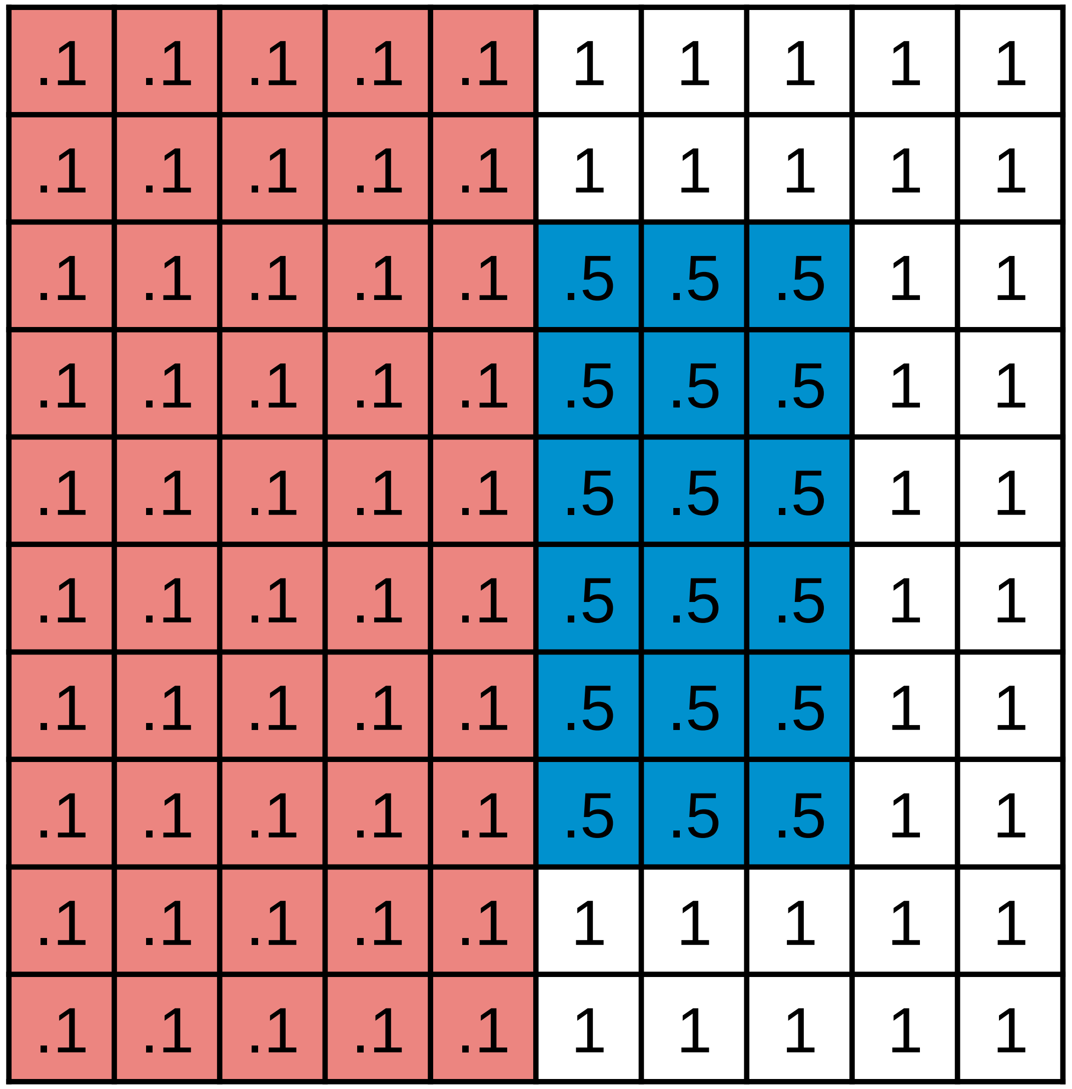
Front-to-back

Front Polygon



		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		
		.5	.5	.5	.5	.5	.5		

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

Wrong

Transparency

If you want to implement transparency, you need to ensure your (transparent) polygons are drawn in back-to-front order.

A BSP tree is one solution for this problem.

Other fudges are to draw your transparent polygons last, but turn 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.

Illumination

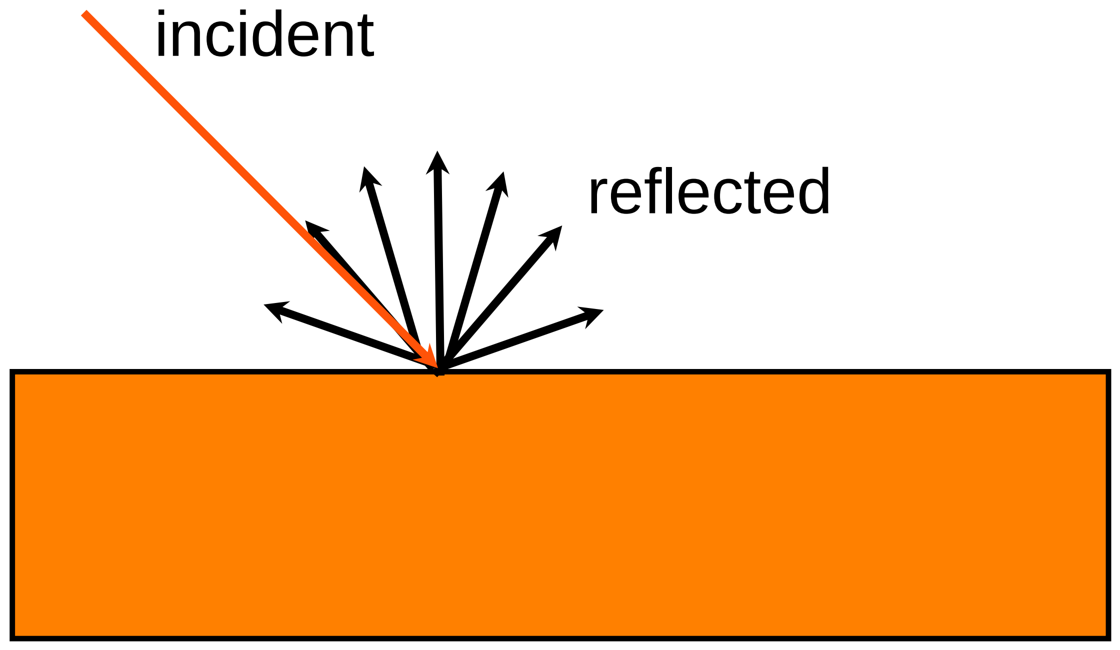
The colour of an object in a scene depends on:

- The colour and amount of light that falls on it.
- The color and reflectivity of the object

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

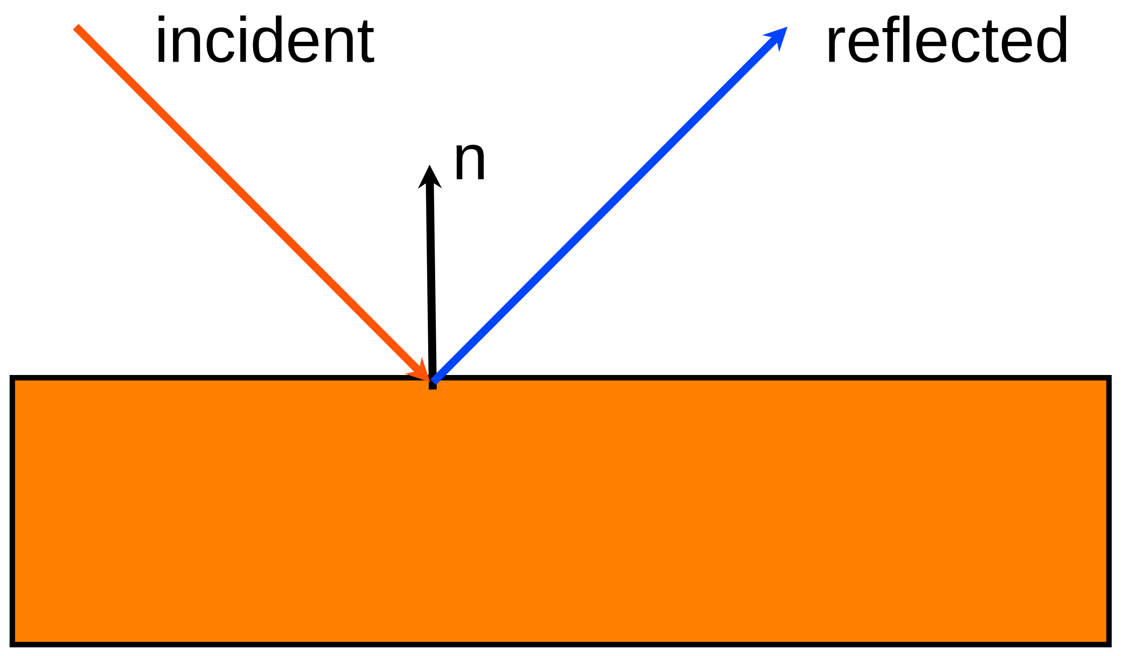
Diffuse reflection

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.



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:

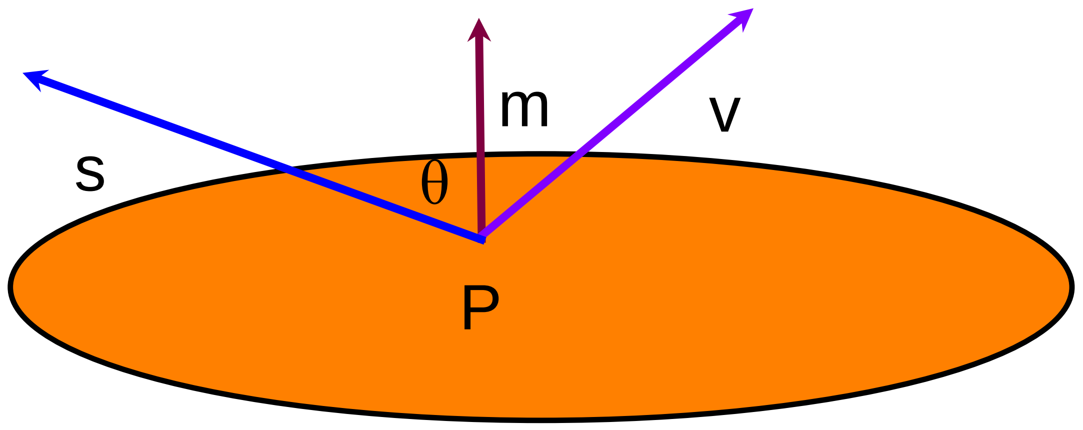
$$I(P) = I_{ambient}(P) + I_{diffuse}(P) + I_{specular}(P)$$

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:

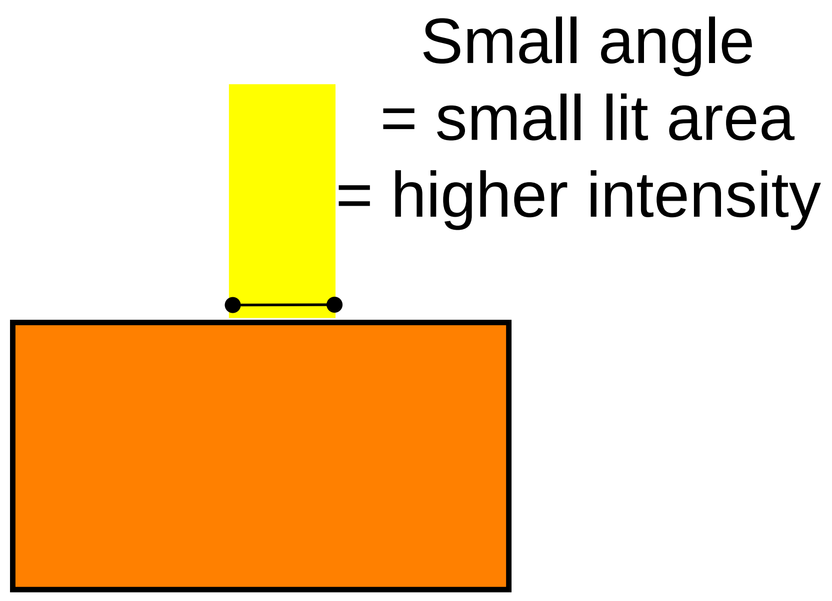
- The normal vector **m** to the surface at P
- The view vector **v** from P to the camera
- The source vector **s** from P to the light source.

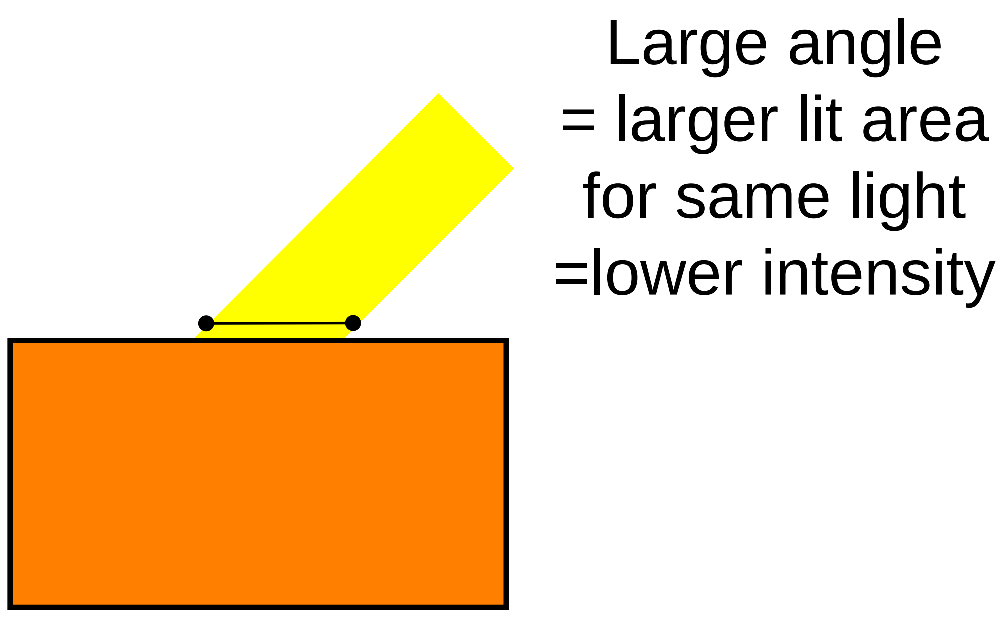


Diffuse illumination

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

The amount of reflected light is inversely proportional to the area of the face subtended by the source.

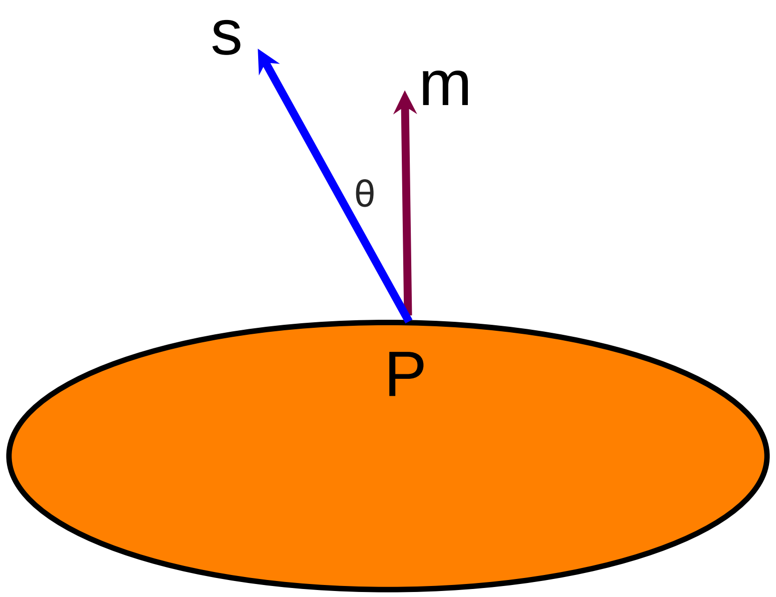




Lambert's Law

We can formalise this as Lambert's Law:

$$\begin{aligned} I_d &\propto I_s \cos\theta \\ &= I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}}) \end{aligned}$$



where:

I_s is the source intensity, and

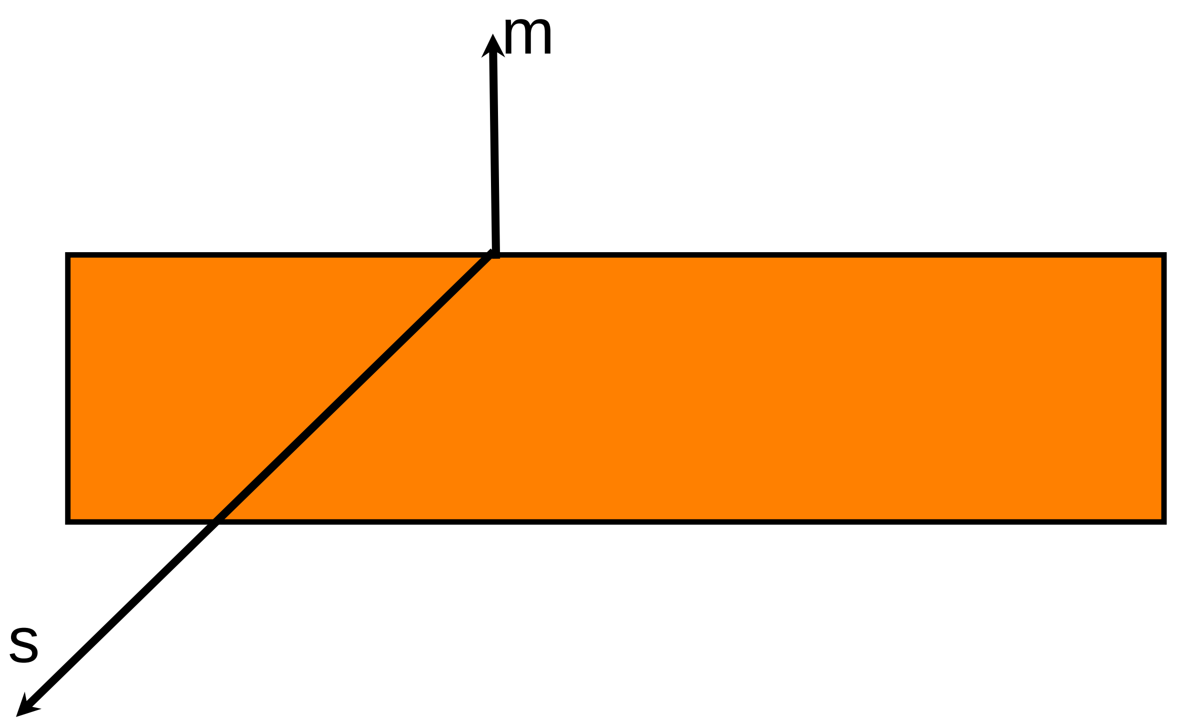
ρ_d is the diffuse reflection coefficient in (0,1)

Note: both vectors are normalised

Lambert's Law

Note when the light is on the wrong side of the surface, 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

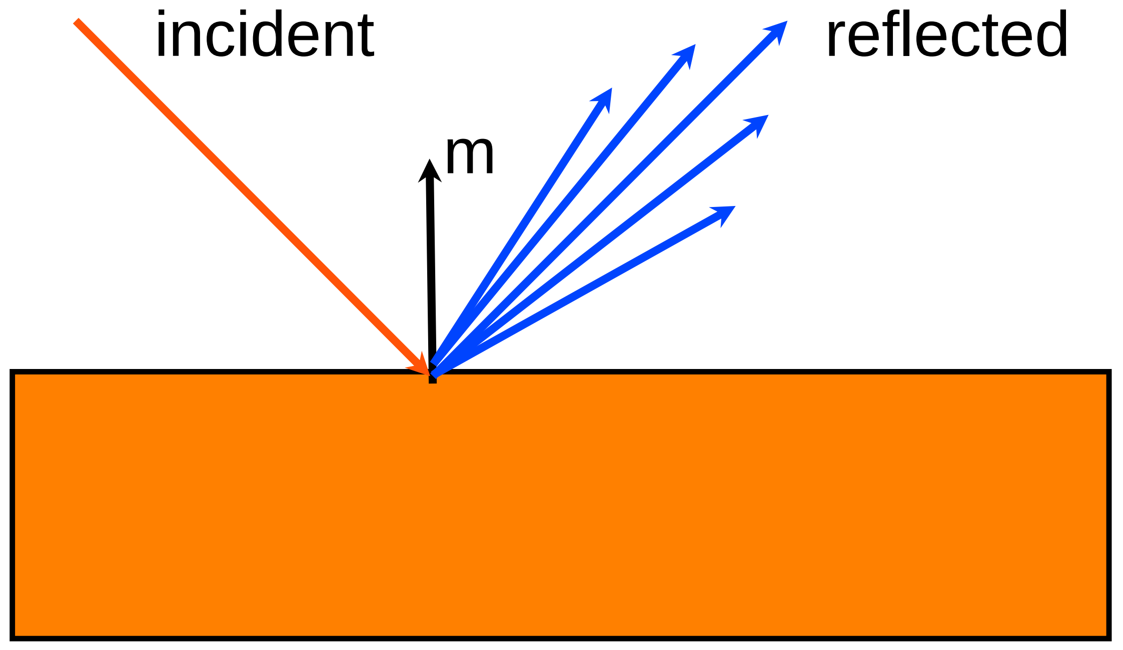
The coefficient ρ_d is a property of the surface.

- 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

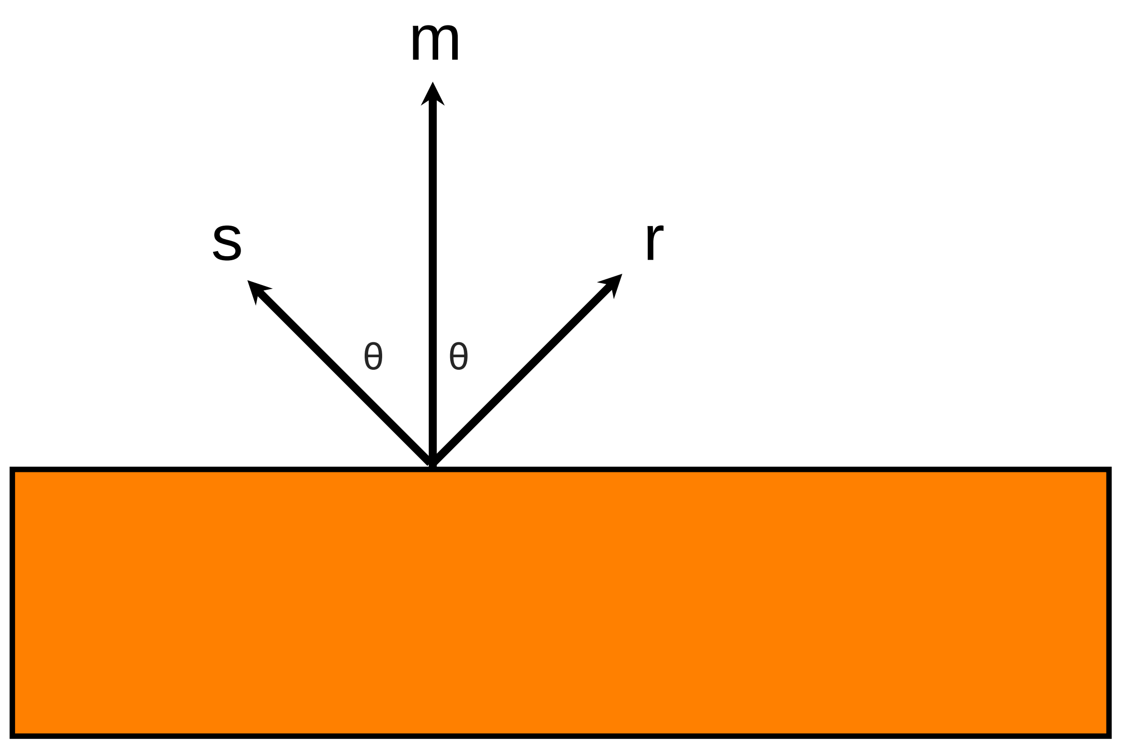
The Phong model is an approximate model of specular reflection. It allows us to add highlights to shiny surfaces.

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:

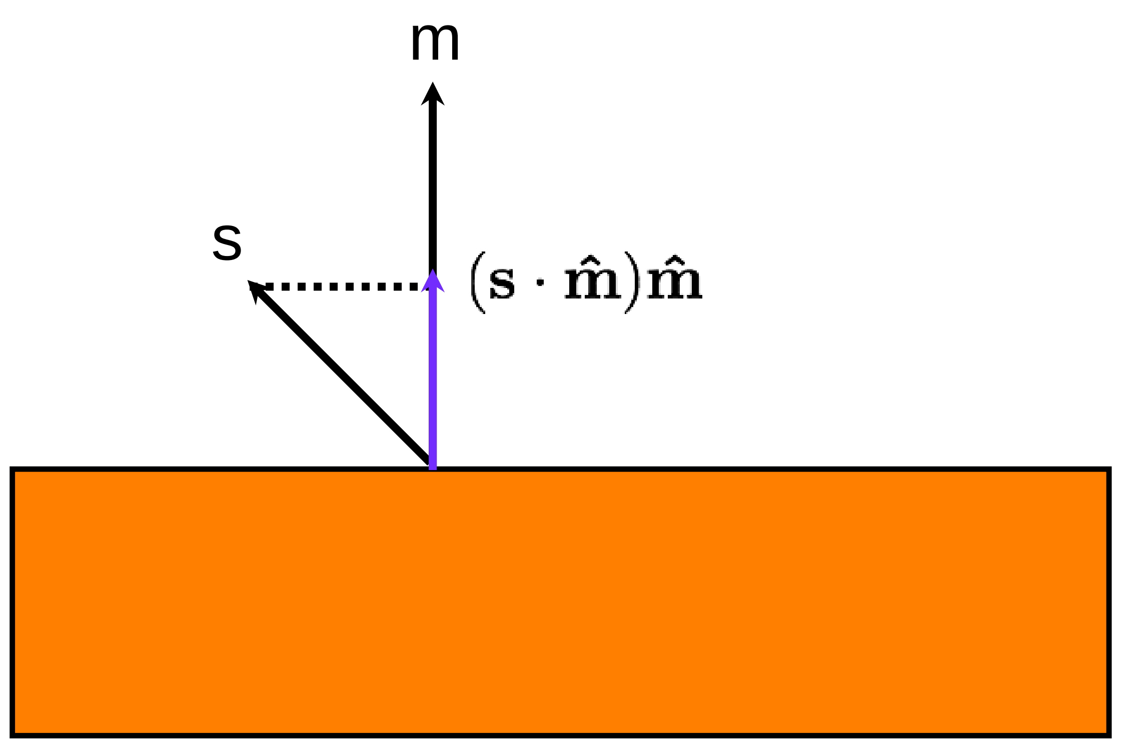
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

Reflection is brightest around the reflection vector :

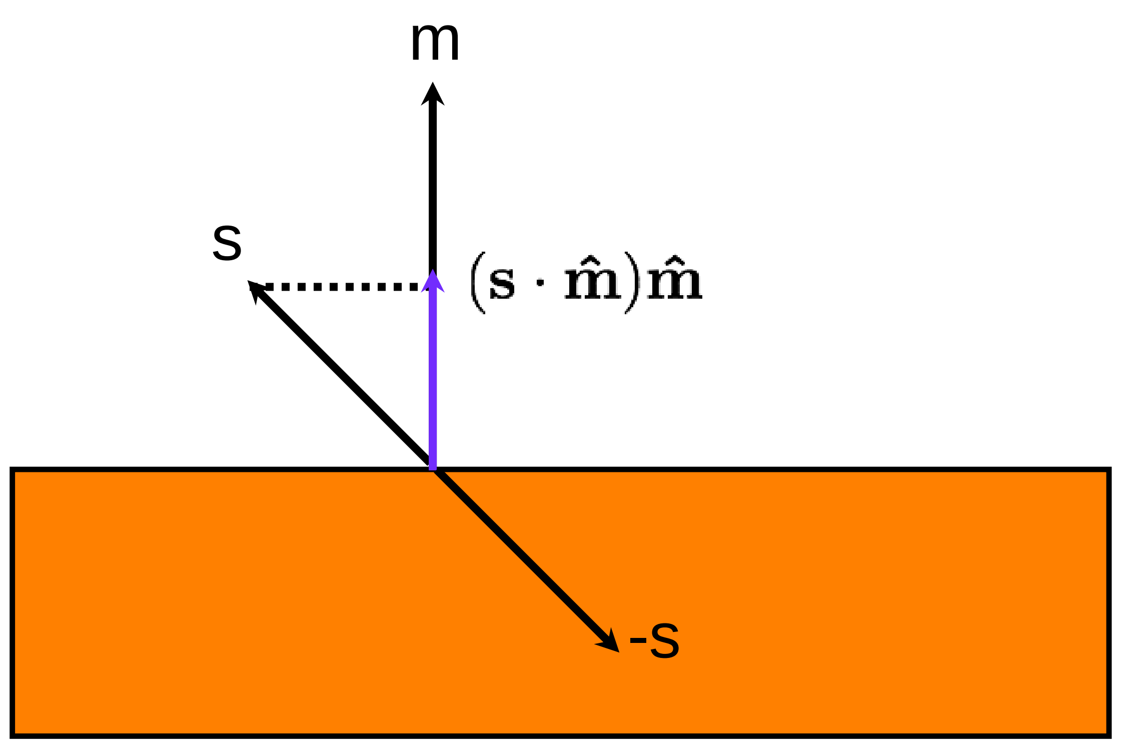
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

Reflection is brightest around the reflection vector:

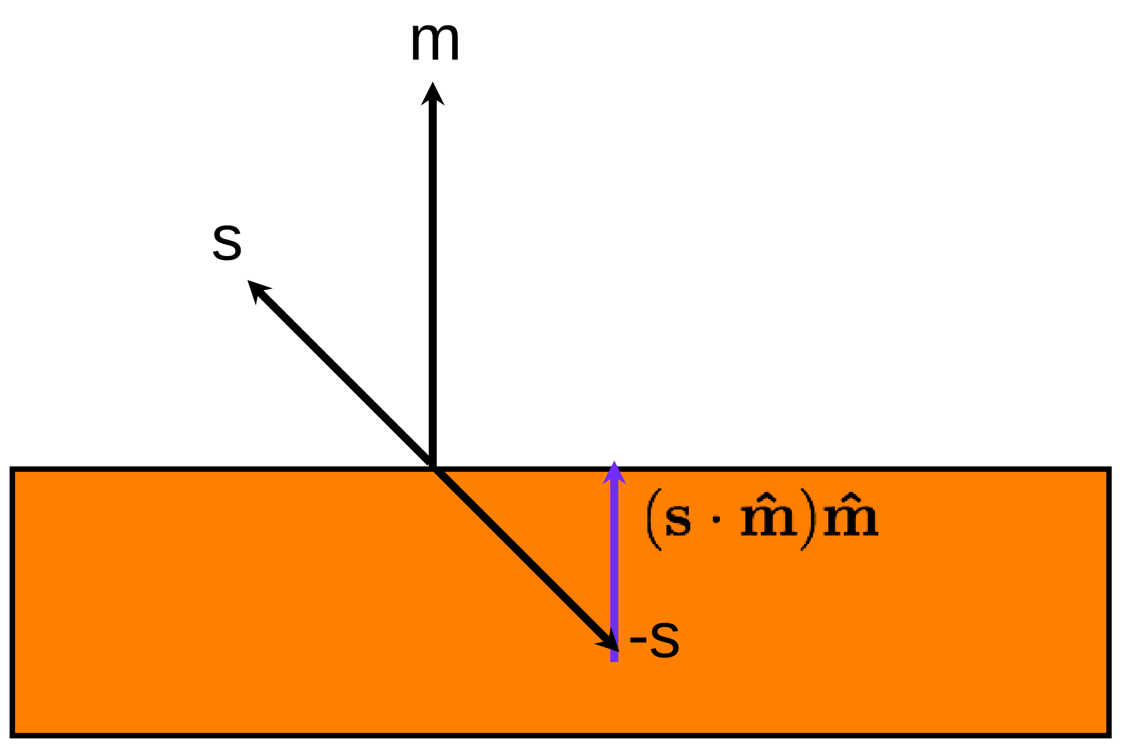
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

Reflection is brightest around the reflection vector:

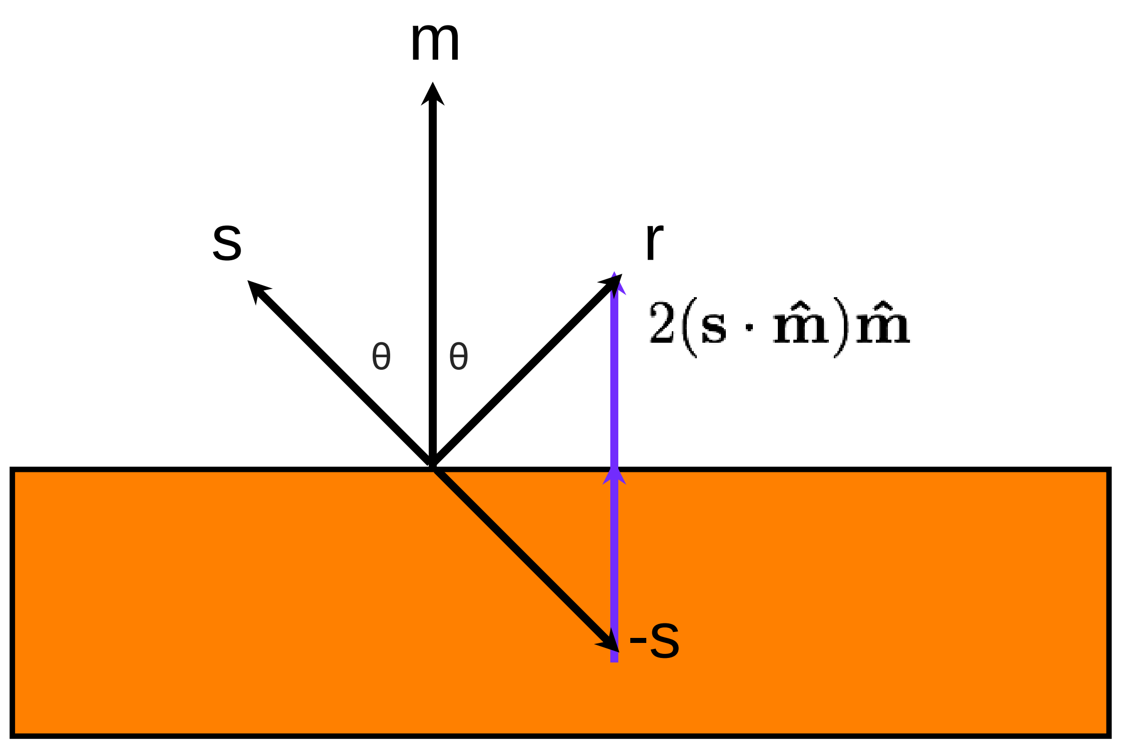
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

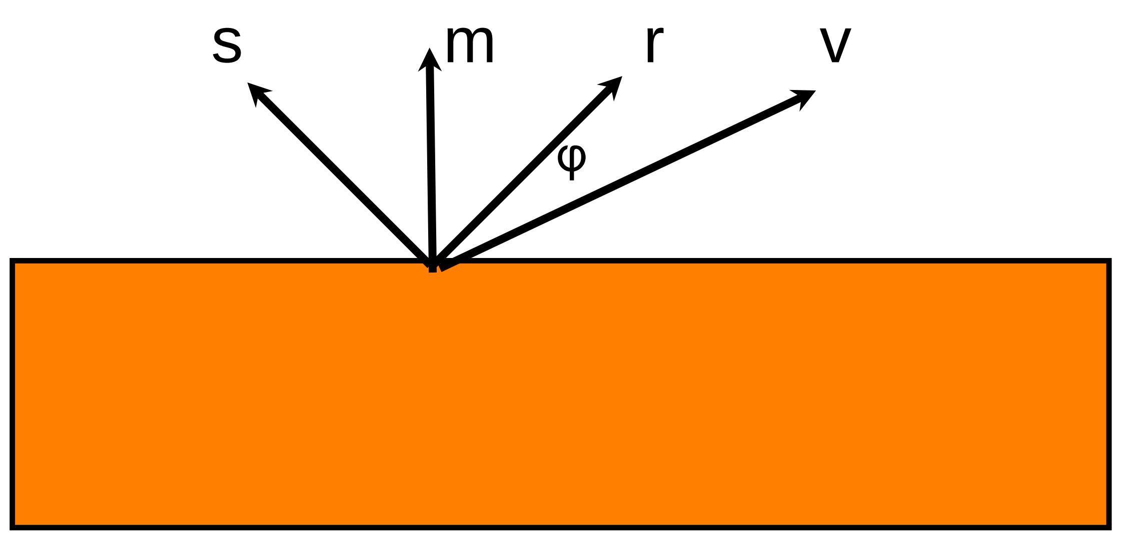
Reflection is brightest around the reflection vector:

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



Phong model

The intensity falls off with the angle φ between the reflected vector and the view vector.



Phong model

The Phong equation is:

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

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

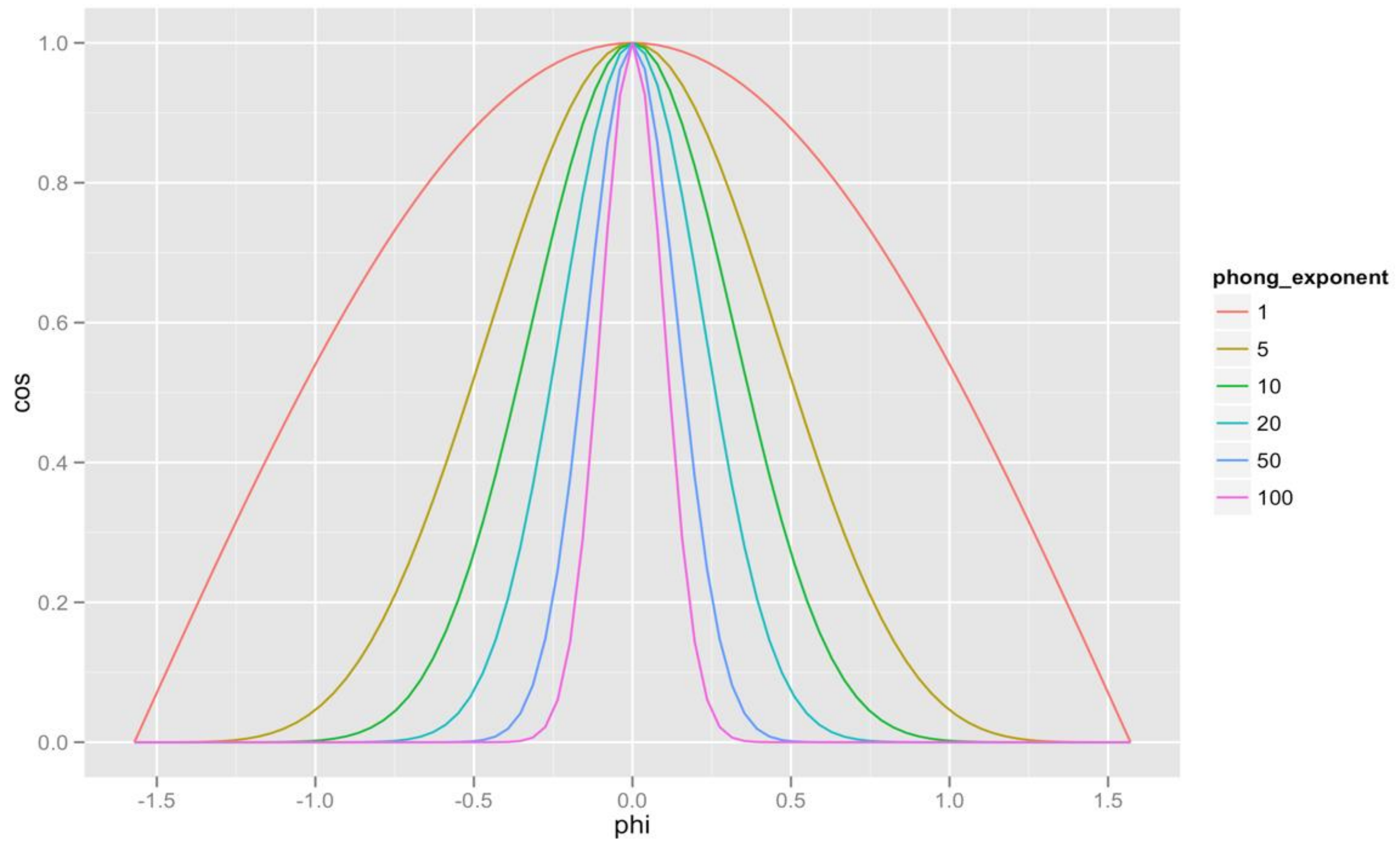
where:

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(φ)^{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.

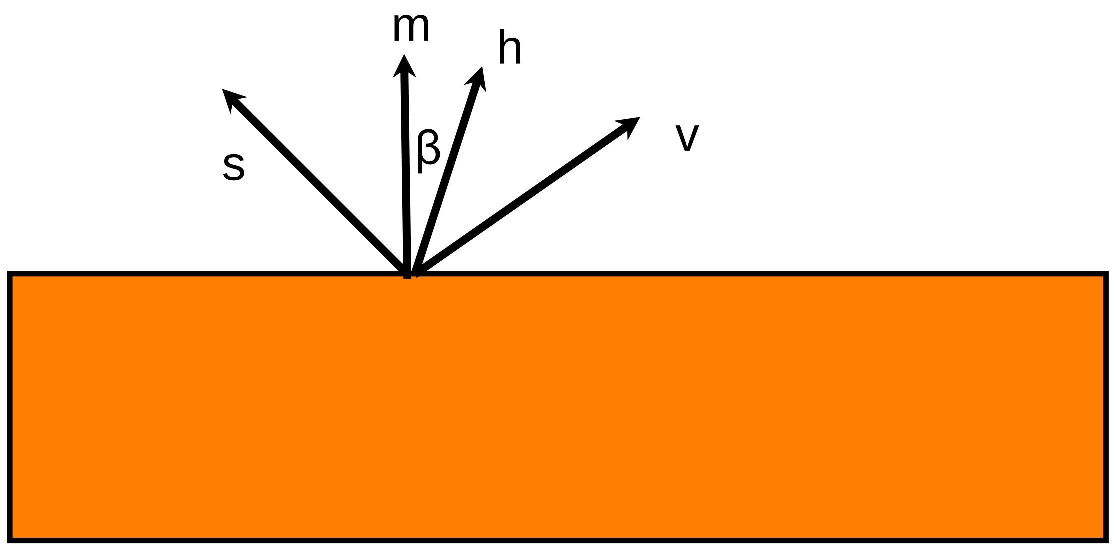
Blinn-Phong Specular Light

Note: where s and v are normalised

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

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

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



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.

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

lambert = $\max(0, \hat{\boldsymbol{s}} \,.\, \widehat{\boldsymbol{m}})$

blinnPhong = $\max(0, \widehat{\boldsymbol{h}} \,.\, \widehat{\boldsymbol{m}})$

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.

$$\mathrm{I} = \underset{\substack{\big\uparrow \\ \text{emissive} \\ \text{light}}}{\rho_{\mathrm{e}}} + \underset{\substack{\big\uparrow \\ \text{global} \\ \text{ambient} \\ \text{term}}}{\mathrm{I}_{ga}\rho_{\mathrm{a}}} + \sum_{l\, \in lights} (\mathrm{I}_{\mathrm{a}}\rho_{\mathrm{a}\,+}\,\mathrm{I}_{\mathrm{d}}\rho_{\mathrm{d}} * \mathrm{lambert} + \mathrm{I}_{\mathrm{sp}}\rho_{\mathrm{sp}*}\mathrm{blinnPhong})$$

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 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();
```

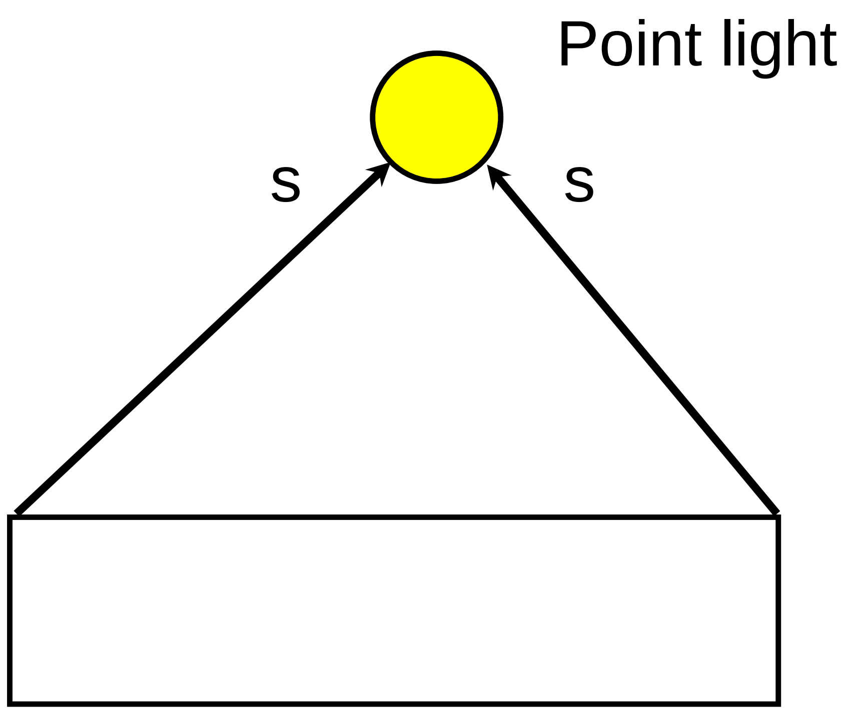
Alpha Values

- 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



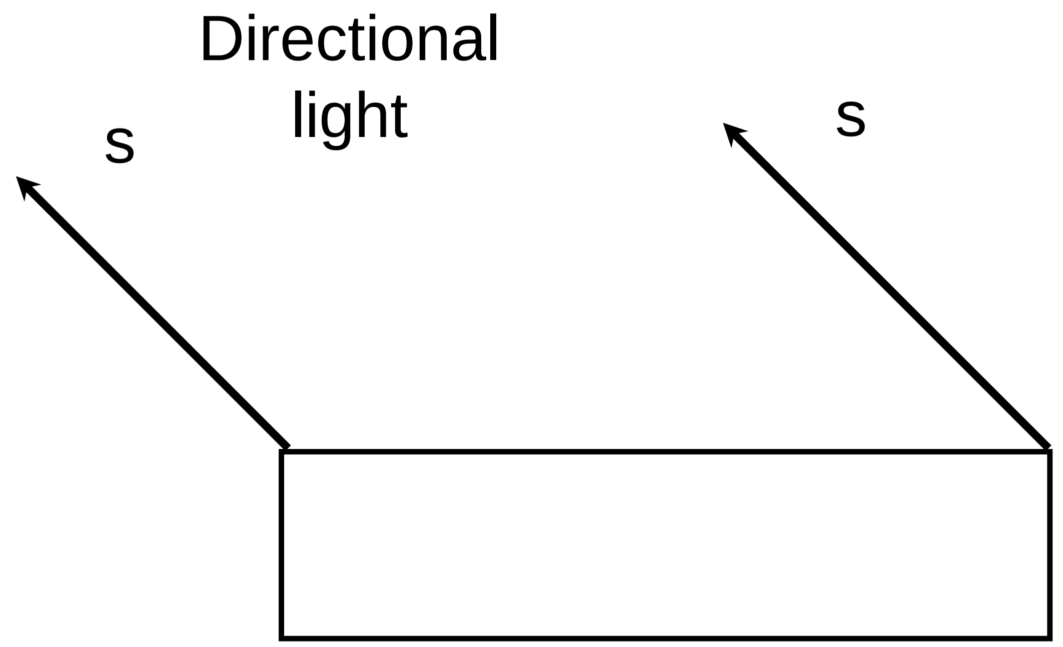
Positional lights

```
// set the position to (5,2,3) in the
// current coordinate frame
// Note: we use homogeneous cords
// 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 effective the same everywhere.

These are called directional lights.



Point and directional lights

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

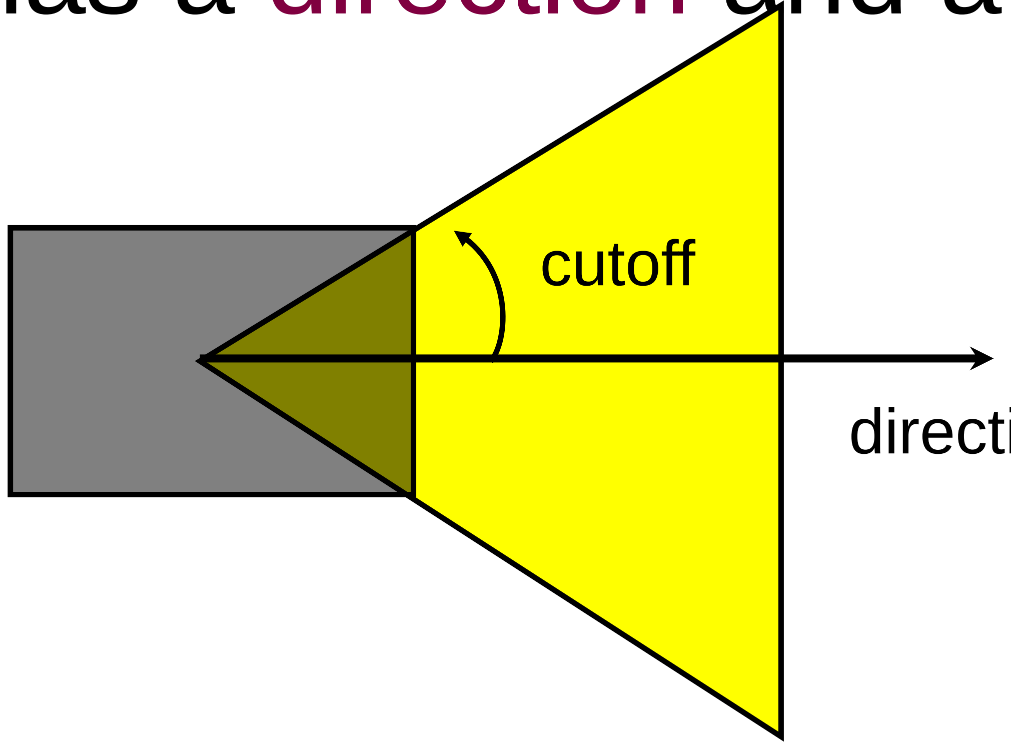
```
// a light pointing straight down
// note: the fourth component is 0

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

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

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,

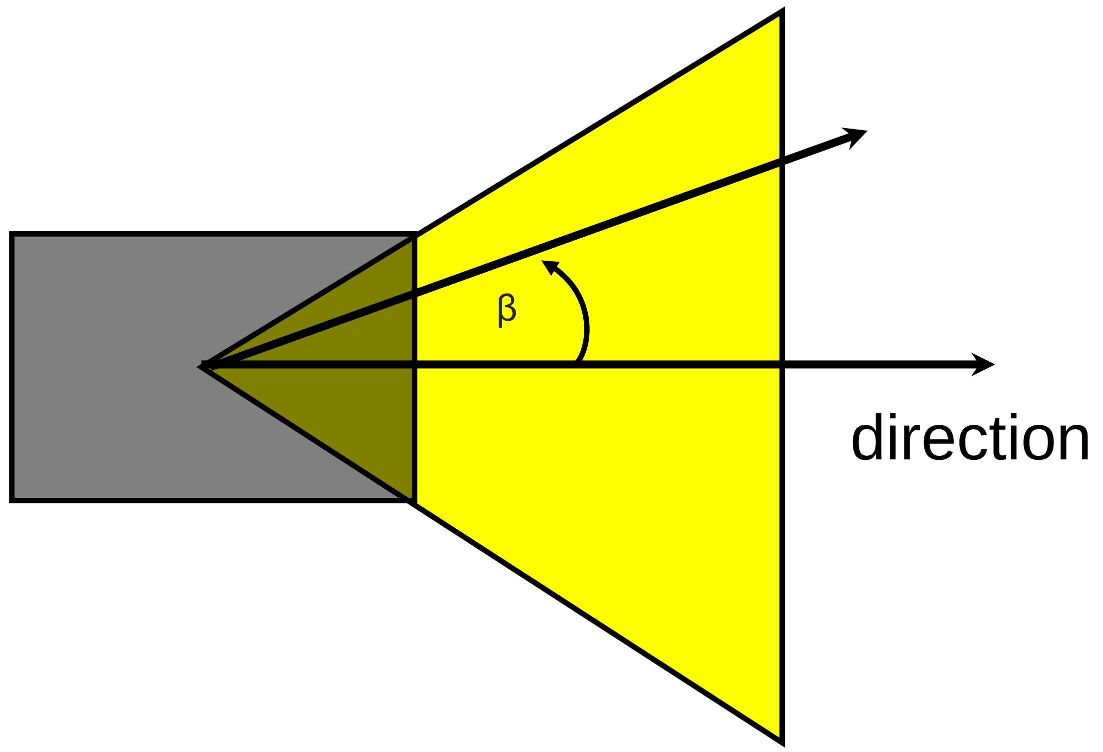


Spotlights

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);
```

Distance attenuation

All real light sources also lose intensity with distance. We usually ignore this.

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);
```

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) while the modelview transform is the identity

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. In particular, the lights' positions must be set before rendering any geometry.

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?

Solution 1

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

Specular (1,1,1,1)

Light: Specular,diffuse(1,1,1,1)

The co-efficients get multiplied so for diffuse we will get 0 for everything but **red.**

For specular we will get 1 for all so it will be **white.**

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?

Solution 2

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

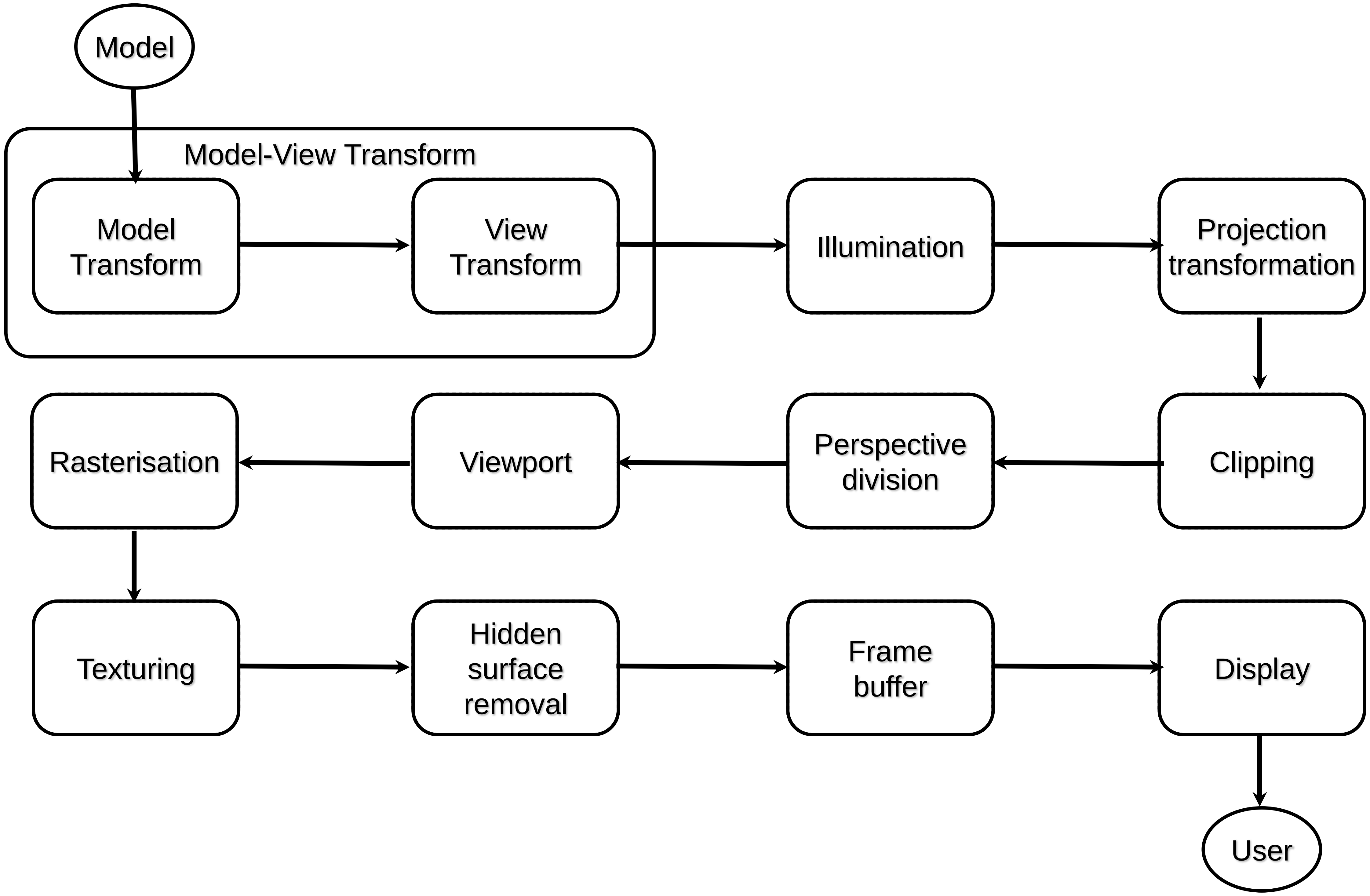
Specular (1,1,1,1)

Light: Specular, diffuse(0,0,1,1)

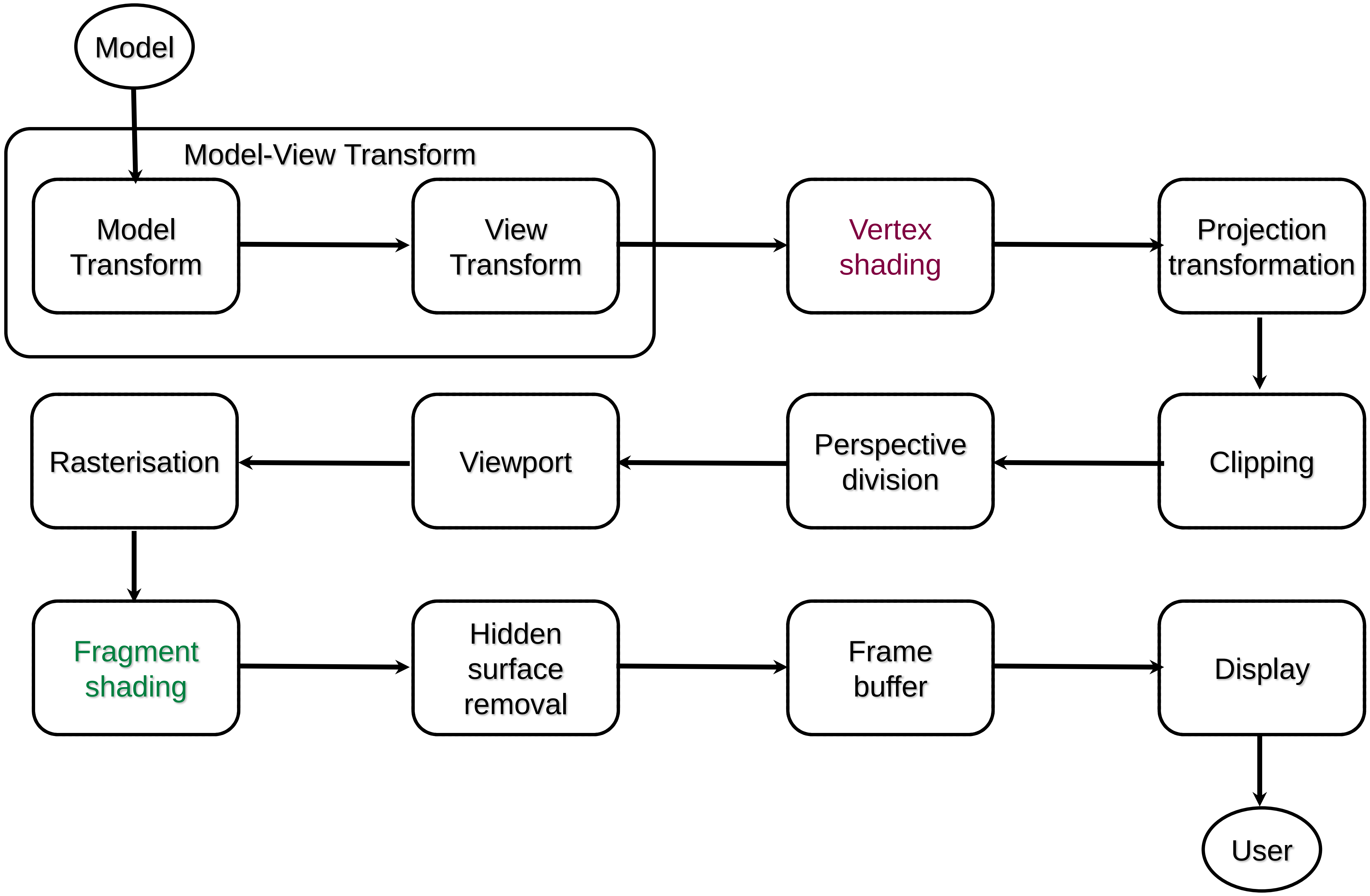
The co-efficients get multiplied so for diffuse we will get 0 for everything except the alpha channel so we get **black**.

For specular we will get 1 for **blue.**

The graphics pipeline



The graphics pipeline



Shading

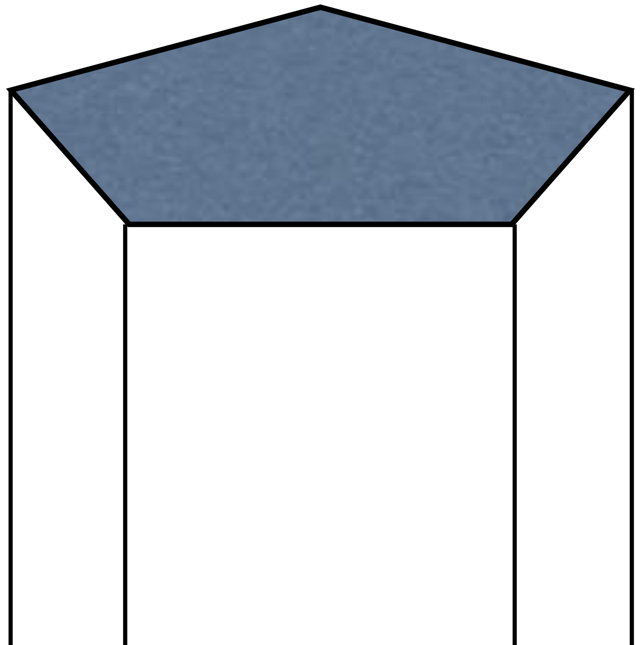
Illumination (a.k.a shading) is done at two points in the fixed function pipeline:

- Vertices in the model are shaded before projection.
- Pixels (fragments) are shaded in the image after rasterisation.
- Doing more work at the vertex level is more efficient. Doing more work at the pixel level can give better results.

Vertex shading

The built-in lighting in OpenGL is mostly done as vertex shading.

The lighting equations are calculated for each vertex in the image using the associated vertex normal.

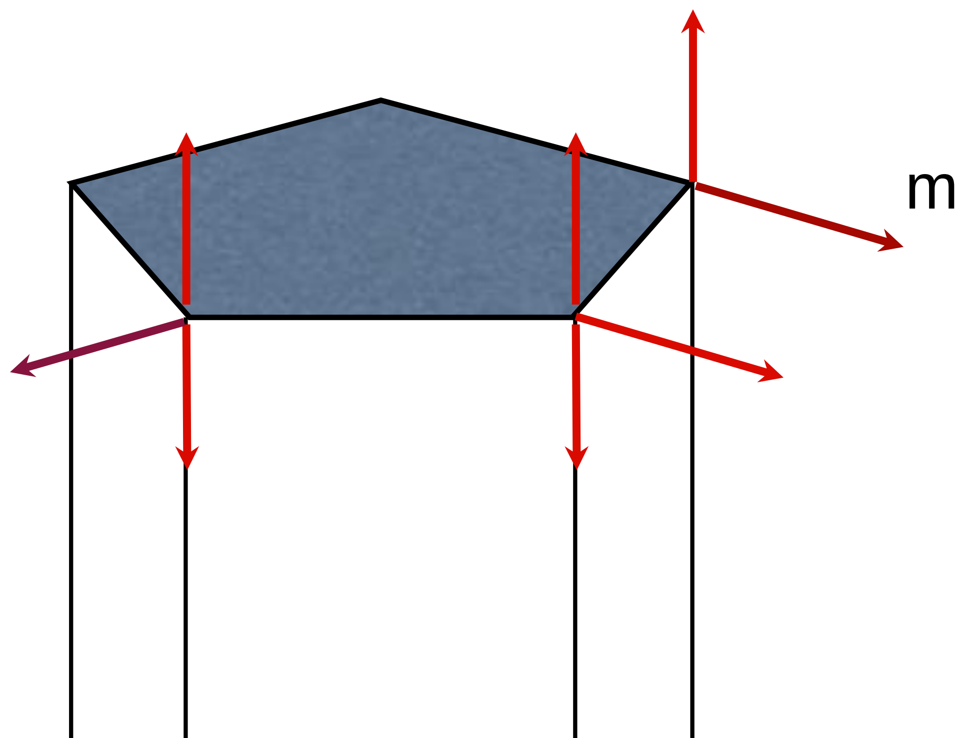


Illumination
is calculated at
each of these
vertices.

Vertex shading

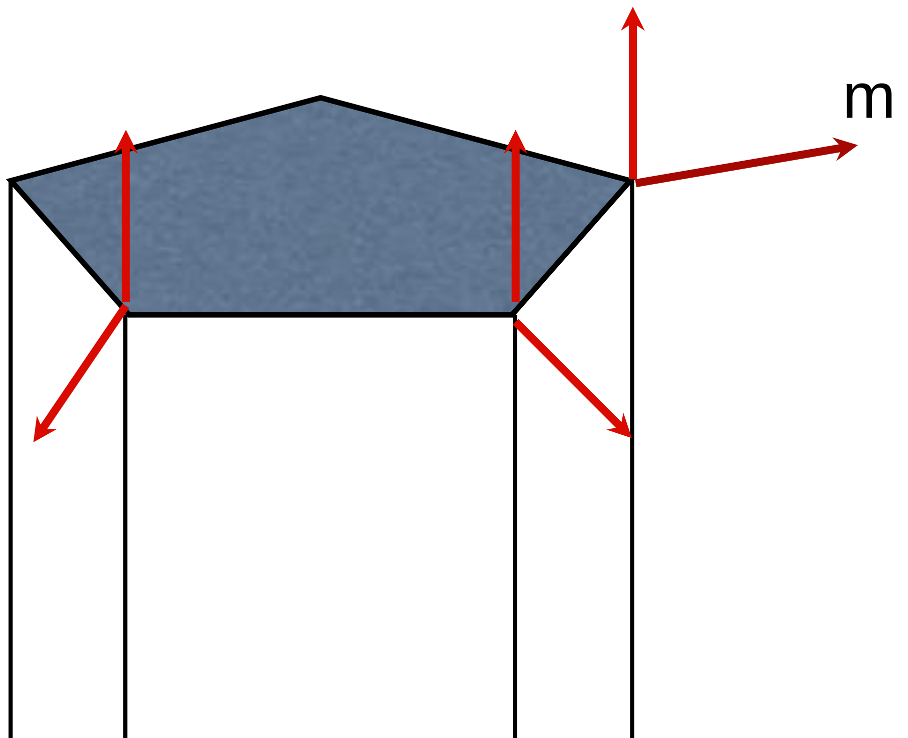
The normal vector **m** used to compute the lighting equation is the normal we specified when creating the vertex using

```
gl.glNormal3d(mx, my, mz);
```



Vertex shading

This is why we use different normals on curved vs flat surfaces, so the vertex may be lit properly.



Vertex shading

Illumination values are attached to each vertex and carried down the pipeline until we reach the fragment shading stage.

```
struct vert {

   float[4] pos;   // vertex coords
   float[4] light; // rgba colour
                   // other info...

}
```

Fragment shading

We need to translate vertex illumination values into appropriate colours for every pixel that makes up the polygon.

There are three common options:

- Flat shading
- Gouraud shading
- Phong shading

In OpenGL

```
// Flat shading :
gl.glShadeModel(GL2.GL_FLAT);
// Gouraud shading (default):
gl.glShadeModel(GL2.GL_SMOOTH);
// Phong shading:
// No built-in implementation
```

Flat shading

The simplest option is to shade the entire face the same colour:

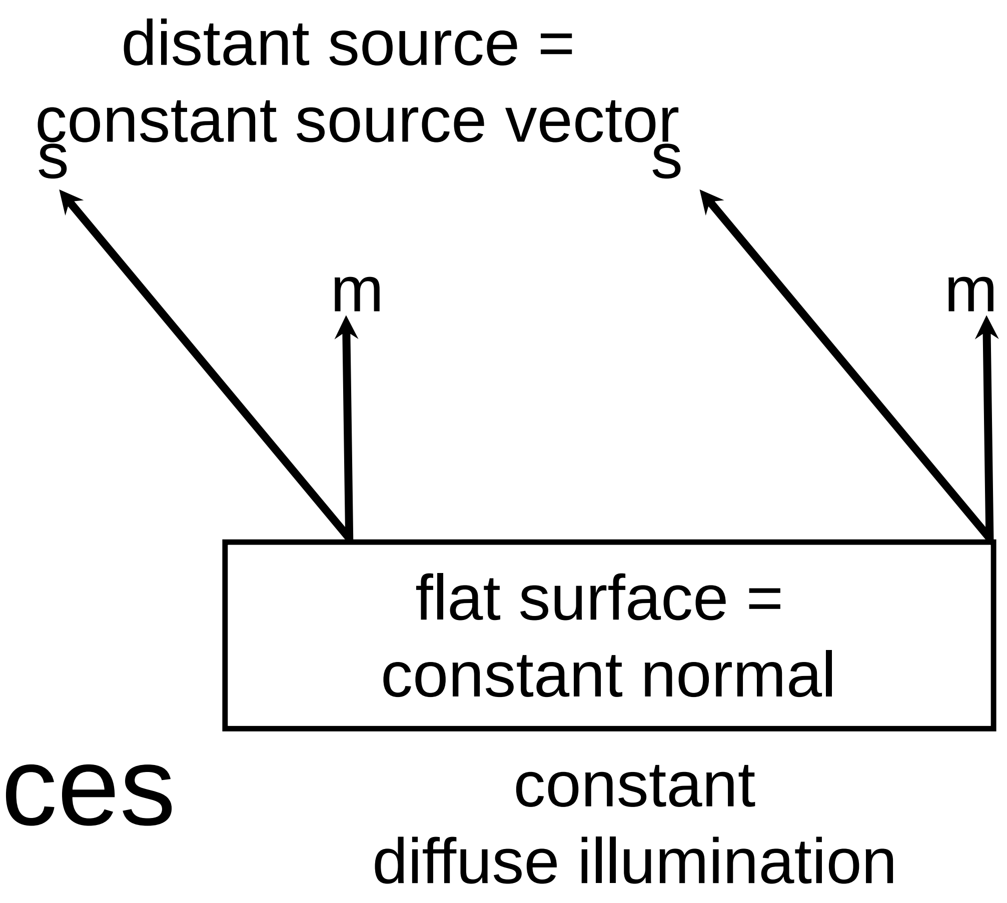
- Choose one vertex (first for a polygon, third for a triangle…)
- Take the illumination of that vertex
- Set every pixel to that value.

Flat shading

Flat shading is good for:

- Diffuse illumination
- for flat surfaces
- with distant light sources

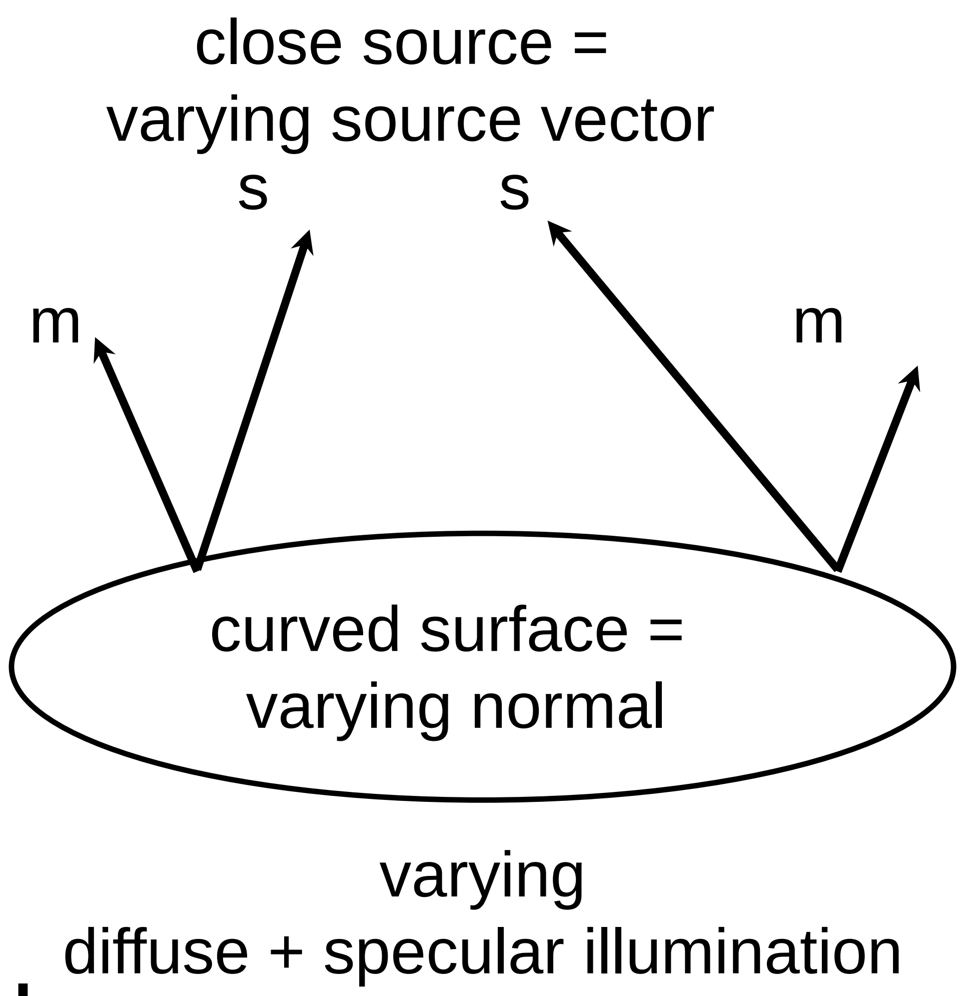
It is the fastest shading option.



Flat shading

Flat shading is bad for:

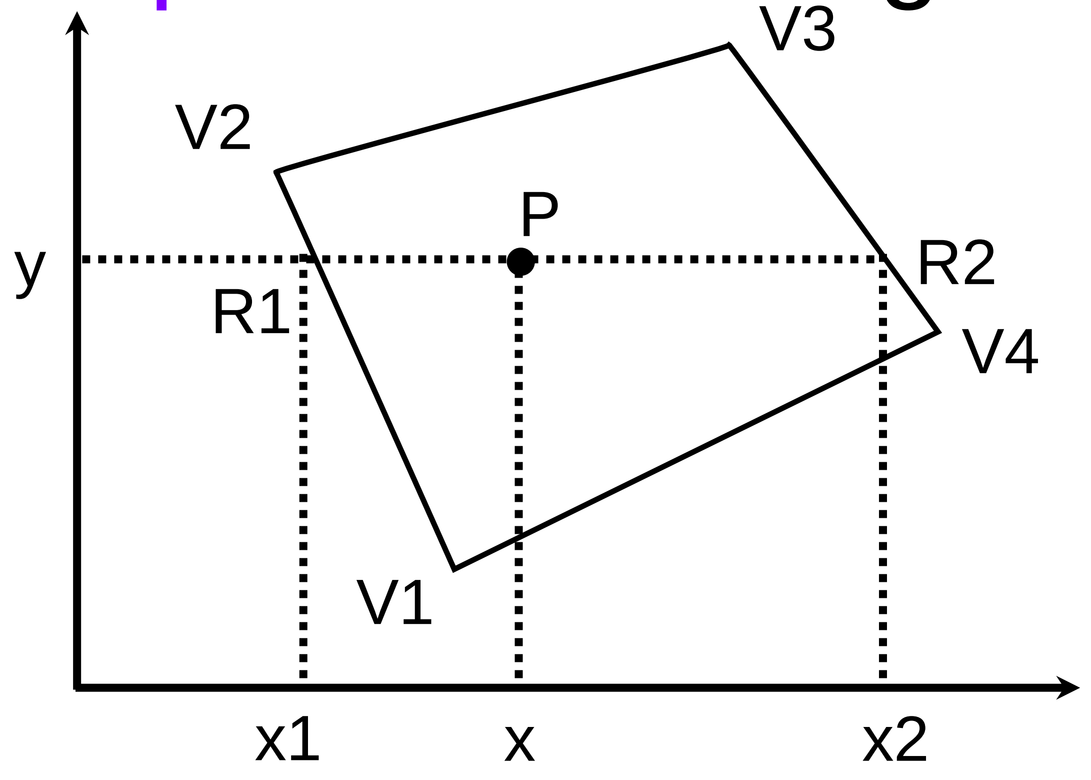
- close light sources
- specular shading
- curved surfaces
  - Edges between faces become more pronounced than they actually are (Mach banding)



Gouraud shading

Gouraud shading is a simple smooth shading model.

We calculate fragment colours by bilinear interpolation on neighbouring vertices.

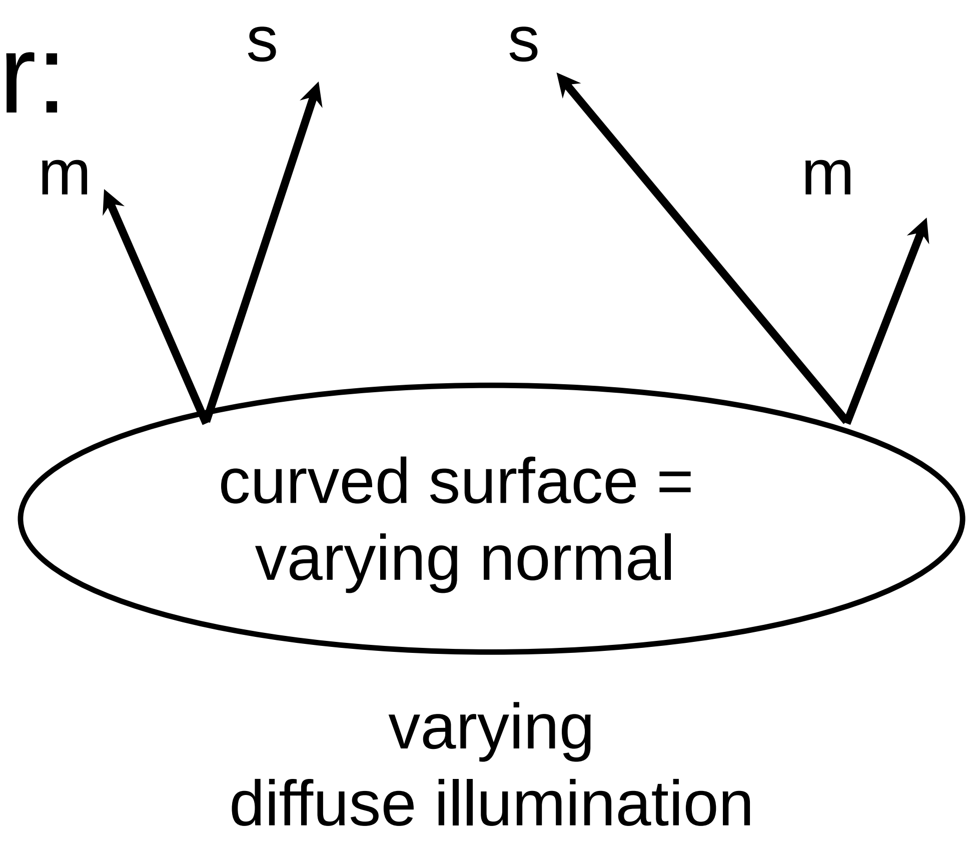


$$
\begin{aligned}
colour(R_1) &= lerp(V_1, V_2, \tfrac{y-y_1}{y_2-y_1}) \\
colour(R_2) &= lerp(V_3, V_4, \tfrac{y-y_3}{y_4-y_3}) \\
colour(P) &= lerp(R_1, R_2, \tfrac{x-x_1}{x_2-x_1})
\end{aligned}
$$

Gouraud shading

Gouraud shading is good for:

- curved surfaces
- close light sources
- diffuse shading



Gouraud shading

Gouraud shading is only slightly more expensive than flat shading.

It handles specular highlights poorly.

- It works if the highlight occurs at a vertex.
- If the highlight would appear in the middle of a polygon it disappears.

Phong shading

Phong shading is designed to handle specular lighting better than Gouraud. It also handles diffuse better as well.

It works by deferring the illumination calculation until the fragment shading step.

So illumination values are calculated per pixel rather than per vertex.

Not implemented on the fixed function pipeline. Need to use the programmable pipeline.

Phong shading

For each pixel we need to know:

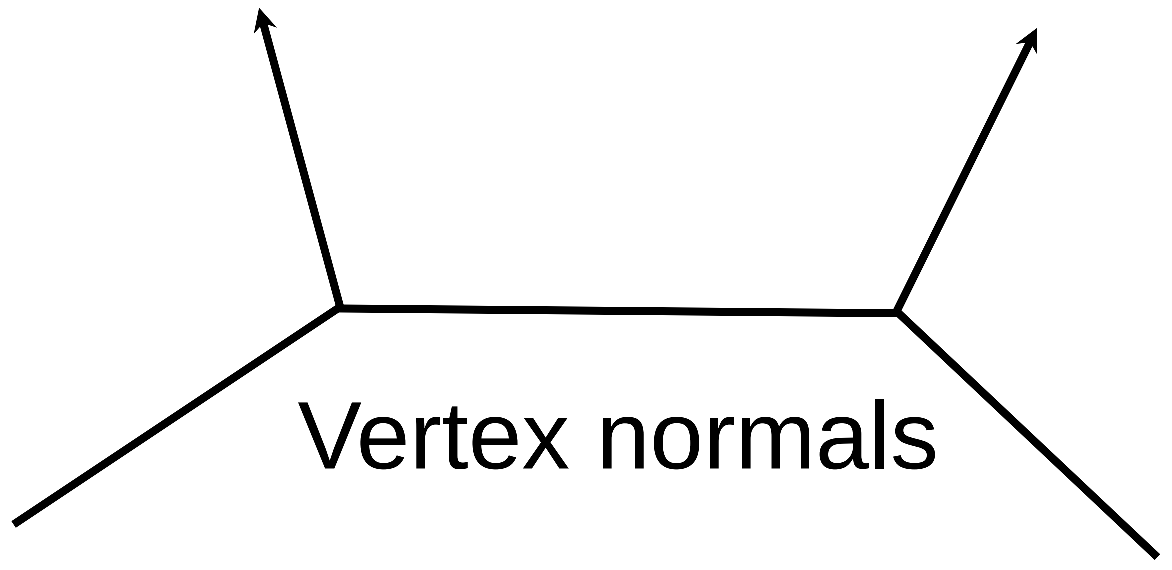
- source vector **s**
- eye vector **v**
- normal vector **m**

Knowing the source location, camera location and pixel location we can compute **s** and **v**.

What about **m**?

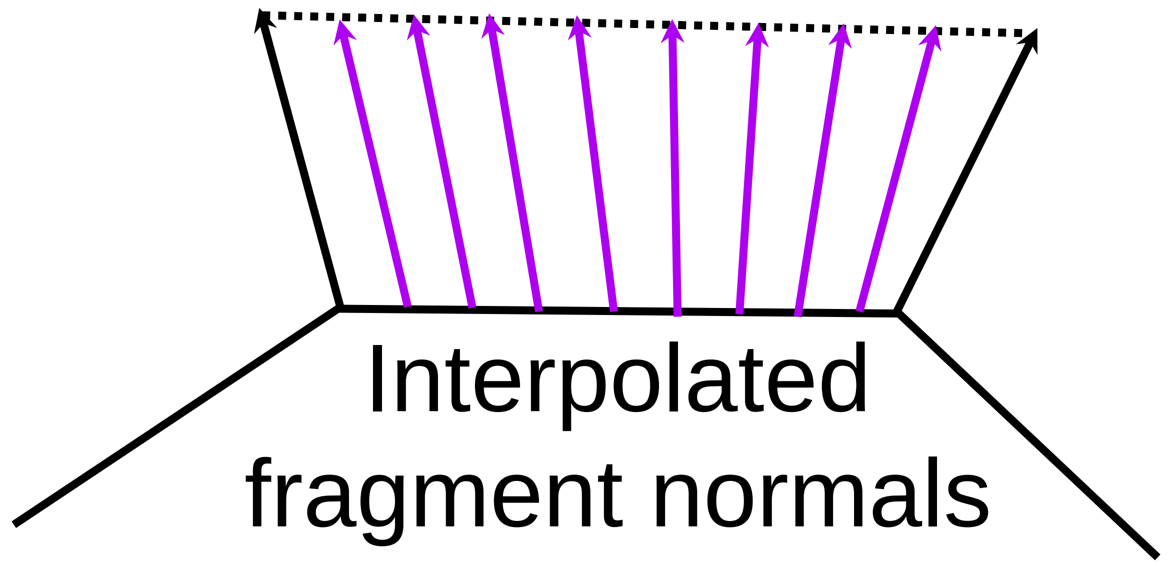
Normal interpolation

Phong shading approximates **m** by interpolating the normals of the polygon.



Normal interpolation

Phong shading approximates **m** by interpolating the normals of the polygon.



Normal interpolation

In a 2D polygon we do this using (once again) bilinear interpolation.

However the interpolated normals will vary in length, so they need to be normalised (set length = 1) before being used in the lighting equation.

Phong shading

Pros:

- Handles specular lighting well.
- Improves diffuse shading
- More physically accurate

Phong shading

Cons:

- Slower than Gouraud . Normals and illumination values have to be calculated per pixel rather than per vertex.