Computer Graphics

Textures, part II

Konstantin Tretyakov
kt@ut.ee

Oct 30, 2013

Standard Graphics Pipeline

Vertex transform	**Determine clip-space position of a triangle**
Culling and clipping	**Determine whether the triangle is visible**
Rasterization	**Determine all pixels belonging to the triangle**
Fragment shading	**For each pixel, determine its color**
Visibility tests & blending	**Draw pixel *(if needed)***

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$I_A M_A +$$

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$I_A M_A + (I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ +$$

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$I_A M_A + \left(I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ + I_S M_S \cdot (\boldsymbol{n}^T \boldsymbol{h})_+^{\,10} \right)$$

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$I_A M_A + \left(I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ + I_S M_S \cdot (\boldsymbol{n}^T \boldsymbol{h})_+{}^{10}\right)\frac{1}{d^2}$$

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$I_A M_A + \left(I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ + I_S M_S \cdot (\boldsymbol{n}^T \boldsymbol{h})_+{}^{10} \right) \frac{1}{d^2} (-\boldsymbol{l}^T \boldsymbol{s})^{15}$$

Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$\textbf{mix}\Big(I_A M_A + \left(I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ + I_S M_S \cdot (\boldsymbol{n}^T \boldsymbol{h})_+^{10} \right) \frac{1}{d^2} (-\boldsymbol{l}^T \boldsymbol{s})^{15},$$

$$\text{black},$$

$$\exp(-2|z|) \Big)$$

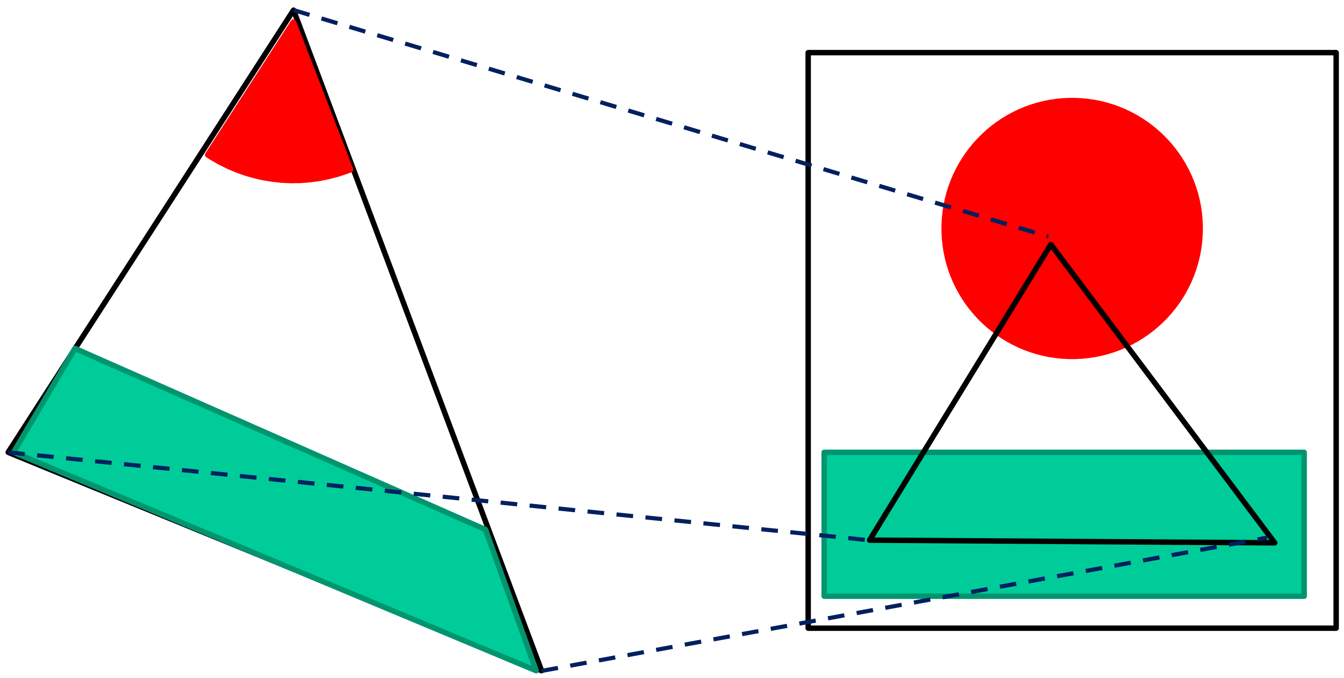
Quiz

- Formulate the Blinn model:
  - One light source, one color component
  - Specular exponent 10
  - Quadratic attenuation
  - Spotlight exponent 15
  - Exponential fog with coefficient 2
  - Fog color black

$$\exp(-2|z|) \cdot \left(I_A M_A + \left(I_D M_D \cdot (\boldsymbol{n}^T \boldsymbol{l})_+ + I_S M_S \cdot (\boldsymbol{n}^T \boldsymbol{h})_+^{\,10} \right) \frac{1}{d^2} (-\boldsymbol{l}^T \boldsymbol{s})^{15} \right)$$

In the previous lecture

- We can specify color of every pixel of a triangle by mapping it from a *texture image*.



In the previous lecture

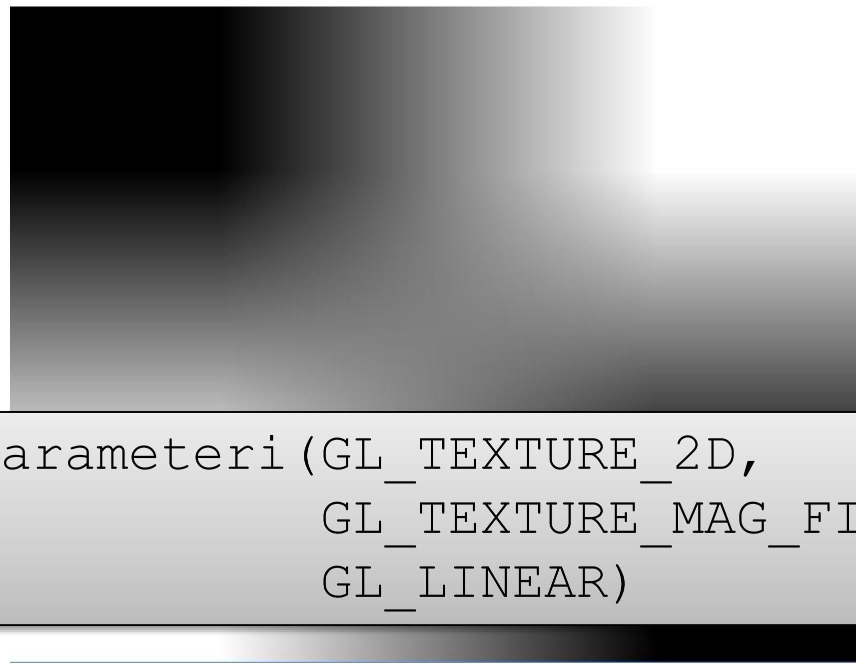
- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, environment maps, shadow maps
- Procedural textures

Quiz

- This is a 2x2 pixel checkerboard texture spread on a rectangle. What filtering setting is used?

Quiz

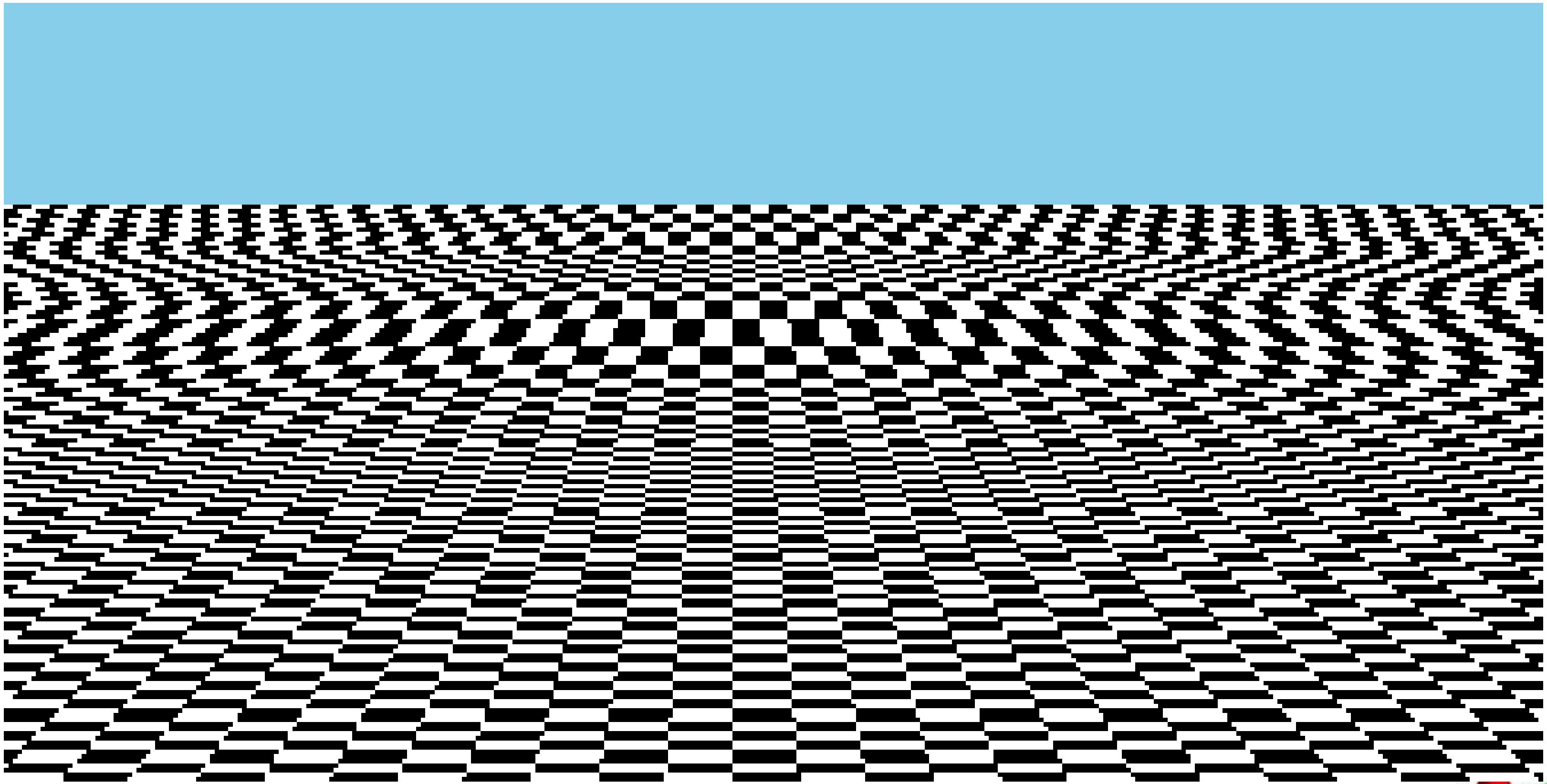
- This is a 2x2 pixel checkerboard texture spread on a rectangle. What filtering setting is used?



```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER,
                GL_LINEAR)
```

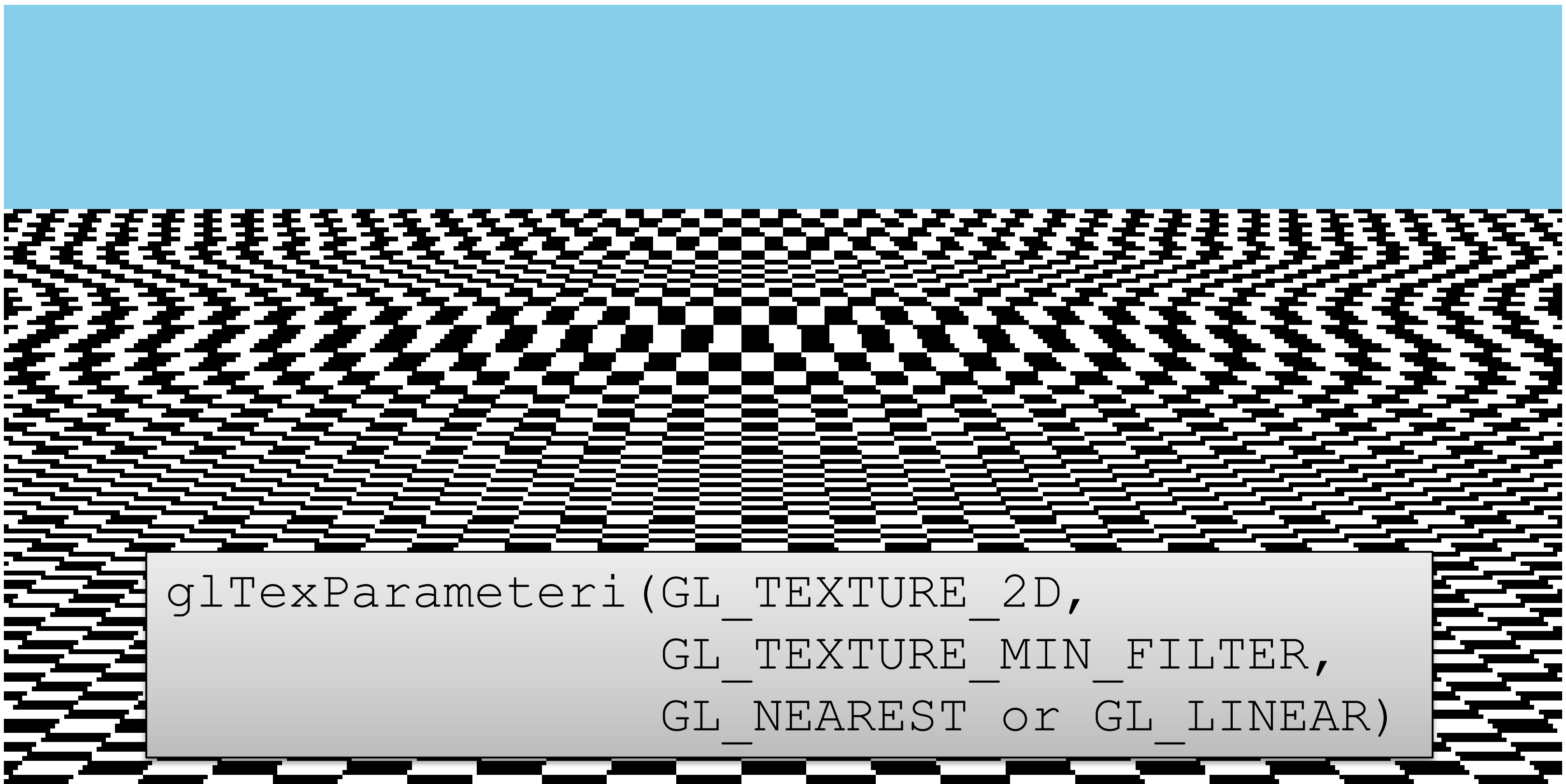
Quiz

- What filtering setting is used here?



Quiz

- What filtering setting is used here?



```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER,
                GL_NEAREST or GL_LINEAR)
```

Next

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, environment maps, shadow maps
- Procedural textures

Textures as look-up tables

```
uniform sampler1D complexFunction;

void main() {
    ...
    float light_attenuation =
             texture1D(complexFunction, angle).x;
    ...
}
```

Textures as look-up tables

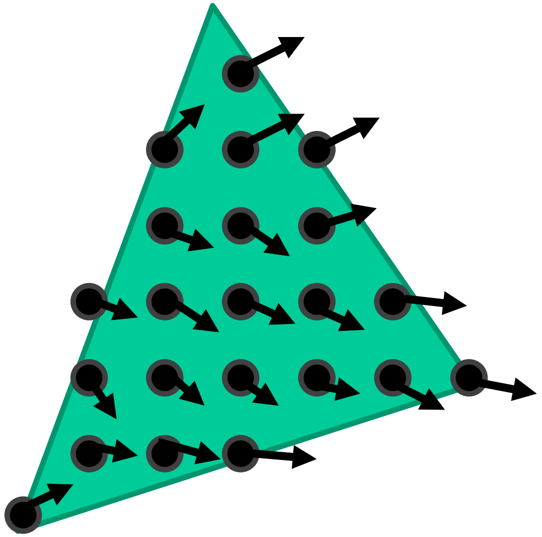
```
uniform sampler3D specularIntensity;

void main() {
   ...
   float specular_term =
            texture3D(specularIntensity,
                      reflectionDirection).x;
   ...
}
```

Textures as look-up tables

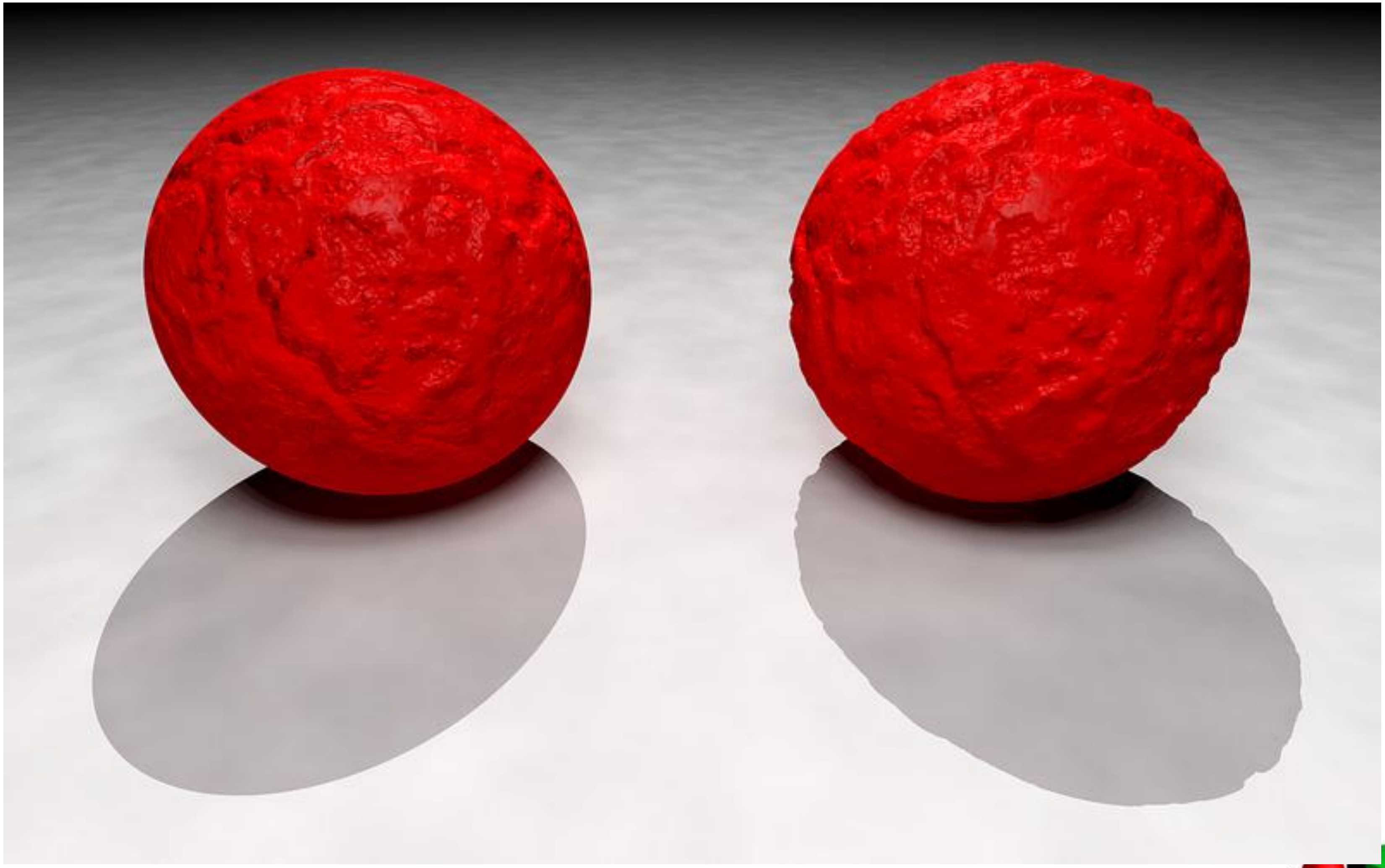
```
uniform sampler2D randomness;

void main() {
    ...
    float effect =
              texture2D(randomness, gl_FragCoord.xy).x;
    ...
}
```

Normal mapping

Use a texture to specify (or modify) the *normal* at each point of the surface.



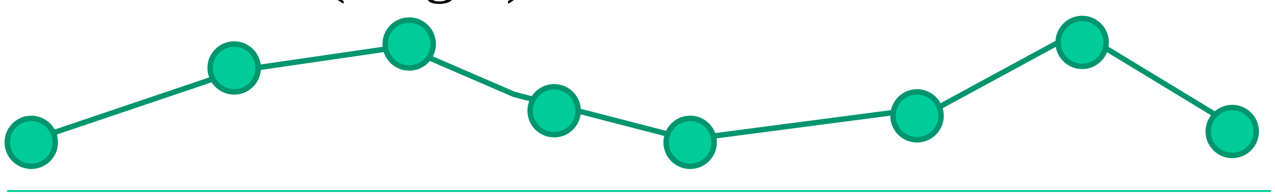
Normal mapping



Bump-mapping

- Although it is possible to store the actual (x,y,z) normal direction for each texel, it is more common to use the texture to store a single number per texel – the *height* above the corresponding point.

Texture values (height)

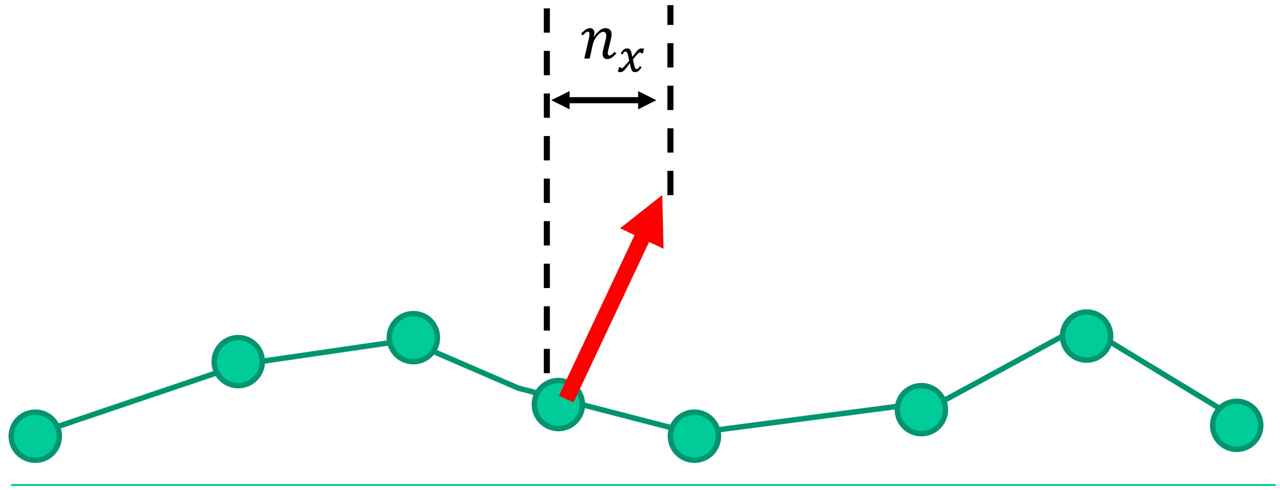


polygon

Bump-mapping

- The normal direction can be inferred from the height map via discrete differentiation:

$$n_x(\mathrm{uv}) \propto h(\mathrm{uv} - (\varepsilon, 0)) - h(\mathrm{uv})$$



Bump-mapping

- The normal direction can be inferred from the height map via discrete differentiation:

$$\boldsymbol{n} \propto \begin{pmatrix} h(\mathrm{uv} - (\varepsilon, 0)) - h(\mathrm{uv}) \\ h(\mathrm{uv} - (0, \varepsilon,)) - h(\mathrm{uv}) \\ 1 \end{pmatrix}$$

Bump-mapping

- The normal direction can be inferred from the height map via discrete differentiation:

$$\boldsymbol{n} \propto \begin{pmatrix} h(\mathrm{uv} - (\varepsilon, 0)) - h(\mathrm{uv}) \\ h(\mathrm{uv} - (0, \varepsilon,)) - h(\mathrm{uv}) \\ 1 \end{pmatrix}$$

Note that those are normal coordinates in the polygon-local coordinate system.

Bump-mapping

- The normal direction can be inferred from the height map via discrete differentiation:

$$\boldsymbol{n} \propto \begin{pmatrix} h(\mathrm{uv} - (\varepsilon, 0)) - h(\mathrm{uv}) \\ h(\mathrm{uv} - (0, \varepsilon,)) - h(\mathrm{uv}) \\ 1 \end{pmatrix}$$

The normal's object coordinates are then

$\boldsymbol{Mn}$

Where $\boldsymbol{M} = (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$ provides the "up", "right" and "out" directions wrt polygon

Reflections

Suppose we are looking at some point of a highly reflective object

Reflections

Suppose we are looking at some point of a highly reflective object

We should see the reflection
of whatever is in that direction

Reflections

Suppose we are looking at some point of a highly reflective object

We could in principle trace this ray until we figure out what is there in that direction.

Environment mapping

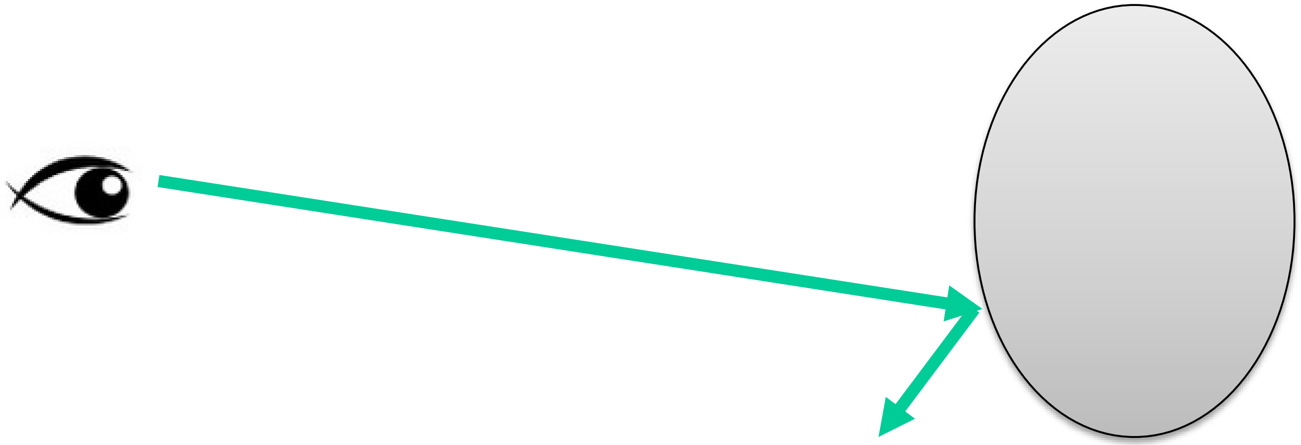
Use a texture to store what is seen from a point for each possible direction.

Suppose we are looking at some point of a highly reflective object

No need to trace the ray, if we have the necessary value precomputed.

Environment mapping

Assume that this mapping is **the same for all points**, because the reflections are "distant" (hence "environment" mapping).



Environment map

Environment map – a texture that stores for any possible direction the color of a distant point in that direction.

Environment map

Environment map – a texture that stores for any possible direction the color of a distant point in that direction.

Sphere map

Cube map

HEALpix map

… or any cartographic projection

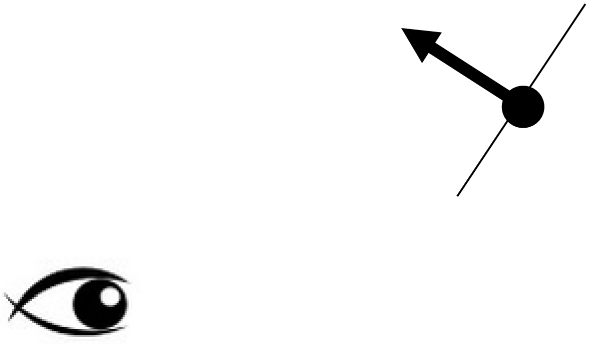
Sphere map

An orthogonal view of a perfectly reflective sphere shows reflections for all possible directions.

Sphere map

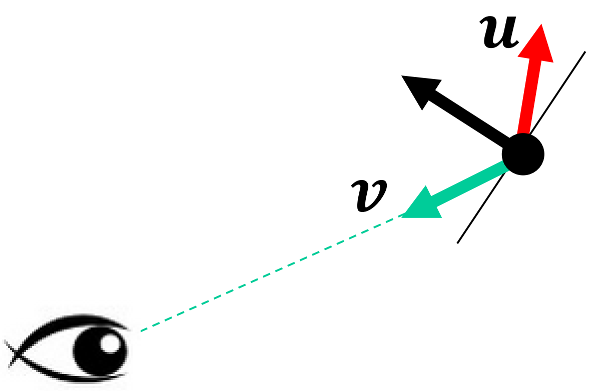
So now for any reflection direction we simply have to find the point on this image, where the sphere had the same reflection vector.

Sphere mapping



Say we are shading this pixel on a reflective polygon

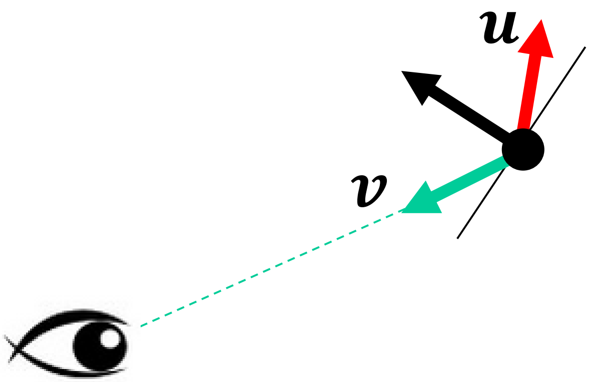
Sphere mapping

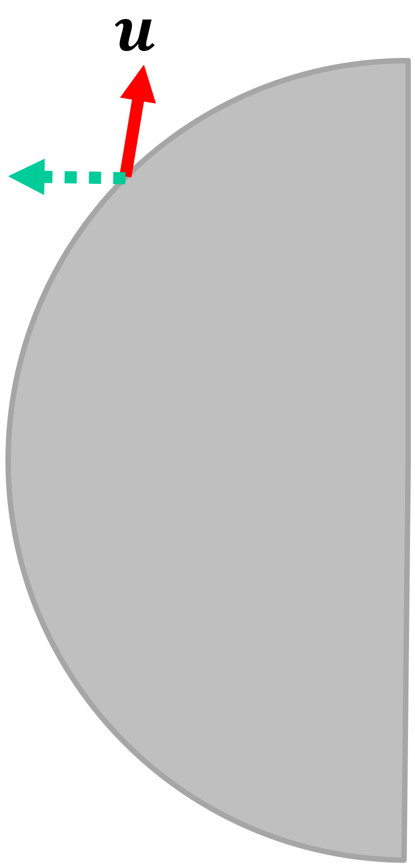


Say we are shading this pixel on a reflective polygon

Start by computing the viewer direction vector and its reflection against the normal

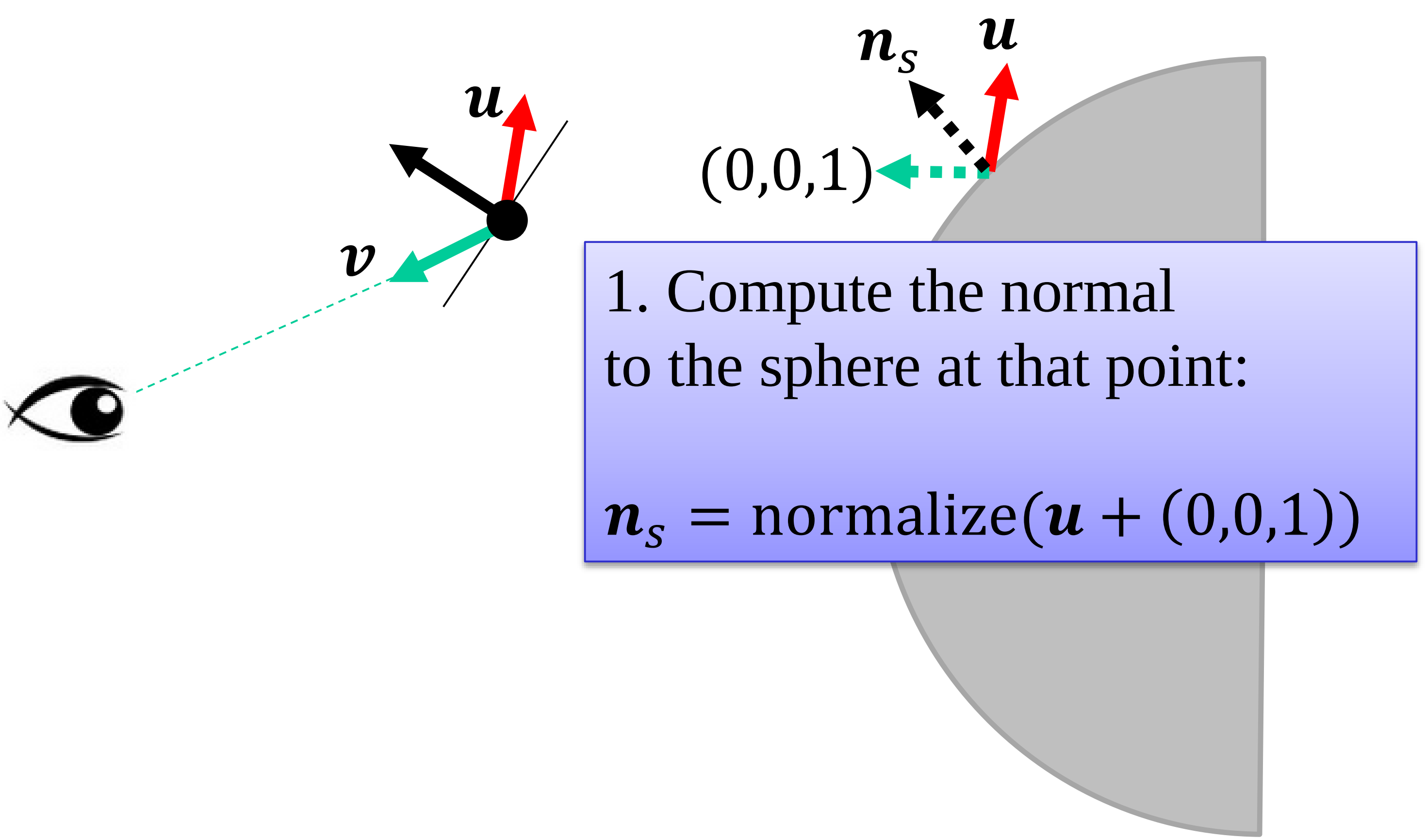
Sphere mapping





Now we need to find the point on the sphere that would have **the same reflection vector** (when viewed orthogonally straight)

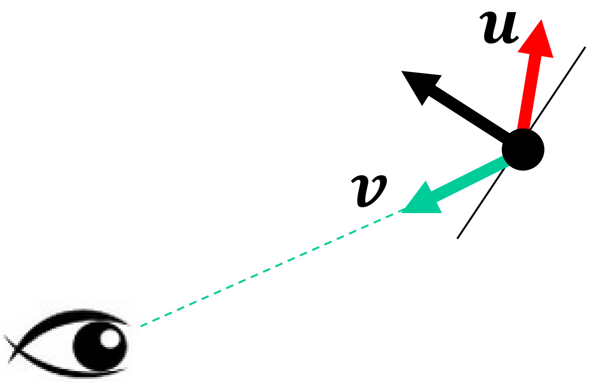
Sphere mapping

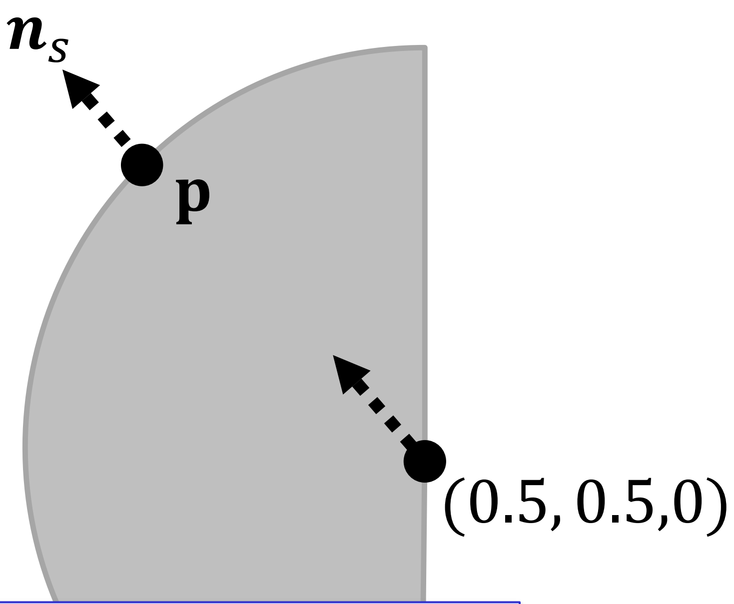


1. Compute the normal
to the sphere at that point:

$\boldsymbol{n}_s = \text{normalize}(\boldsymbol{u} + (0,0,1))$

Sphere mapping

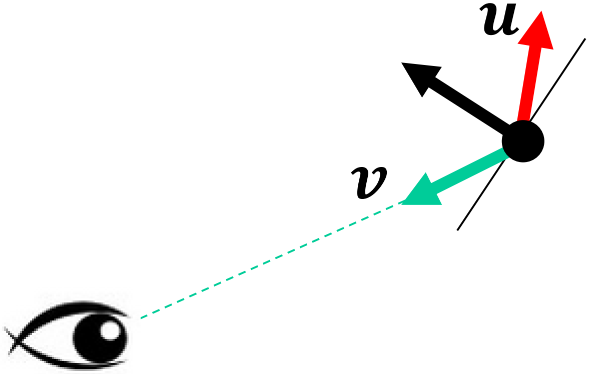


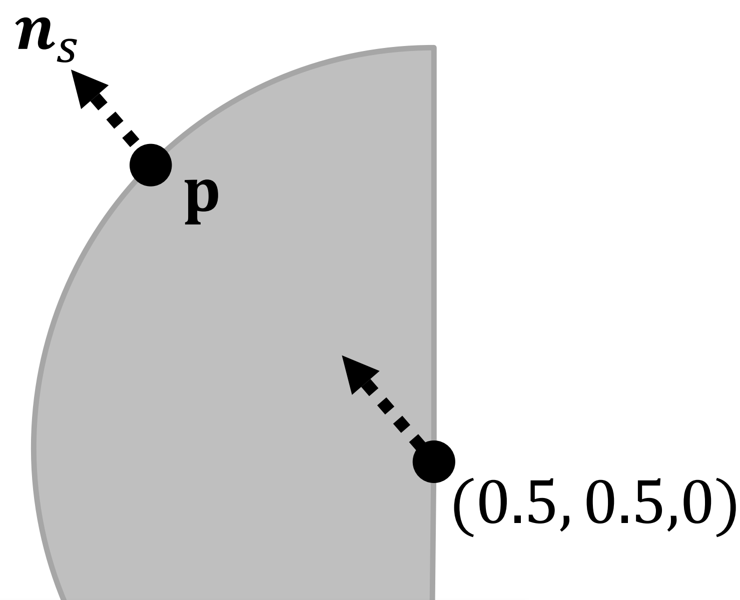


2. Find the actual point on the sphere

$$\mathbf{p} = (0.5,0.5,0) + 0.5\boldsymbol{n}_s$$

Sphere mapping





2. Find the actual point on the sphere

$$\mathbf{p} = (0.5,0.5,0) + 0.5\boldsymbol{n}_s$$

3. Sample the sphere map texture at (p_x, p_y)

Sphere map

OpenGL can do it for us via automated texture coordinate generation:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

See e.g. glTexGeni manual

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n´ be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\,\mathbf{n}'\,\mathbf{n}'^{T}\mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + \left(f_z + 1\right)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

http://msdn.microsoft.com/en-us/library/windows/desktop/dd368634(v=vs.85).aspx

Cube map

- Sphere map only works well for a fixed viewer direction, as the “sides” of the sphere map are heavily undersampled.

Cube map

- Sphere map only works well for a fixed viewer direction, as the “sides” of the sphere map are heavily undersampled.
- **Cube map** is a set of 6 images, forming the “insides” of a cube. Now for the point in the center of the cube we can easily compute what is seen in any direction.
- This lets us compute reflections for any viewpoint with equal accuracy.

Cube map

Cube map

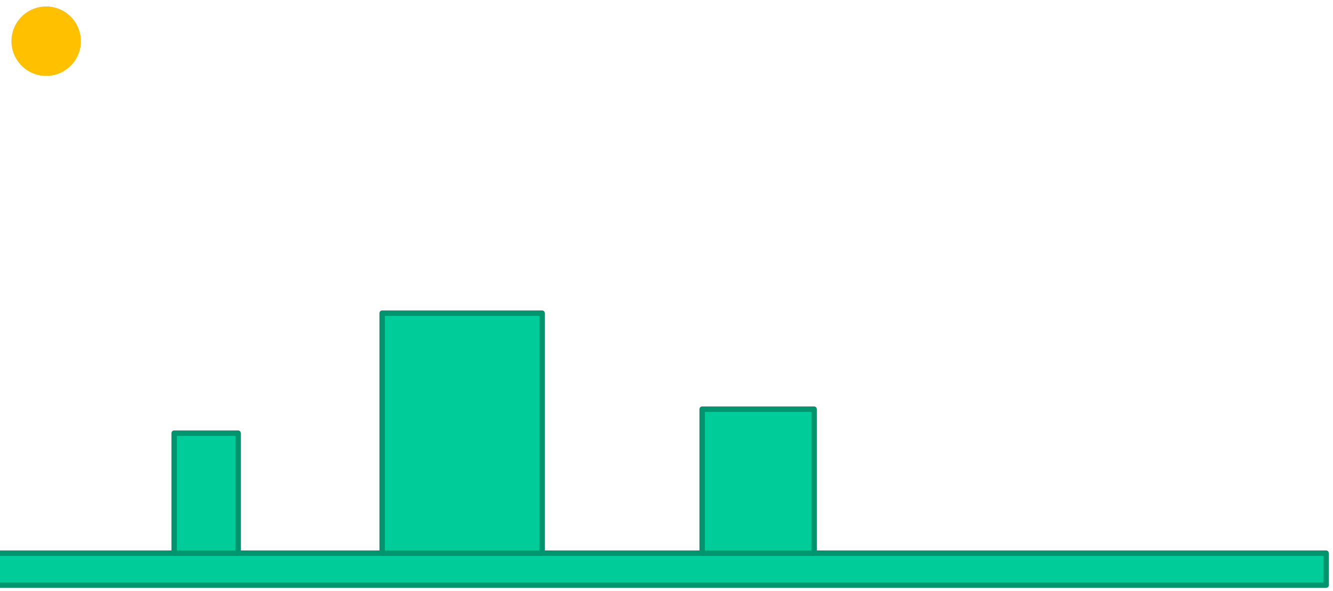
```
glEnable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureHandle);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, …);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, …);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, …);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, …);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, …);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, …);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

.. or use explicitly in the shader:

```
uniform samplerCube myCubeMap;
...
vec4 color = textureCube(myCubeMap, vec3(x, y, z));
```

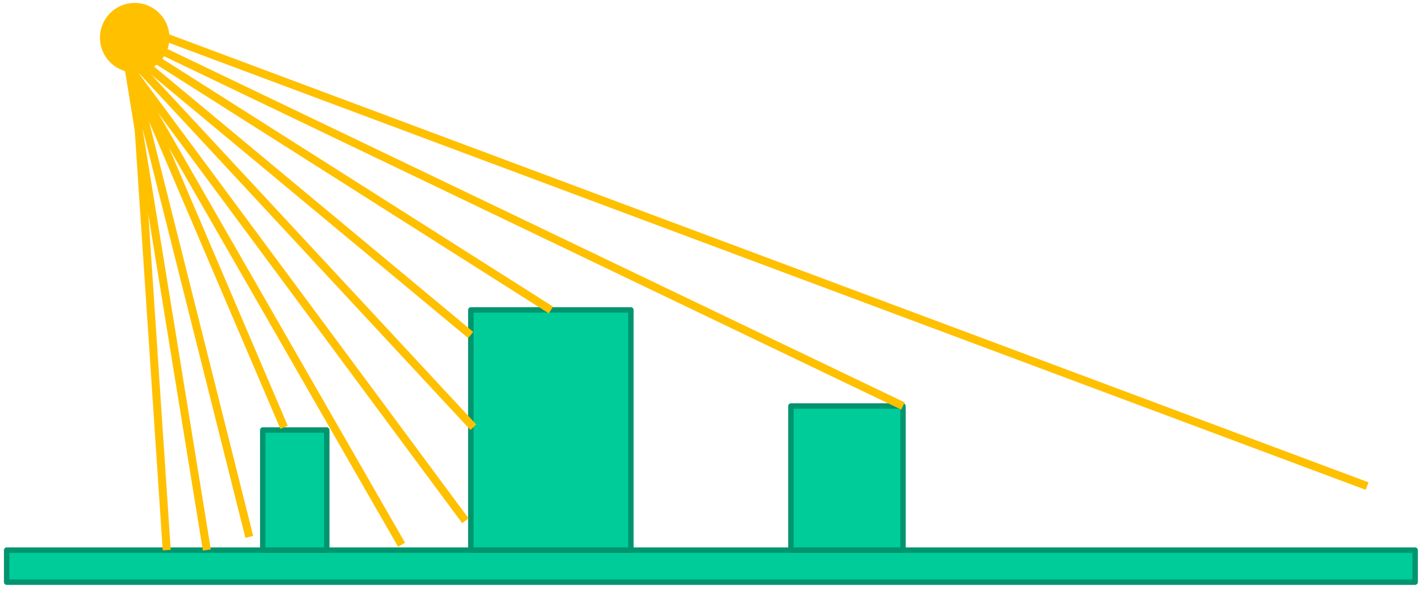
Shadow maps

Texture-based precomputation is a popular way of rendering shadows.



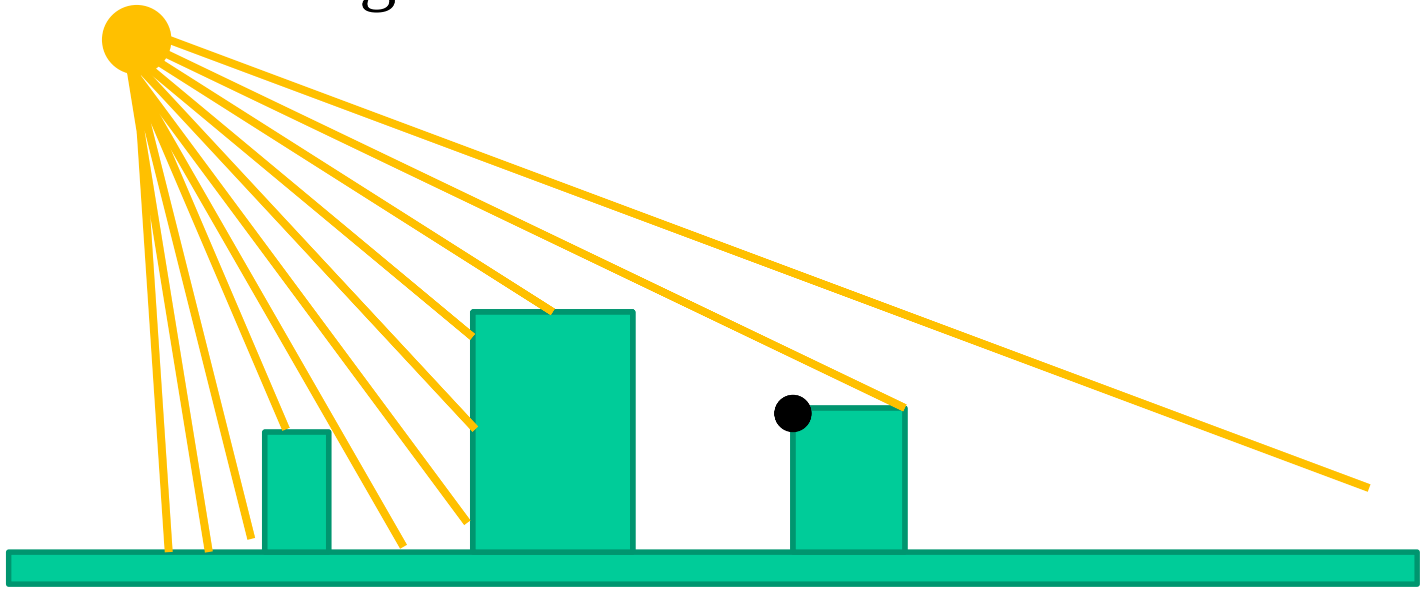
Shadow maps

For a given light source we can render a texture, that stores for every light ray its length until it hits an object.



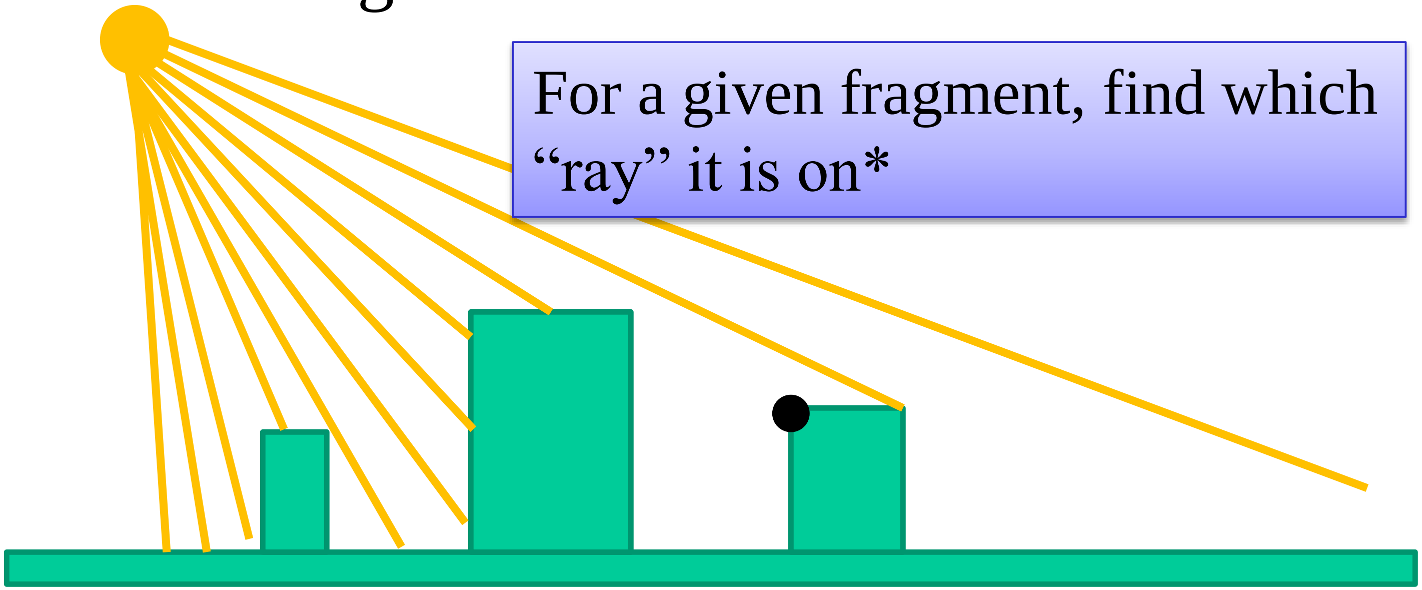
Shadow maps

This texture can then be used when rendering the scene, to test for each fragment, whether it is occluded from the light source.



Shadow maps

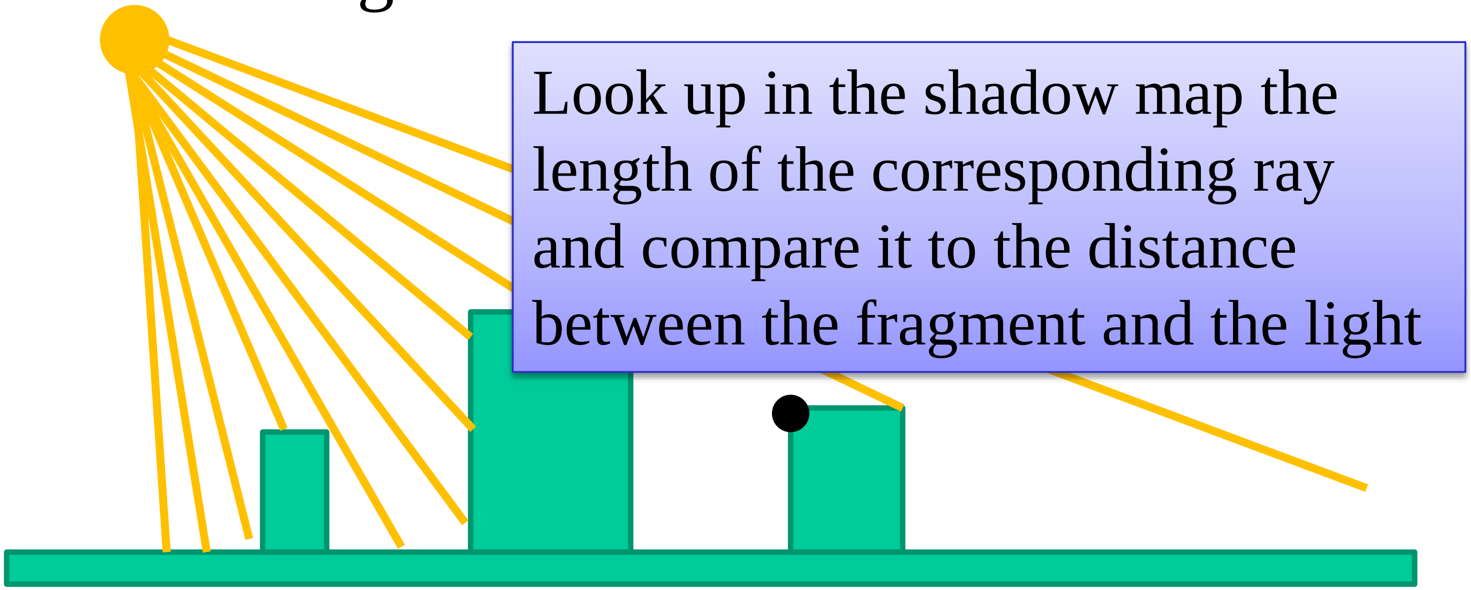
This texture can then be used when rendering the scene, to test for each fragment, whether it is occluded from the light source.



* More formally, find its coordinates in the light source's "projective frustum"

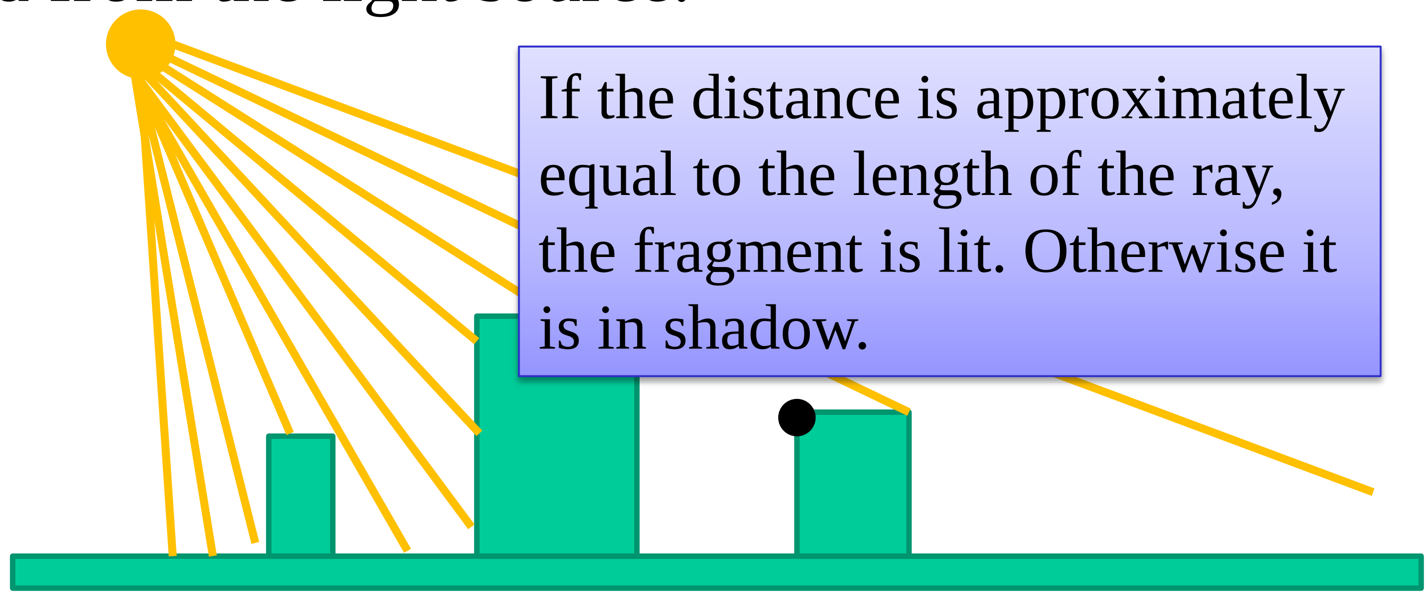
Shadow maps

This texture can then be used when rendering the scene, to test for each fragment, whether it is occluded from the light source.



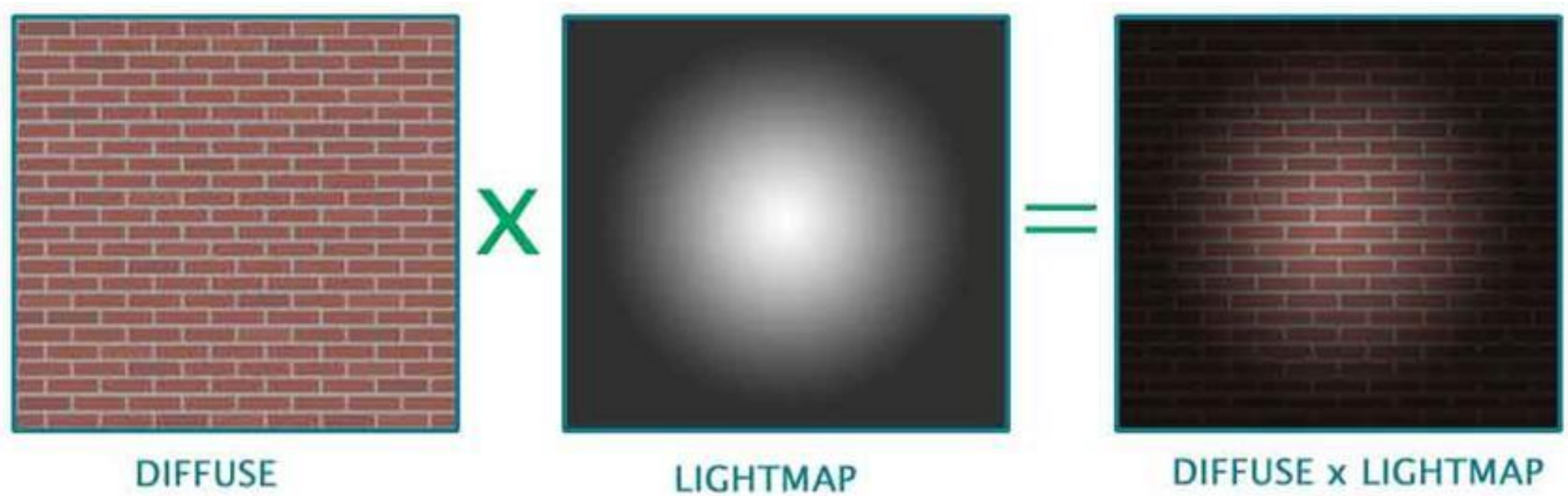
Shadow maps

This texture can then be used when rendering the scene, to test for each fragment, whether it is occluded from the light source.



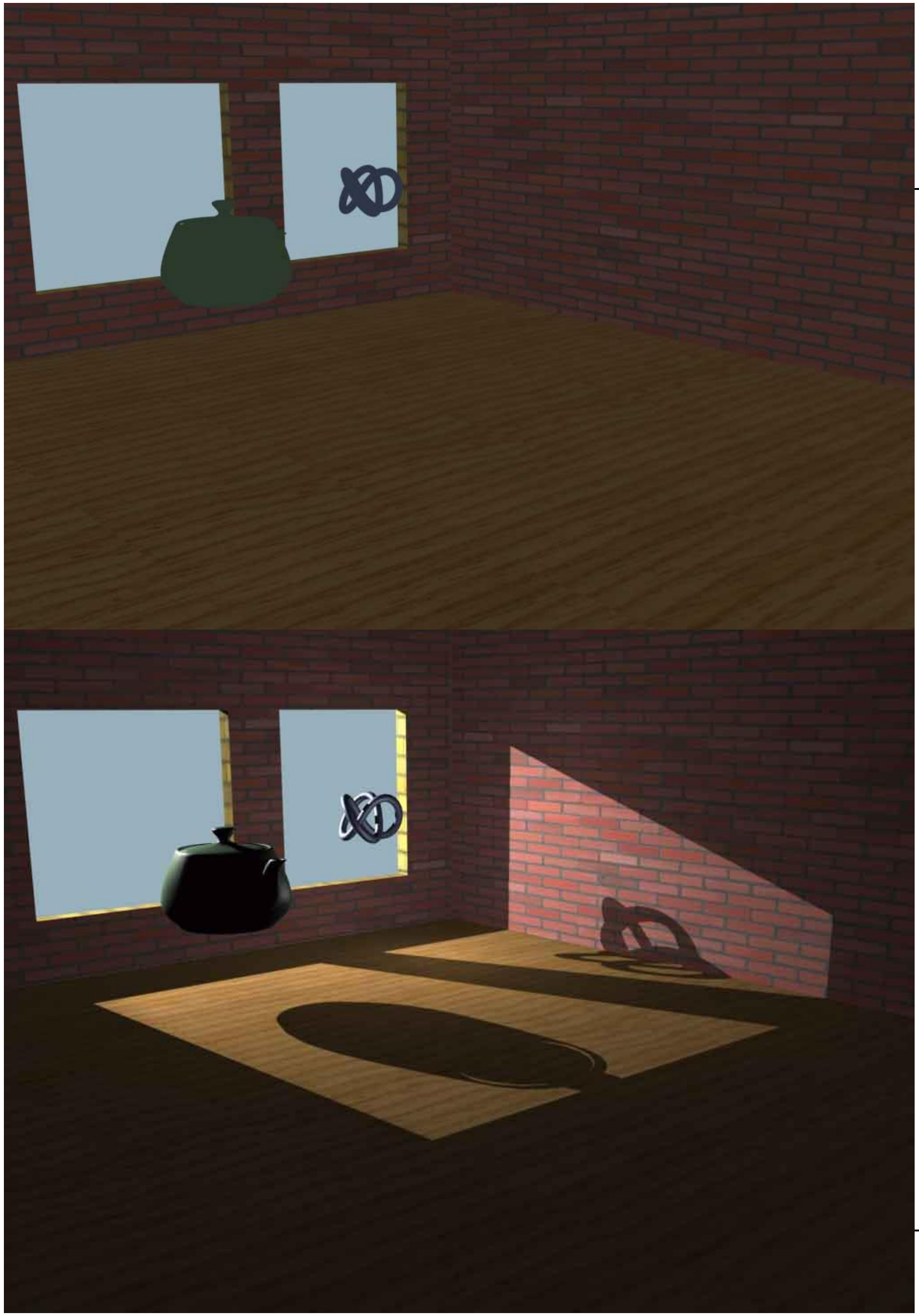
Light maps

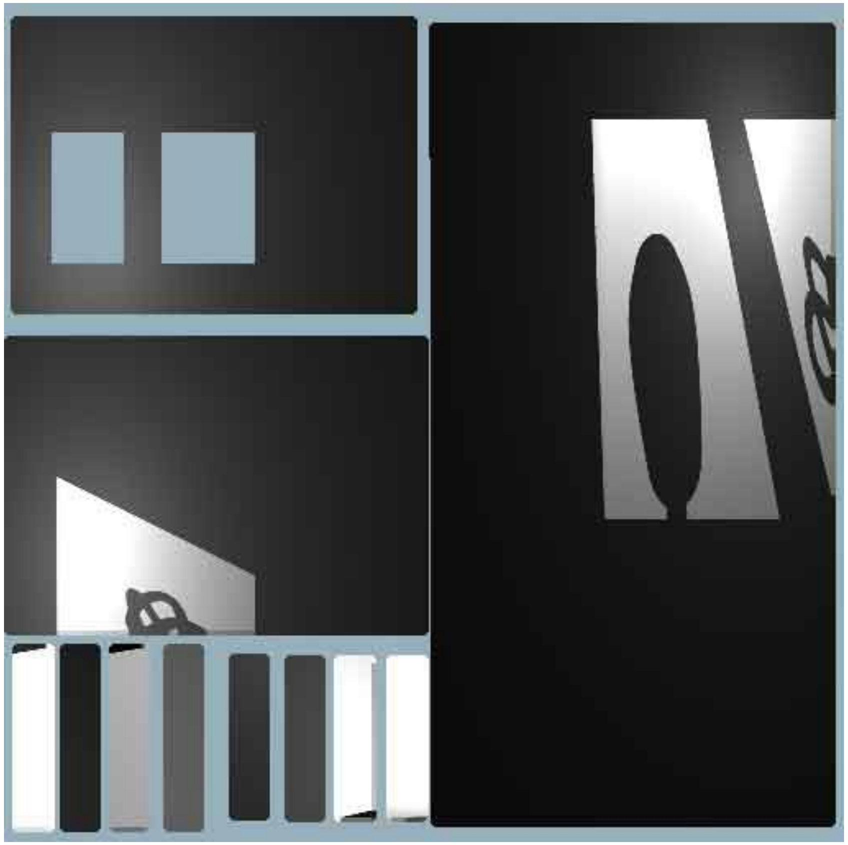
- Do not confuse the shadow map method with another straightforward approach to render shadows: simply precompute the texture for each object that takes lighting and shadows into account. This texture is called a **light map**.



http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/Theory/mappinglighting.htm

Light maps





http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/Theory/mappinglighting.htm

Texturing

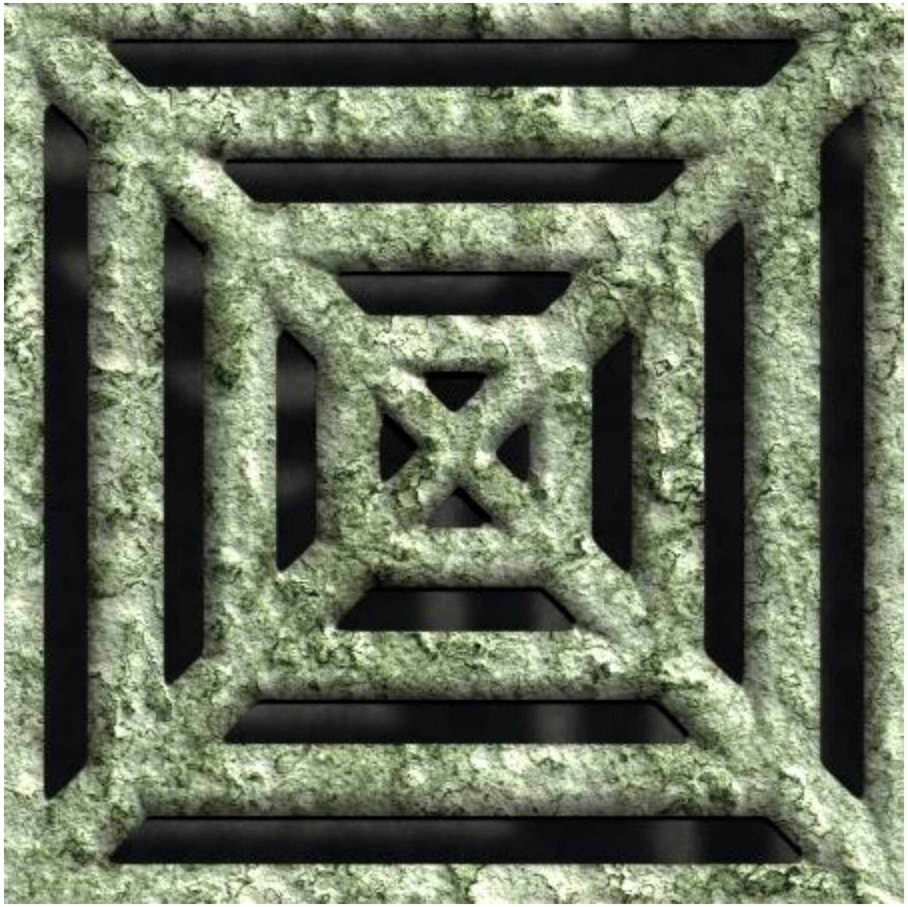
- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, environment maps, shadow maps
- Procedural textures

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, environment maps, shadow maps
- Procedural textures

Procedural textures

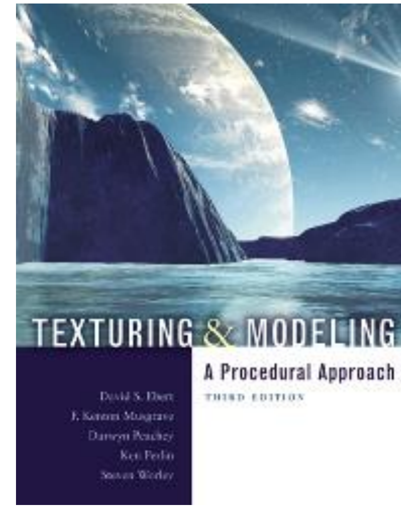
- Often it makes sense to generate texture images procedurally.
- Moreover, you may sometimes do it on the fly rather than read from a file.
- I.e. a "texture" in this case is simply a function f(x) or f(x,y) or f(x,y,z).

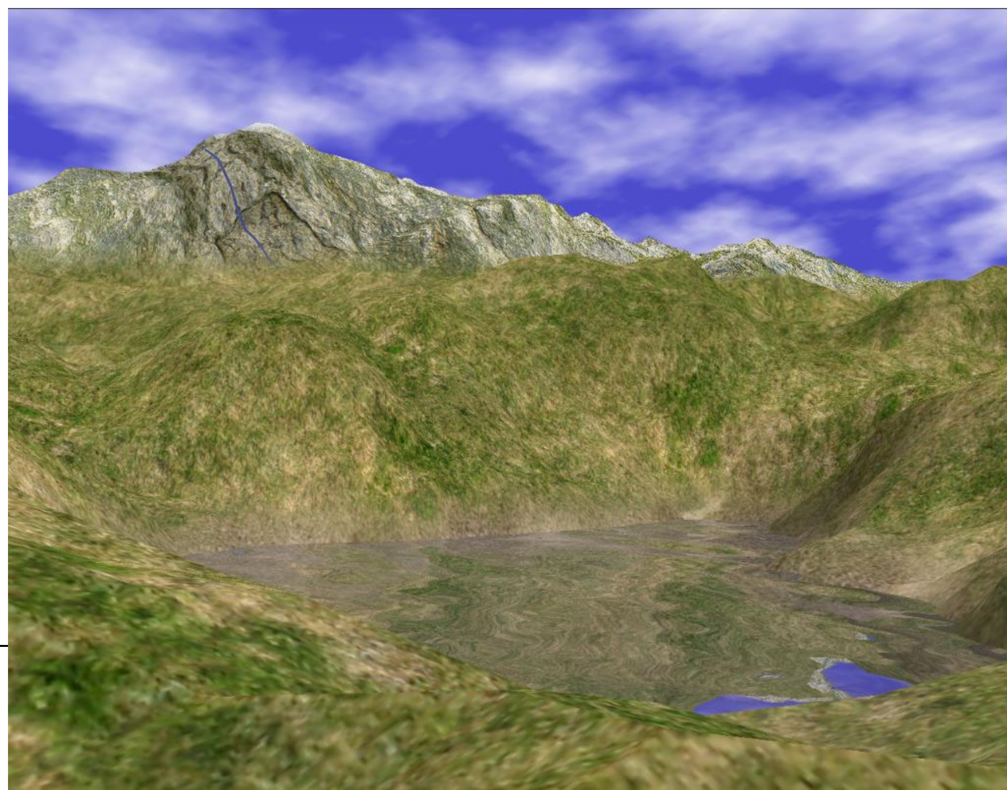


http://en.wikipedia.org/wiki/File:Procedural_Texture.jpg

Texture synthesis

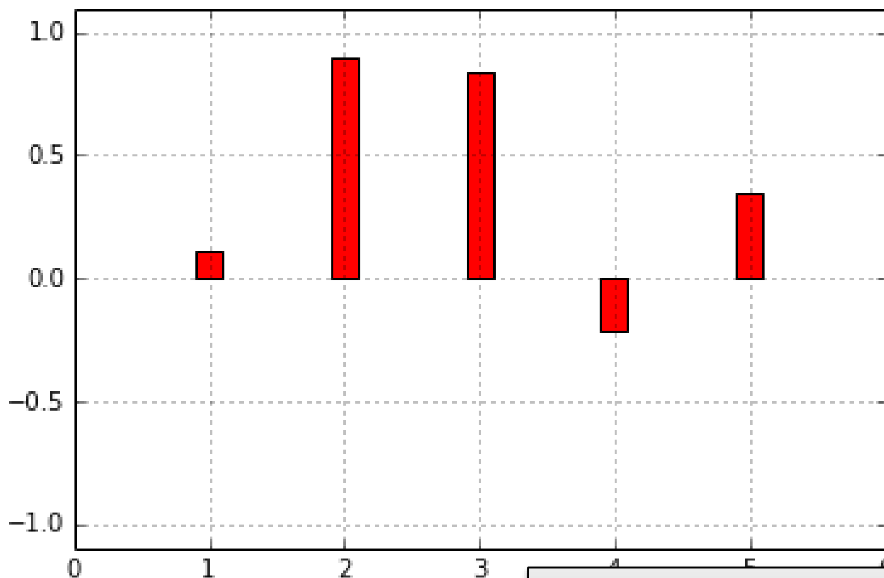
- Texture synthesis is a large field in itself, here we shall only cover one most relevant algorithm: **Perlin fractal noise**.





Basic 1D Perlin noise

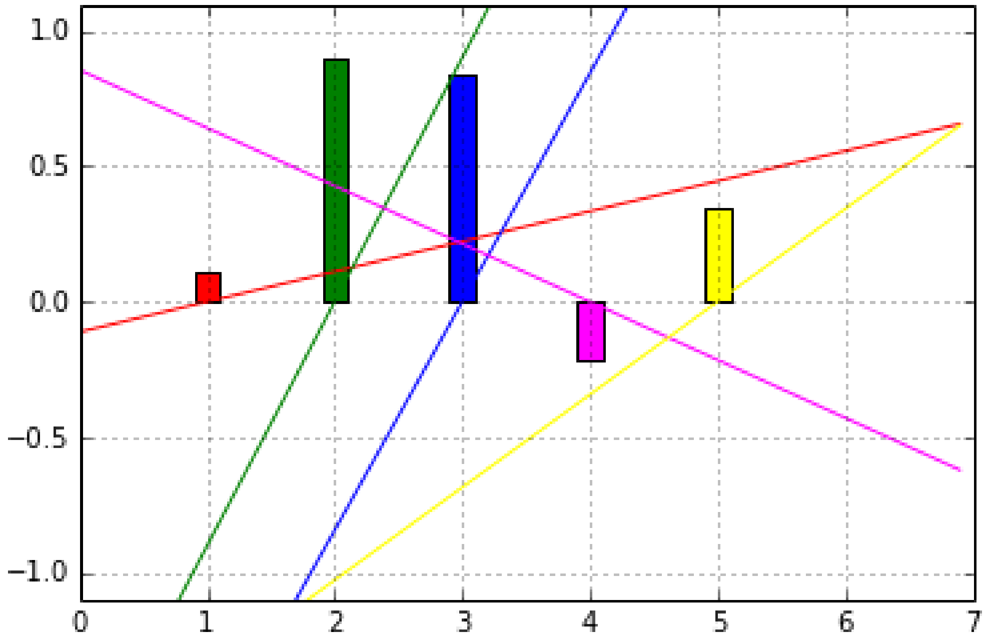
- Fix a function *f*, that can produce a random-looking float for a fixed integer input.




```
float my_rand(int n) {
    return cos(12345*n*n);
}
```

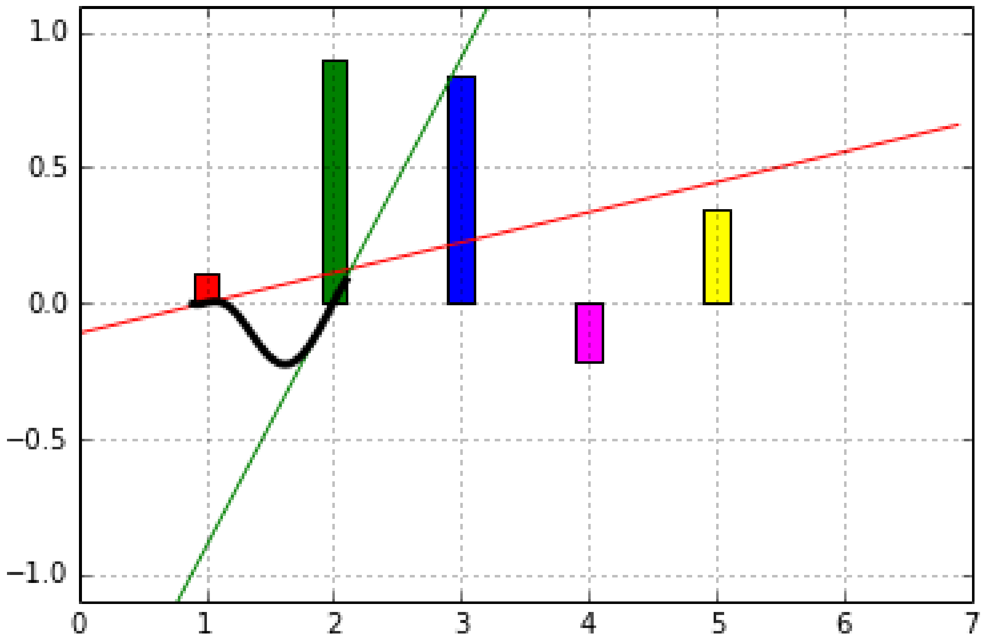
Basic 1D Perlin noise

- Use the value of f at each integer point n as the derivative of a linear function passing through $(n, 0)$:



Basic 1D Perlin noise

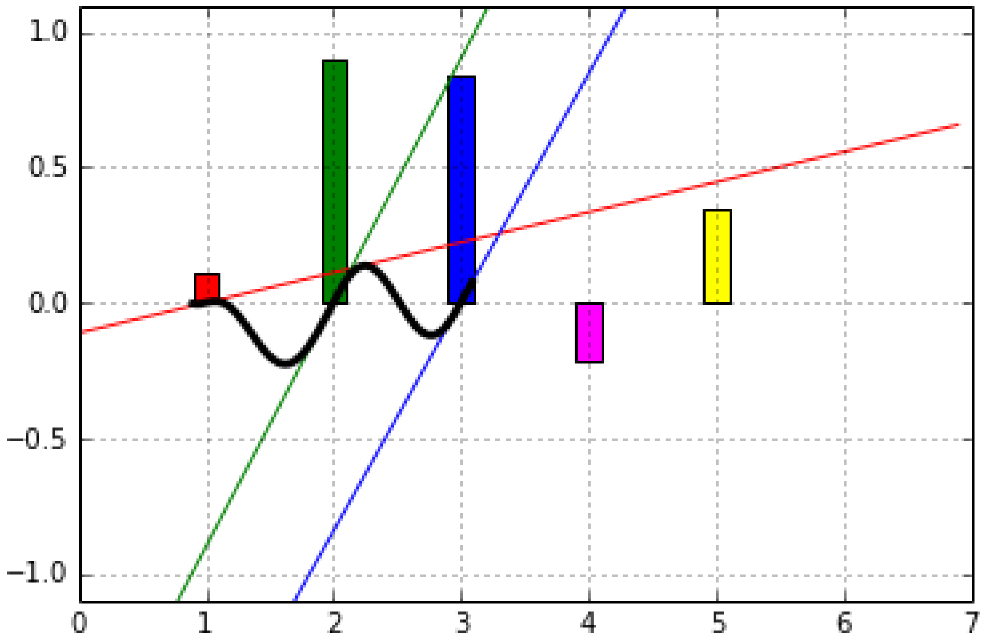
- For points between integer positions, blend between corresponding functions:




```
t = fade(frac(x))
return (1-t)*f1 + t*f2
```

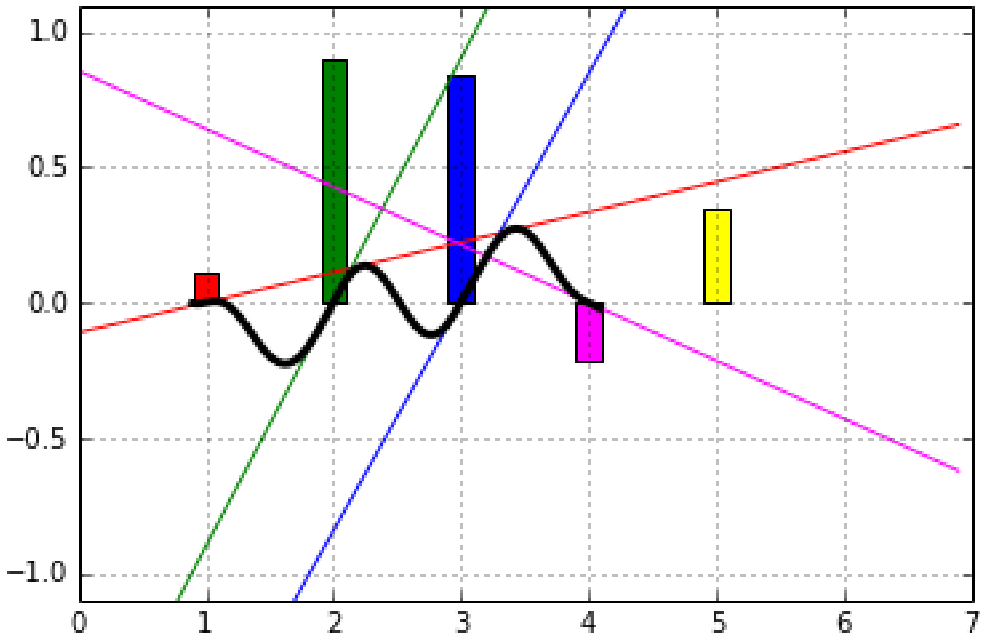
Basic 1D Perlin noise

- For points between integer positions, blend between corresponding functions:



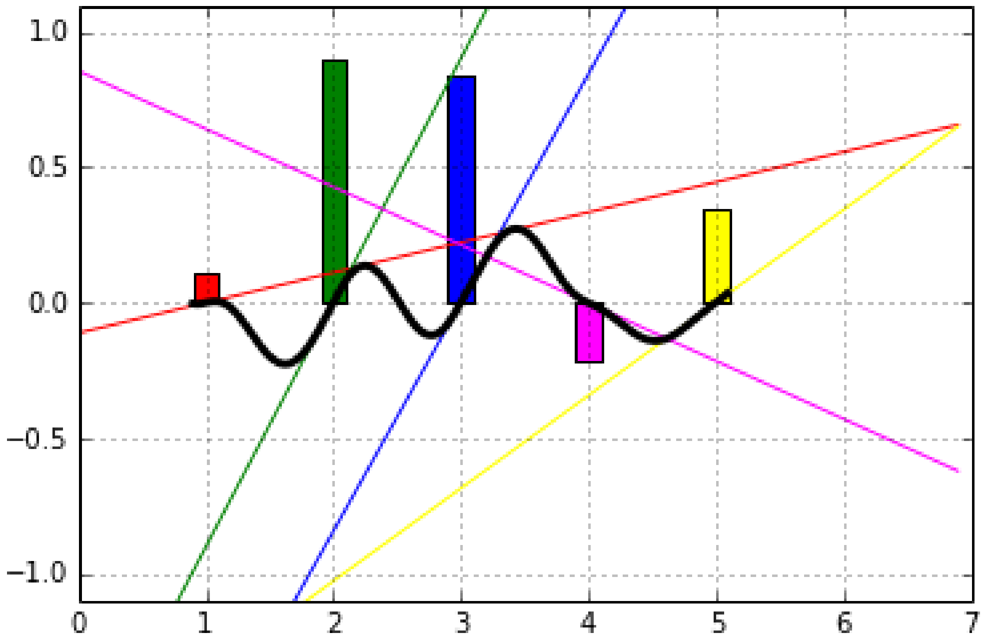
Basic 1D Perlin noise

- For points between integer positions, blend between corresponding functions:



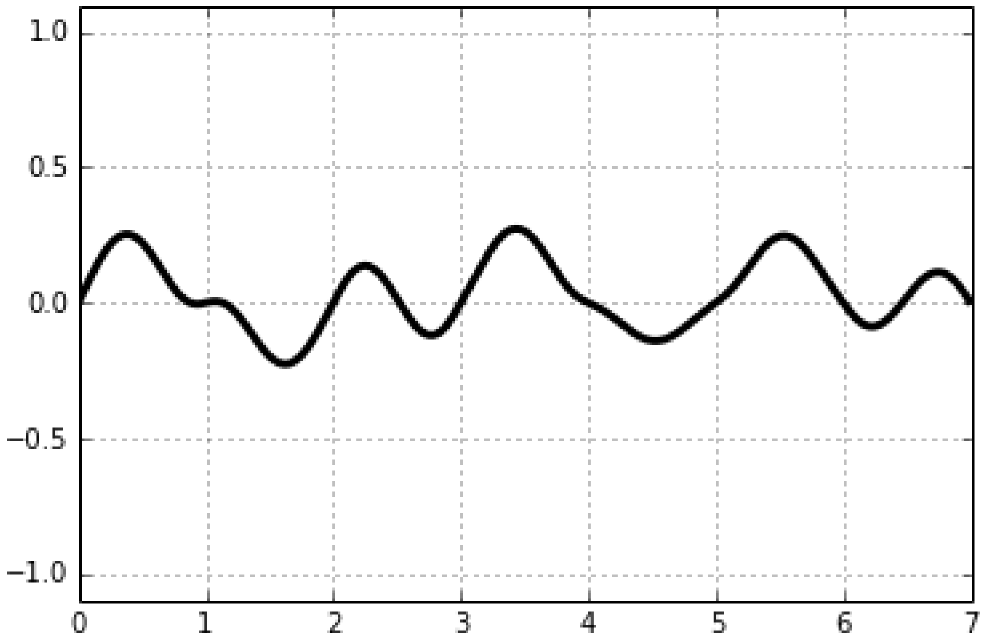
Basic 1D Perlin noise

- For points between integer positions, blend between corresponding functions:



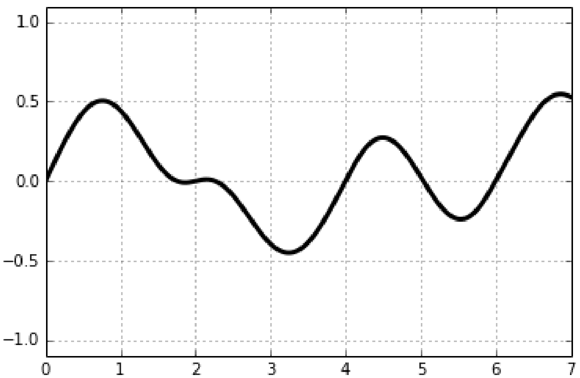
Basic 1D Perlin noise

- For points between integer positions, blend between corresponding functions:

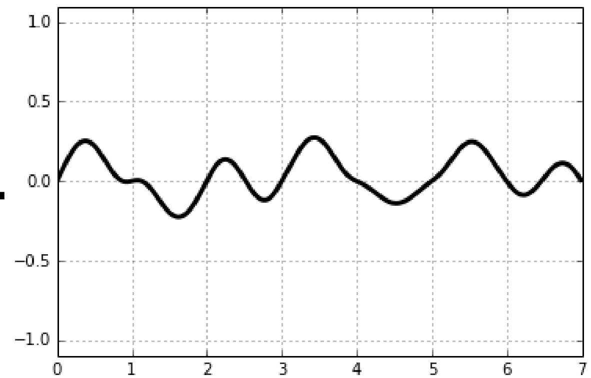


Fractal noise

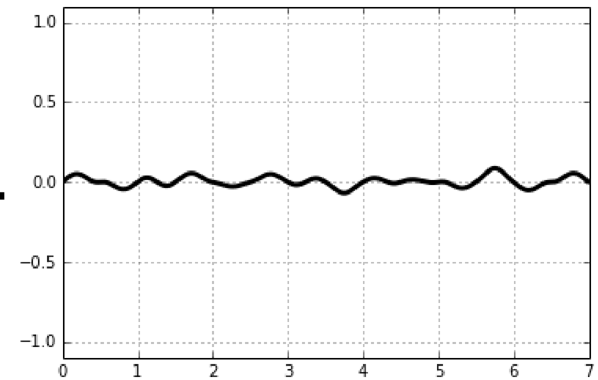
- Add noise generated at several scales.



+

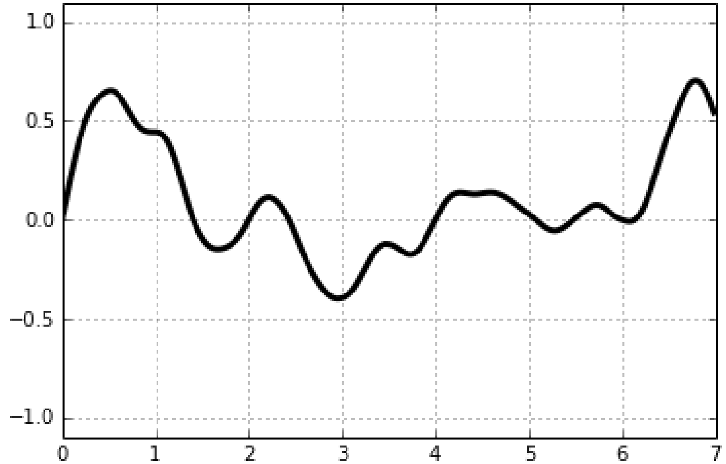


+



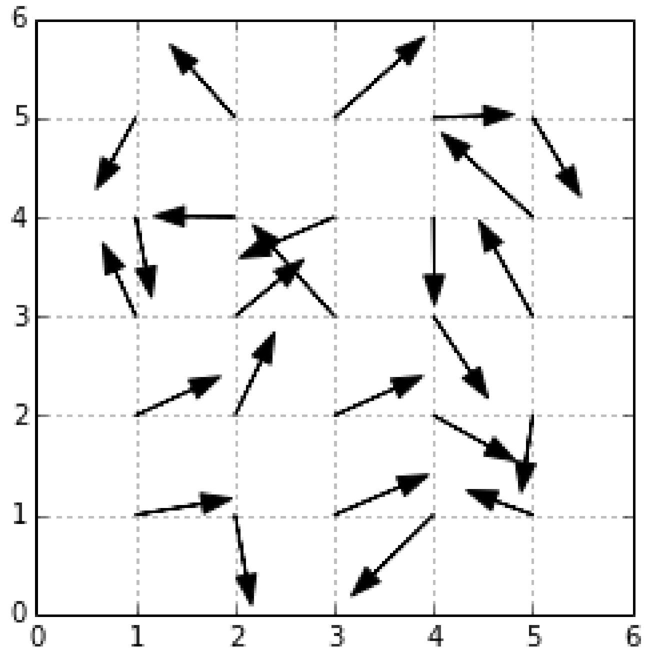
`2.0*perlin(0.5*x)` `1.0*perlin(1.0*x)` `0.2*perlin(2.0*x)`

=



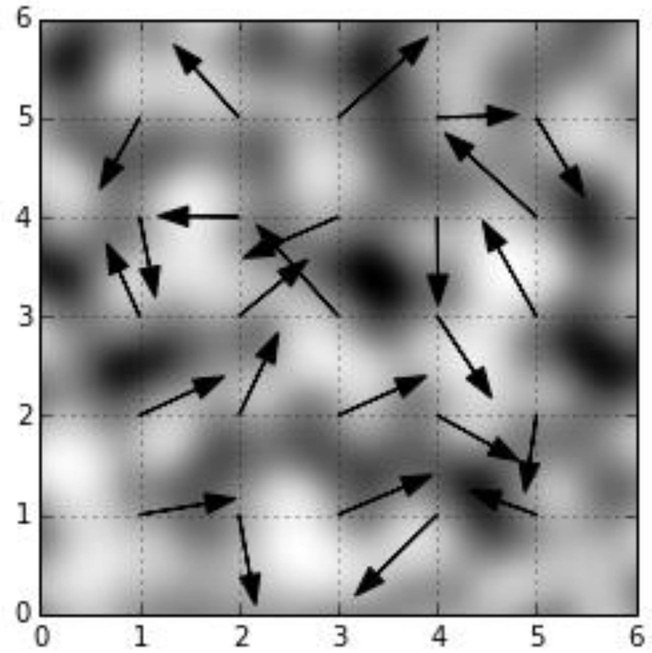
Multidimensional noise

- For 2D, 3D, etc, the logic is the same, but now we generate a **gradient vector** at each grid point.



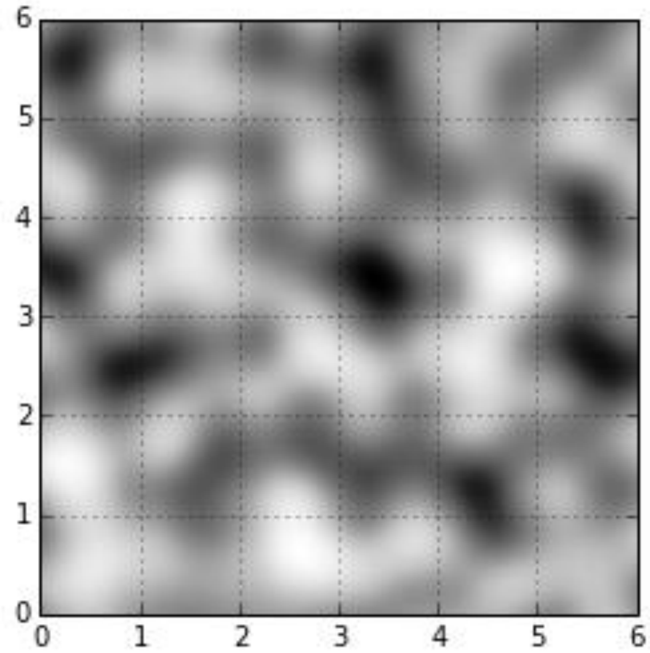
Multidimensional noise

- For 2D, 3D, etc, the logic is the same, but now we generate a **gradient vector** at each grid point.



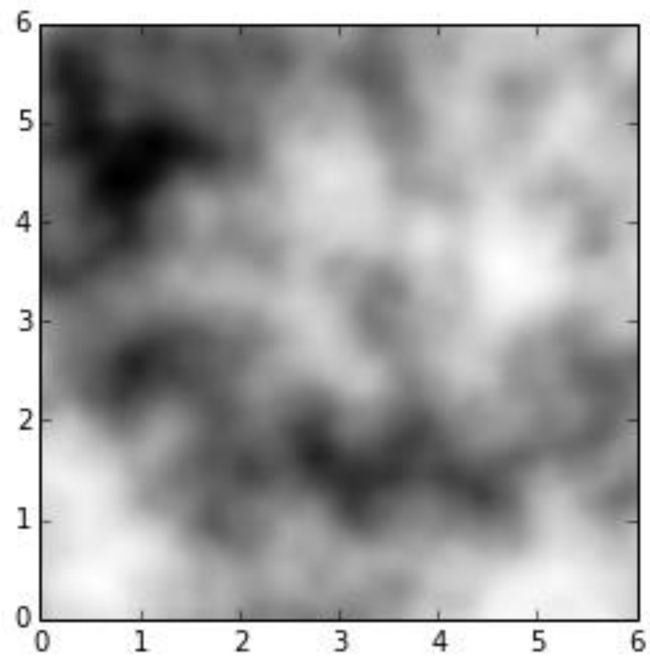
Multidimensional noise

- For 2D, 3D, etc, the logic is the same, but now we generate a **gradient vector** at each grid point.



Fractal multidimensional Perlin noise

- Same idea for generating fractal noise – add together noise at several scales:




```
2.0*perlin2d(0.3*x) + 1.0*perlin2d(x) +
                      0.3*perlin2d(2*x) + 0.1*perlin2d(4*x)
```

Texturing

- How attribute interpolation is *actually* done
- Texture filtering
  - Bilinear, mipmapping, anisotropic
- Textures & OpenGL
- Textures beyond images:
  - Precomputation & look-up tables
  - Normal maps, environment maps, shadow maps
- Procedural textures

Standard Graphics Pipeline

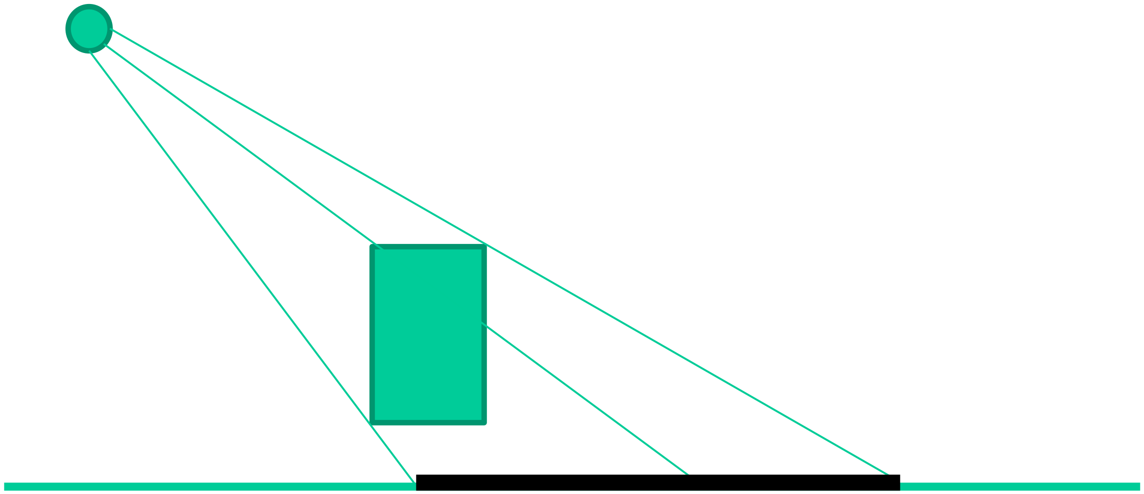
Vertex transform	**Determine clip-space position of a triangle**
Culling and clipping	**Determine whether the triangle is visible**
Rasterization	**Determine all pixels belonging to the triangle**
Fragment shading	**For each pixel, determine its color**
Visibility tests & blending	**Draw pixel *(if needed)***

Interlude: Shadows

- Light maps
- Shadow maps
- Projective shadows
- Stencil shadows

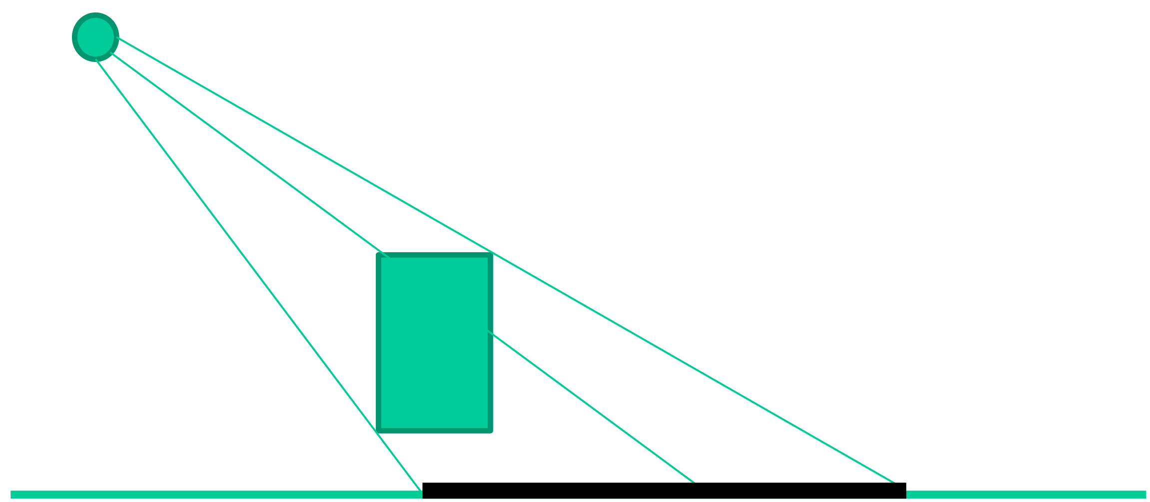
Projective shadows

- “Flatten” objects into their shadows via a projection transform from the light source onto a plane.



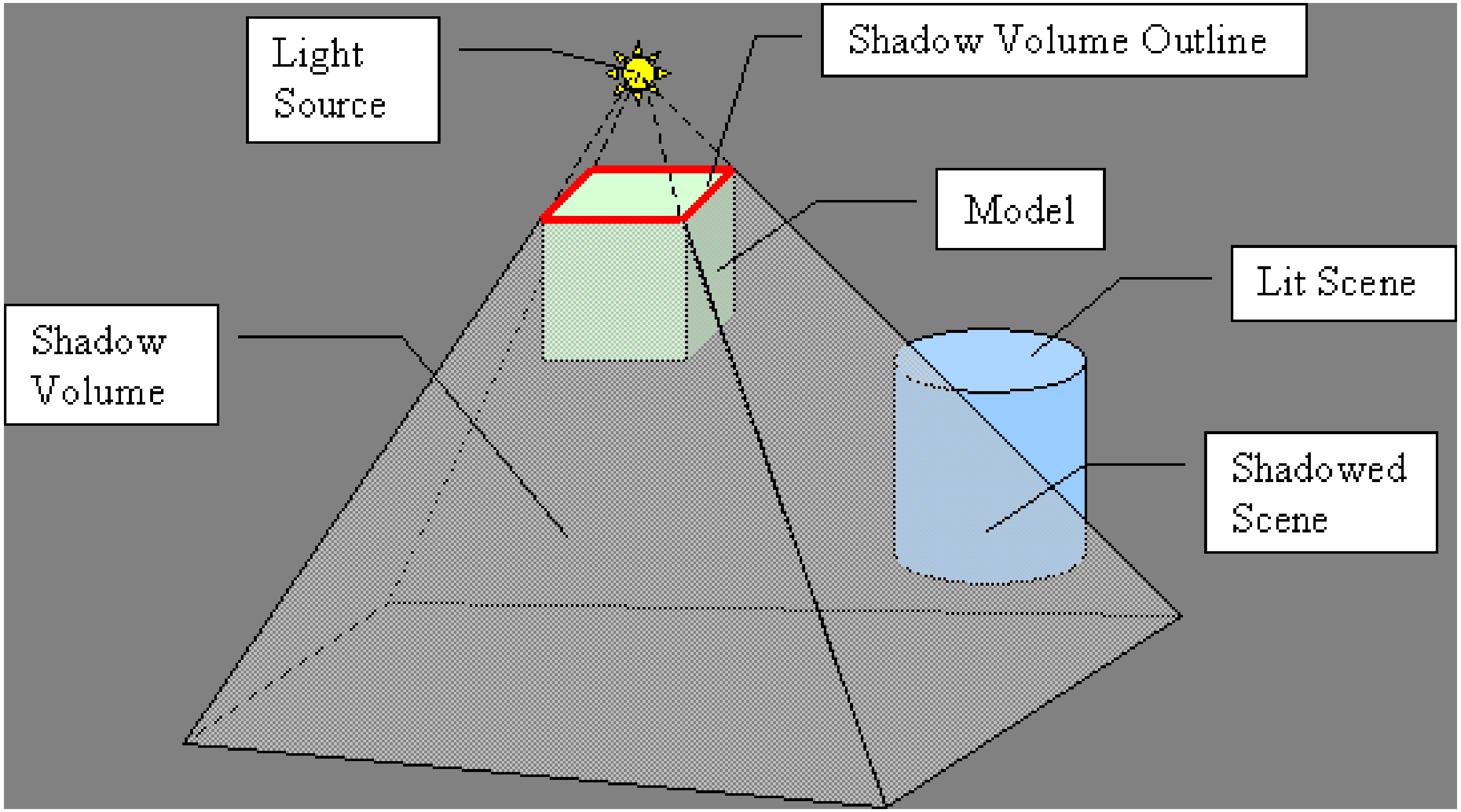
Projective shadows

```
draw_object();
set_shadow_color();
glPushMatrix();
    shadowProjectionMatrix(light, plane);
    draw_object();
glPopMatrix();
```



Stencil Shadows

Stencil Shadows



Stencil Shadows

- Idea:
  - For each pixel, check whether it is within a shadow volume.
  - This can be done by counting
    - How many front-facing shadow volume polygons are in front of the pixel
    - How many back-facing shadow volume polygons are in front of the pixel
  - If the numbers do not match, the pixel is in shadow

Stencil Shadows

- Implementation:
  - First render the scene as if it was all in shadow.
  - Render the front-facing shadow volume polygons, incrementing the stencil buffer for each pixel where the depth test passes.
  - Render back-facing shadow volume polygons, decrementing the stencil buffer where the depth test passes.
  - Now whenever stencil buffer is 0, render the scene as fully lit.

http://en.wikipedia.org/wiki/Shadow_volume