Bump Mapping

Andrea Fumagalli

June 2025

1 Introduction

With the term bump mapping, we indicate a family of small-scale techniques used in real-time graphic programming to give a more 3D appearance without adding any geometry. They are usually implemented in the fragment shader and make use of some parameters read from a texture. What we aim to model with bump mapping is called meso-geometry, a definition that includes everything in between macro-geometry (represented by vertices and triangles) and micro-geometry (encapsuled in the shading model which simulates rough or smooth surfaces), so for example, musculature details, folds and seams on clothing, rocks or bricks in a wall. All these techniques adjust the shading parameters at the pixel level in such a way that the viewer perceives small perturbations away from the base geometry, which actually remains flat. The key idea is that, instead of using a texture to change a color component in the illumination equation, we access a texture to modify the surface normal. The geometric normal of the surface remains the same; we merely modify the normal used in the lighting equation. The result is an illusion that changes the perception of the triangle surface itself, without modifying its geometry. In doing so, we can significantly boost the visual effect with a small impact on performance. The original idea for this was introduced by Blinn in 1978; since then, different approaches varying for texture usage, level of detail or complexity have been developed. In this project, we will implement some of them using OpenGL to compare and evaluate them.

2 Scene

Since the goal of the project is to showcase the different bump mapping techniques, we choose to realize a very simple scene composed of three objects of different complexity: a plane, a sphere and a pot. We used the same functions used in class to move the camera around in the scene, and we implemented similar ones to be able to move the lights in the scene, a very important feature to be able to appreciate the 3D effects simulated by the shaders.

3 Shaders

3.1 Basic shader

The basic shader implements the Blinn-Phong reflection model handling repetitions and more than one light. The vertex shader (*basic.vert*) simply computes the light direction and view direction in world space and passes through the UV coordinates and the normal (after transforming it in world space) of the vertex. In the end, it applies the view-projection transformation to the vertex position. The fragment shader (*basic.frag*) implements the Blinn-Phong illumination model by computing diffuse, ambient and specular components and combining them to obtain the fragment color. The diffusive color is sampled from a diffusive texture.

3.2 Blinn's bump shader

The Blinn shader implements the technique proposed by Blinn for bump mapping. The idea is to make use of two parameters: b_u and b_v, which correspond to the amount to vary the normal along the u and v image axes. The parameters b_u and b_v are calculated from a height map, a monochrome texture in which each value represents a height, so in the texture, white is a high area and black a low one. To obtain the parameters, we take the differences between neighboring columns to get b_u, and between neighboring rows to get b_v. The perturbed normal is obtained by doing the cross product between the perturbed bitangent and tangent, two vectors perpendicular to the normal that point respectively in the u and v direction. We read the original tangent and bitangent from the mesh and perturb them by adding the normal scaled by the parameters calculated from the sampled height map as expressed by the formula:

$$P'(u,v) = P(u,v) + B(u,v) \cdot N$$

Where $P(u,v)$ is the tangent-bitangent plane, $B(u,v)$ are the parameters and N is the original normal. In the end, we use the cross-product to obtain the new normal.

The vertex shader (*blinn_bump.vert*) is identical to the basic one, except that now we also pass through tangent and bitangent in world coordinates. The fragment shader (*blinn_bump.frag*) computes the perturbed normal as described before and uses it in all computations. The actual computations require two other hyper parameters to be set: the offset at which we sample the adjacent height and a scale factor to emphasize the 3D effect. The offset can be a value between 1 and 1/texture size, we choose this last option (approximately 0.0005 in our case), but a fine tune for each specific texture would probably produce better results without strange artifacts. The scale factor is necessary because in case of a difference of 1 in height between two adjacent samples we would want to obtain a rotation of 90° of the tangent/bitangent while, using a scale factor of 1 we obtain a rotation of 45° due to the use of the addiction operation. The scale factor can therefore vary between 1 and infinite, a small value produces

a faint 3D effect while a big value causes every small variation in height to be emphasized with a big shadow. In our application a value of 100 gave the best results.

3.3 Normal shader

A more common way to do bump mapping, especially nowadays, is normal mapping. The main difference from Blinn's bump mapping is that we read directly the normal from a texture: the normal map. These normals are expressed in tangent space a reference system oriented as the surface of the mesh, defined by normal, tangent and bitangent. This allows for deformation of the surface and maximal reuse of the normal texture. This also means that we will do all our computation in tangent space instead that in world space as we did before, and so we need to build a transformation matrix (called TBN matrix and build using normal, tangent and bitangent) to transform everything we need. Another specification of the normals from a normal map is that they are mapped to [-1,1], but we read them in range [0,1] (ex: the up direction: [0,1,0] is encoded as [128,255,128], which we read as [0.5,1,0.5]), so we need to transform them and to do so we multiply for 2 and subtract 1. The vertex shader (*tangent.vert*) calculates the light direction and the view direction, this time in tangent space. We don't pass through the normal since we will read it from the normal map; instead we build the TBN matrix using normal, tangent and bitangent read from the model on which we applied model transformations. The fragment shader (*normal.frag*) is the same as before except that now we sample the normal to use in the computations form the normal map for each fragment.

3.4 Parallax shader

The main problem with Blinn's bump mapping and also normal mapping is that the illusion of 3D is lost when looking at the surface from an angle, this happens because the effect is independent from the view direction. Parallax mapping is a technique that tries to make up for this. The key concept is that we want to alter the UV coordinates for each fragment so that instead of having the UV coordinates, we would get on a flat/smooth surface; we consider the UV we would get if the surface of the mesh was displaced according to the height map.

Parallax occlusion mapping does this in a more refined way: it takes numerous samples at increasing depth on the view direction, searching for the point where the view direction crosses the 'displaced' volume represented by the height map. Once it finds it, for a smoother result, it interpolates between the UV corresponding to that point and the previous one. The number of layers in which we subdivide the height determines how smooth and refined the result is; more layers also mean more computation. To mitigate this, we pick a variable number of layers between a max value and a min value according to the angle between the view direction and the Z direction (in tangent space). This way for fragments that we look at from the front and where little displacement

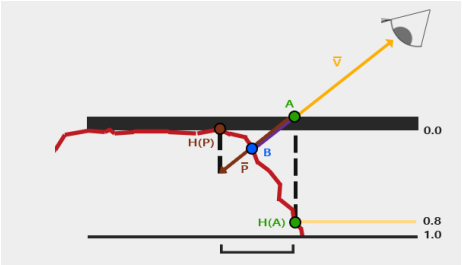

Figure 1: Parallax Mapping

happened, we use a few layers while for fragments that we look at from the side, we use more layers.

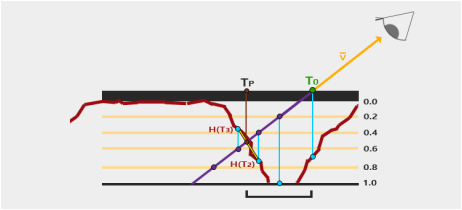

Figure 2: Parallax Occlusion Mapping

As the vertex shader we use again *tangent.vert* because we need to do the computations in tangent space since we will use normals from a normal map. The fragment shader (*parallax.frag*) firstly uses the *OcclusionParallaxMapping* function to calculate the displaced UV as explained before, then uses these new UV coordinates to sample from normal and diffuse maps. The other light computations follow as usual.

3.5 Displacement shader

The last technique we implemented isn't exactly a bump mapping technique since we are actually going to modify the mesh surface, but we included it in this work to show the highest level of meso-geometry definition achievable starting from a low-detailed surface and using only textures to modify it. With displacement mapping we are going to displace the vertex of the mesh according the values we read from the height map. The problem is that meshes, especially simpler ones, have a low number of vertices and triangles, so if we apply displacement mapping on them without any previous elaboration the results are going to be really poor. Instead what we are going to do is apply tessellation. Tessellation is a technique in which each triangle of a mesh is subdivided at run time into smaller triangles which vertices are generated interpolating the original vertices of the triangle. This way we can achieve a higher level of detail

without needing a high-poly mesh. Once we tessellated the mesh we can displace the newly created vertices adding to their position the their normal scaled by the height value sampled from the height map. To achieve tessellation some changes were needed outside of the shaders: we modified the Draw method of the model and mesh classes so that it requires a Boolean input that specify if we are going to render with or without tessellation, in which case we are going to render not triangles, as we usually do, but patches, specifying it with a proper parameter in the *glDrawElements* function. Also in the main function, before the rendering loop we need to specify that we are going to tessellate using a triangular patches and not quads by using the *glPatchParameteri* function. The vertex shader (*displacement.vert*) has almost no work to do, because we are using tessellation, it simply passes through position, UV, normal, tangent and bitangent. Using tessellation, we need to implement two more shader than usual: Tessellation Control Shader and Tessellation Evaluation Shader. The tessellation control shader (*tessellation.tcs*) has two tasks: the first is to pass through the data coming from the vertex shader, so vertex position, UV, normal, tangent and bitangent. These values are grouped in arrays of the dimensions of the primitives we are working with, in our case triangles and so the size is 3. The second task is to control the level of tessellation to apply from which the name of the shader. To do so we need to specify 4 parameters called tessellation levels: 3 of which (*gl_TessLevelOuter*) define in how many section we are going to split the sides of the triangle and one (*gl_TessLevelInner*) which defines in in how many section to split the height of the triangle. These values define how many triangles we will obtain from the original one, so they have a key role in the complexity that we will get. So we made them adaptive in two different ways: firstly, we define a max a min tessellation level, we calculate them so that the tessellated mesh will not have less then 5k triangles and no more of 500k triangles (these values can be tweaked according to the desired performances). We do so by passing to the shader the original number of triangles of the mesh red with a newly implemented method (*numFaces()*) and considering that a tessellation level L produces about $L * (L + 1)$ triangles from one. Done that, we choose the effective tessellation levels by interpolating between max and min level according to the distance between the camera and the triangle obtaining a rudimentary LOD effect. The tessellation evaluation shader (*tessellation.tes*) is where the actual parameters of the new texels are calculated, in a way, it can be seen as the equivalent of the vertex shader, but for texels. We calculate position, UV, normal, tangent and bitangent all in the same way: by interpolating the values of the vertices of the original triangle that we are subdividing, weighted according to the position inside of it of the texel. Done this we displace the texel position by adding to it its normal multiplied by the height sampled from the height map scaled by a *height_scale* parameter. If we stopped here, the mesh would be correctly deformed, but its normals would still be the ones of the smooth surface, making all the previous work almost unnoticeable. To calculate the displaced normals we use the Blinn's bump method including the *height_scale* parameter used before in the calculations so that the simulated behavior of the light change accordingly to the effective bumps' height.

4 GUI

We implemented a really simple GUI using *Dear ImGui* to allow an easy interaction with some of the parameters. The GUI consists of three menus:

- **Blinn-Phong parameters**, here we can access all the parameters involved in the illumination model.
- **Object appearance**, here we can modify the appearance of the rendered objects changing shader, diffusive texture, height, texture repetitions, etc.
- **Light panel**, here we can interact with the lights, creating or destroying them, and selecting the one to move.

Since the camera movements and the GUI both rely on the mouse movements, they would end up conflicting. We implemented a simple way to alternate between *move_mode* and *GUI_mode* by holding the spacebar.

5 Performances

We implemented a very rudimentary way to calculate the FPS and we display them in the window title. The rendered scene is really simple, still on a low/mid end PC (Ryzen 3 2200G, GTX 1650 Super, 16Gb RAM), rendering in 1920x1080 we achieve steadily 480 FPS (at which we capped the application) with every shader. Some drops in FPS are noticeable using the displacement shader and texture repetition, we can get as low as 250 FPS, still way over the 60 FPS standard (and even 120 or 240 FPS considering some more recent standards).