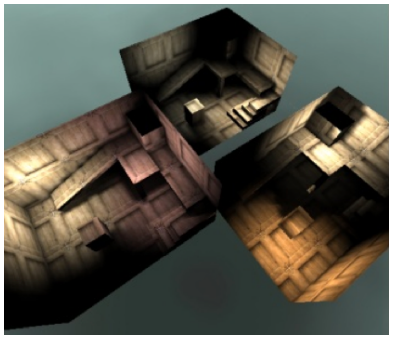
Applications of Texture Mapping

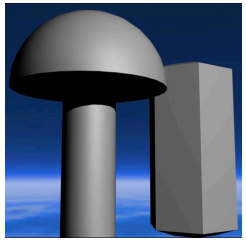
Prof. Vladlen Koltun
Computer Science Department
Stanford University

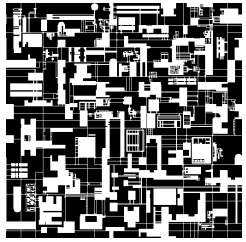
Light maps

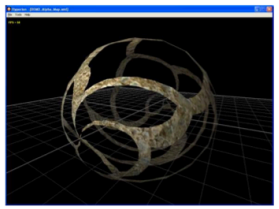


http://www.irrlicht3d.org

Opacity mapping

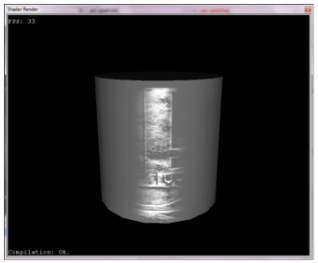




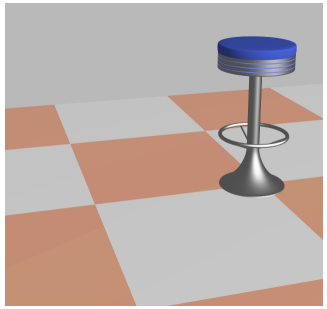


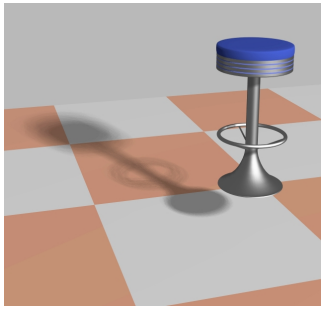
http://lectrablog.lectralizard.com/page/2

Specular mapping



Shadows

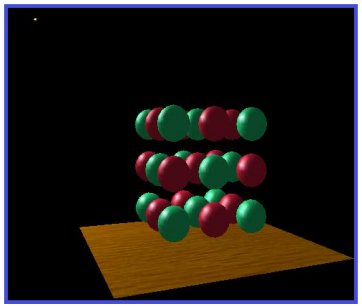


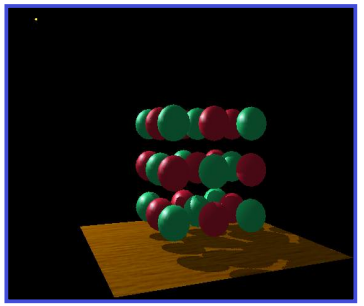


- Valuable cue of spatial relationships
- Increases realism

Akenine-Moeller and Haines

Shadows

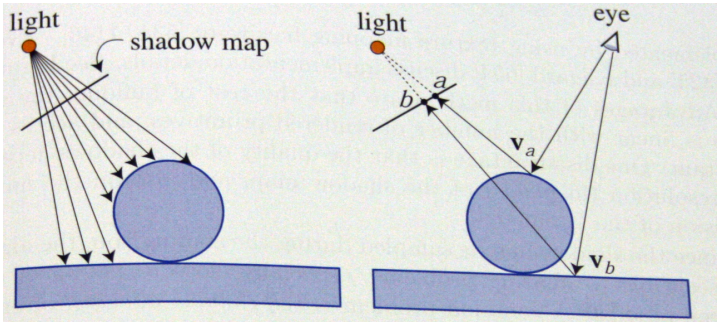




- Valuable cue of spatial relationships
- Increases realism

Mark J. Kilgard

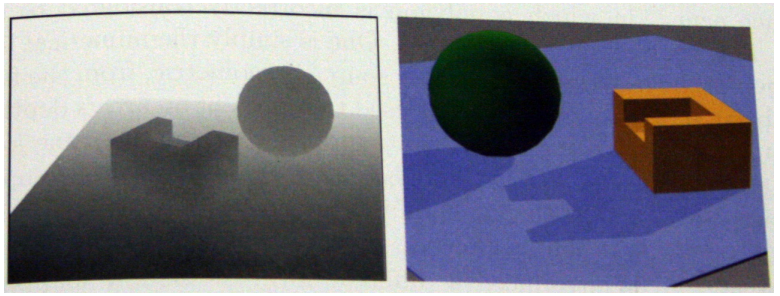
Shadow mapping



- First pass: render the scene from the viewpoint of the light, store depth buffer as texture (shadow map)
- Second pass: project vertices into shadow map and compare depth values

Akenine-Moeller et al., Real-Time Rendering

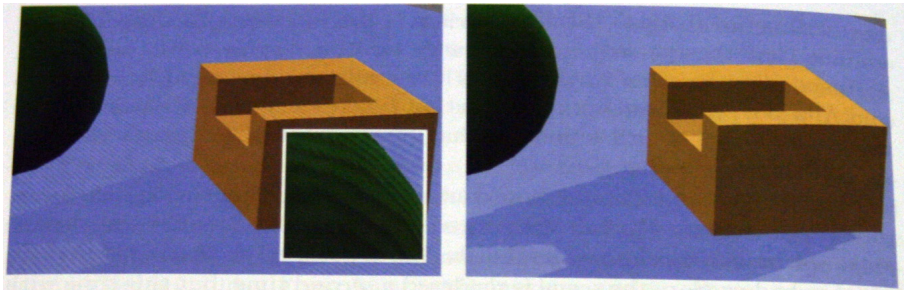
Shadow mapping



- First pass details: can disable all rendering features that do not affect depth map.
- Second pass details: For each fragment, use the light's modelview and projection transforms to obtains (u,v) coordinates in the shadow map and the depth w of the vertex.
- Compare w with value w' stored in (u,v) in the shadow map. If $w \leq w'$, perform lighting calculations with this light. Otherwise, do not.

Akenine-Moeller et al., Real-Time Rendering

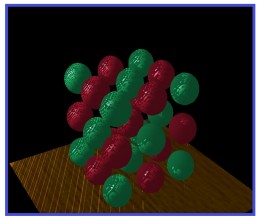
Bias



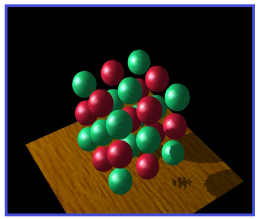
- Numerical imprecision leads to self-shadowing
- Solution: add a bias ε. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$
- Can use glPolygonOffset

Akenine-Moeller et al., Real-Time Rendering

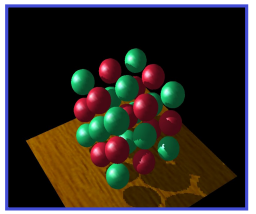
Setting the bias



Too little



Too much

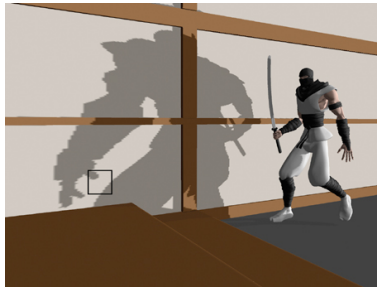


Just right

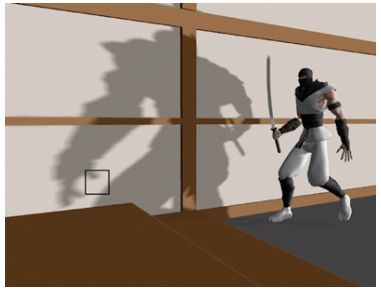
- Numerical imprecision leads to self-shadowing
- Solution: add a bias ε. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$
- Can use glPolygonOffset

Mark J. Kilgard

Shadow map aliasing



Unfiltered



Filtered

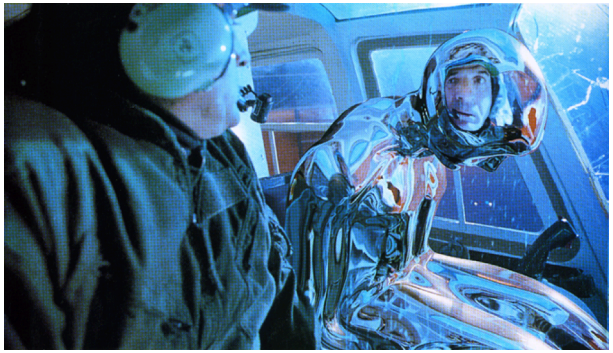
- Insufficient shadow map resolution leads to blocky shadows
- No easy solution. Should not filter depth values: leads to errors at object boundaries
- Percentage-closer filtering: filter comparison results

Bunnell and Pellacini

Other issues

- Additional rendering pass for each shadow-casting light
- Setting the "field of view" of the light. Can use spotlights, or a cube map (six shadow maps) for a point light.
- For directional lights, use orthographic projection

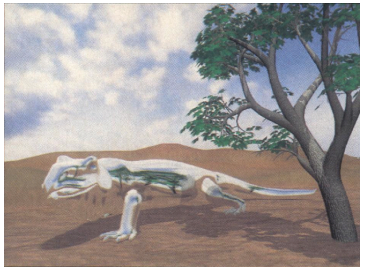
Reflection mapping



- Render the scene from a single point inside the reflective object. Store rendered images as textures.
- Map textures onto object. Determine texture coordinates by reflecting view ray about the normal.

Terminator 2

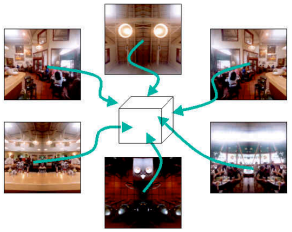
Cube mapping

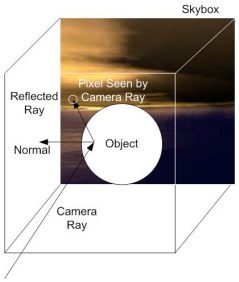


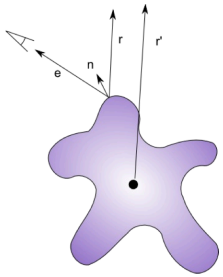
- Render the scene six times, through six faces of a cube, with 90-degree field-of-view for each image.
- Store images in six textures, which represent an omni-directional view of the environment

Greene, 1986

Cube mapping







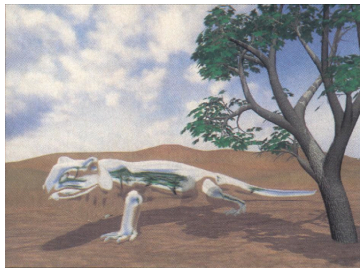
- To compute texture coordinates, reflect the view vector **v** about the normal **n**:

$$\mathbf{r} = 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} - \mathbf{v}$$

- The highest (in absolute value) coordinate of r identifies which of the six maps we need. The texture coordinates in this map are obtained by normalizing the other two coordinates of r.

http://developer.nvidia.com/object/cube_map_ogl_tutorial.html; TopherTG (Wikipedia)

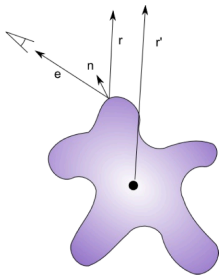
Sphere mapping



- Cube maps require maintaining six texture in memory
- Sphere mapping uses a single viewpoint-specific environment map, updated every frame
- Map depicts a perfectly reflective sphere viewed orthographically

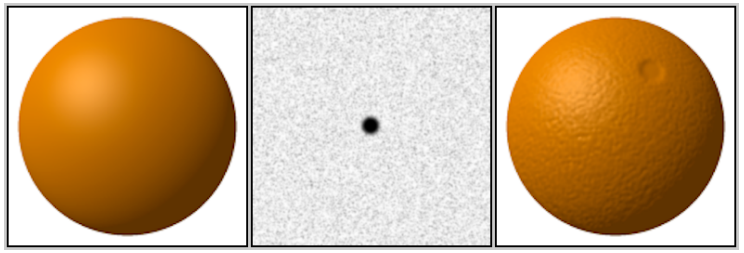
Greene, 1986

Reflection mapping limitations



- Self-reflections not supported. A concave object will not reflect parts of itself.
- Environment map only correct for the point from which it was rendered. Good approximation for distant reflected objects, but can lead to substantial artifacts in general.

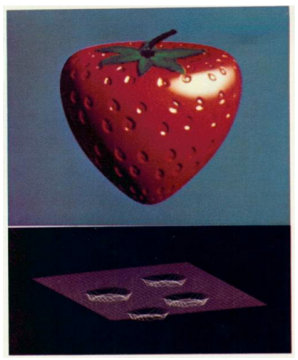
Bump mapping

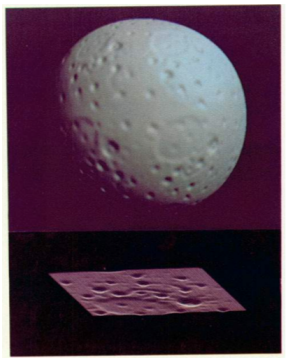


- Simulates roughness ("bumpiness") of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitation: silhouette is unaffected

GDallimore (Wikimedia Commons)

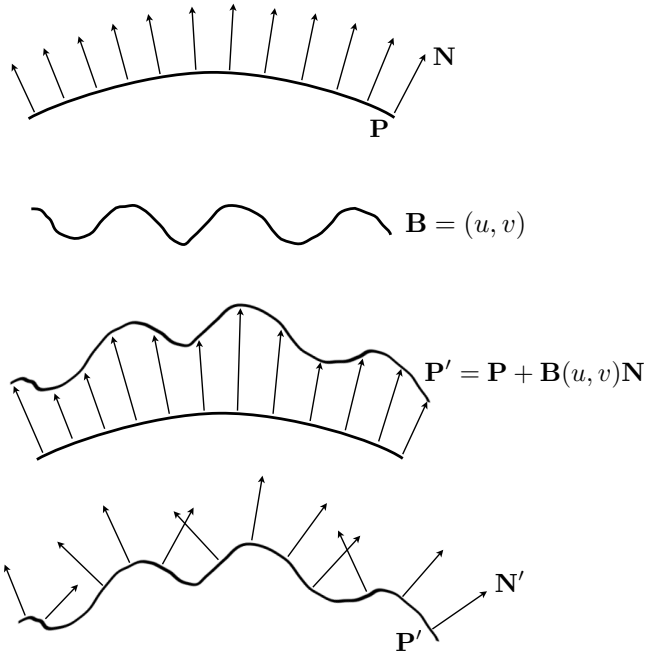
Bump mapping



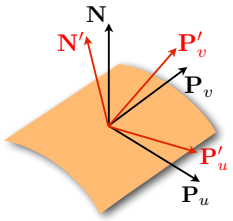


- Simulates roughness ("bumpiness") of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitation: silhouette is unaffected

Blinn, SIGGRAPH 1978



Bump mapping derivation



$$\mathbf{N} = \mathbf{P}_u \times \mathbf{P}_v$$
$$\mathbf{N}' = \mathbf{P}'_u \times \mathbf{P}'_v$$

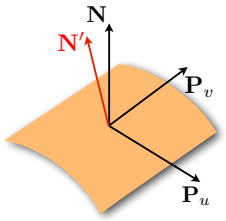
The normals are always appropriately normalized, but we omit that for simplicity

$$\begin{aligned}\mathbf{P}'_u &= \frac{\partial \mathbf{P}'}{\partial u} \\ &= \frac{\partial}{\partial u}\left(\mathbf{P} + \mathbf{B}(u,v)\mathbf{N}\right) \\ &= \mathbf{P}_u + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N} + \mathbf{B}(u,v)\frac{\partial \mathbf{N}}{\partial u} \\ &\approx \mathbf{P}_u + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N}\end{aligned}$$

$$\begin{aligned}\mathbf{P}'_v &= \frac{\partial \mathbf{P}'}{\partial v} \\ &= \frac{\partial}{\partial v}\left(\mathbf{P} + \mathbf{B}(u,v)\mathbf{N}\right) \\ &= \mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{N} + \mathbf{B}(u,v)\frac{\partial \mathbf{N}}{\partial v} \\ &\approx \mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{N}\end{aligned}$$

These are not really partial derivatives, but we follow Blinn's original notation

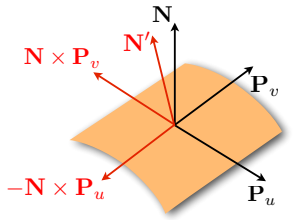
Bump mapping derivation



$$\begin{aligned}\mathbf{N}' &= \mathbf{P}'_u \times \mathbf{P}'_v \\ &= \left(\mathbf{P}_u + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N}\right) \times \left(\mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{N}\right) \\ &= \mathbf{N} + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N} \times \mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{P}_u \times \mathbf{N} \\ &= \mathbf{N} + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N} \times \mathbf{P}_v - \frac{\partial \mathbf{B}}{\partial v}\mathbf{N} \times \mathbf{P}_u\end{aligned}$$

The "partial derivatives" are just differences between adjacent pixel values in the bump map

Bump mapping derivation

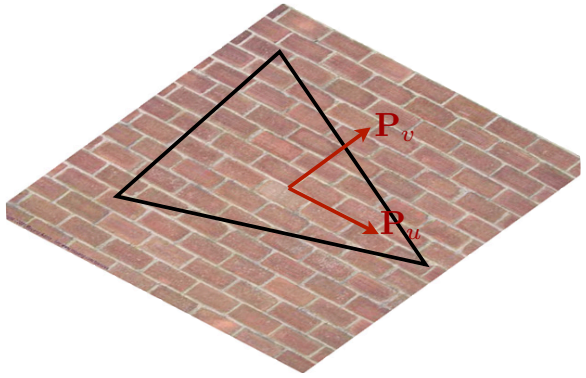


$$\begin{aligned}\mathbf{N}' &= \mathbf{P}'_u \times \mathbf{P}'_v \\ &= \left(\mathbf{P}_u + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N}\right) \times \left(\mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{N}\right) \\ &= \mathbf{N} + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N} \times \mathbf{P}_v + \frac{\partial \mathbf{B}}{\partial v}\mathbf{P}_u \times \mathbf{N} \\ &= \mathbf{N} + \frac{\partial \mathbf{B}}{\partial u}\mathbf{N} \times \mathbf{P}_v - \frac{\partial \mathbf{B}}{\partial v}\mathbf{N} \times \mathbf{P}_u\end{aligned}$$

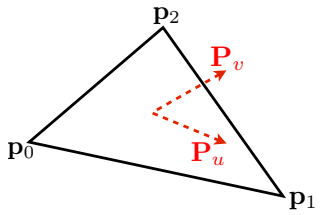
The "partial derivatives" are just differences between adjacent pixel values in the bump map

Computing tangent space basis vectors

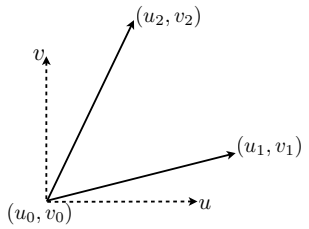
We now need to compute the vectors $\mathbf{P}_u$ and $\mathbf{P}_v$. These are the directions on the surface that correspond to zero change in the *v* parameter and the *u* parameter, respectively.



Computing tangent space basis vectors

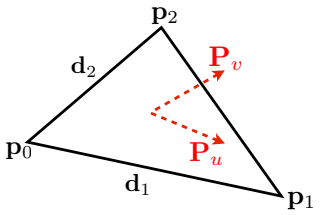


object space

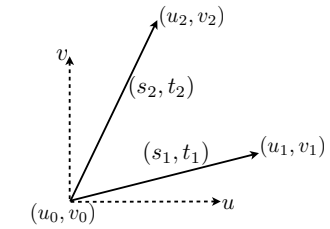


texture space

Computing tangent space basis vectors



object space

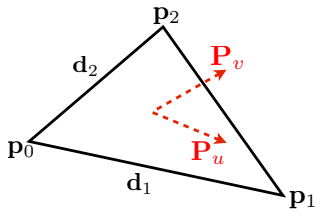


texture space

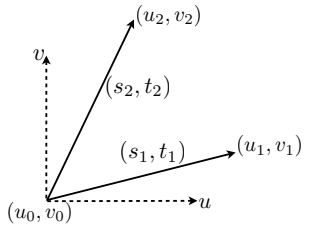
$$\mathbf{d}_1 = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{d}_2 = \mathbf{p}_2 - \mathbf{p}_0$$

$$(s_1, t_1) = (u_1 - u_0, v_1 - v_0)$$
$$(s_2, t_2) = (u_2 - u_0, v_2 - v_0)$$

Computing tangent space basis vectors



object space



texture space

$$\mathbf{d}_1 = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{d}_2 = \mathbf{p}_2 - \mathbf{p}_0$$

$$(s_1, t_1) = (u_1 - u_0, v_1 - v_0)$$
$$(s_2, t_2) = (u_2 - u_0, v_2 - v_0)$$

$$\mathbf{d}_1 = s_1\mathbf{P}_u + t_1\mathbf{P}_v$$
$$\mathbf{d}_2 = s_2\mathbf{P}_u + t_2\mathbf{P}_v$$

Computing tangent space basis vectors

$$\mathbf{d}_1 = s_1\mathbf{P}_u + t_1\mathbf{P}_v$$
$$\mathbf{d}_2 = s_2\mathbf{P}_u + t_2\mathbf{P}_v$$

Computing tangent space basis vectors

$$\mathbf{d}_1 = s_1\mathbf{P}_u + t_1\mathbf{P}_v$$
$$\mathbf{d}_2 = s_2\mathbf{P}_u + t_2\mathbf{P}_v$$

$$\begin{pmatrix} \mathbf{d}_1^\mathsf{T} \\ \mathbf{d}_2^\mathsf{T} \end{pmatrix} = \begin{pmatrix} s_1 & t_1 \\ s_2 & t_2 \end{pmatrix} \begin{pmatrix} \mathbf{P}_u^\mathsf{T} \\ \mathbf{P}_v^\mathsf{T} \end{pmatrix}$$

Computing tangent space basis vectors

$$\mathbf{d}_1 = s_1\mathbf{P}_u + t_1\mathbf{P}_v$$
$$\mathbf{d}_2 = s_2\mathbf{P}_u + t_2\mathbf{P}_v$$

$$\begin{pmatrix} \mathbf{d}_1^\mathsf{T} \\ \mathbf{d}_2^\mathsf{T} \end{pmatrix} = \begin{pmatrix} s_1 & t_1 \\ s_2 & t_2 \end{pmatrix} \begin{pmatrix} \mathbf{P}_u^\mathsf{T} \\ \mathbf{P}_v^\mathsf{T} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{P}_u^\mathsf{T} \\ \mathbf{P}_v^\mathsf{T} \end{pmatrix} = \frac{1}{s_1t_2 - s_2t_1} \begin{pmatrix} t_2 & -t_1 \\ s_2 & s_1 \end{pmatrix} \begin{pmatrix} \mathbf{d}_1^\mathsf{T} \\ \mathbf{d}_2^\mathsf{T} \end{pmatrix}$$

We can now normalize these vectors and use them for bump mapping. To compute tangent vectors at a vertex, we average the vectors from the adjacent faces, as we did with normals.

Normal mapping

original wall

normal map

shaded wall

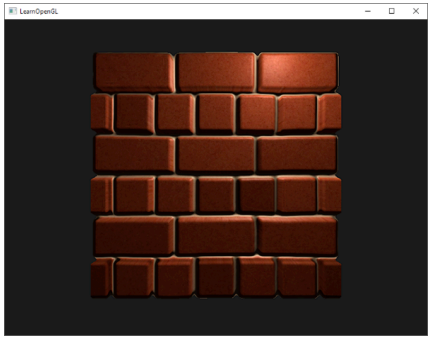
- Store the displaced normals directly. Reduces runtime overhead, at the expense of memory requirements
- (x,y,z) values in the tangent space are stored in the RGB channels. To compute the normal at a fragment, we simply multiply the (interpolated) tangent space basis by (x,y,z)

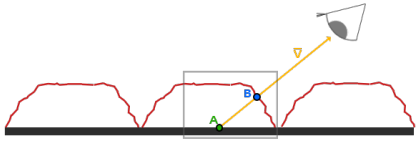
http://www.blacksmith-studios.dk/projects/downloads/bumpmapping_using_cg.php

Parallax mapping

Image from Terry Welsh

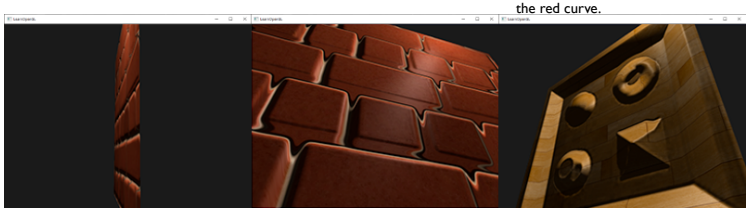
Parallax mapping





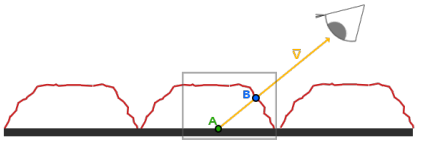
Credit learnopengl.con

The pixel (fragment) hit by ray $\vec{v}$ returns not its own color, but the color of the texture that would be returned had the surface followed the red curve.



Can be combined into

Parallax occlusion mapping



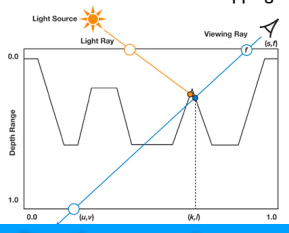
Relief mapping

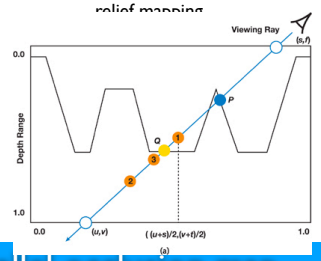
normal mapping relief mapping

Similar results, different math. Impacts both shading and outcome image.

Relief mapping

normal mapping





1. Similar results to parallel occlusion map.
2. Different math/algorithm
3. Impacts both shading and outcome image.
4. Credit https://developer.nvidia.com/gpugems

Relief mapping

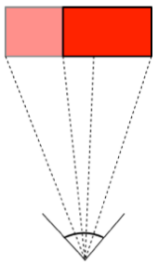
Image from Policarpo et al. (2005)

Other Buffers

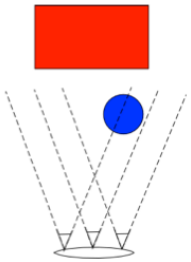
Accumulation buffer

- High-precision image buffer. Can integrate images that are rendered into the framebuffer. Supports anti-aliasing, motion blur, depth of field, soft shadows, etc.
- 16 bits for each red, green, blue, and alpha component: total of 64 bits per pixel.
- Supports the following operations:
  - Clear: set all values to zero.
  - Add with weight: Each pixel in the drawing buffer is added to the accumulation buffer after being multiplied by a floating-point weight that can be positive or negative.
  - Return with scale: The contents of the accumulation buffer are returned to the drawing buffer after being scaled by a positive floating-point constant
- Can integrate up to 256 images without loss of prevision, and even more using weight less than 1.0

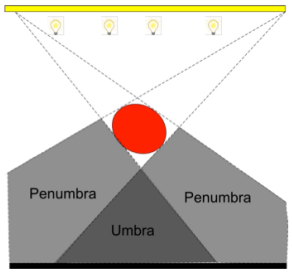
Accumulation buffer



motion blur

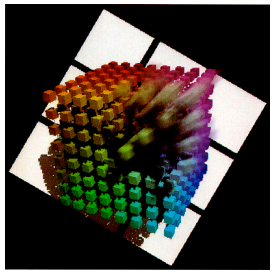


depth of field

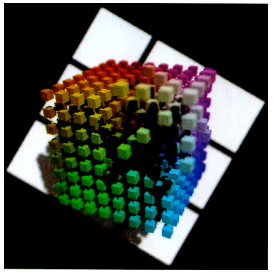


soft shadows

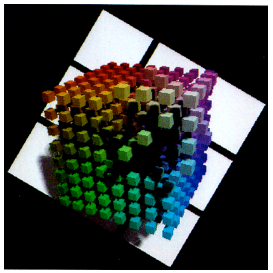
Accumulation buffer



motion blur



depth of field



soft shadows

Haeberli and Akeley, SIGGRAPH 1990