COMP3421

Lighting, Transparency

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

Recap: Lighting summary

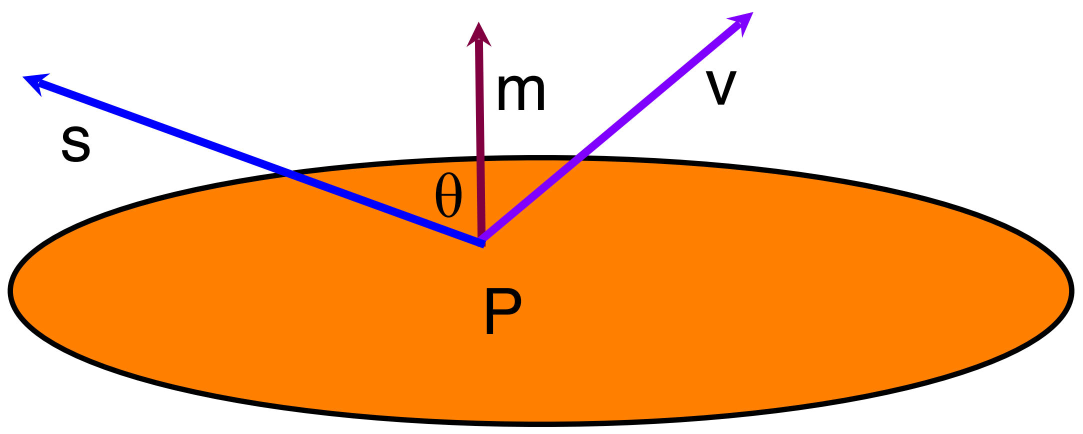
Relative position of the object to…

Component

	Light	Viewer
Ambient	❌	❌
Diffuse	✅	❌
Specular	✅	✅

Recap: 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.



Lighting summary

- Ambient light

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

- Diffuse light

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

- Specular light

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

$$I_{sp} = \max(0, I_s \rho_{sp} (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f)$$

Shading

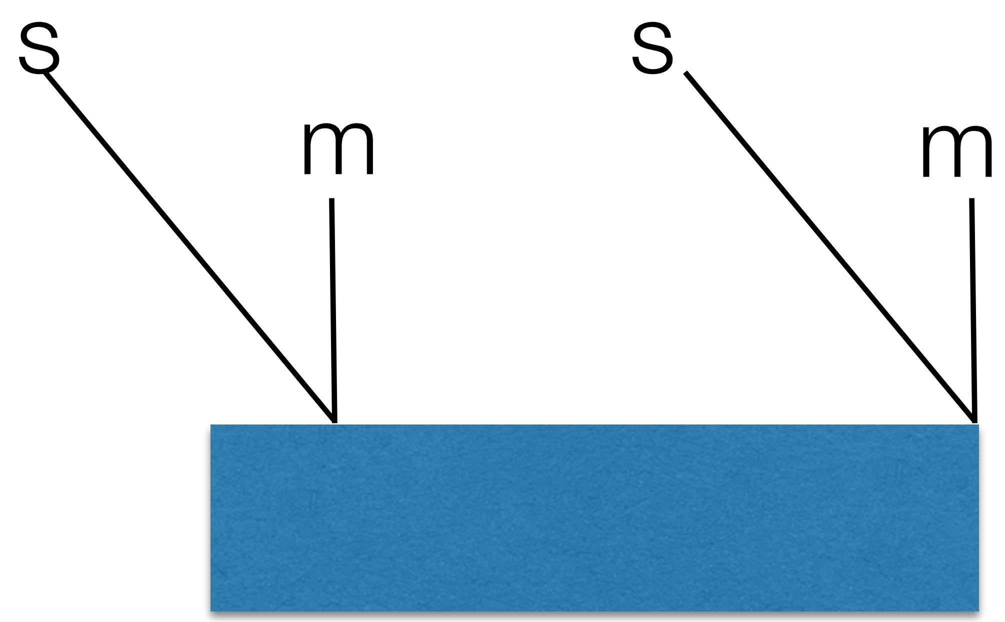
- Illumination (a.k.a shading) can be done at different points in the graphics pipeline.
- There are three common alternatives:
  - Flat shading — Calculated for each face
  - Gouraud shading — Calculated for each vertex and interpolated for every fragment
  - Phong shading — Calculated for every fragment

Flat shading

- The simplest option is to shade the entire face the same colour:
  - Compute the intensity for some point on the face (usually the first vertex)
  - Set every pixel to that value.
- See FlatShadedCube.java

Flat shading

- Flat shading is good for:
  - Diffuse illumination
  - For flat surfaces with distant light sources
  - Non-realistic/retro rendering
- It is the fastest shading option.

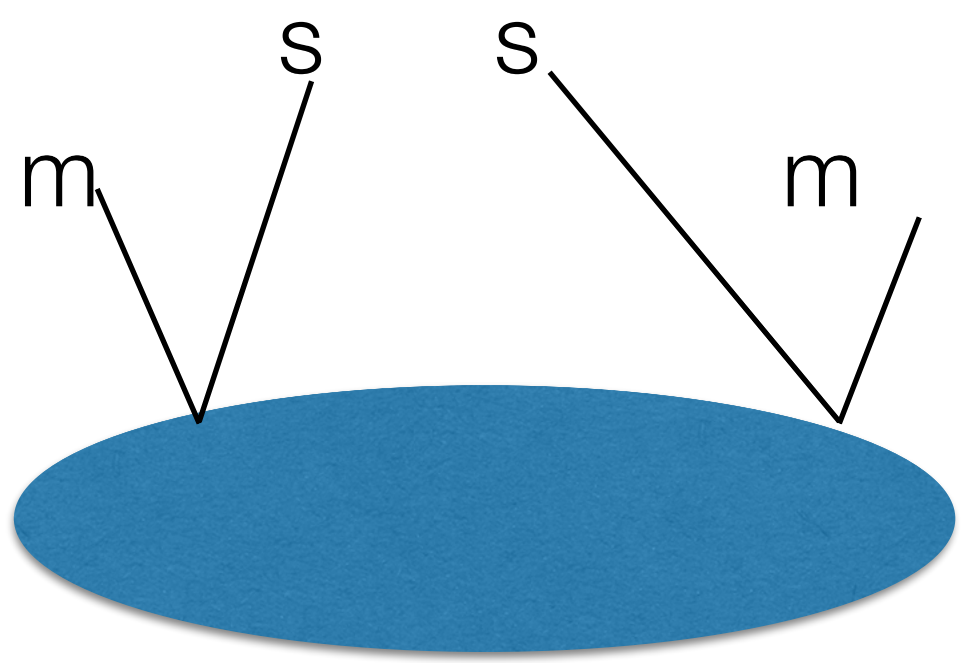


constant diffuse illumination

flat surface = constant normal

Flat shading

- Flat shading is bad for:
  - close light sources
  - specular shading
  - curved surfaces

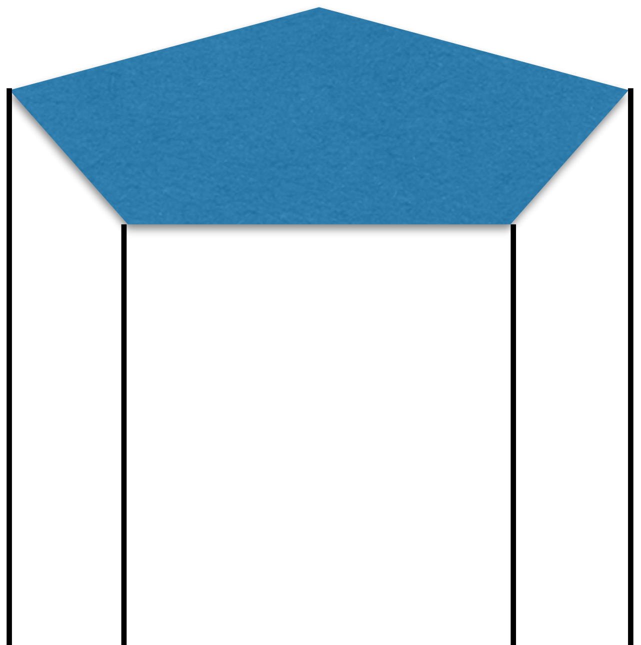


curved surface =
varying normal

varying
diffuse + specular illumination

Gouraud shading

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

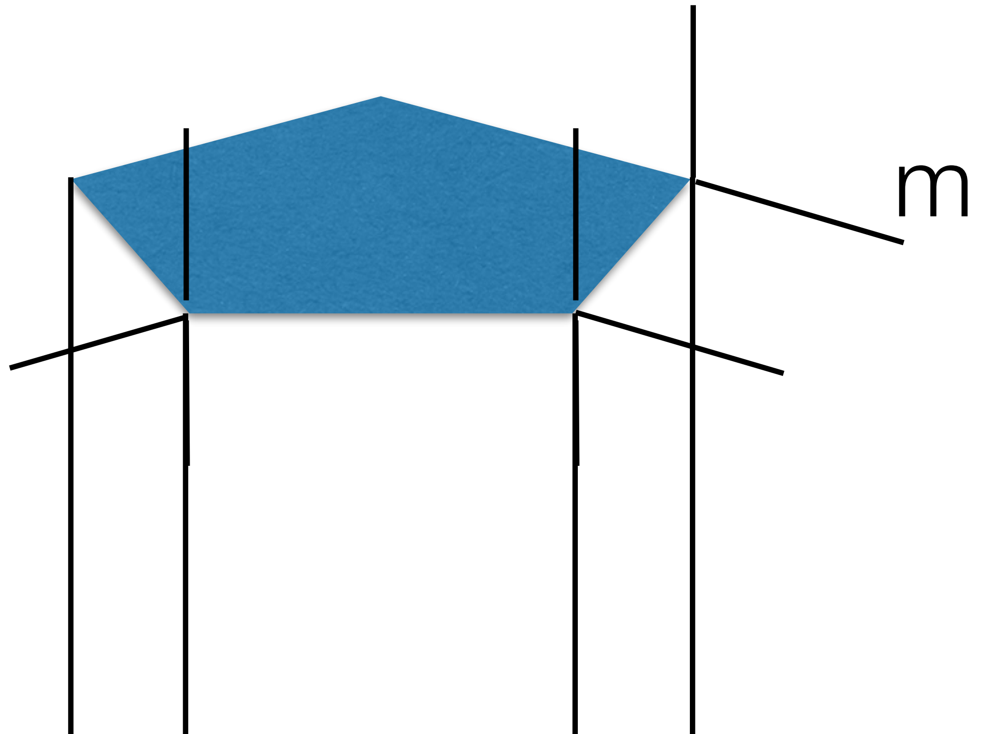


Illumination
is calculated at
each of these
vertices.

Gouraud shading

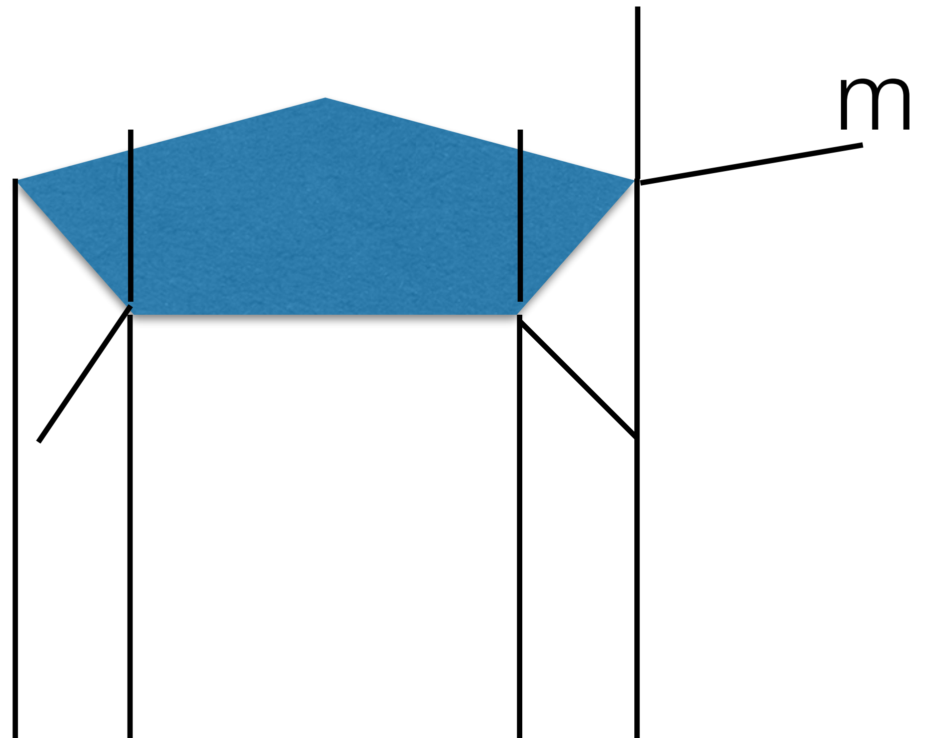
- The normal vector m used to compute the lighting equation is accessed from a buffer the same size as the vertex buffer

```
gl.glBindBuffer(GL.GL_ARRAY_BUFFER, normalsName);
gl.glVertexAttribPointer(Shader.NORMAL, 3, GL.GL_FLOAT, false, 0, 0);
```



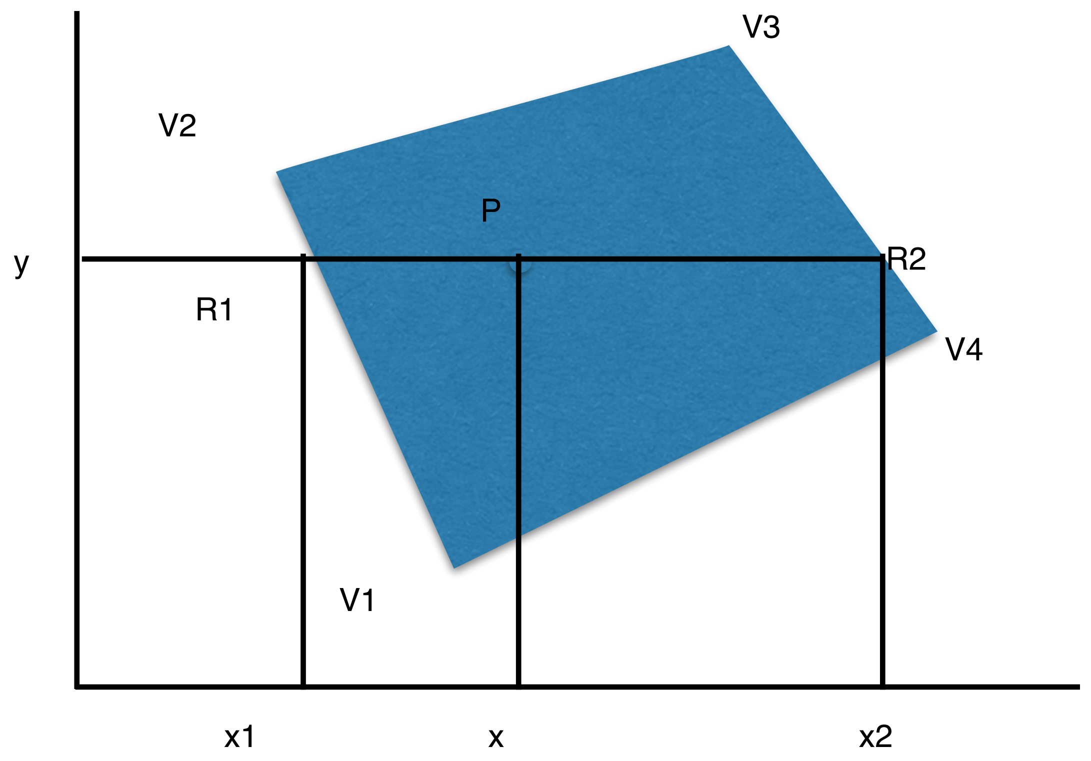
Gouraud shading

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



Gouraud shading

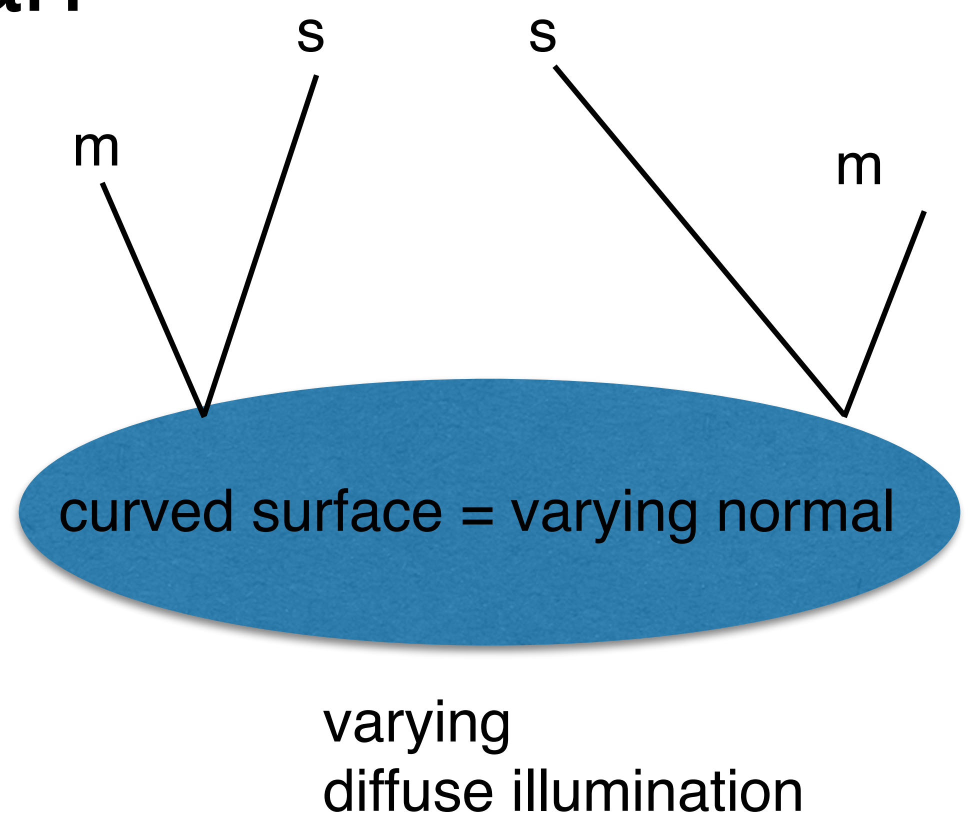
- Gouraud shading is a simple smooth shading model.
- We calculate fragment colours by bilinear interpolation on neighbouring vertices.



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

Gouraud shading

- Gouraud shading is better than flat shading for:
  - curved surfaces
  - close light sources
  - diffuse shading



Gouraud shading

- Gouraud is 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.
  - Highlights can appear to jump around from vertex to vertex with light/camera/object movement

Implementing Gouraud Shading

- Use a vertex shader to do the lighting calculation with the intensity as an output
- Intensity values are interpolated inputs to the fragment shader
- We modulate the intensity with the colour
- See vertex_gouraud.glsl and fragment_gouraud.glsl

Lighting summary

Shading

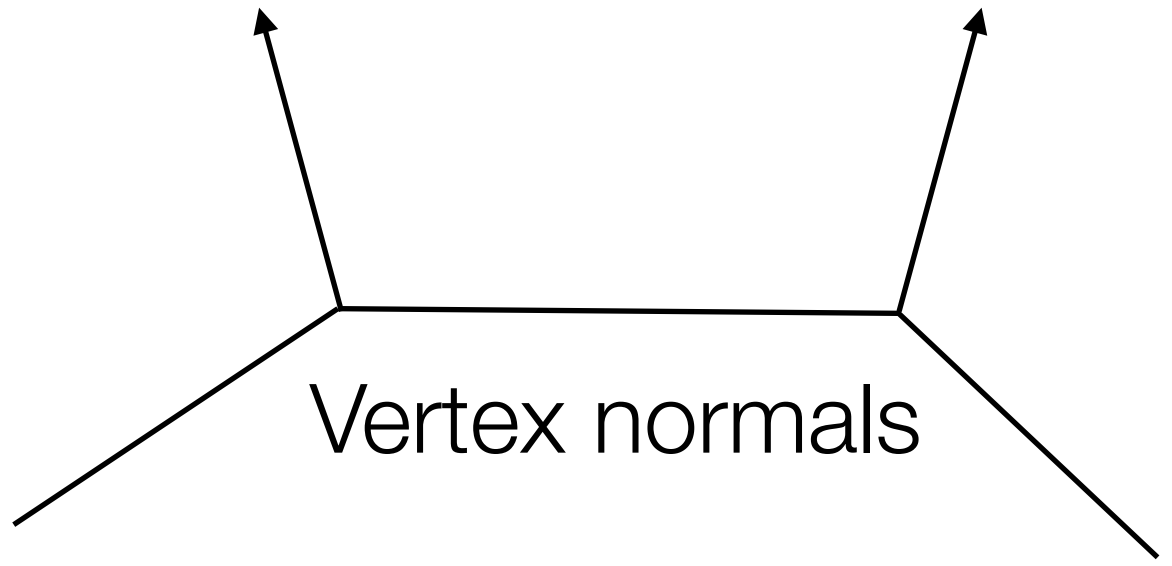
	Where	Good for	Bad for
Flat	Face	Flat surfaces, a retro blocky look	Curved surfaces, specular highlights
Gouraud	Vertex	Curved surfaces, diffuse shading	Specular highlights
Phong	Fragment	**?**	**?**

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 fragment rather than per vertex.

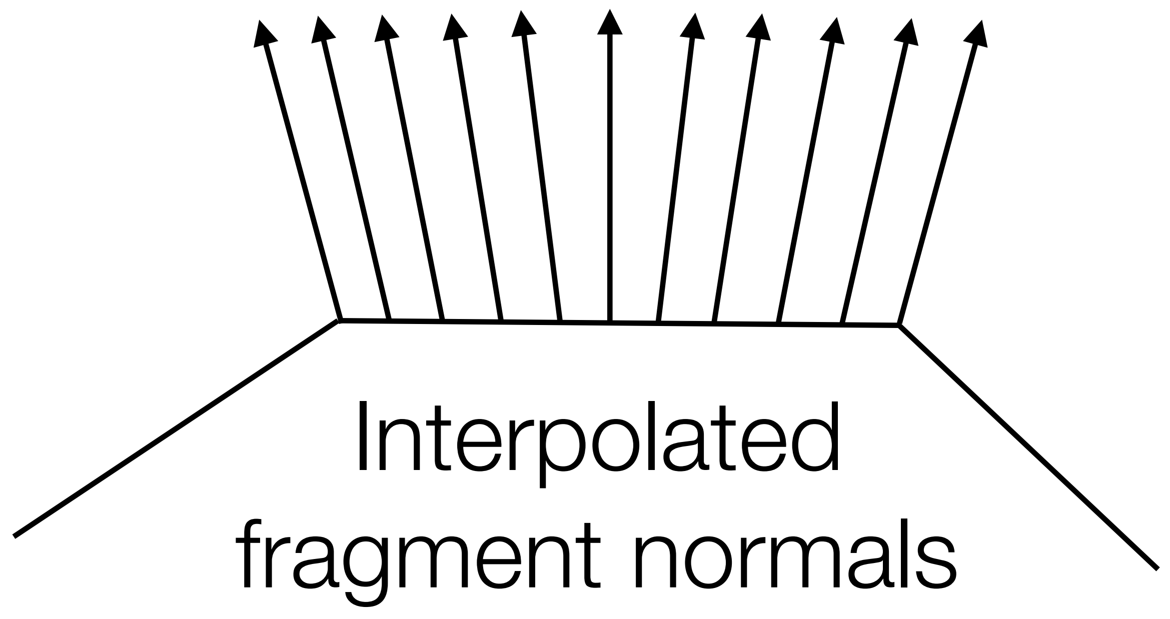
Normal interpolation

- 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 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.

Implementing Phong Shading

- See vertex_phong.glsl and fragment_phong.glsl

Phong shading

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

Phong shading

- Cons:
  - Slower than Gouraud as normals and illumination values have to be calculated per pixel rather than per vertex. In the old days this was a BIG issue. Not so much any more.

Lighting summary

		Where	Good for	Bad for
Shading	Flat	Face	Flat surfaces, a retro blocky look	Curved surfaces, specular highlights
	Gouraud	Vertex	Curved surfaces, diffuse shading	Specular highlights
	Phong	Fragment	Diffuse and specular shading	Old hardware

Things to try

- Our shaders do not handle
  - two sided lighting
  - multiple lights
  - directional lights, spotlights etc
  - attenuation…

More...

There are many more shading algorithms designed to implement different lighting techniques with different levels of speed and accuracy.

For example, Cook-Torrance is a more realistic model than Phong or Blinn-Phong

http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx

Colour

- We implement colour by having separate red, green and blue components for:
    - Light intensities $I\ I_a\ I_s$
    - Reflection coefficients $\rho_a\ \rho_d\ \rho_{sp}$
- The lighting equation is applied three times, once for each colour.

Colored Light and surfaces

- We can do this by treating the values in the equation as vectors

$$I = I_a\rho_a + \max(0, I_s\rho_d(\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})) + \max(0, I_s\rho_{sp}(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f)$$

- Where the multiplication of 2 vectors is done component-wise— i.e.

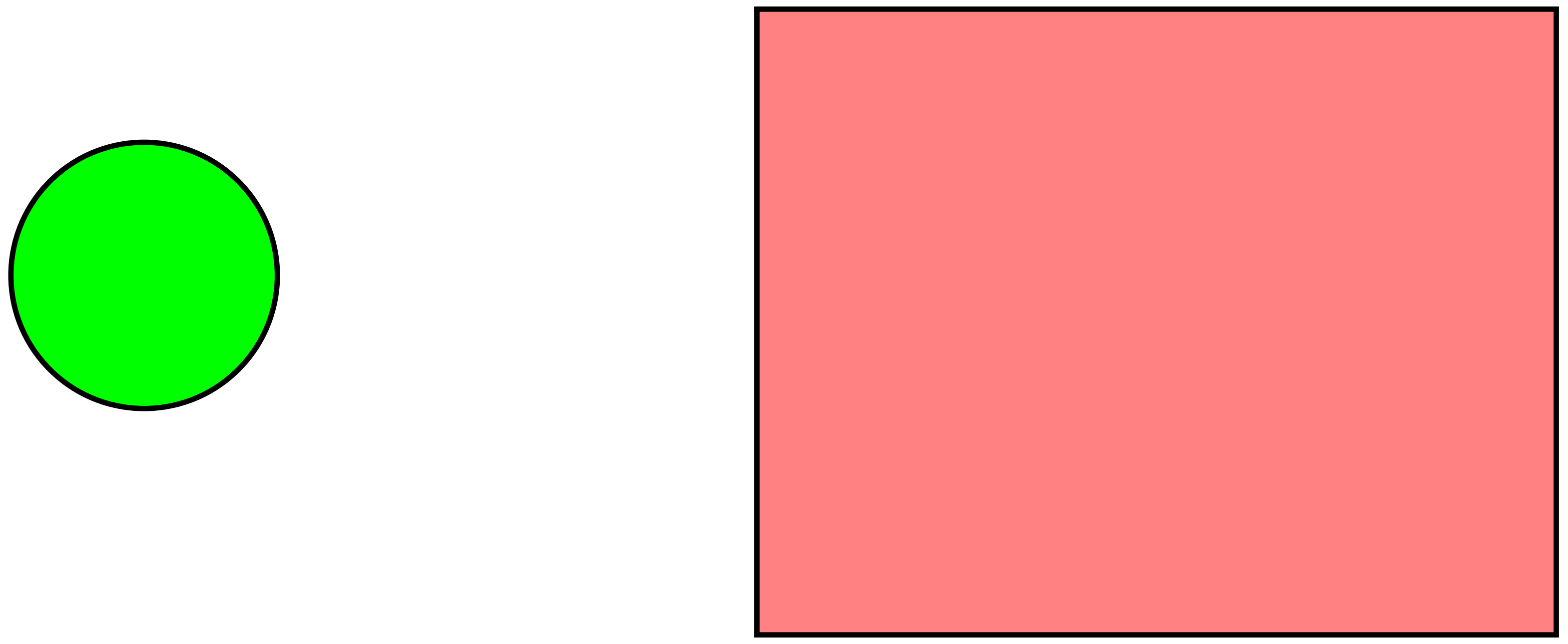
$$(u_1, u_2, u_3)(v_1, v_2, v_3) = (u_1v_1, u_2v_2, u_3v_3)$$

Caution

- Using too much light 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.

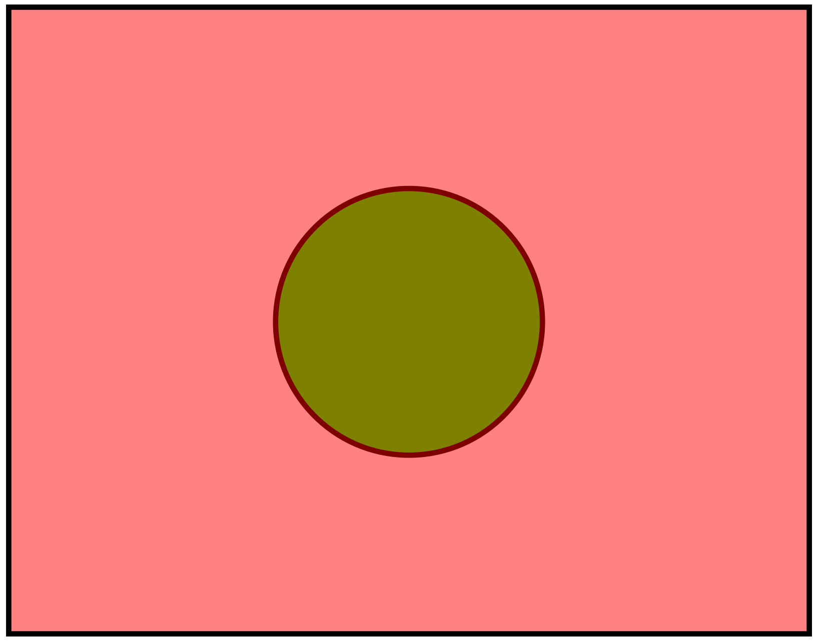
Transparency

- 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 3 components (red/green/blue)
- To make things transparent we specify an alpha component as well
- 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

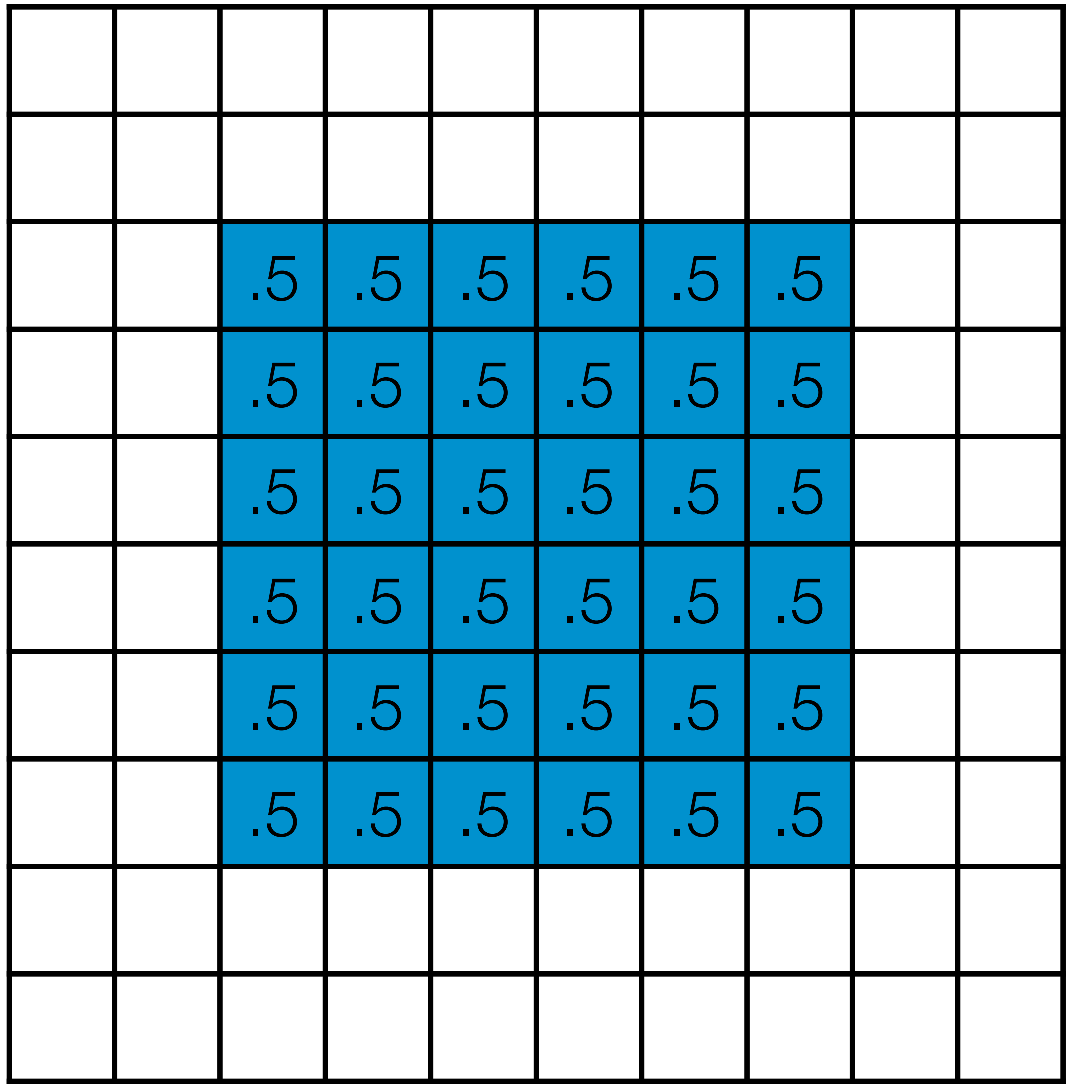
```
// Alpha blending is disabled by
// default. To turn it on:

gl.glEnable(GL2.GL_BLEND);

// other blend functions are
// also available
gl.glBlendFunc(
    GL2.GL_SRC_ALPHA,
    GL2.GL_ONE_MINUS_SRC_ALPHA);
```

Problems

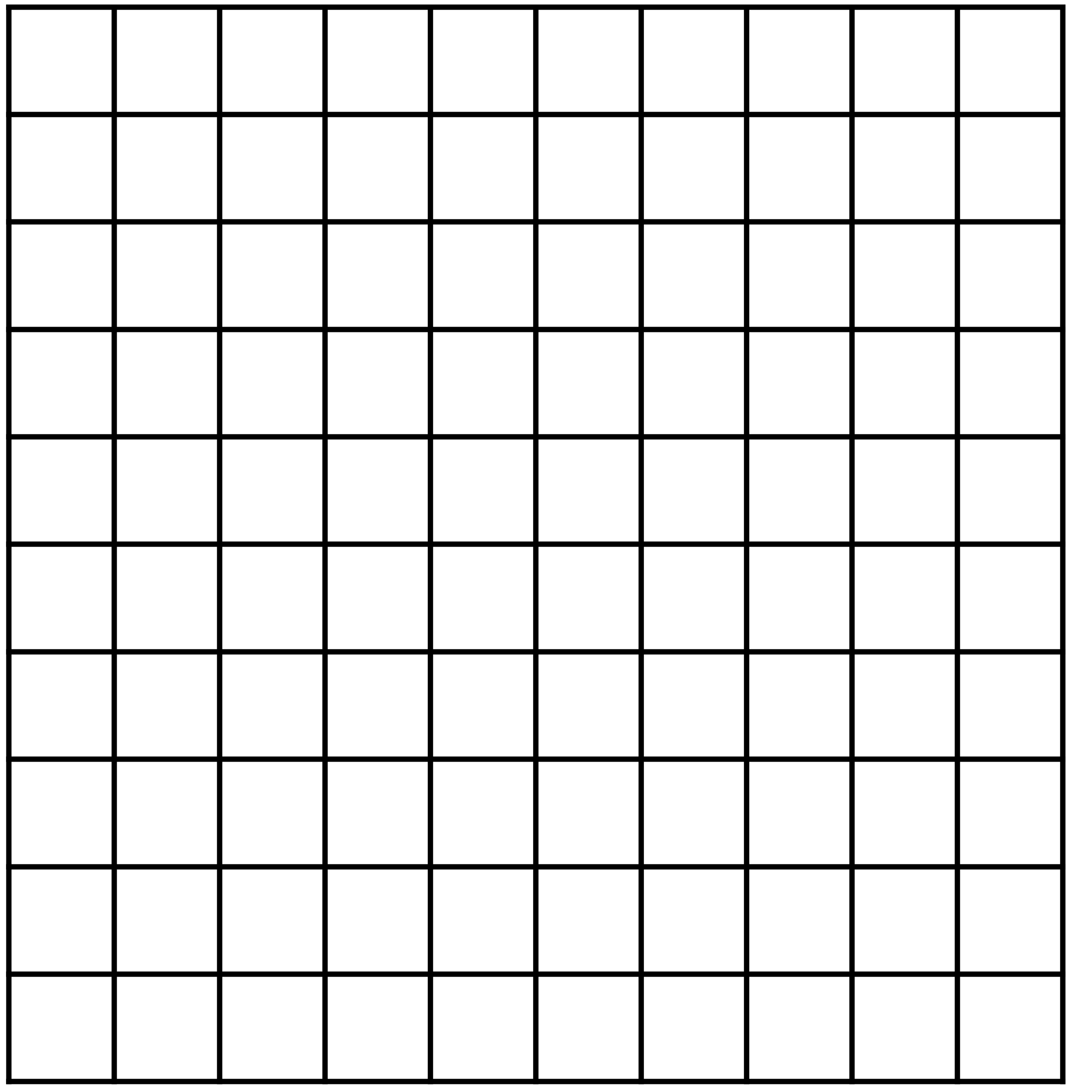
- Alpha blending depends on the order that pixels are drawn.
- You need to draw transparent polygons after the polygons behind them.
- If you are using the depth buffer and you try to draw the transparent polygon before the objects behind it, the later objects will not be drawn.

Back-to-front

Back Polygon

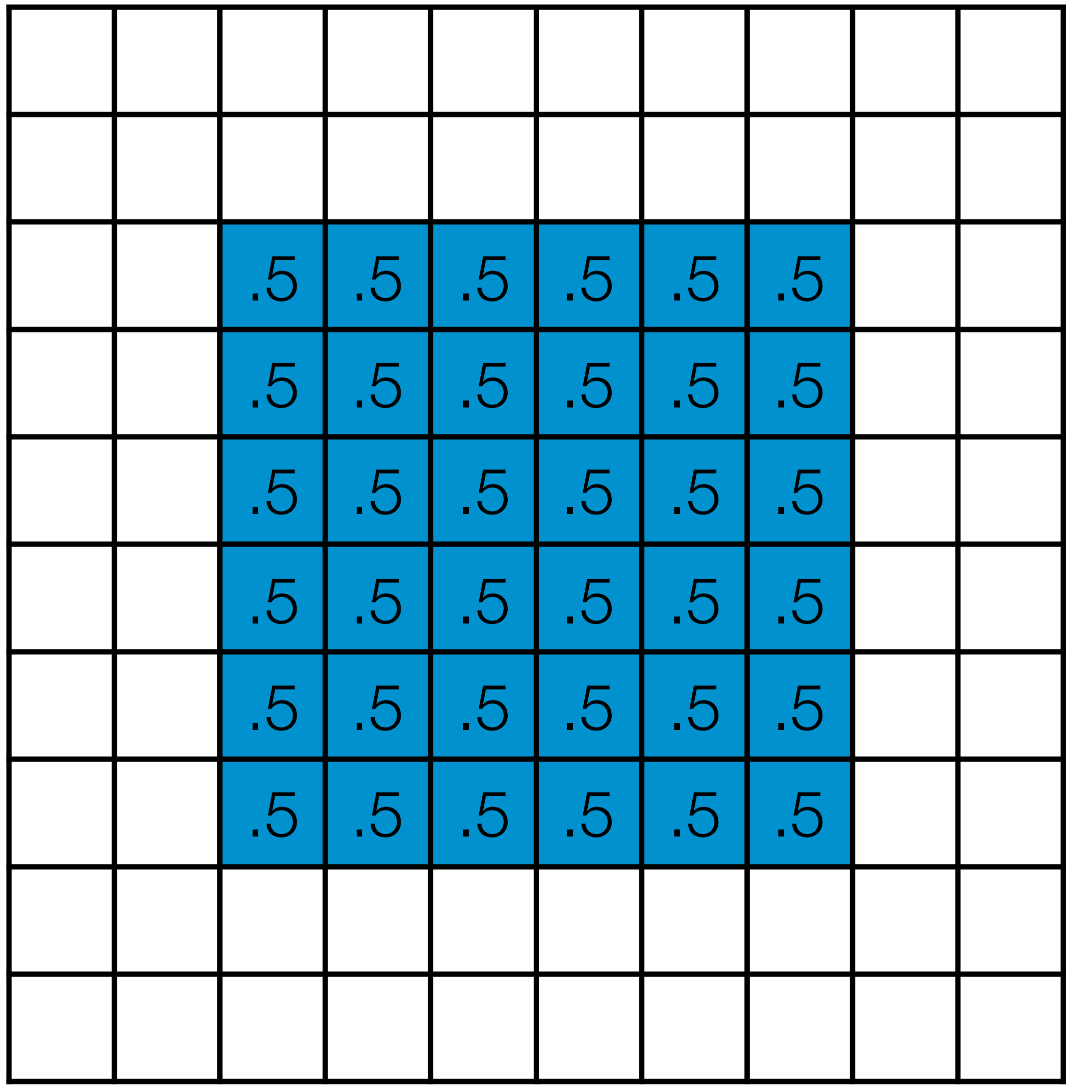


Buffer

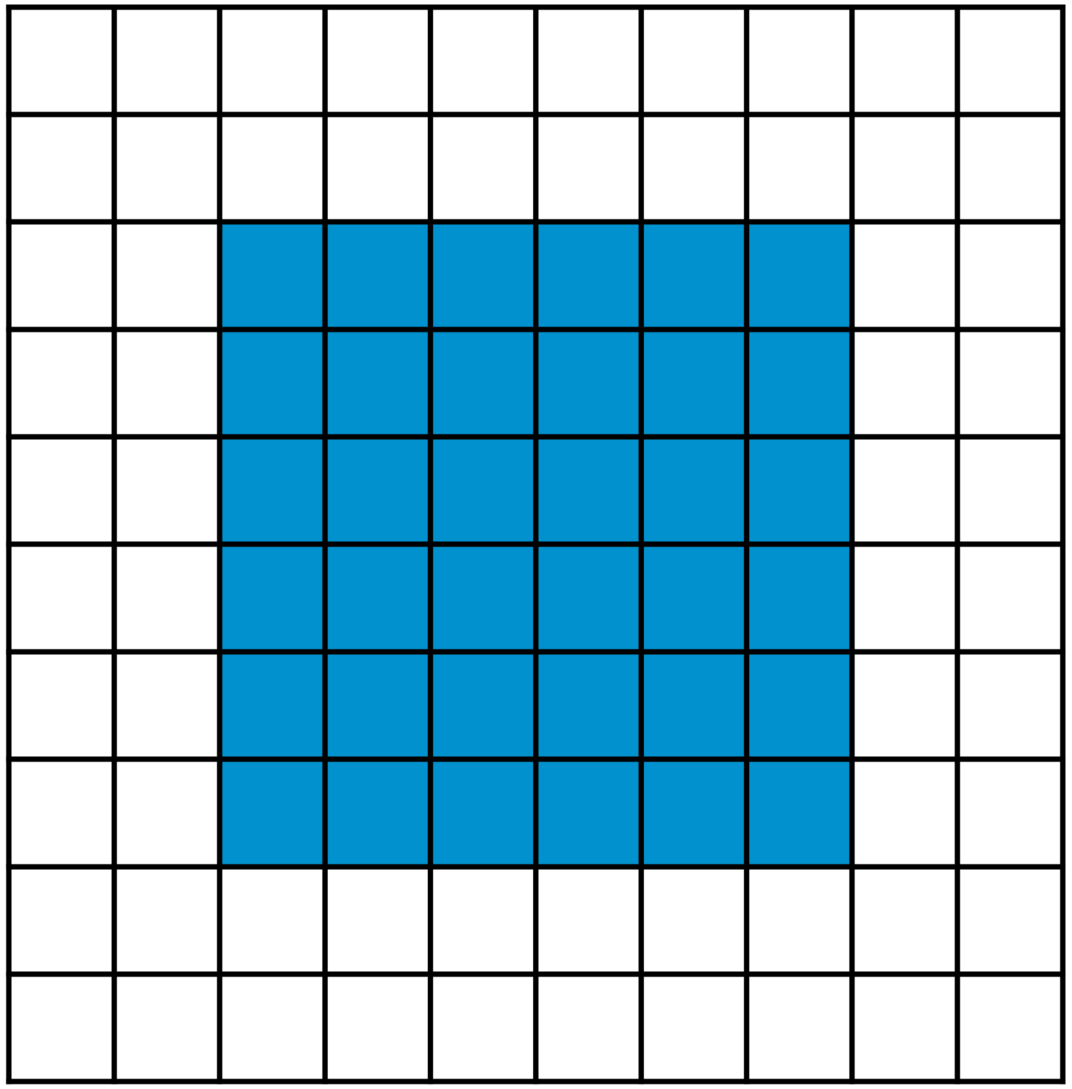


Back-to-front

Back Polygon

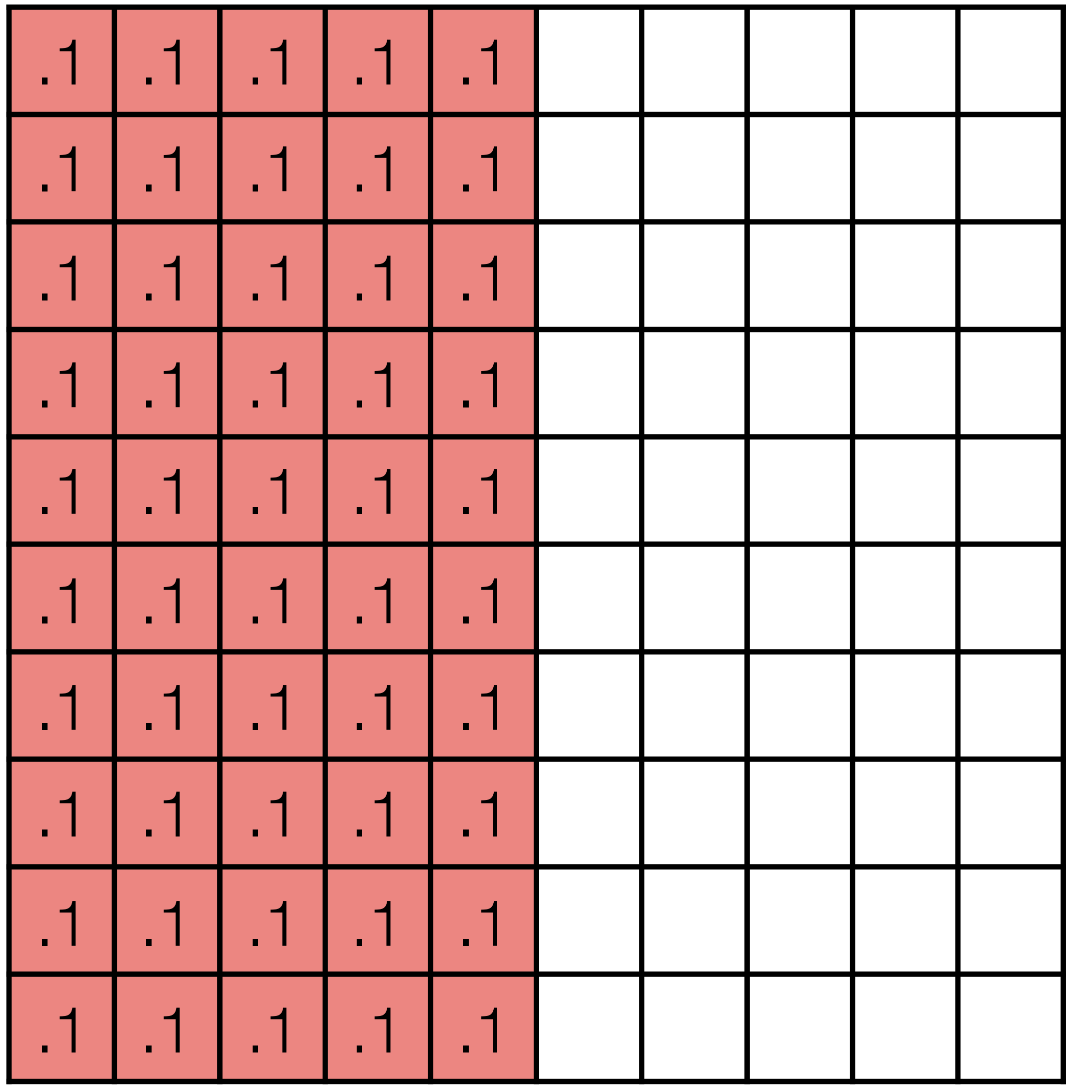


Buffer

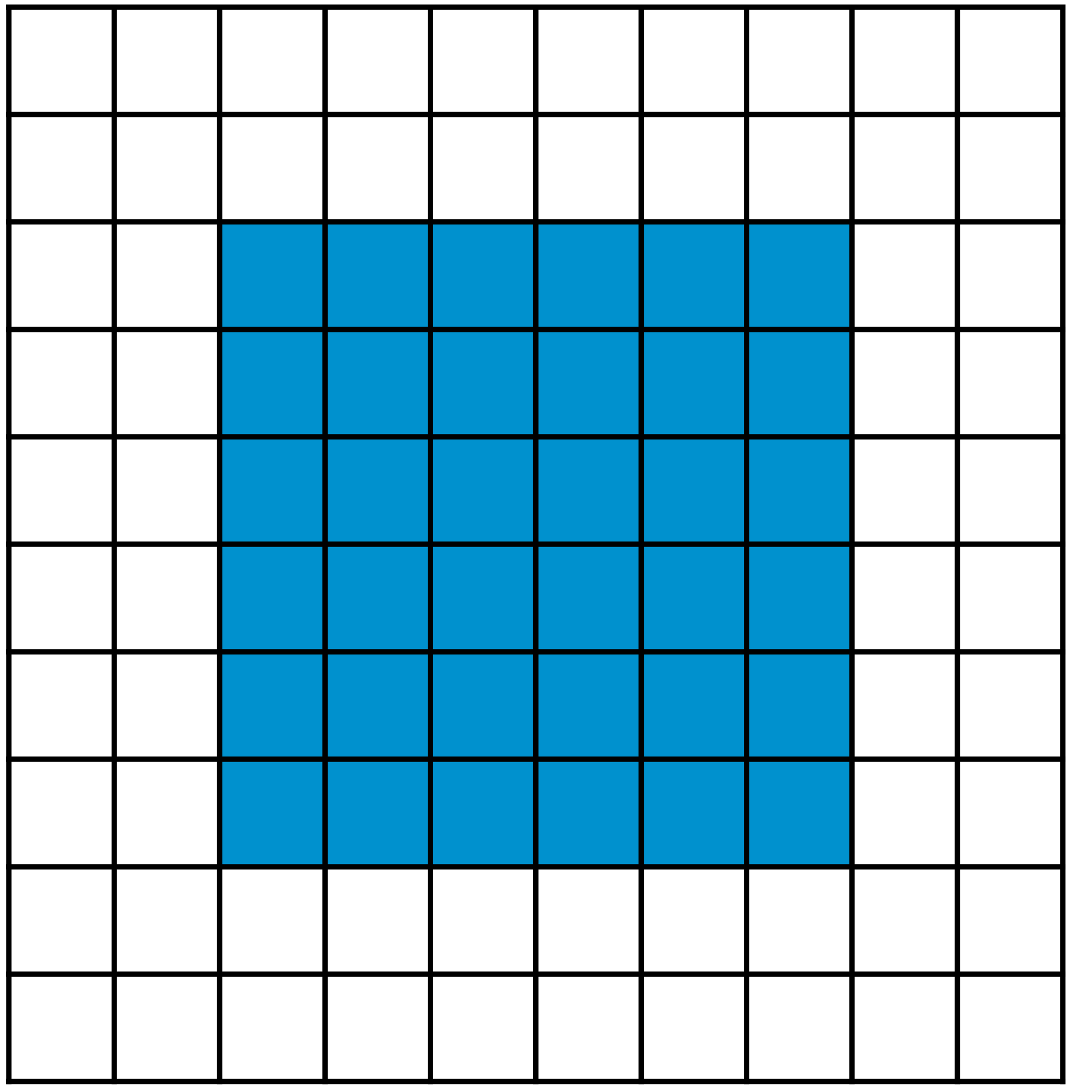


Back-to-front

Front Polygon

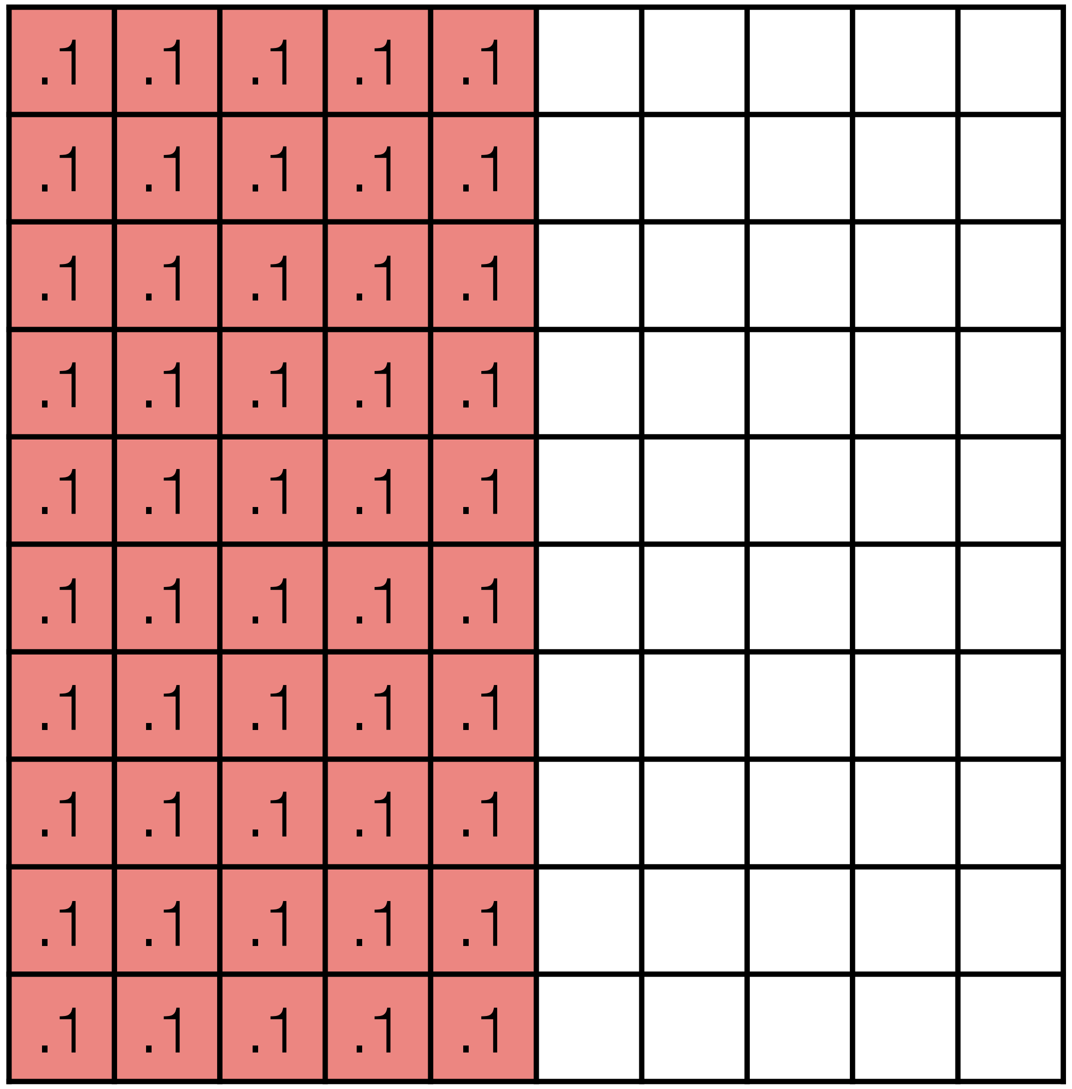


Buffer

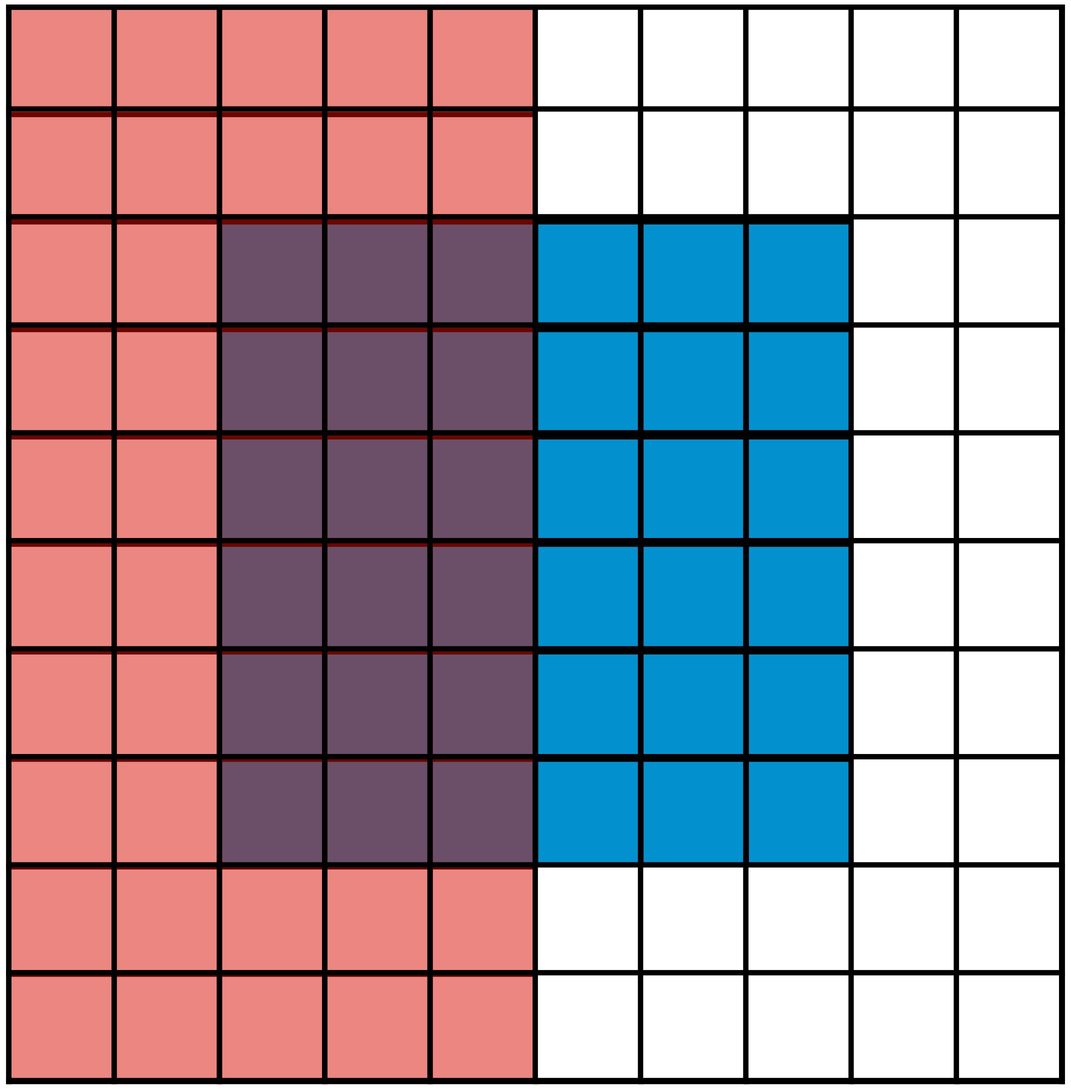


Back-to-front

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

Front-to-back

Front Polygon

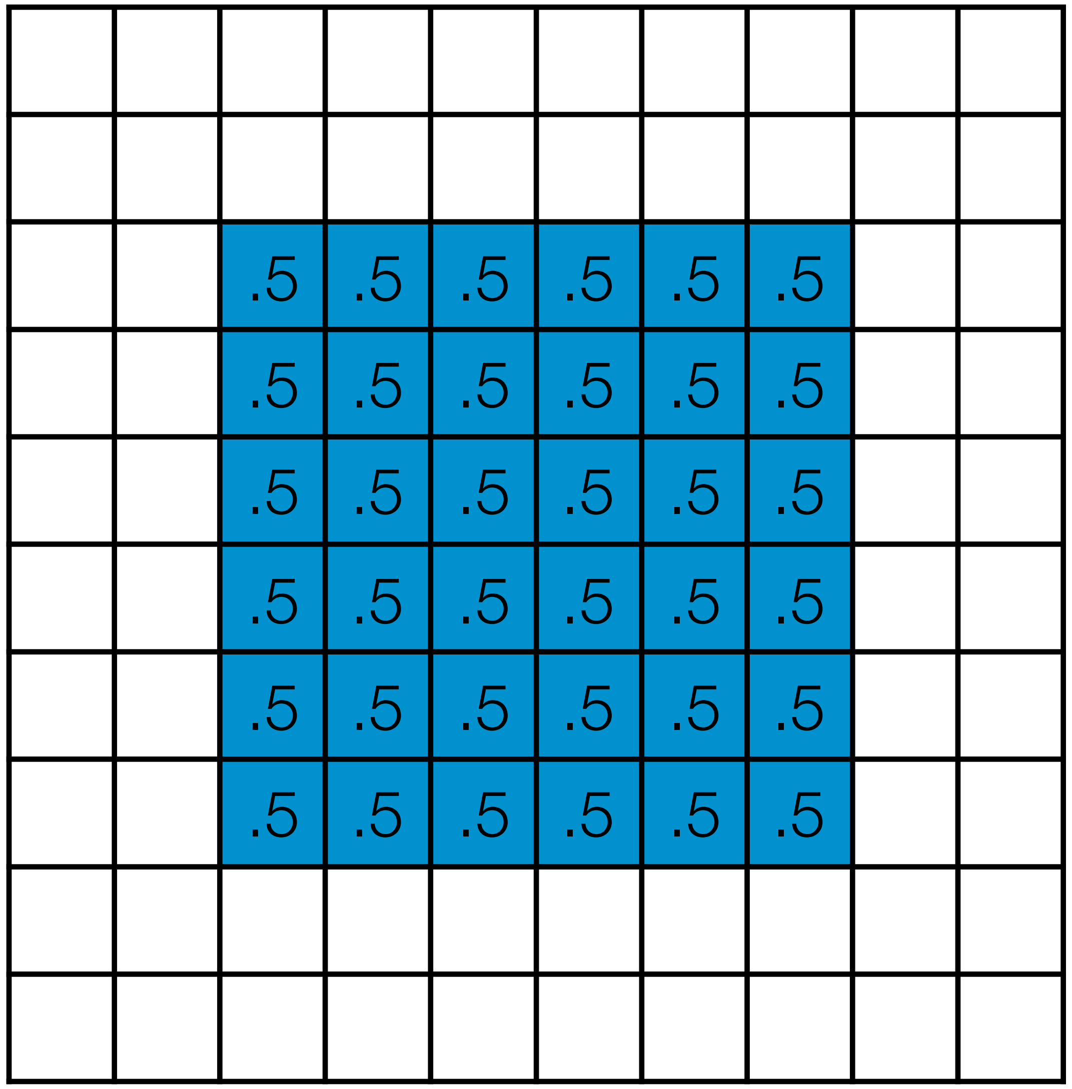
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					
.1	.1	.1	.1	.1					

Buffer

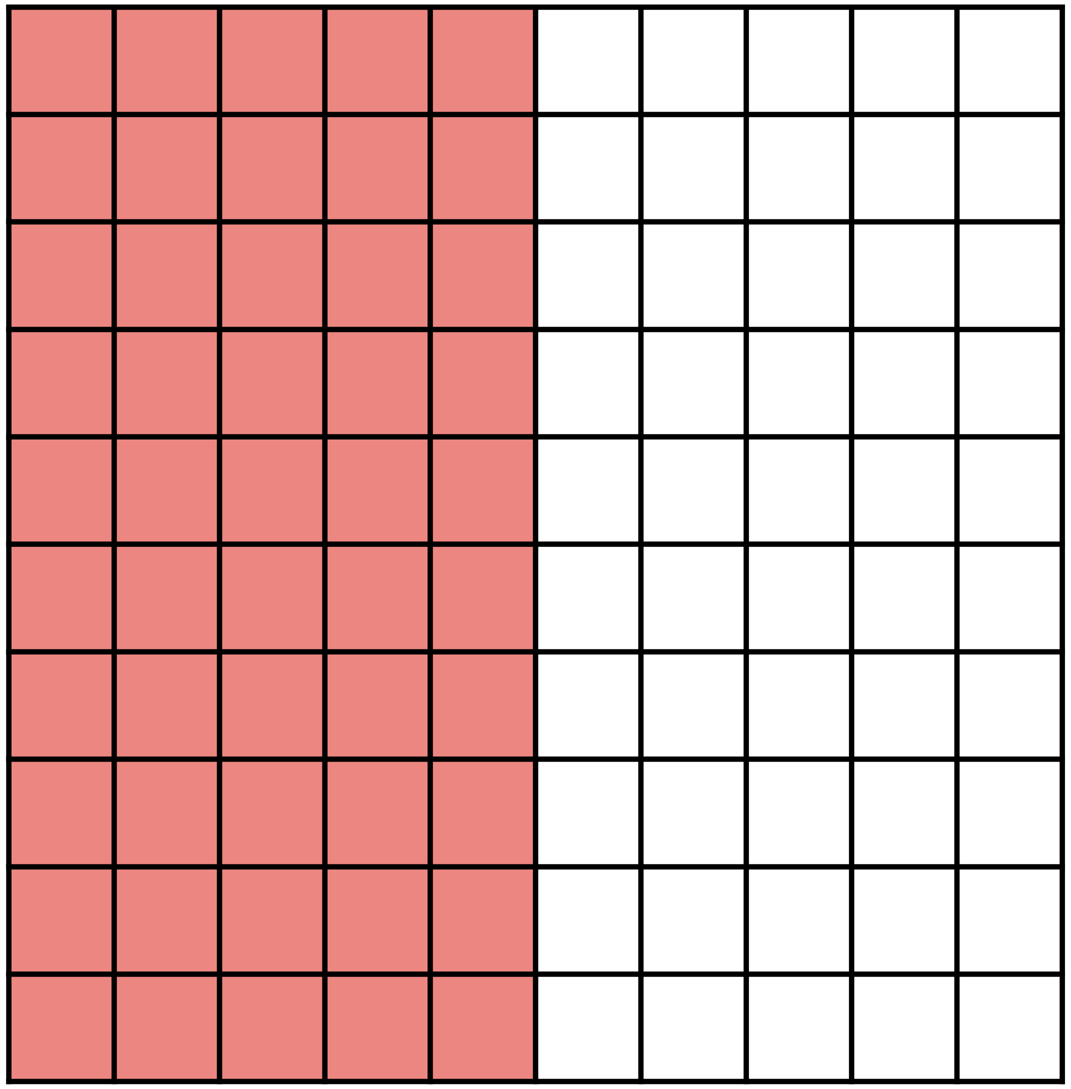
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					
🟥	🟥	🟥	🟥	🟥					

Front-to-back

Back Polygon

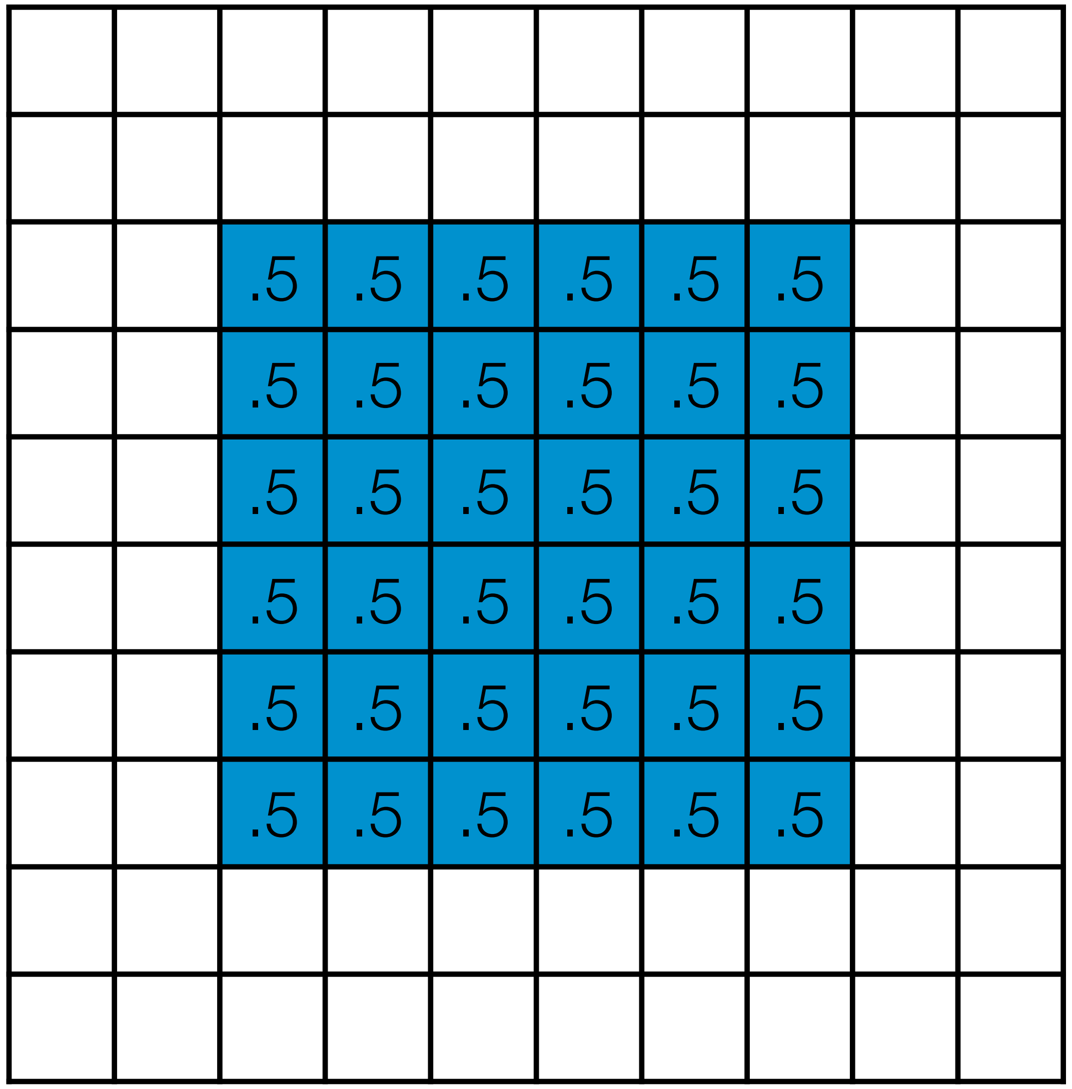


Buffer

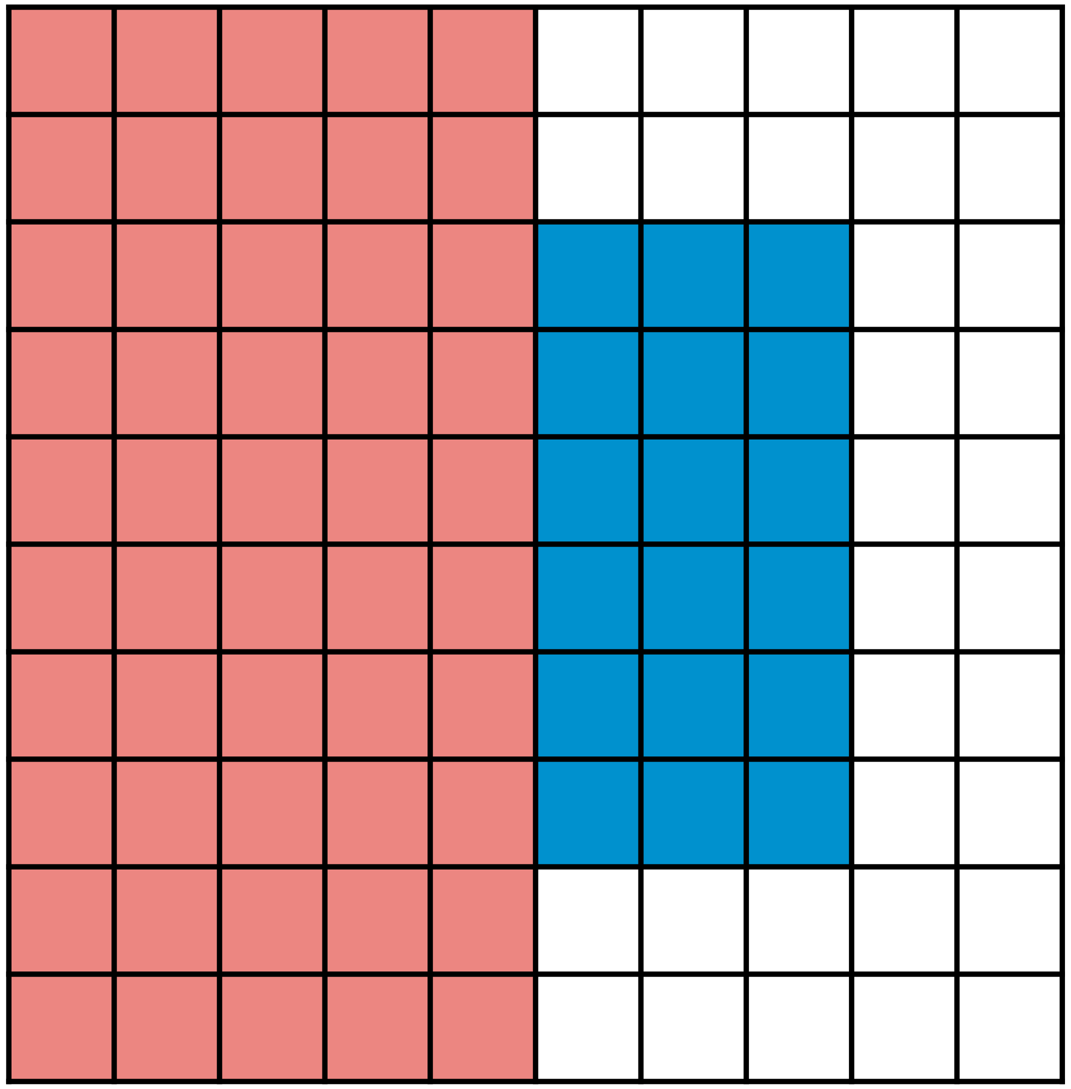


Front-to-back

Back Polygon



Buffer



Wrong

Transparency

- Transparent polygons should be drawn in back-to-front order after your opaque ones.
- Other fudges are to draw your transparent polygons after the opaque ones 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).
```