COSC 414/519I: Computer Graphics

2023W2

Shan Du

Texture Generation

- Use a photograph and paint it on an object by texture mapping is faster than generating the 2D- or 3D- texture and more realistic (e.g., grass).
- Using a 3D texture field $T(s, t, r)$ can make the texture mapping easier. The process is similar to sculpting the 3D object from a solid block whose volume is colored by the specified texture.

Environment Mapping

- The advantage of a raytracer is the ability to show objects reflecting one another.
- It is possible to create a relatively convincing, but somewhat fake, implementation of reflections in our rasterizer.

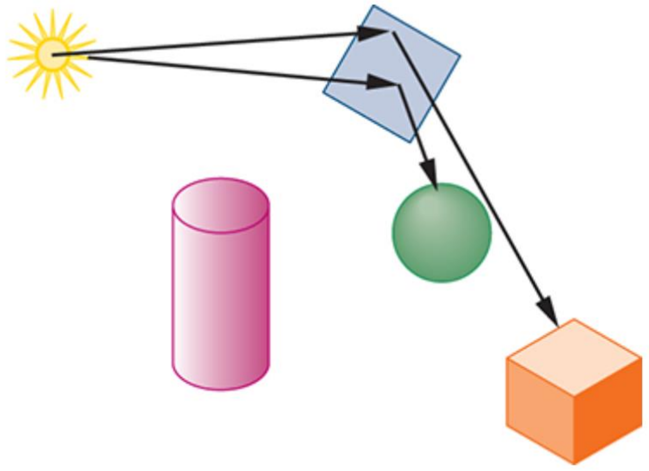
Environment Mapping

- Highly reflective surfaces are characterized by specular reflections that mirror the environment.
- We can use variants of texture mapping that can give approximate results that are visually acceptable through environment maps or reflection maps.

Environment Mapping

- The mirror is a polygonal object whose surface is a highly specular material.
- The position of the viewer and the normal to the mirror are known.
- The angle of reflection

can be determined.

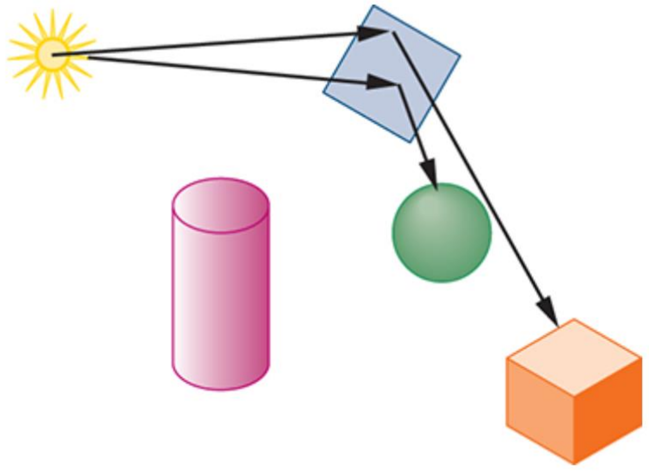


Environment Mapping

- If we follow along this angle until we intersect the environment, we obtain the shade that is reflected in the mirror.
- This shade is the result of a shading process that involves the light

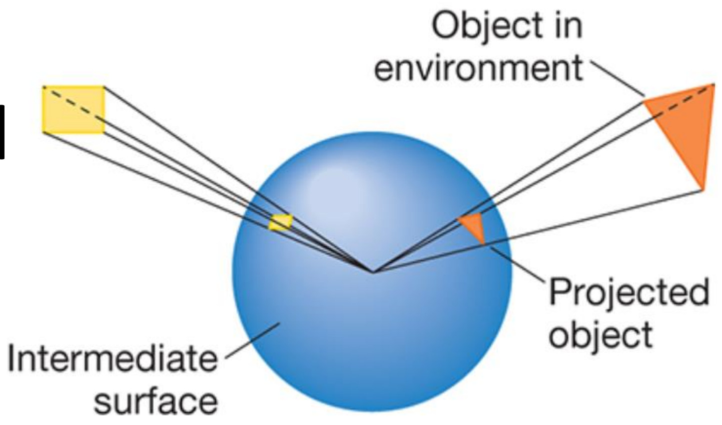
sources and materials in

the scene.



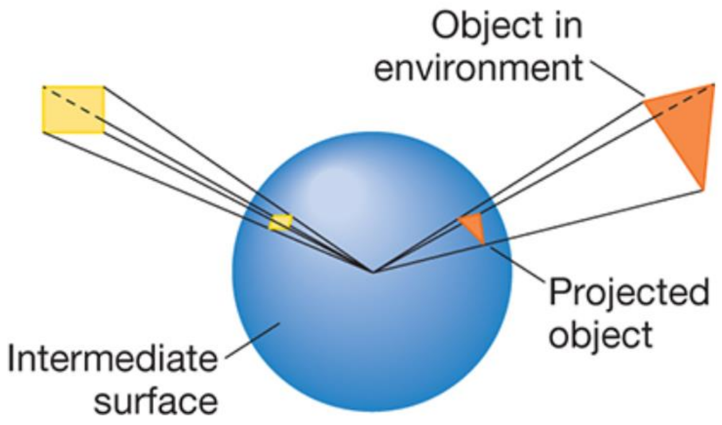
Environment Mapping

- We can obtain an approximately correct value of this shade as part of a two-step rendering pass, similar to the two-step texture-mapping process.
- In the first pass, we render the scene without the mirror polygon, with the camera placed at the center of the mirror pointed in the direction of the normal of the mirror.



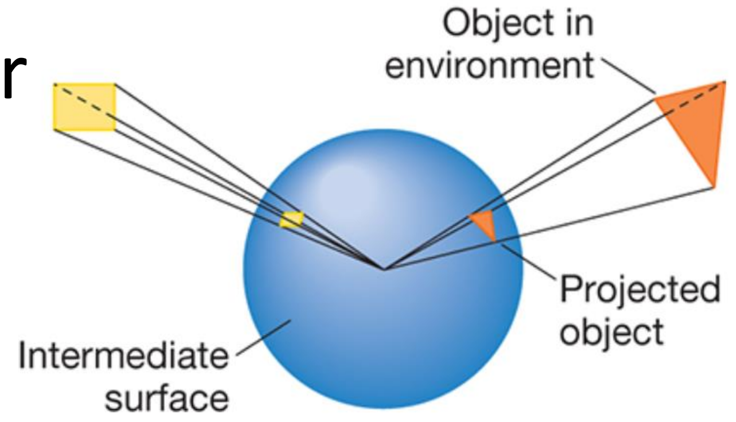
Environment Mapping

- Thus, we obtain an image of the objects in the environment as “seen” by the mirror.
- Then we can use this image to obtain the shades (texture values) on the mirror polygon.



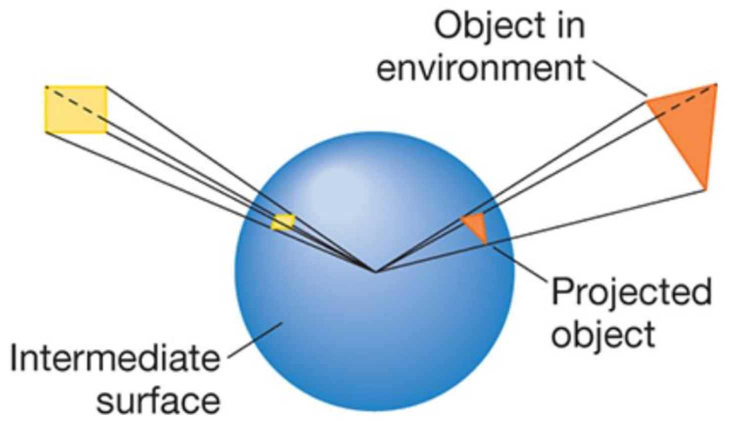
Environment Mapping

- The classic approach is to project the environment onto a sphere centered at the center of projection.
- A reviewer located at the center of the sphere cannot tell whether she is seeing the polygons in their original positions or their projections on the sphere.



Environment Mapping

- This illusion is similar to what we see in a planetarium. The “stars” that appear to be an infinite distance away are actually the projection of lights onto the hemisphere that encloses the audience.



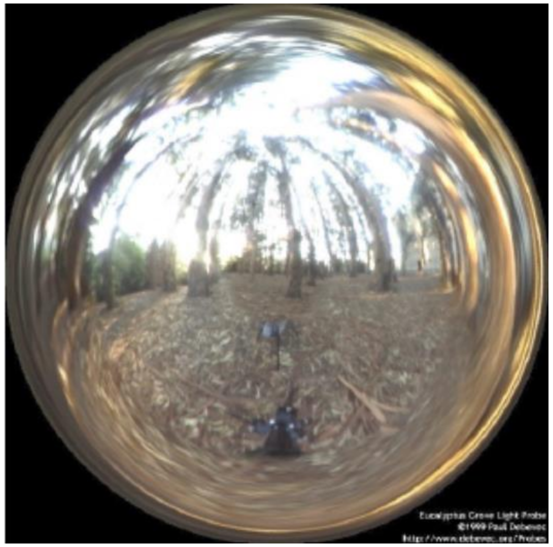
Environment Mapping

- In the original version of environment mapping, the surface of the sphere was then converted to a rectangle using lines of longitude and latitude for the mapping.
- Although conceptually simple, there are problems at the poles where the shape distortion becomes infinite.

Environment Mapping

- Sphere mapping:

A sphere map is a 2D representation of the full 360-degree view of the scene surrounding of an object, as if taken through a fish-eye lens.



Environment Mapping

- Cube mapping:

Cube mapping uses the six faces of a cube as the map shape. The environment is projected onto the sides of a cube and stored as six square textures, or unfolded into six regions of a single texture. The cube map is generated by rendering the scene six times from a viewpoint.

Environment Mapping

- Imagine we have a scene representing a room in a house, and we want to render a reflective object placed in the middle of the room.
- For each pixel representing the surface of that object, we know the 3D coordinates of the point it represents, the surface normal at that point, and, since we know the position of the camera, we can also compute the view vector to that point.
- We could reflect the view vector with respect to the surface normal to obtain a reflection vector.

Environment Mapping

- At this point, we want to know the color of the light coming from the direction of the reflection vector.
- Suppose that before rendering the objects inside the room, we place a camera in the middle of it and render the scene six times—once for each perpendicular direction (up, down, left, right, front, back).

Environment Mapping

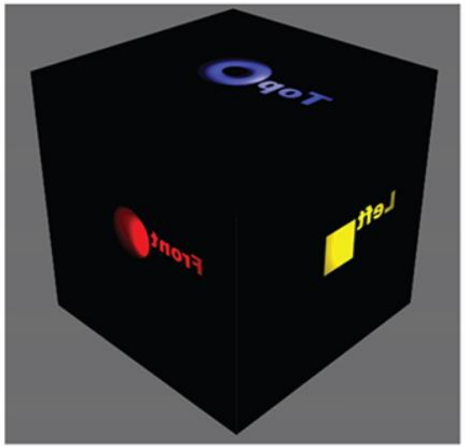
- You can imagine the camera is inside an imaginary cube, and each side of the cube is the viewport of one of these renders. We keep these six renders as textures. We call this set of six textures a cube map, which is why this technique is also called cube mapping.

Environment Mapping

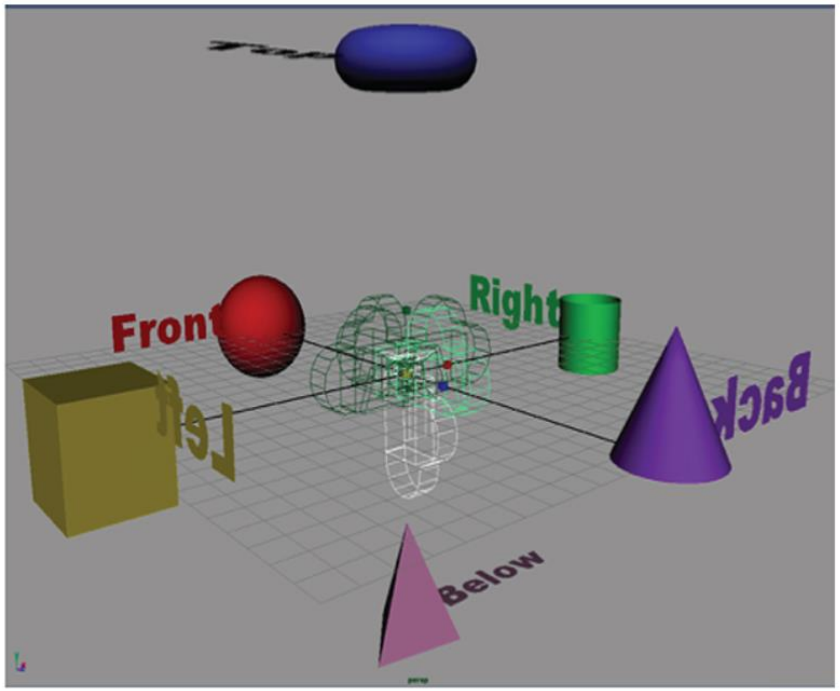
- Then we render the reflective object. When we get to the point of needing a reflected color, we can use the direction of the reflected vector to choose one of the textures of the cube map, and then a texel of that texture to get an approximation of the color seen in that direction.

Environment Mapping

- Cube mapping:



(a) Cube environment

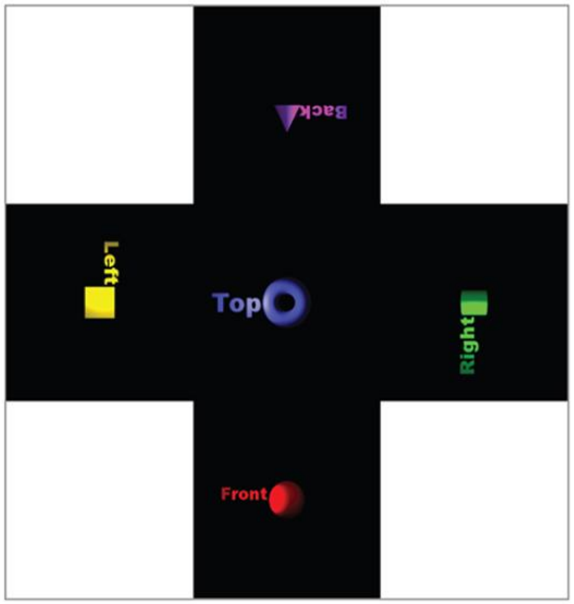


(b) Five virual cameras at center of cube

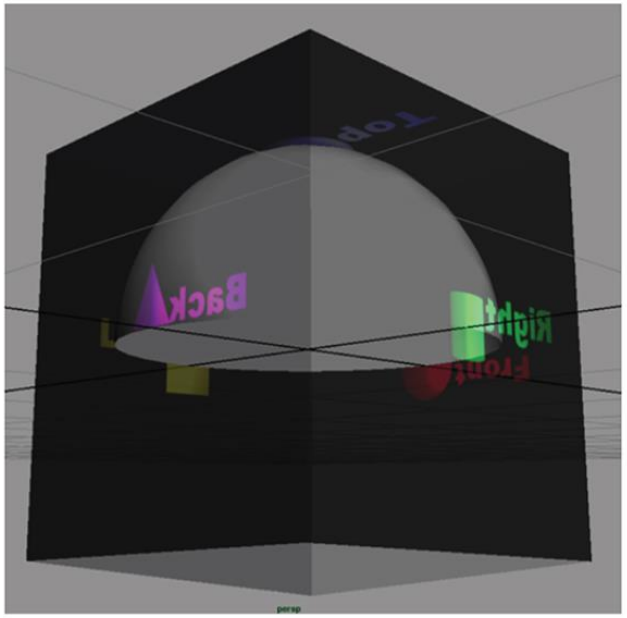
Environment Mapping

- Cube mapping:

Figure 7.33 **Mapping a cube map to a hemisphere.**



(a) Five texture maps displayed on an unfolded box



(c) Cube textures mapped to hemisphere

Environment Mapping

- This technique has some drawbacks. The cube map captures the appearance of a scene from a single point. If the reflective object we’re rendering isn’t located at that point, the position of the reflected objects won’t fully match what we would expect, so it will become clear that this is just an approximation.

Bump Mapping

- Bump mapping is a texture-mapping technique that can give the appearance of great complexity in an image without increasing the geometric complexity.
- Unlike simple texture mapping, bump mapping will show changes in shading as the light source or object moves, making the object appear to have variations in surface smoothness.

Bump Mapping

- A real, e.g., orange is characterized primarily by small variations in its surface rather than by variations in its color, and the former are not captured by texture mapping.
- The technique of bump mapping varies the apparent shape of the surface by perturbing the normal vectors as the surface is rendered.
- The colors that are generated by shading then show a variation in the surface properties.

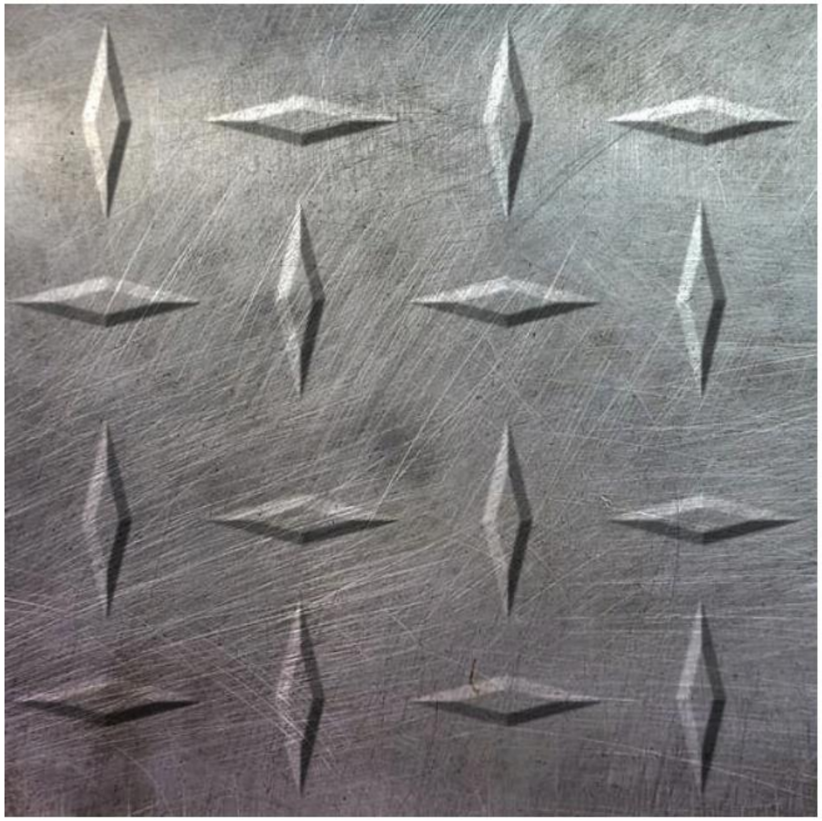
Bump Mapping

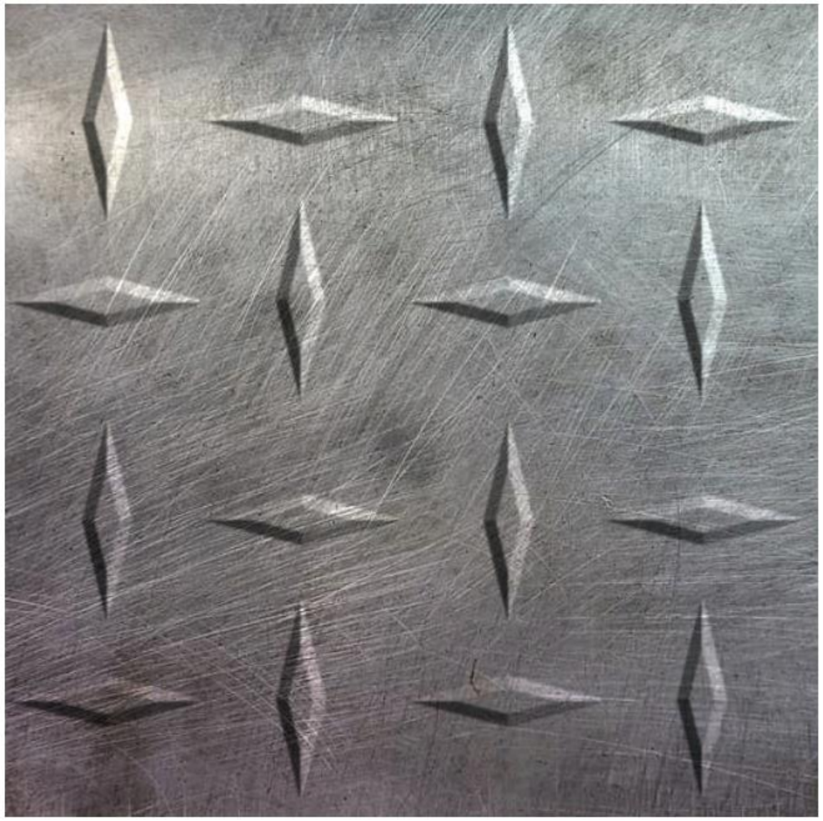
- We can use normals to change the way light interacts with a surface and thus change the apparent shape of the surface.
- To do this, we associate a normal map. A normal map is similar to a texture map, but its elements are normal vectors instead of colors.
- At rendering time, instead of computing an interpolated normal like Phong shading does, we use the normal map to get a normal vector for the specific pixel we're rendering, in the same way that texture mapping gets a color for that specific pixel. Then we use this vector to compute lighting at that pixel.

Bump Mapping

A flat surface with a texture map applied

Bump Mapping


(b) Normal map plus light from the left


(c) Normal map plus light from the right

Bump Mapping

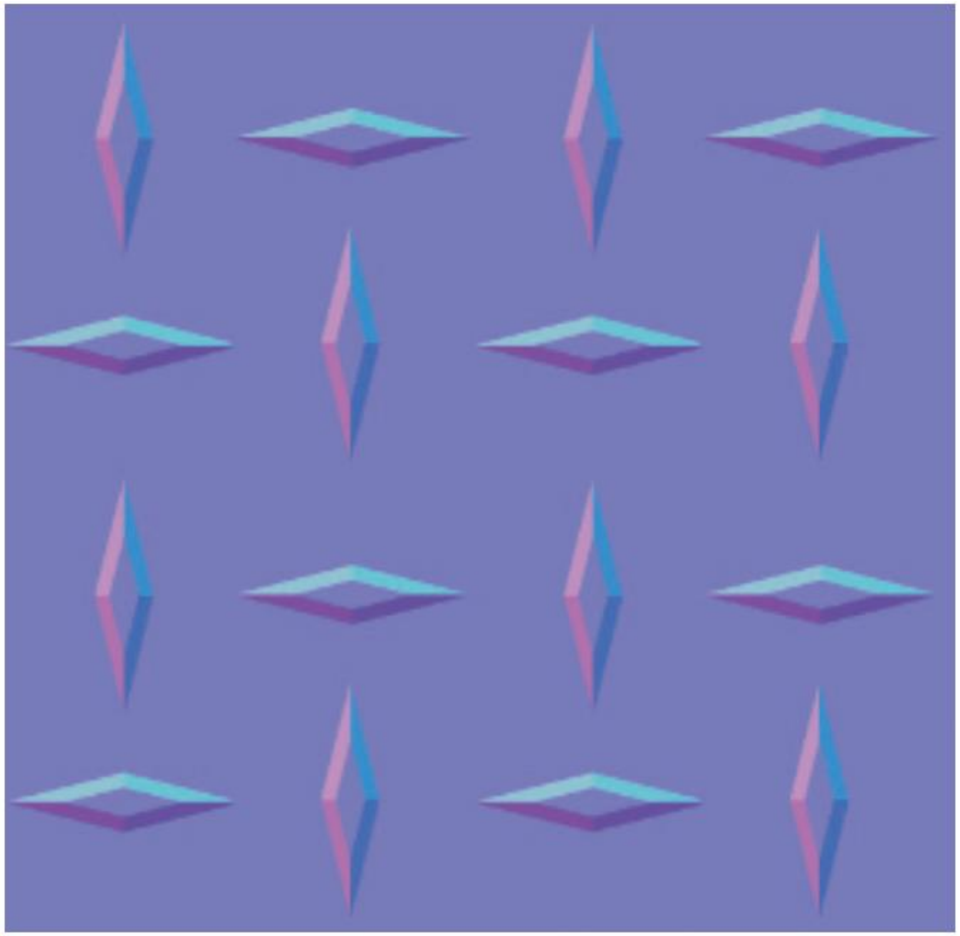


Figure 15-2: The normal map used for the examples

Bump Mapping

- A very popular way to encode normal maps is as textures, mapping the values of *X*, *Y*, and *Z* to *R*, *G*, and *B* values. This gives normal maps a very characteristic purple-ish appearance, because purple, a combination of red and blue but no green, encodes flat areas of the surface.
- This technique is better suited to subtle detail, such as pores on the skin, the pattern on a stucco wall, or the irregular appearance of an orange peel.

Bump Mapping

- Bump mapping is done on a fragment-by-fragment basis, we must be able to program the fragment shader.
- The normal at any point on a surface characterizes the orientation of the surface at that point.
- If we perturb the normal at each point on the surface by a small amount, then we create a surface with variations in its shape.

Bump Mapping

- We can perturb the normals in many ways.
- Let $p(u,v)$ be a point on a parametric surface. The partial derivatives at the point

$$P_u = \begin{bmatrix} \frac{\partial x}{\partial u} \\ \frac{\partial y}{\partial u} \\ \frac{\partial z}{\partial u} \end{bmatrix} \quad P_v = \begin{bmatrix} \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial v} \\ \frac{\partial z}{\partial v} \end{bmatrix}$$

lie in the plane tangent to the surface at the point.

Bump Mapping

- Their cross product can be normalized to give the unit normal at that point:
$$n = \frac{P_u \times P_v}{|P_u \times P_v|}$$
- Suppose that we displace the surface in the normal direction by a function called the bump, or displacement, function, $d(u, v)$, which we can assume is known and small ($|d(u, v)|$<<1). The displaced surface is given by

Bump Mapping

$$p' = p + d(u, v)n$$

- However, we do not want to create the displaced surface because it will have a higher geometric complexity and slow down the rendering process.
- We just want to make it look as though we have displaced the original surface.
- We can achieve the desired look by altering the normal n, instead of p, and using the perturbed normal in the shading calculation.

Bump Mapping

- The normal at the perturbed point p' is given by the cross product

$$n' = p'_u \times p'_v$$

- We can compute the two partial derivatives by differentiating the equation for p', obtaining

$$p'_u = p_u + \frac{\partial d}{\partial u} n + d(u, v) n_u$$

$$p'_v = p_v + \frac{\partial d}{\partial v} n + d(u, v)\, n_v$$

Bump Mapping

- If d is small, we can neglect the last term of these two equations and take their cross product, noting $n \times n = 0$, to obtain the approximate perturbed normal:

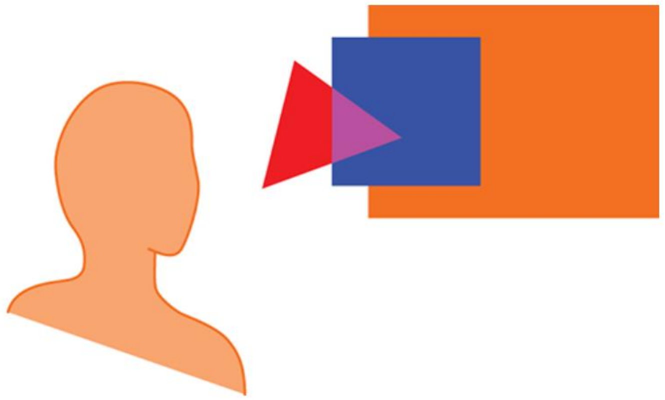
$$n' \approx n + \frac{\partial d}{\partial u} n \times p_v + \frac{\partial d}{\partial v} n \times p_u$$

- The last two terms are the displacement, the difference between the original and perturbed normal.

Blending

- WebGL provides a mechanism, through alpha (α) blending to create images with translucent objects.
- The alpha channel is the fourth color in RGBA (or RGBα) color mode.

Figure 8.1 **Translucent and opaque polygons.**



Blending

- If blending is enabled, the value of α controls how the RGB values are written into the framebuffer.
- Because fragments from multiple objects can contribute to the color of the same pixel, we say that these objects are blended or composited together.
- We can use a similar mechanism to blend together images.

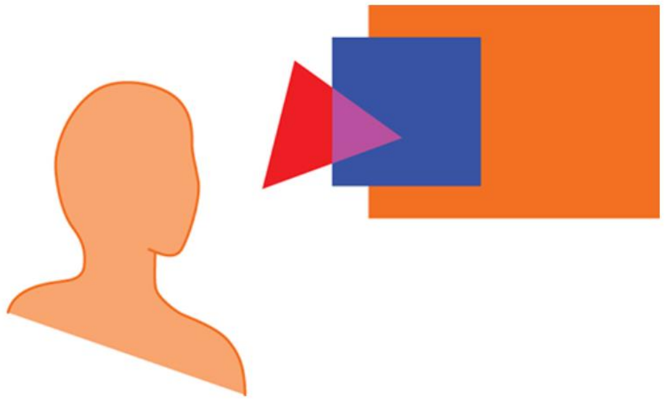
Opacity and Blending

- The opacity of a surface is a measure of how much light penetrates through that surface.
- An opacity of 1 (α = 1) corresponds to a completely opaque surface that blocks all light incident on it. A surface with an opacity of 0 is transparent; all light passes through it.
- The transparency or translucency of a surface with opacity α is given by $1-\alpha$.

Opacity and Blending

- In computer graphics, we usually render polygons one at a time into the framebuffer. Consequently, if we want to use blending, we need a way to apply opacity as part of fragment processing.
- We can use the notion of source and destination pixels.

Figure 8.1 **Translucent and opaque polygons.**



Opacity and Blending

- We regard the fragment being processed as the source pixel and the framebuffer pixel as the destination.
- If we represent the source and destination pixels with the four-element (RGBα) arrays

$$s = [s_r \quad s_g \quad s_b \quad s_a]$$
$$d = [d_r \quad d_g \quad d_b \quad d_a]$$

then a blending operation replaces d with

Opacity and Blending

$$d' = [b_r s_r + c_r d_r \quad b_g s_g + c_g d_g \quad b_b s_b + c_b d_b \quad b_a s_a + c_a d_a]$$

The constants
$b = [b_r \quad b_g \quad b_b \quad b_a]$ and $c = [c_r \quad c_g \quad c_b \quad c_a]$ are the source and destination blending factors.

Image Blending

- The most straightforward use of α blending is to combine and display several images that exist as pixel maps.
- We wish to keep our RGB colors between 0 and 1 in the final image. Hence, we can either scale the values of each image or use the source and destination blending factors.

Image Blending

- Suppose that we have n images that should contribute equally to the final display. At a given pixel, image i has components $C_i\alpha_i$. Here, we are using C_i to denote the color triplet (R_i, G_i, B_i). If we replace C_i by $\frac{1}{n}C_i$ and α_i by $\frac{1}{n}$, then we can simply add each image into the framebuffer (assuming the framebuffer is initialized to black with an $\alpha = 0$).

Image Blending

- Alternately, we can use a source blending factor of $\frac{1}{n}$ by setting the α value for each pixel in each image to $\frac{1}{n}$, and using 1 for the destination blending factor and α for the source blending factor.