COMP3421

Textures 2 and Rasterisation

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

lostandtaken.com

opengameart.org/textures

textures.com

Rendering to a texture

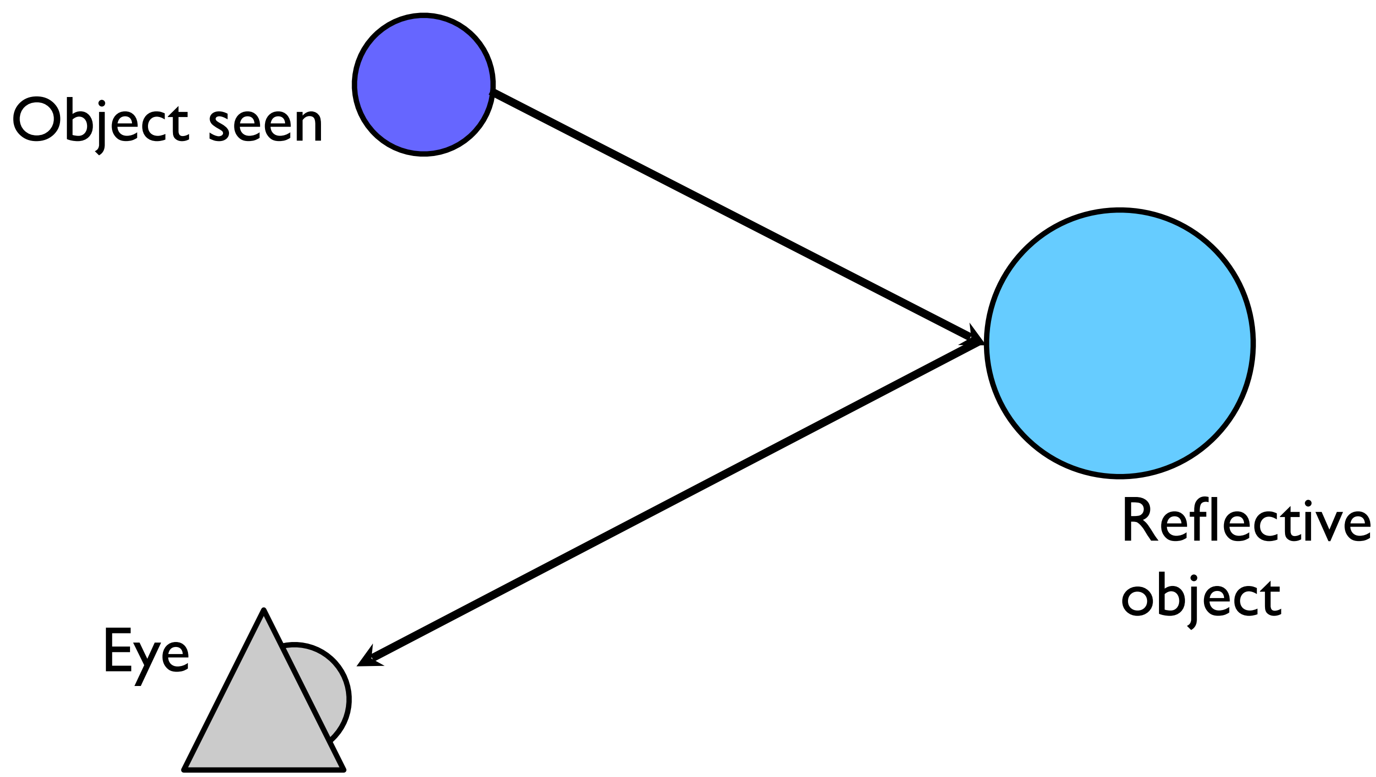
A common trick is to set up a camera in a scene, render the scene into an offscreen buffer, then copy the image into a texture to use as part of another scene.

E.g. Implementing a security camera in a game.

```
//In openGL you can use

gl.glCopyTexImage2D(…);
```

Reflection

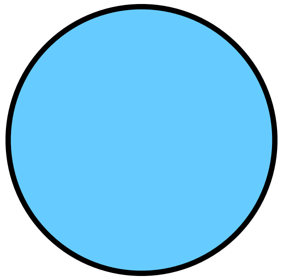
- To do better quality reflections we need to compute where the reflected light is coming from in the scene.



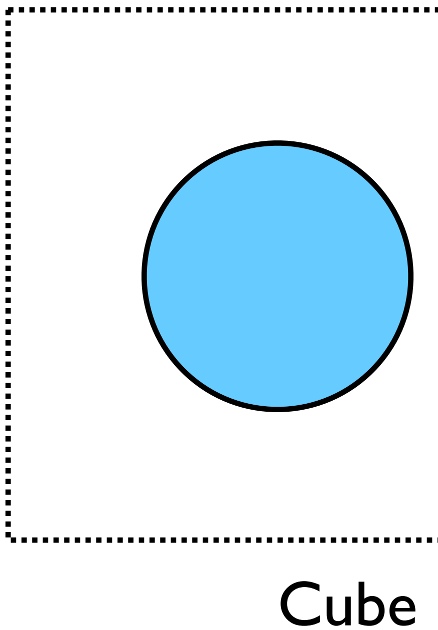
Reflection mapping

- Doing this in general is expensive, but we can make a reasonable approximation with textures:
  - Generate a cube that encloses the reflective object.
  - Place a camera at the centre of the cube and render the outside world onto the faces of the cube.
  - Use this image to texture the object

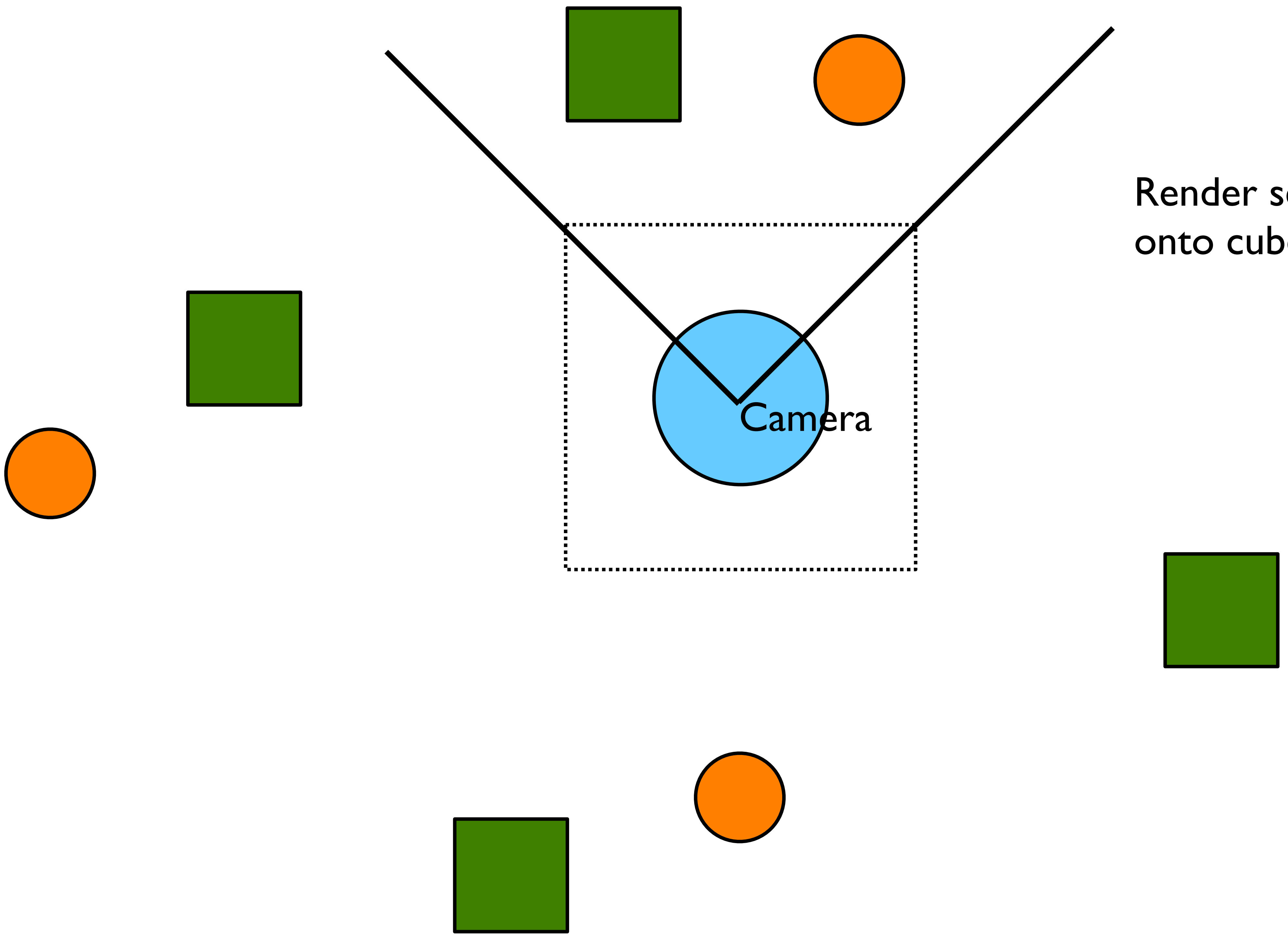
Reflection mapping



Reflection mapping

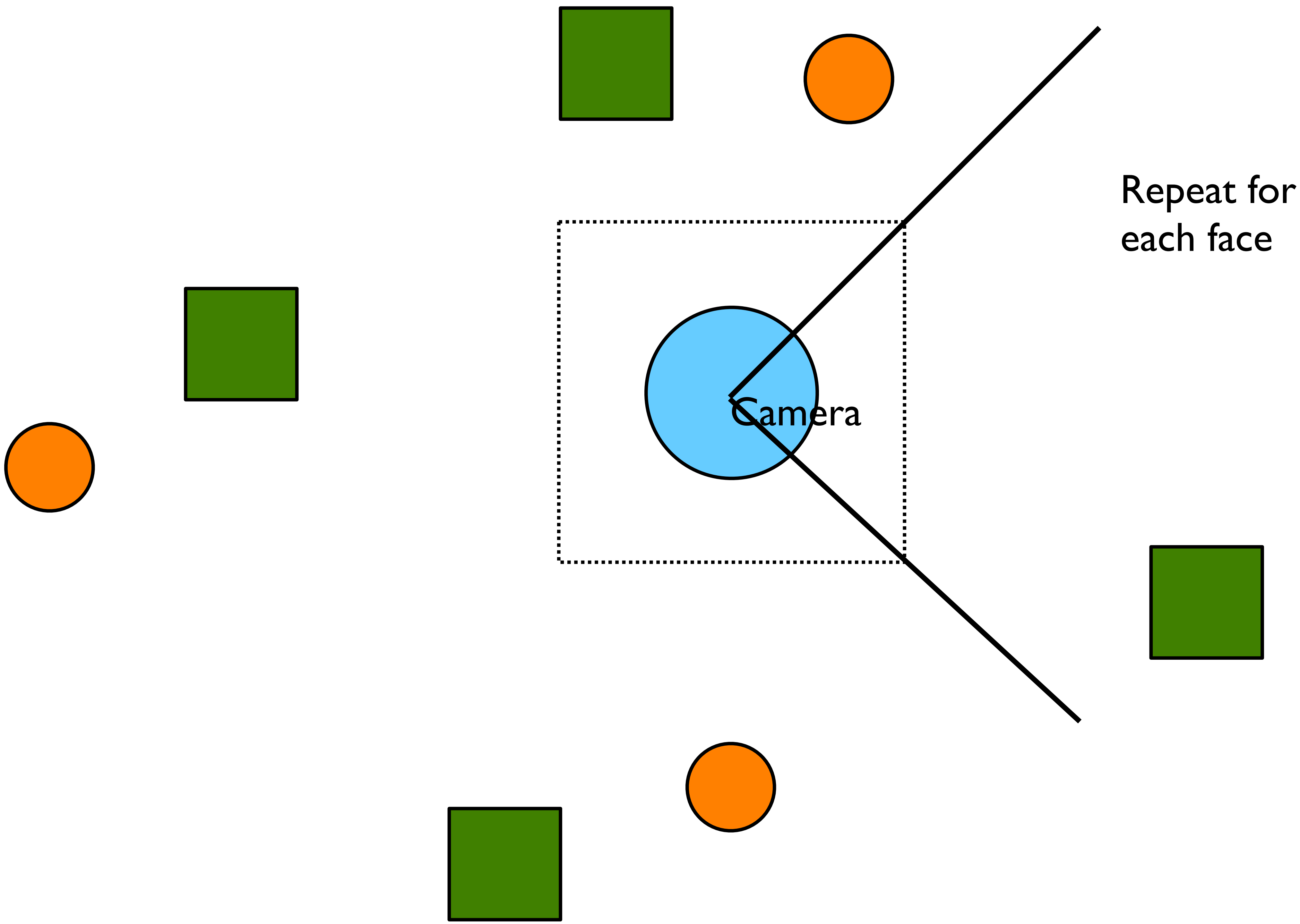


Reflection mapping

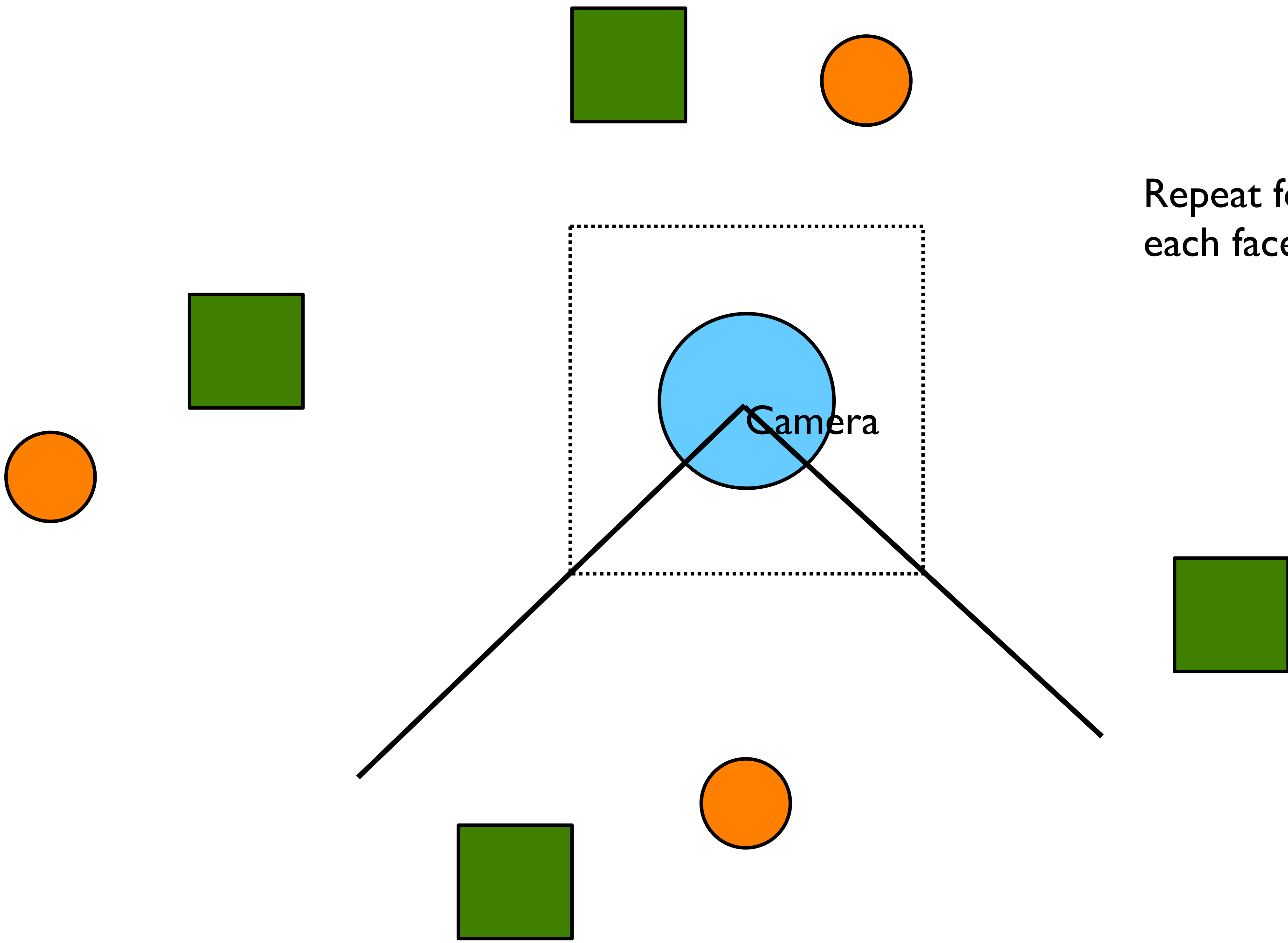


Render scene
onto cube

Reflection mapping

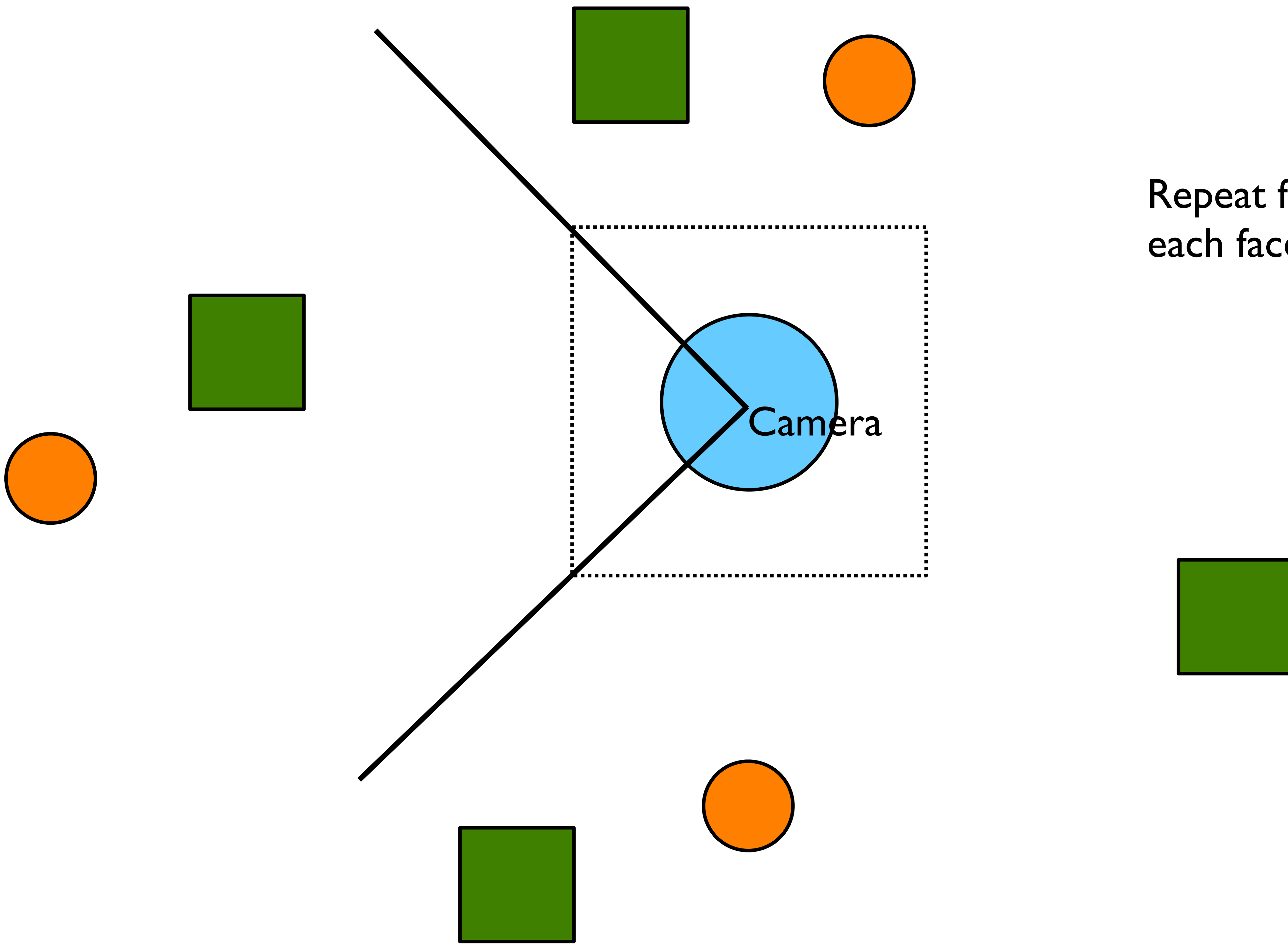


Reflection mapping



Repeat for each face

Reflection mapping



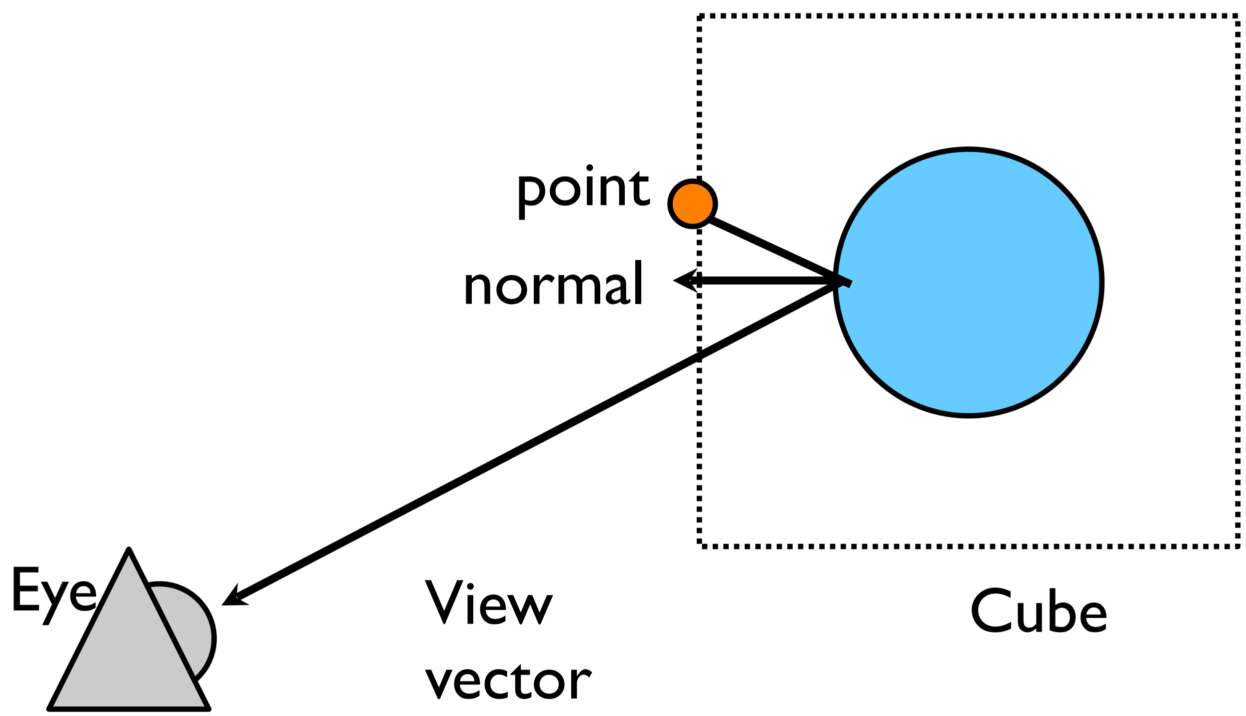
Repeat for each face

Reflection mapping

- To apply the reflection-mapped texture to the object we need to calculate appropriate texture coordinates.

- We do this by tracing a ray from the camera, reflecting it off the object and then calculating where it intersects the cube.

Reflection mapping



Reflection mapping

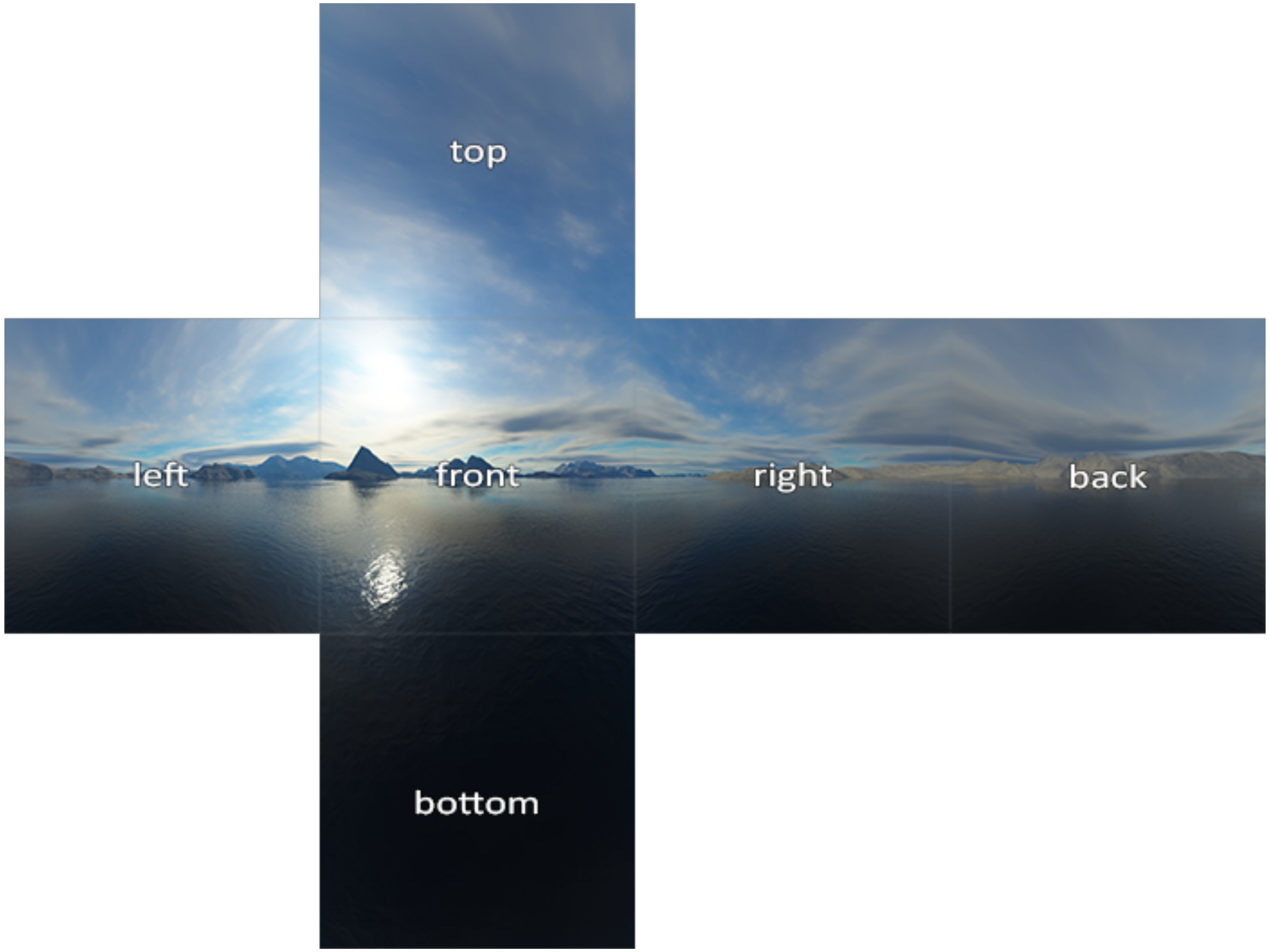
- Pros:
  - Produces reasonably convincing polished metal surfaces and mirrors

Reflection mapping

- Cons:
  - Expensive: Requires 6 additional render passes per object
  - Angles to near objects are wrong.
  - Does not handle self-reflections or recursive reflections.

OpenGL

- Cube maps can also be used for sky boxes.



OpenGL

- OpenGL has built in support for fast approximate reflection mapping (cube mapping).

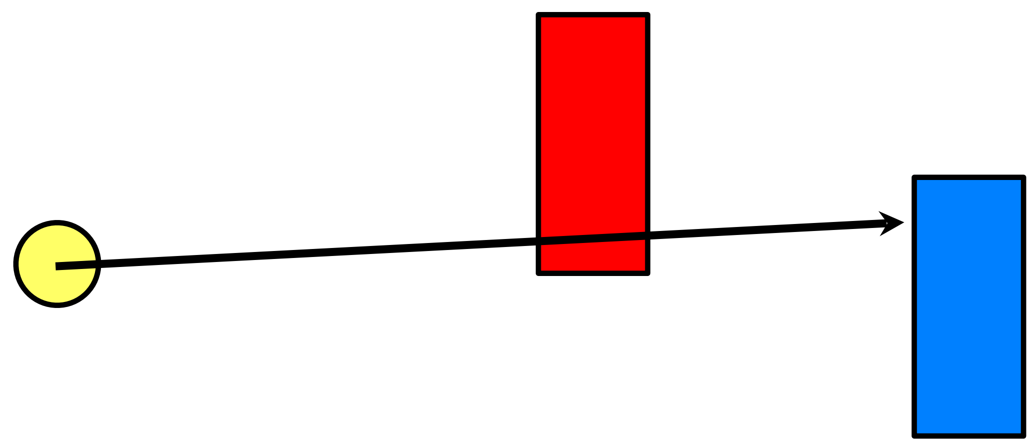
- See ModelViewer

- OpenGL also has sphere mapping support, although this usually produces more distortion and is not as effective as cube mapping.

Shadows

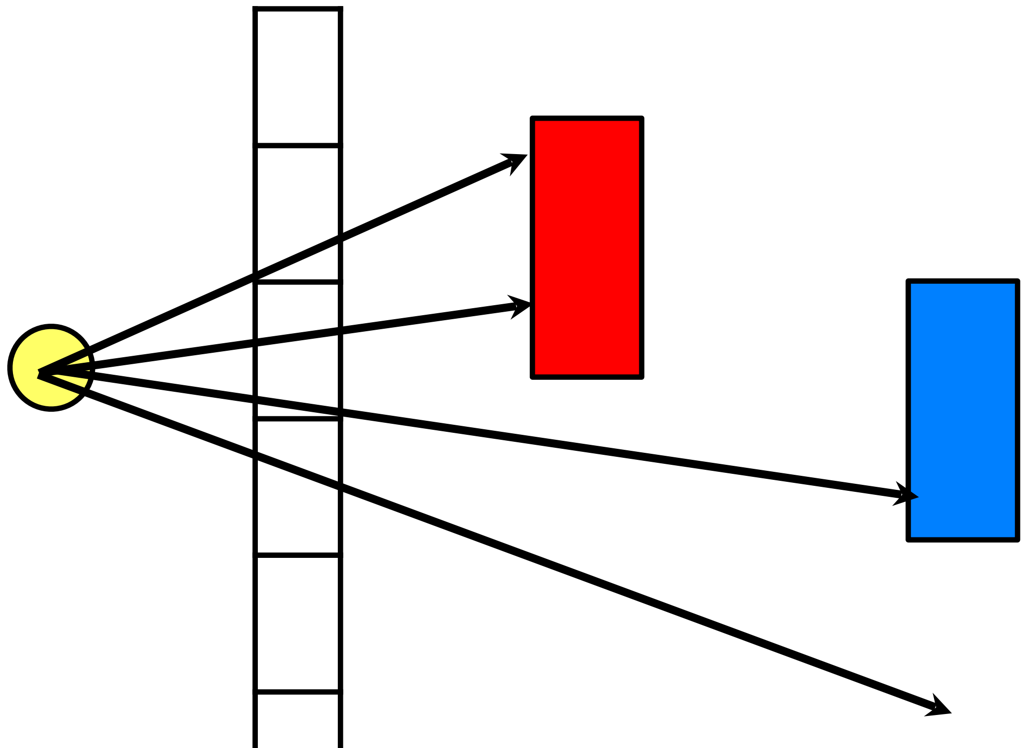
- Our lighting model does not currently produce shadows.

- We need to take into account whether the light source is occluded by another object.



Shadow buffering

- One solution is to keep a shadow buffer for each light source.
- The shadow buffer is like the depth buffer, it records the distance from the light source to the closest object in each direction.



Shadow buffer

- Shadow rendering is usually done in multiple passes:

    1. Render the scene from each light's viewpoint capturing only z-info in shadow (depth) buffer (color buffer turned off)
    2. Render the scene from camera's point of view, using the previously captured shadow buffers to modulate the fragments

Shadow buffer

- When rendering a point P:
  - Project the point into the light's clip space.
  - Calculate the index (i,j) for P in the shadow buffer
  - Calculate the pseudodepth d relative to the light source
  - If shadow[i,j] < d then P is in the shadow

Shadow buffer

- Pros:
  - Provides realistic shadows
  - No knowledge or processing of the scene geometry is required

Shadow buffer

- Cons:
  - More computation
  - Shadow quality is limited by precision of shadow buffer. This may cause some aliasing artefacts.
  - Shadow edges are hard.
  - The scene geometry must be rendered once per light in order to generate the shadow map for a spotlight, and more times for an omnidirectional point light.

OpenGL

- http://www.paulsprojects.net/tutorials/smt/smt.html

Light Mapping

- If our light sources and large portions of the geometry are static then we can precompute the lighting equations and store the results in textures called light maps.

- This process is known as baked lighting.

Light Mapping

- Pros:
  - Sophisticated lighting effects can be computed at compile time, where speed is less of an issue.

Light mapping

- Cons:
  - Memory and loading times for many individual light maps.
  - Not suitable for dynamic lights or moving objects.
  - Potential aliasing effects depending on the resolution of the light maps.

Normal mapping

- When we interpolate normals in a Phong shader we are assuming that the surface of the polygon is smoothly curved.
- What if the surface is actually rough with many small deformities?
- Putting a rough texture on a smooth flat surface looks wrong.

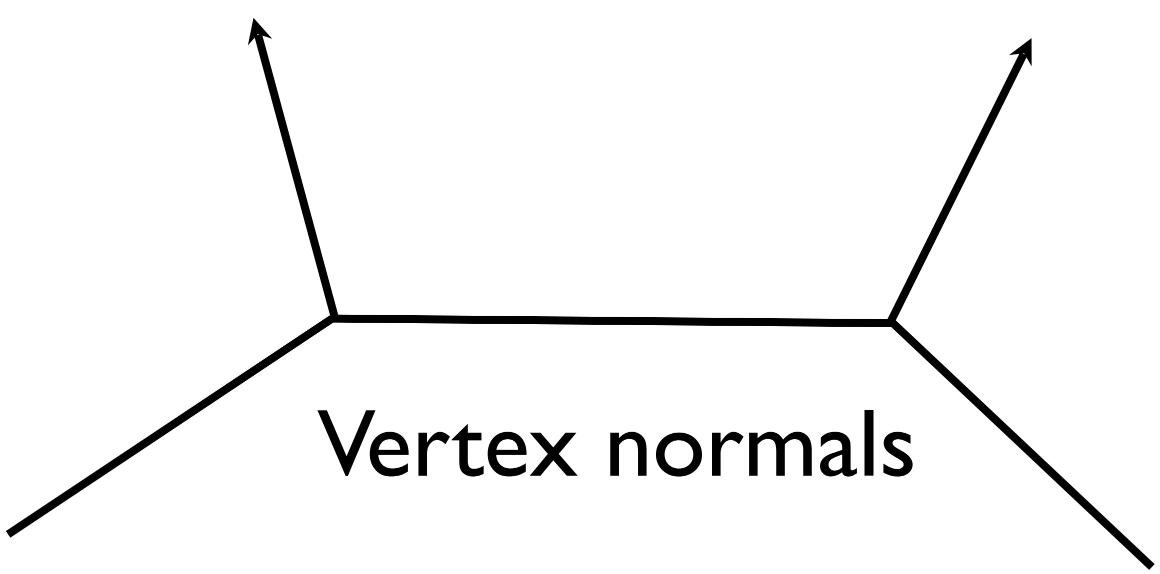
Normal mapping

- One solution would be to increase the number of polygons to represent all the deformities, but this is computationally unfeasible for most applications.

- Instead we use textures called normal maps to simulate minor perturbations in the surface normal.

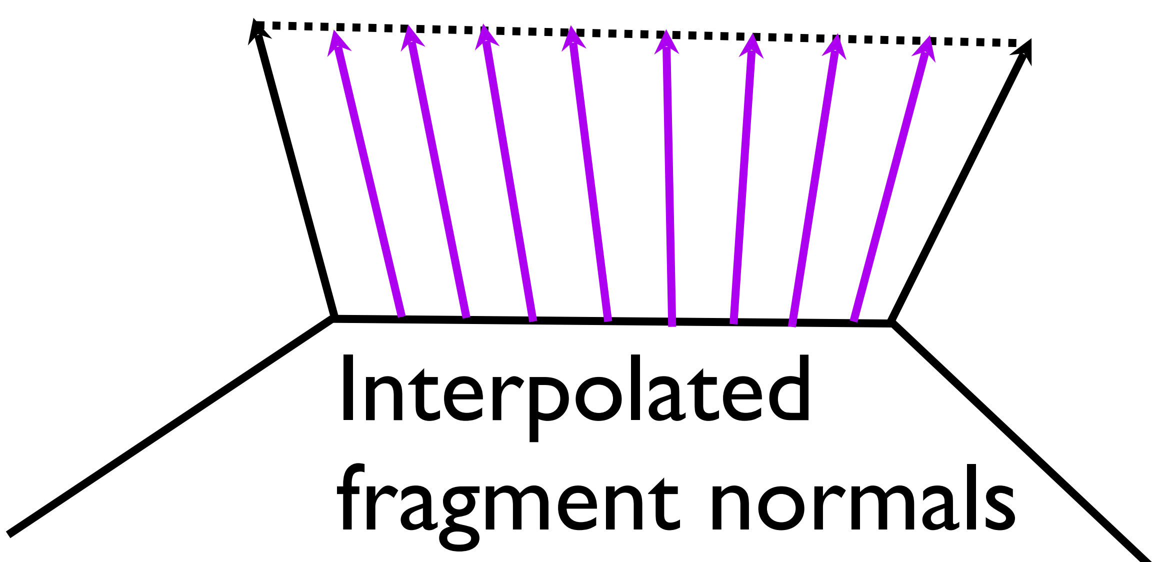
Normal maps

- Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.



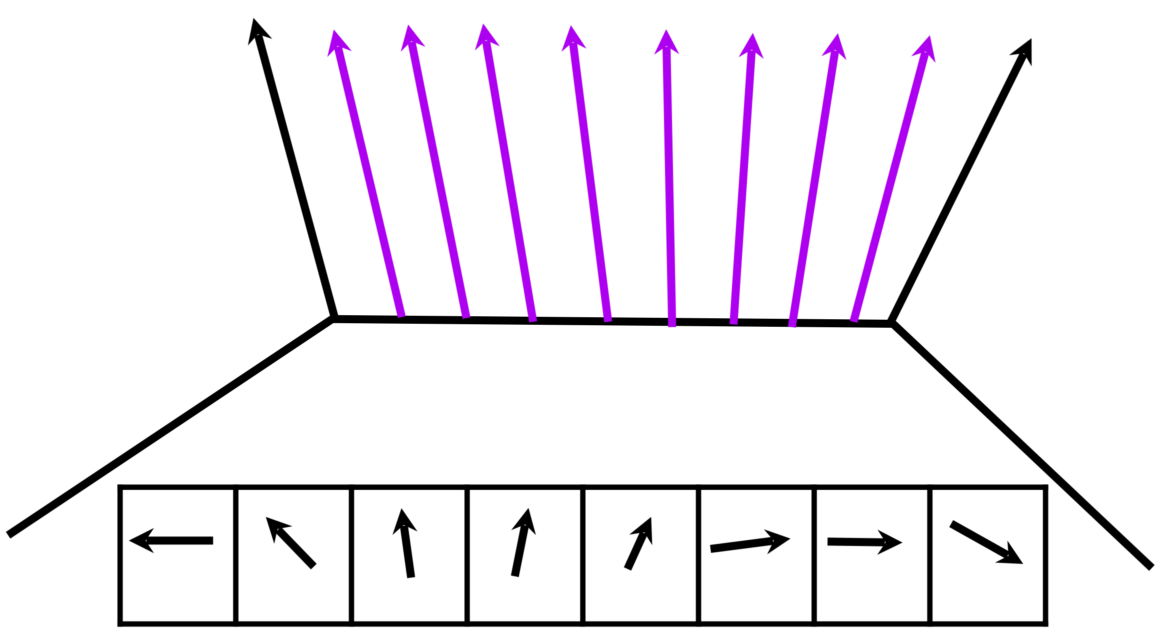
Normal maps

- Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.



Normal maps

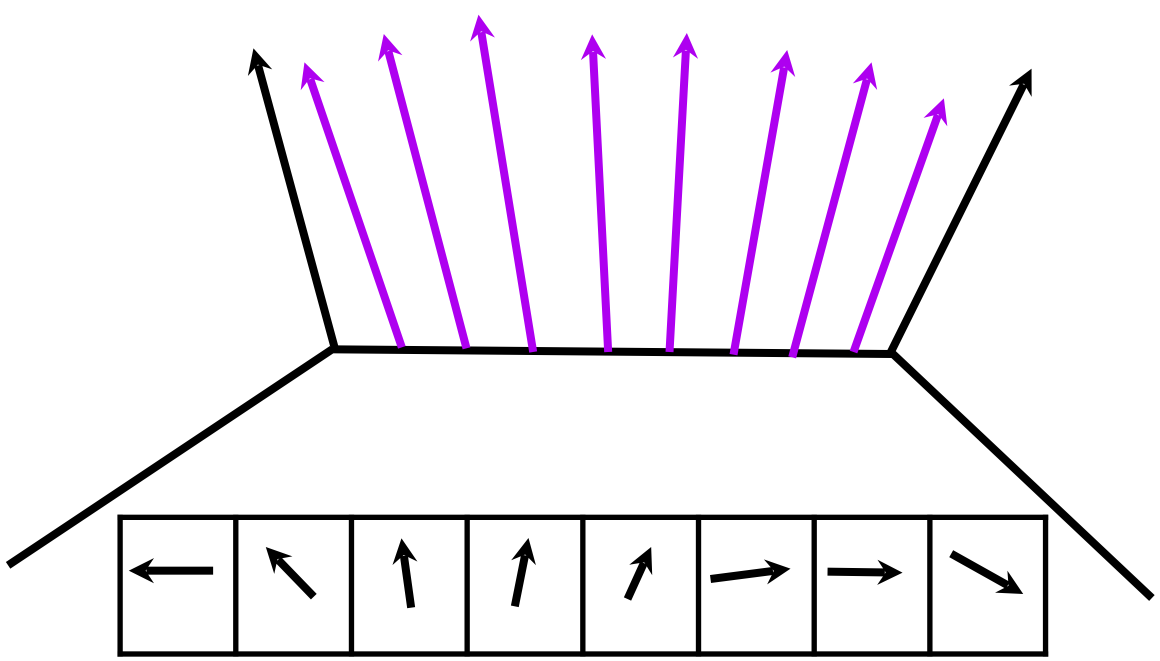
- Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.



Normal map

Normal maps

- Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.

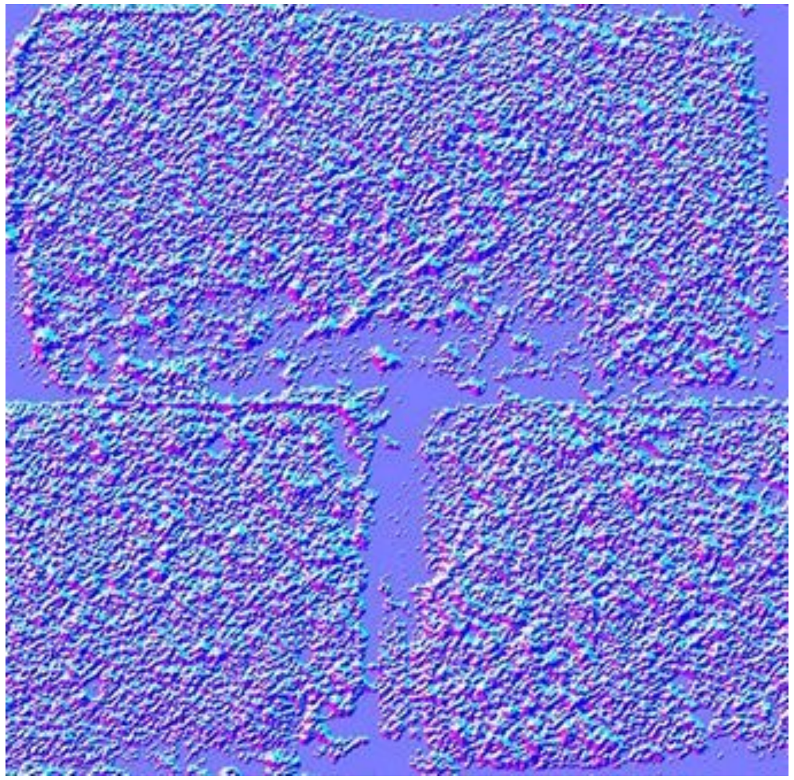


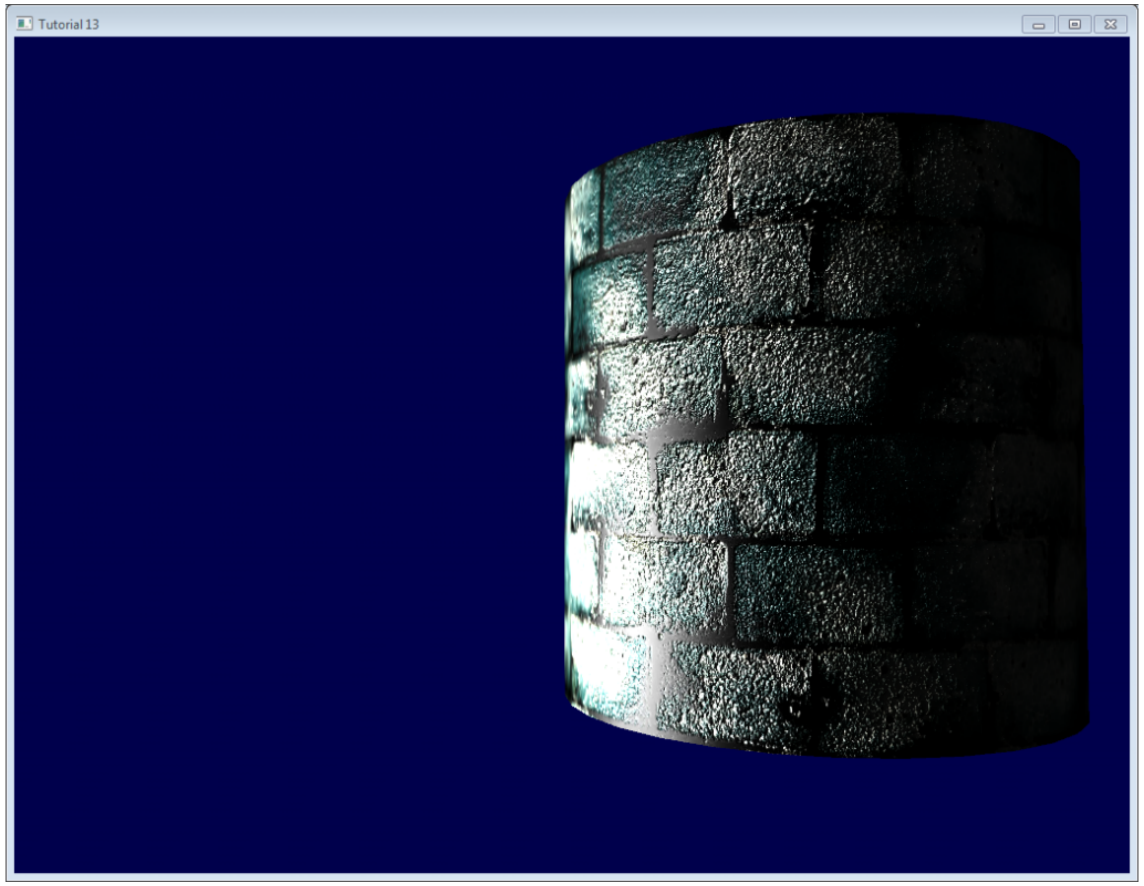
Perturbed normals

Normal maps

- Pros:
  - Provide the illusion of surface texture
- Cons:
  - Does not affect silhouette
  - Does not affect occlusion calculation

Normal Mapping





OpenGL

- http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/
- https://hacksoflife.blogspot.com/2009/11/per-pixel-tangent-space-normal-mapping.html
- http://www.terathon.com/code/tangent.html

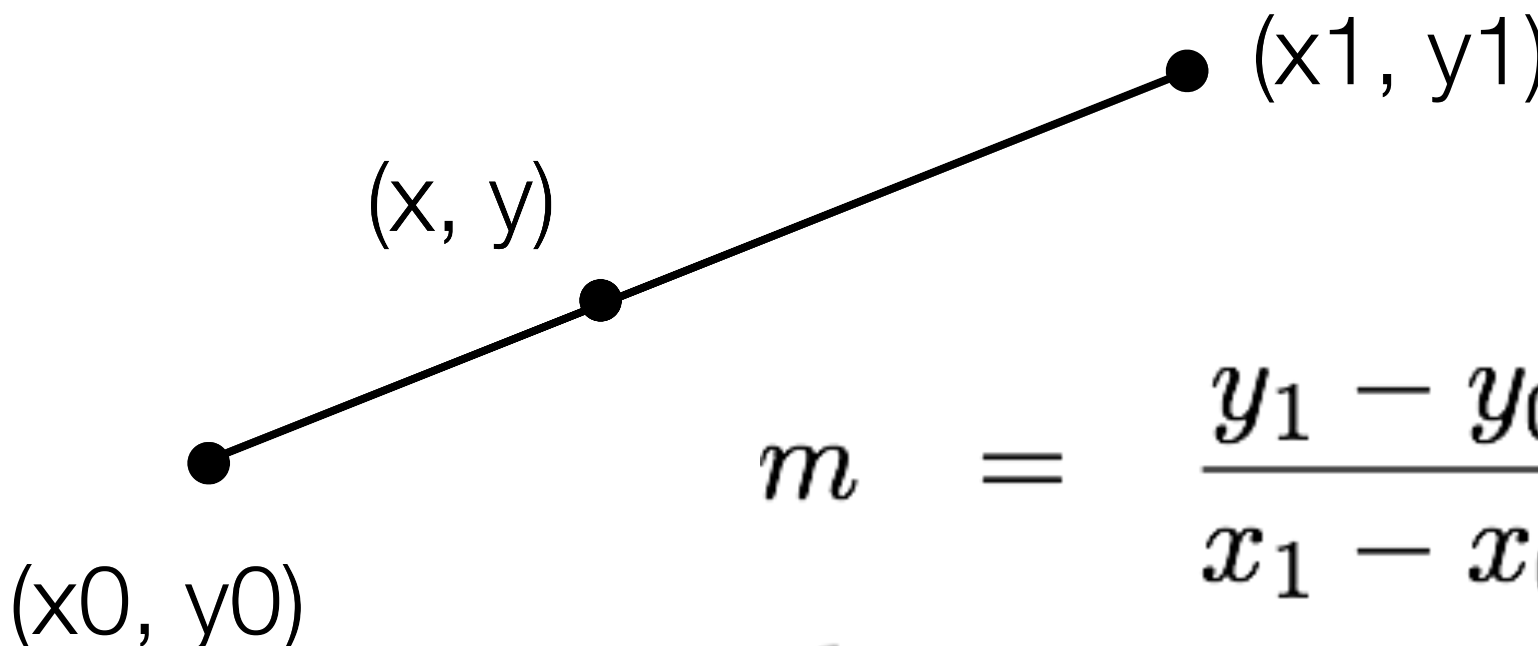
Rasterisation

- Rasterisation is the process of converting lines and polygons represented by their vertices into fragments.
- Fragments are like pixels but include color, depth, texture coordinate. They may also never make it to the screen due to hidden surface removal or culling.

Rasterisation

- This operation needs to be accurate and efficient.
- For this reason we prefer to use simple integer calculations.
- All our calculations are now in 2D screen space.

Drawing lines



$$
\begin{aligned}
m &= \frac{y_1 - y_0}{x_1 - x_0} \\
b &= y_0 - m x_0 \\
y &= m x + b
\end{aligned}
$$

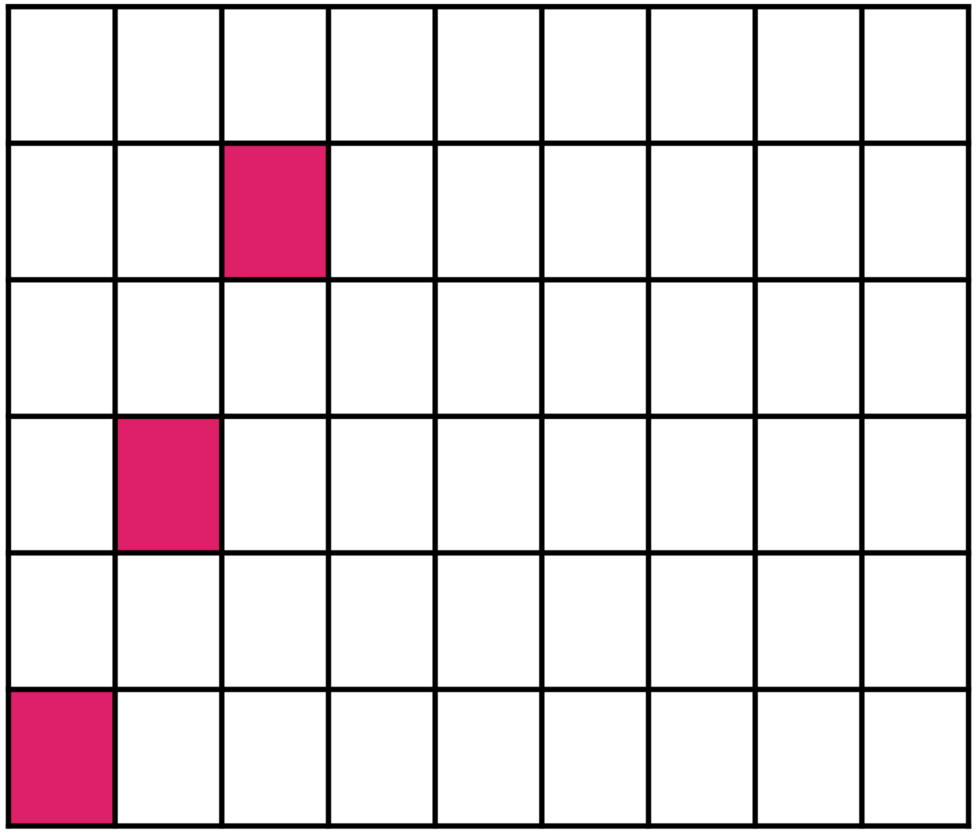
Drawing lines - **bad**

```
float m = (y1-y0)/(float)(x1-x0);
float b = y0 - m * x0;

for (int x = x0; x <= x1; x++) {
    int y = (int) Math.round(m * x + b);
    drawPixel(x, y);
}
```

Problems

- Floating point math is slow and creates rounding errors
  - Floating point multiplication, addition and round for each pixel
- Code does not consider:
  - Points are not connected if $m > 1$
  - Divide by zero if $x0 == x1$ (vertical lines)
    - Doesn't work if $x0 > x1$

Example: $y = 2x$



Incremental – still bad

```
// incremental algorithm
float m = (y1-y0)/(float)(x1-x0);
float y = y0;

for (int x = x0; x <= x1; x++) {
  y += m; //one less multiplication
  drawPixel(x, (int) Math.round(y));
}
```

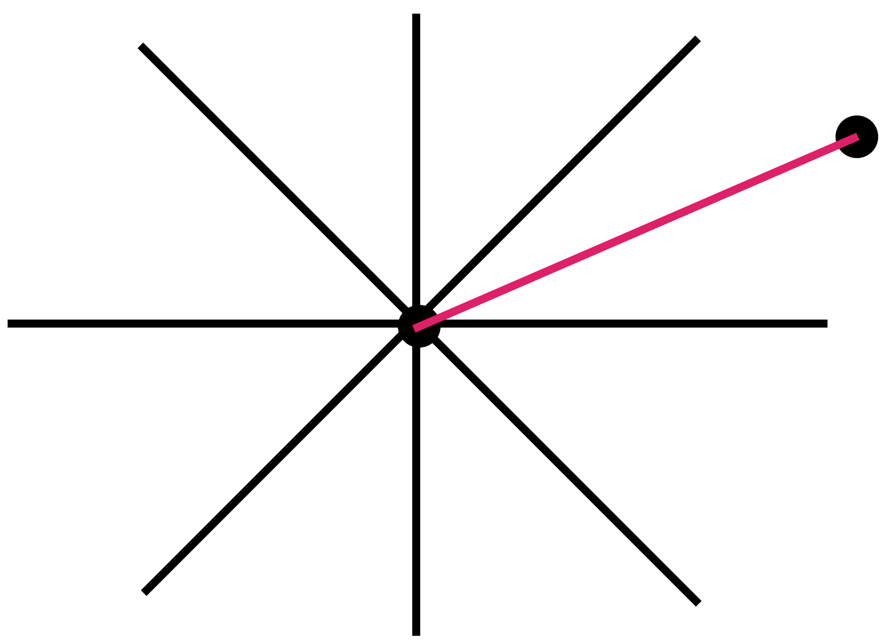
Bresenham's algorithm

- We want to draw lines using only integer calculations and avoid multiplications.
- Such an algorithm is suitable for fast implementation in hardware.
- The key idea is that calculations are done incrementally, based on the values for the previous pixel.

Bresenham's algorithm

- We shall assume to begin with that the line is in the first octant.

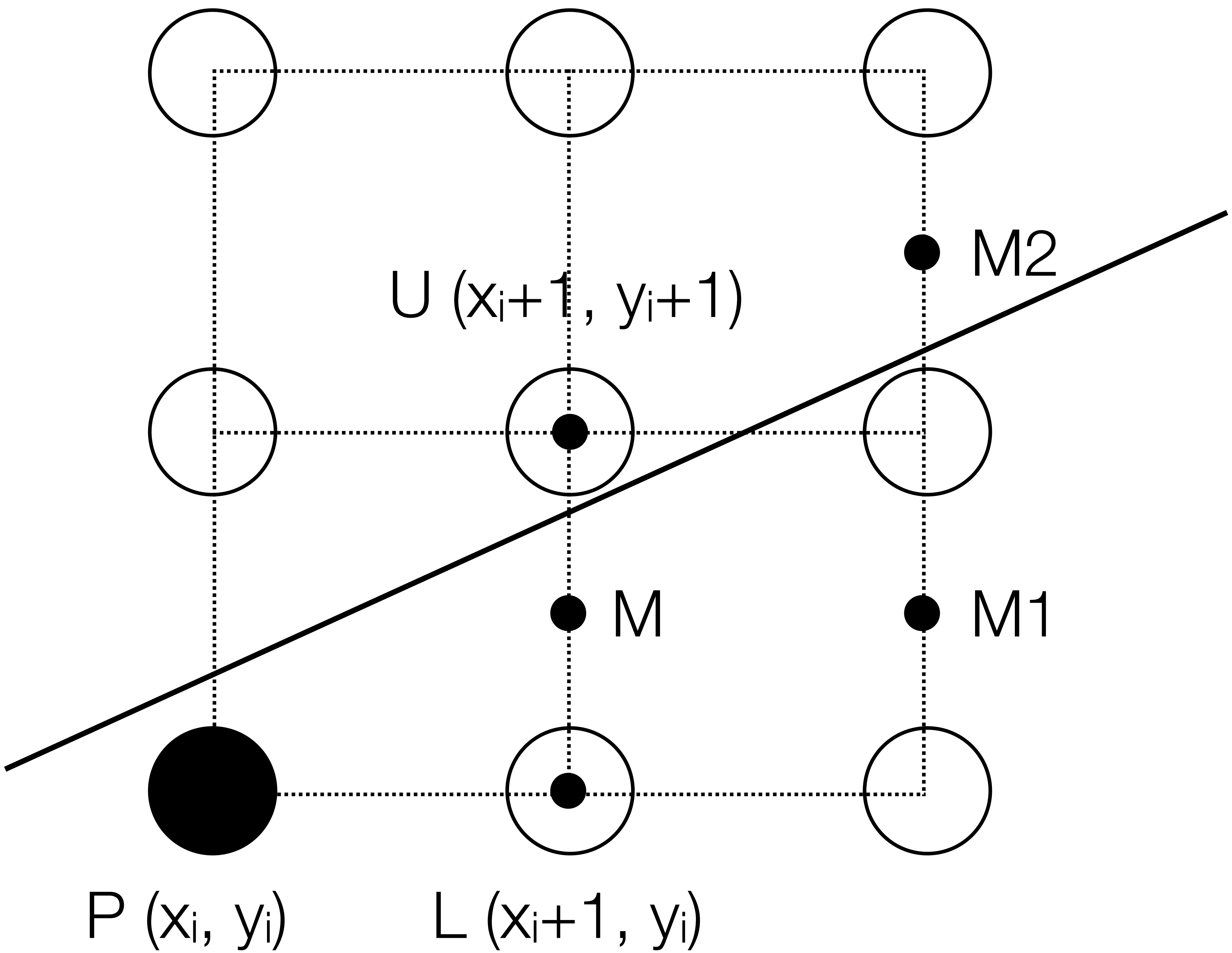
- I.e. x1 > x0, y1 > y0 and m <= 1



Bresenham’s Idea

- For each x we work out which pixel we set next
  - The next pixel with the same y value if the line passes below the midpoint between the two pixels
  - Or the next pixel with an increased y value if the line passes above the midpoint between the two pixels

Bresenham's algorithm



Pseudocode

```
int y = y0;
for (int x = x0; x <= x1; x++) {
    setPixel(x,y);
    M = (x + 1, y + 1/2)
    if (M is below the line)
        y++
}
```

Testing above/below

- We're on the line when:

$$m = \frac{y - y_0}{x - x_0}$$

$$y - y_0 = m(x - x_0)$$

$$0 = m(x - x_0) - (y - y_0)$$

Testing above/below

- We're below the line when:

$$0 < m(x - x_0) - (y - y_0)$$

$$0 < (h / w)(x - x_0) - (y - y_0)$$

$$0 < h(x - x_0) - w(y - y_0)$$

$$0 < 2h(x - x_0) - 2w(y - y_0)$$

Testing above/below

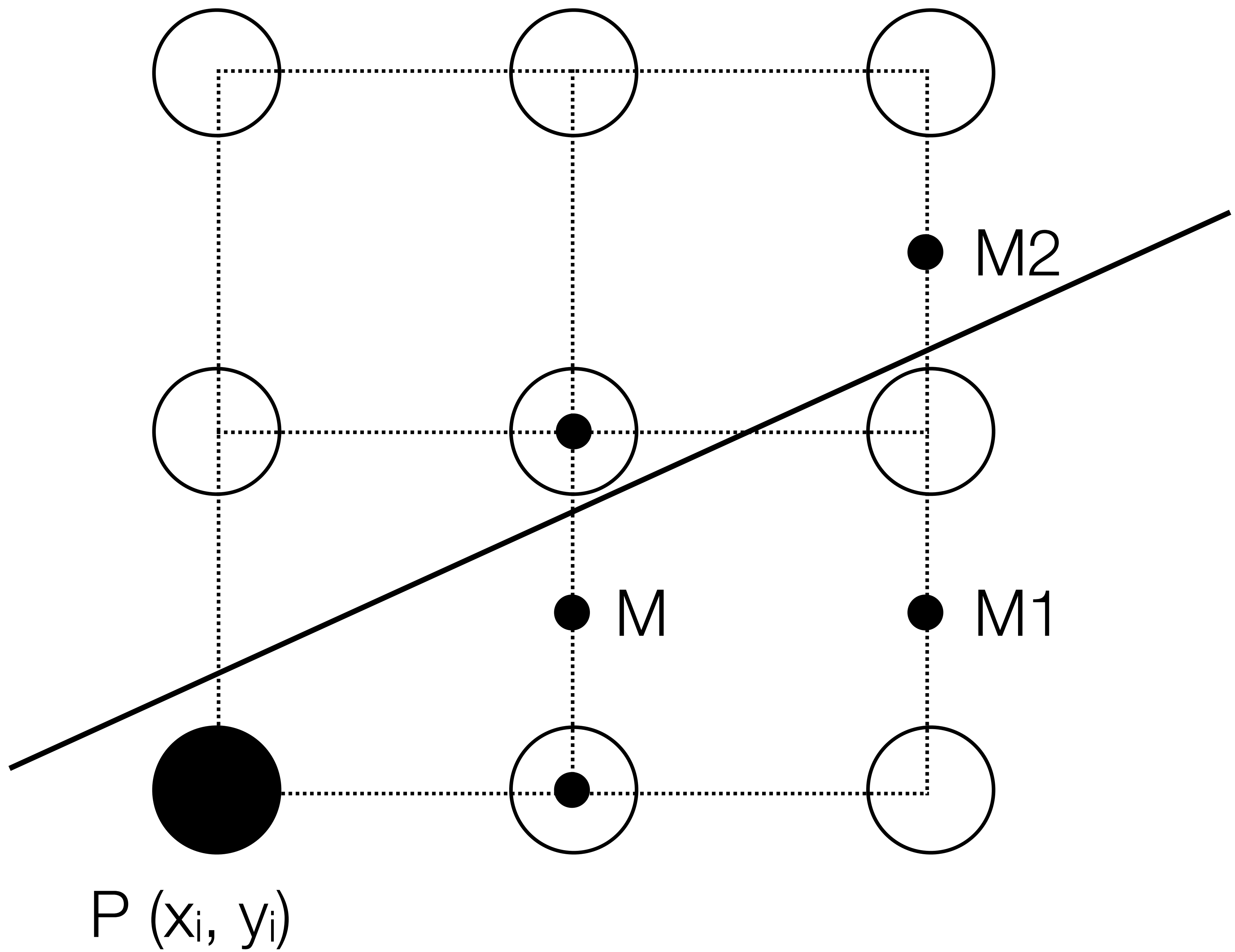
We call this value F

$$F(x,y) = 2h(x - x_0) - 2w(y - y_0)$$

$$F(x,y) < 0 \Rightarrow (x,y) \text{ is above line}$$

$$F(x,y) > 0 \Rightarrow (x,y) \text{ is below line}$$

Midpoints



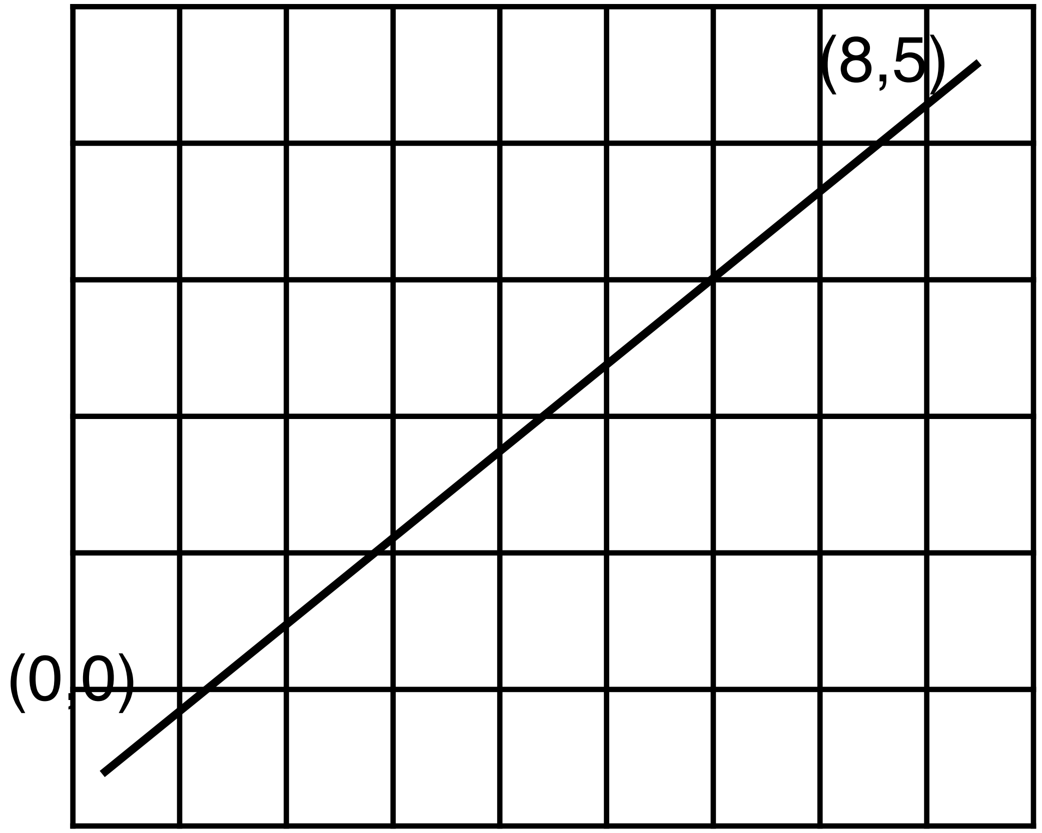
Incrementally

$$
\begin{aligned}
F(M) &= 2h(x_0+1-x_0) - 2w(y_0+\tfrac{1}{2}-y_0) \\
&= 2h - w \\
F(M_1) &= 2h(x_0+2-x_0) - 2w(y_0+\tfrac{1}{2}-y_0) \\
&= F(M) + 2h \\
F(M_2) &= 2h(x_0+2-x_0) - 2w(y_0+\tfrac{3}{2}-y_0) \\
&= F(M) + 2h - 2w
\end{aligned}
$$

Complete

```
int y = y0;
int w = x1 - x0; int h = y1 - y0;
int F = 2 * h - w;

for (int x = x0; x <= x1; x++) {
  drawPixel(x,y);
  if (F < 0) F += 2*h;
  else {
     F += 2*(h-w);  y++;
  }
}
```

Example



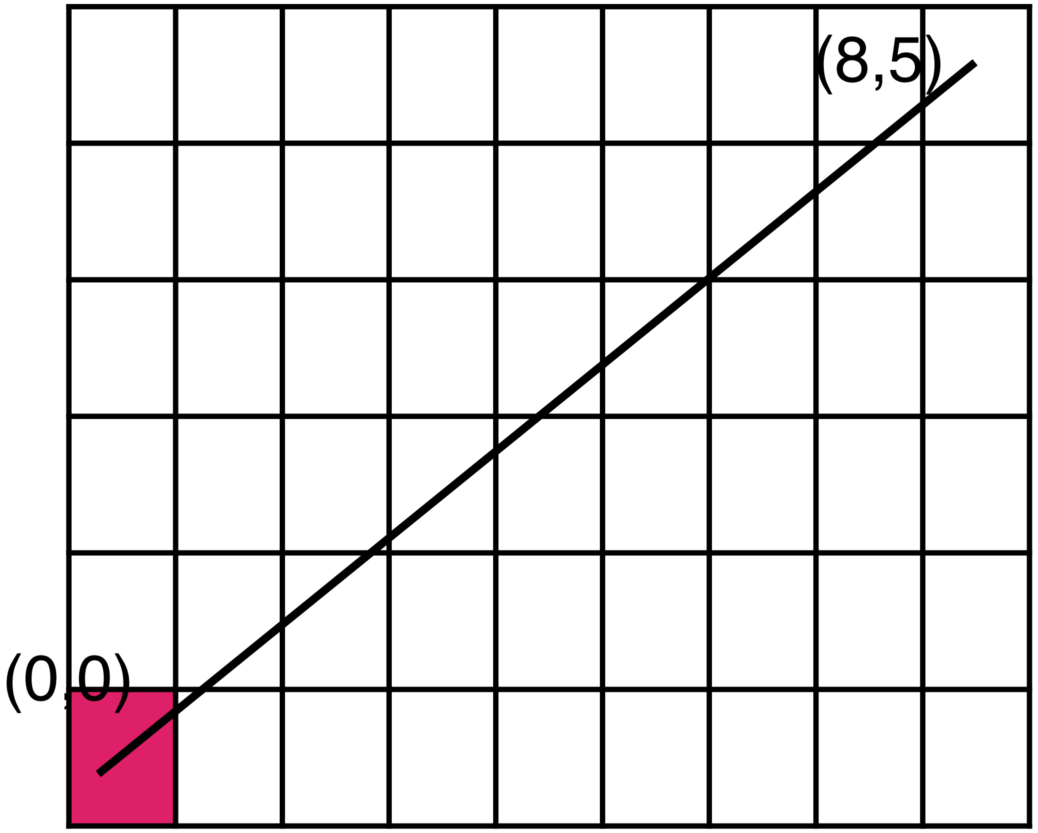
x	y	F
0	0	2

w = 8

h = 5

```
int F = 2 * h - w;
```

Example

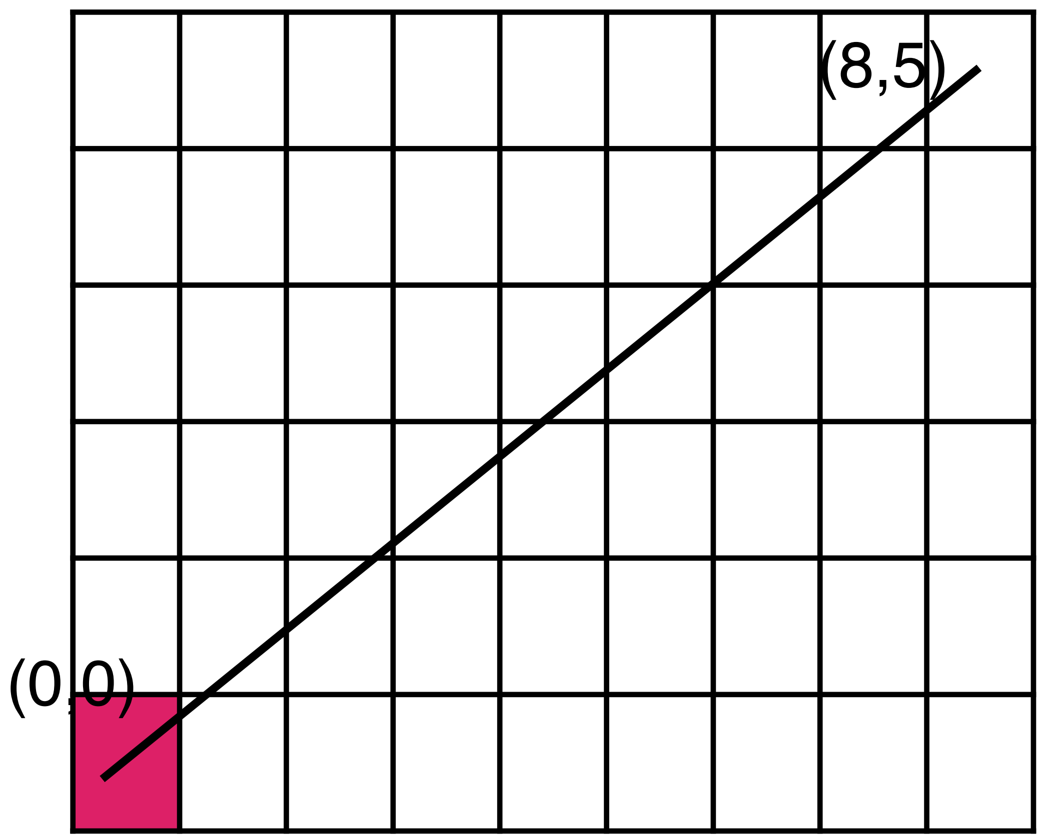


w = 8

h = 5

x	y	F
0	0	2

Example



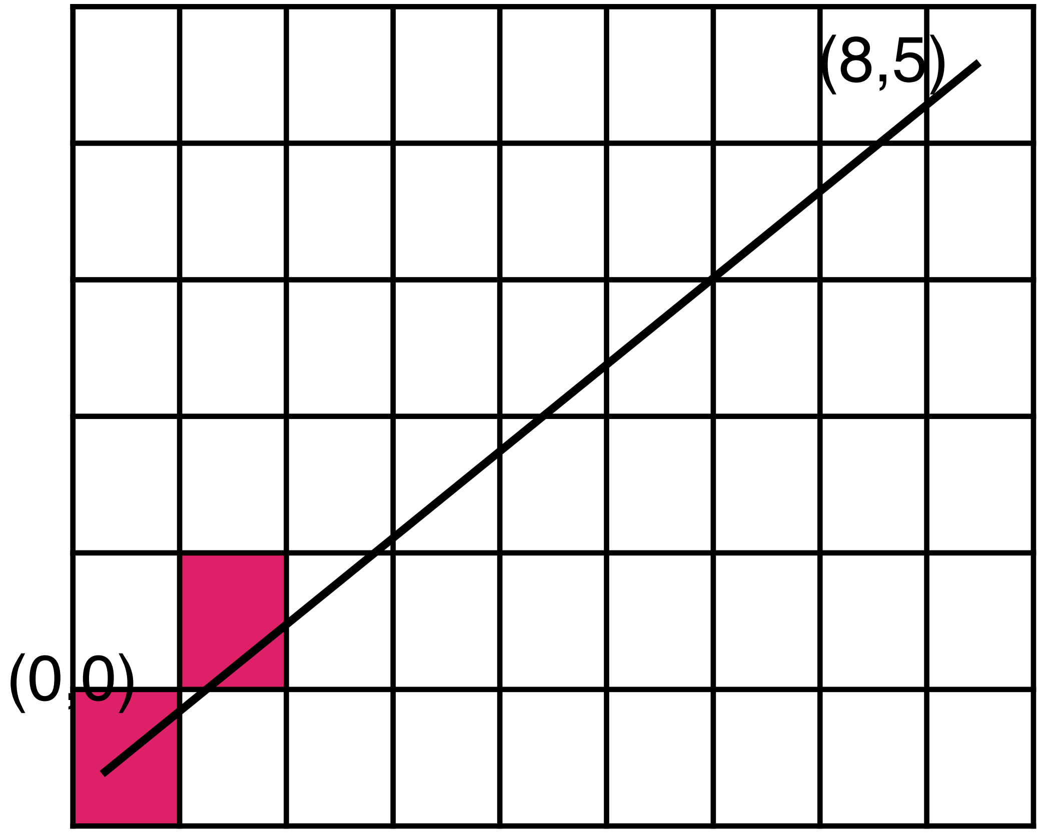
x	y	F
0	0	2
1	1	-4

w = 8

h = 5

`2 * (h - w) = -6`

Example



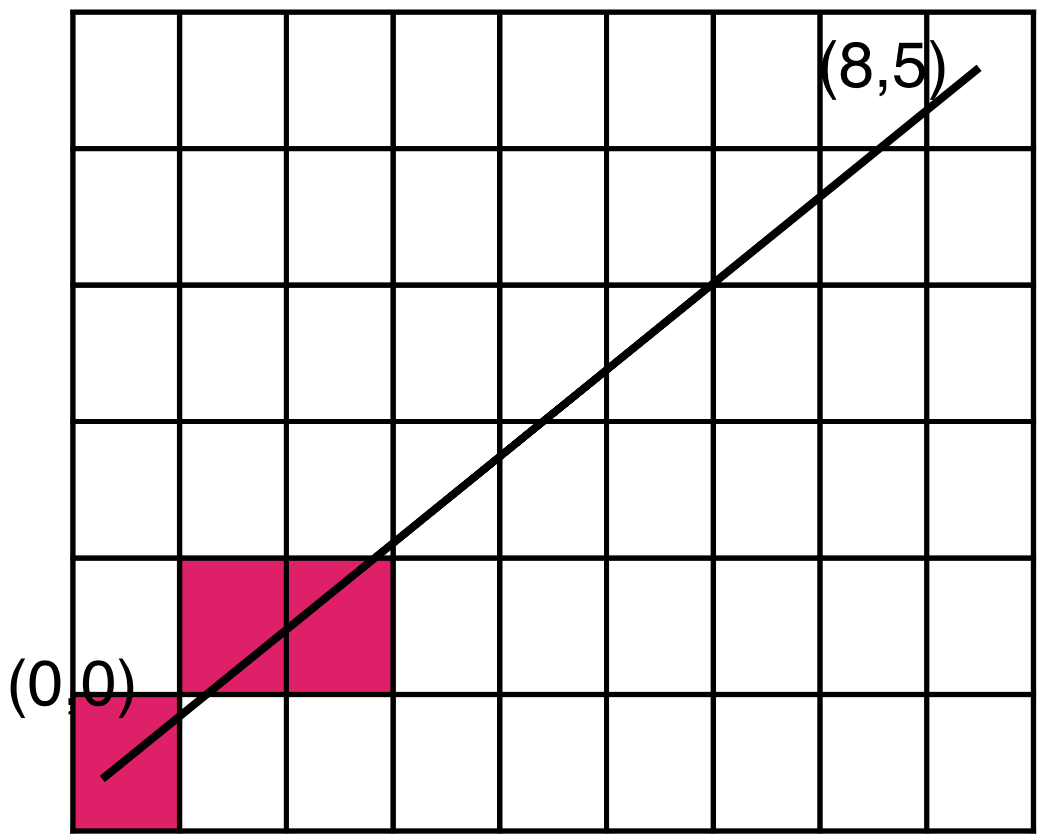
$$w = 8$$

$$h = 5$$

```
2 * (h - w) = -6
2 * h = 10
```

x	y	F
0	0	2
1	1	-4
2	1	6

Example



x	y	F
0	0	2
1	1	-4
2	1	6
3	2	0

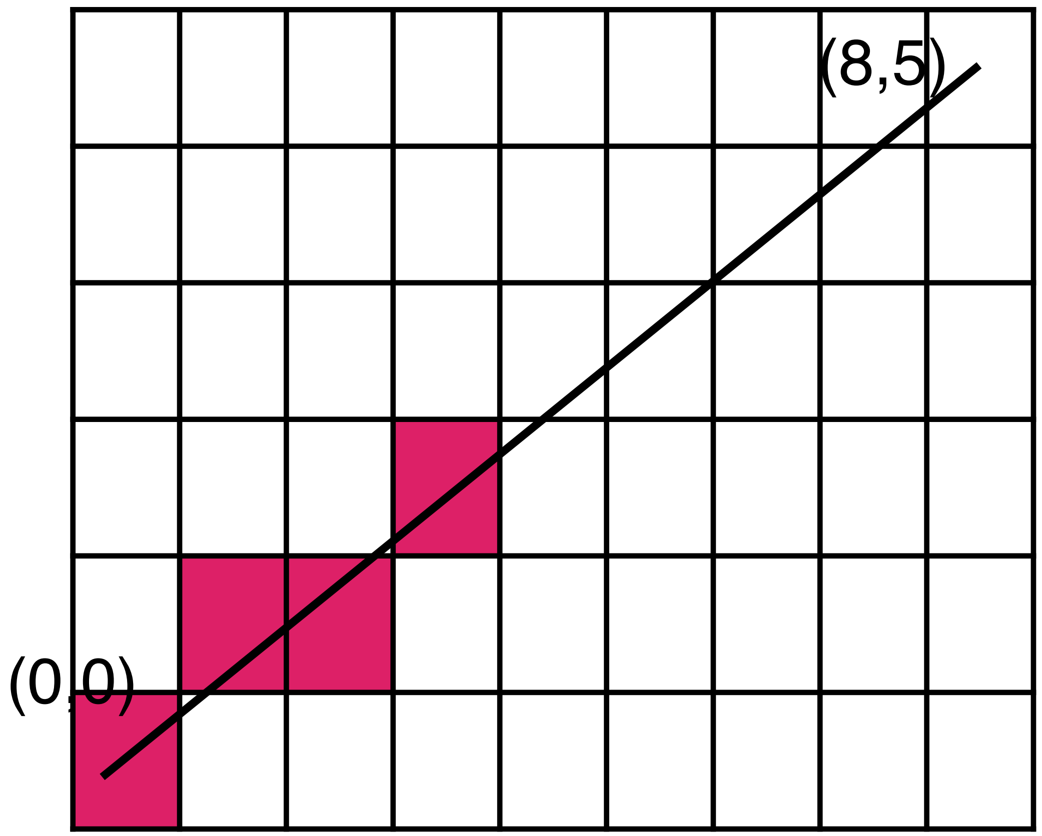
w = 8

h = 5

`2 * (h - w) = -6`

`2 * h = 10`

Example



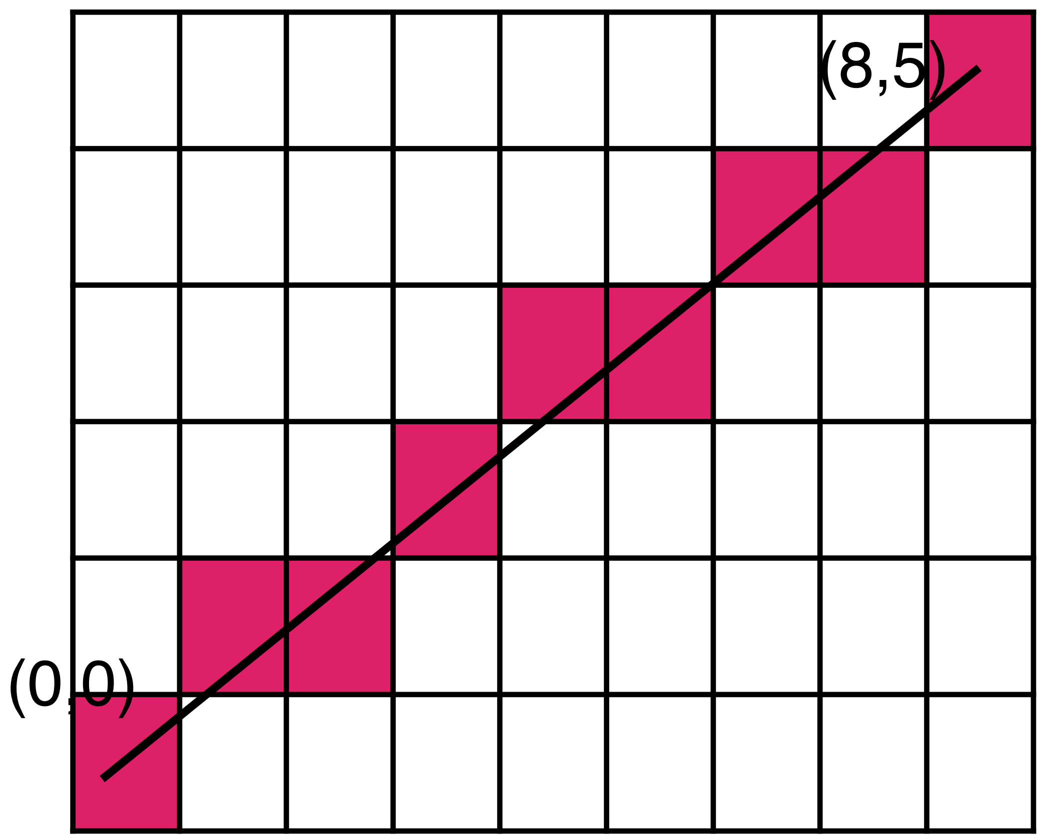
x	y	F
0	0	2
1	1	-4
2	1	6
3	2	0
4	3	-6

w = 8

h = 5

```
2 * (h - w) = -6
2 * h = 10
```

Example



w = 8

h = 5

`2 * (h - w) = -6`

`2 * h = 10`

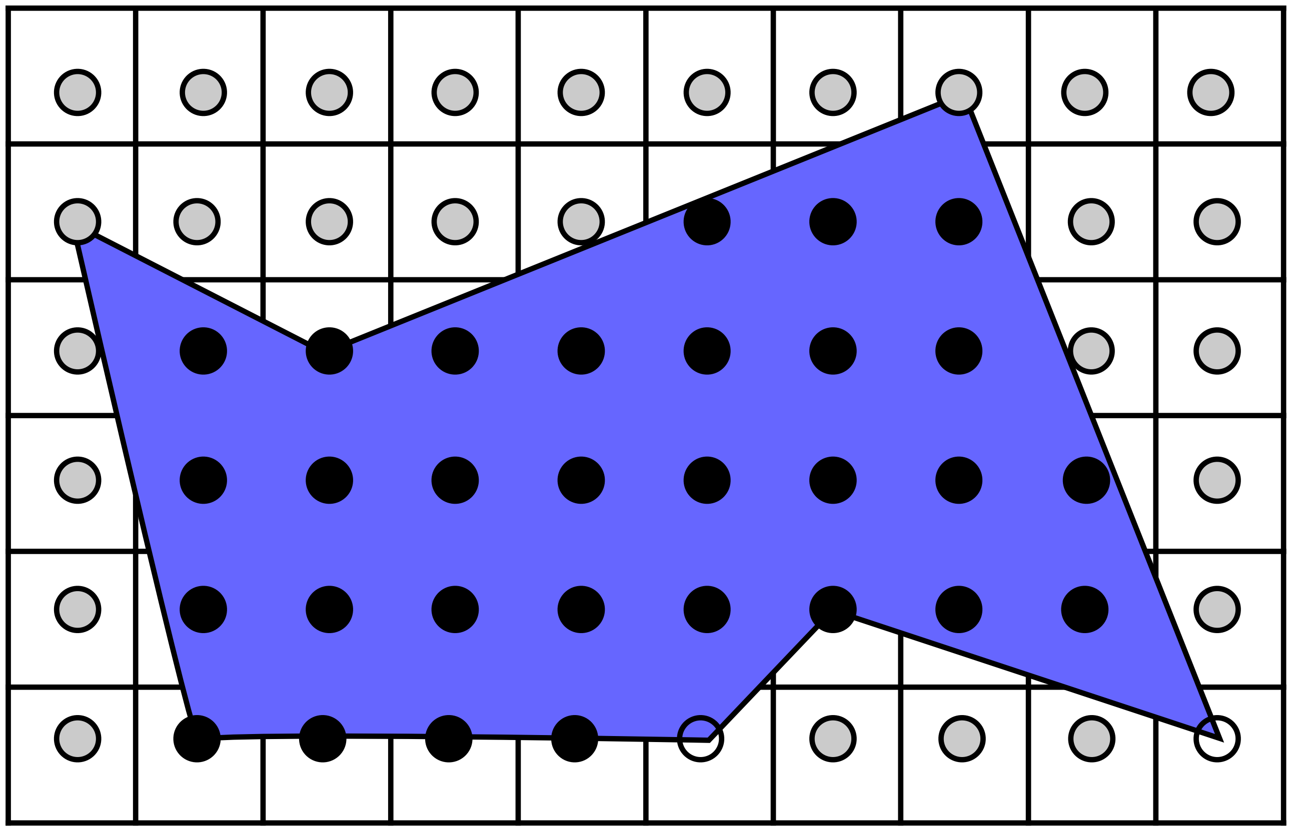
x	y	F
0	0	2
1	1	-4
2	1	6
3	2	0
4	3	-6
5	3	4
6	4	-2
7	4	8
8	5	2

Relaxing restrictions

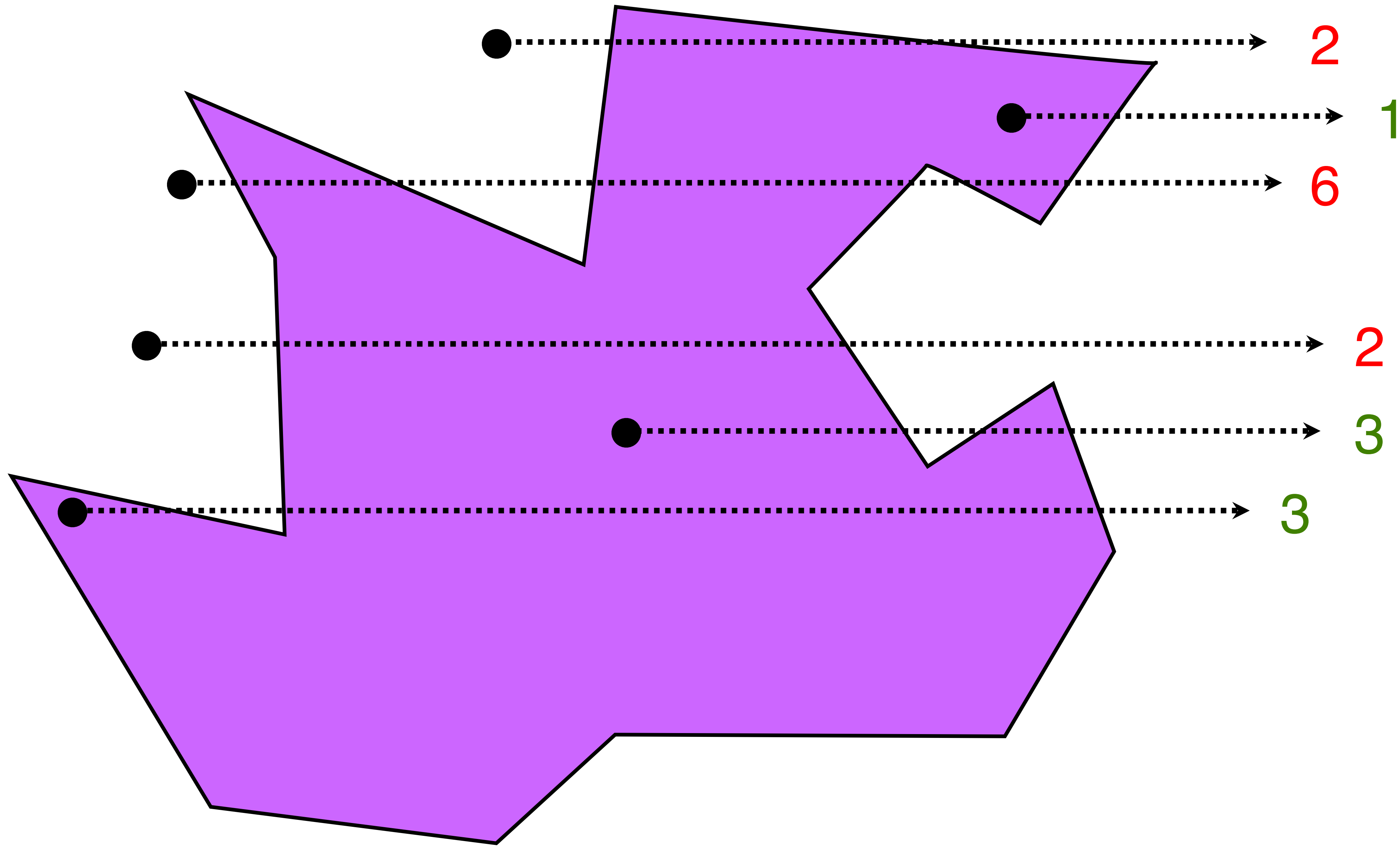
- Lines in the other quadrants can be drawn by symmetrical versions of the algorithm.
- We need to be careful that drawing from P to Q and from Q to P set the same pixels.
- Horizontal and vertical lines are common enough to warrant their own optimised code.

Polygon filling

- Determining which pixels are inside a polygon is a matter of applying the edge-crossing test (from week 3) for each possible pixel.



Point in polygon

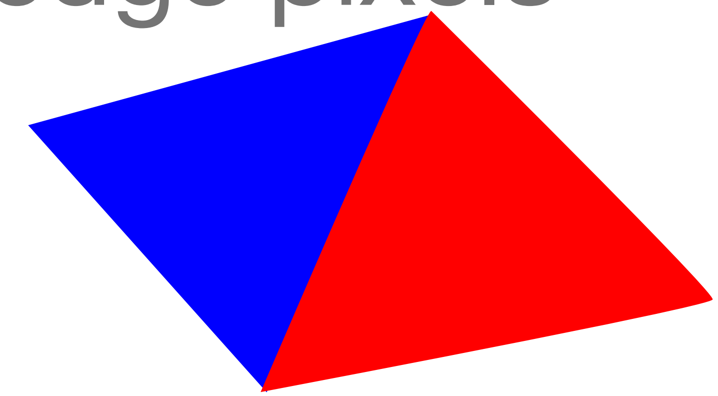


Problems

- Shared edges
  - If the pixel is on the edge of a polygon, should we draw it?
  - What if two polygons are adjacent (e.g. if they form part of a mesh)?
- Performance
  - Do we have to do the edge crossing test for every pixel?

Shared edges

- Pixels on shared edges between polygons need to be drawn consistently regardless of the order the polygons are drawn, with no gaps.
- We adopt a rule:
  - The edge pixels belong to the rightmost and/or upper polygon; i.e. do not draw rightmost or uppermost edge pixels



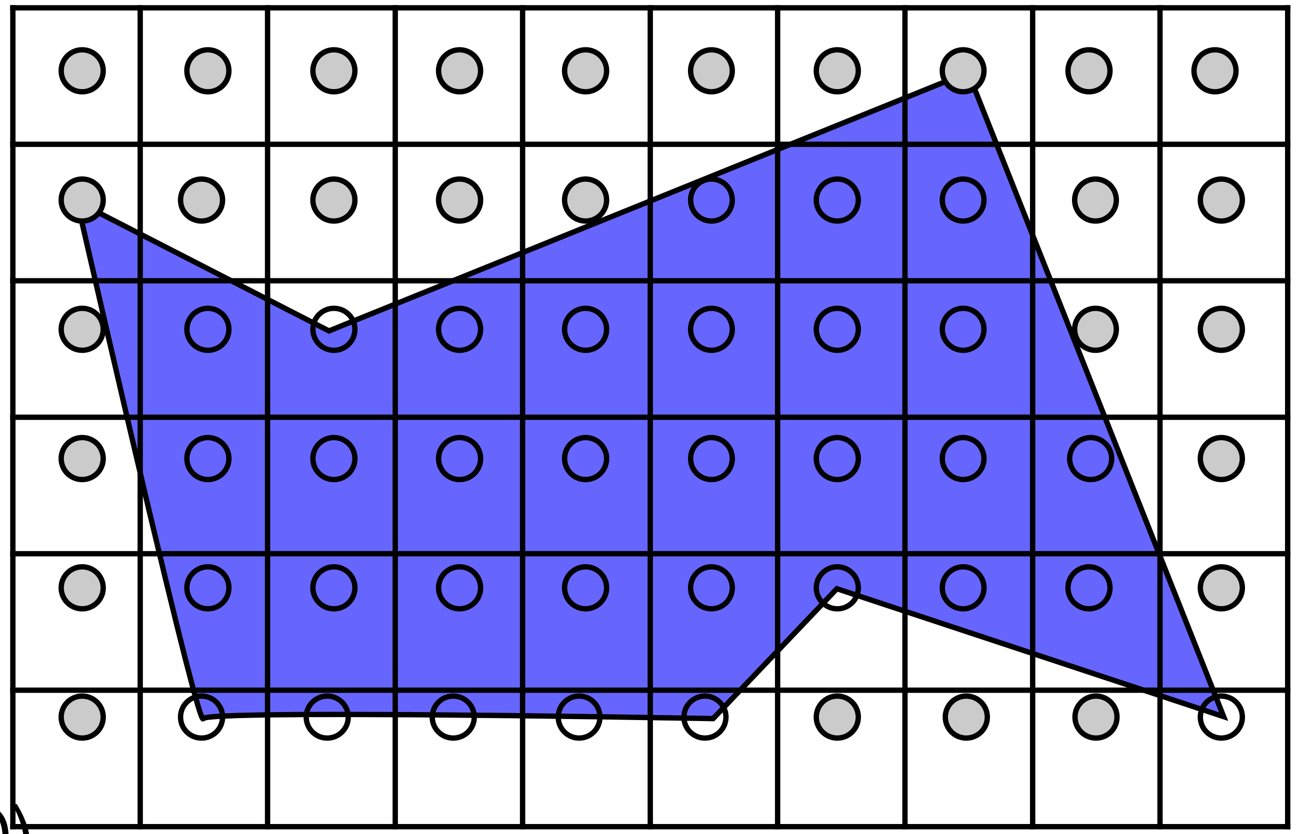
Scanline algorithm

- Testing every pixel is very inefficient.
- We only need to check where the result changes value, i.e. when we cross an edge.
- We proceed row by row:
  - Calculate intersections incrementally.
  - Sort by x value.
  - Fill runs of pixels between intersections.

Edge table

- The edge table is a lookup table indexed on the y-value of the lower vertex of the edge.
- Horizontal edges are not added
- We store the the x-value of the lower vertex, the increment (inverse gradient) of the edge and the y-value of the upper vertex.

Example



Edge table

y in	x	inc	y out
0	1	-0.25	4
0	5	1	1
0	9	-3	1
0	9	-0.4	5
3	2	-2	4
3	2	2.5	5

Active Edge List

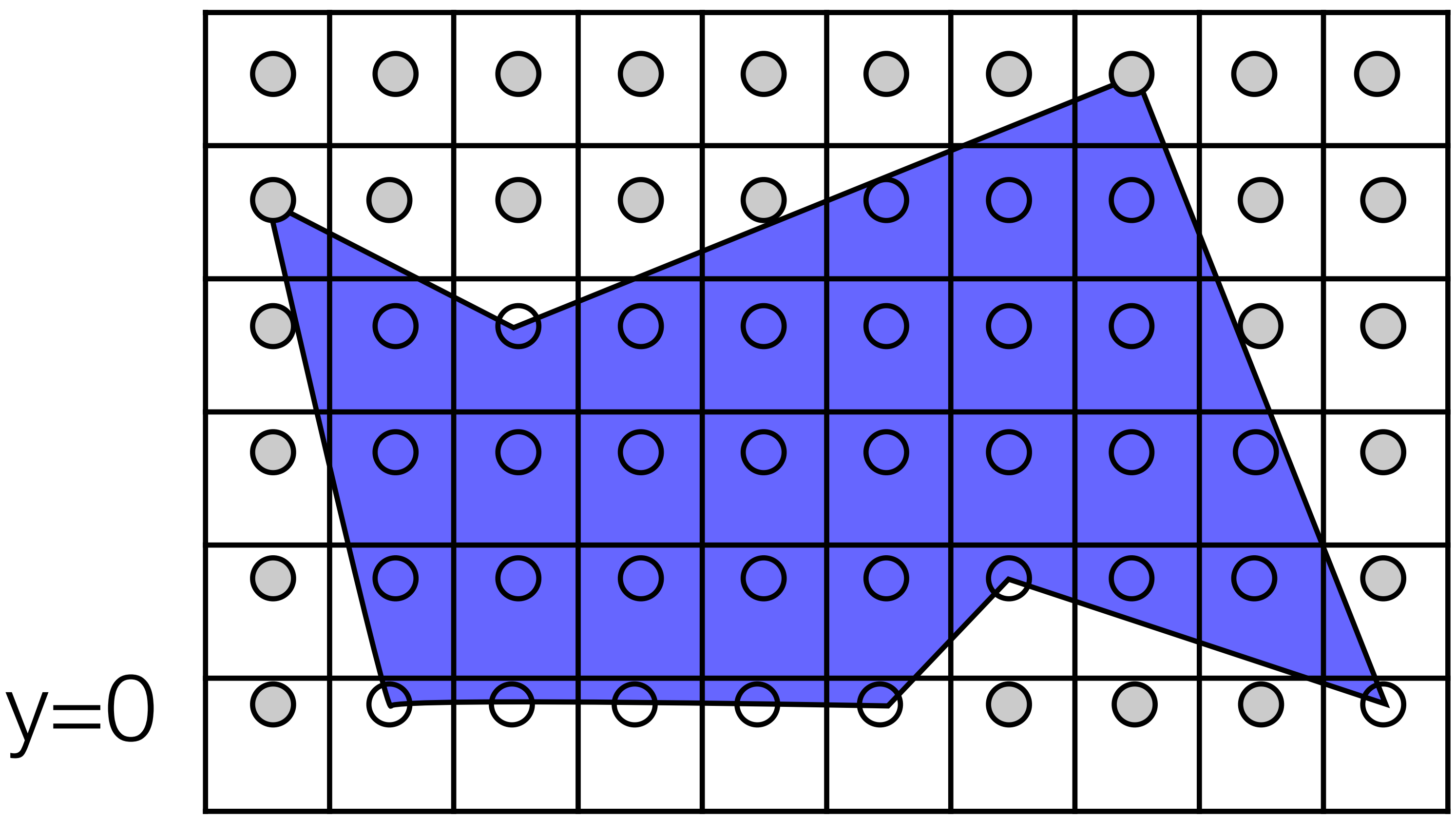
- We keep a list of active edges that overlap the current scanline.
- Edges are added to the list as we pass the bottom vertex.
- Edges are removed from the list as we pass the top vertex.
- The edge intersection is updated incrementally.

Edges

- For each edge in the AEL we store:
  - The x value of its crossing with the current row (initially the bottom x value)
  - The increment (inverse gradient)
  - The y value of the top vertex.

```
//For every scanline
for (y = minY; y <= maxY; y++){

   remove all edges that end at y

   for (Edge e : active) {
      e.x = e.x + e.inc;
   }

   add all edges that start at y – keep list
sorted by x

  for (int i=0; i < active.size; i+=2){
    fillPixels(active[i].x, active[i+1].x,y);
  }
}
```

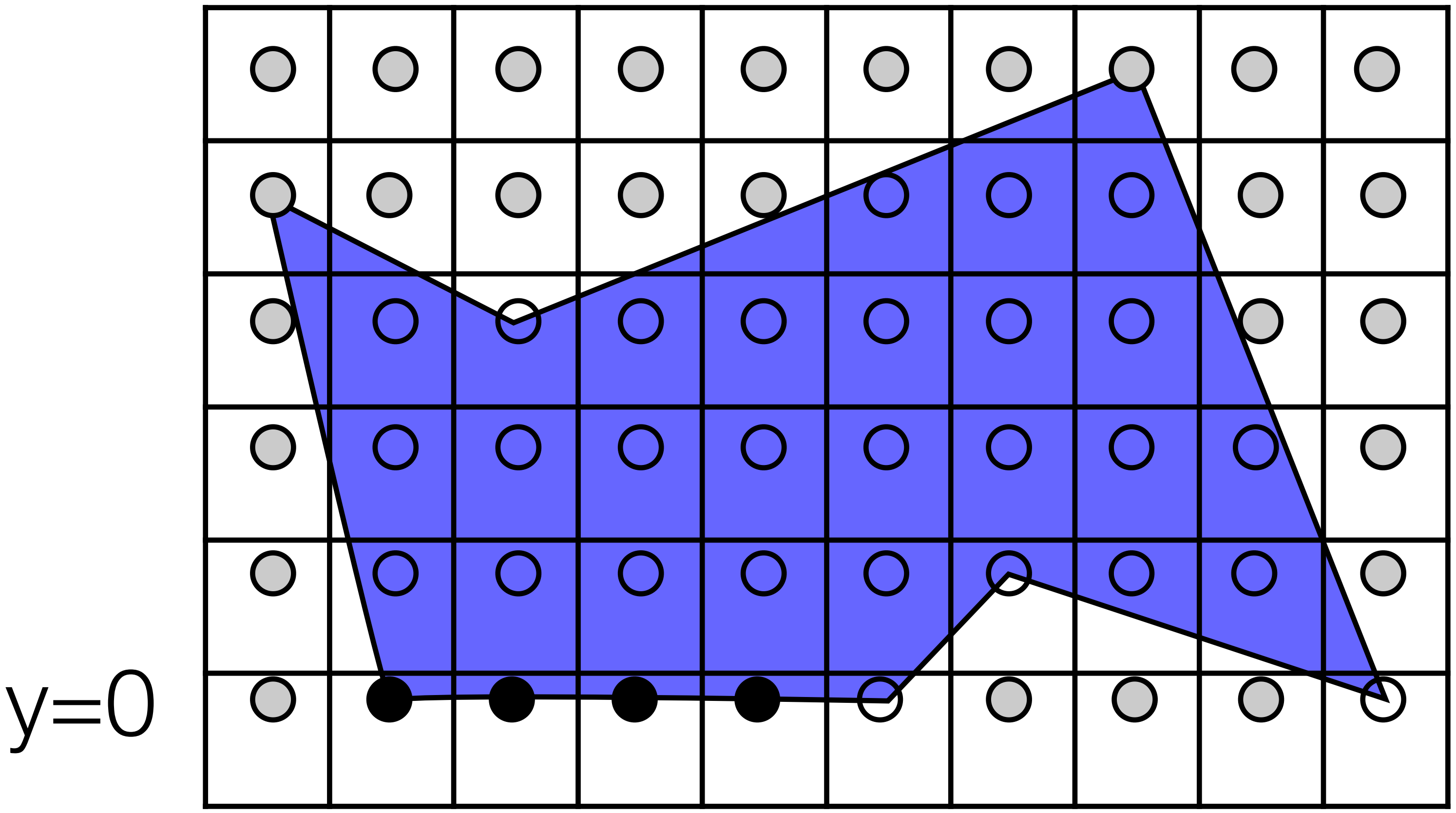
Example



Active edge list

x	inc	y out
1	-0.25	4
5	1	1
9	-3	1
9	-0.4	5

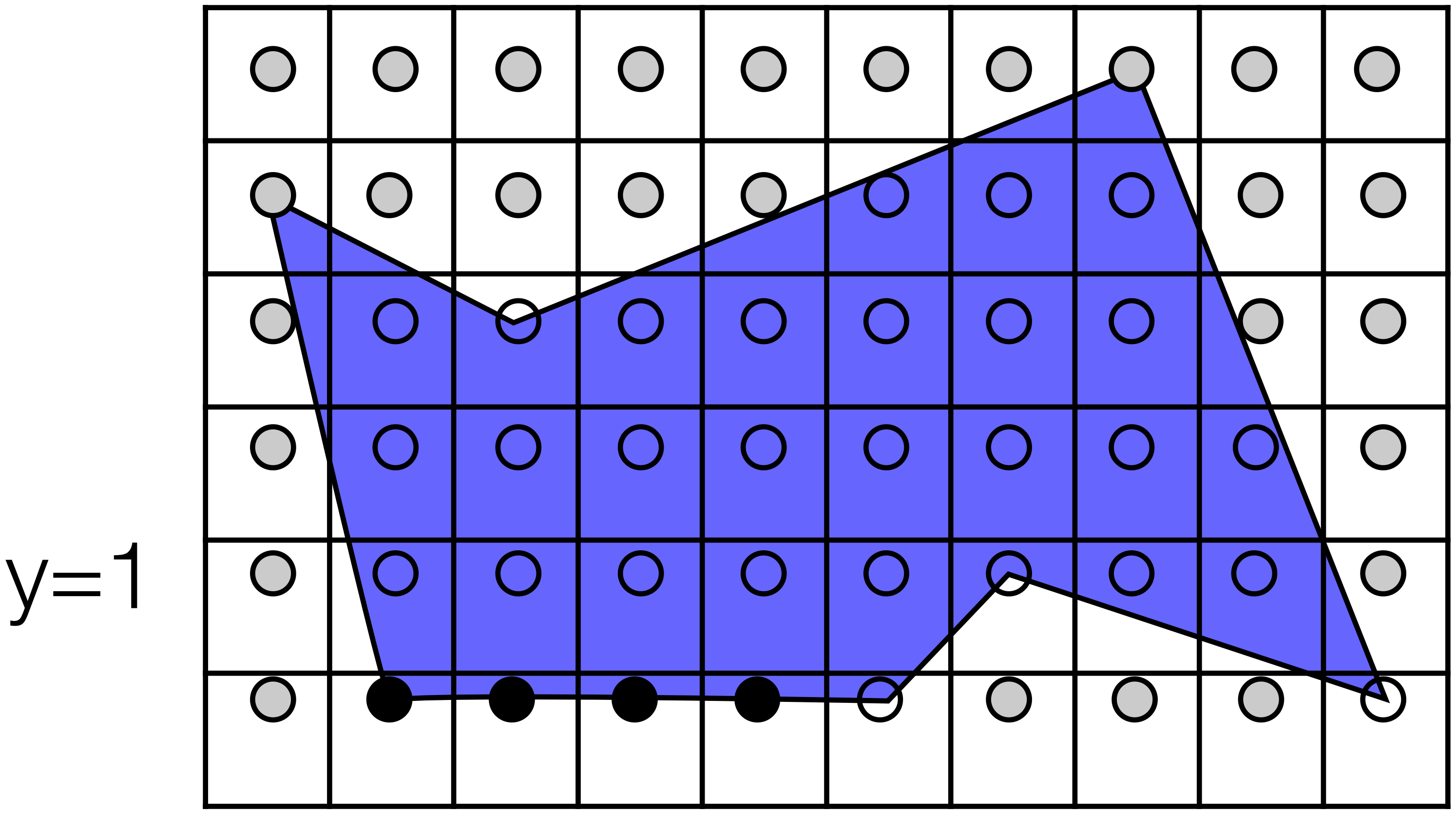
Example



Active edge list

x	inc	y out
1	-0.25	4
5	1	1
9	-3	1
9	-0.4	5

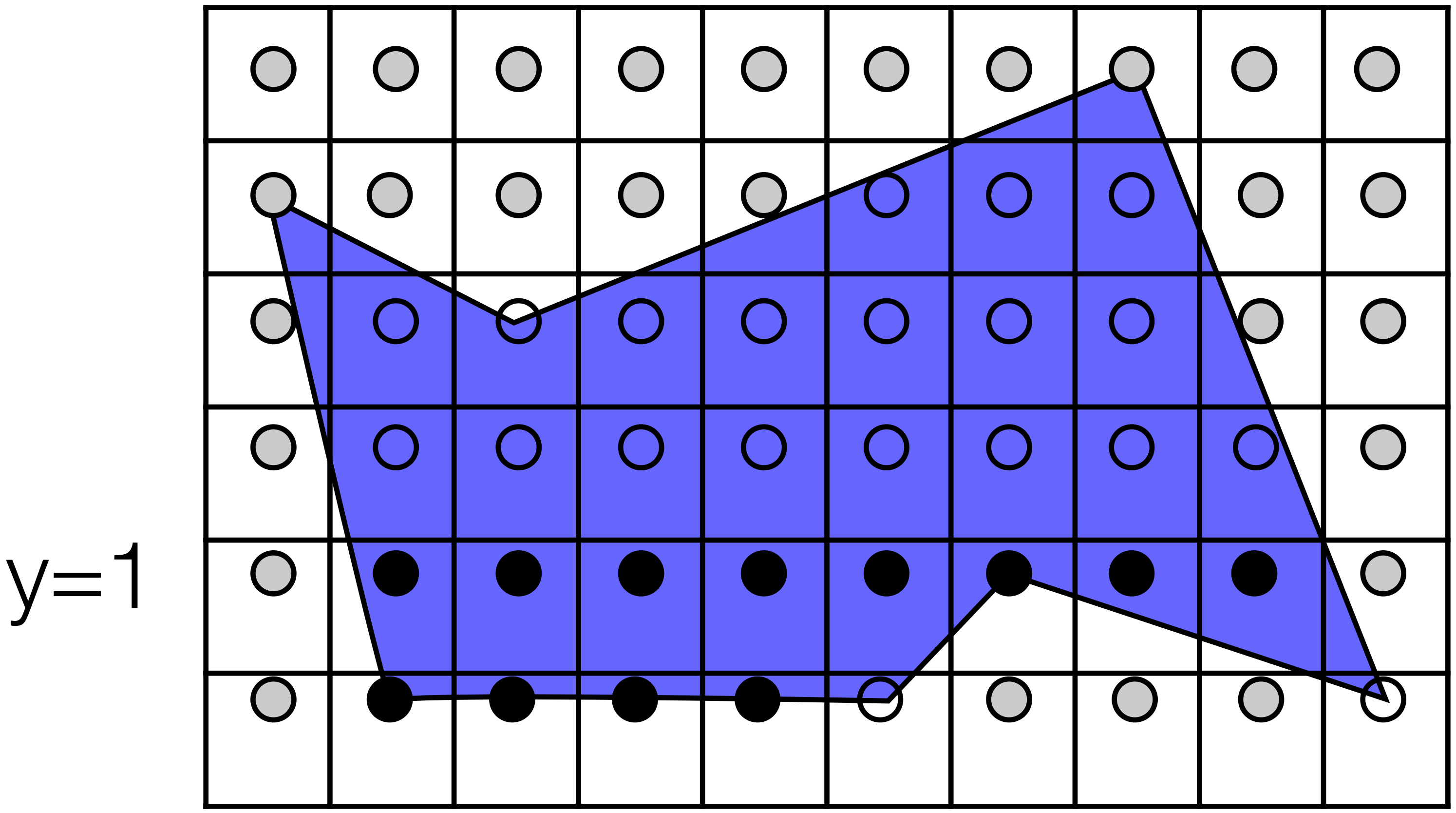
Example



Active edge list

x	inc	y out
0.75	-0.25	4
8.6	-0.4	5

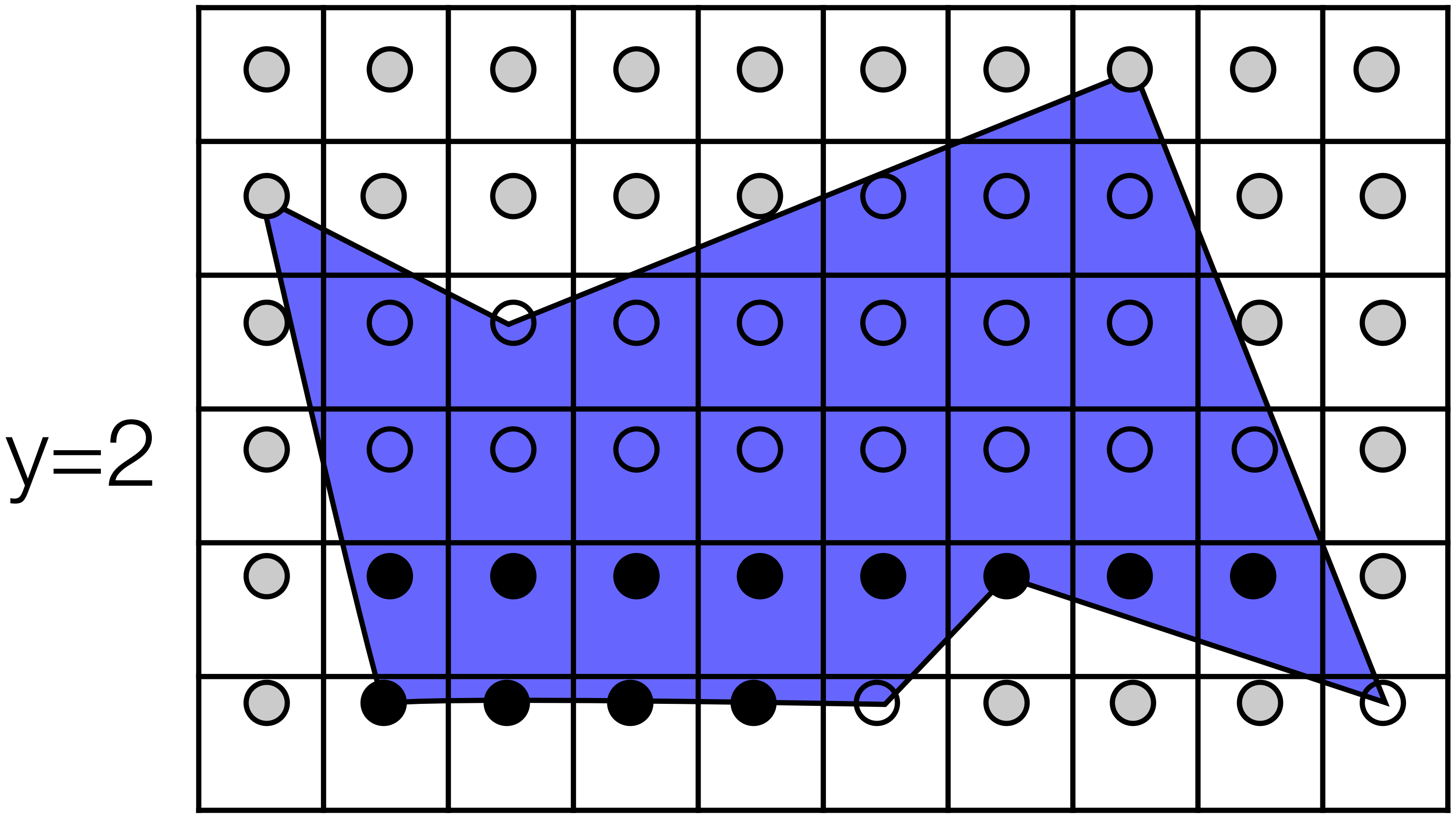
Example



Active edge list

x	inc	y out
0.75	-0.25	4
8.6	-0.4	5

Example

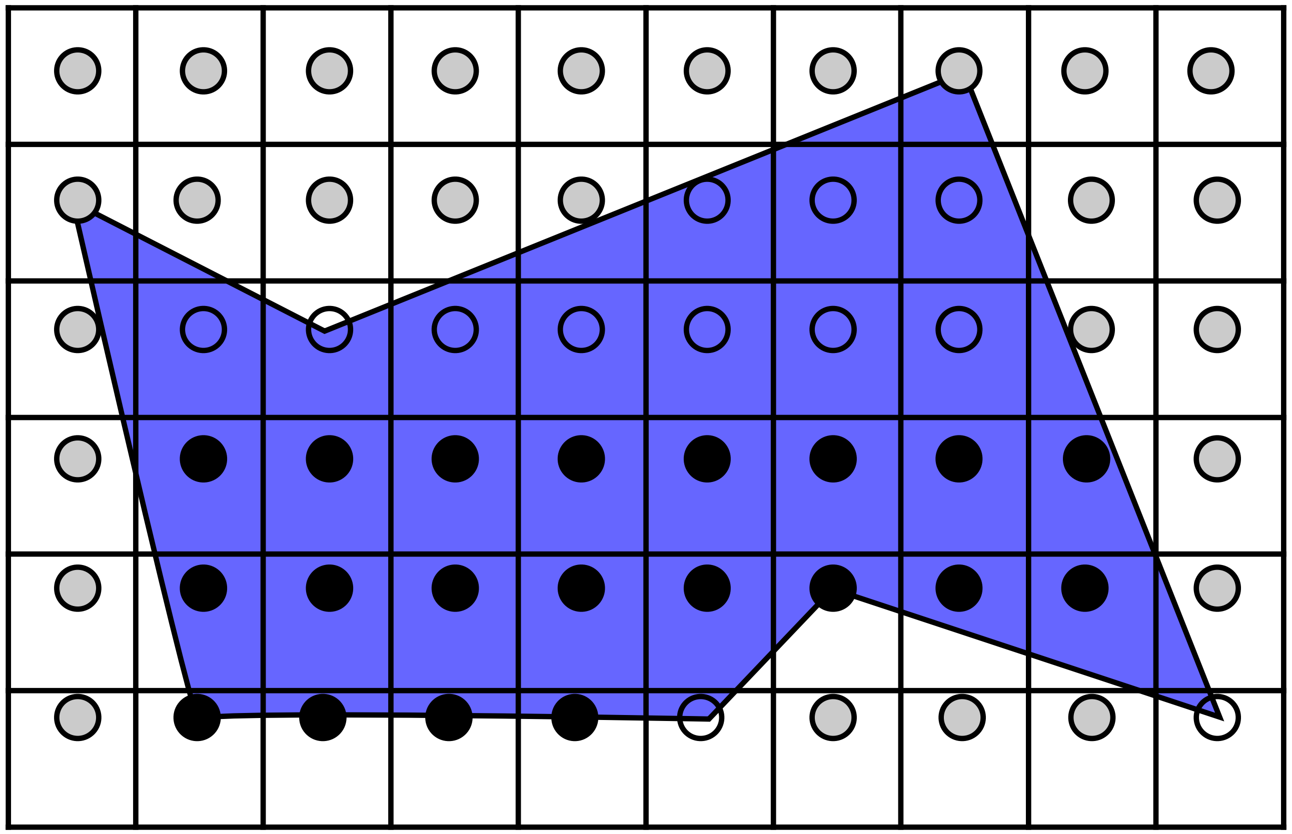


Active edge list

x	inc	y out
0.5	-0.25	4
8.2	-0.4	5

Example

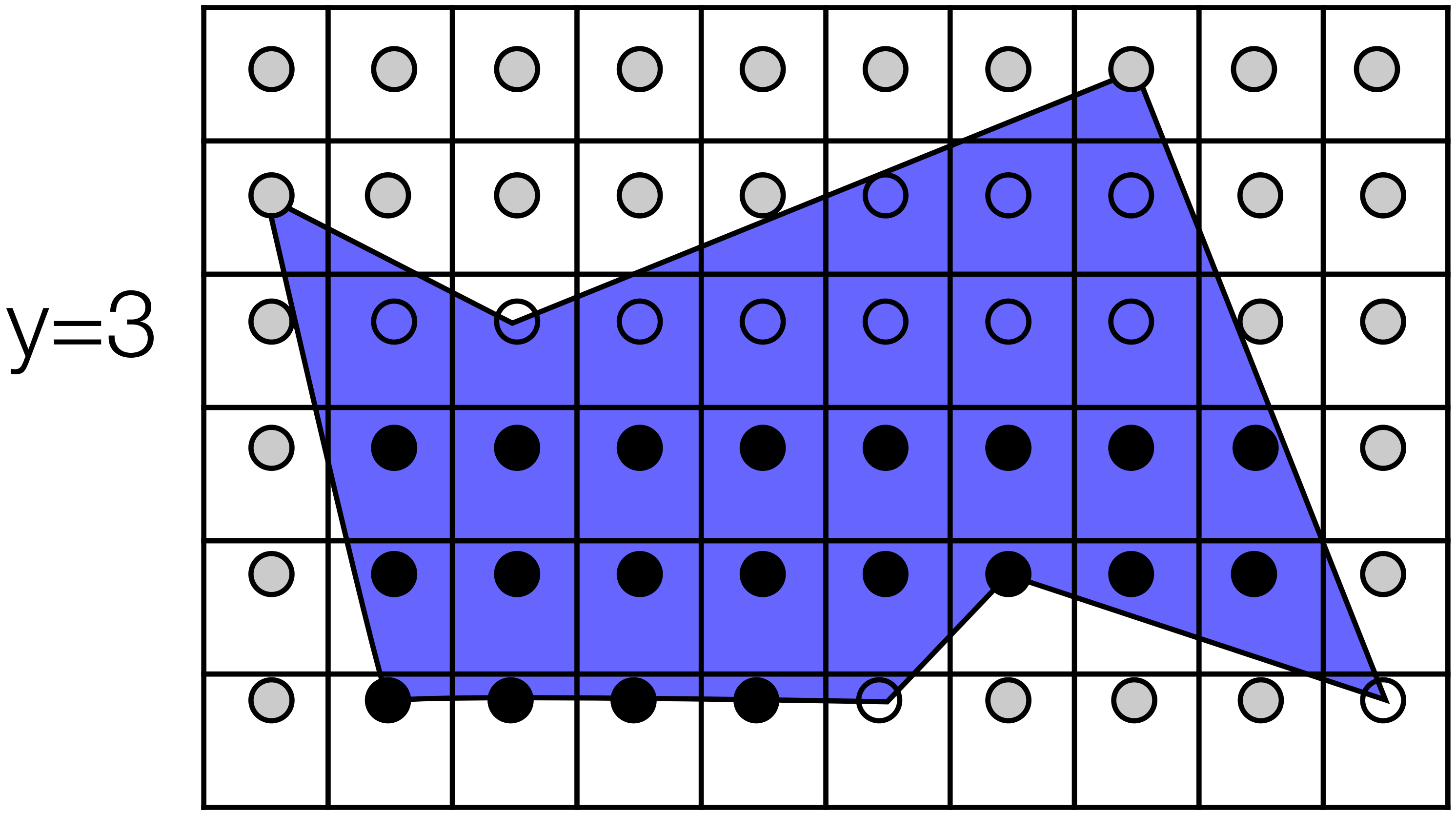
y=2



Active edge list

x	inc	y out
0.5	-0.25	4
8.2	-0.4	5

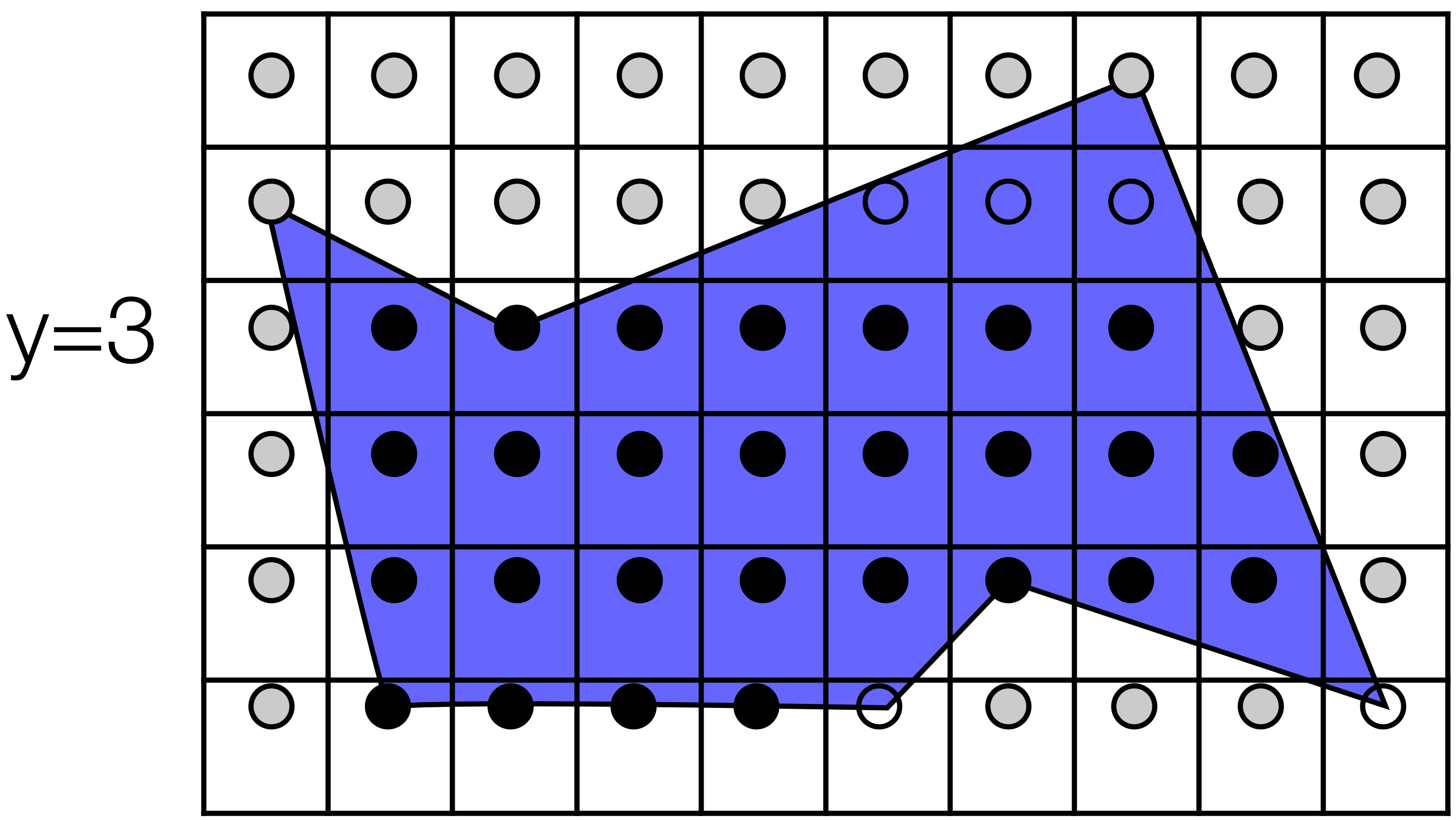
Example



Active edge list

x	inc	y out
0.25	-0.25	4
2	-2	4
2	2.5	5
7.8	-0.4	5

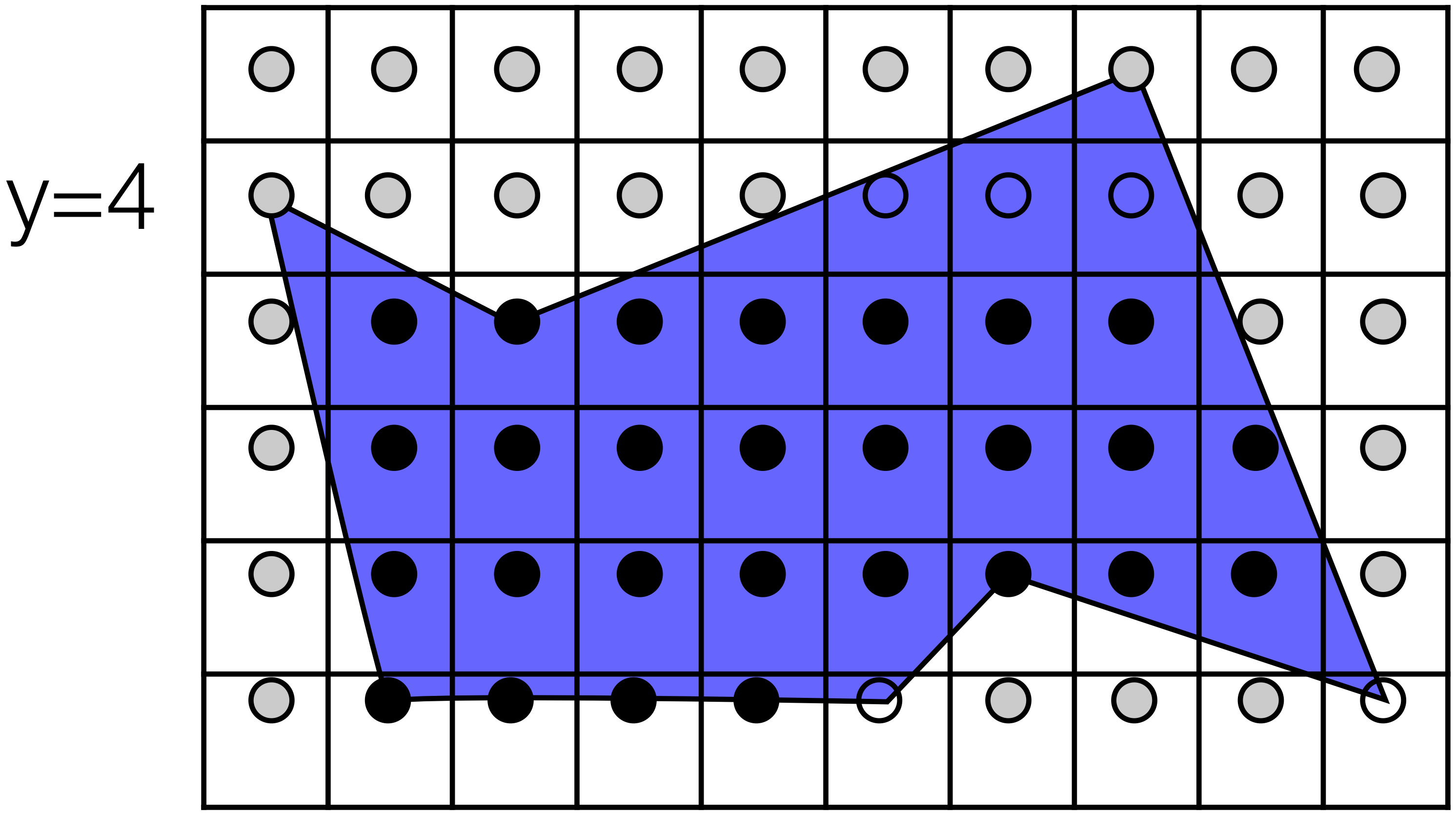
Example



Active edge list

x	inc	y out
0.25	-0.25	4
2	-2	4
2	2.5	5
7.8	-0.4	5

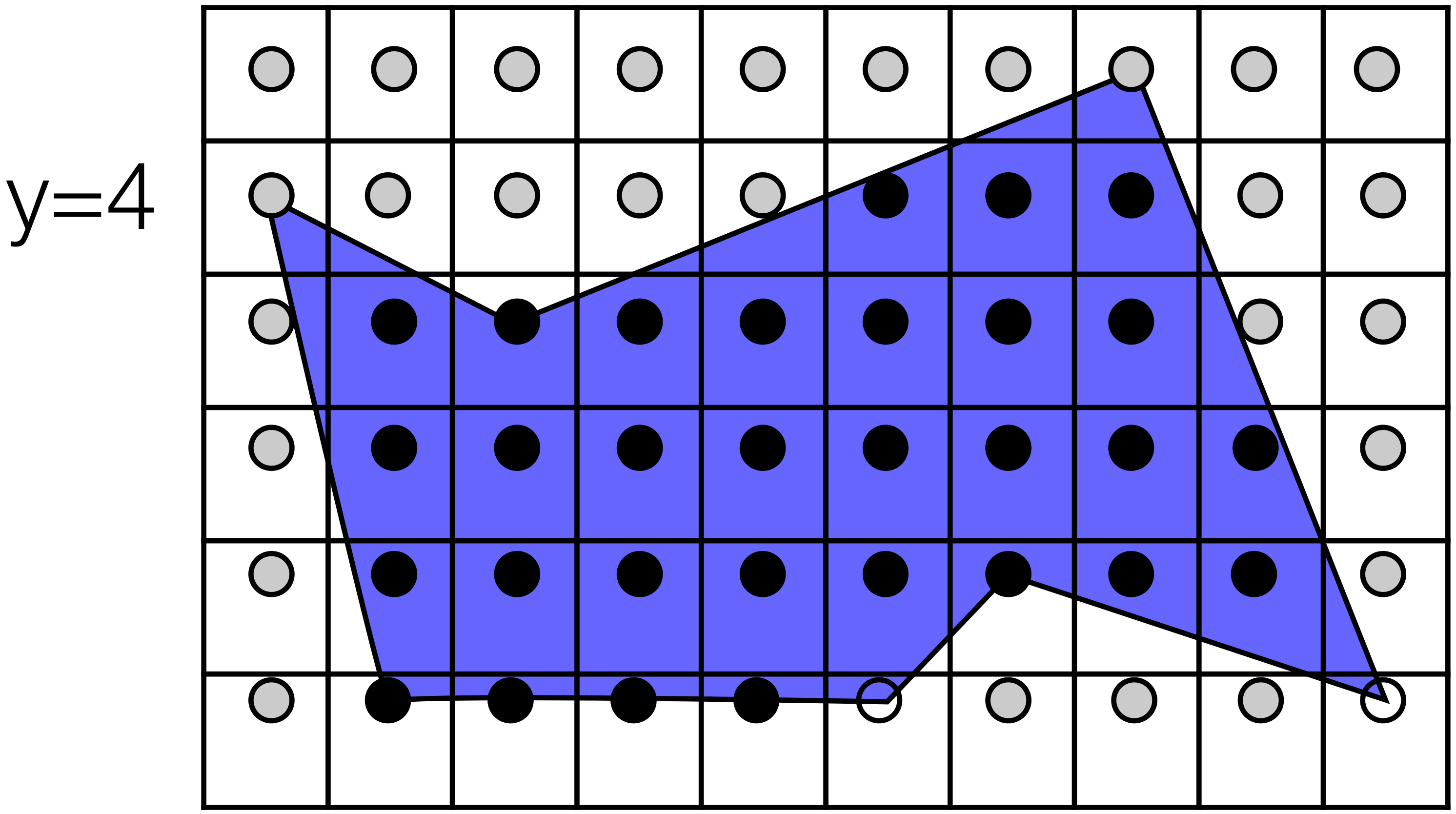
Example



Active edge list

x	inc	y out
4.5	2.5	5
7.4	-0.4	5

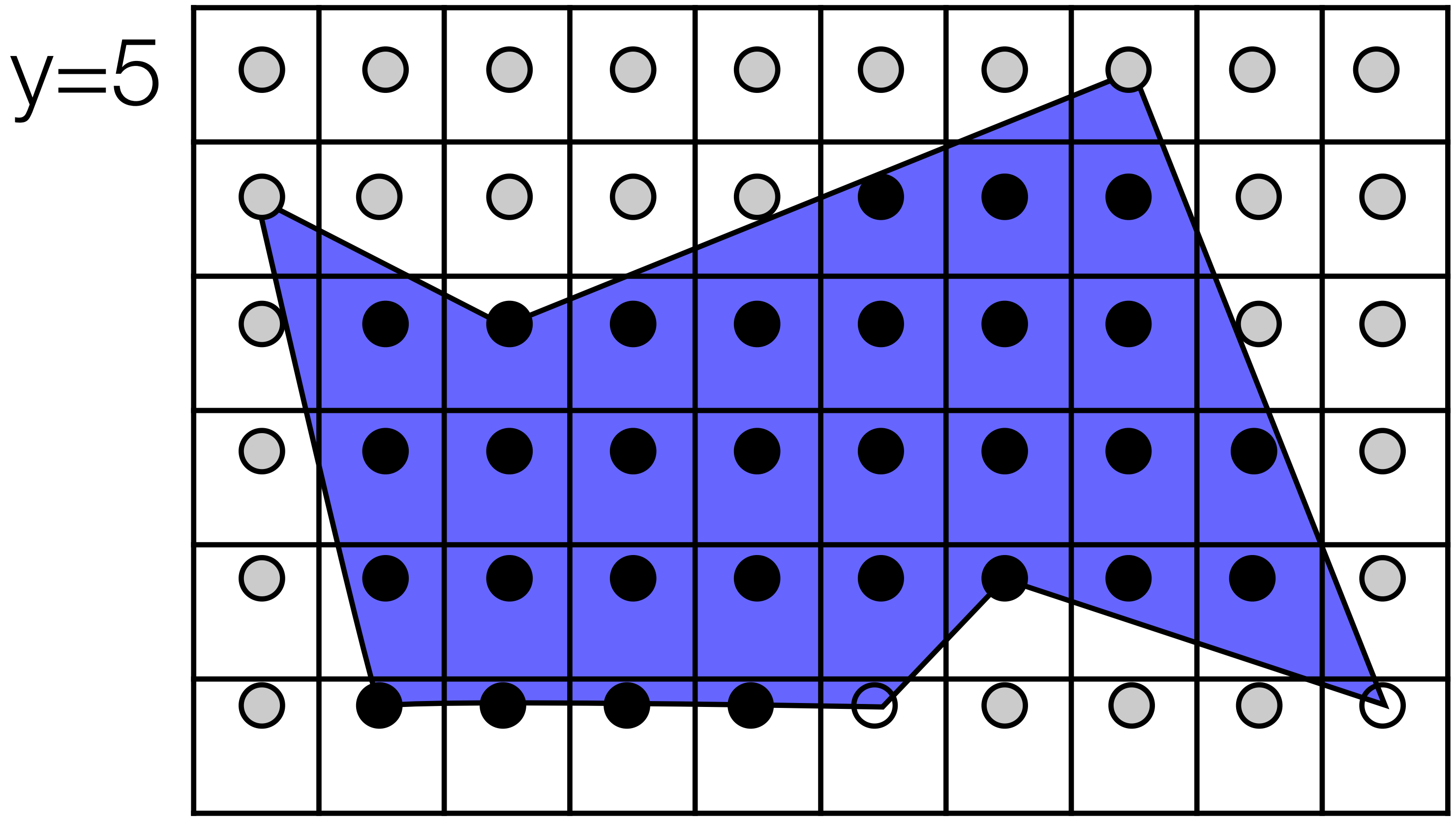
Example



Active edge list

x	inc	y out
4.5	2.5	5
7.4	-0.4	5

Example



Active edge list

x	inc	y out

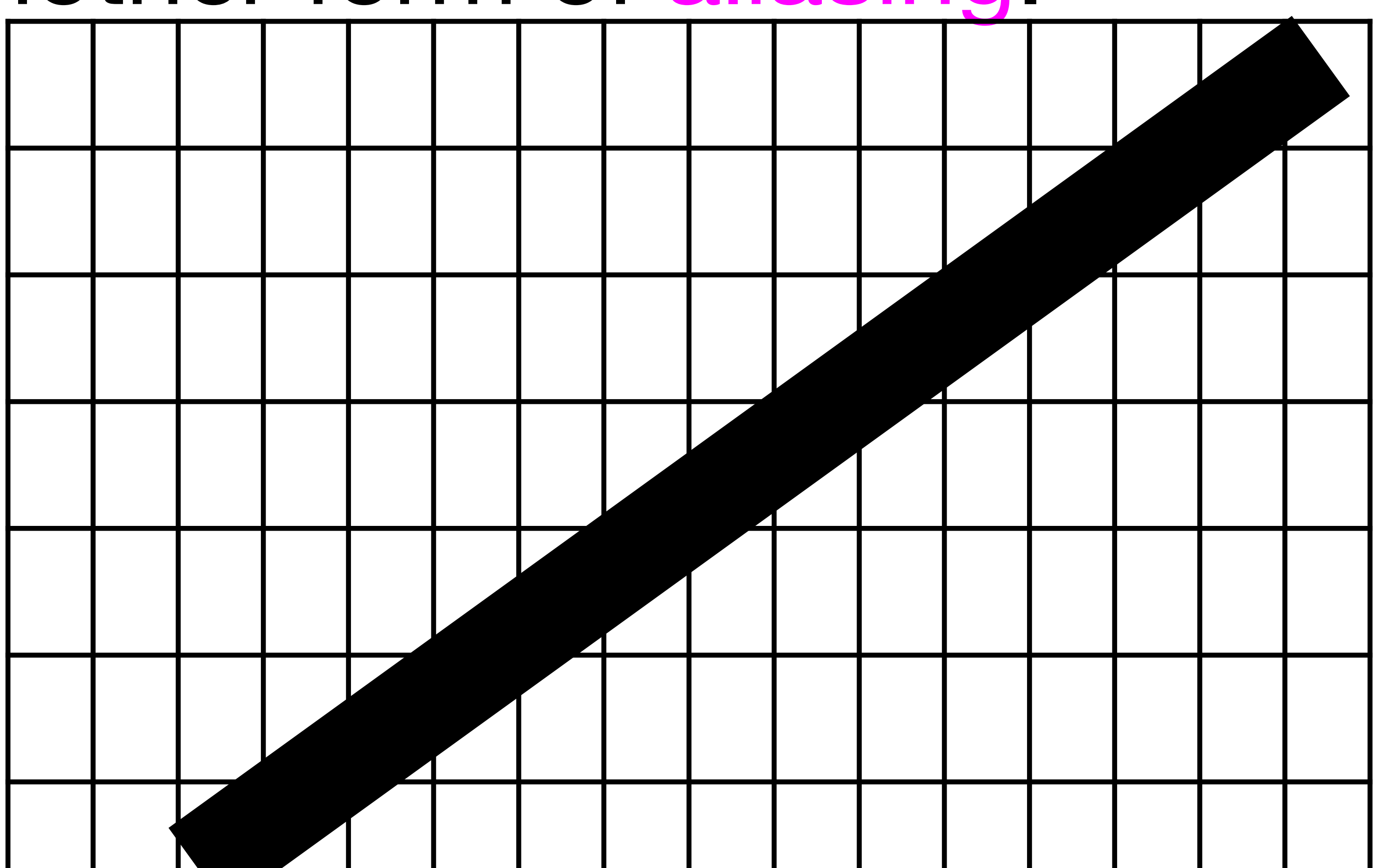
OpenGL

- OpenGL is optimised for implementation on hardware.
- Hardware implementations do not work well with variable length lists.
- If polygons are convex the active edge list always has 2 entries.

Aliasing

- Lines and polygons drawn with these algorithms tend to look jagged if the pixel size is too large.

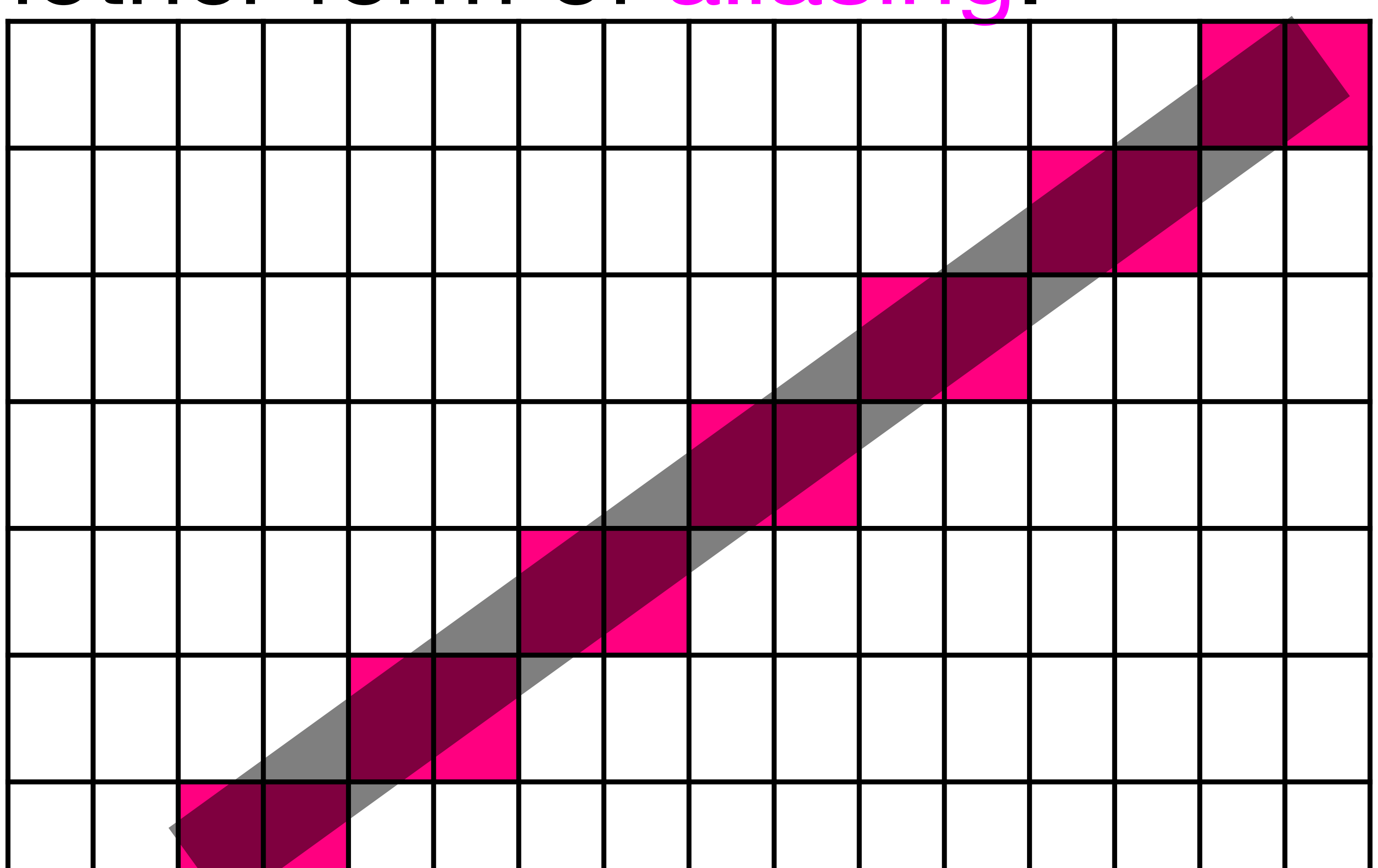
- This is another form of aliasing.



Aliasing

- Lines and polygons drawn with these algorithms tend to look jagged if the pixel size is too large.

- This is another form of aliasing.



Antialiasing

- There are two basic approaches to eliminating aliasing (antialiasing).
- Prefiltering is computing exact pixel values geometrically rather than by sampling.
- Postfiltering is taking samples at a higher resolution (supersampling) and then averaging.

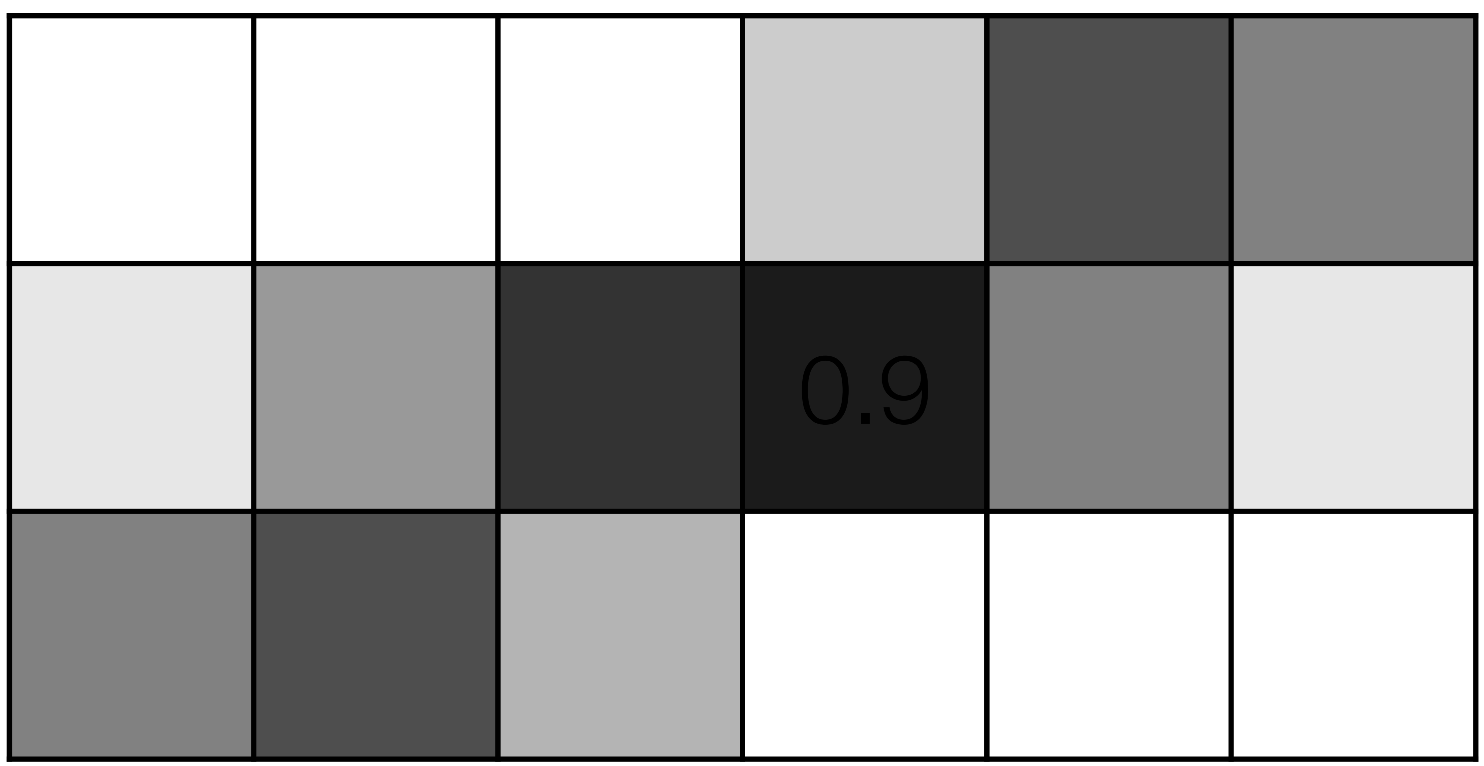
Prefiltering

- For each pixel, compute the amount occupied and set pixel value to that percentage.

0	0	0	0.2	0.7	0.5
0.1	0.4	0.8	0.9	0.5	0.1
0.5	0.7	0.3	0	0	0

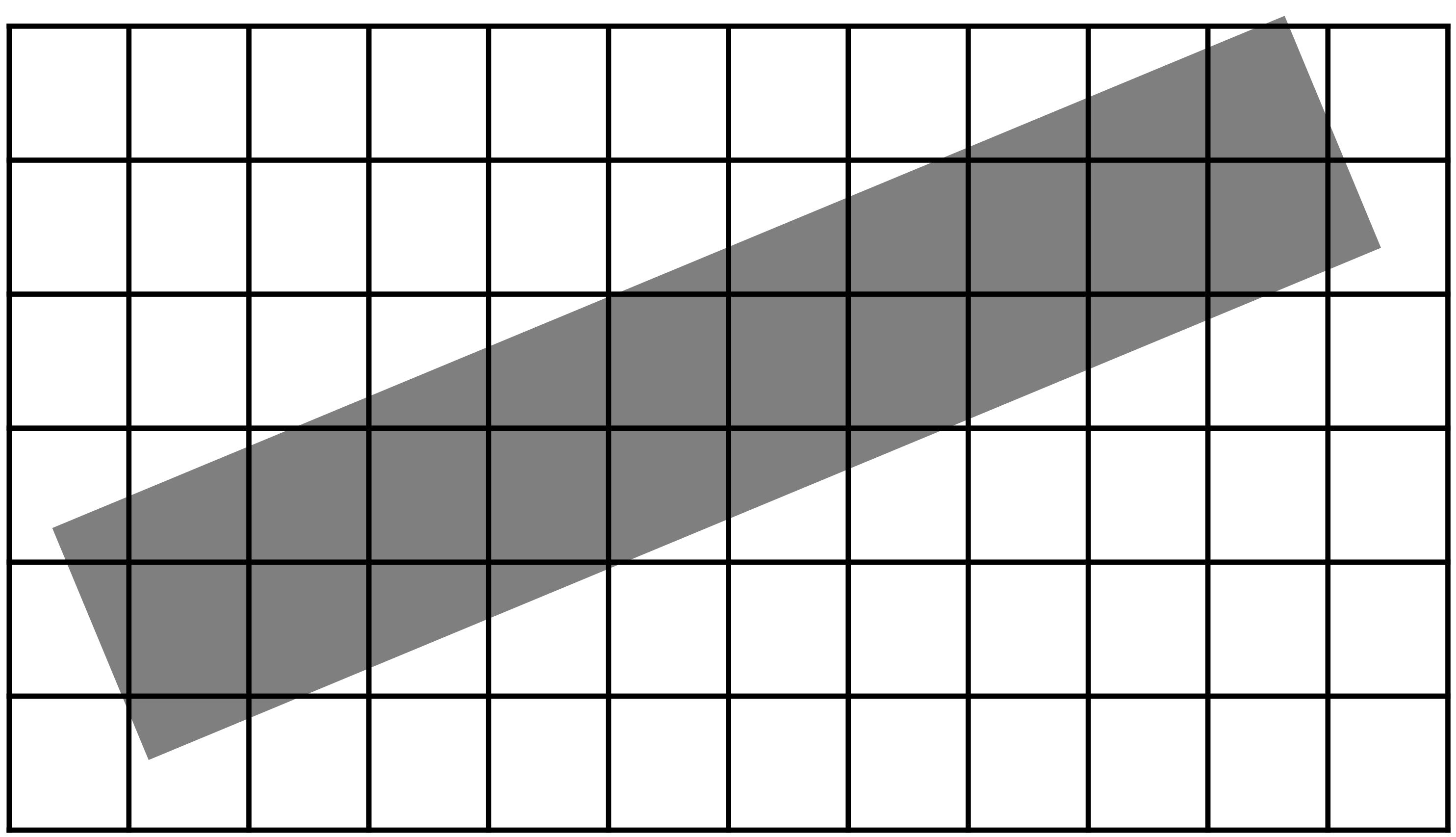
Prefiltering

- For each pixel, compute the amount occupied and set pixel value to that percentage.



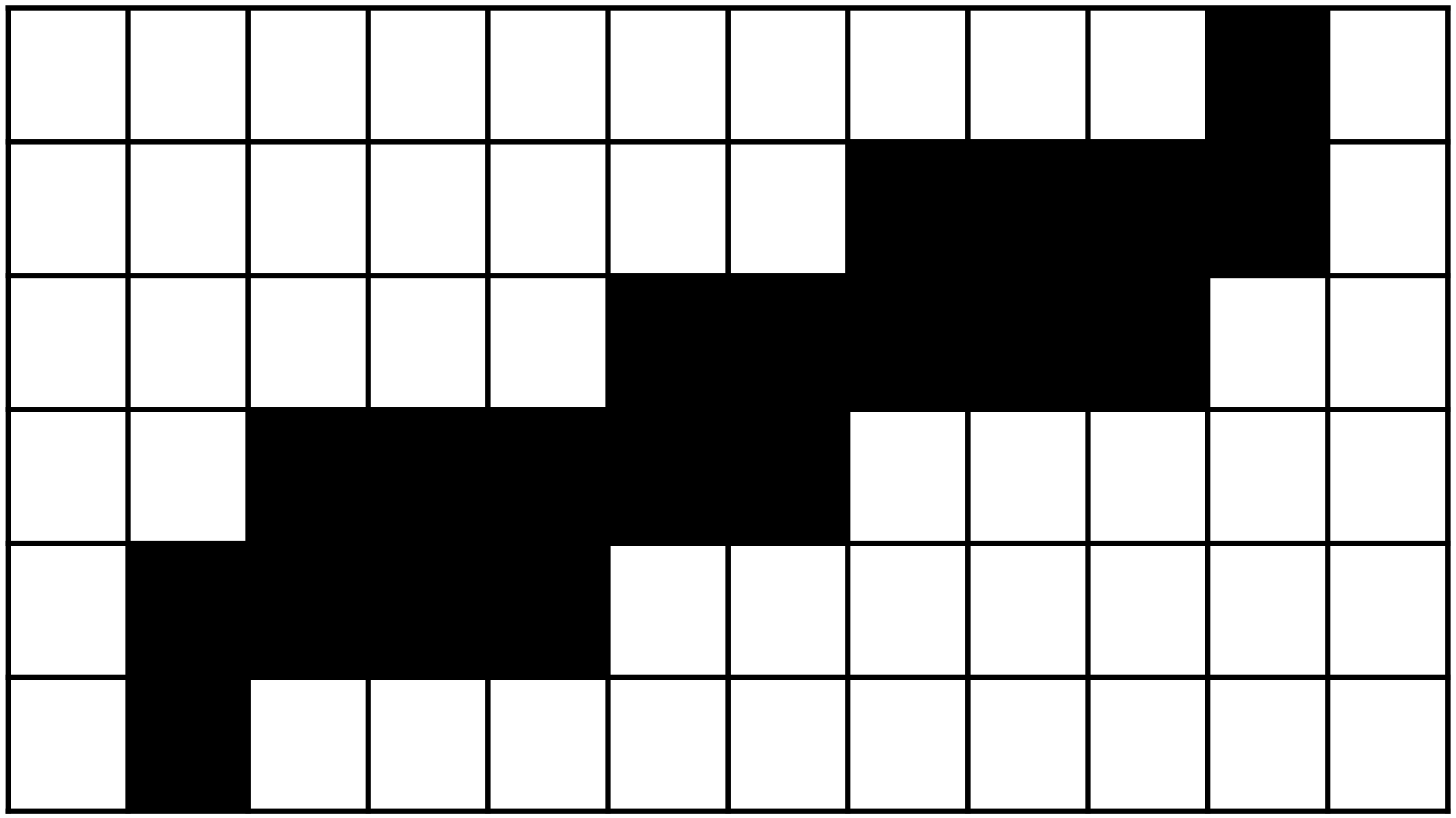
Postfiltering

- Draw the line at a higher resolution and average (supersampling).



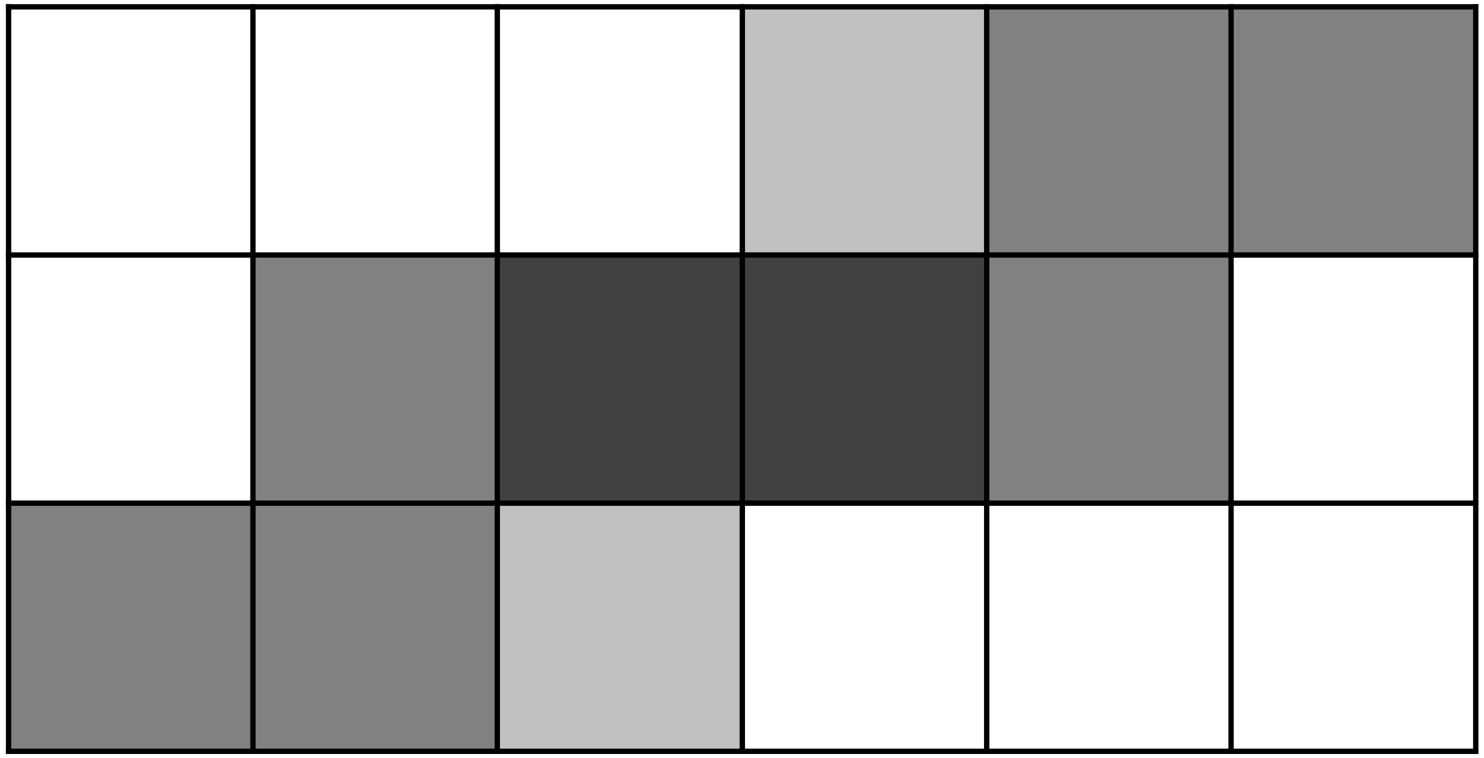
Postfiltering

- Draw the line at a higher resolution and average (supersampling)



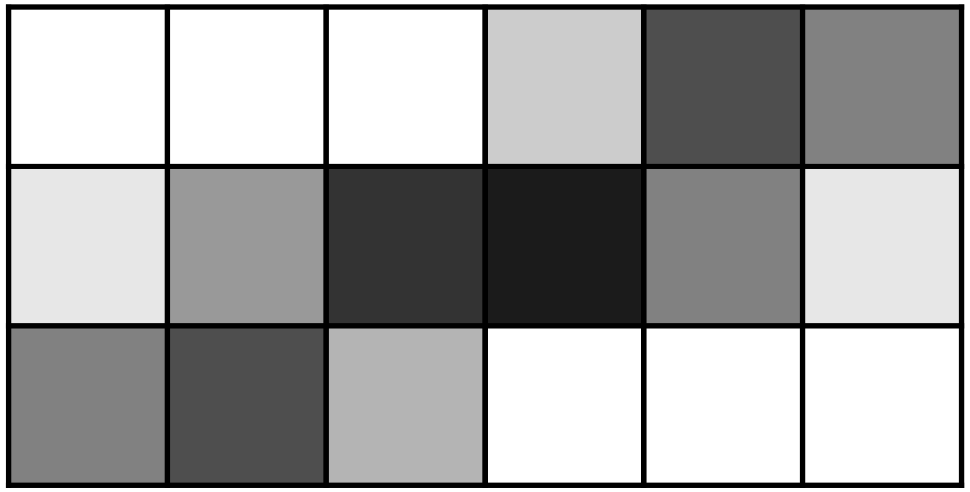
Postfiltering

- Draw the line at a higher resolution and average (supersampling)

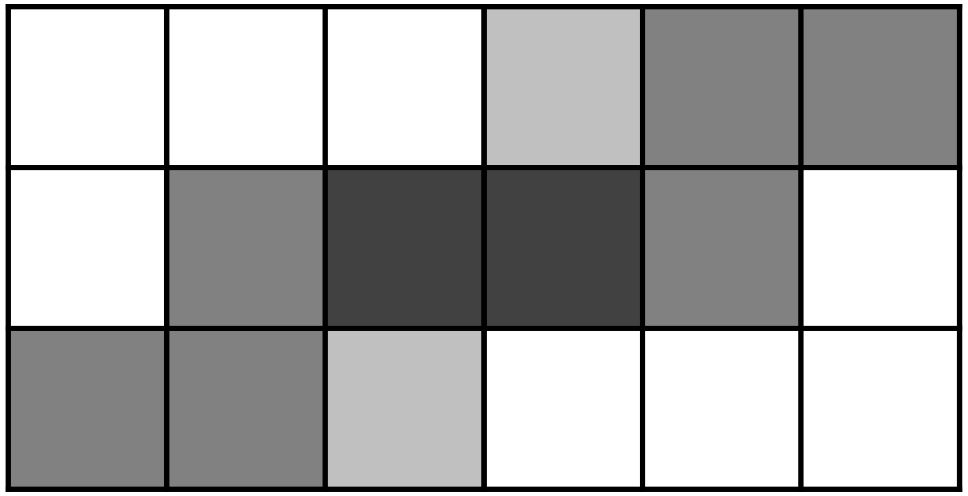


Comparing

Prefiltering



Postfiltering



Weighted postfiltering

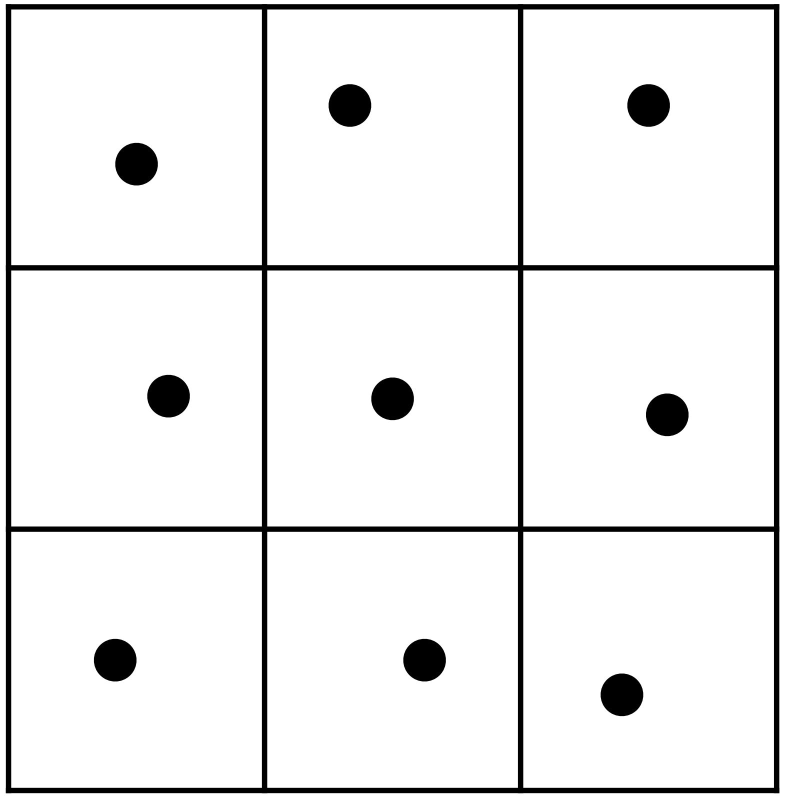
- It is common to apply weights to the samples to favour values in the center of the pixel.

1/16	1/16	1/16
1/16	1/2	1/16
1/16	1/16	1/16

Stochastic sampling

- Taking supersamples in a grid still tends to produce noticeably regular aliasing effects.

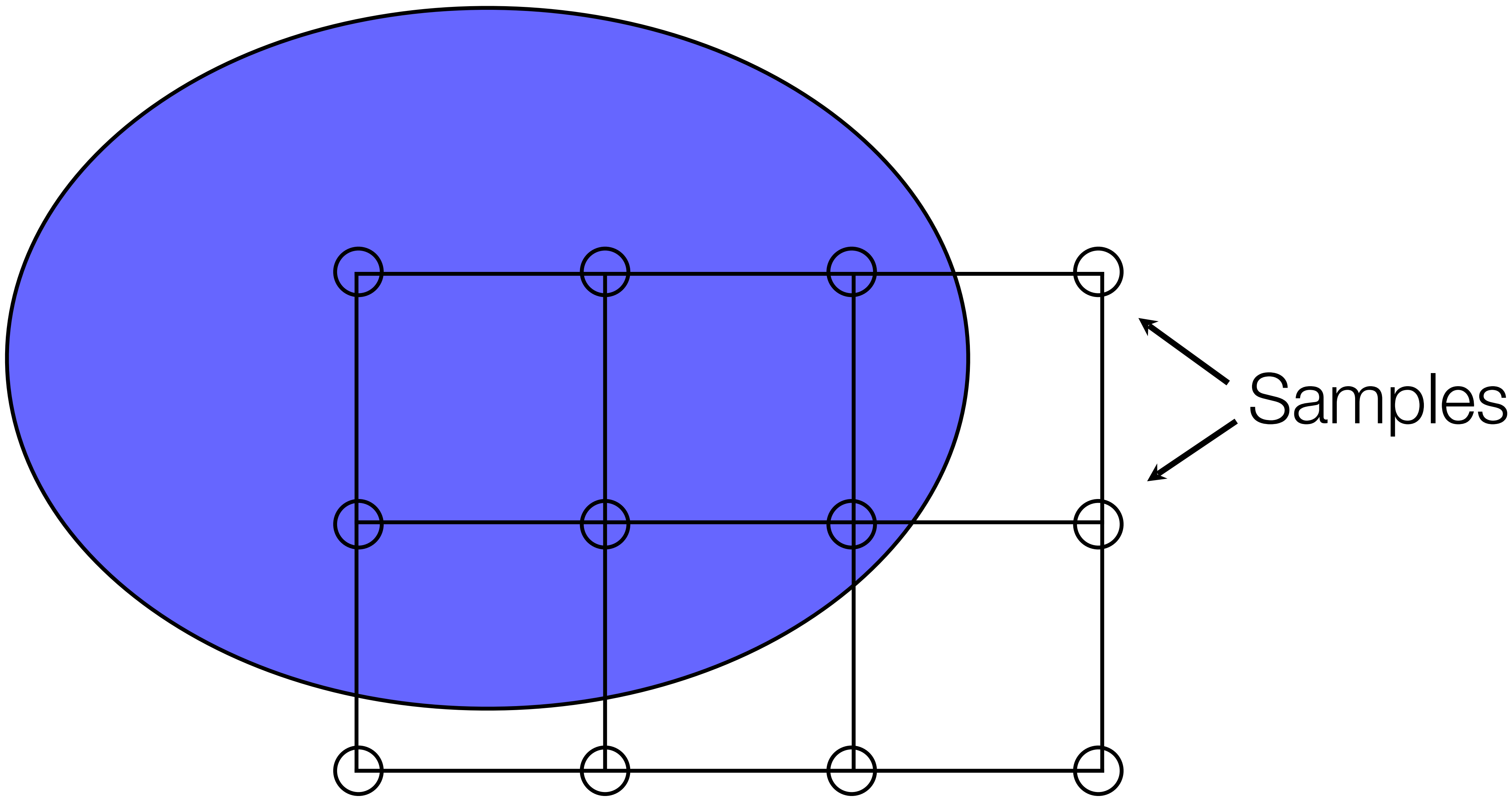
- Adding small amounts of jitter to the sampled points makes aliasing effects appear as visual noise.



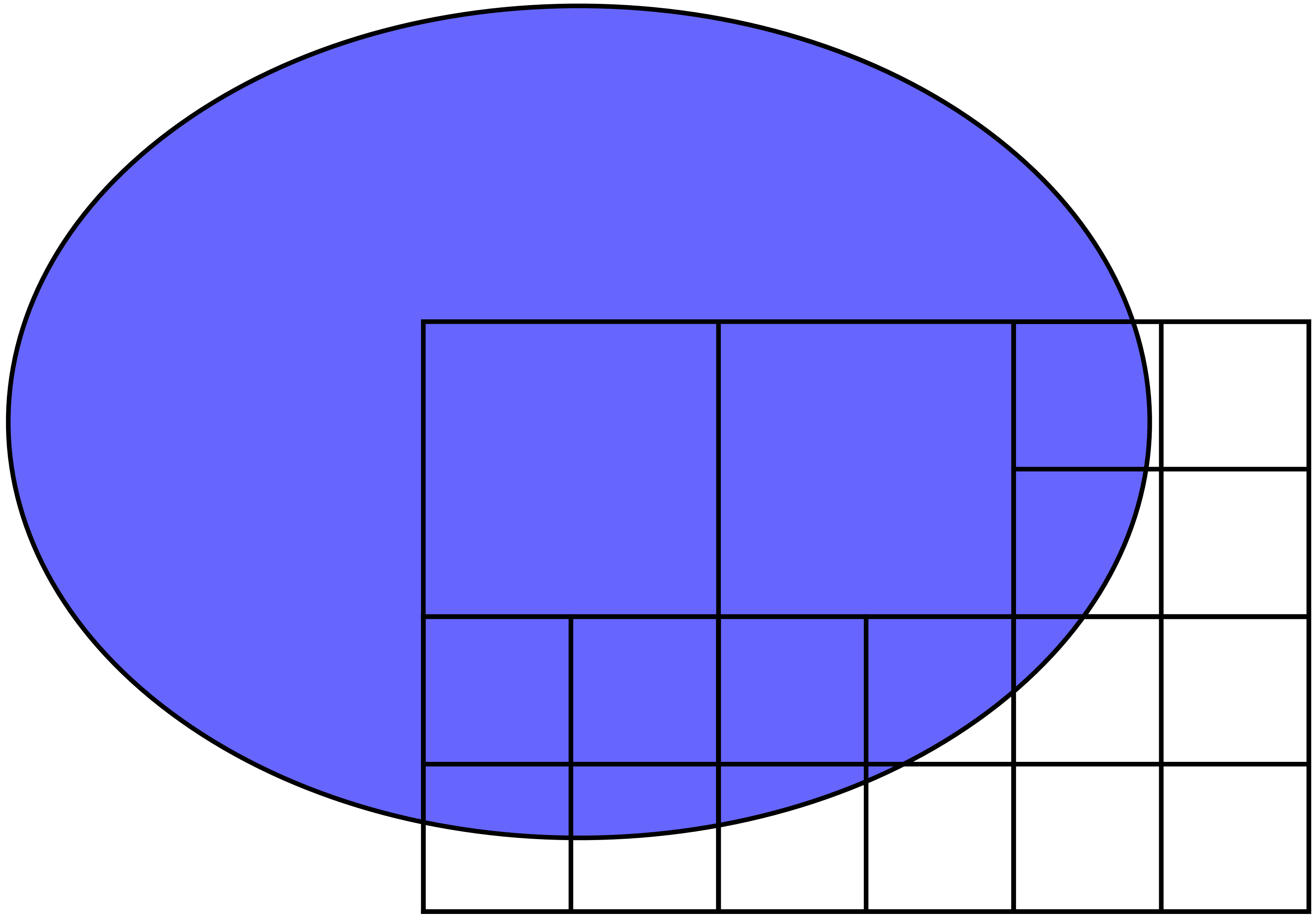
Adaptive Sampling

- Supersampling in large areas of uniform colour is wasteful.
- Supersampling is most useful in areas of major colour change.
- Solution: Sample recursively, at finer levels of detail in areas with more colour variance.

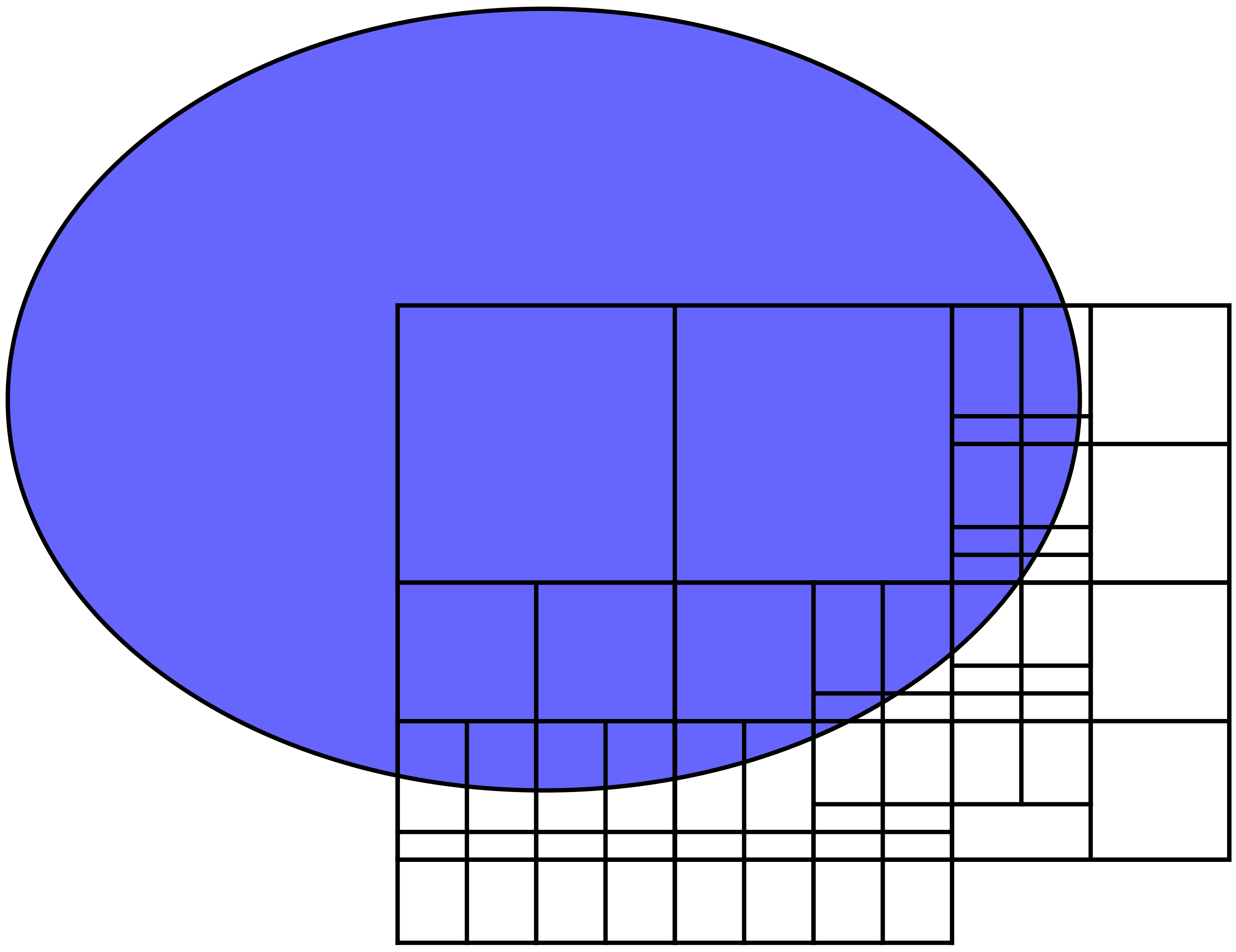
Adaptive sampling



Adaptive sampling



Adaptive sampling



Antialiasing

- Prefiltering is most accurate but requires more computation.
- Postfiltering can be faster. Accuracy depends on how many samples are taken per pixel. More samples means larger memory usage.

OpenGL

```
// implementation dependant, may not
even do anything ☺️
gl.glEnable(GL3.GL_LINE_SMOOTH);
gl.glHint(GL3.GL_LINE_SMOOTH_HINT,GL3.GL
_NICEST);

// also requires alpha blending
gl.glEnable(GL2.GL_BLEND);
gl.glBlendFunc(GL2.GL_SRC_ALPHA,
       GL2.GL_ONE_MINUS_SRC_ALPHA);
```

OpenGL

```
// full-screen multi-sampling
GLCapabilities capabilities =
     new GLCapabilities();
capabilities.setNumSamples(4);
capabilities.setSampleBuffers(true);
...
gl.glEnable(GL.GL_MULTISAMPLE);
```