UiO : University of Oslo

INF3320: Computer Graphics and Discrete Geometry

Texturing

Christopher Dyken and Martin Reimers
Corrections and additions by André R. Brodtkorb

September 24, 2014

Texturing

Linear interpolation

Real Time Rendering:

- Chapter 5: Visual Appearance
- Chapter 6: Texturing
  - The Texturing Pipeline
  - Image Texturing
  - Procedural Texturing

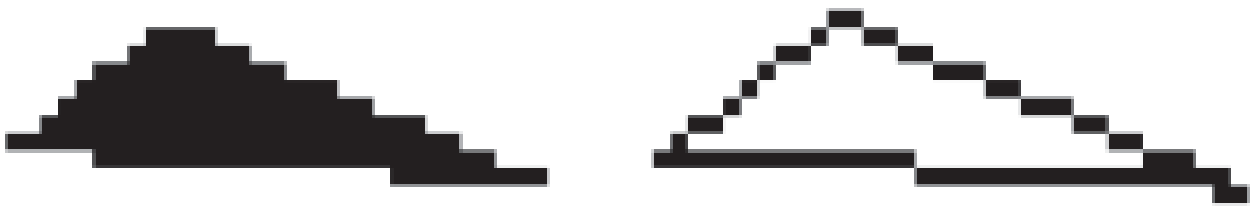
The Red Book:

- Chapter 9: Texture Mapping

Anti-aliasing

The problem

Simple rendering of primitives gives images with jagged edges.



The cause

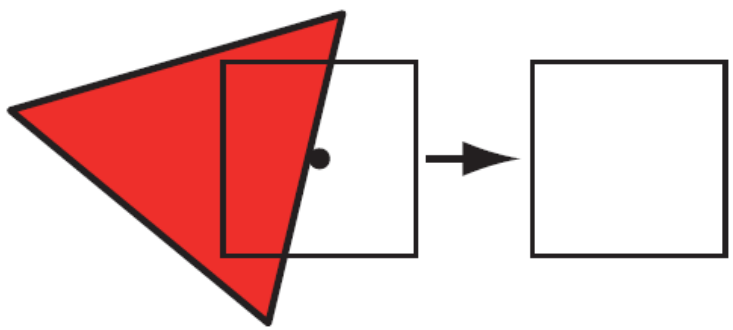
The primitives are described by their vertex coordinates which are precise (subject to floating point arithmetics), thus edges are described precisely.

Simple rendering (with no anti-aliasing) checks the center of the pixel in order to determine if the pixel is covered by a primitive and then assigns the color at that point to the pixel. But what about partially covered pixel?

The cause

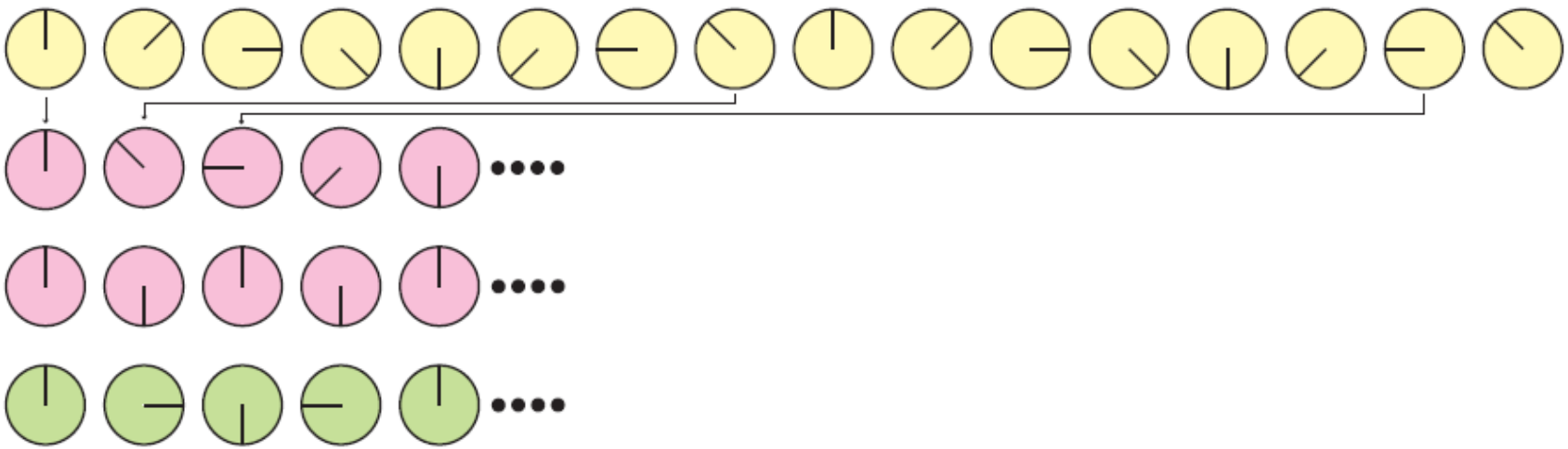
The primitives are described by their vertex coordinates which are precise (subject to floating point arithmetics), thus edges are described precisely.

Simple rendering (with no anti-aliasing) checks the center of the pixel in order to determine if the pixel is covered by a primitive and then assigns the color at that point to the pixel. But what about partially covered pixel?



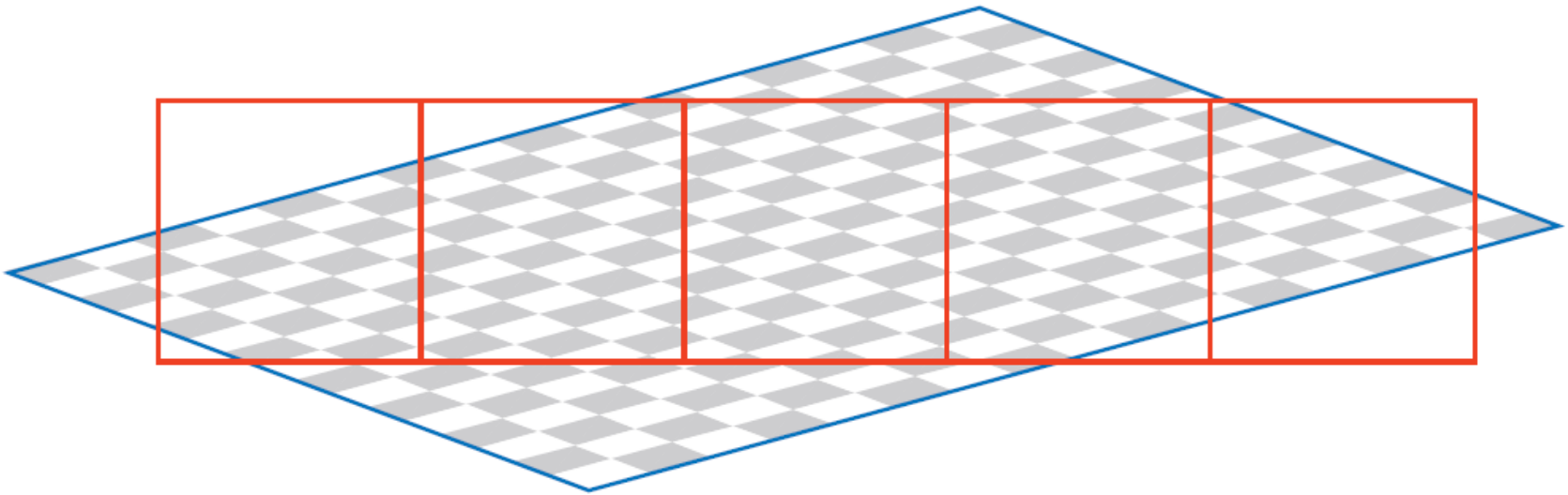
Temporal aliasing

Form the perspective of the computer animation we can also experience temporal aliasing - that is the object movement is too fast for the camera and thus appears to jump, flicker or move unnaturally.



Texture aliasing

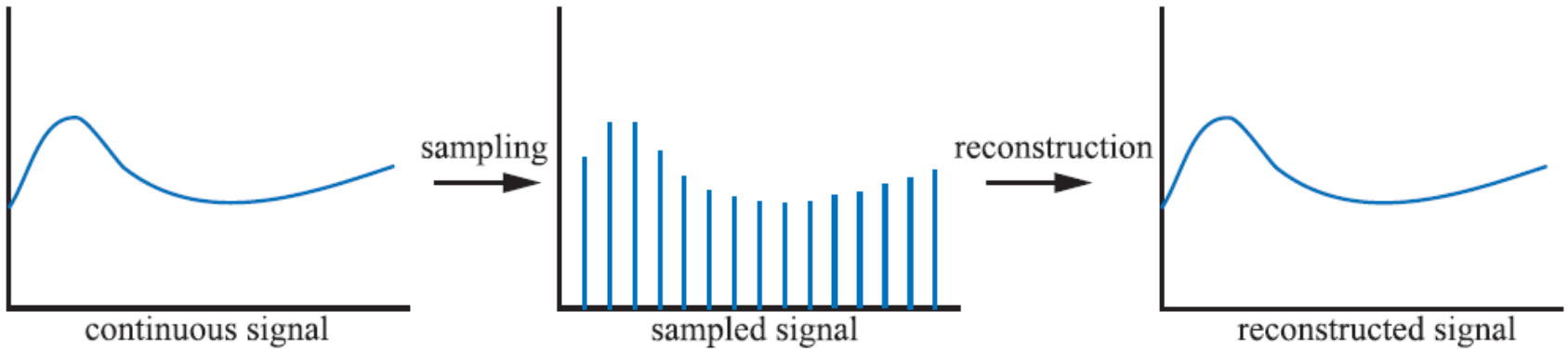
It is a third form of aliasing that is common in computer graphics. The reason for this type of aliasing is due to one pixel usually convening more than one or much less than one texture element (also called a *texel*).



There exist specialized methods of dealing with this problem - see Texturing section.

Sampling theory

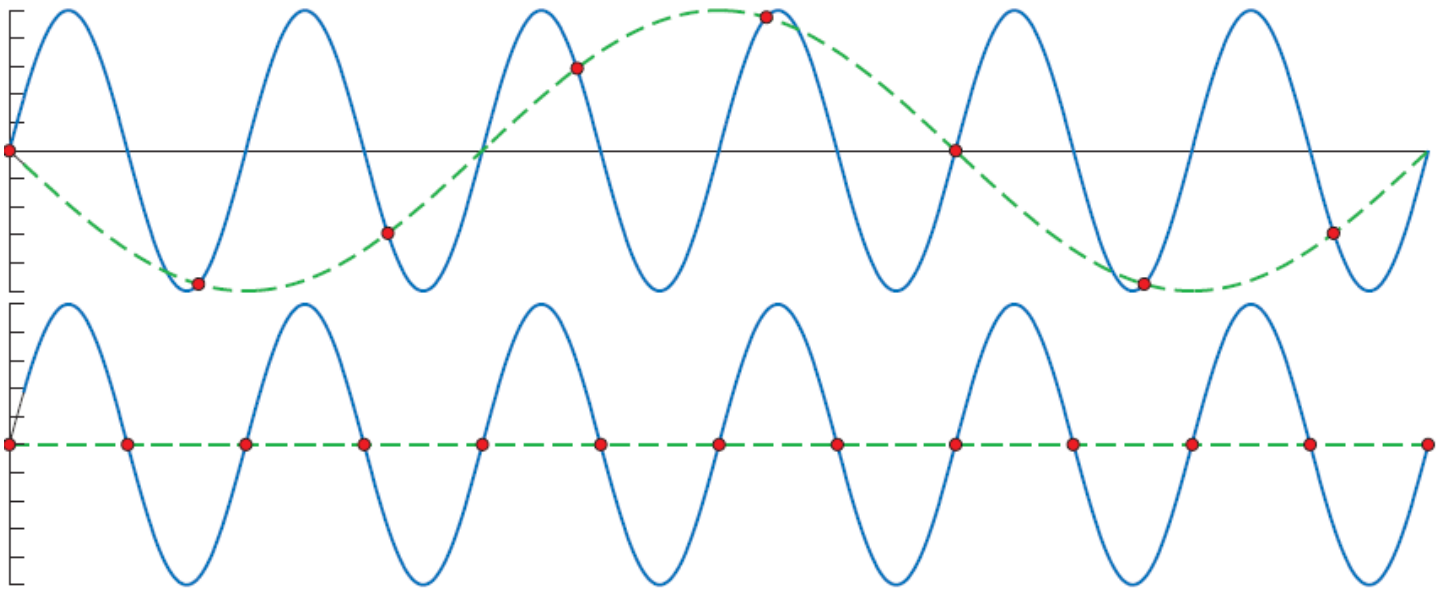
The problem described is a well known problem in the Sampling Theory. Since the primitive is described exactly it can be viewed as a two dimensional signal. In this context the rasterisation of the primitive corresponds to *sampling* and the process of displaying the rasterised image corresponds to *reconstruction* of the original signal.



Nyquist-Shannon theorem

Nyquist-Shannon theorem

If a signal contains no frequencies higher than B, it can be completely reconstructed from samples at a series of points spaced $1/(2B)$ apart.



The Nyquist-Shannon theorem states that to sample a signal with max frequency of B we need $2B$ samples per period.

Edges in image

It appears (Nyquist-Shannon theorem) that we could just increase the resolution until the sampling rate reaches the twice the frequency of the image. However the edge in the color function is a discontinuity and thus has an infinite maximum frequency.

Edges in image

It appears (Nyquist-Shannon theorem) that we could just increase the resolution until the sampling rate reaches the twice the frequency of the image. However the edge in the color function is a discontinuity and thus has an infinite maximum frequency.

We resort to taking multiple samples per-pixes and combining them to get the final color closer to the expected one.

Basic anti-aliasing algorithms

Two historical approaches:

- FSAA (Full screen anti-aliasing)
- MSAA (Multi sample anti-aliasing)

There exist various variations of the two methods (especially the second one) that can be mostly characterized by the spatial pattern used to take samples, the way the sampling results are combined and the knowledge about the neighboring pixels.

Basic anti-aliasing algorithms - FSAA

Is the simplest possible approach - we render the scene with a higher resolution and average over the blocks of pixels. Two typical modes:

- 2x2 - twice the resolution, blocks of 4 pixels
- 4x4 - four time the resolution, blocks of 16 pixels

Disadvantages are speed reduction memory requirements.

- Lighting computation costs 4 times higher for 2x2 (16 for 4x4)
- Modes higher than 4x4 are not feasible due to memory requirements.

Basic anti-aliasing algorithms - MSAA

Render the scene, primitive by primitive (in a regular resolution) and for each pixel do the following:

1. Perform the lighting computations for the pixel in a regular way. (Note that the rendering pipeline will only consider pixels that have a non-empty intersection with primitive).

Basic anti-aliasing algorithms - MSAA

Render the scene, primitive by primitive (in a regular resolution) and for each pixel do the following:

1. Perform the lighting computations for the pixel in a regular way. (Note that the rendering pipeline will only consider pixels that have a non-empty intersection with primitive).
2. Compute the exact depth for a number of samples in the pixel (usually arranged in some pattern).

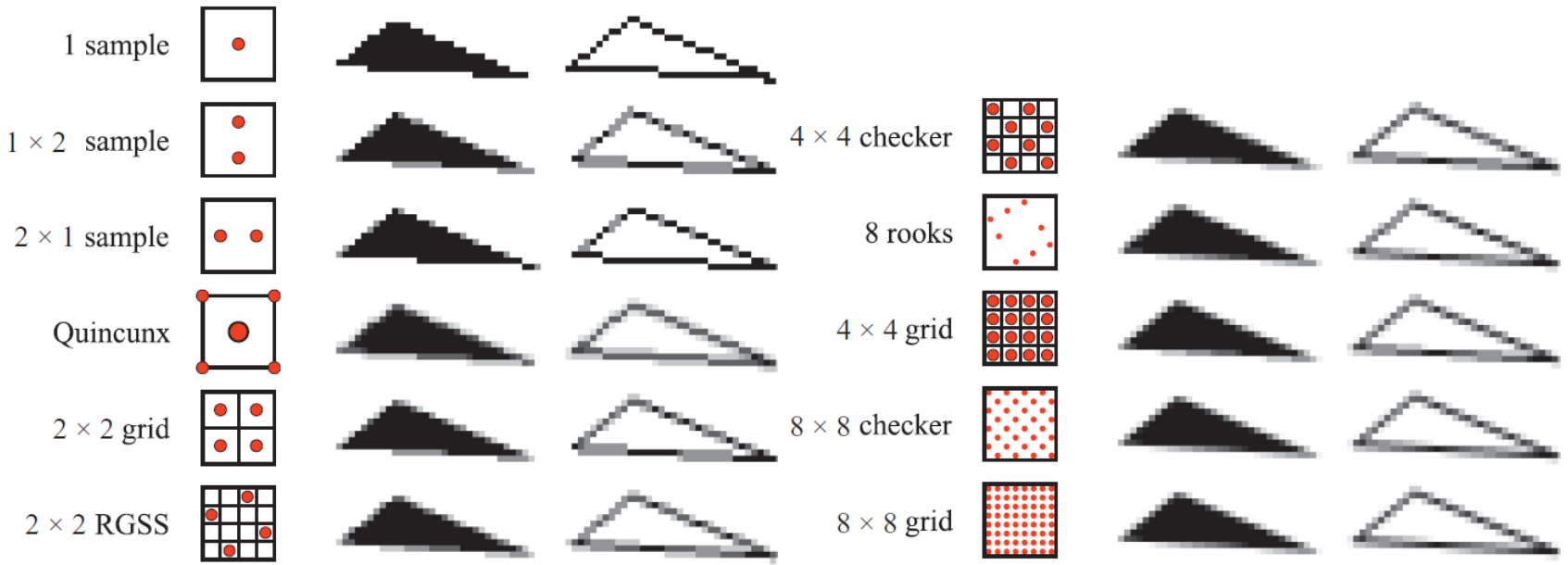
Basic anti-aliasing algorithms - MSAA

Render the scene, primitive by primitive (in a regular resolution) and for each pixel do the following:

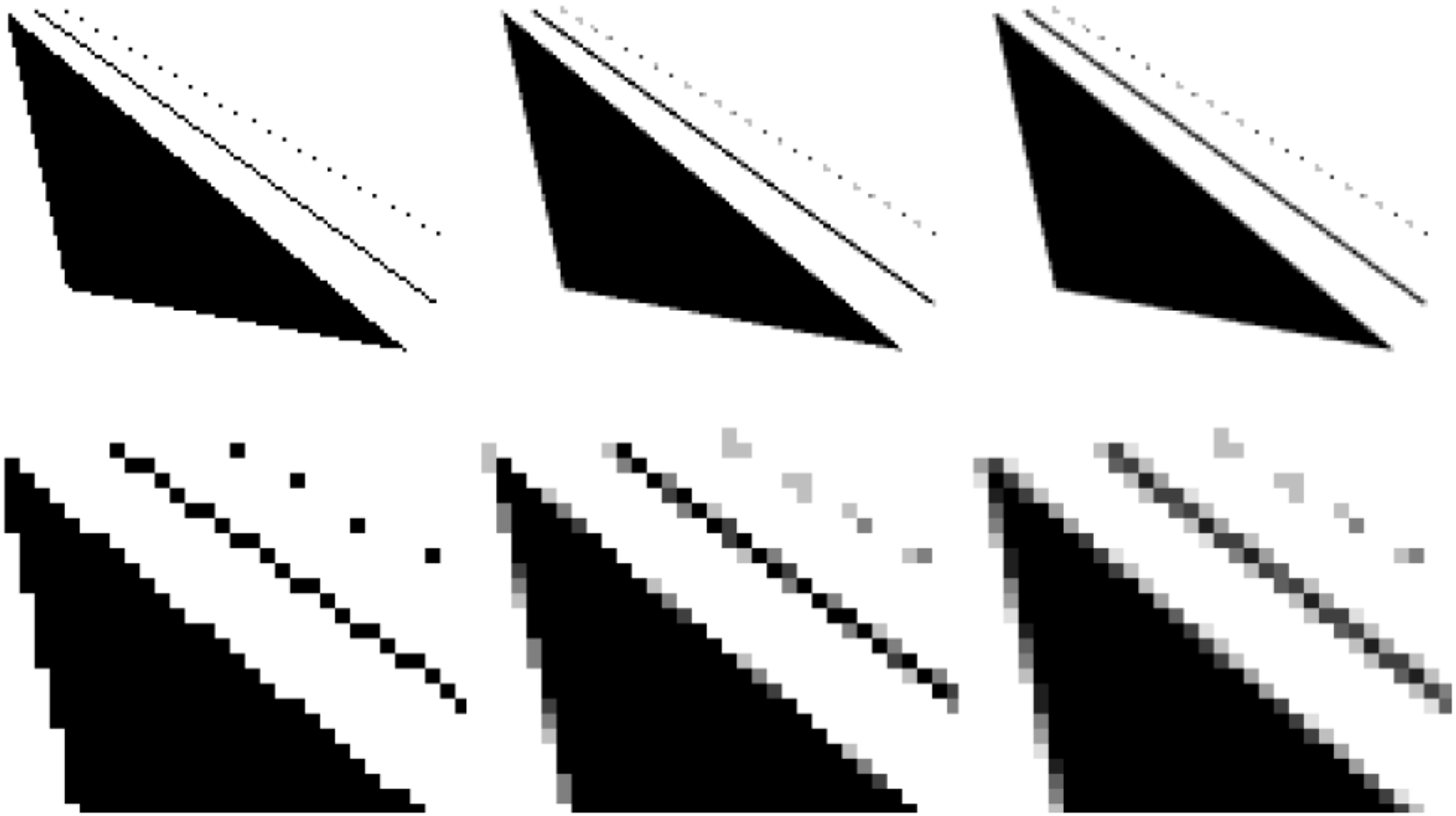
1. Perform the lighting computations for the pixel in a regular way. (Note that the rendering pipeline will only consider pixels that have a non-empty intersection with primitive).
2. Compute the exact depth for a number of samples in the pixel (usually arranged in some pattern).
3. Store the previously computed color in each sample that passes the test.

After the whole scene is rendered the colors in each pixels samples are averaged resulting in a final pixel color.

Anti-aliasing sample patterns



Anti-aliasing effects



Line and point anti-aliasing in OpenGL

OpenGL provides the possibility of mixing colors of pixels as is they were partially transparent (*blending*).

1. Enable blending using `glEnable(GL_BLEND)`.
2. Set blending parameters. Recommended is `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.
3. Enable line/point blending using `glEnable(GL_LINE_SMOOTH)` or `glEnable(GL_POINT_SMOOTH)`.
4. Optionally provide a quality hint using `glHint` function (e.g.,`GL_NICEST`)

Line and point anti-aliasing in OpenGL

OpenGL provides the possibility of mixing colors of pixels as is they were partially transparent (*blending*).

1. Enable blending using `glEnable(GL_BLEND)`.
2. Set blending parameters. Recommended is `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.
3. Enable line/point blending using `glEnable(GL_LINE_SMOOTH)` or `glEnable(GL_POINT_SMOOTH)`.
4. Optionally provide a quality hint using `glHint` function (e.g.,`GL_NICEST`)

WARNING

Should not be used in modern OpenGL for antialiasing!

Multisampling in OpenGL

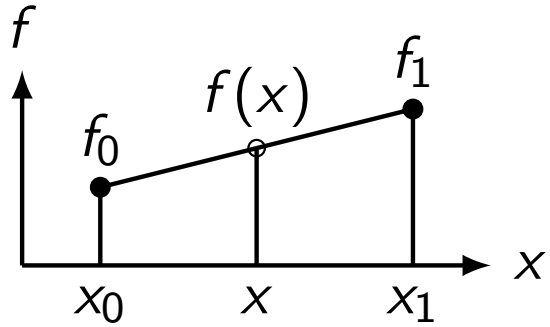
It is easy to enable basic multisampling in OpenGL with GLUT. The downside is lack of control.

1. Set number of sample points using `glutSetOption(GLUT_MULTISAMPLE, int value)`
2. Include `GLUT_MULTISAMPLE` value in the call to `glutInitDisplayMode`
3. Enable multisampling by a call to `glEnable(GL_MULTISAMPLE)`

Interpolation

Linear interpolation

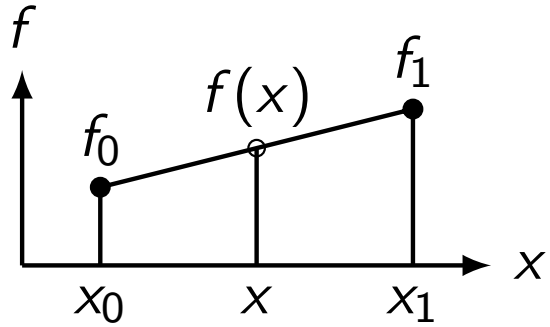
Problem: approximate a func. f from samples f_0, f_1 at x_0, x_1.
Typical case is data given in a table.



Solution: find $f(x) = ax + b$ such that $f(x_0) = f_0$ and $f(x_1) = f_1$

Linear interpolation

Problem: approximate a func. f from samples f_0, f_1 at x_0, x_1.
Typical case is data given in a table.



Solution: find $f(x) = ax + b$ such that $f(x_0) = f_0$ and $f(x_1) = f_1$

Remember: affine functions $f(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$ satisfy

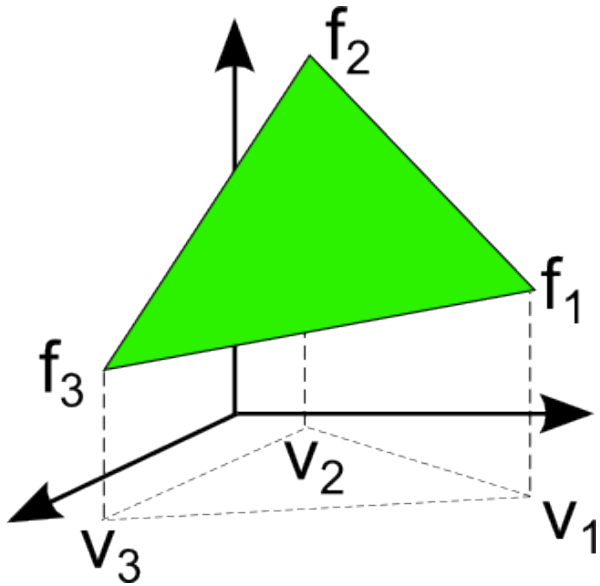
$$f((1-\alpha)\mathbf{x}_0 + \alpha\mathbf{x}_1) = (1-\alpha)f(\mathbf{x}_0) + \alpha f(\mathbf{x}_1)$$

Yields a formula (Bernstein form) for linear interpolation: only need to find (unique) α s.t. $\mathbf{x} = (1-\alpha)\mathbf{x}_0 + \alpha\mathbf{x}_1$

Note: $\mathbf{x}$ and $f(\mathbf{x})$ are weighted averages (convex combinations)

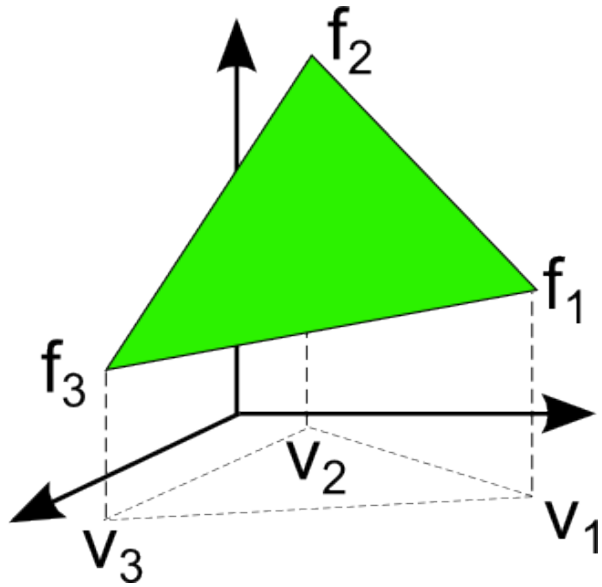
Linear interpolation in 2D

Similar problem in 2D: find affine $f(x, y) = ax + by + c$ such that $f(\mathbf{v}_i) = f_i$ for 3 points $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$.



Linear interpolation in 2D

Similar problem in 2D: find affine $f(x, y) = ax + by + c$ such that $f(\mathbf{v}_i) = f_i$ for 3 points $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$.



Three equations with three unknowns: solve for a, b, c:

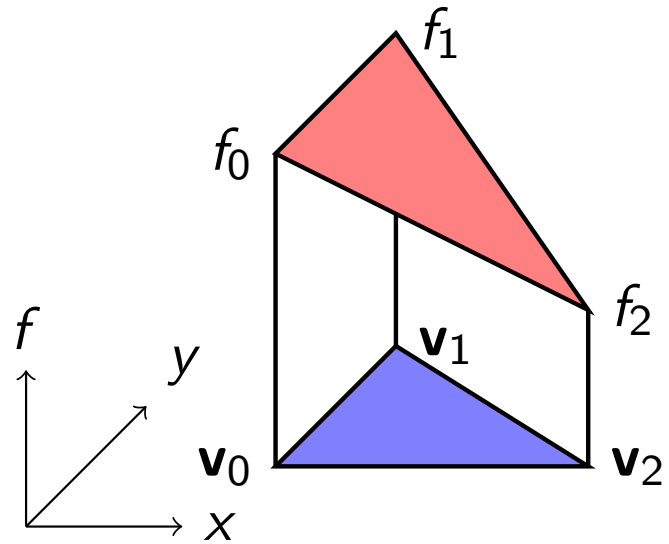
$$f(v0) = f0$$
$$f(v1) = f1$$
$$f(v2) = f2$$

Barycentric coordinates

Given data (f_i) at the vertices of a triangle $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2]$, we can interpolate at an arbitrary point $\mathbf{v}$ by

$$f(\mathbf{v}) = \alpha_0 f(\mathbf{v}_0) + \alpha_1 f(\mathbf{v}_1) + \alpha_2 f(\mathbf{v}_2)$$

$f(\mathbf{v})$ is a *convex combination* (average) of f at the vertices.



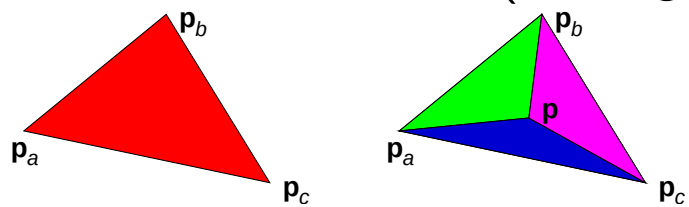
These are so-called barycentric coordinates.

Barycentric interpolation 2D

Given a point $\mathbf{v}$ and a triangle $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2]$, there are unique *barycentric coordinates* $\alpha_i = \alpha_i(\mathbf{v})$ such that

$$\mathbf{v} = \alpha_0\mathbf{v}_0 + \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2, \qquad \alpha_0 + \alpha_1 + \alpha_2 = 1$$

$\mathbf{v}$ is expressed as a *convex combination* (average) of the vertices.



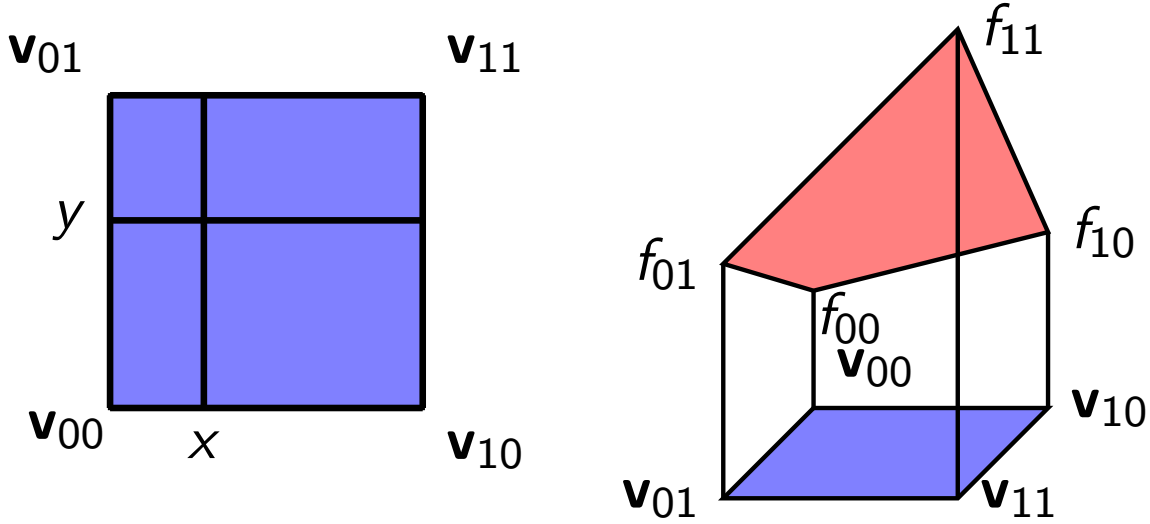
$$\alpha_0 = \frac{\text{area}(\mathbf{v}, \mathbf{v}_1, \mathbf{v}_2)}{\text{area}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)} \quad \alpha_1 = \frac{\text{area}(\mathbf{v}, \mathbf{v}_0, \mathbf{v}_2)}{\text{area}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)} \quad \alpha_2 = \frac{\text{area}(\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1)}{\text{area}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)}$$

- $0 \leq \alpha_i \leq 1$ for $\mathbf{v}$ inside T
- One coordinate is redundant - e.g. $\alpha_2 = 1 - \alpha_0 - \alpha_1$
- Also called homogeneous coordinates
- Generalize to nD, important and useful

Bi-linear interpolation

Given tabulated data f_{ij} on a square $\mathbf{v}_{ij} = (i, j)$, with $i, j = 0, 1$, we can find a bi-linear function f such that $f(i, j) = f_{ij}$:

$$f(x, y) = (1 - x)(1 - y)\ f_{00} + x(1 - y)\ f_{10} + xy\ f_{11} + (1 - x)y\ f_{01}$$



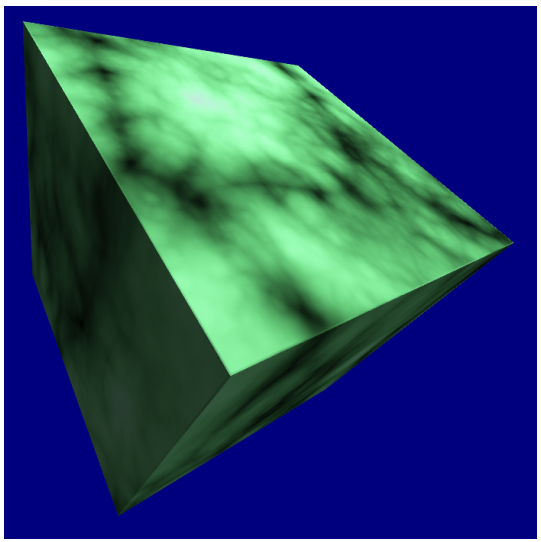
- A linear function in each variable
- ... but a quadratic on the diagonal $x = y$
- Generalizes to any dimension (tri-,..., n-linear interpolation)
- Used to interpolate gridded data, eg. textures ...

Texturing

What is texturing?

Texturing in computer graphics

Texturing means modifying a surface's appearance using an image, function or other data source.



Texturing may for example modify:

- Color (image mapping).
  - The most common case.
- Normals (bump mapping).
- Geometry (displacement mapping).

Motivation

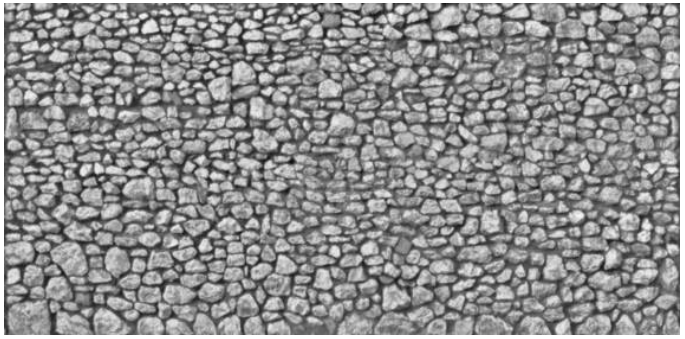
- Better appearance at lower cost.
  - May use lower quality geometry, i.e. fewer vertices.
  - May use less advanced lighting algorithms.
- Easy to add decals, text, images.

Example textures

Example textures from www.arroway.de.

Diffuse

Bump



Specular

Example rendering

Tradeoffs

Basic rule: Fine details in textures, coarser features need geometry.

The optimal boundary between the two may depend on a multitude of factors.

Typical case

1. Far away: Image texturing is fine.
2. Closer: Illusion breaks down, bump mapping optimal solution.
3. View approaches tangential: Illusion breaks down again, displacement mapping or proper geometry necessary.

Defining textures in OpenGL

Textures are managed through *texture names*:

```
GLuint texture_names[n];
glGenTextures(n, texture_names);
glDeleteTextures(n, texture_names);
```

OpenGL supports several *texture types* (targets):

`GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`,...

We *bind texture* to use/manipulate it:

```
glBindTexture( GL_TEXTURE_2D, texture_names[0]);
```

Then we can *specify texture*, manipulate it, etc:

```
glTexImage2D( GL_TEXTURE_2D, level, iformat
              width, height, border,
              format, type, &image[0][0][0] );
```

Defining textures in OpenGL - II

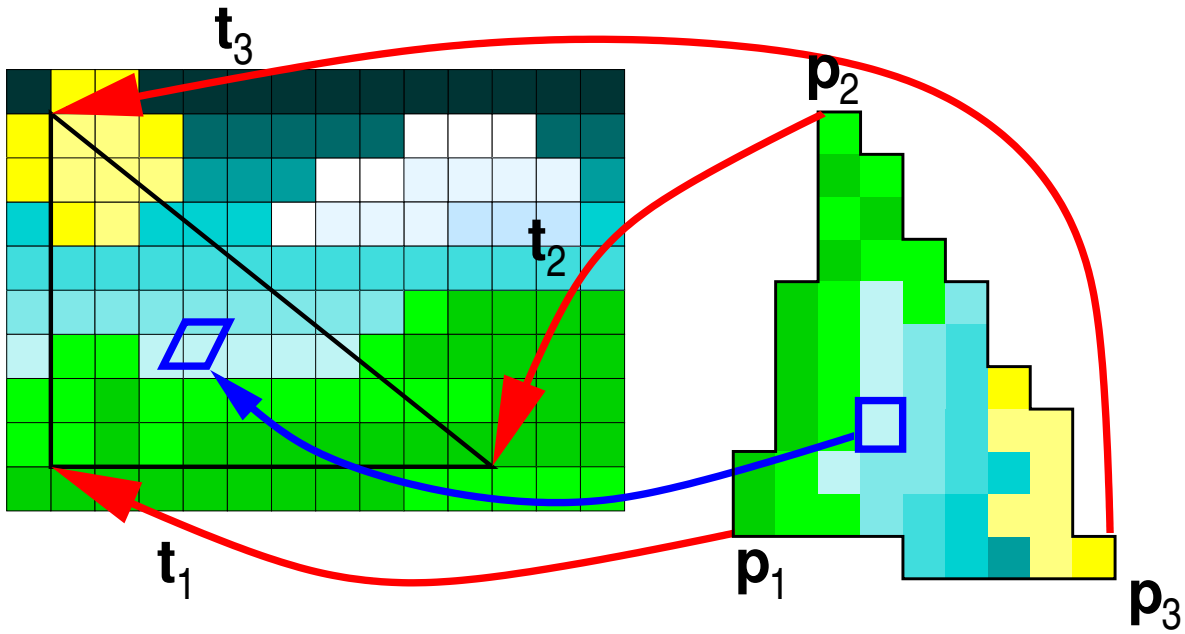
Many parameters controls the bound texture (wrapping, filtering, ...) `glTexParameteri(GL_TEXTURE_2D,...);`

Texturing must be enabled: `glEnable(GL_TEXTURE_2D);`

Many functions for manipulating textures `glCopyTexImage2D();` `glTexSubImage2D();` ...

Texture coordinates

Texture coords $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3$ are texture space positions that assign a position in the texture to each vertex $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ (red arrows)



- Specified arbitrarily at each vertex.
- Interpolated over each triangle.
  $\Longrightarrow$ texture coordinate for each pixel (blue arrow).

Notice that pixels and texels rarely match

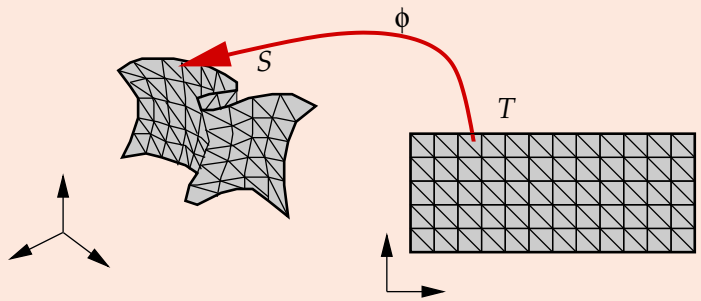
Parametric surfaces usually have a natural set of texcoords

Parameterization

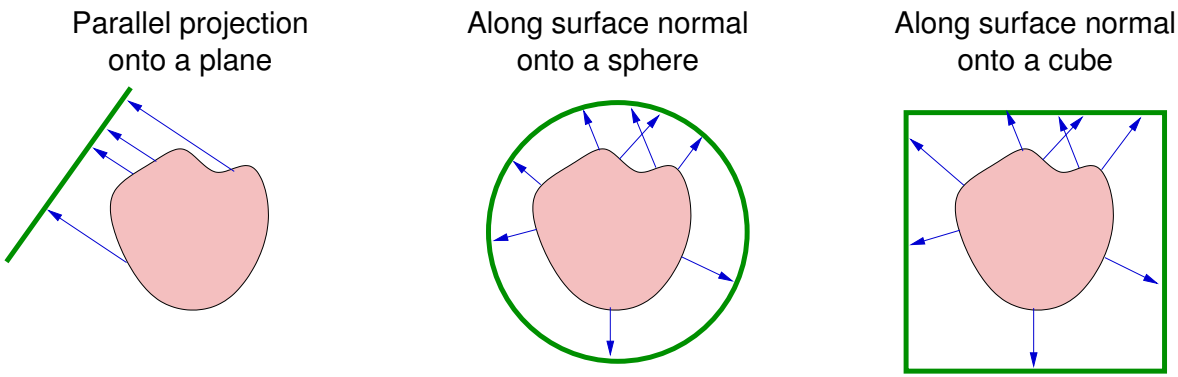
Objects without parameterization needs to be *parameterized*.

We must find a mapping ϕ from the domain T to the image S.

Parameterization is not trivial and is an open research question.



Sometimes we can obtain texcoords by *projecting* the vertex positions onto an intermediate geometry (projector funcs):



In OpenGL we use `glTexCoord` to specify the current texcoord, `glVertex` creates a vertex with the current texcoord associated.

`glTexCoord` *before* `glVertex`

Corresponder functions

Corresponder functions specifies the mapping between texture coordinates and texels.

Vanilla OpenGL corresponder function

```
glTexCoord( 0.0, 0.0 )  ⟺  my_texels[0][0]
glTexCoord( 1.0, 0.0 )  ⟺  my_texels[511][0]
glTexCoord( 0.0, 1.0 )  ⟺  my_texels[0][511]
glTexCoord( 1.0, 1.0 )  ⟺  my_texels[511][511]
```

Generalized Texturing

Texturing works by modifying surface attributes over the triangle.

1. The fragment location in space is the starting point.
2. A *projector function* gives texture space values.
3. The *corresponder function* transforms texture space values to a location in texture image (not a pixel in image).
4. *Sampling* the texture at the image location gives a value.
5. The *value transform function* transforms the value[1].
6. The resulting value is used to modify a surface property[2].

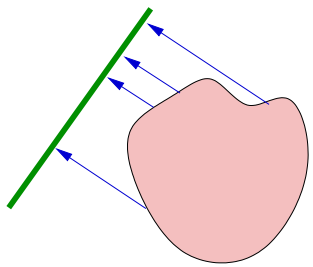
[1] e.g. gamma correction
[2] e.g. surface color

The projector function

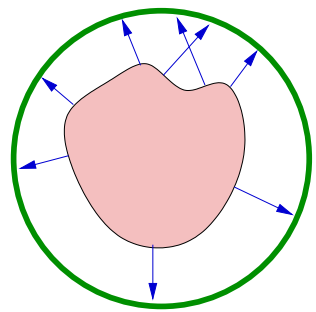
Takes an object space pos. and gives a texture space pos.

We can project onto a simple intermediate object and use that object's parameterization to determine the texture position:

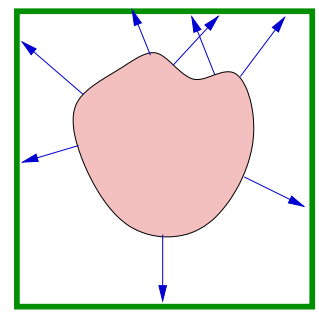
Parallel projection onto a plane



Along surface normal onto a sphere



Along surface normal onto a cube

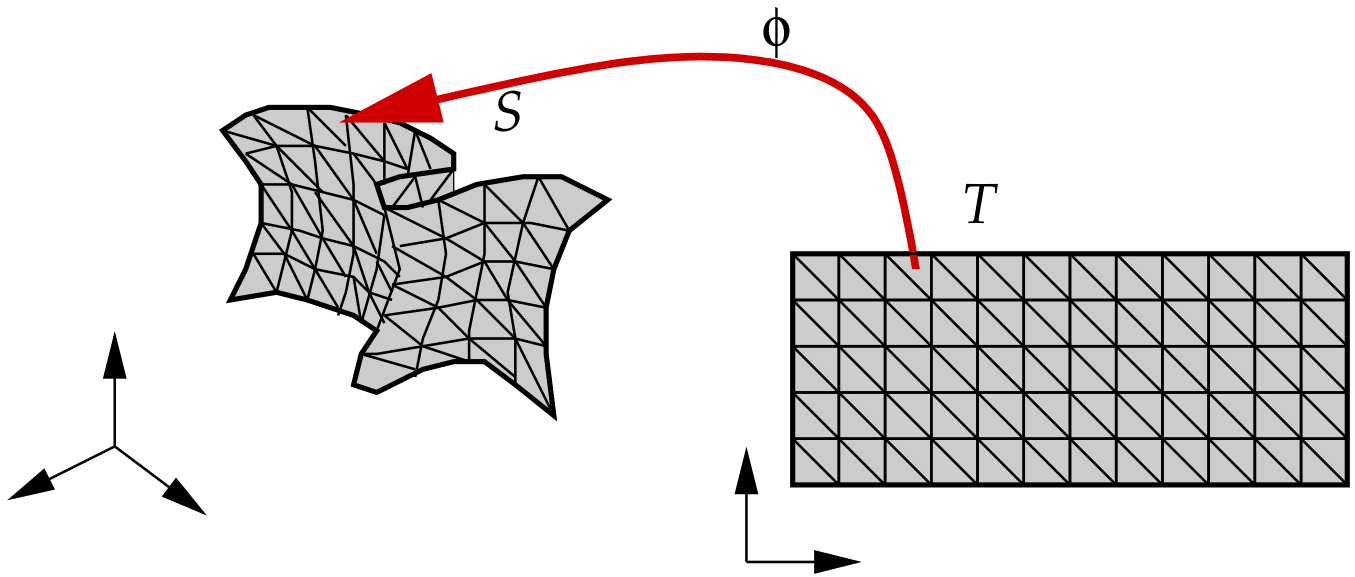


Parametric curved surfaces usually have an intrinsic parameter space, and we can use this directly.

OpenGL's `glTexGen` provides some different projector functions.

Ideally, the texture is glued to the object in a way that:

- minimizes distortion
- good correspondence between object and parameter space.



The problem of fitting a texture space (parameter space) onto an object is known as *parameterization* and is an open research question.

The corresponder function

Maps from texture space position to image locations.

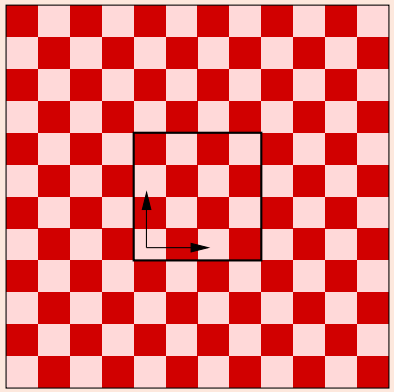
- Texture domains usually 2D with (u, v) in $[0, 1] \times [0, 1]$.
- Sometimes 3D where the third coordinate can represent depth
  $\Longrightarrow$ volumetric texture.
- And in some cases 4D (homogeneous coordinates)
  $\Longrightarrow$ e.g. spotlight effects.

Often, the corresponder function simply scales the coordinates

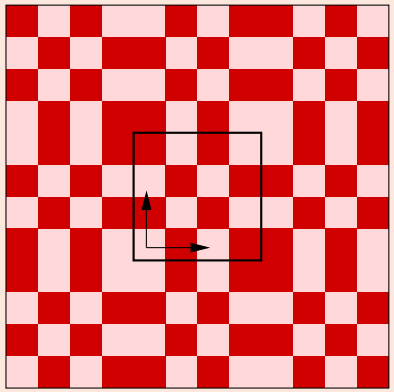
$$[0, 1] \times [0, 1] \quad \rightarrow \quad [0, 256] \times [0, 256]$$

for a texture of size 256×256.

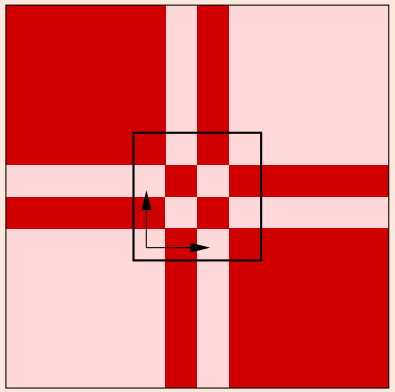
Corresponder func determines behavior outside $[0, 1] \times [0, 1]$



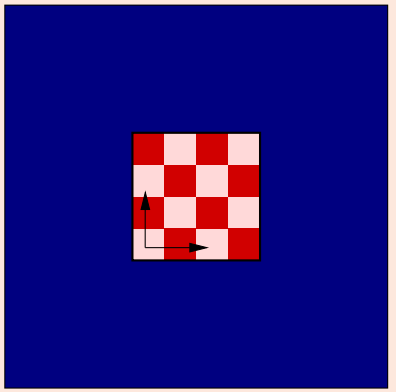
repeat



mirror



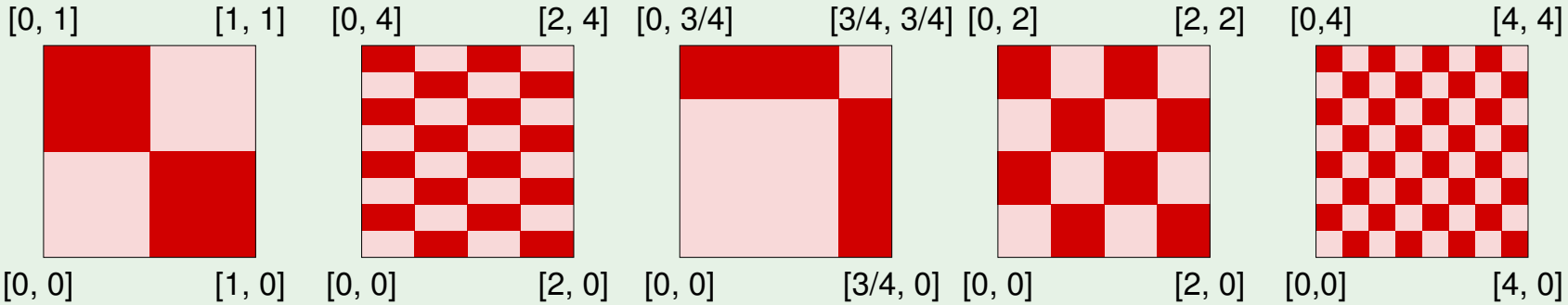
clamp



clamp to border

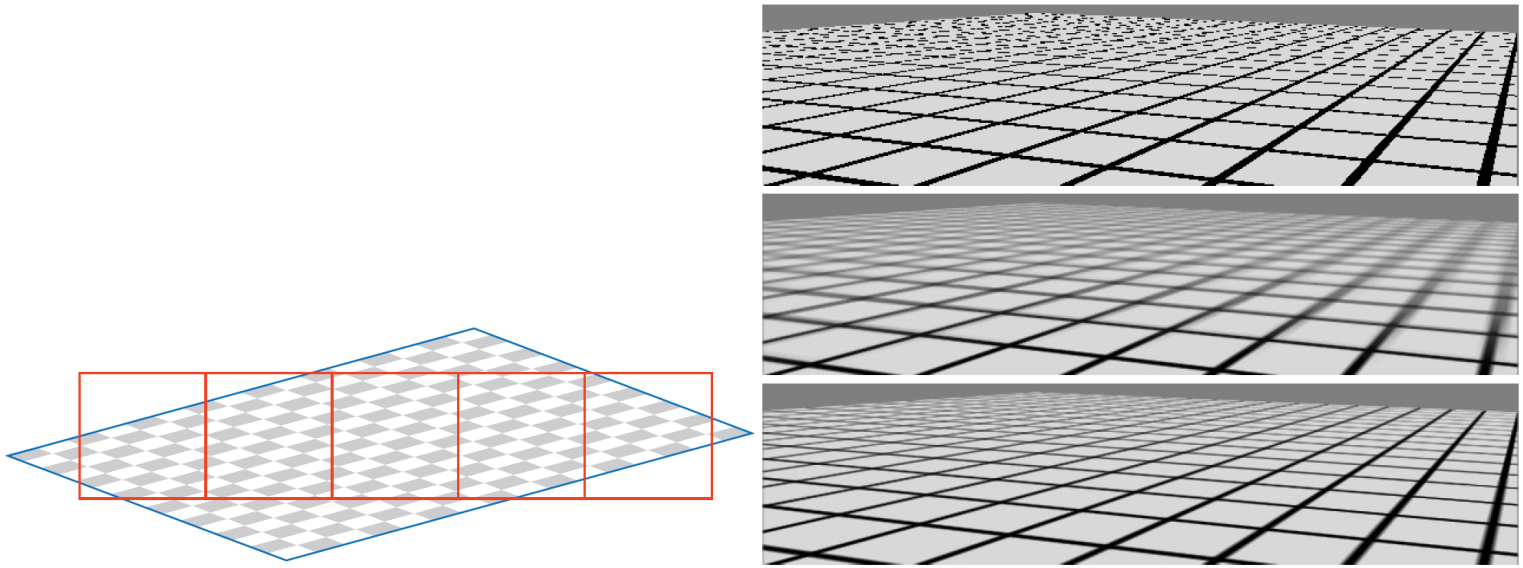
This is called wrap mode and is specified with `glTexParameter`

We can make the texture repeat itself



Texture filtering

Texels and pixels rarely match:



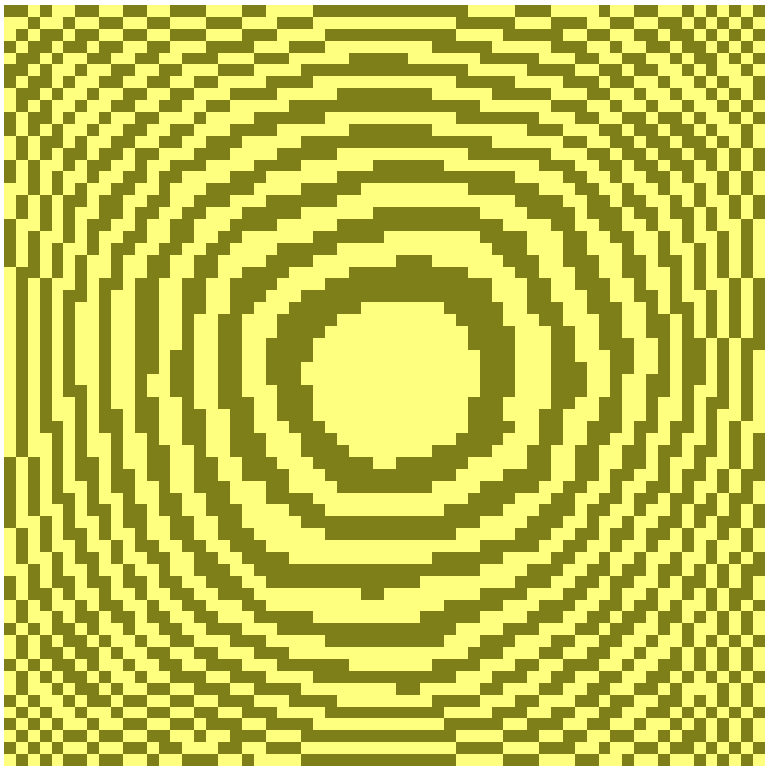
Use filtering techniques (interpolation)

- Mipmapping, anisotropic filtering, antialiasing,...
- Tradeoff: quality vs speed

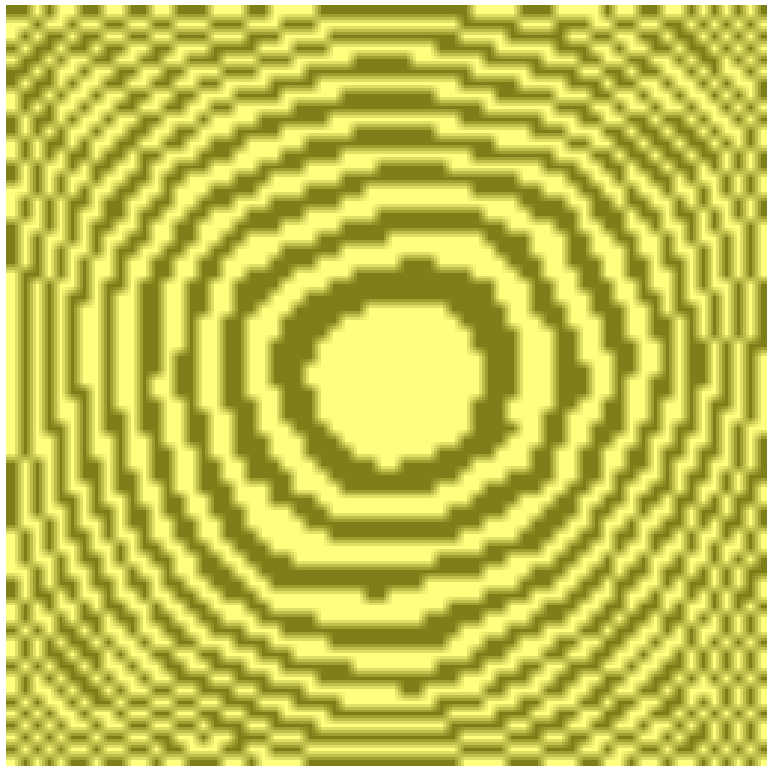
Texture magnification I

In this case, a texel covers several pixels.

OpenGL has two strategies to deal with this:



Nearest neighbour.



Bilinear interpolation.

Texture magnification II

Nearest neighbor strategy means that every pixel whose center is covered by a texel in question has its color. This can cause rapid changes in color.

Bilinear interpolation strategy considers four texels closest to the pixel center and biliearly interpolates between their colors using distances from four texel centers to pixel center. This gives smooth color transitions over many pixels.

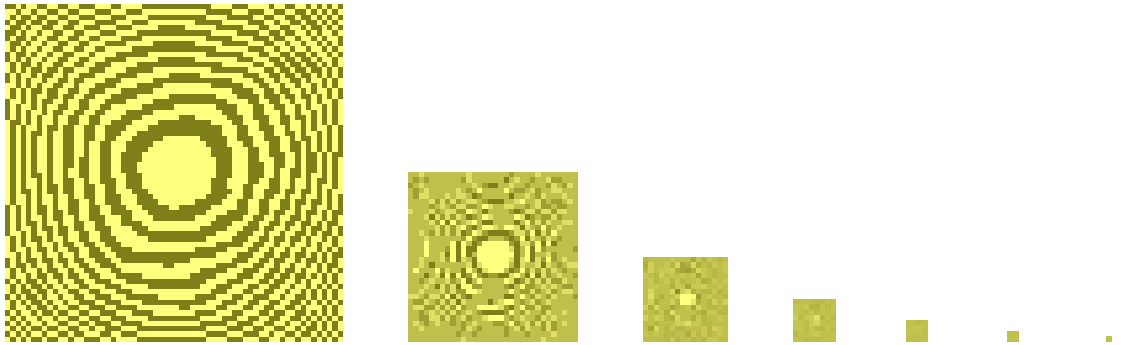
The texture magnification filter is specified using one of

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Texture minification

If a pixel covers several texels, the texture is under-sampled.
We get *aliasing* artifacts.

Mipmaps are pre-filtered textures, halving the size for each level:



Then, use the mipmap level s.t. pixel and texel size matches and

- Choose the nearest mipmap level.
- Interpolate between two adjacent mipmap levels.

In each mipmap either choose the nearest texel or interpolate.

The texture minification filter is specified using

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter);
```

Texture combine function

The texture combine function specifies how the texture color is combined with the color of the fragment.

OpenGL fixed function texture combine functions

Replace replaces the texel color with the texture color.

Decal lets texture α blend between pixel color and texture color.

Blend uses texture color to blend texel and a pre-specified color.

Modulate multiplies the surface colour with texture colour.

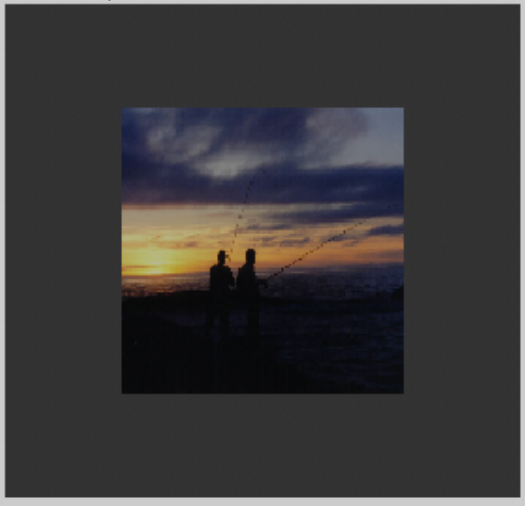
Example: Lighted textures

Render white shapes with lighting enabled and use the modulate texture combine.

In OpenGL, `glTexEnv*` to specifies the texture combine function.

Nate robbins tutorial

Screen-space view



Texture space view

Command manipulation window

```
GLfloat border_color[ ] = {  1.00 ,  0.00 ,  0.00 ,  1.00 };
GLfloat env_color[ ] = {     0.00 ,  1.00 ,  0.00 ,  1.00 };

glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border_color);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, env_color);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  GL_REPEAT);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  GL_MODULATE);

glEnable(GL_TEXTURE_2D);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, w, h, GL_RGB, GL_UNSIGNED_BYTE, image);
glColor4f(   0.60  , 0.60  , 0.60  , 1.00  );
glBegin(GL_POLYGON);
glTexCoord2f( 0.0  , 0.0  ); glVertex3f( -1.0 , -1.0 , 0.0  );
glTexCoord2f( 1.0  , 0.0  ); glVertex3f( 1.0  , -1.0 , 0.0  );
glTexCoord2f( 1.0  , 1.0  ); glVertex3f( 1.0  , 1.0  , 0.0  );
glTexCoord2f( 0.0  , 1.0  ); glVertex3f( -1.0 , 1.0  , 0.0  );
glEnd();
```

Click on the arguments and move the mouse to modify values.

Texturing in GLSL - Basic samplers

To access the textures we use variable of the `Sampler*` type. The sampler variable must be declared as `uniform`. To access a value stored in a `Sampler*` variable we use the `texture` function family - first parameter is a sampler, second texture coordinates. Mipmaping, interpolation etc. are done for us.

Basic sampler types:

- `Sampler1D` - represents a 1 dimensional texture
- `Sampler2D` - represents a 2 dimensional texture
- `Sampler3D` - represents a 3 dimensional texture

Texturing in GLSL - Example

Vertex Shader

```
varying vec2 TexCoord;

void main() {
   TexCoord = gl_MultiTexCoord0.st;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Texturing in GLSL - Example

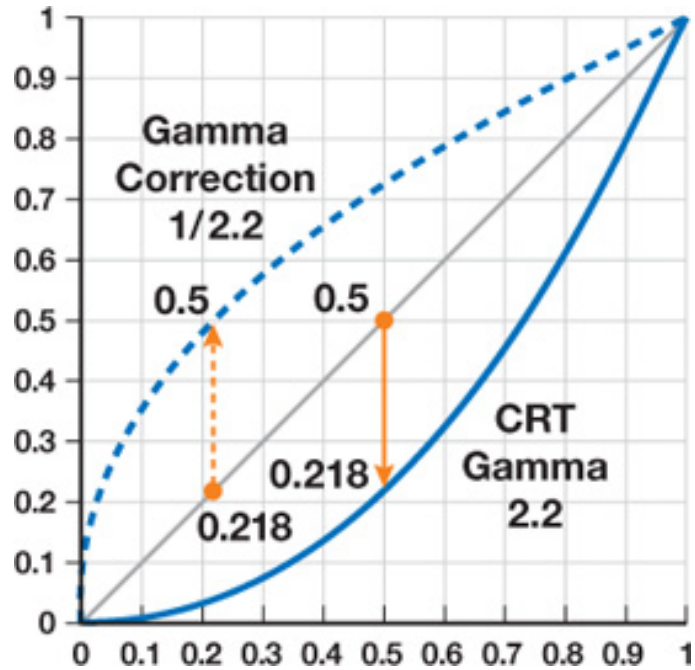
Fragment Shader

```
varying vec2 TexCoord;

uniform sampler2D ColorMap;

void main (void) {
   gl_FragColor = texture(ColorMap, TexCoord);
}
```

Gamma correction

Gamma Correction - the problem

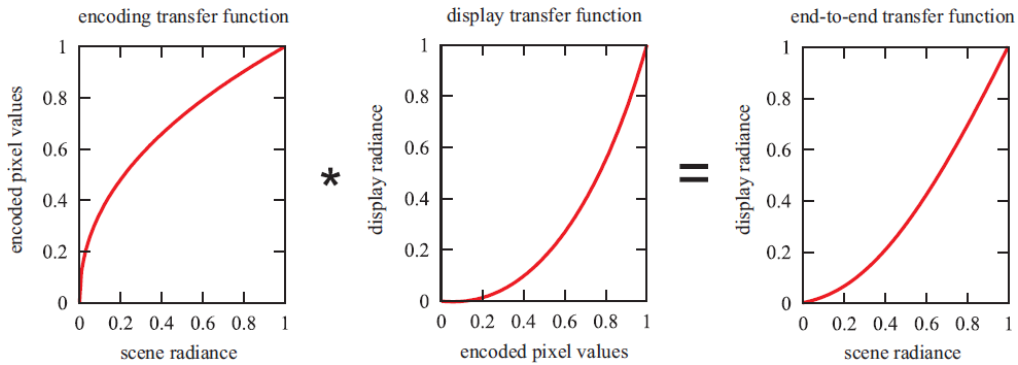
Percieved intensity is usually not a linear function of light intensity - the eye is more sensitive to variation in low intensity



For example a typical gamma of 2.2 means that a pixel with 0.5 intensity emits less then a quarter (0.218) of light than a pixel with 1.0 intensity, whereas the expected emitted light intensity is 0.5. Typically γ is in $2.0 - 2.4$.

Gamma Correction - the solution

In order to represent color correctly we modify input signal by $V_{out} = V_{in}^{\gamma}$ to account for the hardware specific pixel intensity to emitted light intensity curve.



γ-correction can be handled by modern GPUs or operating systems. Without that support we can still resolve the problem ourselves using shaders.

Bottom line - γ-correction is important since we add intensities: portability across platforms, image quality, texturing and interpolation.

The last slide

Next time: The OpenGL pipeline