SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

http://sglab.kaist.ac.kr/~sungeui/render

First printing, July 2018

9
Texture

Achieving higher realism has been one of main goals of computer graphics. For this goal, we have developed many modeling techniques by using more triangles, lights, and materials. Unfortunately, using additional resources (e.g., triangles and lights) come with sides effects such as additional running time and memory overheads.

Texture mapping adds additional details, without much overheads.

Since achieving the interactive performance has been another main goal of computer graphics, various approximation rather than directly relying upon additional geometry and lights has been studied. Among various techniques, texture mapping has been one of main approximation techniques (Fig. 9.1).

One thing that we need to understand is that while textures are originally designed for representing complex shapes of geometry, they can be utilized for various other purposes. At a high level, a texture is simply a 2D array, which is one of the most simplest data representations in computer architectures, and can be readily precomputed and used at runtime. Note that the Z-buffer used for visibility determination can be considered as a type of a texture. Thanks to these nice properties, textures have been widely used.

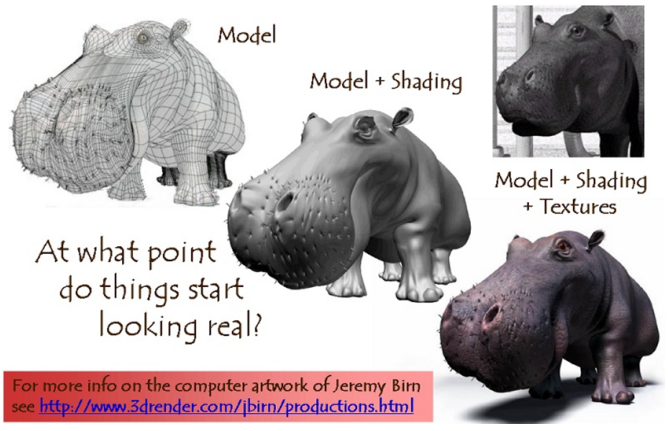

Figure 9.1: Texture mapping adds a lot of details to the geometry illuminated by lights, enabling higher realism without adding much overheads.

We first explain the main purpose of using texture mapping, followed by its various applications.

9.1 *Texture Mapping*

A texture is a 2 D (or 3 D and even a higher dimensional) buffer, whose each element represents a pixel color or some other values. Commonly a texture refers to a 2D image. Texture mapping indicates a mapping from a part of the texture to a part of a model. Fig. 9.3 shows an example of a 2D texture mapping.

We use 2 D texture coordinates, commonly (u, v), to locate a particular location of a 2D texture. We link the texture location to a particular location or a vertex of a mesh by using 2D or higher geometry coordinates (e.g., 3D coordinate (x, y, z) of a vertex). Fig. 9.2 shows that we map (u, v) coordinates of multiple texture locations to a 2D mesh, i.e., quad in the 2D space, represented by (x, y) coordinates. You may also recall that we use vt token to specify (u, v) texture coordinates to a vertex for an obj file format (Ch. 5.1).

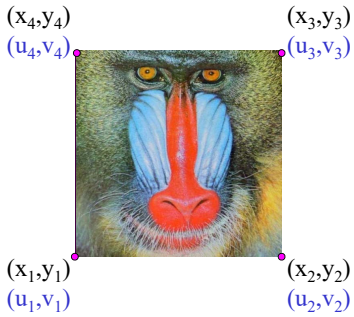

Figure 9.2: Texture mapping.

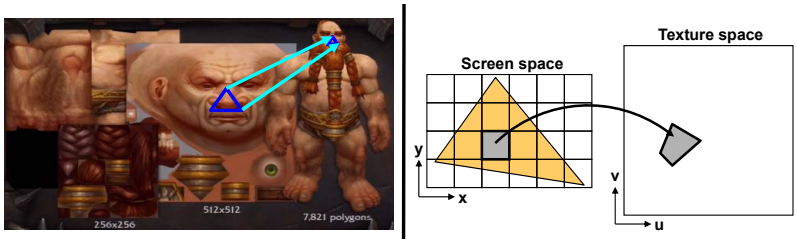

Figure 9.3: The left figure shows a mapping from a texture to a triangle representing a part of a character. We use a few texture maps with different resolutions. The image is excerpted from a MMO-champion site. To perform texture mapping, we compute a representative color of a pixel within the triangle from the texture space, as shown in the right figure. Note that a pixel of the triangle maps to an arbitrarily shaped quadrilateral in the texture space.

To apply the texture mapping, we compute a texture coordinate of a fragment of a triangle, while rasterizing the triangle. We compute the texture coordinate by interpolating texture coordinates associated with vertices of a triangle, as we did for other attributes (e.g., color) (Ch. 7.3).

Once we compute the texture coordinate, we compute the 2D indices of the corresponding texture pixel, known as texel, and use the color of the texel for illumination or other purposes.

Perspective-correct interpolation. Note that a naive interpolation of various vertex attributed in the image space does not provide the expected results that are supposed to be computed by the object-space interpolation. To achieve the correct result even in the image-space interpolation, the perspective-correct interpolation is developed.

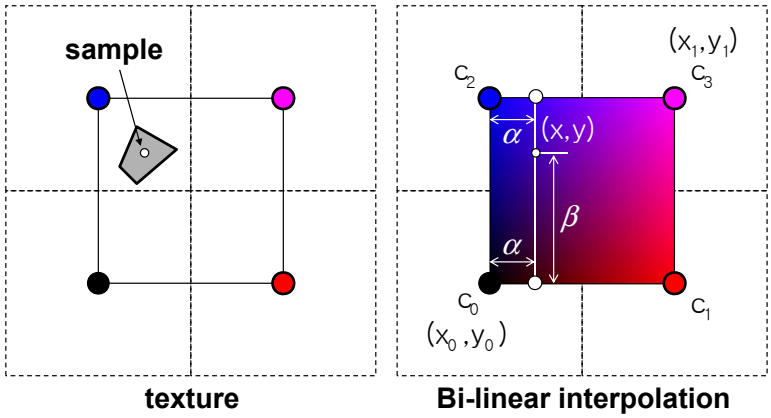

Figure 9.4: The left image shows the case of oversampling. The sampling in the texture space is too small compared to the texture resolution. The right image shows the bi-linear interpolation to address the oversampling problem.

9.2 *Oversampling of Textures*

Let's take a look at the right image of Fig. 9.3. A box-shaped fragment generated for rasterizing a triangle maps to a quadrilateral, i.e., a polygon with four sides, instead of the uniform box shape. This phenomenon occurs due to various transformations (e.g., modeling and projective transformations) and the angle of the triangle against the view direction.

Oversampling occurs when we zoom in the triangles.

Since the pixel in the image space does not match with that in the texture space, we have two cases: oversampling and undersampling cases. Oversampling refers to the case where the sampling resolution in the texture space is smaller than the available resolution of the texture. Fig. 9.4 shows this oversampling case. The quadrilateral in the texture space is even contain in a texel of the texture. The oversampling issue occurs when we magnify the geometry.

We aim to reconstruct the original signal out of available texture samples and compute the signal value at the sampled texture location.

Please take a moment to think about how to compute the representation color for the quadrilateral. Surprisingly, this kind of issues is quite common in computer graphics, image processing, etc. A simple method is to identify the nearest neighbor texel center and use its color for the quadrilateral. In the case of the left image of Fig. 9.4, the blue pixel is the closest to the quadrilateral, more exactly, the sampling point location. Note that during the rasteriation, we compute colors or other attributes based on center positions of pixels.

While this nearest neighbor approach is quite fast, its visual quality is poor, especially along the boundary of texels. In other words, when two sampling locations are very close, but are in different texels, they get different colors, resulting in visual gaps in the image space (Fig. 9.5).

Another approach is to use linear interpolation. Given the sam-

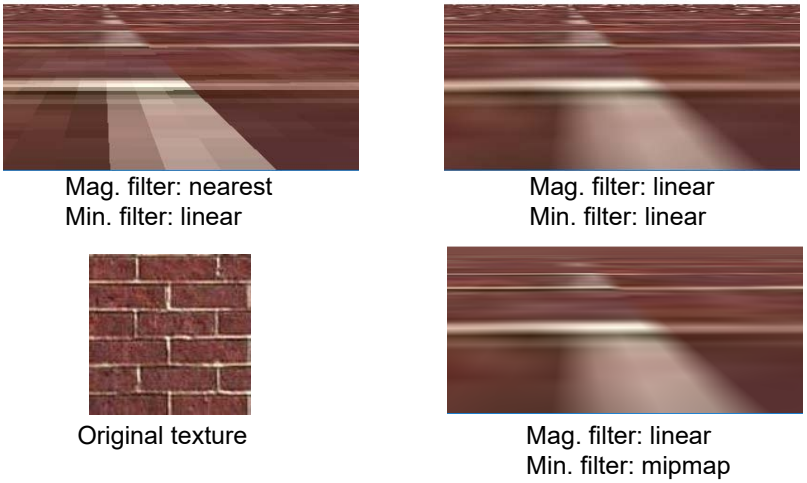

Figure 9.5: Different rendering results with different mag. and min. filters for texture mapping.

pling location, we identify four nearby texels, e.g., c_0 to c_3 in the right image of Fig. 9.4. We then perform the linear interpolation along the U and V texture directions, thus named as bi-linear interpolation. Let us define α and β to be a blending factor along X and Y directions, respectively. They are then defined as the following:

$$\alpha = \frac{x - x_0}{x_1 - x_0}, \beta = \frac{y - y_0}{y_1 - y_0}. \tag{9.1}$$

The color, c, at the sampling location under the bi-linear interpolation is computed as follows:

$$c = (1 - \beta)\left((1 - \alpha)c_0 + \alpha c_1\right) + \beta\left((1 - \alpha)c_2 + \alpha c_3\right). \tag{9.2}$$

The effect of using the bi-linear interpolation over the nearest neighbor one is shown in Fig. 9.5. We can see that boundary shapes of texels were smoothened. Nonetheless, we can also see that the edge information inherent in the original texture was filtered out too. We can thus see that there are trade-off in terms of filtering unnecessary edges and preserving original edges. This boils down to the classical reconstruction and sampling problem.

9.3 *Under-sampling of Textures*

Under-sampling arises when we zoom out the geometry, and thus a fragment of a triangle maps to a large area in the texture space.

Let us know discuss the other sampling issue, undersampling. Undersampling rises when we zoom out from the geometry and thus each triangle become small in the image space. Therefore, a pixel of a triangle maps to a large quadrilateral area in the texture space. The problem is thus to compute a representative color out of many texels covered by the quadrilateral.

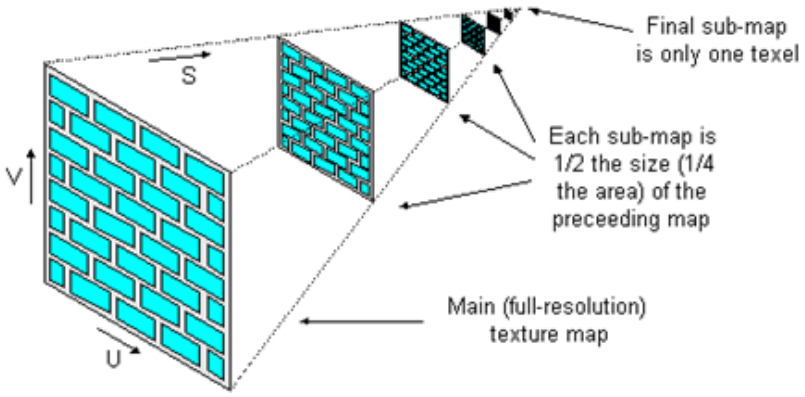

Figure 9.6: This shows a mipmap or image pyramid of an image.

A naive approach to the undersampling is to compute all those texels under the quadrilateral and compute a representative color value, e.g., the average value of them. This approach, unfortunately, slows, since it requires us to access many texels and computations. Instead of this on-demand approach, pre-filtering that pre-filters the original texture in a way to efficiently handle undersampling has been more widely used and studied. In this section, we discuss two approaches, mip mapping and summed area table, for the undersampling problem.

Mipmap or mipmapping is a multi-scale representation for a texture (or any other types of images) to efficiently handle the undersampling issue. Given an input image, a mipmap is composed of a sequence of images whose U and V resolutions are reduced half over its higher resolution (Fig. 9.6). As a result, mipmap is also called an image pyramid. Each low-resolution image is a pre-filtered version of its higher one.

At runtime when we use the mipmap, we pick a particular image level among the available image resolutions of the mipmap given the required texture resolution. If necessary, we can also perform interpolation between two image resolutions, resulting in tri-linear interpolation for computing a color for the sampling location. In whatever cases, we access only a few samples on the mipmap and get pre-filtered texture values, resulting in faster and better visual quality.

The memory requirement of using a mipmap is $\frac{1}{3}$, about 33%, since the total size of using the mipmap is computed as the following:

$$\sum_{0}^{\infty} \left(\frac{1}{4}\right)^{i} = \frac{1}{1-\frac{1}{4}} = \frac{4}{3}. \tag{9.3}$$

Fig. 9.5 shows different rendering results w/ linear filtering or

mipmap. By using the mipmap, we get smoother image results over linear filtering for far-away regions where we minify the geometry. Again, the mipmap is a fast way of handling the undersampling problem, but can remove the original edge information.

A reason why the mipmap produces a over-smoothed result is that the mipmap computes its image pyramid only based on an isotropic filtering shape, i.e., square shapes. As a result, when we have have an very elongated quadrilateral shape in the texture space, we cannot find filtering resolutions along both U and V directions. A solution to this case of anisotropic filtering is the summed area table.

Summed-area table. A summed-area table is proposed to support anisotropic filtering, specifically, a rectangular shape, not the squared shape, on the texture space. Given a texture, $T(u, v)$, its summed-area table, $S(u, v)$, is computed by summing all the elements whose elements are smaller than u or v:

$$S(u, v) = \sum_{i \leq u \wedge j \leq v} T(u, v). \tag{9.4}$$

We then compute the average color value, c_a, on a rectangular regions, e.g., the blue region given by $[u_0, u_1] \times [v_0, v_1]$ as shown in Fig. 9.7, as the following:

$$c_a = \frac{T(u_1, v_1) - T(u_1, v_0) - T(u_0, v_1) + T(u_0, v_0)}{(u_1 - u_0)(v_1 - v_0)}. \tag{9.5}$$

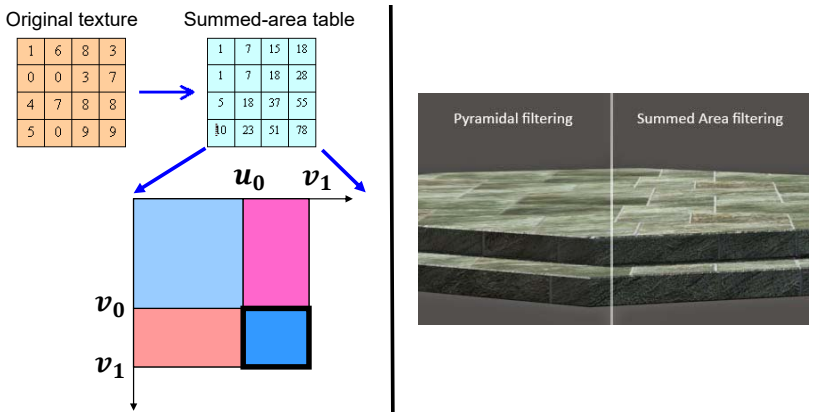

Figure 9.7: The left images show a configuration of the summed area table, while the right image shows rendering results of the summed area table and mipmapping. The right image is created by Denny.

Fig. 9.7 also compares the rendering results computed by the mipmap and summed-area table. The summed-area table shows better quality, since it provides anisotropic filtering. Nonetheless, it has additional runtime and memory overheads over the mipmap.

9.4 *Approximating Lights*

It is easy to paint on images and capture images than constructing geometry, and thus textures have been widely used for various applications. In this section, we discuss two techniques, shadow mapping and environment mapping, of using textures for approximating complex lights.

Before we move on to them, let us first discuss light maps. Light maps are images that contain light intensity. We then use these light maps as textures for adjusting colors of triangles. A simple method of computing colors with a light map is to multiply the intensity contained in the light map with the color computed by illumination or other functions. Fig. 9.8 shows an example of using textures and light maps. Creating complex lighting effects requires high computation, and thus pre-computing, also called baking, them in light maps are still commonly used in many interactive applications.

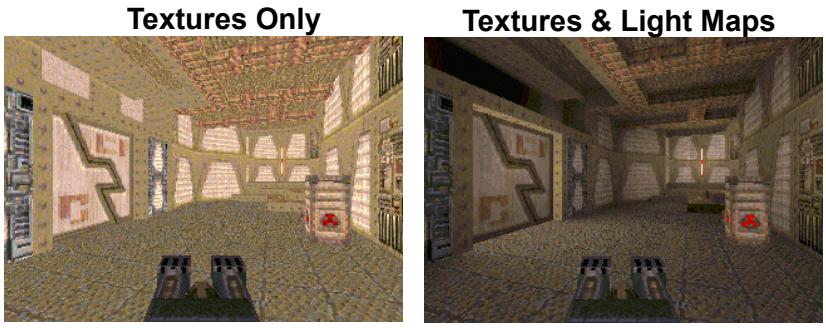

Figure 9.8: This shows results only with textures and both with textures and light maps used for a game called Quake.

9.4.1 *Shadow Mapping*

Shadow is one of fundamental lighting effects that we can see in daily life, and provide various 3D depth cues. While providing shadows is important, it is not that easy to efficiently and correctly generate shadows in rasterization. This problem has been studied for many decades and shadow mapping as a type of texture mapping is proposed for creating realistic rendering results within the rasterization framework.

Please recall our discussion on the Phong illumination model (Ch. 8.3). The model has three components of ambient, diffuse, and specular terms. Unfortunately, diffuse and specular terms do not consider any other objects that block lights from light sources, while the ambient term is a drastic simplification by using a constant for considering inter-reflection. Essentially, the Phong illumination model does not consider the case of having shadows, i.e., the existence of

other objects that block the light. This is mainly attributed since the Phong illumination model, more importantly, rasterization itself, is a local model that mainly aims for high efficiency.

Our challenge is to generate shadows within the rasterization framework. While considering shadows itself requires us to access other objects, resulting in global access on various data, we approach this problem as a two-pass algorithm using shadow mapping. Its main concept is shown in Fig. 9.9.

Shadow mapping is a two pass rendering method to generate shadows without global and random access on other objects.

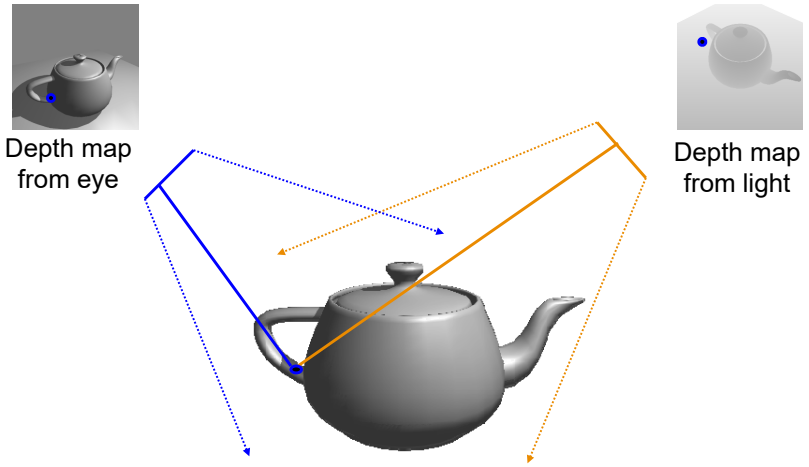

Figure 9.9: This visualizes the process of using shadow mapping to generate shadows on the rendering result seen by the eye.

The problem of the rasterization process is that when we perform an illumination on a fragment of a triangle, we cannot know whether the fragment can receive the light energy from a light source. When we do not receive the light energy due to a blocking object, we add only the ambient term, since the diffuse and specular terms become zero. To know whether a fragment can receive the light energy from a light source, we rasterize the whole scene at the position of the light source and treat its depth map as a shadow map for the light. This is the first-pass of generating shadows.

The depth map generated from the light position contains depth values of visible geometry from the light. We then raster the whole scene at the viewer's position, similar to the regular rasterization process. This is the second pass of our method. A difference in this second pass compared to the regular rasterization is that we check whether we can receive the light energy on a fragment that we generate at the second pass.

To know whether the fragment receives the light energy or not, we compute its depth from the light position, d_l. When d_l is bigger than the stored depth value, d, of the shadow map, we determine that the fragment cannot receive the light energy and we thus give only the

ambient term to the fragment, not the diffuse nor specular terms.

While we explain shadow mapping in a concise manner above, there are a lot of technical issues. Most of them are related to the oversampling and undersampling that we discussed for texture mapping; note that the shadow map is another type of textures and thus inherits issues of texture mapping. Nonetheless, it is very important to understand how we address a kind of global illumination, shadow generation, through a texture, the shadow map.

Shadows maps are just a type of textures and thus inherits pros. and cons. of texture mapping.

9.4.2 *Environment Mapping*

In the prior section, we discussed how to generate shadows using shadow mapping. Another common rendering effect is to support reflection on mirrors or other metal-like objects. For those models, we see other objects reflected on such reflecting objects. In other words, supporting this effect belongs to a type of global illumination requiring the access to other objects.

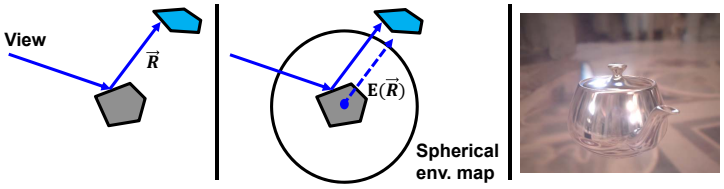

Figure 9.10: Two images in the left visualize how we use the spherical environment mapping, while the right image shows an example of using the environment map to simulate the reflection effect.

The rasterization framework also relies upon using another type of texture mapping, environment mapping, for this reflection effect. Suppose that we have a view direction on a reflecting object shown in the gray color in Fig. 9.10. When the object is the specular object, the reflected ray, $\vec{R}$, is computed by the Snell's law (Ch. 8.3). We then need to access an object along the reflected ray, $\vec{R}$. Unfortunately, this is a ray tracing process, and is not efficiently adopted for rasterization.

To enable the reflection efficiently, we introduce environment mapping, which captures colors of the surrounding environment in a texture. For environment mapping, we can use different types of geometry capturing the environment. Examples include sphere, cube maps, etc. In this chapter, we explain environment mapping based on a sphere for the sake of the simplicity.

An environment map captures surrounding geometry or lights, and can be used as a texture to approximate them at runtime.

As shown in the middle image of Fig. 9.10, we place a sphere at the center of the reflection object. We map the sphere into a 2D texture space; since we can represent the sphere with two angles, θ and ϕ, the 2D texture space can be constructed by these two angles. We then generate a ray starting from the center of the sphere to each texel of the sphere and encode the color of the ray at that texel; we

use a projection instead of ray tracing for efficiently building the map.

At runtime, when we raster a triangle of the reflecting model, we know the viewing direction, and thus identify a texel ID that the reflection ray from the center of the sphere, $E(\vec{R})$, will access. Unfortunately, since the environment map is generated at the center of the object, not each location that we have reflection, there are visual gaps between the computed one and the ground truth. Nonetheless, we can support an approximate reflection by using an additional texture.

The environment map is also used to encode complex types of lights and used for providing realistic lighting for rasterization.

9.5 *Approximating Geometry*

Textures are also used to approximate complicated geometry. Especially, when we have many geometry, it requires long running computation time with high memory requirement. A single or multiple textures are effective ways of approximating them with reduced running and memory overheads.

Figure 9.11: We use the bump map (shown in the middle) to adjust normals of the geometry during the rasterization, to enrich the appearance of the model (shown in the right.) Since we do not change the actual geometry, we can see that the geometry is unchanged at its silhouette.

Bump and normal mapping. Bump mapping modifies normals of geometry, not the actual geometry. The texture used for bump mapping encodes an amount of changes to normals of the geometry (Fig. 9.11). This is an approximate, yet effective way of enriching the geometry. Nonetheless, we can observe that the actual geometry is not aligned with the adjusted normals, especially when we look at the silhouette of the object. Normal mapping is similar to bump mapping, but the normal map directly gives the normal that we use on top of a simple geometry (Fig. 9.12).

Displacement mapping. Unlike bump and normal mapping, displacement mapping adjusts the actual geometry based on a provided displacement map. A common usage of displacement mapping is to encode a height change on the displacement map and adjust the geometry along its normal direction according to the height. Adjusting the geometry requires tessellation, subdividing the geometry into

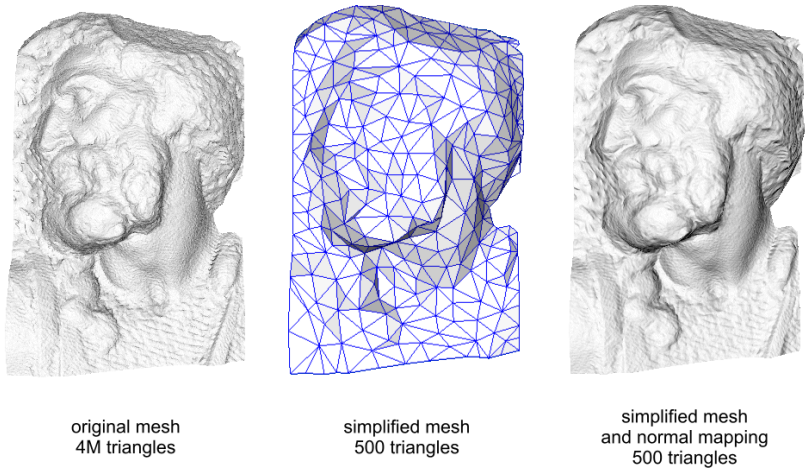

Figure 9.12: We can provide detailed look on a simple geometry by using normal mapping. The image is created by Paolo Cignoni.

smaller patches and adjusting them to accommodate the given height (Fig. 9.13).

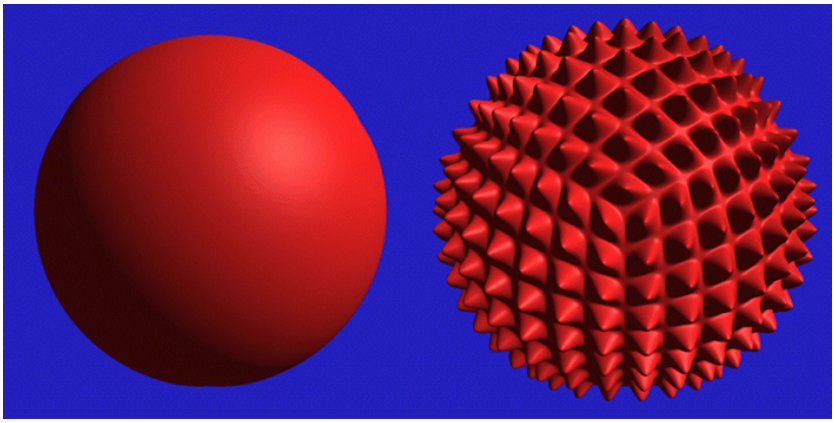

Figure 9.13: Displacement mapping changes the actual geometry according to its map unlike bump mapping. To enable displacement mapping, we tessellate the initial geometry into smaller ones.

We covered only a few examples of approximating geometry. Other notable examples include 3D or solid textures representing 3D shapes and billboards, which are a set of 2D textures representing complex geometry (e.g., trees).