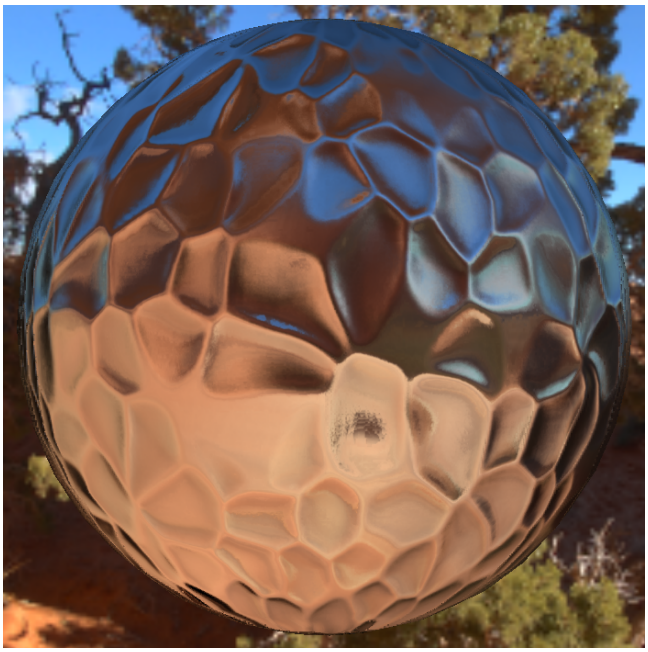
Bumped Environment Mapping for Anisotropic Shaders

Federico Marcolongo, 940085

November 29, 2020



Abstract

Anisotropic shaders need information about the whole tangent space orientation to render illumination models, as opposed to their isotropic counterpart, which only needs information concerning the normal direction to the rendered surface. This implies new challenges to address in the implementation of a real-time application, as well as many possibilities to render effects isotropic shaders couldn't describe. Keeping in mind the requirements a real-time rendering process needs to run smoothly, I try to make the best use of the aforementioned possibilities while maintaining accetpable performance.

Contents

1 Introduction **3**
- 1.1 About this document . . . 3
- 1.2 Notation . . . 4

2 Quick Overview of Anisotropic Illumination Models **4**
- 2.1 Anisotropic Shading as opposed to Isotropic Shading . . . 4
- 2.2 Heidrich-Seidel . . . 5
- 2.3 Ward . . . 6
- 2.4 Ashikhmin-Shirley . . . 7

3 Bump Mapping **7**
- 3.1 From vertex-level to fragment-level detail . . . 7
- 3.2 Tangent Space . . . 8
- 3.3 Normal Mapping . . . 9
- 3.4 Quaternion Mapping . . . 10
- 3.5 Tangent Plane Rotation (TPR) Mapping . . . 14
- 3.6 Parallax Occlusion Mapping (POM) . . . 16

4 Environment Mapping **17**
- 4.1 Overview . . . 17
  - 4.1.1 The Epic Games method for isotropic specular EM . . . 18
  - 4.1.2 Adapting the Epic Games method to anisotropic models . 19
- 4.2 Diffuse Component . . . 20
- 4.3 Specular Component . . . 20
  - 4.3.1 Why Ashikhmin-Shirley? . . . 20
  - 4.3.2 Importance Sampling . . . 21
  - 4.3.3 BRDF integration . . . 23
  - 4.3.4 Real-Time Monte-Carlo integration of environment luminance . . . 27

5 Implementation Details **28**
- 5.1 Languages and Libraries . . . 28
- 5.2 File System organization for Asset import . . . 28
- 5.3 Compiling and using the pre-processing applications . . . 30
- 5.4 Compiling and running the rendering . . . 31

6 Results and Performance **32**
- 6.1 Rendered Models . . . 32
- 6.2 Performance and Quality analysis . . . 33

1 Introduction

1.1 About this document

Implementing anisotropic illumination models with support for both environment and bump mapping required me to span many different topics on top of the OpenGL implementation itself. As a consequence, to keep the description of the project I implemented as concise as possible, I had to make the choice of glossing over most topics for which an outside reference is available. In those cases, a quick description is provided to explain how that particular aspect intervenes in the project and then a reference is specified for further reading. This let me focus more on the explanation of topics I had to come up with to make the project work. Having no other reference for such topics, I wanted to make sure that the explanation of mathematical models and implementation had all the space it required.

To better understand the structure of this document, here is a quick overview of its parts:

Section 2 focuses on anisotropic illumination in general and a few relevant models, describing their main purpose and the distribution function they center on. Further information can be found in the original papers[1], while more detail is given only where it matters for other parts of the project.

Section 3 describes bump- and displacement-mapping, going faster on normal- and parallax occlusion-mapping, while leaving more space to quaternion- and tangent-space-rotation-mapping, where a full explanation of the underlying mathematical model and of the choices made to implement them is needed. This section is fairly mathematical due to its relation to the geometry of surfaces.

Section 4 addresses the issue of implementing environment mapping when working with anisotropic models. It focuses most on the implementation choices I had to make to adapt the isotropic method it is based on, for which it provides references to allow the reader to have a more complete picture of the method. This section covers topics spanning from numerical integration to rendering optimizations and is therefore more technical than the previous.

Section 5 focuses more on the actual implementation, providing information on the programs making the project work and the required structure of its file system. It details how to build and use the programs generating the textures required by the main rendering application, as well as how to build and use the application itself.

Section 6 finally presents the end results with a few rendered models, showing off some of the capabilities of the complete project, whereas pictures in the previous sections were more aseptic, having to underline a specific technical aspect each. It also spends a few words on the overall performance of the rendering, including some stress-testing.

[1] They can be found in the bibliography

1.2 Notation

Throughout this document I stick to the same notation, except where explicitly specified, which is as follows:

- **v**: bold fonts denote vectors
- (**vw**) denotes the dot product of vectors **v** and **w**
- **t**, **b**, **n** are respectively the tangent, bitangent and normal (unit) vectors on a point of the surface of a model
- **v** is the view direction. It is a unit vector that always points from its base point on a surface to the observers's position
- **l** is the light direction. It is a unit vector that always points from its base point on a surface to the position of the light source
- **h** is the **v**-**l** half-vector (more properly the "half-way vector"). It is a unit vector depending on the view and light directions and it's computed as $\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$

Parts of this document referring to implementation details *outside* of section 5 are framed in a box just like this one.

```
Code excerpts are shown like this, with partial "syntax hilighting"
```

Folders', files' and programs' names are all written in `typewriter font`.

2 Quick Overview of Anisotropic Illumination Models

2.1 Anisotropic Shading as opposed to Isotropic Shading

The Bidirectional Radiance Distribution Function (BRDF) is, in the phisical description of the illumination of surfaces, the function describing how a particular material emits back the light it receives from the environment. The graph of this function is 4-dimensional, as its domain and codomain are both bidimensional hemispheres. The description of such a function for any possible kind of actual physical material is a very complex task, which is why many different models have been proposed in computer graphics to strike a balance between physical accuracy and computational complexity, all the more so in the case of real-time graphics processing.

The two realms of illumination models I mean to discuss, as per the title of this subsection, derive their differences precisely from the intention of describing real life materials. In particular, they make different assumptions concerning the

overall orientation of microscopical imperfections on the surface of these different kinds of material. Where isotropic models mean to describe surfaces whose imperfections have radial symmetry (given also their abundance in nature), their anisotropic counterpart concerns itself with the description of surfaces where these imperfections are aligned along a particular direction than another. This is the case, for example, in the natural occurrences of brushed metal, hair, vynil records and CDs (whose grooves have a concentrical orientation), and surfaces varnished with a stroke of coating.

Given the dependence of the re-emitted light on the orientation of these imperfections, an anisotropic illumination model needs to account for the orientation of the tangent plane to the surface (the direction of its tangent and bitangent vectors, as described in subsection 3.2), and not just for the relative position of the view- (or incident light-) vector to the surface normal vector - which is the case for most isotropic models.

Below are presented a few models addressing the issue of anisotropic surfaces. Since the models are fairly old (in terms of computer graphics development history) we have to keep in mind that the simplest (and least physically accurate) of them, the Heidrich-Seidel [HeSe98] model was developed for real time applications running on 1990s GPUs, and consequently concentrates on having the fastest implementation possible, while the Ward [Ward92] and Ashikhmin-Shirley [AsSh00] models where developed with (offline) ray-tracing applications in mind. The fast development of GPU processing power during the last two decades is what allows me, in this particular instance, to use these models for demanding tasks, even on real-time applications. This project will be mainly focused on the Ashikhmin-Shirley model, for reasons that will be discussed later in subsection 4.3.1

2.2 Heidrich-Seidel

This Phong-like model from 1998 means to describe anisotropic surfaces as (2D) bundles of one-dimensional strands (curves in 3D space) each having a tangent vector in each of its points. Given a point on a surface, it uses this tangent T and the light-incidence direction L to calculate a "shading normal vector" N' which corresponds to the normal vector used for rendering one-dimensional strands. This vector is obtained implicitly as

$$\langle L, N' \rangle = \sqrt{1 - \langle L, T \rangle^2}$$

which represents the diffuse component of the model. Here and below, following the authors' notation, $\langle V, W \rangle$ represents the dot product of vectors V and W. Similarly, the dot product of the view vector V and the reflected light vector R (against N' instead of the surface normal N) is found to be

$$\langle V, R \rangle = \sqrt{1 - \langle L, T \rangle^2}\sqrt{1 - \langle V, T \rangle^2} - \langle L, T \rangle \langle V, T \rangle$$

These values are then insterted in a Phong-like (-direct-light) equation representing the model:

$$I_o = k_a I_a + \left(k_d \langle L, N' \rangle + k_s \langle V, R \rangle^n\right) I_i$$

- k_a, k_d, k_s are respectively the ambient, diffuse and specular coefficients
- I_o, I_a, I_i are respectively the observed, ambient and incident light intensities
- n is the Phong shininess coefficient

This model is clearly not bidirectional, it doesn't account for surface roughness, and doesn't conserve radiant energy, just like the (isotropic) Phong illumination model it is based on. Despite this, the authors describe a method to preprocess lookup textures storing the values of the square root-terms involved in the calculations, achieving the goal of the paper in terms of assuring a fast real-time rendering of anisotropic properties.

2.3 Ward

This hybrid empirical- and theoretical- model by Ward means to offer a physics-based, energy conserving, fully bidirectional anisotropic BRDF along with an explicit method for importance sampling of the halfway vectors **h**, for BRDF integration purposes.

It is parameterized by diffuse and specular coefficientsρ_d and ρ_s and directional roughness coefficients α_x (along the tangent direction) and α_y (along the bitangent direction), all of which have a physical interpretation and can be used to fit empirical data obtained from sampling the BRDF of actual materials in a lab setting.

The model's BRDF is then built around a Gaussian distribution function with elliptical simmetry:

$$f(\mathbf{v}, \mathbf{l}) = \frac{\rho_d}{\pi} + \rho_s \frac{1}{\sqrt{(\mathbf{nv})(\mathbf{nl})}} \frac{1}{4\pi\alpha_x\alpha_y} e^{-2\frac{\left(\frac{(\mathbf{th})}{\alpha_x}\right)^2 + \left(\frac{(\mathbf{bh})}{\alpha_y}\right)^2}{1+(\mathbf{nh})}}$$

Although the lighting model is solidly grounded on empirical data and a widely accepted distribution for microsurface orientation, it intentionally lacks geometry visibility and Fresnel reflectance terms. This is what most significantly separates it from the class of "modern" PBR (phisically-based rendering) shaders.

For the purposes of this project, it should also be noted that the Ward model, although it is endowed with an importance sampling method for the generation of the half-vectors h, doesn't offer an explicit distribution function for the said vectors, which would be useful in a Monte-Carlo setting (as pointed in [AsSh00]).

2.4 Ashikhmin-Shirley

This mostly theoretical Phong-based model proposes a bidirectional, energy conserving, Fresnel-reflectance weighted anisotropic specular BRDF. It also proposes an energy conserving diffuse term to use along the specular model, instead of a lambertian one. This diffuse term won't however be the focus of this project.

The specular BRDF is parameterized by two directional shininess coefficients n_u and n_v, which split the role of the Phong shininess coefficient n in the tangent and bitangent directions.

The model crucially provides both an importance sampling generation process for half-vectors and their explicit distribution function. Since integration of the BRDF is done not in the space of half-vectors but in the space of incident light vectors, the authors provide the explicit relation between the distribution functions of the vectors in the two spaces mentioned above: $p_h(\mathbf{h}) = 4(\mathbf{vh})p_l(\mathbf{l})$ for $\mathbf{l} = 2(\mathbf{vh})\mathbf{h} - \mathbf{v}$ ($\mathbf{h}$ is the half-vector of vectors $\mathbf{v}$ and $\mathbf{l}$).

The authors describe:

$$p_h(\mathbf{h}) = \frac{\sqrt{(n_u+1)(n_v+1)}}{2\pi}(\mathbf{nh})^{\frac{n_u(\mathbf{th})^2+n_v(\mathbf{bh})^2}{1-(\mathbf{nh})^2}}$$

And obtain the full specular BRDF as:

$$f_s(\mathbf{v},\mathbf{l}) = \frac{p_h(\mathbf{h})F((\mathbf{vh}))}{4(\mathbf{vh})\max((\mathbf{nv}),(\mathbf{nl}))}$$

In the original paper as well as for the purposes of this project, the Fresnel reflectance term is estimated making use of the Schlick approximation:

$$F((\mathbf{vh})) = F_0 + (1-F_0)(1-(\mathbf{vh}))^5 = F_0(1-(1-(\mathbf{vh}))^5) + (1-(\mathbf{vh}))^5$$

Anticipating the discussion contained in subsection 4.3.3, the final form of the Ashikhmin-Shirley (specular) BRDF, ready for Monte-Carlo integration with importance sampling, is:

$$f_s(\mathbf{v},\mathbf{l}) = p_l(\mathbf{l})\left(F_0\frac{1-(1-(\mathbf{vh}))^5}{\max((\mathbf{nv}),(\mathbf{nl}))} + \frac{(1-(\mathbf{vh}))^5}{\max((\mathbf{nv}),(\mathbf{nl}))}\right)$$

3 Bump Mapping

3.1 From vertex-level to fragment-level detail

It is common in computer graphics to want to obtain a certain level of detail on the rendered surface without having to indeterminately increase the resolution of the underlying mesh. This is most common in the texturing of a 3D model: the values of the color attribute attribute are removed from the vertices and given to

a proxy, which in the case of UV-mapping (a common technique addressing the issue) is an image whose pixels (named texels) are accessed by their coordinates. The vertices then store their (u,v) position on such image and rasterization takes care of interpolating the attribute over the faces of the mesh.

The most important aspect of this process, for the goal of this project, is that not only can UV-mapping manage a fragment-level description of color, but a fragment-level description of vectors as well. The x, y, z components of a 3D vector are mapped respectively to the R, G, B channels of the texture image (which range between 0 and $2^i - 1$, where i is the bit depth of the image).

To maintain decent resolution in the mapping of vectors, this process works best for vector "fields" whose components are uniformly bounded by "small" constants, which is fortunately the case for the objects closest to our needs when implementing illumination models: unit vectors in 3D space. Their components all range, in fact, between -1 and 1.

UV-mapping is what provides fragment-level precision (albeit with fixed resolution) to all of the calculations we are tasked to perform when implementing illumination models. It is a very powerful instrument in real-time rendering, and one whose preservation is the main driving factor behind many of the (hardest) implementation choices when treating Environment Mapping in section 4.

3.2 Tangent Space

Since when mapping vectors to textures we are not actually mapping directly vector objects, but their components with respect to a certain frame of reference, we want this reference to be the most neutral space we can achieve. The meaning of this is that not only it shouldn't depend on model and view transformations, but it shouldn't depend on the coordinates of vertices in object space either. This is the reason why we express UV-mapped vectors in tangent space, which we define, per vertex, as the span of the plane *tangent* to the mesh in the vertex (which has a precise geometrical meaning) and the direction *normal* to the mesh in that same vertex (with the convention of taking it *outwards-pointing* in the case of a 2-manifold closed mesh)

Thus, to find the vectors which form a basis of this tangent space, we must explain both how to find the *normal* $\mathbf{n}$ vector and the *tangent* $\mathbf{t}$ and *bitangent* $\mathbf{b}$ vectors that form the basis of the tangent plane, to obtain an actual $(\mathbf{t}, \mathbf{b}, \mathbf{n})$ orthonormal frame of reference for the UV-mapped vectors.

Finding the $\mathbf{n}$ vector in a certain vertex V is usually achieved in computer graphics by averaging (with according weights) the *face normals* of all the faces sharing V, which themselves are easily computed through linear operations involving the vertices adjacent to V on each face.

Computing $\mathbf{t}$ and $\mathbf{b}$ is, instead, where UV-mapping comes into play. This is because UV-mapping is the closest thing there is, in computer graphics, to

explicitly determining a chart of the 2-manifold embedded in 3D space represented by the mesh. The process goes like this: the texture image is itself a 2D linear space, whose tangent space (in each texel) is the span of two orthogonal vectors $\mathbf{u}$ and $\mathbf{v}$, respectively aligned along the u and v axis of the texture. Two vectors $\mathbf{t}'$, $\mathbf{b}'$ are constructed on V so that their UV-mapping (mathematically, the *differential* of the mapping) are resp. $\mathbf{u}$ and $\mathbf{v}$. The issue with taking directly $\mathbf{t} = \mathbf{t}'$ and $\mathbf{b} = \mathbf{b}'$ is that $\mathbf{t}'$ and $\mathbf{b}'$are only orthogonal if the UV-mapping is conformal (i. e. it preserves angles) and even so, they are unit vectors only if the mapping is an isometry (i. e. it preserves distances), which it rarely is.

To obtain the orthonormal frame of reference mentioned above, $(\mathbf{t}', \mathbf{b}', \mathbf{n})$ is orthonormalized via the Gram-Schmidt process, setting

$$\mathbf{t} = \frac{\mathbf{t}' - (\mathbf{n}\mathbf{t}')\mathbf{n}}{\|\mathbf{t}' - (\mathbf{n}\mathbf{t}')\mathbf{n}\|}$$
$$\mathbf{b} = \mathbf{n} \times \mathbf{t}$$

where the order of the cross product means the frame of reference will have positive orientation. Notice that, since the setting is 3-dimensional, we didn't actually have to determine $\mathbf{b}'$ for this purpose, but computer graphics usually compute it for completeness.

The $(\mathbf{t}, \mathbf{b}, \mathbf{n})$ frame of reference obtained this way completely describes the space where both the UV-mapping of vectors and lighting calculations will be made in the implementation of this project. The fact that this reference frame is orthonormal grants that the transformation from tangent space to object space is an orthogonal linear transformation, and as such preserves distances (hence dot products). In particular, this transformation is represented by the 3x3 matrix having the reference frame vectors as columns.

3.3 Normal Mapping

Having set the context in which we can use textures to store values of vectors varying on the surface to be rendered, it is now time to mention the most widely used and readily available kind of vector mapping: Normal Maps.

The idea behind this method is that we can mimic the appearance of bumps and indentations over the surface without the need for tessellating and displacing elements of the mesh, achieving the goal instead by perturbing the surface normals at a fragment level.

The image value of the normal vector is store directly at coordinates (u,v) of the lookup texture, making the implementation of the mapping a straight texture read performed in the `fragment shader`.

The value of the normal at each point is then used to perform lighting calculations againts view and light vectors (in this implementation, all of them are first mapped to tangent space).

Figure 1 shows (isotropic) environment mapping with and without normal mapping, as well as the texture used to achieve the effect. Notice the bluish

Figure 1: Normal Maps

(a) no normal mapping (b) with normal mapping (c) the normal map texture

tint of the texture: normal maps describe "small" perturbations about the mesh normal, so each texel stores a vector value that in tangent space is "close" to the tangent space's $\mathbf{n} = (0, 0, 1)^\top$ normal vector (corresponding to a fully blue RGB triplet[2]).

3.4 Quaternion Mapping

Normal maps are enough in all of the cases where information about the normal vector alone is sufficient to perform lighting calculations. But if we were to apply normal maps as is, when implementing, for example, anisotropic shaders, the perturbed normals would reflect bumps over the surface, while tangents and bitangents would still be interpolated from the mesh's own vectors.

In a technical sense, the tangent space basis vectors could not be guaranteed to be orthonormal, and as such couldn't hope to describe an actual reference frame.

This issue, when left unattended, gives birth to weird lighting effects and has to be addressed when bump mapping with anisotropic shaders: not only should normals be perturbed, but the perturbation should involve the whole tangent space in the same way.

One way of solving the issue (on the programmer's part) would be to ask artists authoring PBR materials to provide a tangent- and bitangent-map textures for anisotropic materials, then sampling them just like normal maps to obtain the perturbed versions of the corresponding vectors.

Such method would however amount to both unnecessary work on the artist's part and a threefold increase in memory occupation on the GPU.

[2]Actually, (x, y, z) vectors are mapped to $(r, g, b) = \frac{1}{2}(x, y, z) + \frac{1}{2}$, which means that the up vector in tangent space corresponds to a lighter $(0.5, 0.5, 1)$ blue on the texture

A different approach I think better suits the problem is to consider the normal map authored by PBR artists not as the bump mapping itself, but as where the normals end up after a tangent-space-wide change of reference frame. This amounts to finding, for each (u, v) coordinate, that particular linear transformation[3] which, following the shortest path, maps the $\mathbf{n}$ tangent space vector to the $\mathbf{n}'$ value stored in the normal map texture.

This is a rotation from $\mathbf{n}$ to $\mathbf{n}'$, whose axis is perpendicular to both vectors, and as such surely lies in the tangent plane[4]. Linear transformations are usually represented, in computer graphics, by their representation matrix with respect to known vector bases. Constructing and storing such a 3x3 matrix would however incur into issues similar to those discussed above, namely memory occupation.

The hunt for a solution to this problem brings us to the title of this subsection: it is a well known fact that unit quaternions represent rotations of the 3-dimensional space. In particular, to each (unit) quaternion $\mathbf{q} = a+b\mathbf{i}+c\mathbf{j}+d\mathbf{k}$ is associated a unique rotation $r_{\mathbf{q}}$ (with the caveat that opposite quaternions represent the same rotation, i.e. $r_{\mathbf{q}} = r_{-\mathbf{q}}$), which acts on a 3D vector $\mathbf{v} = x\mathbf{i}+y\mathbf{j}+z\mathbf{k}$ in the following way:

$$\begin{aligned} r_{\mathbf{q}}(\mathbf{v}) &= \mathbf{q}^{-\mathbf{1}}\mathbf{v}\mathbf{q} \\ &= \bar{\mathbf{q}}\mathbf{v}\mathbf{q} \\ &= (a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k})(x\mathbf{i} + y\mathbf{j} + z\mathbf{k})(a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}) \end{aligned}$$

Where all of the products are quaternion products, which incidentally distribute over the sum.

Furthermore, the quaternion representing a rotation of axis $\mathbf{l} = r\mathbf{i} + s\mathbf{j} + t\mathbf{k}$ ($\mathbf{l}$ being a unit vector) and angle α can be calculated as:

$$\mathbf{q}_{\mathbf{l},\alpha} = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(r\mathbf{i} + s\mathbf{j} + t\mathbf{k}) = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

In the setting discussed above, the fact that the tangent space rotation has axis lying in the tangent plane[5] translates directly to its representing quaternion having $d = t\sin\left(\frac{\alpha}{2}\right)$ component equal to 0.

We then search for one of the two unit quaternions satisfying:

$$\begin{aligned} \mathbf{n}' &= \bar{\mathbf{q}}\mathbf{n}\mathbf{q} \\ x\mathbf{i} + y\mathbf{j} + z\mathbf{k} &= (a - b\mathbf{i} - c\mathbf{j})(0\mathbf{i} + 0\mathbf{j} + 1\mathbf{k})(a + b\mathbf{i} + c\mathbf{j}) \\ x\mathbf{i} + y\mathbf{j} + z\mathbf{k} &= (a - b\mathbf{i} - c\mathbf{j})(a\mathbf{k} + b\mathbf{j} - c\mathbf{i}) \\ x\mathbf{i} + y\mathbf{j} + z\mathbf{k} &= -2ac\mathbf{i} + 2ab\mathbf{j} + (a^2 - b^2 - c^2)\mathbf{k} \end{aligned}$$

[3] This transformation maps the $(\mathbf{t}, \mathbf{b}, \mathbf{n})$ orthonormal basis to a perturbed orthonormal basis $(\mathbf{t}', \mathbf{b}', \mathbf{n}')$, preserving orientation. By definition, this transformation is a *rotation*

[4] Which is the subspace of vectors orthogonal to $\mathbf{n}$

[5] i.e. $t = 0$

Which amounts to solving, for parameters a, b, c, the nonlinear system:

$$\begin{cases} ac & = -\frac{x}{2} \\ ab & = \frac{y}{2} \\ a^2 - b^2 - c^2 & = z \\ a^2 + b^2 + c^2 & = 1 \end{cases}$$

This gives, choosing $a \geq 0$ (for $a \leq 0$ we obtain the opposite quaternion, representing the same rotation):

$$\begin{cases} a & = \sqrt{\frac{z+1}{2}} \\ b & = \sqrt{\frac{y^2}{2(z+1)}} \\ c & = \sqrt{\frac{x^2}{2(z+1)}} \end{cases}$$

Notice that in the implementation, once the value of a has been computed, the other two parameters can be obtained as $b = \frac{y}{2a}$ and $c = -\frac{x}{2a}$, simplifying calculations.

Having no need to store a d value, it is possible to store the vector (a, b, c) in a texture, just like a normal map and most importantly with exactly the same impact on memory, but with information ready to compute the whole perturbed tangent space.

This quaternion texture can and therefore should be precomputed, which, in the implementation of this project, is what the `setupRotationMapping.exe` executable was designed for.
It takes a normal map as input, it performs the calculations described above, and it returns a texture of the same width and height of the input image, storing the quaternion map.

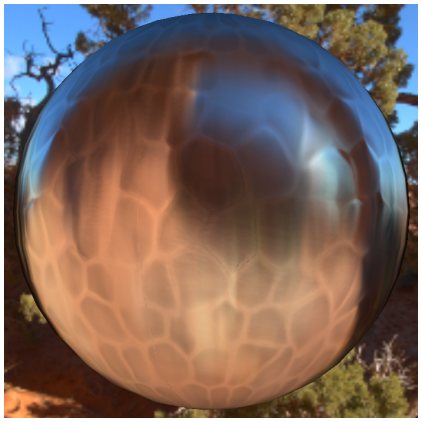
Subsequently, at the `fragment shader` stage of rendering a model, the texture is read and its values are used to compute the perturbed $\mathbf{t}'$, $\mathbf{b}'$ and $\mathbf{n}'$ vectors, by having $\mathbf{q} = a + b\mathbf{i} + c\mathbf{j}$ act on the tangent space basis vectors:

$$\begin{aligned} \mathbf{t}' = \bar{\mathbf{q}}\mathbf{t}\mathbf{q} &= (a - b\mathbf{i} - c\mathbf{j})\mathbf{i}(a + b\mathbf{i} + c\mathbf{j}) = (a^2 + b^2 - c^2, 2bc, 2ac) \\ \mathbf{b}' = \bar{\mathbf{q}}\mathbf{b}\mathbf{q} &= (a - b\mathbf{i} - c\mathbf{j})\mathbf{j}(a + b\mathbf{i} + c\mathbf{j}) = (2bc, a^2 - b^2 + c^2, 2ab) \\ \mathbf{n}' = \bar{\mathbf{q}}\mathbf{n}\mathbf{q} &= (a - b\mathbf{i} - c\mathbf{j})\mathbf{k}(a + b\mathbf{i} + c\mathbf{j}) = (-2ac, 2ab, a^2 - b^2 - c^2) \end{aligned}$$

Quaternion mapping overall achieves the goal of uniformly perturbing the tangent space, with minimal calculations to be performed in the fragment shader, while having the same impact on memory as usual normal mapping (the normal map itself is not needed anymore, as the perturbed normal can be calculated from the quaternion map, just like the other vectors).

Figure 2: Quaternion Maps

(a) only normal mapping



(b) only tangent and bitangent mapping

(c) full quaternion mapping

(d) the quaternion map obtained from the normal map in figure 1

As shown in figure 2, full tangent space perturbation is needed to properly model anisotropic reflections, all the more so near parts of the surface where the perturbation achieved by bump mapping is more pronounced (in this particular case, near the edges of the bumps).

The reddish tint of figure 3d underlines how mapped quaternions are "small" perturbations of the identity quaternion $1 + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$[6].

[6]Which is mapped to the RGB triplet $\left(1, \frac{1}{2}, \frac{1}{2}\right)$

3.5 Tangent Plane Rotation (TPR) Mapping

In subsection 3.4, a choice was made of the particular rotation mapping $\mathbf{n}$ to $\mathbf{n}'$, selecting the "shortest" rotation, i.e. the one with axis perpendicular to both $\mathbf{n}$ and $\mathbf{n}'$, among all of the possible rotations achieving the same result.

In fact, this class of rotations can be characterized as the collection of all rotations of the kind $r = \bar{r} \circ r^{\top}$, where $\bar{r}$ is the rotation mentioned above, as selected in subsection 3.4, and $r^{\top}$ is *any* rotation of the tangent *plane*, that is to say any one having $\mathbf{n}$ as rotation axis.

This behaviour underlines the complexity of the space of 3D rotations, compared to their 2D counterpart, and allows for effects only achievable by anisotropic shaders, given their reliance on the state of the whole tangent space. Tangent and bitangent vectors can in fact be rotated in the tangent plane to describe non-trivial variations of the microsurface distribution orientation across the surface.

This subsection focuses on how to describe an arbitrary tangent plane rotation $r^{\top}$, how to implement it in the shader and how to compose it with other transformations like tangent space perturbations discussed before.

To rotate the whole tangent plane, both the $\mathbf{t}$ and $\mathbf{b}$ vectors should be rotated the same way. Since they should always be perpendicular, a quick simplification is to fix, for any arbitrary tangent vector $\mathbf{t}$, its corresponding[7] bitangent vector $\mathbf{b}$ as $\mathbf{b} = J\mathbf{t}$, where:

$$J = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This convention allows us to only have to rotate $\mathbf{t}$ in the tangent plane and retrieve $\mathbf{b}$ immediately after.

Two paths can be followed to rotate $\mathbf{t}$: the first would be to store an angle ϑ in a 1-channel texture, representing the angle of rotation about the $\mathbf{n}$ axis. This however would require calculating goniometric functions and matrix multiplication in the fragment shader.

The second path is the one I implemented in this project: I store the two components[8] of the rotated tangent vector (in tangent space coordinates), in a 2-channel texture. They are then passed to the `fragment shader` as they are to compute the modified $\mathbf{t}$ vector.

[7] $\mathbf{b}$ is effectively obtained from $\mathbf{t}$ by rotating it 90° around the normal $\mathbf{n}$

[8] The two nontrivial components: a tangent vector always has its third component equal to 0

The generation of this texture, as implemented for this project, is performed in the `generateTangentPlaneTexture.exe` program. It revolves around two lines of code computing the vector's x and y components as functions of the texture's (u, v) coordinates.
Here is an example of the functions generating the TPR mapping used in figure 3

```
float x = glm::cos(PI*v); // x(u,v)
float y = glm::sin(PI*v); // y(u,v)
```

This program actually generates a 4-channel RGBA texture, since it supports, on its third and fourth channels, the definition of a UV map for the directional roughnesses α_x and α_y used by anisotropic shaders. However, as explained in section 4, the preprocessing required for achieving smooth real-time bumped environment mapping needs the roughness values to be fixed across the whole application. These components are thus ignored by the main application, but their values can be set across the texture by modifying these lines of code (their current state reflects the RGBA TPR map in figure 3):

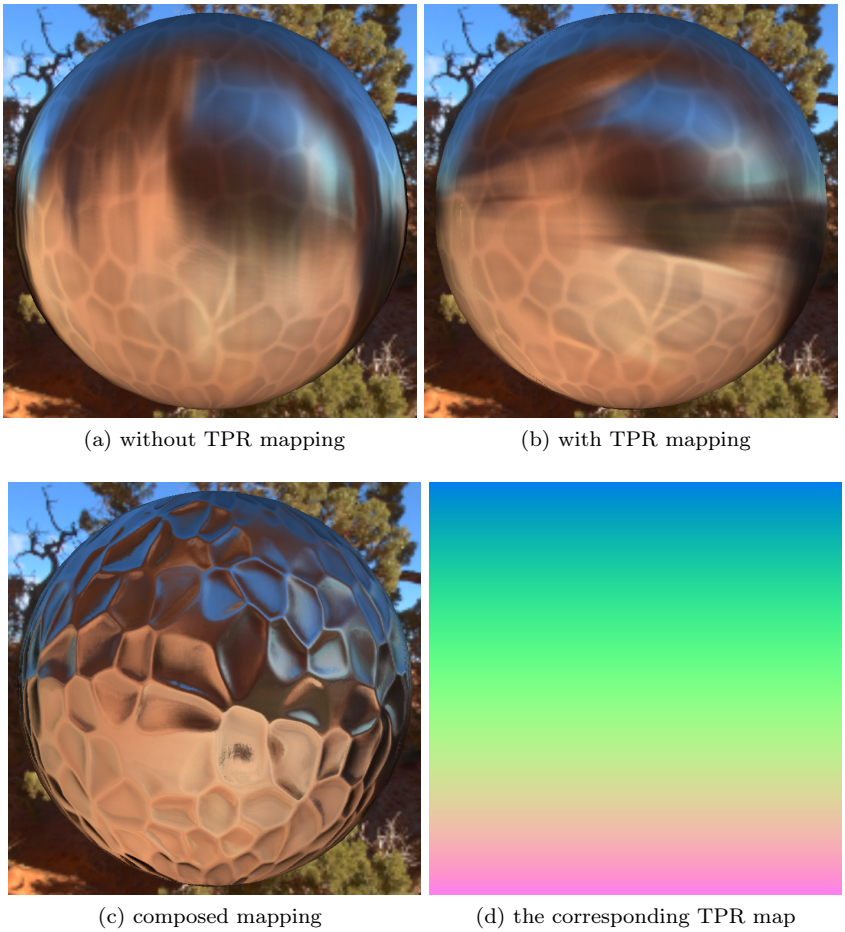
```
float m = 0.1; // principal roughness, i.e. alpha_x
float n = 1.0; // orthogonal roughness, i.e. alpha_y
```

In direct-light anisotropic renderings, the full TPR/roughness mapping can be tailor-made to achieve a wider range of effects, albeit sacrificing the beauty of environment mapping

A different use for the same tool would be to help artists authoring complex UV-mapped models with anisotropic reflecting surfaces in the UV-mapping itself. It is often the case, in fact, that UV maps must be split along seams in the mesh, mapping parts of the mesh to different areas in the texture image, cramming and rotating them to make use of the most possible space. To maintain correct anisotropy, the orientation of these mapped regions would be constrained to a fixed direction, introducing issues when cramming them. Through the use of a parallel TPR texture, instead, the artist is allowed to rotate each mapped mesh component as needed, letting TPR manage the issue of anisotropy directions on the surface of the model.

The results of this process are shown in figure 3, as well as the result of composing quaternion and TPR mapping on the same model. Composing the effects is actually really simple, since rotations (in this case in three dimensions) are linear transformations. This lets us apply bump mapping straight to the rotated $\mathbf{t}$ and $\mathbf{b}$ ($\mathbf{n}$ is unaffected by TPR mapping) and achieve their composite

Figure 3: Tangent Space Rotation Mapping



(a) without TPR mapping

(b) with TPR mapping

(c) composed mapping

(d) the corresponding TPR map

effect.

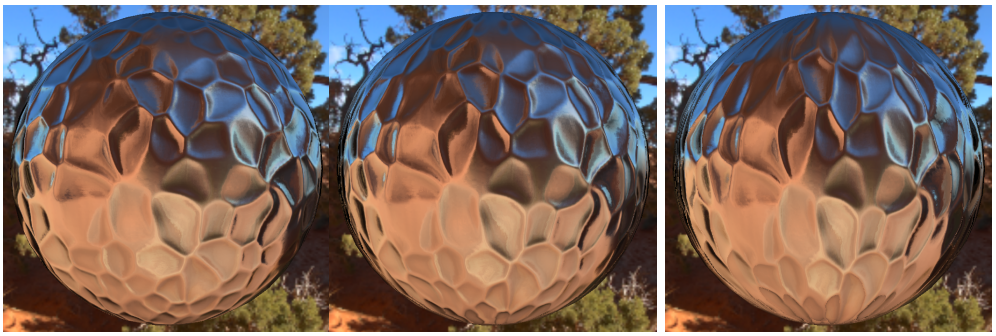
3.6 Parallax Occlusion Mapping (POM)

POM is a displacement technique that modifies the (u,v) coordinates used for texture sampling based on the view direction **v** and the height map of the material. It is a method for simulating visibility occlusion due to parallax effects without actually displacing elements of the mesh.

Its effects are scalable by a `uniform float heightScale` factor ranging between 0 and 1 passed to the `fragment shader` by the application. It works best on isometric UV-mappings, due to the fact that the resulting displacement step is calculated in texture space and it is uniform in all directions within this

Figure 4: Parallax Occlusion Mapping

(a) heightScale = 0 (b) heightScale = 0.1 (c) heightScale = 0.3



space (meaning it is not uniform in all directions on the surface in the case of a non-isometric UV-mapping)[9]

I implemented this method based on [AsSh00]'s implementation with minimal modifications, since its inclusion is marginal to the goal of the project. As a consequence, I refer to LearnOpenGL - Parallax Mapping for details on its implementation.

Unlike the previous methods, which needed a thorough explanation of the underlying mathematical models and choices in implementation, lacking an external reference, I have opted for a quicker overview of this particular method.

Figure 4 shows POM effects for increasing values of `heightScale`.

4 Environment Mapping

4.1 Overview

Environment mapping is a lighting technique whose purpose is to account for incoming radiance from the whole environment surrounding the model to be rendered, instead of just summing the contributions of a discrete collection of point and directional lights.

It is best suited for a PBR setting and it achieves a physical realism hardly obtainable by discrete light sources. Since the integration of the lighting equation (for arbitrary environment radiance) is a heavy task for real-time rendering, this method has to introduce a set of approximations that let us perform as much numerical integration as possible before the application doing the rendering even runs. Its results are stored in appropriate texture formats read by the application at the time of rendering.

[9]There is a geometric way to compensate this effect, but it would involve computing the riemannian metric of the immersed surface (the first fundamental form), which could easily be done for any UV-mapping. This is however outside the scope of this project and it wasn't implemented.

The main ingredient in this kind of lighting is the data describing incoming radiance. The most widely available arrangement for this data is the 3-channel `hdr` image format[10], whose RGB values are stored as floats to span a High Dynamic Range. Artists usually author these environment maps in spherical coordinates, resulting in a rectangular picture (called an *equirectangular* format). To ensure faster computation[11], it is usually preferred to remap this picture to six `hdr` files representing a cubemap, more easily accessible by the rendering process (in the `fragment shader`).

4.1.1 The Epic Games method for isotropic specular EM

[LearnOpenGL] does a great job of describing how to implement environment mapping for the Cook-Torrance GGX (isotropic) illumination model. Its [LearnOpenGL: Specular IBL] chapter is based on a paper by Epic Games [Karis13] describing the implementation of GGX in Unreal Engine 4[12]. This paper is also the source of the approximation choices having to be made to achieve fully pre-processed numerical integrations, whose results are used by IBL[13]. The two main simplifications operated by this paper are:

1. the split sum approximation, which splits the specular lighting equation integral into the product of the integrals of the radiance and that of the BRDF

2. the Fresnel-Schlick approximation of the Fresnel reflectance, which allows for splitting the BRDF integral into two components

The approximation mentioned in point 2 allows for the integration of terms that do not depend on the value of F_0, the Fresnel reflectance at normal incidence, whose value can be set and passed to the `fragment shader` by the application. The fragment shader then uses this value to compute the actual result of the full integral.

The actual numerical integration is eventually performed in a Monte-Carlo setting with importance sampling of the half-vectors **h**. The radiance integral results are stored in a "prefiltered irradiance" cubemap texture, while the two BRDF integral components are stored in a 2-channel texture whose (u,v) coordinates are the value of the (**nv**) product and the value of the (isotropic) roughness α.

[10] Hence the use of the term Image Based Lighting (IBL) for this kind of environment mapping

[11] Avoiding having to evaluate goniometric functions during rendering

[12] The Epic Games paper is itself a real-time adaption of a Disney paper, but I think it's best to limit excessive reference meandering

[13] Albeit these allow for full preprocessing of only isotropic models

4.1.2 Adapting the Epic Games method to anisotropic models

I will refer to [LearnOpenGL: Specular IBL] for the implementation of a preprocessed-GGX-environment-mapped lighting model and its explanation, whereas here I want to detail the design choices involved in adapting the method mentioned above to anisotropic illumination models, while keeping full support for bump mapping and the unpredictability of the tangent space's orientation it entails.

1. *Roughness values α_x and α_y are fixed.* This is due to the fact that the BRDF lookup texture (LUT), for reasons described in subsection 4.3.3, as well as the half-vector LUT used for importance sampling and detailed in subsection 4.3.2, would both need to be 4D textures to accomodate roughness dependance. This is an absurd requirement in terms of storage space occupancy that cannot be satisfied in any way at high enough texture resolution.

2. *Importance sampling is also preprocessed.* This means that, for fixed roughness values, a LUT mapping the unit square to the model's half-vectors[14] is generated by a dedicated program. This texture is then sampled both during the real-time radiance integration detailed below and within the program performing the BRDF integration, which returns the BRDF LUT. This preprocessing step helps to avoid having to evaluate processing-power-heavy functions during rendering, while in the same way it helps speeding up the generation of the BRDF LUT.

3. *Radiance integration must be performed real-time.* Due to the dependence of anisotropic illumination models on the orientation of the tangent space and not just the normal vector, it is impossible to generate a "prefiltered irradiance cubemap" like Epic Games does. This is true in general for any non-convex mesh[15] and even more true as soon as bump-mapping comes into play. This step is therefore the most performance-affecting of the environment mapping steps, and its impact can only be controlled by dynamically adjusting the number of samples used to integrate the environment radiance numerically. Speaking of which, it should be noted that lower levels of roughness always require fewer samples to achieve comparable results[16].

4. *The BRDF LUT is indexed by* $(\mathbf{nv})$ *and* ϕ[17], whereas Epic Games's one was indexed by $(\mathbf{nv})$ and α, the isotropic roughness. Anisotropic illumination models need information regarding the rotational configuration of

[14] In cartesian *tangent space* coordinates, according to their distribution, which is determined by the specific illumination model and the roughness values

[15] Non-convex meshes are very likely to have a pair of points whose light vectors point to the same cubemap texel, but whose tangents have different directions, resulting in different anisotropic lighting

[16] This is due to the fact that lower values of roughness mean that the distribution of the half-vectors will tend to concentrate sample within a smaller region

[17] The azimuthal angle of $\mathbf{v}$, i.e. the angle between $\mathbf{t}$ and the projection of $\mathbf{v}$ onto the tangent plane, following the notation in subsection 1.2

v with respect to **n**, as well as its inclination, unlike isotropic ones which only need the latter. These two parameters allow us to completely reconstruct the view vector in tangent space coordinates, where the half-vector sampling and the BRDF integration are performed.

4.2 Diffuse Component

The diffuse lighting component is not affected by anisotropy, which means that the radiance integration can be performed following [LearnOpenGL]'s method, detailed at [LearnOpenGL: Diffuse Irradiance], step by step.

I use a lambertian diffuse model to keep things simple, since diffusion is an important part of phisically based rendering, but it isn't the focus of this project

For ease of use, remapping of equirectangular `hdr` files (to generate the environment cubemap) and generation of the diffuse irradiance cubemap are both done by the same program: `cubeMapping_fromEquirectangular.exe`. It takes the name of the environment map directory (within the textures folder) as input, which can be set at this line of code in the `cpp` file:

```
std::string folderName = "arches/";
```

The EM directory should contain an `hdr` file named `equirectangular.hdr` and, at least on computers running on Windows, folders `environment` and `irradiance`. Running the program renders a cube with shaders `cubemap.vert` and `equi_to_cube.frag` and saves the cube's faces in the `environment` folder. It then convolutes the environment map to obtain its diffuse irradiance map while rendering the same cube with shaders `cubemap.vert` and `convolution.frag` and saves its faces in the `irradiance` folder.

4.3 Specular Component

4.3.1 Why Ashikhmin-Shirley?

Section 2 presents three different approaches to anisotropic illumination. Only one of those models was implemented in this project[18]: Ashikhmin-Shirley.

The reason behind this choice is mainly due to the fact that environment mapping looks the best in a physically based setting, which immediately eliminates the Heidrich-Seidel model from the picture.

As for Ward's model, although it finds its roots in physical reality, it purposefully lacks both Fresnel reflectance and geometry visibility terms, which have now become staples of PBR. More importantly, while it does provide a method

[18]Previous versions were point-light-based instead of environment-mapped, showing the results of rendering with all three models, as well as with various isotropic ones

for importance sampling, it doesn't explicitly provide the distribution of the sampled half-vectors, making Monte-Carlo integration converge more slowly, since it can't be weighted against the probability density.

As explained in subsection 2.4, the Ashikhmin-Shirley provides the perfect tools and setting for implementing efficient environment mapping.

Important note: I have until now referred to α_x and α_y as (directional) roughness values, ranging between 0 and 1. Ashikhmin-Shirley's analogous parameters are instead terms n_u and n_v, each with values greater or equal to 1, which can be considered respectively as $\frac{1}{\alpha_x^2}$ and $\frac{1}{\alpha_y^2}$.

4.3.2 Importance Sampling

As explained in subsection 4.1.2, preprocessing the sampling of half-vectors plays an important part in reducing the load of work left to real-time rendering. Luckily, Ashikhmin-Shirley provides[19] a way to map two uniformly distributed (ξ_1, ξ_2) sample values on the interval $[0, 1]$ to the spherical coordinates of half-vectors $\mathbf{h}$ whose distribution (in the tangent space) is p_h[20]. That is, it provides functions $(\phi, \theta) : [0,1] \times [0,1] \to [0, 2\pi) \times \left[0, \frac{\pi}{2}\right]$, for fixed n_u, n_v, mapping the unit square to the upper hemisphere (in spherical coordinates) of the unit sphere:

$$\phi_{red}(t) = \arctan\left(\sqrt{\frac{n_u+1}{n_v+1}}\tan\left(\frac{\pi}{2}t\right)\right)$$

$$\phi(\xi_1,\xi_2) = \begin{cases} \phi_{red}(4\xi_1) & \xi_1 \in \left[0, \frac{1}{4}\right) \\ \pi - \phi_{red}(2-4\xi_1) & \xi_1 \in \left[\frac{1}{4}, \frac{1}{2}\right) \\ \pi + \phi_{red}(4\xi_1-2) & \xi_1 \in \left[\frac{2}{4}, \frac{3}{4}\right) \\ 2\pi - \phi_{red}(4-4\xi_1) & \xi_1 \in \left[\frac{3}{4}, 1\right] \end{cases}$$

$$\theta(\xi_1,\xi_2) = \arccos\left((1-\xi_2)^{\frac{1}{n_u\cos^2\phi + n_v\sin^2\phi + 1}}\right)$$

Using (ξ_1, ξ_2) as (u,v) coordinates, it is then possible to map the half-vector (ϕ, θ) to cartesian coordinates ($\mathbf{h} = (x, y, z)$) and store its components in the first three channels of an RGBA texture, whose alpha channel can also be used to store the value of $p_h(\mathbf{h})$ (after renormalization), to avoid having to compute it again at run-time.

[19] Having fixed n_u and n_v

[20] See 2.4

`halfVectorSampling.exe` does all of this. When launched, it waits for command-line input specifying the desired n_u and n_v values, after which it performs the calculations described above and saves a file called `halfVectorSampling`$[n_u, n_v]$`.png`[a] representing said texture in the `textures` folder.

[a]It substitutes the actual n_u, n_v values passed as input

Wherever **h**-sampling is needed, it is now as easy as generating a low discrepancy sequence[21] of N points on the unit square and using their coordinates to sample the half-vector texture

The low discrepancy sequences used for integration across this project are all Hammersley sequences. More information can be found here: [Holger Dammertz: Hammersley Sequence]. One thing to specify is that the first sample in a Hammersley sequence is always the point (0,0), which in Ashikhmin-Shirley importance sampling is always mapped to the vector (0,0,1) in tangent space: the **n** vector. This means in particular that setting the number of samples to 1 in the `fragment shader` of the rendering application always replicates the effects of a perfectly reflecting surface (isotropic refection with null roughness).

Figure 5: Half-vector sampling visualization

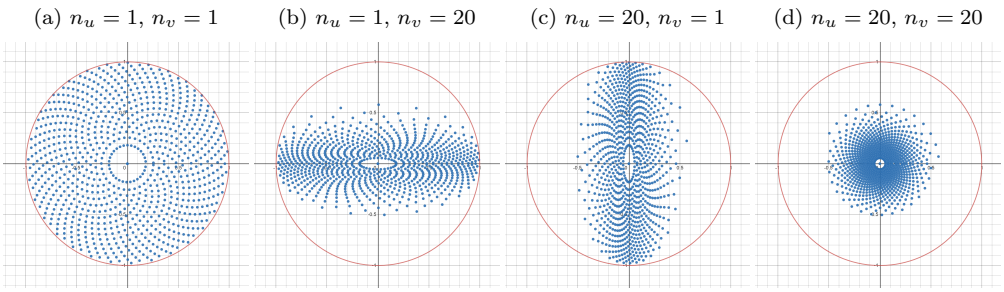


Figure 5 shows how a semi-regular sequence on the unit square (not Hammersley) is mapped in tangent space. The view is from the top, with **n** pointing out of the screen, **t** to the right and **b** upwards.

[21]Low discrepancy sequences tend to have faster convergence of Monte-Carlo integration, compared to uniformly distributed sequence, which is why they were preferred in this project. The half-vector lookup texture however works with any kind of sample sequence on the unit square.

Figure 6: Corresponding lookup textures

(a) $n_u = 1$, $n_v = 1$ (b) $n_u = 1$, $n_v = 20$ (c) $n_u = 20$, $n_v = 1$ (d) $n_u = 20$, $n_v = 20$

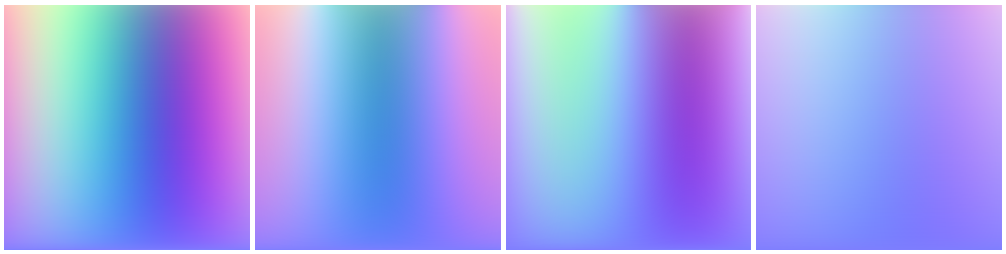


Figure 6 shows instead the corresponding half-vector LUT for the same values of n_u and n_v.

4.3.3 BRDF integration

Recall from subsection 2.4 that we had derived the integration-ready form of the Ashikhmin-Shirley bidirectional reflectance distribution function (with the Fresnel-Schlick approximation) as follows:

$$f_s(\mathbf{v}, \mathbf{l}) = p_l(\mathbf{l}) \left(F_0 \frac{1 - (1 - (\mathbf{vh}))^5}{\max((\mathbf{nv}), (\mathbf{nl}))} + \frac{(1 - (\mathbf{vh}))^5}{\max((\mathbf{nv}), (\mathbf{nl}))} \right)$$

Where $p_l(\mathbf{l})$ is the probability density of the light vector $\mathbf{l}$, determined by view vector $\mathbf{v}$ and half-vector $\mathbf{h}$.

Recall also that the specular component of the lighting equation, for an arbitrary BRDF f_s, is:

$$L_o(\mathbf{v}) = \int_{\mathbf{l}} f_s(\mathbf{v}, \mathbf{l}) L_i(\mathbf{l}) (\mathbf{nl}) d\omega_{\mathbf{l}}$$

Where L_o and L_i are respectively the observed and incident radiance, and the integral covers the upper hemisphere of the unit sphere in tangent space (where $(\mathbf{nl}) > 0$).

The split sum approximation mentioned at the beginning of section 4 consists in evaluating this integral as the product of two simpler ones:

$$L_o(\mathbf{v}) = \int_{\mathbf{l}} f_s(\mathbf{v}, \mathbf{l}) (\mathbf{nl}) d\omega_{\mathbf{l}} \int_{\mathbf{l}} L_i(\mathbf{l}) d\omega_{\mathbf{l}} = F(\mathbf{v}) L(\mathbf{v})$$

The first of these two integrals is the one this subsection focuses on, while the second will be the main topic of the next subsection.

Substituting the current form of the Ashikhmin-Shirley BRDF and setting $\chi(\mathbf{v}, \mathbf{h}) = (1 - (\mathbf{vh}))^5$we obtain:

$$F(\mathbf{v}) = F_0 s(\mathbf{v}) + b(\mathbf{v})$$

where

$$s(\mathbf{v}) = \int_{\mathbf{l}} p_l(\mathbf{l}) \left(1 - \chi(\mathbf{v}, \mathbf{h})\right) \frac{(\mathbf{nl})}{\max((\mathbf{nv}), (\mathbf{nl}))} d\omega_{\mathbf{l}}$$
$$b(\mathbf{v}) = \int_{\mathbf{l}} p_l(\mathbf{l}) \chi(\mathbf{v}, \mathbf{h}) \frac{(\mathbf{nl})}{\max((\mathbf{nv}), (\mathbf{nl}))} d\omega_{\mathbf{l}}$$

Which, performing Monte-Carlo integration with importance sampling and sample half-vectors $\{\mathbf{h}_i\}_{i=1}^N$, becomes:

$$s(\mathbf{v}) \approx \sum_{i=1}^{N} \frac{(\mathbf{nl}_i) \left(1 - \chi(\mathbf{v}, \mathbf{h}_i)\right)}{\max((\mathbf{nv}), (\mathbf{nl}_i))}$$
$$b(\mathbf{v}) \approx \sum_{i=1}^{N} \frac{(\mathbf{nl}_i) \chi(\mathbf{v}, \mathbf{h}_i)}{\max((\mathbf{nv}), (\mathbf{nl}_i))}$$

having set $\mathbf{l}_i = 2(\mathbf{vh}_i)\mathbf{h}_i - \mathbf{v}$ for $i = 1, ..., N$

The functions s and b are called, respectively, "size" and "bias", deriving their name from the role they play in calculating the BRDF integral[22].

Their values only depend on the components of $\mathbf{v}$ in tangent space, which allows them to be stored in a lookup texture, as soon as a convenient mapping of the view vectors $\mathbf{v}$ to the unit square is chosen.

The values of $(\mathbf{nv})$ and the azimuthal angle ϕ[23] provide all the necessary information to fully determine $\mathbf{v}$ in tangent space coordinates.

A convenient aspect of the Ashikhmin-Shirley BRDF is that it has rectangular symmetry. That is to say, its value doesn't change if $\mathbf{t}$ or $\mathbf{b}$ are flipped. This makes it possible to have ϕ range between 0 and $\frac{\pi}{2}$ (instead of 2π) and still retain all necessary information.

Determining the value of this ϕ, corresponding to a certain $\mathbf{v}$, is done this way:

$$\phi(\mathbf{v}) = \arctan \left| \frac{(\mathbf{bv})}{(\mathbf{tv})} \right|$$

[22] The analogy here is with the line equation $f(x) = sx + b$, having F_0 play the role of the variable

[23] See 17

Mapping ($\mathbf{nv}$) straight as the v coordinate of the texture, though, tends to mean having worse resolution in areas of the texture where the variations of s and b are most pronounced. To compensate for this effect, I have chosen to set $\sqrt{(\mathbf{nv})}$ as the v coordinate of the texture.

The full mapping becomes:

$$u(\mathbf{v}) = \frac{2}{\pi} \arctan \left| \frac{(\mathbf{bv})}{(\mathbf{tv})} \right|$$
$$v(\mathbf{v}) = \sqrt{\max\left((\mathbf{nv}), 0\right)}$$

Where the max is taken to ensure $\mathbf{v}$ is in the upper hemisphere, which may not always be verified due to precision issues.

The inverse of this mapping[24] (used in the program generating the texture to reconstruct the view vector corresponding to coordinates (u, v)) is:

$$(\mathbf{tv}) = \cos\left(\frac{\pi}{2}u\right)\sqrt{1-v^4}$$
$$(\mathbf{bv}) = \sin\left(\frac{\pi}{2}u\right)\sqrt{1-v^4}$$
$$(\mathbf{nv}) = v^2$$

The program `brdfIntegration.exe`, when launched, waits for command-line input specifying the desired n_u and n_v shininess values to generate the LUT from. It then looks for the half-vector LUT with corresponding shininess values in the `textures` folder. If it doesn't find it, the program exits and the user should first launch `halfVectorSampling.exe` inserting these values. Otherwise, the program generates a Hammersley sequence to sample half-vectors using the half-vector LUT. It then uses them to perform Monte-Carlo integration of the BRDF as detailed above, obtaining the values of s and b for each (u, v) coordinate.

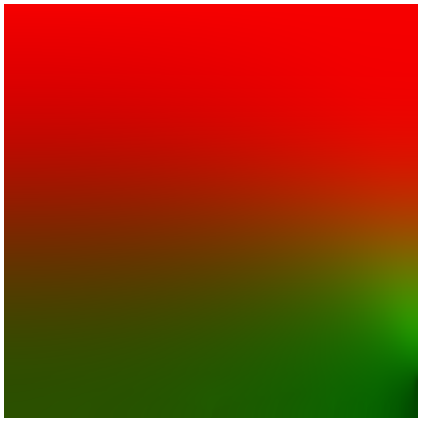
It eventually saves an RGB texture called `brdfIntegration`$[n_u, n_v]$`.png` in the `textures` folder, storing s in the red channel, b in the green channel and 0 in the blue one. The reason for using 3 channels instead of just the necessary 2 is just that 2-channel-texture images are saved as grey-alpha images, making it harder to figure out by sight how the values vary on the picture. `png` compression however ensures that setting the whole blue channel to 0 doesn't occupy unnecessary space.

[24] Notice that $(\mathbf{nv}) = \cos\theta$,where θ is the *polar angle* of $\mathbf{v}$, i.e. the angle between $\mathbf{n}$ and $\mathbf{v}$. This means $\sin\theta = \sqrt{1-(\mathbf{nv})^2}$

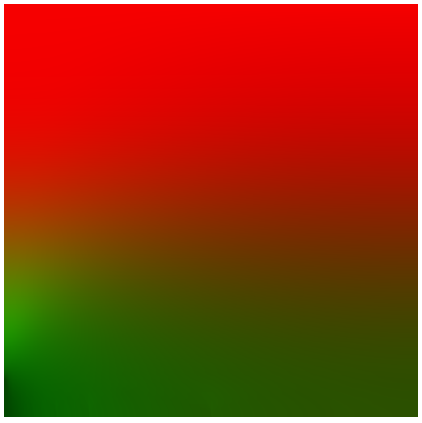
Figure 7: Integrated BRDF lookup textures

(a) $n_u = 70$, $n_v = 70$

(b) $n_u = 70$, $n_v = 20000$



(c) $n_u = 20000$, $n_v = 70$



(d) $n_u = 20000$, $n_v = 20000$

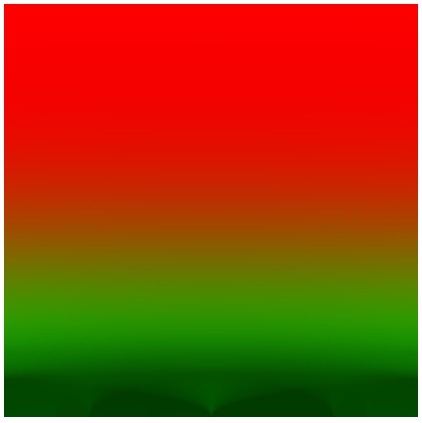


Figure 7 shows the resulting BRDF LUT for various shininess values. Notice how lower values (corresponding to rougher surfaces) mean darker textures, and thus less intense reflections. This is due to more sample $\mathbf{l}$ vectors going below the horizon . This is the main source of energy loss in the Ashikhmin-Shirley model.

4.3.4 Real-Time Monte-Carlo integration of environment luminance

This is the last component to be addressed for implementing specular environment mapping. It centers around of the radiance integral obtained from the split sum at the beginning of the last subsection:

$$L(\mathbf{v}) = \int_{\mathbf{l}} L_i(\mathbf{l}) d\omega_{\mathbf{l}}$$

Where $L_i(\mathbf{l})$, the incident radiance, is the data contained in the environment cubemap.

As explained at the beginning of section 4, this integration cannot be preprocessed when dealing with anisotropic illumination models.

The integration is then performed in the same way as explained in the last subsection, regarding the BRDF integral, but is now done in the `fragment shader` (`env_bump_aniso.frag`) with the help of the preprocessed half-vector sampling.

The fragment shader generates a Hammersley sequence, samples the half-vectors (whose coordinates are as of now in un-bumped tangent space), bumps the half vectors using the bumped $\mathbf{t}$, $\mathbf{b}$, $\mathbf{n}$ and determines the corresponding $\mathbf{l}$ vectors.

These light vectors are still in tangent space, which is why they are first sent to world (model) space using the TBN matrix (mapping tangent space to world space) `in mat3 wTBNt`, which is provided by the `vertex shader` (`env_bump_aniso.vert`).

Their world coordinates are eventually used to sample the environment map, obtaining the incident light value for the corresponding direction.

These values are summed and their normalized sum plays the role of the "prefiltered specular irradiance" described in the Epic Games paper [Karis13].

All of the ingredients are now ready to obtain the final result.

5 Implementation Details

5.1 Languages and Libraries

This project is written in C++, both the main application and the preprocessing applications. Vertex and fragment shaders are all in GLSL (OpenGL Shading Language).

It includes the following libraries:

- < `glfw/glfw3.h` > which manages the application window and context
- < `glad/glad.h` > which loads the OpenGL functionalities
- < `utils/shader_v1.h` > which manages shader program compiling and linking
- < `utils/model_v2.h` > which manages mesh import in OpenGL from `obj` files
- < `utils/camera.h` > which manages camera movement
- < `stb_image/stb_image.h` > which manages reading image files
- < `glm/glm.hpp` >, < `glm/gtc/matrix_transform.hpp` >, < `glm/gtc/matrix_inverse.hpp` > for vector calculations
- < `glm/gtc/type_ptr.hpp` > to pass glm vectors to shaders as uniforms
- < `imgui/imgui.h` > to handle the GUI within the running application

Preprocessing programs also include < `stb_image/stb_image_write.h` > to save texture files as images.

5.2 File System organization for Asset import

Preprocessing programs and the main application require assets to be stored in the right directories and with the correct filenames to work properly.

```
project directory/
   include/
   libs/
   models/
   src/
   textures/
   Workspace_Organization.code-workspace
```

Models in `obj` format should all be in the `models` folder. The `src` folder contains the source code and eventually the executables of the main application and of the preprocessing programs.

The `textures` folder should be organized like this:

```
textures/
   material_1/
   ...
   material_n/
   environment_map_1/
   ...
   environment_map_m/
   //the following are generated here by the corresponding programs
   halfVectorSampling [n_u_1,n_v_1].png
   ...
   halfVectorSampling [n_u_i,n_v_i].png
   brdfIntegration [n_u_1,n_v_1].png
   ...
   brdfIntegration [n_u_i,n_v_i].png
```

`material` folders like this, providing all of the files shown below (the image formats may vary):

```
material/
   albedo.jpg
   ao.png
   depth.png
   metallic.png
   normal.jpg
   quaternion.png //as generated by setupRotationMapping.exe
   rotation.png //as generated bt generateTangentPlaneRotation.exe
```

And finally `environment_map` folders like this:

```
environment_map/
  environment/
    //this folder is empty before running
        cubeMap_fromEquirectangular.exe
    back.hdr
    down.hdr
    front.hdr
    left.hdr
    right.hdr
    up.hdr
  irradiance/
    //this folder is empty before running
        cubeMap_fromEquirectangular.exe
    back.hdr
    down.hdr
    front.hdr
    left.hdr
    right.hdr
    up.hdr
  equirectangular.hdr
```

5.3 Compiling and using the pre-processing applications

Within the `src` folder in the main directory there are three folders related to preprocessing: `Quaternion Mapping`, `Tangent Plane Rotation Mapping` and `Environment Preprocessing`. Each of these folders contains a file `MakefileWin.bat` with all the instructions for compiling the `cpp` files in that folder. The Build command should be launched from Visual Studio Code after having imported the main directory as a workspace, to ensure visibility of the `include` directory. Include flags are all specified in the Visual Studio settings in each folder. The executables are built in the same folder as their source code and they work from that position in the file system.

Within the `Environment Preprocessing` folder, `brdfIntegration.exe` and `halfVectorSampling.exe` work as they are.

`cubeMapping_fromEquirectangular.exe` works instead on the environment map directory specified within the source code:

```
std::string folderName = "arches/";
```

To change environment map, this string should be modified before compiling.

The same applies to `setupRotationMapping.exe` inside the `Quaternion Mapping` folder. In this case, the target file (the normal map image) can be modified at

the line:

```
unsigned char *img = stbi_load("../../textures/metal_tiles/normal.jpg",
    &width, &height, &channels, STBI_rgb);
```

Having ensured a correct file system organization and correct target paths, the programs should work as intended.

5.4 Compiling and running the rendering

The source code for the main program of this project can be found in `/src/Main Application/aniso.cpp`. Building the source code via Visual Studio Code (the instructions are contained in the `MakefileWin.bat` file) generates the executable `Aniso.exe`. When launched, it compiles and links the shaders `env_bump_aniso.vert` and `env_bump_aniso.frag`, used for rendering environment-and-bump mapped models, and the shaders `skybox.vert` and `skybox.frag`, used for rendering the background.

The fixed application parameters are set at the top of the source code and should be modified as desired before compiling `aniso.cpp`.

They are the directional shininess components of Ashikhmin-Shirley, n_u and n_v, the relative path to the material directory and the relative path to the environment map directory.

They can be respectively set at the following lines of code:

```
GLfloat nU = 20000.0f, nV = 70.0f;
```

```
std::string materialFolder = "hammered_metal/";
```

```
std::string cubeMapsFolder = "arches/";
```

If the shininess values do not correspond to values already used for generating the preprocessed half-vector and BRDF integration lookup textures, the program can't find the relevant files to import the textures and it exits.

Similarly, if the names of the material or environment map directories are not properly inserted, or if the folders are not organized as detailed in subsection 5.2, the textures can't be imported and the program exits.

Once the program starts, the scene is rendered and a GUI box is overlayed in the top-left corner. It contains instructions describing how to move within the scene.

Pressing the E key stops mouse and keyboard input and changes the GUI to two windows, of which the first allows for dynamically swapping subroutines affecting the rendering, such as enabling or disabling normal mapping, quaternion mapping etc.

The second window contains sliders to dynamically alter the value of the (RGB) Fresnel reflectance at normal incidence F_0, the height-scale for displacement mapping and the number of samples used in environment mapping with the Ashikhmin-Shirley model.

Pressing E again returns the application to the previous state, letting the user move freely within the scene once again.

6 Results and Performance

6.1 Rendered Models

Figure 8: PBR bunnies

(a) dielectric bunny

(b) metallic bunny

Figure 9: PBR spheres

(a) dielectric sphere

(b) metallic sphere

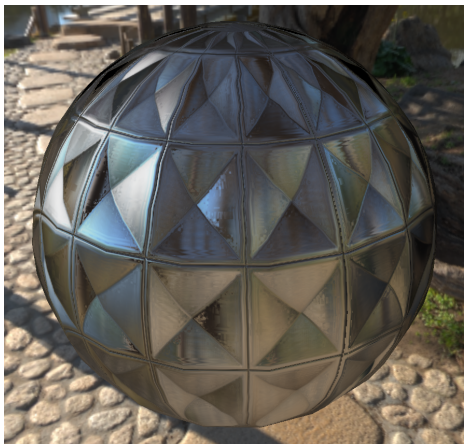
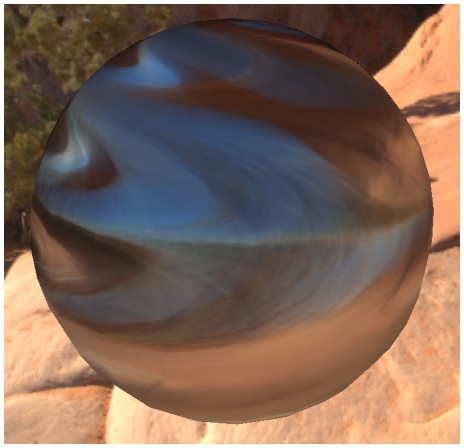


Figure 10: Playing with Tangent Plane Rotation mapping

(a) spiral tangents

(b) wild tangents



6.2 Performance and Quality analysis

The final rendering works as intended, having anisotropic illumination models work with bump mapping, tangent plane rotation and environment mapping. It achieves a good level of realism, as shown in the pictures above and runs pretty smoothly in a real-time application.

The main factor affecting performance is definitely the residual part of numerical integration having to be performed in the fragment shader. This implies

that the rendering can run pretty smoothly on models contributing to a limited fraction of the total fragments (i.e. little screen space), even with a high number of samples (in the order of the hundreds). Conversely, due to how the code is implemented (the number of samples is passed as a uniform to the fragment shader), for models occupying more screen space the number of samples can be dynamically adjusted to avoid an excessive load on the GPU.

On a computer running on a NVidia Geforce 770 GTX with 2GB of RAM on a fullscreen 1080p display, the frame rate remains capped at 60 FPS while rendering (with 200 samples) a model in the middle of the screen covering roughly an eighth of the total screen space. In the extreme case of an object covering the whole screen, the frame rate is roughly inversely proportional to the number of samples, going from $\sim$ 60 FPS at 60 sample-count to $\sim$ 20 FPS at 200 sample-count.

In most cases however the screen occupancy of an object to which one would want to apply anisotropic shading is much more limited. At the same time, in most use-cases the number of samples can be capped well below 100 and still achieve perfectly acceptable results. This is all the more true for shinier surfaces: sampled vectors tend to be closer together and a smaller number of them is needed to obtain similar results to a much higher sample count.

References

[LearnOpenGL] learnopengl.com by Joey de Vries

[Karis13] Brian Karis, Real Shading in Unreal Engine 4, Epic Games, 2013

[AsSh00] Michael Ashikhmin and Peter Shirley. An Anisotropic Phong Light Reflection Model. University of Utah, 2000

[HeSe98] Wolfgang Heidrich and Hans-Peter Seidel. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. University of Erlangen, 1998

[Ward92] Gregory J. Ward. Measuring and modeling anisotropic reflection. Computer Graphics, 26(4):265–272, July 1992. ACM Siggraph '92 Conference Proceedings.