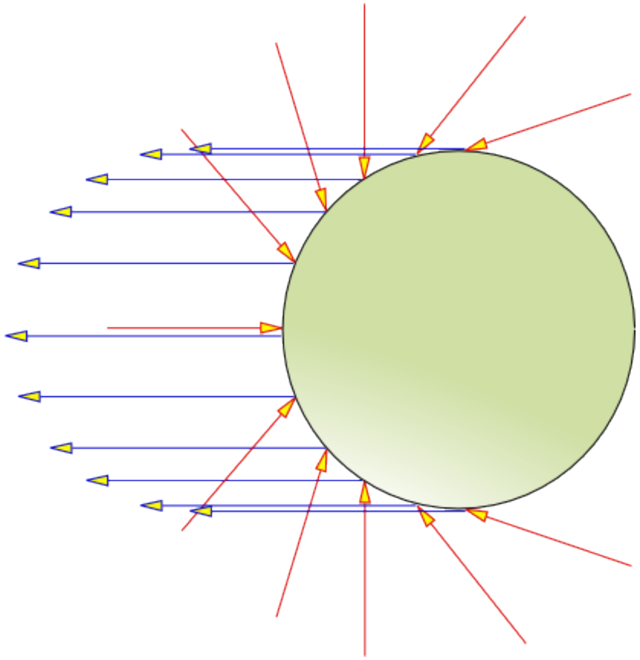
Environment Mapping Steps

- Generate or load a 2D texture that depicts the environment
- For every pixel of the reflected object...
  1. Calculate the normal **n**
  2. Calculate a reflection vector **r** from **n** and the view vector **v**
  3. Calculate texture coordinates (*u*,*v*) from **r**
  4. Color the pixel with the texture value
- The problem: how does one parameterize the space of the reflection vectors?
  - I.e.: how does one map spatial directions onto [0,1]x[0,1]?
- Desired Characteristics:
  - Uniform sampling (number of texels per solid angle should be "as constant as possible" in all directions)
  - View-independent ⟶ only one texture for all camera positions
  - Hardware support (texture coordinates should just be easy to generate)

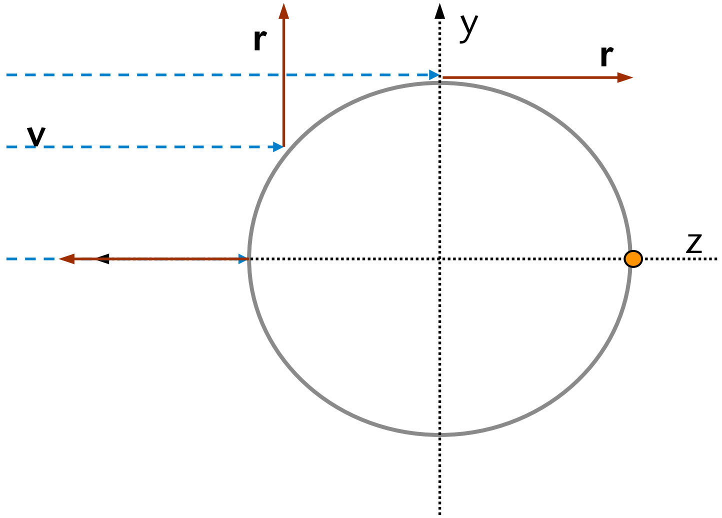
Spherical Environment Mapping

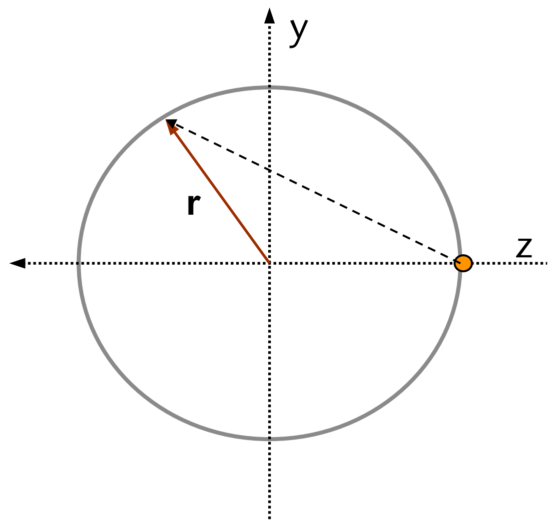
- Generating the environment map (= texture):
  - Photography of a reflective sphere; or
  - Ray tracing of the scene with all primary rays being reflected at a perfectly reflective sphere



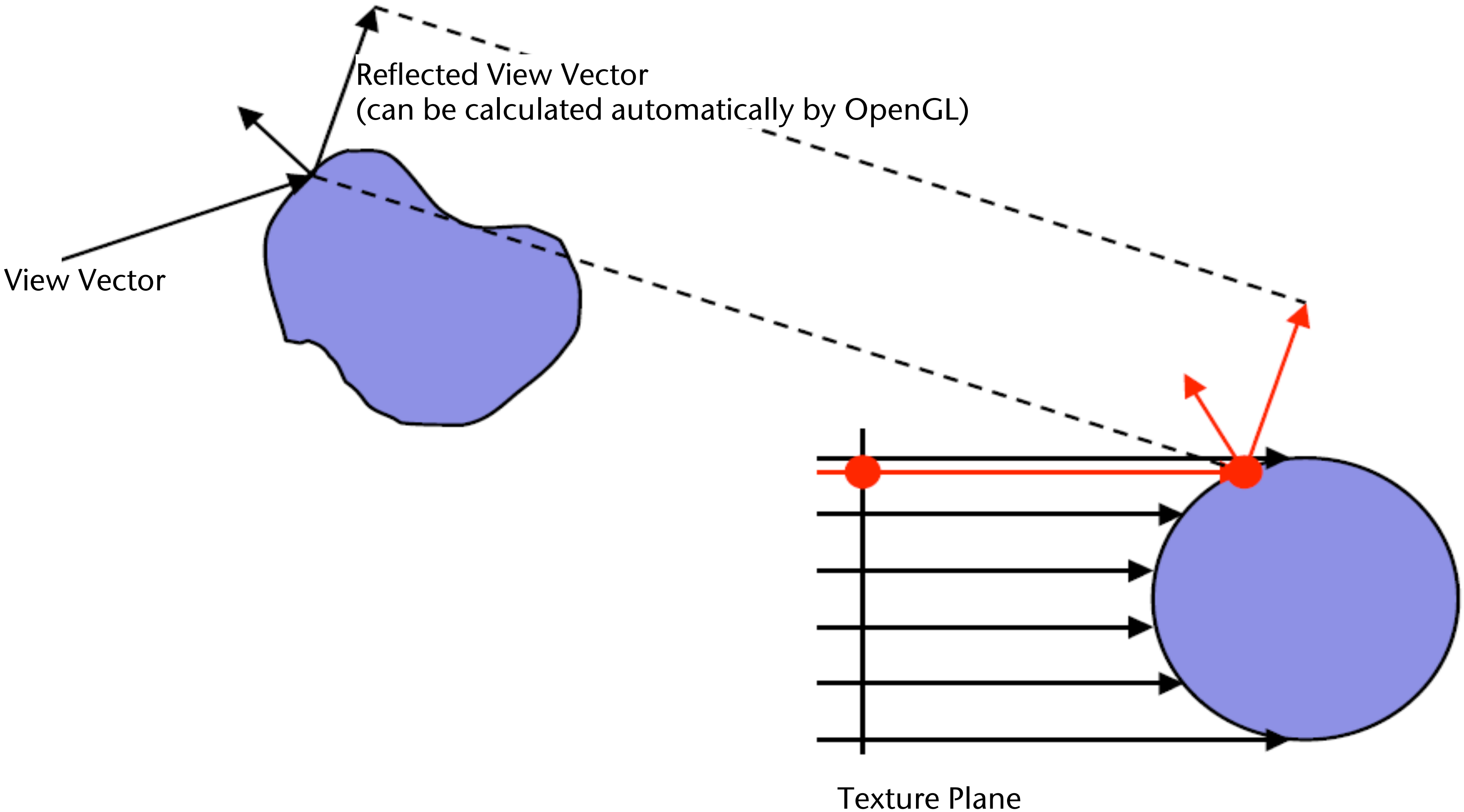
- Mapping of the directional vector **r** onto (u,v):
  - The sphere map contains (theoretically) a color value for every direction, except **r** = (0, 0, -1)
  - Mapping:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \frac{r_x}{\sqrt{r_x^2+r_y^2+(r_z+1)^2}} + 1 \\ \frac{r_y}{\sqrt{r_x^2+r_y^2+(r_z+1)^2}} + 1 \end{pmatrix}$$



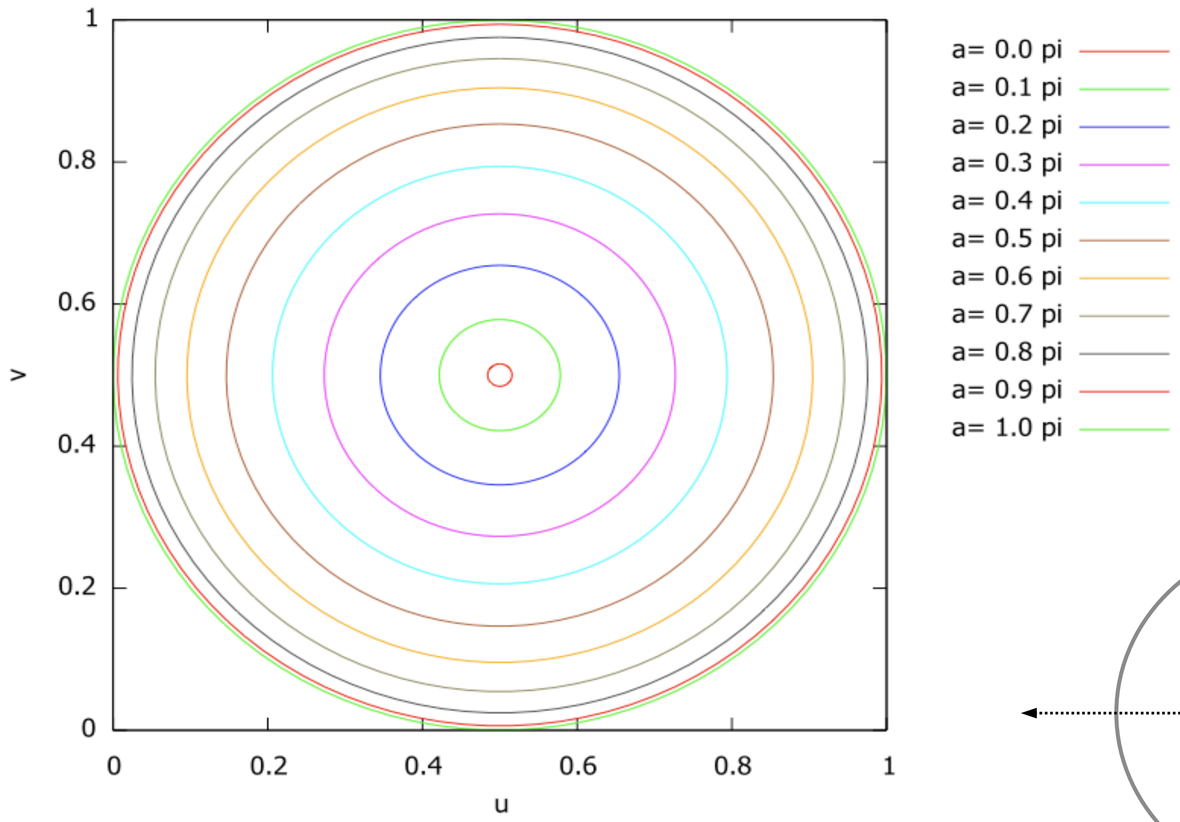


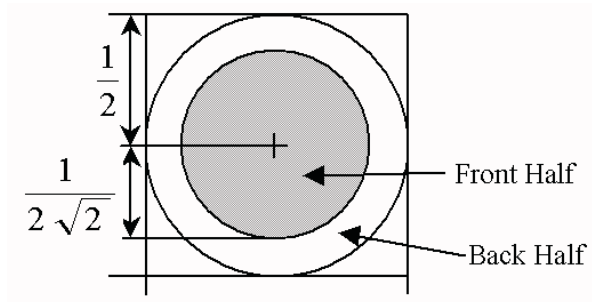
- Application of the sphere mapping to texturing:

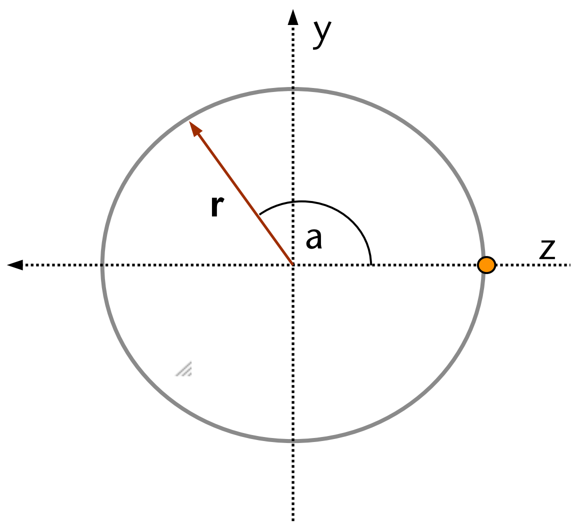


Simple Example

- Unfortunately, the mapping/sampling is not very uniform:

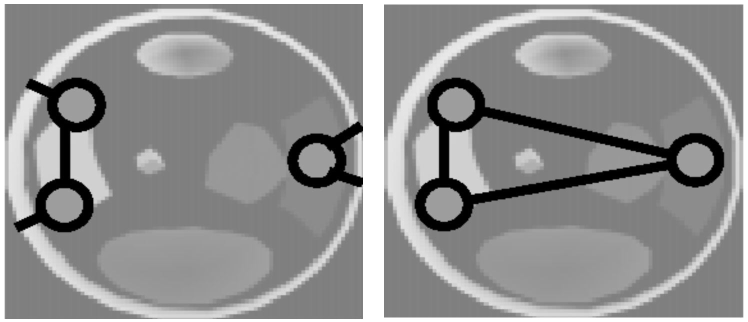




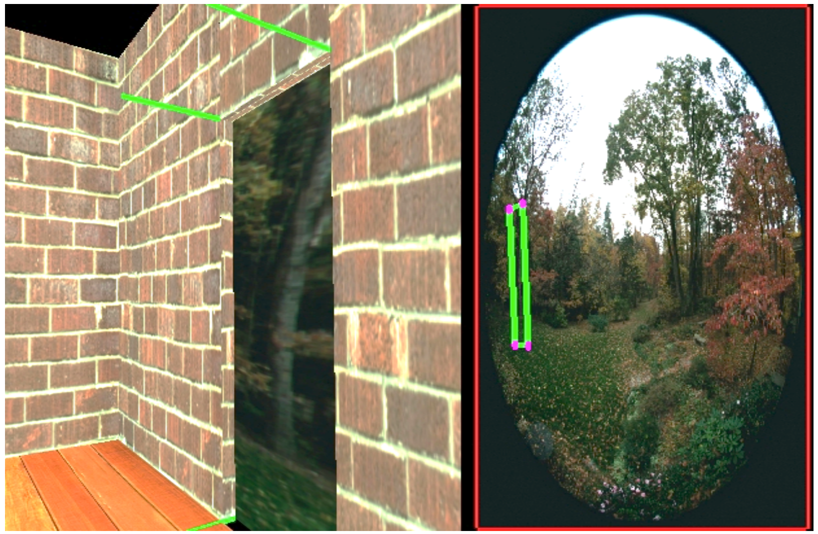


- Texture coords are interpolated incorrectly:
  - Texture coords are interpolated linearly (by the rasterizer), but the sphere map is non-linear
  - Long polygons can cause serious "bends" in the texture
  - Sometimes, incorrect wrap-arounds occur with interpolated texture coords
  - *Sparkles / speckles* if the reflecting vector comes close to the edge of the texture (through aliasing and "wrap-around")

Intended/ correct wrap through the sphere perimenter



2D texturing hardware doesn't know about sphere maps, it just linearly interpolates texture coords



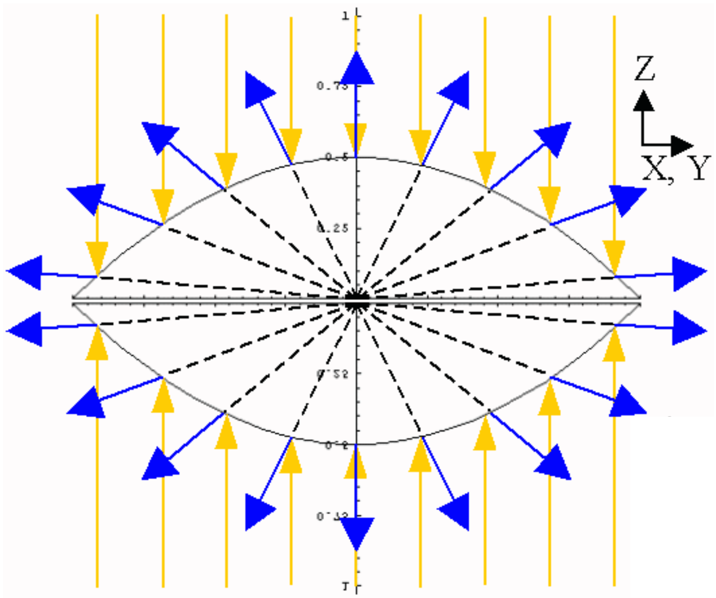
- Other cons:
  - Textures are difficult to generate by program
  - *Viewpoint dependent*: the center of the spherical texture map represents the vector that goes directly back to the viewer!
    - Can be made *view independent* with some OpenGL extensions
- Pros:
  - Easy to generate texture coordinates
  - Supported in OpenGL

A Piece of Artwork

Reflective balls in the main street of Adelaide, Australia

Dual Parabolic Environment Mapping

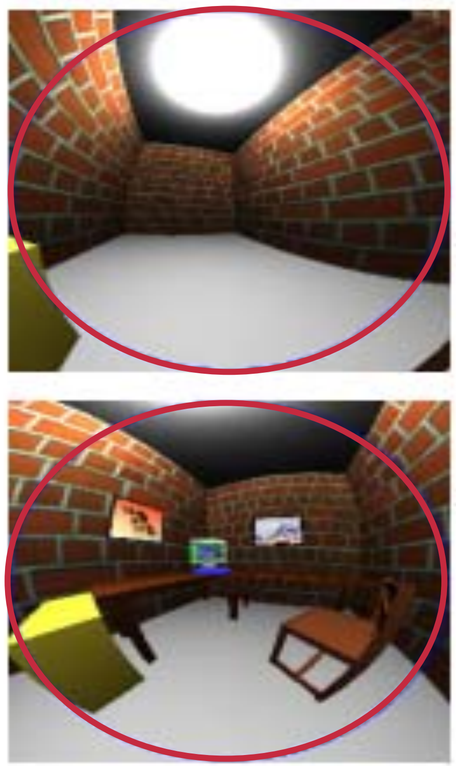
- Idea:
  - Map the environment onto two textures via a reflective double paraboloid
  - Pros:
    - Relatively uniform sampling
    - *View independent*
    - Relatively simple computation of texture coordinates
    - Also works in OpenGL
    - Also works in a single rendering pass (just needs multi-texturing)
  - Cons:
    - Produces artifacts when interpolating across the edge



- Images of the environment (= directional vectors) are still discs (as with the sphere map)
- Comparison:

Back

Front

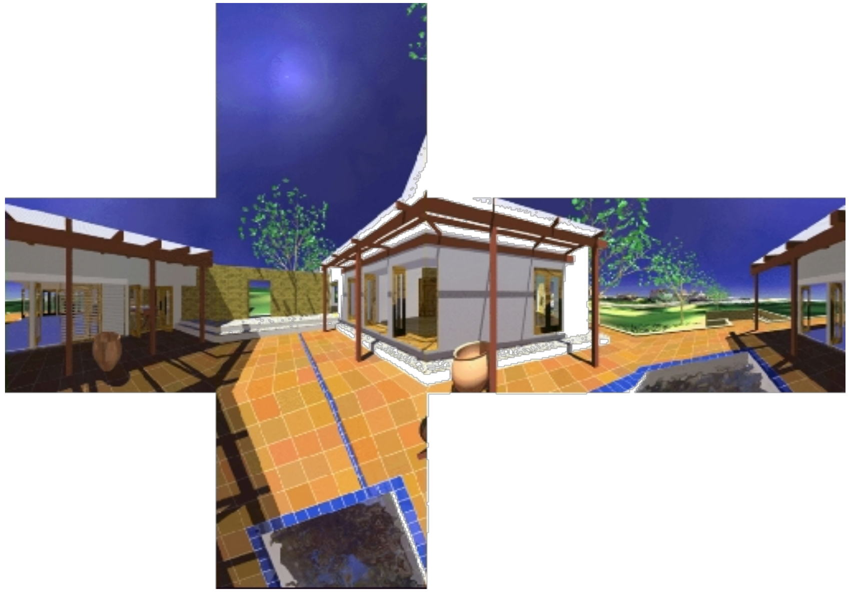


Parabolic environment map

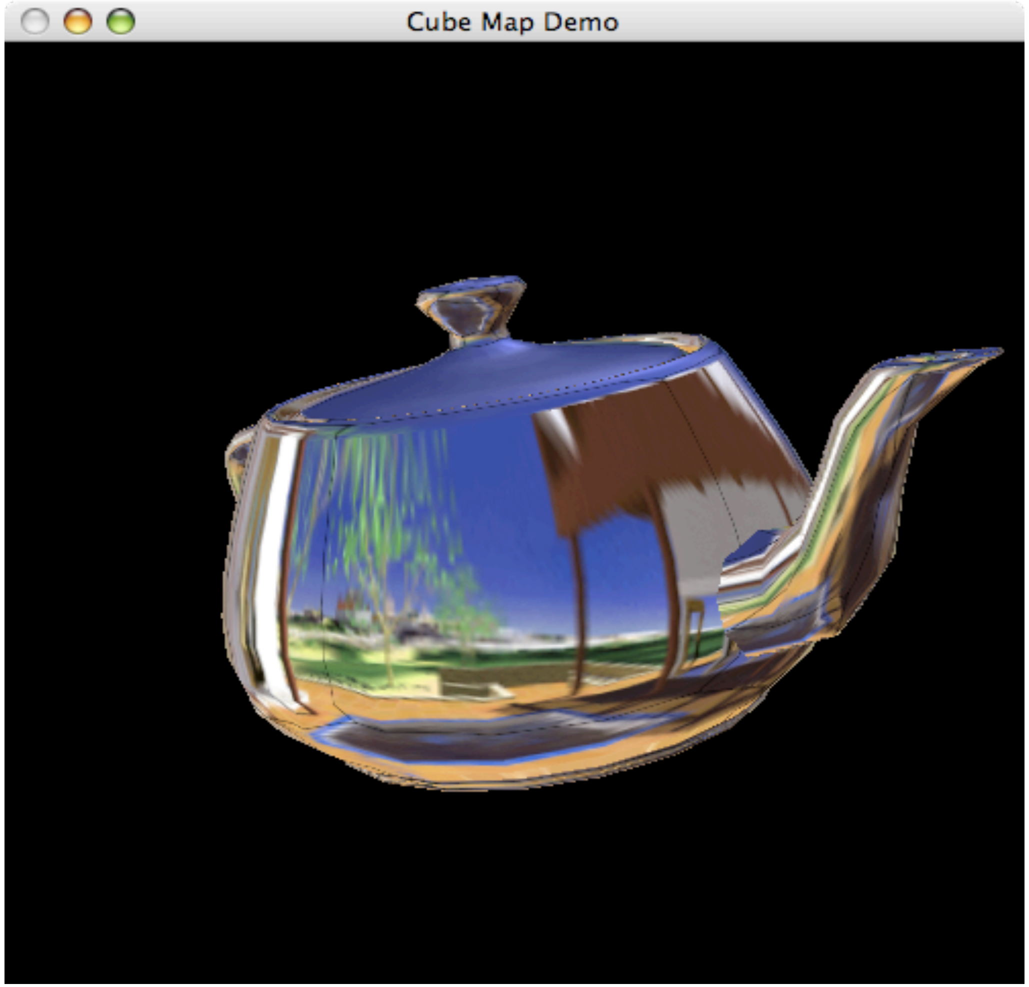
Result

Cubic Environment Mapping

- As before with the "normal" cube maps
- Only difference: use the reflected vector **r** for the calculation of the texture coordinates
- This reflected vector can be automatically calculated by OpenGL for each vertex (**`GL_REFLECTION_MAP`**)



Demo with Static Environment



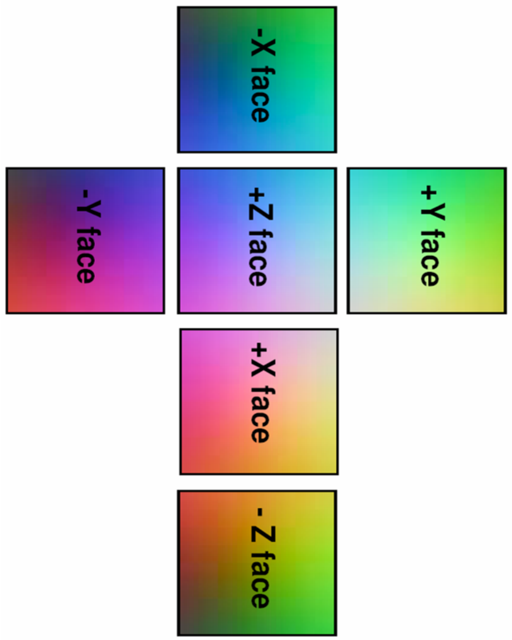
Cube Maps as LUT for Directional Functions

- Further application: one can also use a cube map to store any function of direction! (as a precomputed lookup table)
- Example: normalization of a vector
  - Every cube map texel (s,t,r) stores this vector

$$\frac{(s,t,r)}{\|(s,t,r)\|}$$

  in its RGB channels

  - Now one can specify any texture coordinates using **`glTexCoord3f()`** and receives the normalized vector
- Warning: when using this technique, one should turn off filtering



Dynamic Environment Maps

- Until now: environment map was invalid as soon as something in the environmental scene had changed!
- Idea:
  - Render the scene from the "midpoint" outward (typically 6x for cube map)
  - Transfer framebuffer to texture (using the appropriate mapping)
  - Render the scene again from the viewpoint outward, this time with environment mapping
- ➢ Multi-pass rendering
- Typically used with cube env maps

Dynamic Environment Mapping in OpenGL Using Cube Maps

```
GLuint cm_size = 512;    // texture resolution of each face
GLfloat cm_dir[6][3];    // direction vectors
float dir[6][3] = {
    1.0, 0.0, 0.0,       // right
   -1.0, 0.0, 0.0,       // left
    0.0, 0.0, -1.0,      // bottom
    0.0, 0.0, 1.0,       // top
    0.0, 1.0, 0.0,       // back
    0.0, -1.0, 0.0       // front
};
GLfloat cm_up[6][3] =    // up vectors
{   0.0, -1.0,  0.0,     // +x
    0.0, -1.0,  0.0,     // -x
    0.0, -1.0,  0.0,     // +y
    0.0, -1.0,  0.0,     // -y
    0.0,  0.0,  1.0,     // +z
    0.0,  0.0, -1.0      // -z
};
GLfloat cm_center[3];    // viewpoint / center of gravity
GLenum cm_face[6] = {
    GL_TEXTURE_CUBE_MAP_POSITIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
};
// define cube map's center cm_center[] = center of object
// (in which scene has to be reflected)
...
```

```
// set up cube map's view directions in correct order
for ( uint i = 0, i < 6; i + )
  for ( uint j = 0, j < 3; j + )
        cm_dir[i][j] = cm_center[j] + dir[i][j];

// render the 6 perspective views (first 6 render passes)
for ( unsigned int i = 0; i < 6; i ++ )
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glViewport( 0, 0, cm_size,  cm_size );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective( 90.0, 1.0, 0.1, ... );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
  gluLookAt( cm_center[0], cm_center[1], cm_center[2],
             cm_dir[i][0], cm_dir[i][1], cm_dir[i][2],
             cm_up[i][0],  cm_up[i][1],  cm_up[i][2] );
  // render scene to be reflected
  ...
  // read-back into corresponding texture map
  glCopyTexImage2D( cm_face[i], 0, GL_RGB, 0, 0, cm_size, cm_size, 0 );
}
```

```
// cube map texture parameters init
glTexEnvf(  GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexParameterf( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );
glTexGeni( GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );

// enable texture mapping and automatic texture coordinate generation
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_GEN_R );
glEnable( GL_TEXTURE_CUBE_MAP );

// render object in 7th pass ( in which scene has to be reflected )
...

// disable texture mapping and automatic texture coordinate generation
glDisable( GL_TEXTURE_CUBE_MAP );
glDisable( GL_TEXTURE_GEN_S );
glDisable( GL_TEXTURE_GEN_T );
glDisable( GL_TEXTURE_GEN_R );
```

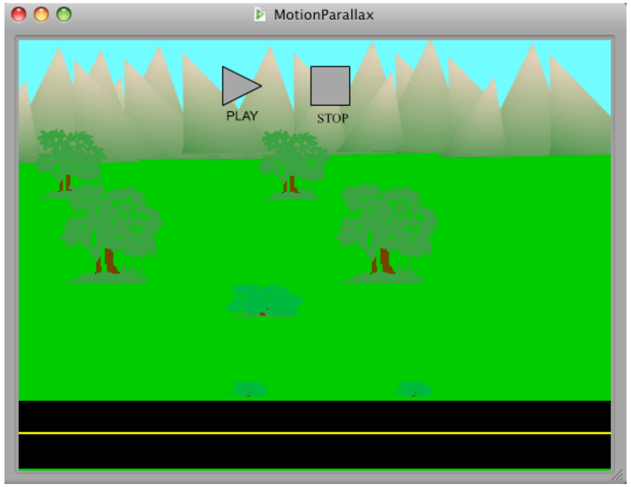
Berechnet den Reflection Vector in Eye-Koord.

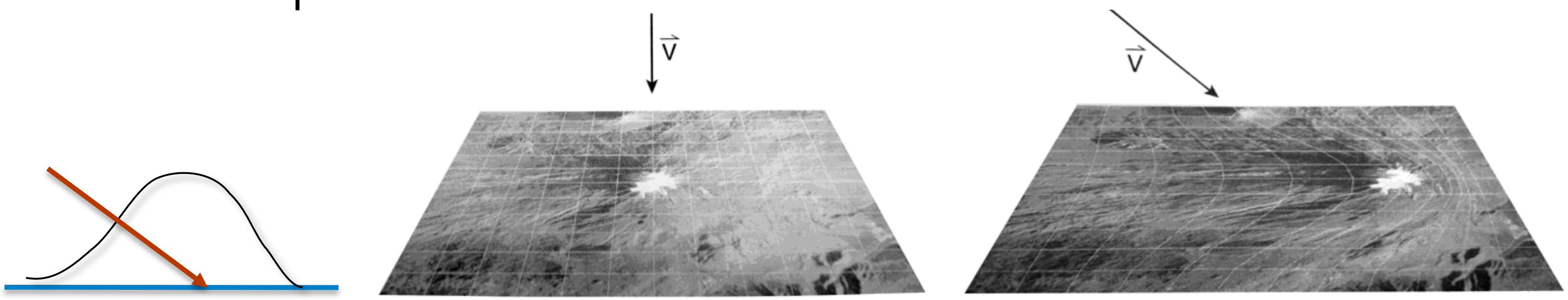
For Further Reading

- On the class's homepage:
  - "OpenGL Cube Map Texturing" (Nvidia, 1999)
    - With example code
    - Here several details are explained (e.g. the orientation)
  - "Lighting and Shading Techniques for Interactive Applications" (Tom McReynolds & David Blythe, Siggraph 1999);
  - SIGGRAPH '99 Course: "Advanced Graphics Programming Techniques Using OpenGL" (ist Teil des o.g. Dokumentes)

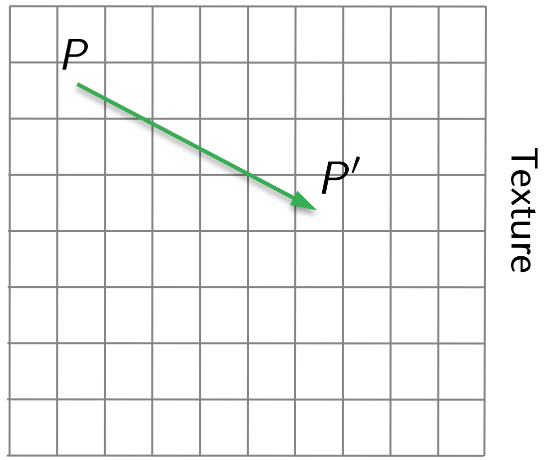
Parallax Mapping

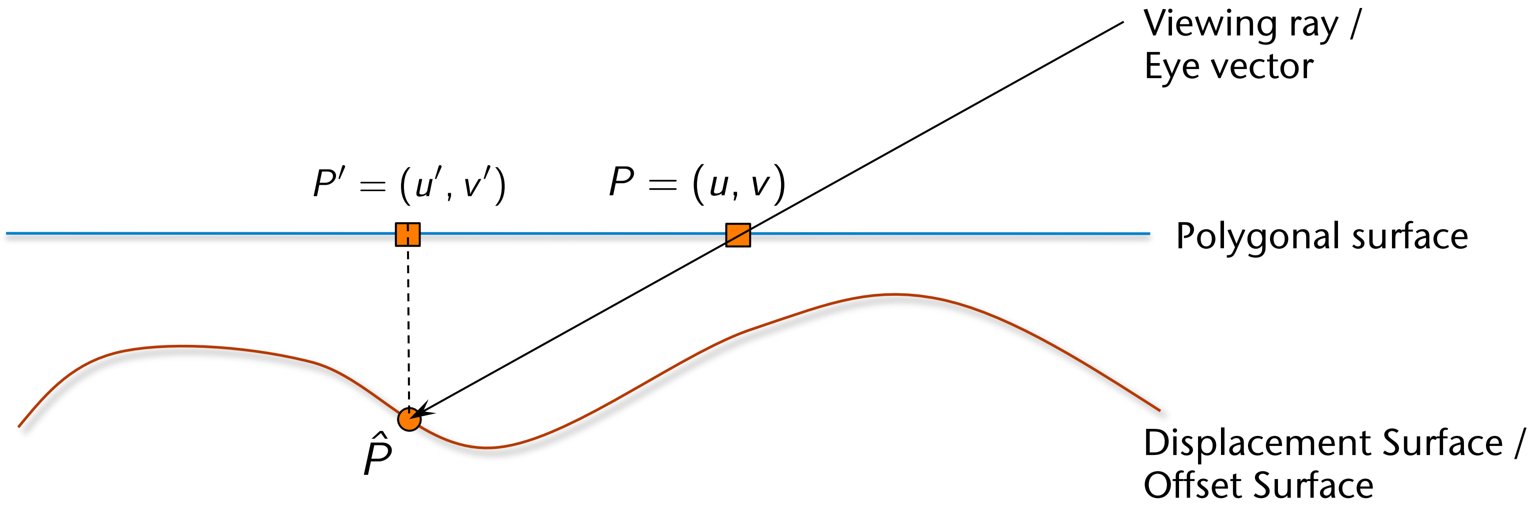
- Problem with bump- / normal mapping:
  - Only the lighting is affected – the image of the texture remains unchanged, regardless of the direction from which one looks
  - Motion parallax: near / distant objects shift very differently relative to one another (or even in a different direction! depending on the point of focus)
  - Extreme example:



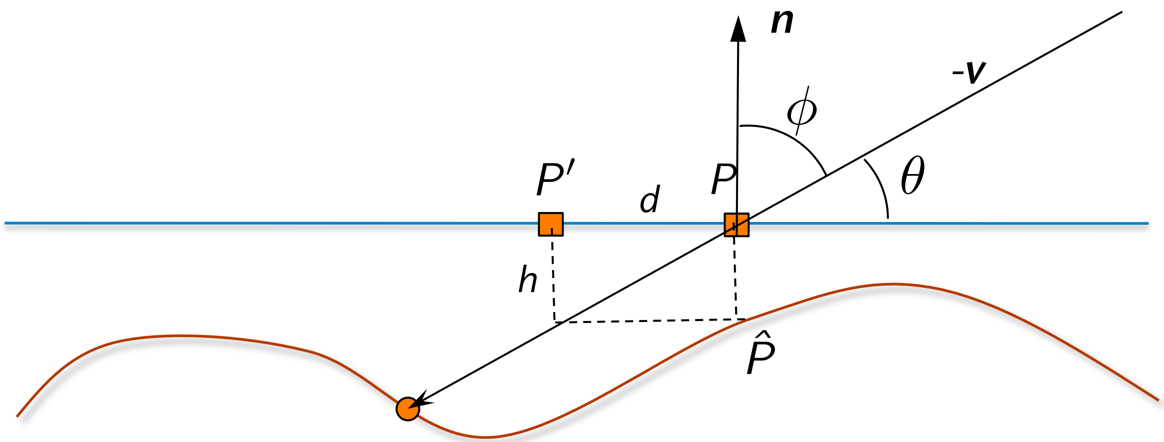


- The basic task in parallax mapping:
  - Assume, scan line conversion is at pixel P
  - Determine point $\hat{P}$, that *would* be seen
  - Project $\hat{P}$ onto P'
  - Write the corresponding texel as a color
- Problem: how does one find P' ?



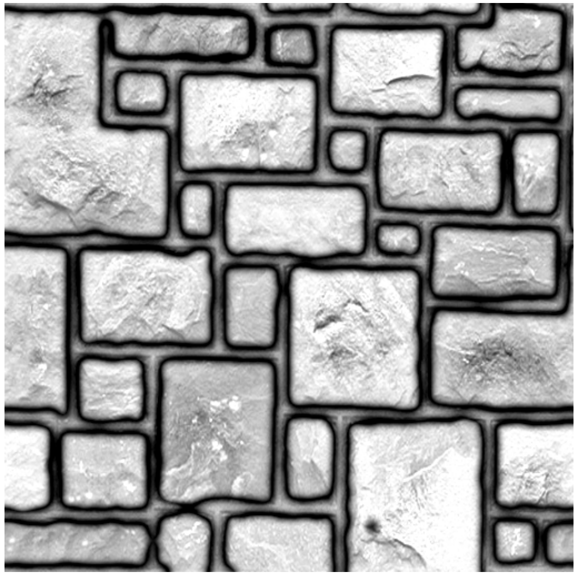


- Simplest idea:
  - We know the height $h = D(u,v)$ at point $P = P(u,v)$
  - Use this as an approximation of $D(u',v')$ in point $P' = P'(u,v)$
  - $\frac{h}{d} = \tan\theta = \frac{\sin\theta}{\cos\theta} = \frac{\cos\phi}{\sin\phi} = \frac{|\mathbf{nv}|}{|\mathbf{n}\times\mathbf{v}|}$



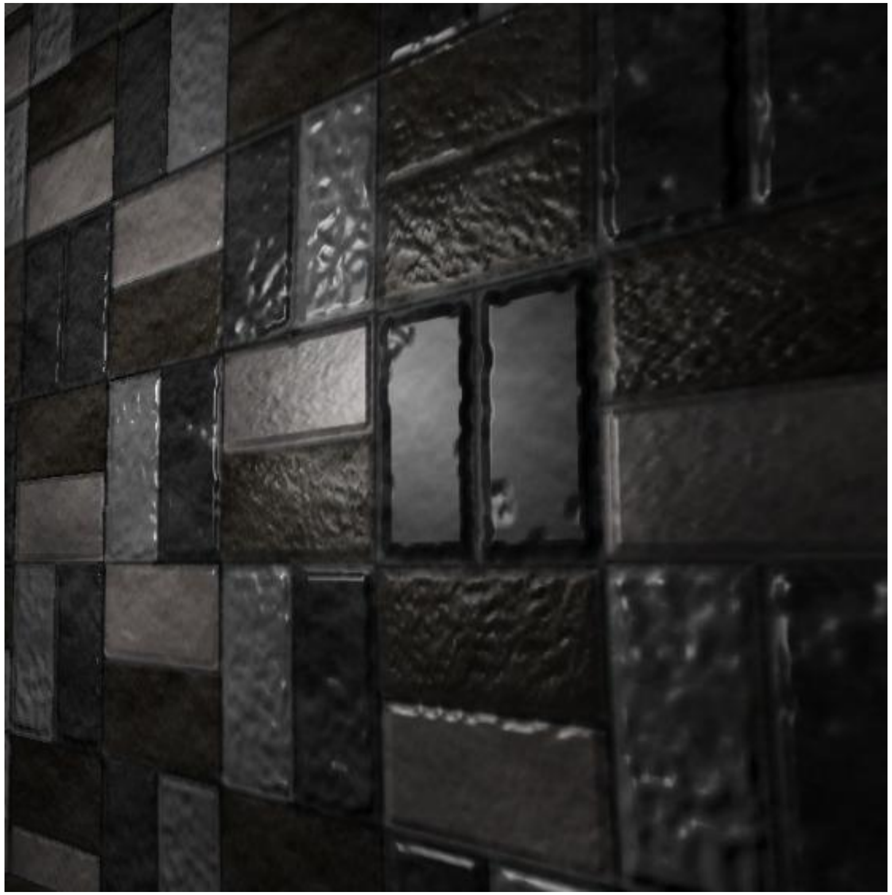
- Storage:
  - The image in the RGB channels of the texture
  - The heightmap in the alpha channel
- Process:
  - Compute P' (see previous slide)
  - Calculate (u',v') of P' ⟶ lookup texel
  - Perturb normal by bump mapping (see CG1)
    - Note: today one can calculate directional derivatives for D_U and D_V "on the fly" (needed in bump mapping algo)
  - Evaluate Phong model with texel color

RGB

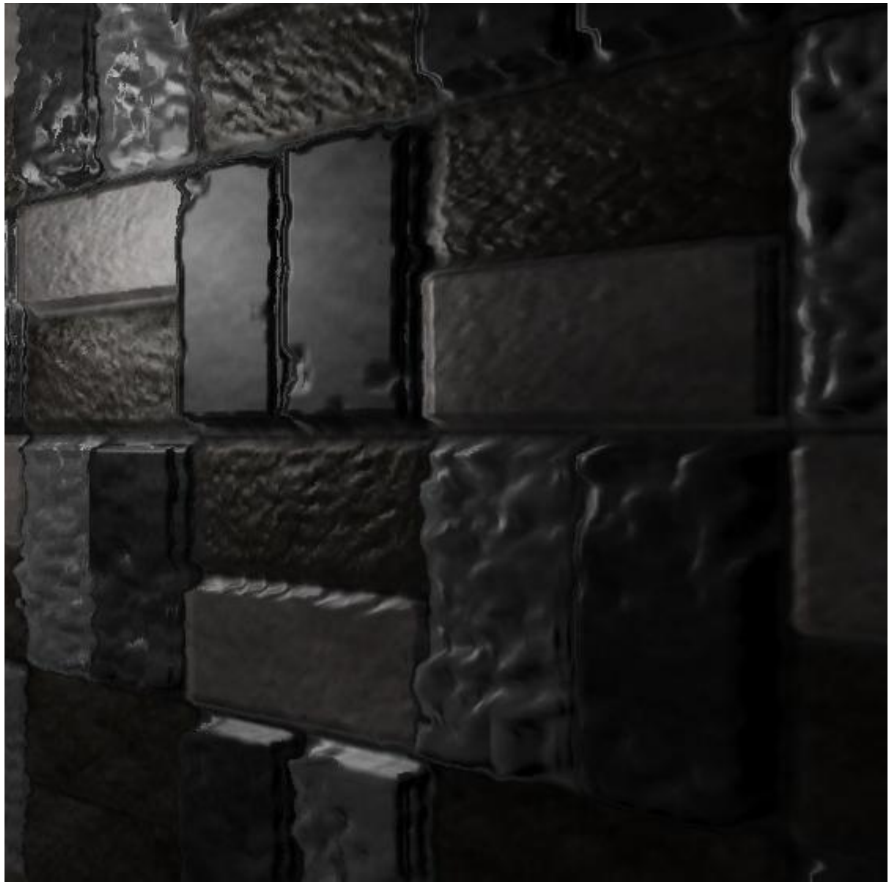


Alpha

Example



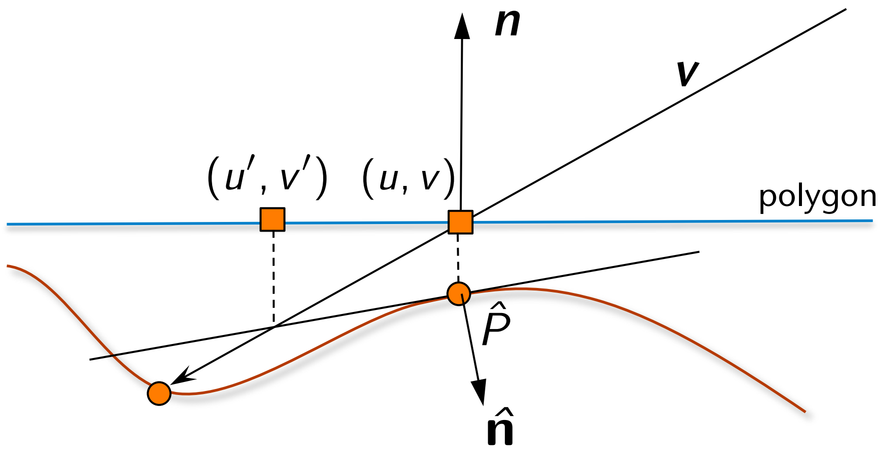
Normal Bump Mapping



Parallax Mapping
(For demonstration purposes,
parallax is strongly exaggerated here)

- Improvement: [Premecz, 2006]
  - Let $\hat{P} = (u, v, h)$ with $h = D(u, v)$
  - Approximate the heightmap in $\hat{P}$ through a plane (similar to bump mapping)
  - Calculate the point of intersection between that plane and the view vector

$$\hat{\mathbf{n}} \left(\begin{pmatrix} u \\ v \\ 0 \end{pmatrix} + t\mathbf{v} - \begin{pmatrix} u \\ v \\ h \end{pmatrix} \right) = 0$$



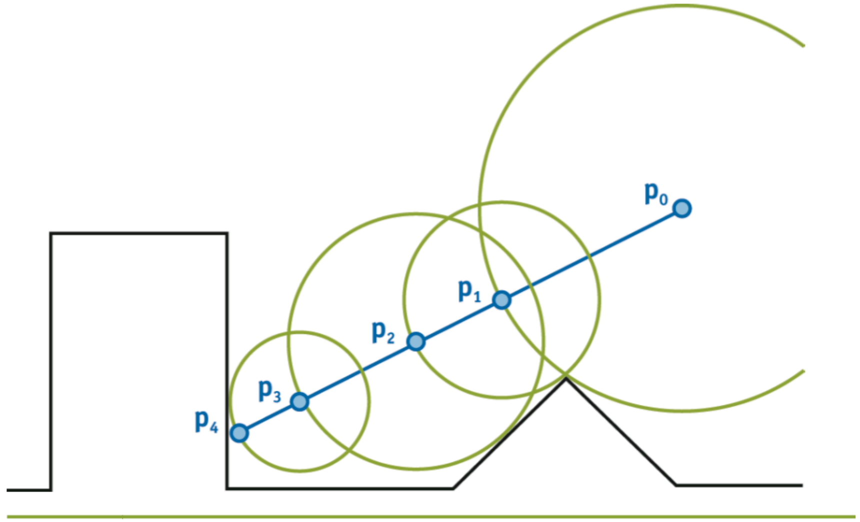
  - Solve 3rd line for t
  - $\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} + t\mathbf{v}'$, with $\mathbf{v}'$ = ($\mathbf{v}$ projected into polygon's plane)
- Additional (closely related) ideas: iteration, higher approximation of the heightmap

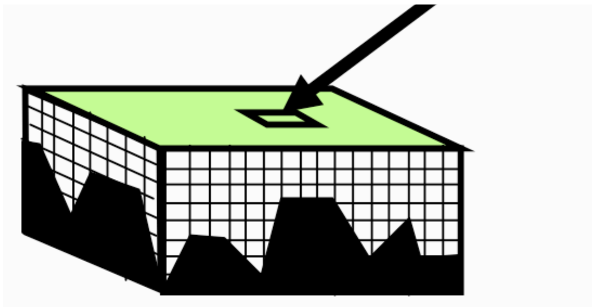
Thesis ...

Alternatives

[Donnelly, 2005]

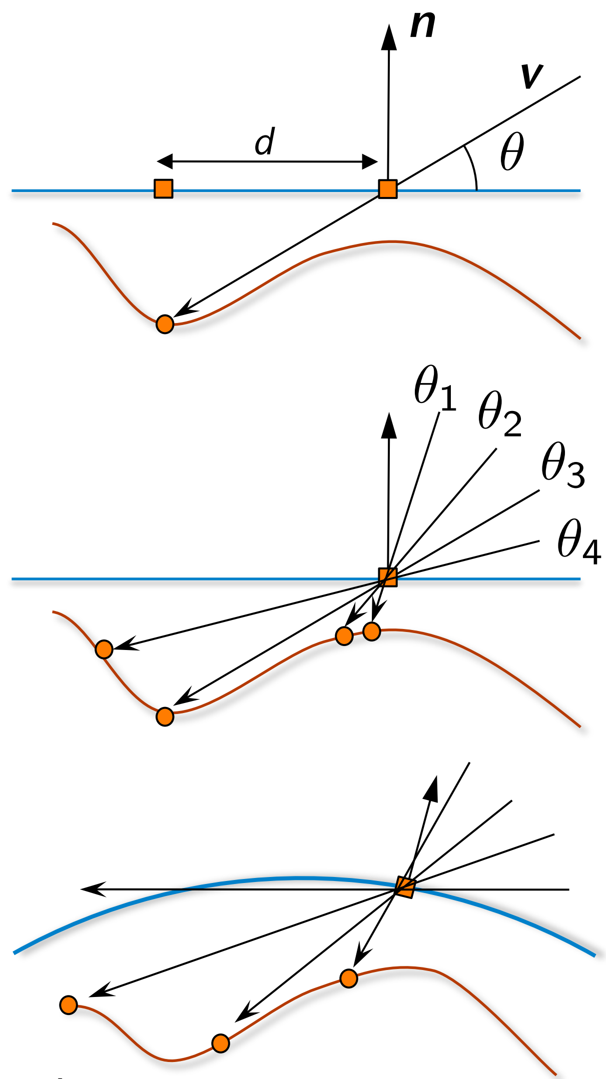
- Do sphere tracing along the view vectors, until you hit the offset surface
  - If the heightmap contains heights that are not too large, it is sufficient to begin relatively close underneath/ above the plane of reference
  - If the angle of the view vector is not too acute, then a few steps are sufficient
- For a layer underneath the plane of reference, save the smallest distance to the offset surface for every cell





View-Dependent Displacement Mapping (VDM) [2003]

- Idea: precompute all possible texture coordinate displacements for all possible situations
- In practice:
  - Parameterize the viewing vector by (θ, ϕ) in the local coordinate system of the polygon
  - Precompute the texture displacement for all (*u*,*v*) and a specific (θ, ϕ)
    - Ray casting of an explicit, temporarily generated mesh
  - Carry this out for all possible (θ, ϕ)
  - Carry out the whole for a set of *possible* curvatures *c* of the base surface
- Results in a 5-dim. "Texture" (LUT): $d(u, v, \theta, \phi, c)$



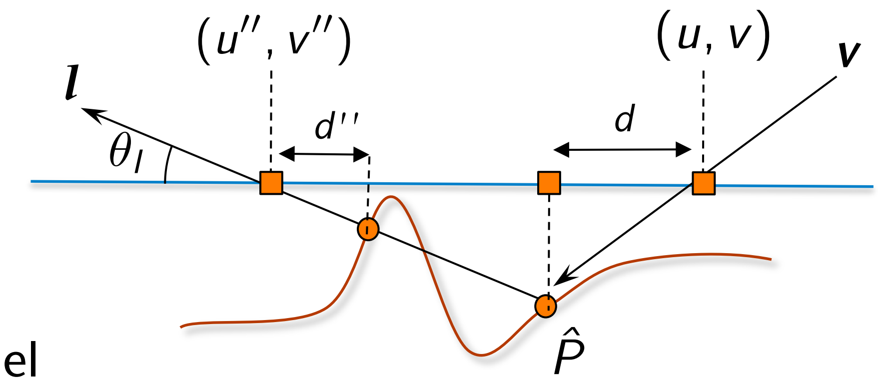
- Pro: results in a correct silhouette
  - Reason: $d(u, v, \theta, \phi, c) = -1$ for many parameters near the silhouette
  - These are the pixels that lie outside of the silhouette!
- Further enhancement: self shadowing
  - Idea like that in ray tracing: use "shadow rays"

1. Determine $\hat{P}$ from d and θ, ϕ (just like before)
2. Determine vektor $\boldsymbol{l}$ from $\hat{P}$ to the light source; and calc θ_l, ϕ_l from that
3. Determine $P'' = (u'', v'')$ from $\hat{P}$ and θ_l and ϕ_l
4. Make lookup in our "texture" d
5. Test:

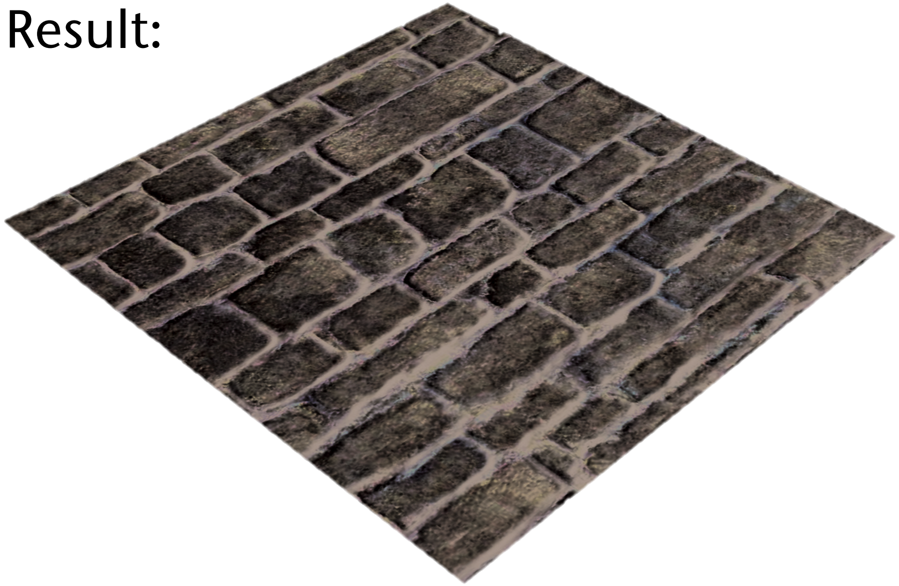
$$d(u'', v'', \theta_l, \phi_l, c) < d(u, v, \theta, \phi, c)$$

→ pixel (u,v) is in shadow

→ don't add light source $\boldsymbol{l}$ in Phong model



- Result:

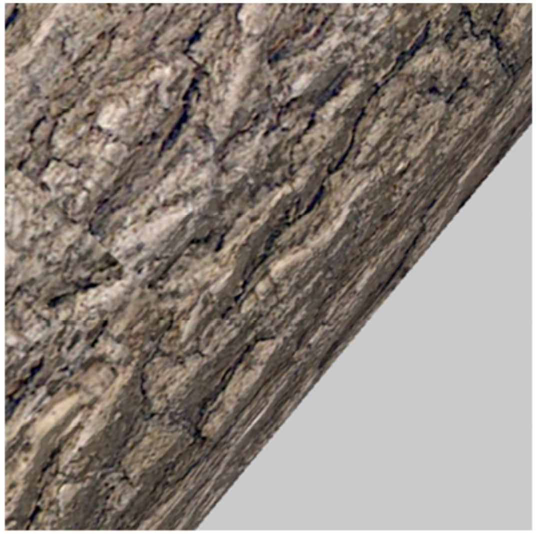


Bump Mapping

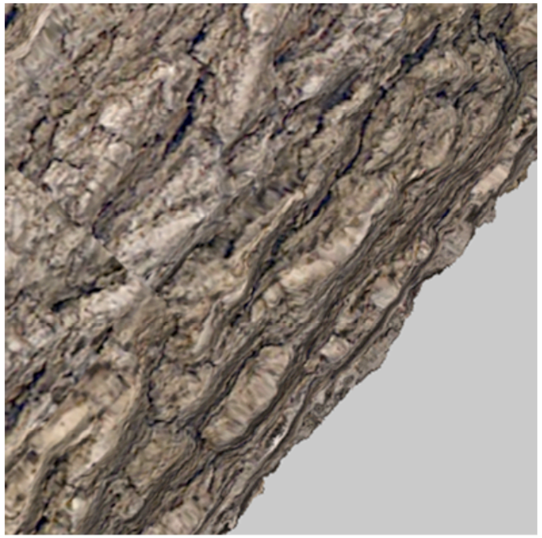
Displacement Mapping

- Names:
  - Steep parallax mapping, parallax occlusion mapping, horizon mapping, view-dependent displacement mapping, ...
  - There are still many other variants ...
  - "Name ist Schall und Rauch!" ("A name is but noise and smoke!")

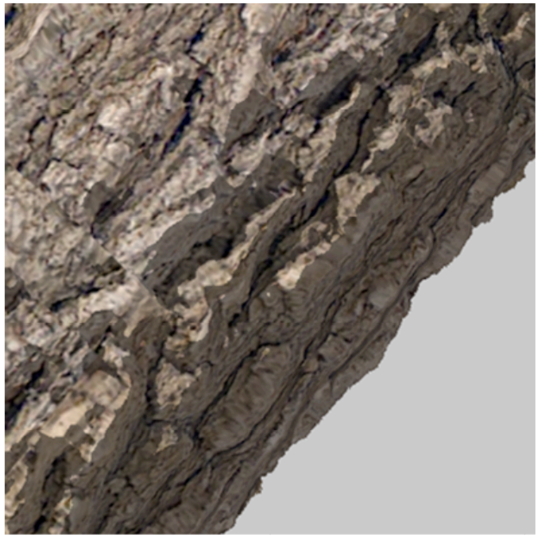
More Results



Bump mapping



Simple Displacement Mapping
(not covered here)



View-dependent displacement mapping with self-shadowing

All Examples Were Rendered with VDM