Week 6A Textures

Debugging Meshes in OpenGL

- Draw only the outline (i.e., without fill)

  ```
  glPolygonMode(GL.GL_FRONT_AND_BACK, GL3.GL_LINE);
  ```

- Turn off backface culling to see the full mesh

  ```
  gl.glDisable(GL.GL_CULL_FACE);
  ```

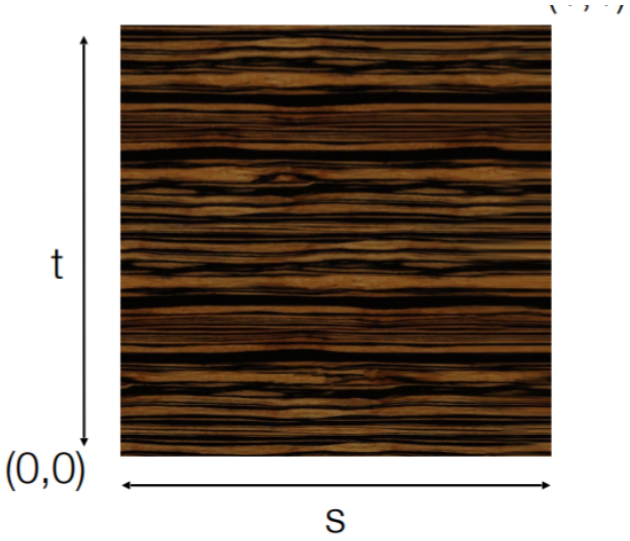
- Lighting problems might due to:
  - Normal calculation
  - vertex position in Shader

Texturing

- Texturing are a way to add detail to our models without requiring to many polygons
- Textures are used to add:
  - Color
  - Reflections
  - Shadows
  - Bumps
  - Lighting effects
  - etc
- A texture is basically a function that maps texture coordinates to pixel values.

$$T(s,t) = \begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix}$$

  - where the left-hand side starts with T is texture coordinates (i.e., texture value at (s,t)) while the right-hand side is the pixel value
- Texture coordinates are usually in ranger (0,1).
- Textures are most commonly represented by bitmaps (2D image files)



Procedural textures

- It is also possible to write code to compute the texture value at a point.
- This can be good to generate materials like marbe or woodgrain.

Using Textures in OpenGL

Three main procedures for using textures in OpenGL.

1. Loading or creating textures
2. Passing the texture to a shader
3. Mapping texture co-ordinates to vertices

Loading textures

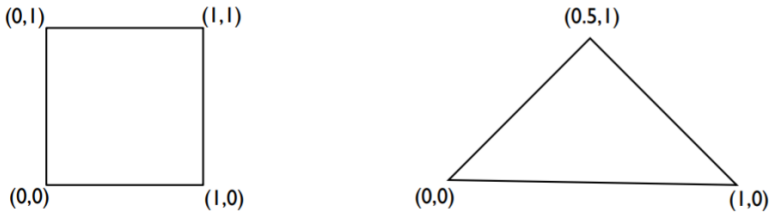
- Similar yo vertex buffers, we have to create buffers on the GPU and copy into them.

```
// Setting data to current texture
gl.glTexImage2D(
    GL.GL_TEXTURE_2D,
    0,  // level of detail: 0 = base
    data.getInternalFormat(),
    data.getWidth(),
    data.getHeight(),
    0,  // border (must be 0)
    data.getPixelFormat(),
    data.getPixelType(),
```

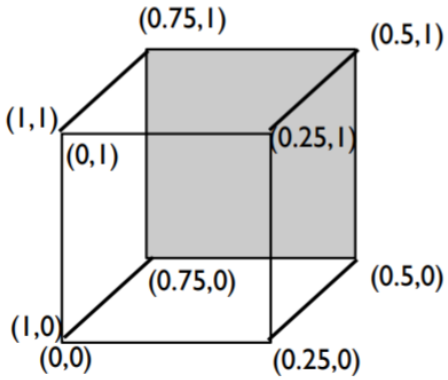
```
    data.getBuffer()
);
```

Texture mapping

- To add textures to surfaces in on our model, we set texture coordinates for each vertex.



- 2D:



- 3D:

Texture Wrap

- You can assign texture coordinates outside the ranger [0,1] and set the texture wrap to
- `GL.GL_REPEAT (default)`
- `GL.GL_MIRRORED_REPEAT`
- `GL.GL_CLAMP_TO_EDGE`
- `GL.GL_CLAMP_TO_BORDER (disable repeat effect)`
- ```
  //
  GL.GL_CLAMP_TO_BOARDER
  ```
  ```

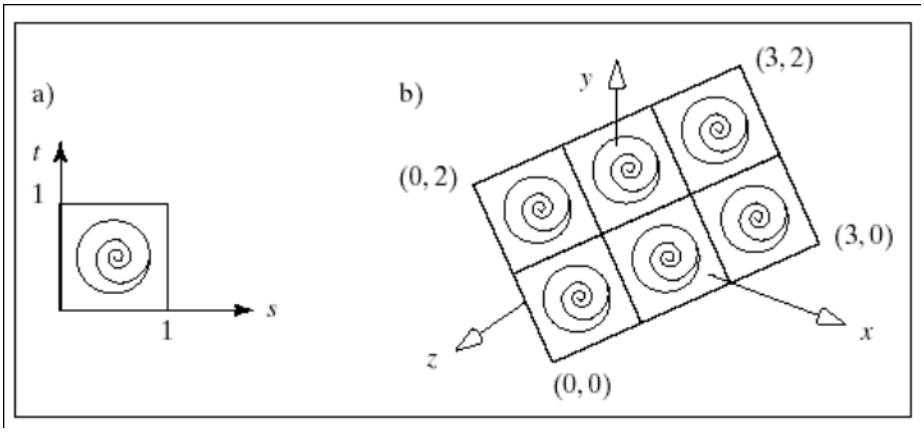
```
        // The boarder color is originally set to black by default.
        GL.GL_BOARDER_COLOR

// Mapping for multiple dimensions

// Repeat mapping in s dimension
gl.glTexParameteri(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT
);

// Repeat mapping in s dimension
gl.glTexParameteri(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_WRAP_T,
    GL.GL_REPEAT
)
```

Repeating a Texture If `GL_REPEAT` is set, the use of texture coordinates outside [0,1] are going to next to the initial texture, like



Binding the texture OpenGL supports **up to 32** simultaneously "active" textures, defined by constants `GL_TEXTURE0` up to `GL_TEXTURE31`. These values are distinct from a texture id

In the fragment shader, tex is defined as `uniform sampler2D tex;`, while passed by callee with

```
Shader.setInt(gl, "tex", 0);
gl.glActiveTexture(GL.GL_TEXTURE0);
gl.glBindTexture(GL.GL_TEXTURE2D, texId);
```

To be concluded by a pipeline: sampler2D in fragment shader –(assigned)–> Active texture number –(associated)–> Texture ID

Textures and shading

One **simple** approach is to wrap up the illumination models with texture enabled (i.e., replace illumination calculations with a texture look-up)

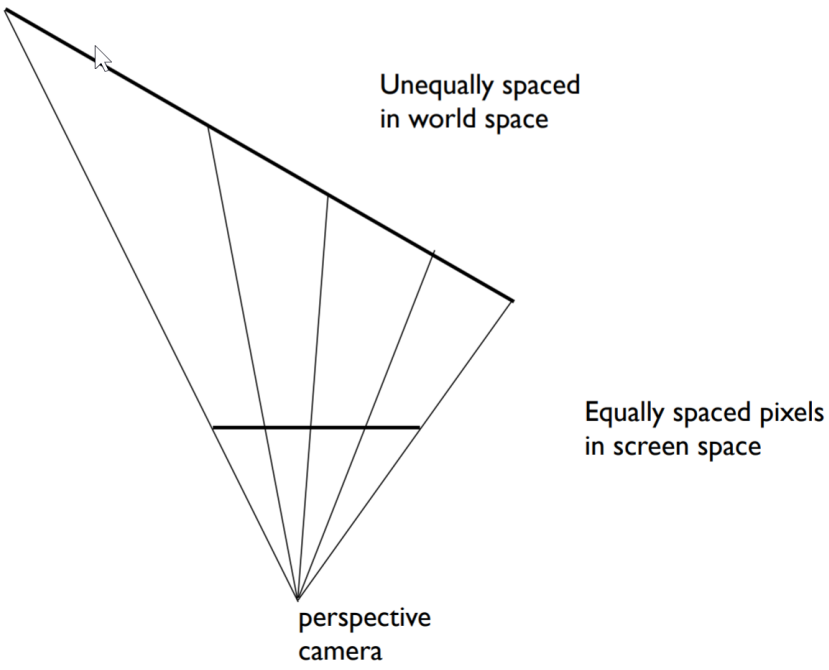
$$I(P) = T(s(P), t(P))$$

that produces objects which are not affected by lights or color.

The **common solution** is to use the texture to modulate the **ambient and diffuse** reflection coefficients:

$$I(P) = T(s,t) * [I_\alpha \rho_\alpha + I_d \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})] + I_s \rho_s (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f$$

Usually leave the specular term unaffected because it is unusual for the material color to affect specular reflections.



Foreshortening

Standard bilinear interpolation does not work because it fails to take into account ***foreshortening in tilted polygons***, as linear intepro-

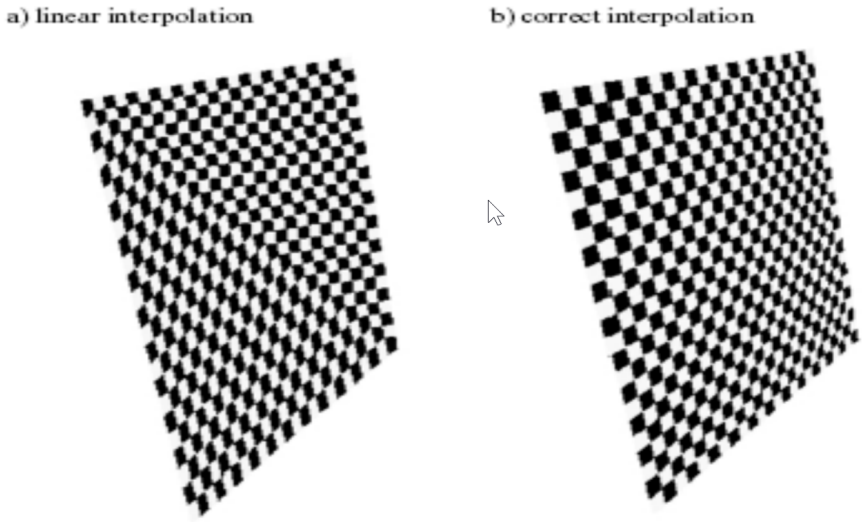
lation perserves the false integrity of original texture, demonstrated below.

Foreshortening is the visual effect or optical illusion that causes an object or distance to appear shorten than it actually is because it is angled toward the viewer.

Additionally, an object is often not scaled evenly: a circle often appears as an ellipse and a square can appear as a trapezoid.

Rendering the Texture

Linear vs. correct interpolation example:



Hyperbolic interpolation (NOT EXAMIBLE) While we want texture coordinates to interpolate linearly in world space, the perspective projection distorts the depth coordinate so that linear interpolation in *screen space* $\neq$ linear interpolation in *world space*. And hyperbolic interpolation does fix it.

More about Hyperbolic interpolation

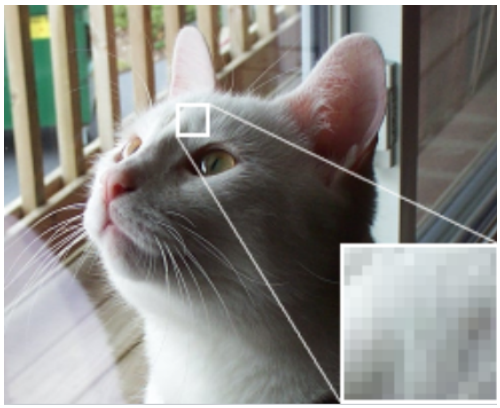
3D textures

3D textures are made by adding an extra texture coordiante, instead of simple repeat in each face. **This, of course, eliminates weird seams and distortions when a 2D texture is wrapper on a curve 3D surface.**

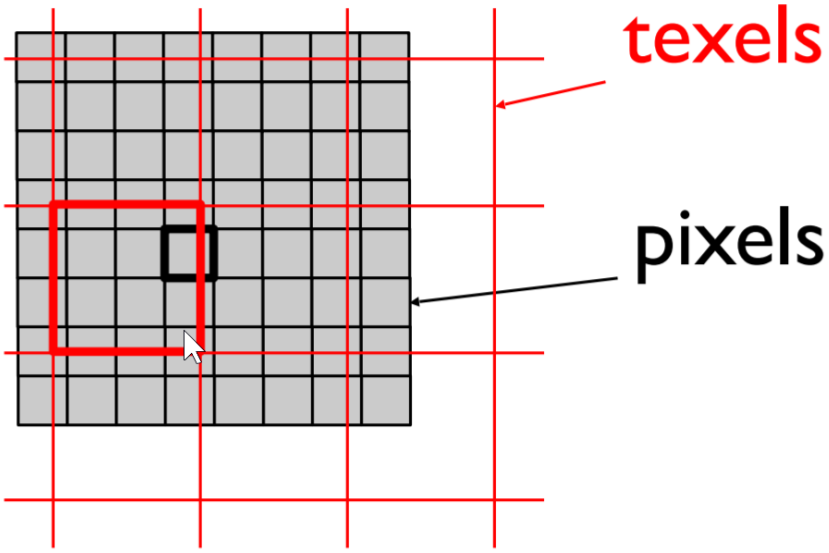
Magnification

Normal bitmap texture have finite detail (i.e., if we zoom in close we can see individual texture pixels). If the camera is close enough to a textured polygon multiple screen pixels may map to the same texture pixels, resulting in *pixelated* effects.

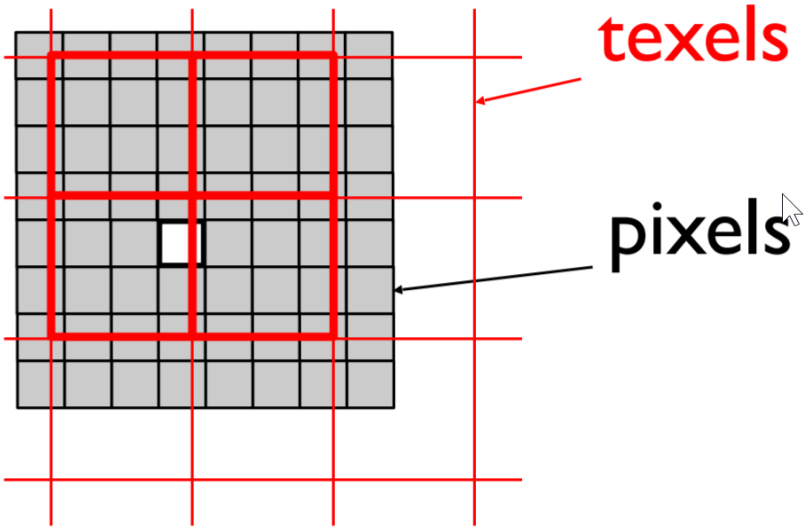
> Displaying a bitmap or a section of a bitmap at such a large size taht individual pixels, small singel-colored square display elements that comprise the bitmap, are visible. Such an image is said to be pixelated.
>
> Or in plain English, the iamge looks smooth when zoomed out, but when a small section is viewed more closely, the eye cam distinguished individual pixels.



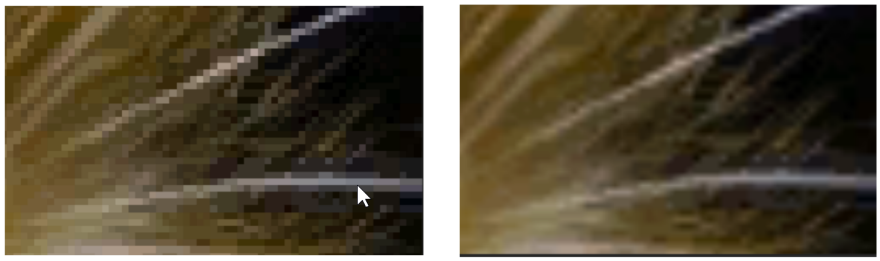
Solution 1 - Nearest Texel The alignment is probably not exact. The way to solve this is to find the nearest texel for that pixel.



Solution 2 - Bilinear Filtering The basic idea is to find the nearest four **texels** (not pixels) around the pixel and use bilinear interpoaltion over them.



No filtering vs. Filtering. Filtering clearly does the better job.



No filtering Filtering

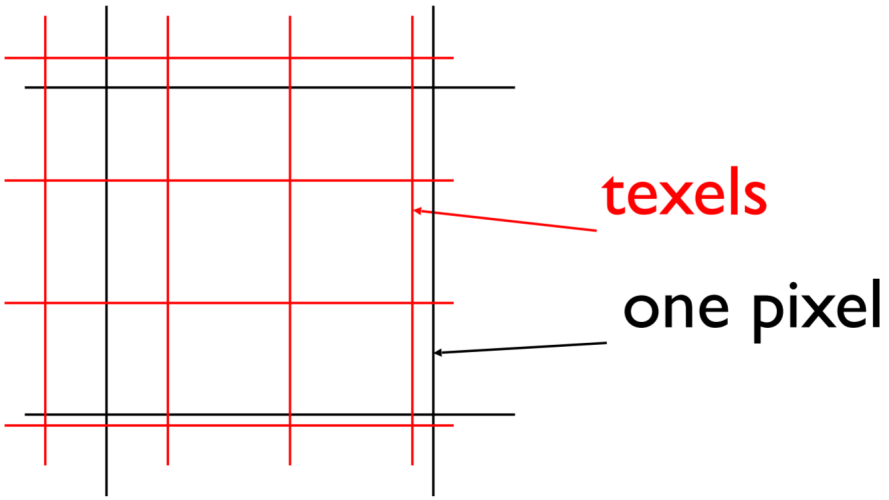
Magnification filtering in OpenGL

```
// Bilinear filtering - solution 2
gl.glTexParameteri(GL.GL_TEXTURE_2D,
        GL.GL_TEXTURE_MAG_FILTER,
        GL.GL_LINEAR);

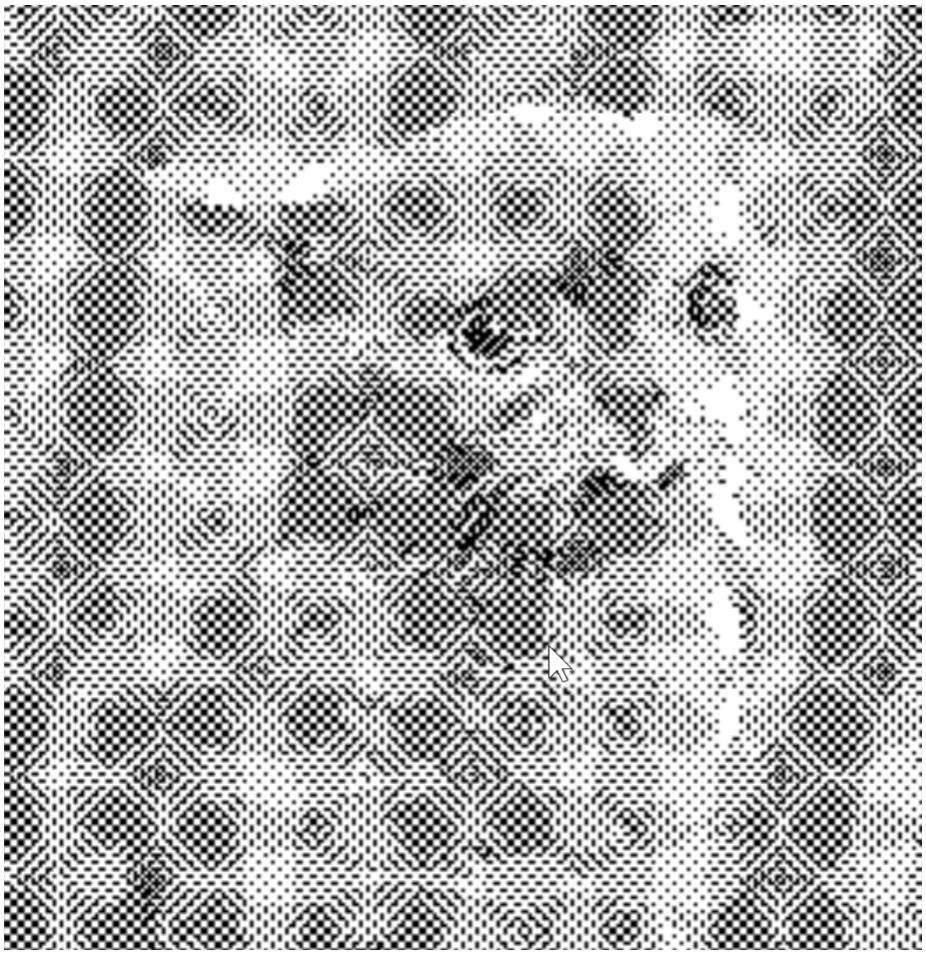
// No bilinear filtering - solution 1
gl.glTexPararmeteri(GL.GL_TEXTURE_2D,
        GL.GL_TEXTURE_MAG_FILTER,
        GL.GL_NEAREST);
```

Minification

Minificaition, on the other hand, occurs when we zoom out too far from a texture, where we can have more than one texel mapping to a pixel. If image pixels line up with regularities in the texture, strange artefacts appear in the ouput such as moire patterns or shimmering in an animation.



Compared to the **Magnification**, **Minification** indicates that there might be more than one texel mapping to a pixel, while **Maginication** indicates there might be one texel mapping to more than one pixel.

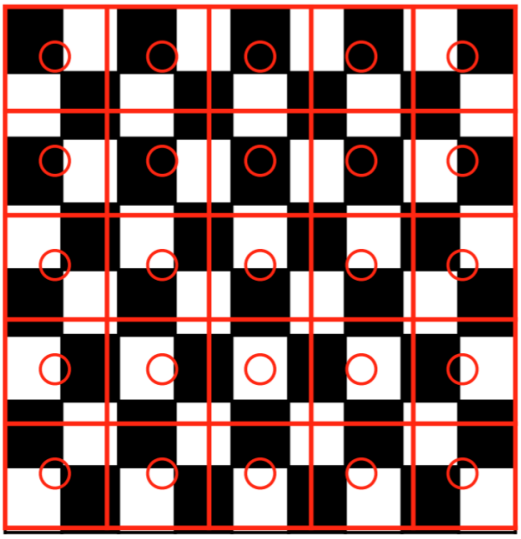


Aliasing - an example of minification

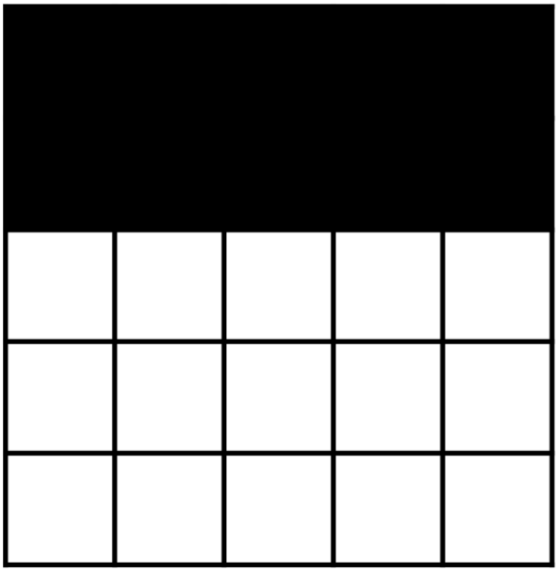
The problem is that one screen pixel overlaps multiple texels but taking its value from only one of those texels.

Aliasing occurs when samples are taken from an image at a lower resolution than repeating detail in the image. Aliasing is an effet that causes different signals to become indistinguishable when sampled.

samples



result



Example of Spatial aliasing in the form of a moire pattern

Solution Aliasing is generally avoided by applying *lowe pass filters* or *anti-aliasing* to the input signal before sampling. Suitable *reconstruction filters* should then be used when restoring the sampled signal to the continuous domain.

Filtering

The problem is that one screen pixel overlaps multiple texels but is taking its value from only one of those texels.

Solution A better appraoch is to **average the texels that contribute to that pixel. However, doing this on the fly is expensive**.

Minification Filtering

```
// Bilinear filtering
gl.glTexPararmeteri(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_LINEAR
);

// No bilineasr filtering
gl.glTexParameteri(
    GL.GL_TEXTURE_2D.
    GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_NEAREST
);
```

MIP mapping

Mipmaps are precomputed low-resolution versions of a texture. Starting with a slightly lower resolution texture we compute and store 256 × 256, 128, 64, 32, 16, 8, 4, 2, 1.

This takes total $\frac{4}{3}$ memory of the original.

Generating Mip-Maps

```
// Get opengl to auto-generate
// Mip-maps
gl.glGenerateMipmap(GL.GL_TEXTURE_2D);
```

Using mipmaps The simplest approach is to use the **NEXT SMALLEST MIPMAP** for the required resolution.

> To render a 40 × 40 pixel image, use the 32 × 32 pixel mipmap and magnify using magnification filter.
>
> **DO NOT forget to magification filter the texture of resulting Mipmap**

MipMap Minification Filtering

```
// Use nearest mipmap
gl.glTexParameteri(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_NEAREST_MIPMAP_NEAREST
);
```

Trilinear filtering A more costly approach is trilinear filtering by finding the two of the nearest MipMaps and do the linear interpolation on both of them.

1. Use bilinear filtering to compute pixel values based on the next highest and the next lowest mipmap resolutions
2. Interpolate between these values depending on the desired resolution.

MipMap Minification Filtering

```
// Use trilinear filtering
gl.glTexParameteri(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_LINEAR_MIPMAP_LINEAR
);
```

Aniso Filtering If a polygon is on an oblique angle away from the camera, then minification may occur much more strongly in one dimension than the other.

Anisotropic filtering is filtering which treats the two axes independently.

```
float fLargest[] = new float[1];
gl.glGetFloatv(GL.GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, fLargest, 0);

gl.glTexParameterf(
    GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MAX_ANISOTROPY_EXT,
    fLargest[0]
);
```

Aniso Filtering vs. Trilinear Filtering Trilinear interpolation takes a much bigger sample instead of what it should be (normally, smaller than what it takes). Hence, the far plane (i.e, far end point seeing on an oblique angle away from the camera) will be bluery.

RIP Mapping

RIP mapping is an extension of MIP mapping which down-samples **each axis** and is a better approach to anisotropic filtering

> A 256×256 iamge has copies at: $> 256 \times 128$, 256×64, $\ldots$ $> 128 \times 256$, 128×128, $\ldots$ $> 64 \times 256$, and etc (all permutations)

Limitations

- Does not handle diagonal anisotropy.
- More memory required for RIP maps (4 times as much)
- Not implemetned in OpenGL

Multi-texturing

Can use mroe than one texture on the same fragment.

```
gl.glActiveTexture(GL.GL_TEXTURE0);
gl.glBindTexture(GL.GL_TEXTURE_2D, texId1);

gl.glActiveTexture(GL.GL_TEXTURE1);
gl.glBindTexture(GL.GL_TEXTURE_2D, texId2);
```

However, the above code (multi-texturing) have a pre-condition that has to pass two different textures to the shader, and hence two different sets of texture coordinates.

Animiated textures

Animated textures can be achieved by loading multiple textures and using a different one on each frame.

Week 6B Textures 2 and Rasterisation

Rendering to a texture

Idea of how a security camera implement

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.

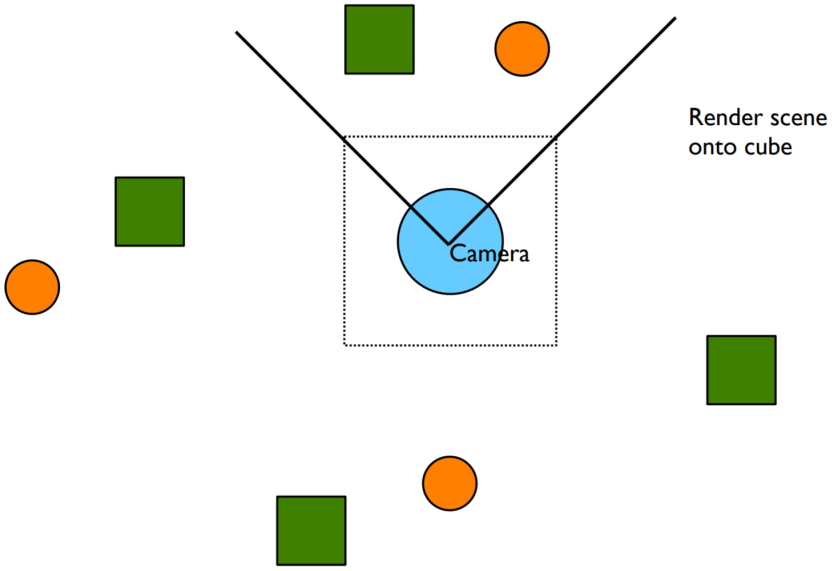
E.g. Implementation a security camera in a game

```
gl.glCopyTexImage2D(...);
```

Reflection Mapping

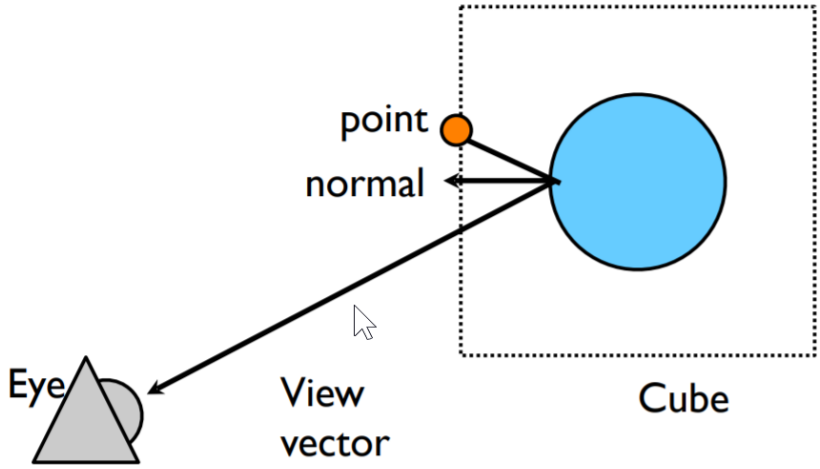
Doing this in general is expensive, but we can make a reasonable approximation with textures

1. Generate a cube that encloses the reflective object.
2. Place a camera at the centre of the cube and render the outside world onto the faces of the cube.
3. Use this image to texture the object.



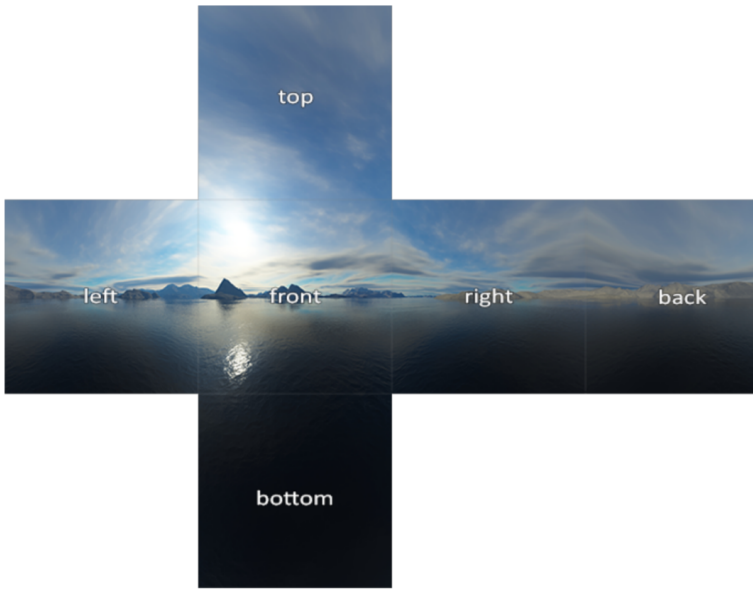
Do the mapping for the rest three faces for 2 dimensional application (supposing fovy is 90 degree here). However, if it is a 3 dimensional application, another two surfaces (top and bottom) also need to be taken into consideration (such as front, back, left, right, up and down).

To apply the reflection-mapped texture to the object we need to calculate appropriate texture coordinates. This can be done by tracing a ray from the camera, reflecting it off the object and then calculating where it intersects the cube.



Pros and Cons

- Pros: Produces reasonably convincing polished metal surfaces and mirrors.
- Cons:
  - Expensive: Requires 6 additional render passes (i.e., six faces in total) per object
  - Angles to near objects are wrong.
  - Does not handle self-reflections or recursive reflections



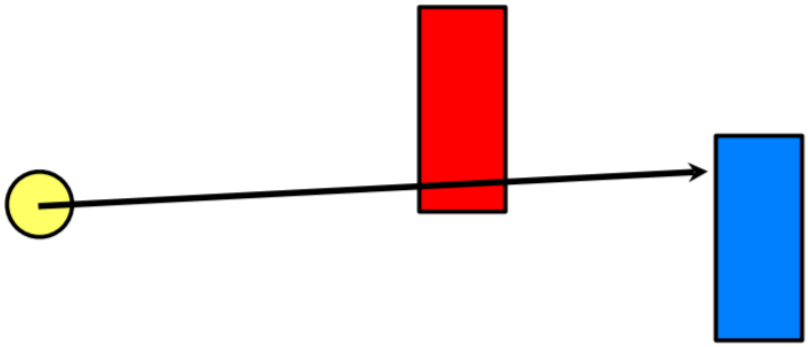
Examples for mapping sky boxes

OpenGL does also support sphere mapping, which produces more distortion and is not as effetive as cube mapping.

Shadows

Shadow mapping is relatively fast and can usually be done in real time. It requires two extra rendering passes per light source, one to compute the shadow buffer and one to apply the shadows to the scene. The quality of the shadows is limited by the resolution of the shadow buffer. A higher resolution buffer gives better looking shadows but is more costly in terms of time and memory. Shadow edges are hard and appear jagged at low buffer resolutions.

Out lighting model does not currently produce shadows, which requires taking whether the light source is occluded by another object into account.



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 teh cloest object in each direction.

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.

When rendering a point P:

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

Similar to the depth buffer, where it does require rendering from both of light's perspective and camera's perspective.

Pros and Cons

- Pro:
  - Provides realistic shadows
  - No knowledge or processing of the scene geometry is required
- Cons:
  - More computation
  - Shadow quality is limited by precision of shadow buffer. This may cause some aliasing artefacts, such as, perspective aliasing and projective aliasing. Referece
  - Shadow edges are hard. Nowadays, one main solution to solve this is when rendering, turning the reolsution really high.

  - 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.

Reading material

Light Mapping

Light mapping is wide used for the scenario when geometry models are static at low computational cost.

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. Also called *baked lighting*.

Pros and Cons

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

What normal mapping does is to render the fake lights for bumps and dents.

We made assumption that the surface of the polygon is smoothly curved when interpolating normals in a Phong Shader.

> What if the surface is actually rough with many small deformities?

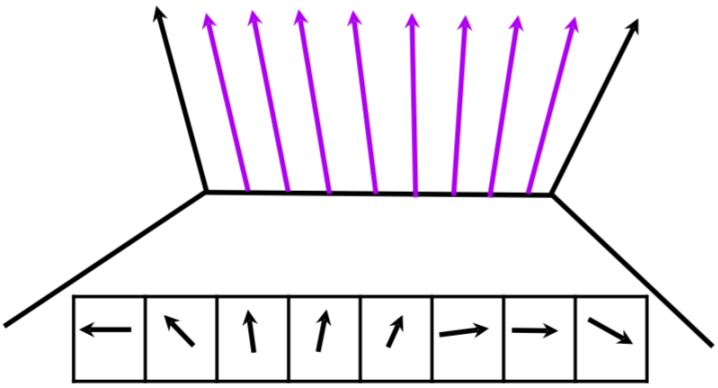
Optional solution One solution would be to increase the number of polygons to represent all the deformities, but this is computationally unfeasible for most applications.

Another solution from the Lighting Instead we use textures called normal maps to simulate minor perturbations in the surface normal.

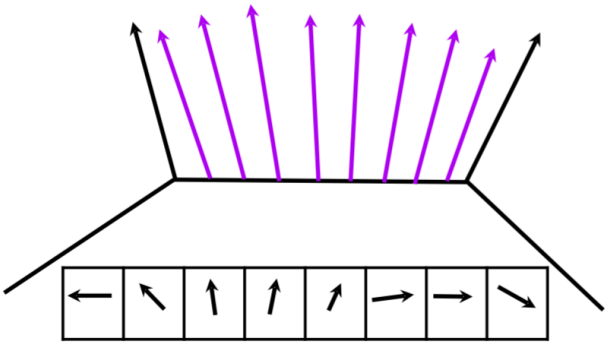
> In 3D computer graphics, normal mapping is a technique used for fakign the lighting of bumps and dents - an implementation of bump mapping. It is used to add details without using more polygons.

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

However, this does not reflect the reality of the shape of the meshes, but only the lighting effects. So, if we really approach the surface close (move across the z-axis), we won't seee any details for those deformities.



Normal map



Perturbed normals

Pros and Cons

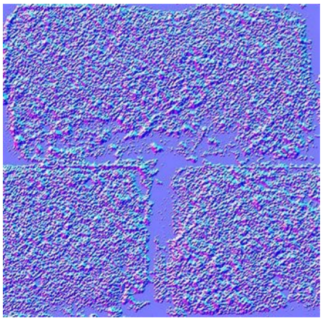
- Pros: Provide the illusion of surface texture
- Cons:
  - Does not affect silhouette
  - Does not affect occlusion calculation

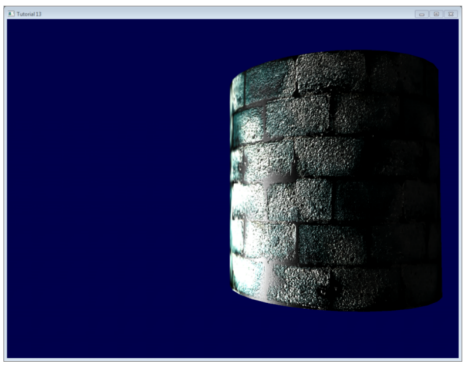
Advantages and disadvantages of normal mapping over adding extra polygons to represent detail

- Advantages: Every polygon we add to the scene creates extra computation at every step of the pipeline: transforming, illumination, clipping,

rasterising, etc. Having a large number of polygons is therefore very computationally expensive. Normal mapping allows us to represent rough surfaces with far fewer polygons.

- Costs: Extra work has to be done in the textureing stage. Normals must be computed for each pixel on the surface and the illumination equation recalculated to include data from the normal map. The map itself must be stored in memory, doubling the amount of texture memory required.
- Disadvantages: One problem with using normal maps is that they do not affect the outline of the object. So if the surface is viewed on an angle, it will appear flat. Also normal mapping only works for minor perturbances in the surface. Larger bumps should occlude other parts of the surface when viewed at an angle. Normal maps do not support this occlusion.





Example

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 needs to be accurate and efficient. Therefore, simple integer calculations are preferred (floating points cost computationally and create round errors).

Problems with drawing lines in Slope-intercept Form

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

Even incremental (replaces multiplication) is still not good enough.

Bresenham's algorithm

Assumption The line is in the first octant (i.e., x1 > x0, y1 > y0 and m $\leq$ 1, also means the y interception is zero)

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.

Testing above/below:

$$F(x,y) = 2h(x-x_0) - 2w(y-y_0)$$
$$F(x,y) < 0 \implies (x,y) \text{ is above line}$$
$$F(x,y) > 0 \implies (x,y) \text{ is below line}$$

How does the $F(x,y) = 2h(x-x_0) - 2w(y-y_0)$ come from?

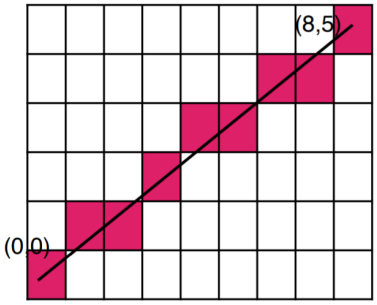
To remove the floating number out of the original equation, which is $0 = m(x-x_0) - (y-y_0) > m$ is a factionaly number (according to the definition of the gradient, $m = \frac{h}{w}) > y$ is going to be some integer plus 1/2 (midpoints).

Hence, rearrange the equation with replacing m with h/w and multiply 2 to it.

Pseudocode:

```
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++;
    }
}
```



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

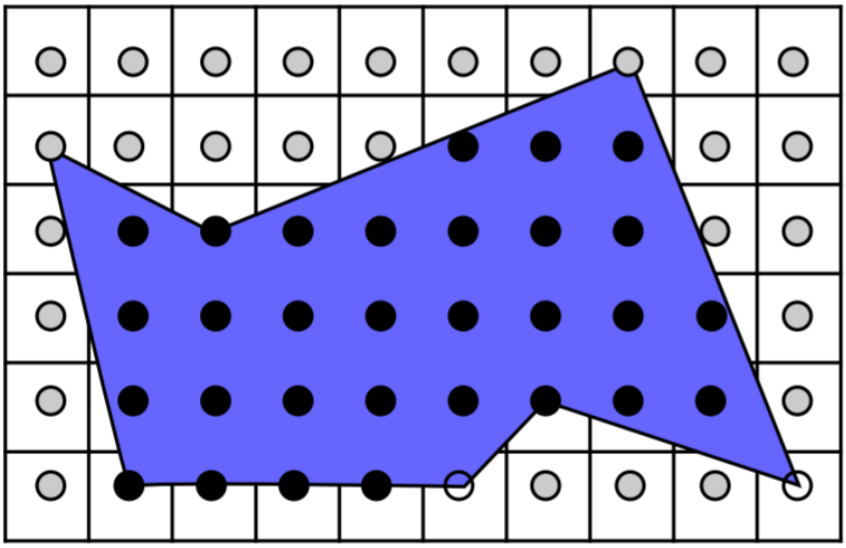
Example

1. The first pixel drawn at (0,0) because
   - The first pixel to be drawn anyway
   - $F = 2 * h - w = 2 * 5 - 8 = 2 \geq 0$
2. The second pixel drawn at (1,1) because
   - $F = 2 \geq 0$ previously. Hence, `y++`, resulting y increasing to 1 from 0, where x increasing by 1 anyway.
3. And etc. . .

Relxing restrictions Lines in the other quadrants can be drawn by symmertical versions of the algorithm. **Careaful 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

1. Determining which pixels are inside a polygon is a matter of applying the edge-crossing test (week3.md) for each possible pixel.

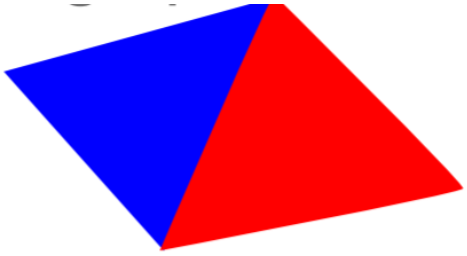


Tracing a ray horizontally crossing a polygon, even interceptions indicating outside, while odd interceptions indicating inside, on the contrary.

Such a solution does involve some of the problems. > Shared edges: If the pixel is on the edge of a polygon, whether should we draw it? What if two polygons are adjacent? > > Performance: Do we have to do the dege crossing test for every pixel?

Shared edges Pixels on shared edges between polygons need to be drawn consistiently regardless of the order the polygons are drawn, with no gaps.

To simplify and hendce form 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

And hence, we are introduced with a efficient algorithm to solving the problem, where fills the polygon.

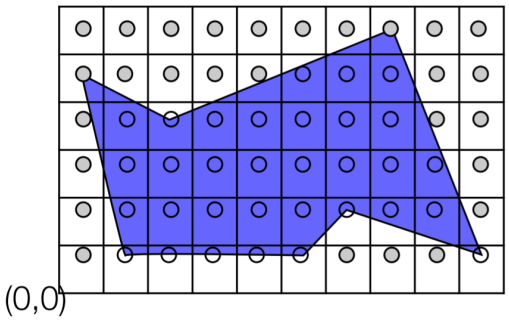
As testing every pixel is very inefficient, whereas that checking where the result *changes value* (i.e., when crossing an edge) is crucial. In addition, we proceed row by row with:

1. Calculating intersections incrementally
2. Sorting by x value
3. Filling runs of pixels between intersections

forming a Edge Table.

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 x-value of the lower vertex, the increment (inverse gradient) of the edge and the y-value of the upper vertex.



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

Example

Active Edge List (AEL) We keep a list of active edges that overlap the current scanline. Edges are added to the list as we pass the bottom vertex, 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

Pseudocode

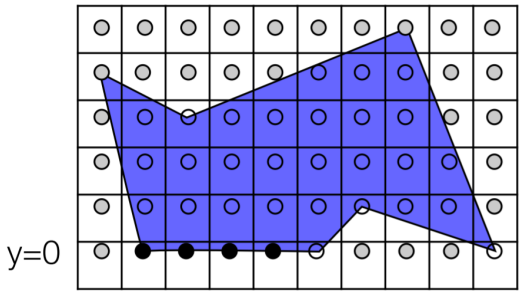
```
// For every scanline
for (y = minY; y <= maxY; y++) {

    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

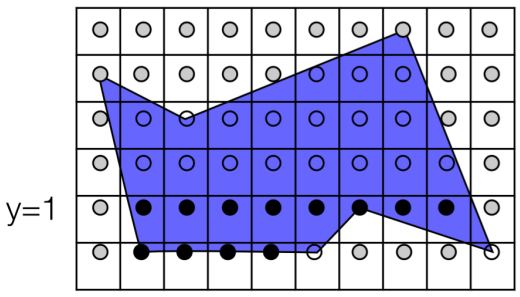
    for (int i = 0; i < active.size; i += 2) {
        fillPixels(actiev[i].x, active[i+1].x, y);
    }
}
```



Active edge list

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

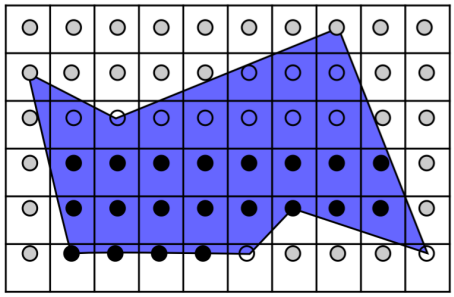
Example



Active edge list

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

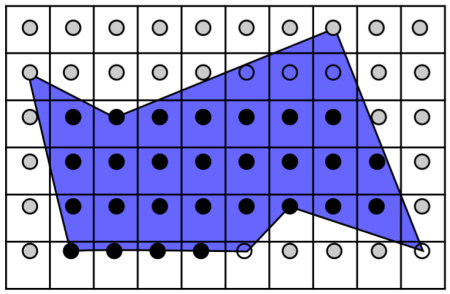
y=2



Active edge list

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

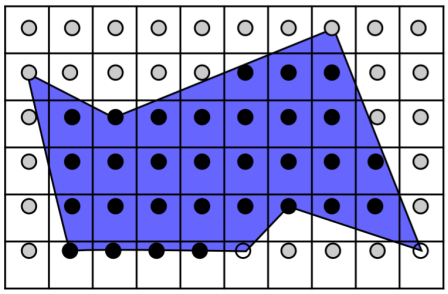
y=3



Active edge list

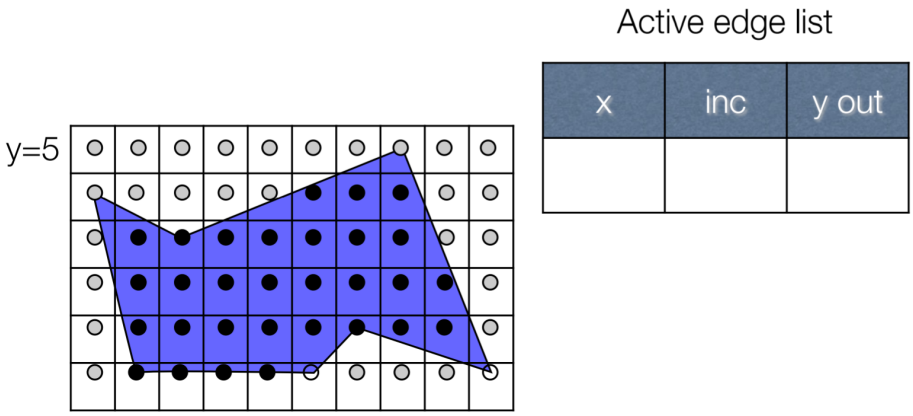
x	inc	y out
0.25	-0.25	4
2	-2	4
2	2.5	5
7.8	-0.4	5

y=4



Active edge list

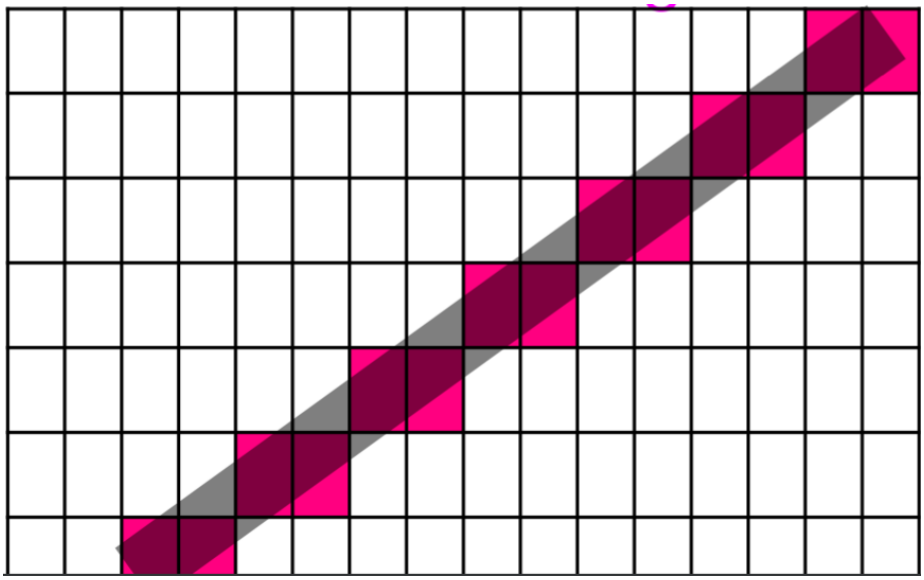
x	inc	y out
4.5	2.5	5
7.4	-0.4	5



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. I.e.,

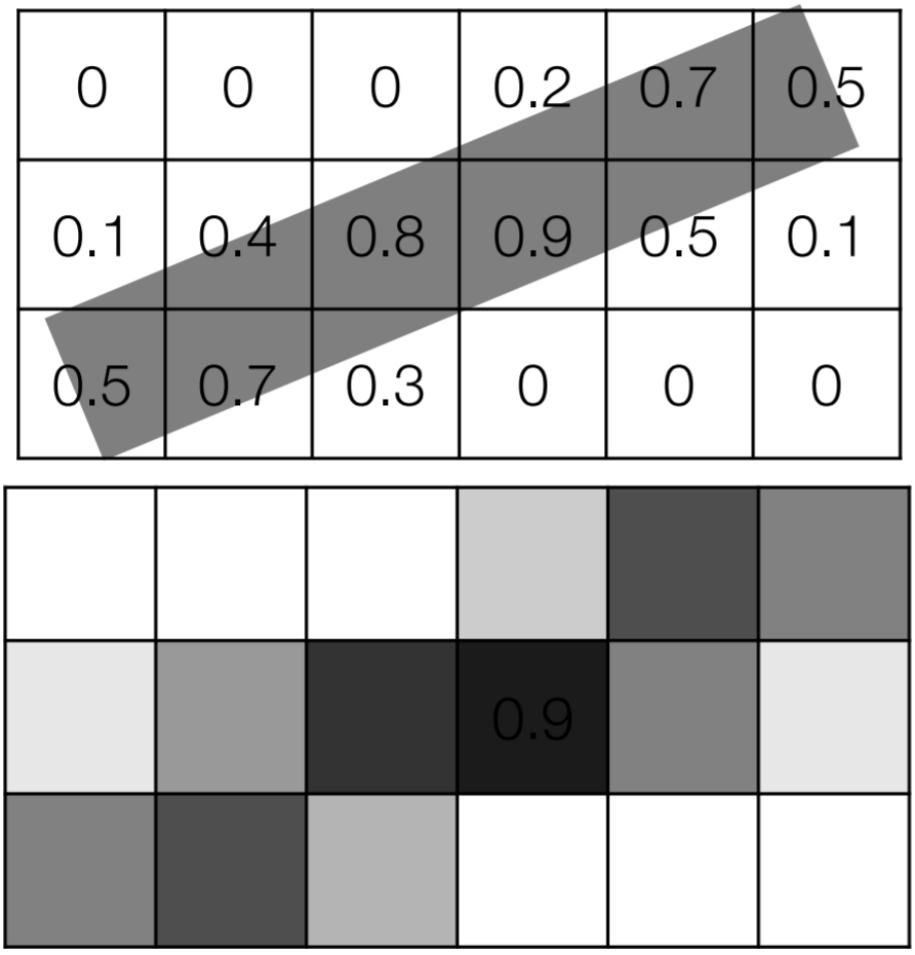


Antialiasing There are two basic approaches to eliminiate aliasing (AKA. antialiasing).

- Prefiltering: computing exact pixel values geometrically rather than by sampling.
  - Prefiltering is most accurate but requires more computation.

- Postfiltering: taking samples at a higher resolution (supersampling) and then averaging.
  - Postfiltering can be faster. Accuracy depends on how many samples are taken per pixel. More samples means large memory usage.

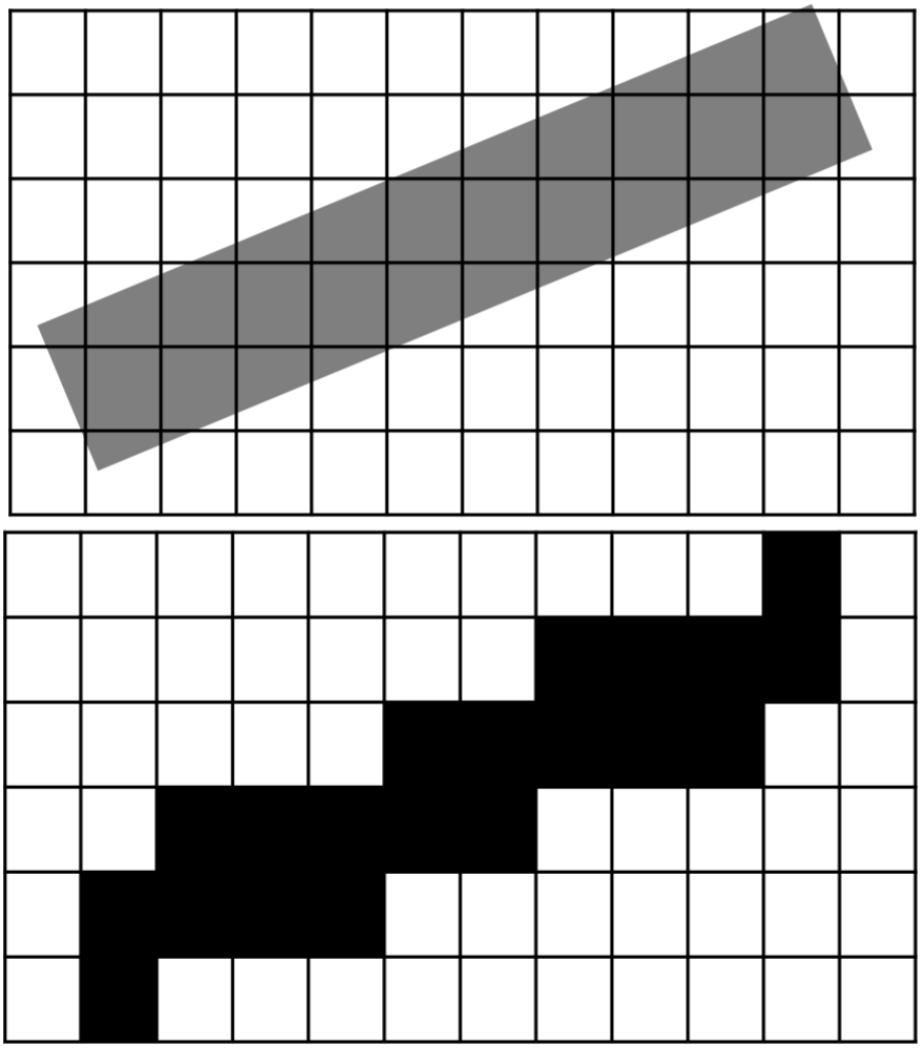
Prefiltering For each pixel, computing the amount occupied and set pixel value to that precentage (i.e., how much of a pixel's area is covered by a object). And based on that precentage dimming or brightening the pixel.



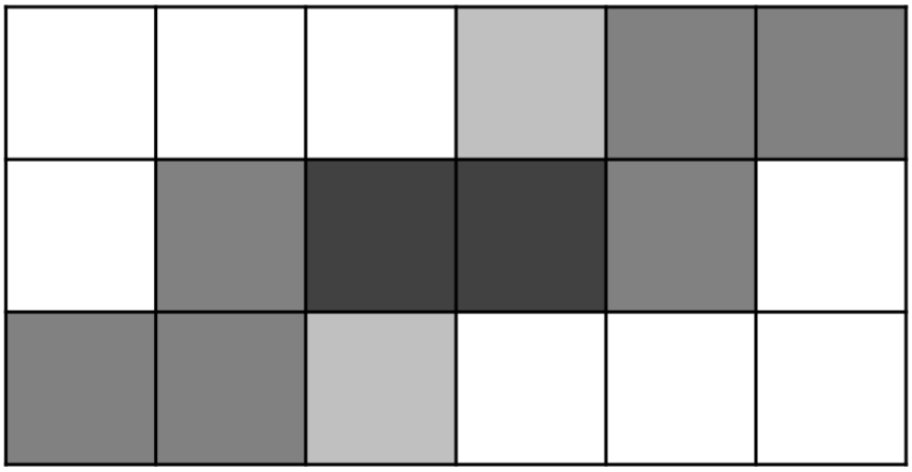
Postfiltering Draw the line at higher resolution and average (supersampling). In terms of postfiltering or supersampling, it is the process by which aliasing effects in graphics are reduced by increasing the frequency of the sampling grid and then average the results down.

This process means calculating a virtual image at a higher spatial resolution

than the frame store resolution and then averaging down to the final resolution.

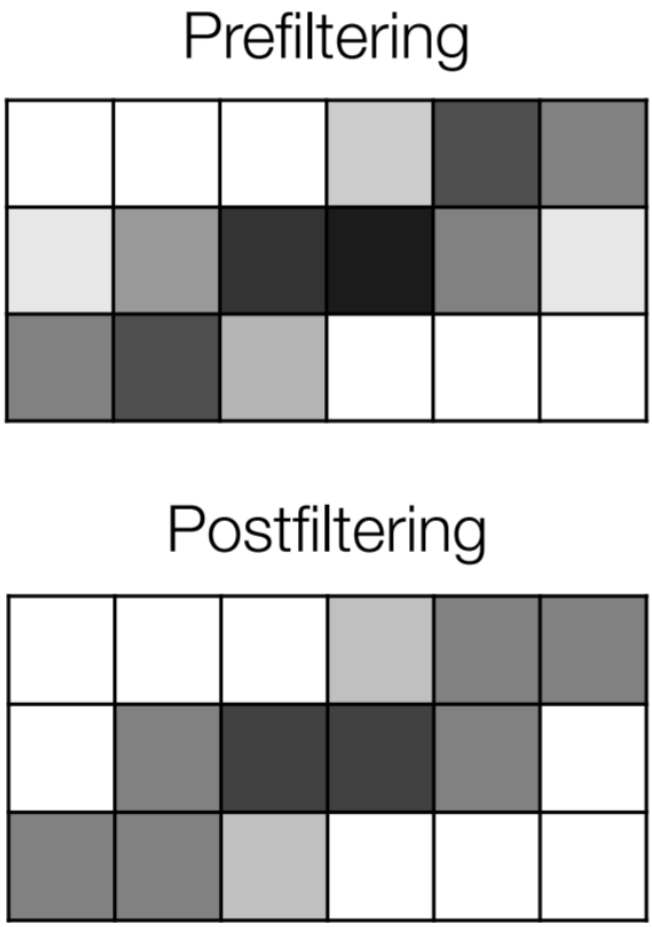


And final result,



Weighted postfiltering It is common to apply weights to the samples to favour values in the centre of the pixel.

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

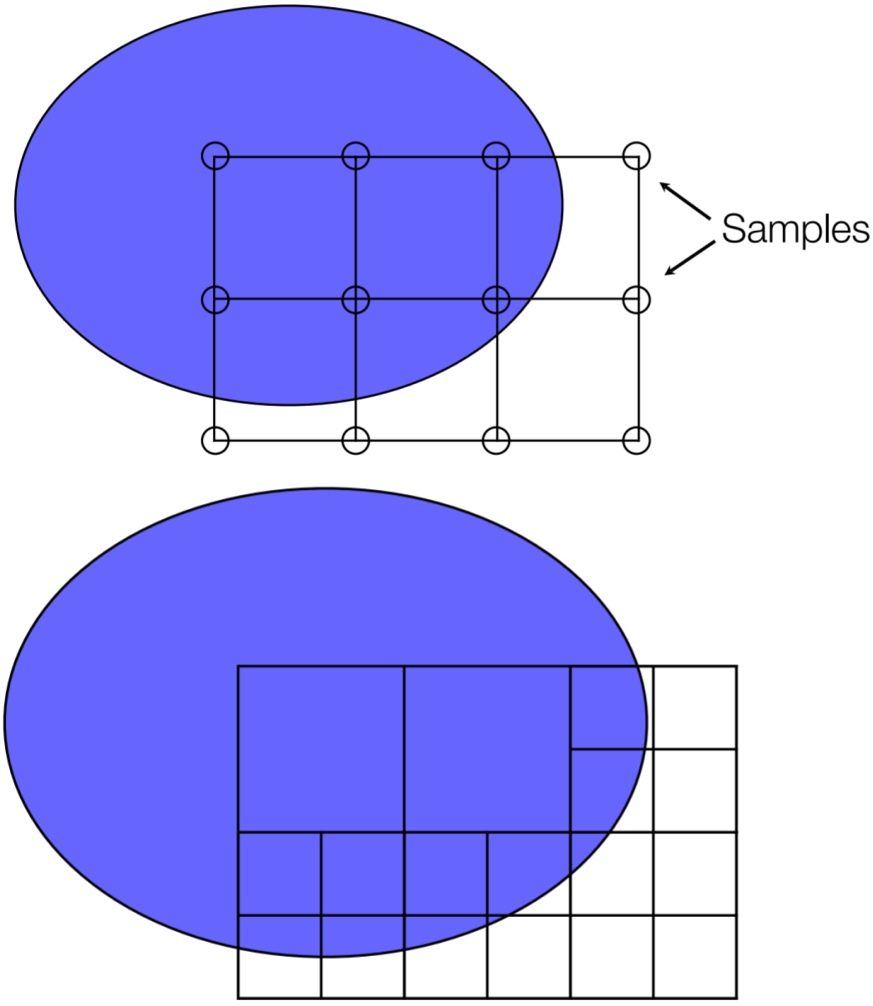


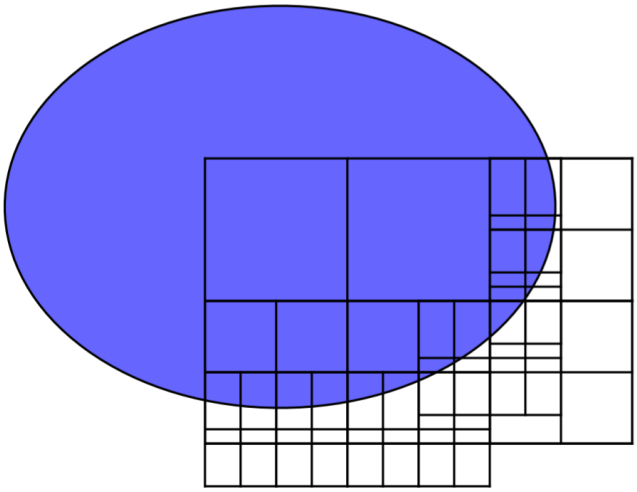
Prefiltering vs. Postfiltering

Stochastic sampling The problem for supersampling in a grid is that still tends to produce noticeably regular aliasing effects. Adding small amounts of *jitter* to the sampled points makes aliasing effects appear as visual noise.

Double Sampling Divide one pixel to four sub-pixels, and do the sampling on the nine according vertices.

Adpative Sampling Supersampling in large areas of uniform color is wasterful, while it is most useful in areas of major color change. One solution to it is to sample recursively, at finer levels of detail in areas with more color variance.


Samples



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

In 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);

// Full-screen multi-sampling
GLCapabilities capabilities = new GLCapabilities();
// Has maximum number of samples
capabilities.setNumSamples(4);
capabilities.setSampleBuffers(true);
...
gl.glEnable(GL.GL_MULTISAMPLE);
```

Exercise

1. Fix the error in the vertex shader:

   ```
   // The following two lines are wrong as they are supposed to be output instead of input
   in vec4 viewPosition;
   ```

```
in vec3 m;

// Hence, modified to
out vec4 viewPosition;
out vec3 m;

void main()
{
    viewPosition = view_matrix * globalPosition;
    // The following line is wrong as mentioned earlier, gl_Position is the CVV coordi
    gl_Position = viewPosition;

    // Hence, applying projection matrix here
    gl_Position = projection_matrix * viewPosition;
}
```

2. Fix the error in the fragment shader:

```
uniform vec3 lightIntensity;
uniform vec3 ambientIntensity;

uniform vec3 diffuseCoeff;
uniform vec3 ambientCoeff;

void main()
{
    // The below two lines are wrong
    float ambient = ambientIntensity * ambientCoeff;
    float diffuse = diffuseIntensity * diffuseCoeff;

    // The RGB-value for the light equation is calculated point-wise
    // Hence the correct version would be
    vec3 ambient = ambientIntensity * ambientCoeff;
    vec3 diffuse = diffuseIntensity * diffuseCoeff;
}
```

3. What is a reason to use texture mapping rather than lots of little polygons? Are the two representations functionally equivalent? What are the differences?

    - Textures are flat and do not respond to the lighting (unless there are normal mapping)
    - Polygons can interact with the physical world, whereas the textures are just flat

    Textures can be used to give the illusion of complex geometry while keeping the actual geometric complexity low. Rather than using textures, we could just add lots of polygons to the figure to model the extra details, but this

would slow down drawing, and it would be a lot of work to figure out the extra points and faces that we want to add.

They are not functionally equivalent. Textures can be used to simulate small feature that are not actually there in the geometry so will not respond to lighting or in the same way. For example, if we shine light nearly parallel to given face on the glof ball, one side of the dimple should be light and the other should be dark, but this won't happend if we'are using textures. Textures also have resolution and aliasing issues.

4. What is the difference between the following two textures filtering settings?

   These are 2 different settings for maginification. If we are in a situation where one texel gets mapped to many pixels (like zooming in or enlarging a photo too much) we have 2 choices of filters.

   `gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);` just chooses the cloest the texture coordinate value ouptut by the rasteriser. This is known as point sampling most subject to visible aliasing.

   `gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR);` is a filter that for 2D textures performs bilinear interpolation across the 4 nearest texels (2 texels in S direction and 2 texels in T direction) to create a smoother (sometimes blurrier) image. This approach is more expensive to compute.

5. What are MipMaps? Why are they used?

   They are pre-calculated, sequences of textures, each of which is a pregressively lower resolution representation of the same image. The height and width of each image, or level, in the mipmap is a power of two smaller than the previous level where each texel in the next level is calculated by averaging 4 of the parent texels.

   These can help with nimification aliasing problems which can arise when more than one texel is mapped to one pixel (like zooming out). Without mip-mapping you can use similar fiters for minification as for magnification - `GL_NEAREST` and `GL_LINEAR`. With mip mappoing you can use `GL_NEAREST_MIPMAP_NEAREST` which returns the nearest mip map or GL_LINEAR_MIPMAP_LINEAR which is trilinear filtering where bilinear filtering is used on 2 of the nearest mipmaps and then interpolated.

6. What does the anisotropic filtering do?

   ```
   float flargest[] = new float[];
   gl.glGetFloatv(GL.GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, fLargest[0]);
   gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAX_ANISOTROPY_EXT, fLargest[0]);
   ```

   Turning on anisotropic filtering. Like trilinear filtering, anisotropic filtering is used to eliminate aliasing effects, but improves on trilinear filtering by

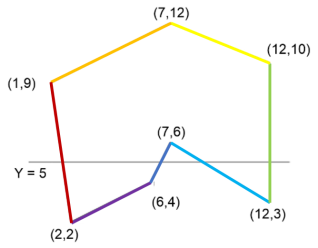
reducing blur and preserving detail at extreme viewing angles. Anisotropic can take a different number of samples in the horizontal vs. vertical directions depending on the projected shape of the texel. Where as isotropic (trilinear/mipmapping) always filters the same in each direction.

Different degrees or ratios of anisotropic filtering can be applied udring rendering and current hardware rendering implementations set an upper bound on this ratio. This code finds out the maximum level of filtering for the current implementation and sets the filter to this max level.

For best results, combine anisotropic filtering with a mipmap minification filter.

7. Scanline Algorithm

What does the Active Edge List (AEL) look like just before filling the pixels on scan line 5?



What pixels will be filled?

$y = 5$ $\quad 1\frac{4}{7} \leq pixel_x < 6\frac{1}{2}$ $\quad x = 2,3,4,5,6$

$\quad 8\frac{1}{2} \leq pixel_x < 12$ $\quad x = 9,10,11$

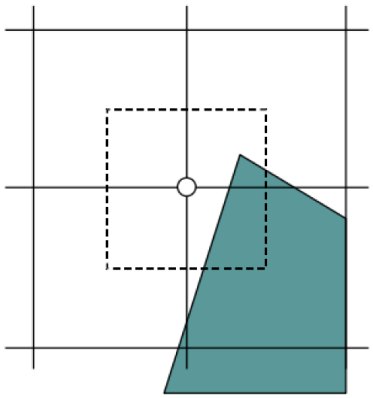
Edge list $\qquad x = x + (y - y_{in}) * inc$

y in	x	inc	y out	
2	2	-1/7	9	$x = 2 + 3 * -\frac{1}{7} = 1\frac{4}{7}$
2	2	2	4	
3	12	-5/3	6	$x = 12 + 2 * -\frac{5}{3} = 8\frac{2}{3}$
3	12	0	10	$x = 12 + 2 * 0 = 12$
4	6	1/2	6	$x = 6 + 1 * \frac{1}{2} = 6\frac{1}{2}$
9	1	2	12	
10	12	-5/2	12	

Active Edge list

Y=5	x	inc	y out
	$1\frac{4}{7}$	-1/7	9
	$6\frac{1}{2}$	1/2	6
	$8\frac{2}{3}$	-5/3	6
	12	0	10

8. Sampling method



1. Sampling in the centre of the pixel

0, as the centre of the pixel is not of any color at all.

2. Sampling at the corners and taking the average

   1/4, as only the bottom right corner is colored.

3. Double sampling and taking the average

   Double sampling will split one pixel into sub-four pixels, and nine vertices in total. 2/9, as the middle right and bottom right are colored.

4. Double sampling and using the following mask to perform a weighted average

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

   $1/16 + 1/16 = 1/8$, same as above but with weights now.

9. Write a vertex and fragment shader that assigns the fragment based on modulating the appropriate texel.

```
// vertex shader

out vec2 texCoord;

void main(void) {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor = gl_Color;
    texCoord = vec2(gl_MultiTexCoord0);
}
```

```
in vec2 texCoord;

uniform sampler2D texUnit;  // first texture
uniform sampler2D texUnit2; // second texture

void main(void) {
    // applying second texture for the back face
    if (gl_FrontFacing)
        gl_FragColor = texture(texUnit, texCoord) * gl_Color;
    else
        gl_FragColor = texture(texUnit2, texCoord) * gl_Color;
}
```