Production Systems and Information Engineering
Volume 12 (3), pp. 42-59
doi: https://doi.org/10.32968/psaie.2024.3.5

PER-PIXEL LIGHTING FOR MULTI-OBJECT MODELS

PÉTER MILEFF
University of Miskolc, Hungary
Institute of Information Technology
peter.mileff@uni-miskolc.hu

Abstract. In computer graphics, per-pixel lighting is a widely used technique for achieving realistic lighting effects by computing light interactions at the pixel level. This approach allows for detailed representation of surface characteristics, such as textures and fine details, which are crucial for high-quality visual rendering. To enhance per-pixel lighting, normal mapping is often employed to simulate complex surface details without increasing the geometric complexity of the model. However, accurate normal vector calculations become challenging in multi-object models where adjacent surfaces with differing orientations and geometries share vertices. This paper presents a method for calculating proper normal vectors for multi-object models, ensuring visually consistent and realistic lighting effects in complex scenes. The proposed technique addresses common issues such as normal interpolation artifacts and lighting inconsistencies by dynamically adjusting normal vectors based on the spatial relationships of multi-objects models. Results demonstrate that the method significantly improves the visual quality of per-pixel lighting with normal mapping, providing a more robust solution for real-time rendering in computer graphics applications.

Keywords: Per-pixel lighting, normal smoothing, multi-object models

1. Introduction

Computer graphics have become an integral part of modern technology, influencing a wide range of industries from entertainment and virtual reality to scientific visualization and industrial design. The ability to create realistic and immersive 3D environments has not only revolutionized visual media but also enhanced our capacity to simulate and analyze complex systems in fields such as engineering, medicine, and education. As the demand for more lifelike and dynamic visualizations grows, the techniques used to render these 3D environments must continually evolve to meet the expectations of realism and performance.

One of the most critical aspects of 3D visualization is the accurate simulation of light. Lighting is not merely a cosmetic feature; it is fundamental to how we perceive shapes, textures, and spatial relationships within a virtual scene. The way light interacts with surfaces, casting shadows, reflecting off objects, and refracting through transparent materials, contributes significantly to the visual realism and the immersive quality of the rendered scene. Without effective lighting, even the most detailed 3D models can appear flat and lifeless, undermining the viewer's ability to interpret the visual information accurately [9].

In multi-object 3D models, where numerous surfaces and materials interact within a shared space, the complexity of lighting calculations increases significantly. Each object not only receives light from various sources but also influences the lighting environment by casting shadows, reflecting light onto neighboring objects, and

contributing to global illumination effects. Accurately modeling these interactions is essential for achieving photorealistic renderings that can be used in applications ranging from cinematic special effects to architectural visualization and virtual prototyping.
This paper aims to explore advanced techniques for calculating and optimizing lighting in multi-object 3D models. We present how vertex attributes should be calculated and handled effectively in case of normal mapping technique.

2. Lights in computer visualization

The need for light modeling emerged at the very beginning of computer visualization. Initially, only offline implementations were possible due to limited hardware and low computational power. However, as hardware improved, the demand for real-time light modeling also arose. Nowadays, increasingly sophisticated solutions are available. Lighting is a fundamental element in computer graphics, playing a crucial role in defining the visual realism, mood, and depth of a scene. It significantly impacts the perception of objects and environments by simulating how light interacts with surfaces. In 3D graphics, lighting models are used to calculate the effects of light on surfaces, including specular highlights, shadows, and reflections. These effects help to create the illusion of depth and enhance the realism of scenes. Advanced techniques like global illumination and ray tracing take into account the complex interactions of light, such as scattering, refraction, and indirect lighting, further improving visual fidelity.

2.1. Lighting models

In real-world environments, the appearance of objects is affected by light sources. These effects can be simulated using a lighting model. A lighting model is a set of equations that approximates (models) the effect of light sources on an object. A Lighting Equation in computer science refers to a formula used to calculate the final color value of an object. The equation helps in determining how light interacts with objects in a scene to create realistic visual effects. The lighting model may include reflection, absorption, and transmission of a light source. The lighting model computes the color at one point on the surface of an object, using information about the light sources, the object position and surface characteristics, and perhaps information about the location of the viewer and the rest of the environment containing the object (such as other reflective objects in the scene, atmospheric properties, and so on):

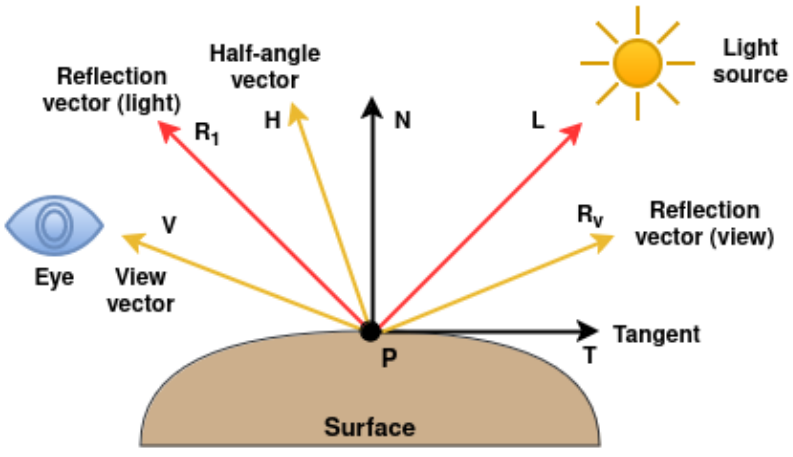

Figure 1. Basic elements of modeling light in a 3D space

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light. In the simplest form (used by OpenGL and Direct3D), the function is simply the sum of these lighting components (clamped to a maximum color value). Thus we have an ambient term and then a sum of all the light from the light sources [15].

$$\square_{total} = \square_{\square\square\square} + \sum(\square_{\square\square\square} + \square_{\square\square\square})$$

where i_{total} is the intensity of light (as an rgb value) from the sum of the intensity of the global ambient value and the diffuse and specular components of the light from the light sources. This is called a local lighting model since the only light on a vertex is from a light source, not from other objects.

During the years many lighting models with different purposes have been developed. These models range from simple approximations to highly sophisticated algorithms, each with its strengths and trade-offs between computational efficiency and visual realism. The most well-known models are:

Types of Lighting Models [5][13]:

1. **Phong Lighting Model (1975)**: One of the earliest and most widely used lighting models, Phong lighting introduces the concept of specular highlights (shiny spots), ambient light, and diffuse reflection. It simplifies light calculations by computing reflections based on the viewer's perspective, making it ideal for real-time rendering.

2. **Blinn-Phong Model**: A variant of Phong lighting developed by Jim Blinn, this model introduces a slight modification to the specular reflection calculation, offering improved performance and more accurate highlights for a wider range of viewing angles.

3. **Lambertian Reflection Model:** This model is used to simulate diffuse reflection where the surface scatters light equally in all directions. It's based on Lambert's cosine law, which states that the intensity of light is directly proportional to the cosine of the angle between the light source and the surface normal.

4. **Physically-Based Rendering (PBR)**: Recent advancements have shifted towards PBR, where lighting is modeled to match real-world physical properties more closely. PBR integrates concepts like energy conservation, fresnel effects, and microfacet theory to deliver highly realistic lighting simulations, essential for modern gaming and film industries.
5. **Oren-Nayar Model:** An extension of the Lambertian model, the Oren-Nayar model simulates rough surfaces by considering the scattering of light due to micro-occlusions on the surface.

6. **Ward Anisotropic Model:** This model is designed to handle anisotropic reflections, which occur on surfaces like brushed metal or hair, where the reflection varies depending on the direction.

Lighting Techniques [5]:

1. **Global Illumination (GI)**: GI techniques simulate how light interacts with multiple surfaces in a scene, allowing light to bounce and scatter,

contributing to more realistic ambient lighting and soft shadows. Techniques such as **ray tracing** and **radiosity** are commonly used for GI.

2. **Ray Tracing**: This technique traces the path of individual rays of light as they travel through a scene, interacting with surfaces, materials, and light sources. While computationally expensive, ray tracing produces highly accurate reflections, refractions, and shadows.

3. **Radiosity**: Radiosity focuses on how diffuse surfaces exchange light. It calculates the light reflecting off surfaces and how this contributes to the overall scene's lighting. Radiosity is particularly effective in rendering interiors and architectural models, where indirect lighting plays a significant role.

2.2. How lighting fits into the graphics pipeline

A graphics pipeline is a conceptual framework used in computer graphics to describe the steps involved in rendering 2D or 3D images from a scene description. It is called a "pipeline" because the data passes through several sequential stages, where different operations are performed to transform the scene's geometry, apply textures and colors, and calculate the lighting, shading, and final image [2]. The fixed-function graphics pipeline and the programmable graphics pipeline represent two different approaches to how graphics hardware handles lighting, shading, and other rendering tasks. While the fixed-function pipeline offers predefined, limited functionality, the programmable pipeline provides more flexibility and control for developers.
Nowadays, we almost exclusively use the programmable graphics pipeline in computer visualization. With the introduction of programmable shaders (vertex and fragment/pixel shaders), modern GPUs allow developers to customize every step of the rendering process, including lighting. This flexibility allows for more sophisticated and realistic lighting effects. The following image shows the OpenGL 4 pipeline [8].

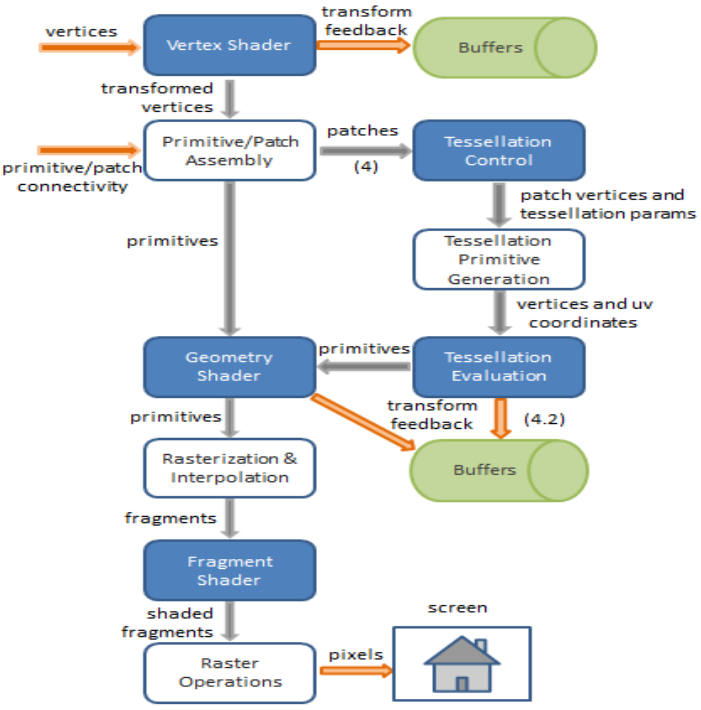

Figure 2. The OpenGL 4 pipeline

In the case of previous fixed-function pipelines, pre-programmed lighting models could be used. Based on Figure 2, it is clear that today the implementation of the applied lighting algorithm has shifted to the programmer's side. It is the programmer's task to develop and program the given lighting model using vertex and fragment shaders [6][7]. The following table compares the two approaches.

Table 1. Fixed vs Programmable pipeline comparison

Feature	Fixed-function Pipeline	Programmable Pipeline
Lighting Model	Predefined models (e.g., Phong)	Customizable lighting models, including PBR
Number of Light Sources	Limited (usually up to 8)	Customizable lighting models, including PBR
Lighting Calculation	Per-vertex lighting, interpolated across surfaces	Per-pixel lighting (more detailed and realistic)
Shadows and Advanced Effects	Limited or absent	Real-time shadows, global illumination, custom effects
Control and Flexibility	Little to no control over lighting equations	Full control over how lighting is calculated
Surface Detail	Basic lighting; limited ability to represent fine details	Fine surface detail through bump mapping, normal mapping, etc.
Shader Use	No shaders, hard-coded operations	Custom vertex and fragment shaders for fine control
Performance	Faster, but less realistic	More demanding, but much more realistic

2.3. Lighting in the virtual world

In computer graphics, lighting models play a crucial role in rendering realistic images by simulating how light interacts with surfaces. Two widely used techniques for light computation are **vertex-based lighting** (also known as Gouraud shading) and **per-pixel lighting** (commonly implemented through Phong shading or more advanced pixel shaders). Both approaches have distinct characteristics, strengths, and limitations that influence their use in different contexts [3].

2.3.1 Vertex-Based Lighting

Vertex-based lighting calculates lighting at the vertices of a 3D model and interpolates these values across the surface of the polygon. This method is computationally efficient because it requires lighting calculations only at the model's vertices, significantly reducing the number of operations required. The efficiency of vertex-based lighting makes it particularly suitable for real-time applications where rendering speed is crucial, such as older video games or applications running on low-power devices. The technique is straightforward to implement and demands less processing power, which was especially advantageous in the early days of computer graphics when hardware capabilities were limited. However, vertex-based lighting has notable limitations. Because the lighting is interpolated between vertices, it can result in visual artifacts or a lack of fine detail,

particularly on surfaces with low tessellation. For instance, specular highlights may appear blurry or washed out. Additionally, vertex-based lighting struggles with complex lighting scenarios, such as sharp reflections, spotlights, or fine surface details, because the interpolation cannot accurately capture rapid changes in light intensity or color.

2.3.2 Per-Pixel Lighting

Per-pixel lighting, also known as Phong shading, on the other hand, performs lighting calculations for each individual pixel on a surface, allowing for a much finer granularity in how light interacts with the surface's material properties. This method results in more detailed and realistic images. The primary advantage of per-pixel lighting is its high visual fidelity. It provides much more detailed and accurate lighting effects than vertex-based methods, accurately modeling specular highlights, reflections, and other intricate lighting phenomena, which greatly enhances realism [10]. In modern graphics programming, per-pixel lighting is implemented using programmable shaders. This approach allows developers to create highly realistic effects like bump mapping, normal mapping, and dynamic shadows, making it a cornerstone of contemporary 3D rendering in video games, simulations, and virtual reality. However, per-pixel lighting comes with higher computational costs. Calculating lighting for each pixel requires significantly more processing power and memory bandwidth, which can be a limiting factor in real-time applications, especially on less capable hardware. Moreover, implementing per-pixel lighting often involves more complex algorithms and shader programming, which can increase development time and complexity.

2.3.3. Choosing Between the Two Approaches

The choice between vertex-based and per-pixel lighting depends on the specific requirements of the application, the desired level of visual fidelity, and the available hardware resources. Vertex-based lighting remains useful in scenarios where simplicity and speed are more important than fine visual details, such as mobile games or applications with low-poly models. In contrast, per-pixel lighting is favored in applications where high-quality visuals are essential, such as modern video games, CGI in movies, or virtual reality experiences.

The evolution from vertex-based to per-pixel lighting reflects a broader trend in computer graphics towards achieving photorealism [15]. As hardware capabilities, including CPUs and GPUs, have advanced, it has become feasible to employ more complex and computationally demanding techniques like per-pixel lighting, which offer significant improvements in visual quality. Today, many graphics engines dynamically adjust the level of detail, using a combination of vertex-based and per-pixel lighting techniques to optimize performance while maintaining high-quality visuals.

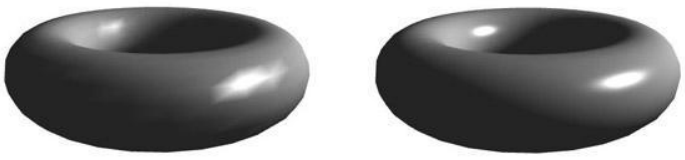

Figure 3. Vertex vs per-pixel lighting

In this paper our objective is to apply per-pixel lighting in the rendering process.

3. Per-pixel lightning for Multi-object models

The field of lighting and its modeling within computer visualization already requires a more complex environment. In such cases, a minimal graphics engine (possibly as part of a game engine) is usually created, which is capable of performing the following functions in a structured form:

- load different models,
- store their data in an appropriate internal structure,
- have some approach or structure to describe the "virtual world",
- handle multiple cameras, which can even be movable (e.g., FPS camera),
- handle basic shaders,
- describe lights and their parameters with an internal object structure,
- have a (simple) mathematical library,
- possibly include additional utility components with various support services.

We can see that a significant amount of work is required to create even a test environment where, by modifying a few lines of code, we are able to test a new setting and visually observe it.

4.1 The process of applying lighting

Naturally, the practical implementation of lighting depends on the API being used, which currently means OpenGL, Vulkan, or DirectX. We can also include software-implemented lighting solutions here. However, regardless of which graphics API we use, illuminating the virtual world requires a lot of preparation.

First and foremost, we need the geometric information that makes up the virtual world (buildings, objects, vegetation, etc.), which we most commonly load from one or more model files. Although there are many different model descriptor structures/formats (Max, Blend, gITF, Obj, FBX, ASE, etc.), they all contain the basic geometry required for lighting. If the application goes beyond a world embedded in the sandbox of the programming environment, some sort of world description structure is usually applied, which also positions the models in space and defines the light sources (along with parameters) that illuminate the virtual world. The preparation phase of the geometric data is considered complete once they are uploaded to the GPU memory.

The rendering process requires the presence of numerous modules/elements, which will not be covered in this article. Typically, this includes loading shaders, managing and switching between them.

In practice, lighting can be implemented in various ways. Certain parts (e.g., matrices) can be calculated on the CPU side or even on the GPU side. This paper will not cover those details. Once all the necessary elements for operation are available, the rendering process within the rendering loop is as follows:

```
Calculate projection, view and model matrix
for loop - model.object.size
    Activate light shader
    for loop - numberOfLights
```

```
        transfer light parameters to GPU shader:
                    -  position,    constant    linear,    quadratic
                       attenuation, ambient, diffuse, specular, etc
        end for
        Set object texture(s)
        Set object's other parameters
       draw object i.
       Stop light shader
end for
```

This pseudo code assumes that all the objects in the scene are affected by the same type of lights, because the code uses the save light shader for all objects. In practice of course different types of light sources can require other types of shader.

4.1 The importance of the normal vectors

The visual effect of a light shining on a surface depends on the properties of the surface and of the light. But it also depends to a great extent on the angle at which the light strikes the surface. The angle is essential to specular reflection and also affects diffuse reflection. That's why a curved, lit surface looks different at different points, even if its surface is a uniform color. To calculate this angle, we need to know the direction in which the surface is facing. That direction is specified by a vector that is perpendicular to the surface. Another word for "perpendicular" is "normal," and a non-zero vector that is perpendicular to a surface at a given point is called a normal vector to that surface.
It is therefore important that the normal vectors are calculated correctly to ensure that the lighting provides the expected visual results. Normal vectors are typically provided in two ways. Very often, the normal vectors are stored together with the model file on the storage medium. In this case, the consistency of the data is ensured by the modeling software used. The graphics engine's only task here is to read these data and apply them.
Alternatively, the normal vectors can be calculated during the model's loading process. While this approach is indispensable during development, in finished games, storing the normals together with the model is preferred because, for larger models, calculating and transforming normals can take several seconds. This could significantly impact the gaming experience.

Basically, two types of normal vectors are typically distinguished:

- **Face normal:** this vector is a face (usually triangle) level vector, which is a perpendicular unit normal vector to the face. The vector's direction is determined by the order in which the vertices are defined and by whether the coordinate system is right- or left-handed. The face normal points away from the front side of the face. One face has one normal vector.

- **Vertex normal:** Vertex normals (sometimes called pseudo-normals) are values stored at each vertex that are most commonly used by a renderer to determine the reflection of lighting or shading models, such as *phong* shading.

4. Normal Mapping

In computer visualization, increasing graphical realism has always played an important role. This is particularly emphasized in real-time software and games, as it is essential to achieve a level of graphical quality that is appropriate for the era in

real-time, compared to offline rendering. One straightforward approach to achieving this is to increase the number of polygons in the virtual world and the models. While this approach is correct and applicable, increasing the vertex count puts a significant burden (especially on early) GPUs: significantly more polygons pass through the graphics pipeline, requiring more polygons to be clipped, rasterized, etc [12].

Another approach that is actually advisable to apply alongside increasing the polygon count is to use larger, higher-resolution textures. However, when viewing surfaces up close, as when we are against a wall in a game, this approach does not result in drastic improvement. This is because, in reality, surfaces are not smooth; they have numerous holes, depressions, or protrusions, which appear in different shades and colors due to the varying angles at which light hits them.

By combining these two approaches, we could model every small surface imperfection of a wall with a vertex mesh, onto which we apply a higher quality texture. However, today's GPUs are not yet capable of handling this; while the result would be visually appealing, it would be slow in terms of performance. A typical example could be a brick wall. Such a wall has a rather rough surface and is certainly not completely smooth: it contains recessed cement sections and numerous smaller holes and cracks. If we view such a surface with the usual rendering, without effects, the sense of depth of the wall disappears.

As a result, other alternative solutions for making surfaces more realistic have developed. These solutions generally work in image space and use additional textures to provide some level of extra detail to an otherwise flat 2D polygon. Several solutions have emerged in the literature (Bump mapping, Displacement mapping, Normal mapping, Parallax mapping) that successfully add "depth" to the 2D image. In this article, we will focus on normal mapping, but what is presented can also be applied to the other methods.

4.1 Normal mapping in practice

When using classic per-pixel lighting, the intensity of light is calculated at the pixel level in the fragment shader. This is done using vertex-level normals. According to the barycentric coordinate system formed by the three vertices, we can linearly interpolate the normals within the shader, resulting in pixel-level normals. This achieves a nice, smooth light diffusion across the surface; however, the model does not interact with the surface material properties, depressions, and cracks. This is particularly noticeable in the deep grooves between bricks, as the surface remains simply smooth.

Therefore, the industry needed a solution that could inform the lighting system about the details of the surface's depths. It is not sufficient to interpolate the normals solely within the triangle. A solution is needed that provides true pixel-level normals. With this technique, the surface can be made much more complex.

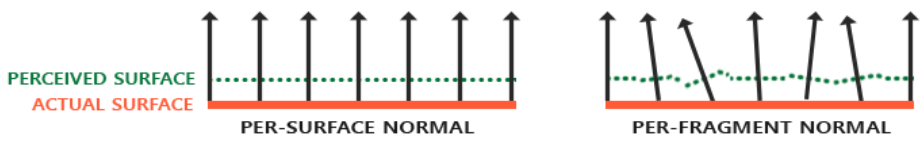

Figure 4. Comparison of surface normals and per-fragment normals [4]

With this solution, we can achieve a visual effect that gives the surface a sense of depth due to the unique light reflections. The technique is essentially a trick that provides the viewer with a much more realistic experience. This approach is collectively referred to as normal or bump mapping.

To implement normal mapping, we need per-pixel normals. A straightforward approach is to use a two-dimensional texture to store the perturbation of the surface normals, similar to a diffuse texture. Since normal vectors can be geometrically interpreted and textures store color information, we need to map the vectors to specific colors in some way.

Colors consist of r, g, b components. We can use these components to store the *x*, *y*, *z* components of the normals. Since the value range of normal vectors is between -1 and 1, the first step is to transform this to the [0,1] range.

```
// transforms from [-1,1] to [0,1]
vec3 rgb_normal = normal * 0.5 - 0.5;
```

This approach enables us to store the normals of a surface on a per-pixel basis in a 2D texture. The following example showcases the normal map of a brick wall:

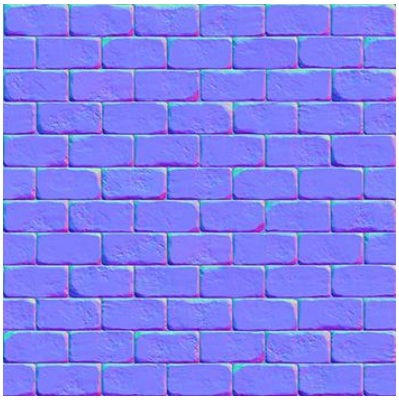

Figure 5. Normal values stored in an RGB texture

It is evident that the normal map has a bluish tint. This is because almost all normals point outward from the surface in the positive *z* direction (0, 0, 1), which corresponds to the blue color. Color variations can be observed where the surface deviates from the typical smoothness. At these points, the normals diverge from the positive *z* direction, and these areas will provide the viewer with a sense of depth. Note the top of the brick, which receives a greenish hue almost everywhere. This is because, at these points, the normals increasingly point in the positive *y* direction (0, 1, 0), which maps to the green color.

4.2 Tangent Space

The tangent space is a space that is locally defined as an orthogonal system relative to the surface of a triangle. Normals are specified in this reference coordinate system relatively. We can think of this local space as the space of the normal map vectors, where each normal is defined to point in the positive *z* direction. Using a special matrix that defines the tangent space, the normal vectors defined in the local space can be transformed into the world or camera space according to the final orientation of the given surface.

This matrix is called the TBN matrix, whose components are the *tangent*, *bitangent*, and *normal* vectors. The vectors form an orthogonal system, meaning they are perpendicular to each other. The normal vector points outward from the surface, while the right and forward-pointing vectors represent the *tangent* and *bitangent*. The following figure illustrates the relationship between the three vectors:

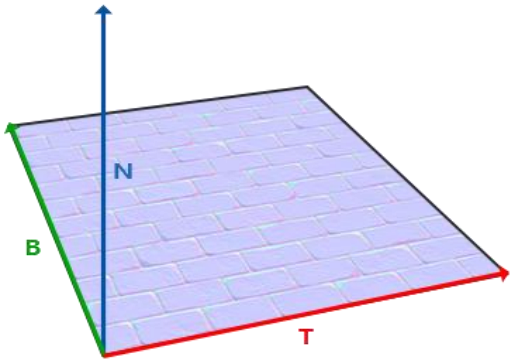

Figure 6. Tangent space described by tangent, bitangent and normal vectors

While the calculation of the normal vector is relatively straightforward, the calculation of the tangent and bitangent vectors is not self-evident. We know that the tangent vector is perpendicular to the normal vector. However, many such vectors can be defined:

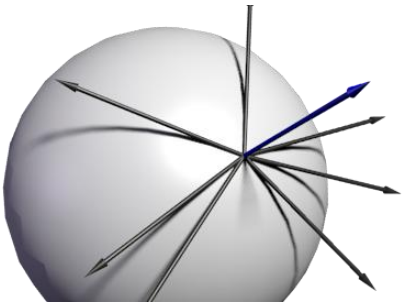

Figure 7. An arbitrary number of tangent vectors can be defined alongside the normal vector indicated in blue

Theoretically, any perpendicular vector can be used as a tangent vector, but it is advisable to be consistent with adjacent faces; otherwise, the edges between the faces will look unattractive during shading. The accepted solution is to direct the tangent vector in the direction indicated by the texture coordinates. Once the tangent vector is determined, the bitangent can be easily calculated. The figure below illustrates this.

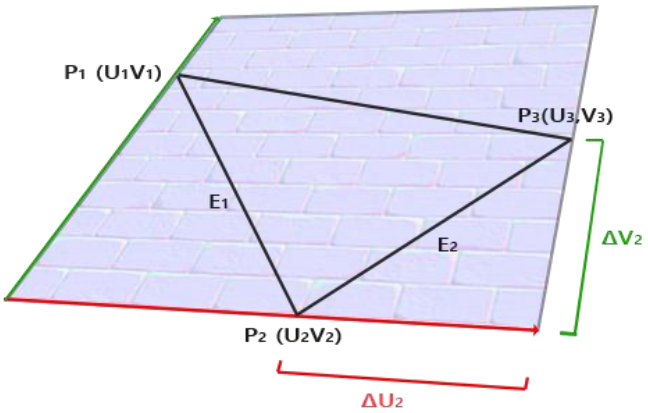

Figure 8. The orientation of the tangent vector always corresponds to the direction of the texture coordinates

In the figure, the texture coordinate differences of the triangle's edge E2 are

denoted by ΔU2 and ΔV2. Their direction corresponds to the directions of the tangent (T) and bitangent (B) vectors. Based on this, both edges (E1, E2) can be expressed as a linear combination of the T and B vectors:

$$E_1=\Delta U_1 T + \Delta V_1 B$$
$$E_2=\Delta U_2 T + \Delta V_2 B$$

Based on this:

$$(E_{1x},E_{1y},E_{1z}) = \Delta U_1(Tx,Ty,Tz) + \Delta V_1(Bx,By,Bz),$$
$$(E_{2x},E_{2y},E_{2z}) = \Delta U_2(Tx,Ty,Tz) + \Delta V_2(Bx,By,Bz)$$

The value of E can be calculated from the position difference of two vectors, while ΔU and ΔV are the differences in the texture coordinates. Thus, we have two unknowns (T and B) and two equations. Therefore, the problem is solvable. We can express the problem differently, in the form of matrix multiplication:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Writing the equations in this form greatly facilitates the solution of the T and B vectors. If we multiply both sides by the inverse of the ΔUΔV matrix, we arrive at the following:

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

This allows us to solve the system of equations for T and B. For this, we need to compute the inverse of the texture coordinate matrix as follows: The resulting system of equations provides the calculation of the T and B vectors.

The following sample example demonstrates the calculation of the T and B vectors in practice for a triangle:

```
// Edges of the triangle : position delta
vec3 deltaPos1 = v1 - v0;
vec3 deltaPos2 = v2 - v0;

// UV delta
vec2 deltaUV1 = uv1 - uv0;
vec2 deltaUV2 = uv2 - uv0;
// calculate tangent and bitangent
float r = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV1.y * deltaUV2.x);
vec3 tangent = (deltaPos1 * deltaUV2.y - deltaPos2 * deltaUV1.y) * r;
vec3 bitangent = (deltaPos2 * deltaUV1.x - deltaPos1 * deltaUV2.x) * r;
// Normalize results
normalize(tangent);
normalize(bitangent);
```

Since a triangle is always a planar shape, we need a tangent/bitangent pair, meaning that each vertex of the triangle will have one corresponding pair. In the following, we will demonstrate the results of the implemented lighting model on a relatively more complex sample model, on a head [1]. Every illustration rendering scene, test case run in our self made platform independent 3D engine. The engine was implemented in C++ and uses OpenGL/GLSL for GPU related programming. We use two light sources for better demonstration results.

Figure 9. Head rendering with face level normal/tangent/bitangent attributes. The result is similar to flat shading

It is evident that the result does not meet the expectations; it resembles flat shading. Several errors can also be detected on the model, which occur because in this case we intentionally applied face-level vertex attributes (normal vector, tangent, bitangent) for the triangles to illustrate the problem. In the case of a more complex model, where most of the displayed surfaces are curved, the face-level normal cannot provide satisfactory results.

5. Lighting correction

5.1 Vertex attributes smoothing

The rendering result presented in Figure 9 is not satisfactory because the normal vectors of adjacent triangles that make up the model are “far” apart in direction, leading to noticeable light refraction during lighting calculations. To achieve satisfactory lighting results, we need to switch to using vertex-level normal vectors. Vertex-level normals mean that we can define the values of normal vectors for each vertex, which can be interpolated within the triangle during rasterization thanks to the barycentric coordinate system defined inside the triangle.

One effective solution to reduce the “error” caused by the refraction of light between surfaces with different orientations is to smooth the normals of triangles that share common vertices.

This method is depicted in the diagram below, which presents two surfaces, S1 and S2, viewed edge-on from above. The normal vectors for S1 and S2 are indicated in blue, while the vertex normal vector is highlighted in red. The angle formed between the vertex normal vector and the surface normal of S1 is identical to the angle between the vertex normal and the surface normal of S2. When these two surfaces are illuminated and shaded using Gouraud shading, the result is a smooth, rounded edge between them. The following illustration displays the two surfaces (S1 and S2) along with their corresponding normal vectors and the vertex normal vector:

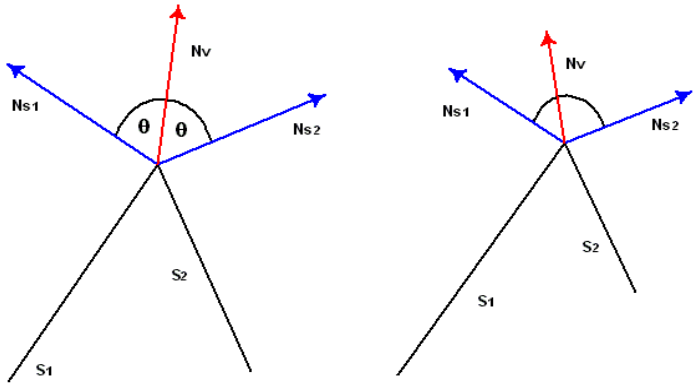

Figure 10. The red arrow indicates the vertex normal vector shared by the two surfaces (S1, S2)

If the vertex normal leans toward one of the faces with which it is associated, it causes the light intensity to increase or decrease for points on that surface, depending on the angle it makes with the light source (Figure 10, right image). The vertex normal leans toward S1, causing it to have a smaller angle with the light source than if the vertex normal had equal angles with the surface normals.

Although we have specifically focused on normal vectors in this case, we must not forget that normal mapping was also applied alongside the lighting model in this example. The *tangent* and *bitangent* vectors used for the tangent space also appear as vertex attributes, so the smoothing must also be applied to the vectors in the tangent space.

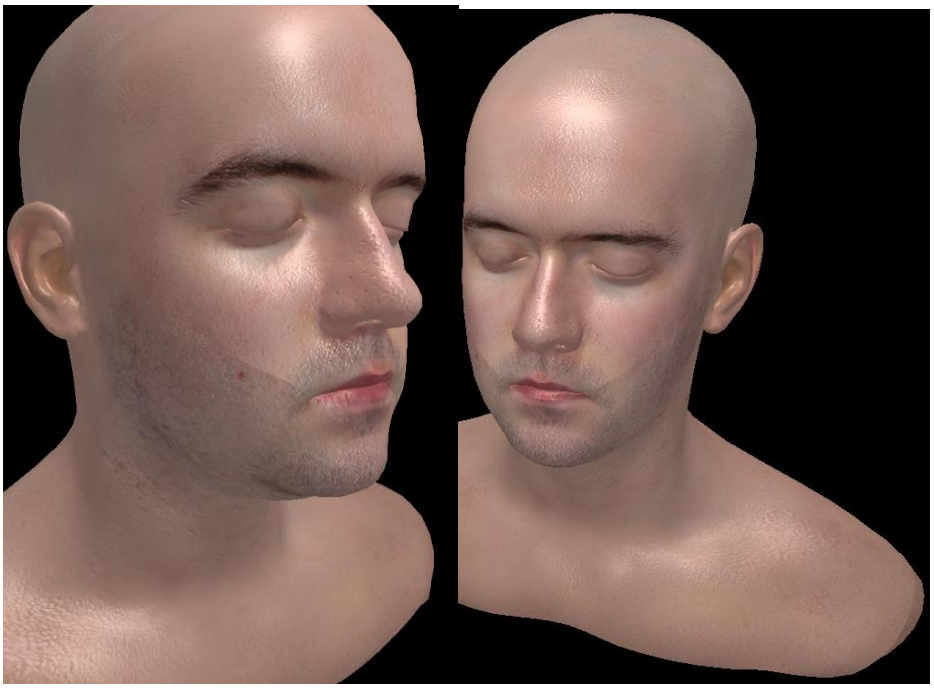

Figure 11. While smoothing vertex attributes greatly improves the results, additional problems/glitches can still be observed in the outcome

5.2 Model level smoothing

The errors observed during rendering, such as the appearance of different "line breaks" on various areas of the model, clearly indicate that further enhancements are necessary. The problem arises because the model was not created as a single set of vertices, but rather as multiple different parts crafted as separate objects. In practice, there can be several reasons for implementing a more complex model in this way. It could be a decomposition based on some logical ordering principle or simply because it was easier to create the final shape this way.

While the previously introduced vertex attribute smoothing approach works well, an algorithm needs to be developed that can take into account the different objects during attribute smoothing. The algorithm should also be extended to neighboring objects that share common vertices. The following sample code demonstrates such an approach:

```
void ComputeAndSmoothVertexAttributes(t3DModel model) {
     CVector3 vVector1, vVector2, vNormal, vPoly[3];
     pModel.numOfAllVertex = 0;

    for (index = 0; index < model.numOfObjects; index++) {
        t3DObject pObject = model.pObject[index];
        model.numOfAllVertex += pObject.numOfFaces*3;

        CVector3 pTempNormals = new CVector3 [pObject.numOfFaces];
        pObject.pNormals = new CVector3 [pObject.numOfVertices];
        pObject.pTangents = new CVector3 [pObject.numOfVertices];
        pObject.pBitangents  = new CVector3 [pObject.numOfVertices];

        for (i = 0; i < pObject.numOfFaces; i++) {
            vPoly[0] = pObject.pVerts[pObject.pFaces[i].vertIndex[0]];
            vPoly[1] = pObject.pVerts[pObject.pFaces[i].vertIndex[1]];
            vPoly[2] = pObject.pVerts[pObject.pFaces[i].vertIndex[2]];

                   // Calculate the face normals
             vVector1 = Vector(vPoly[0], vPoly[2]);
             vVector2 = Vector(vPoly[2], vPoly[1]);
             vNormal = Cross(vVector1, vVector2);
             vNormal = Normalize(vNormal);

             pTempNormals[i] = vNormal;
        }
```

```
    CVector3 vSum(0.0, 0.0, 0.0);
    CVector3 vTagentSum(0.0, 0.0, 0.0);
    CVector3 vBiTagentSum(0.0, 0.0, 0.0);
    CVector3 vZero = vSum;

    int shared = 0;

    for (i = 0; i < pObject.numOfVertices; i++) {
        CVector3 vvertex = pObject.pVerts[i];

        for (subindex = 0; subindex < model.numOfObjects; subindex++) {
            t3DObject pModelObject = model.pObject[subindex];

            for (int j = 0; j < pModelObject.numOfFaces; j++) {
              CVector3 v1 =
             pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[0]];
              CVector3 v2 =
             pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[1]];
              CVector3 v3 =
             pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[2]];

             if (vvertex == v1 || vvertex == v2 || vvertex == v3) {
                vSum = AddVector(vSum, pModelObject.pFaces[j].normal);
                vTagentSum = AddVector(vTagentSum, pModelObject.pFaces[j].tangent);
                vBiTagentSum = AddVector(vBiTagentSum,
             pModelObject.pFaces[j].bitangent);
                        shared++;
              }
           }
        }

         // Get the normal by dividing the sum by the shared
        // Negate the shared so it has the normals pointing out
         pObject.pNormals[i] = DivideVectorByScaler(vSum, shared);
         pObject.pNormals[i] = Normalize(pObject.pNormals[i]);
         pObject.pTangents[i] = DivideVectorByScaler(vTagentSum, shared);
         pObject.pTangents[i] = Normalize(pObject.pTangents[i]);
         pObject.pBitangents[i] = DivideVectorByScaler(vBiTagentSum, shared);
         pObject.pBitangents[i] = Normalize(pObject.pBitangents[i]);

         vSum = vZero;                                    // Reset the sum
         vTagentSum = vZero;                              // Reset the tangent sum
         vBiTagentSum = vZero;                            // Reset the bitangent sum
         shared = 0;                                      // Reset the shared
    }
}
```

The following image shows the result of the algorithm. The generated vertex attributes are now correctly calculated, taking into account the vertices that are shared across the object boundaries.

Figure 12. Multi-object model with smoothing of vertex attributes between objects

5.3 Tangent space orthogonality

It is necessary to mention one final addition to the presented solution, which can further enhance image quality with a slight increase in performance requirements. When working with larger shapes, a vertex can be part of multiple faces. We already know that if we do not smooth or average the tangent vectors, the result is often unsatisfactory. However, the smoothing can have the side effect of causing the new TBN vectors to lose their perpendicularity to each other during the averaging process, meaning the resulting TBN matrix will no longer be orthogonal. Although this is less noticeable in the visual result, it is advisable to take this small detail into account.

To address this issue, we can apply the *Gram-Schmidt* mathematical procedure [14], which allows us to (re)orthogonalize the TBN vectors so that they are perpendicular to each other. This can be done in two ways: after calculating and averaging the TBN vectors or within the vertex shader. The Gram-Schmidt solution is as follows:

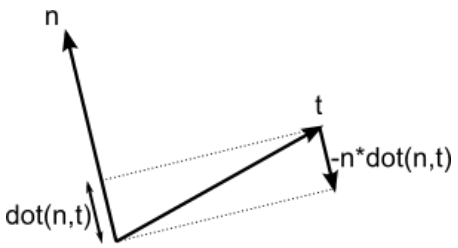

Figure 13. Geometric illustration of the Gram-Schmidt orthogonalization procedure

In the example, the *n* and *t* vectors are nearly perpendicular to each other. To orthogonalize, we simply need to "push" the t vector in the direction of *-n* by the amount of the scalar product of *n* and *t* (*dot(n, t)*). That is:

```
t = normalize(t - n * dot(n, t));
```

Performed in the vertex shader:

```
vec3 T = normalize(vec3(modelMatrix * vec4(tangent, 0.0)));
vec3 N = normalize(vec3(modelMatrix * vec4(normal, 0.0)));
T = normalize(T - dot(T, N) * N); // Re-orthogonalization
vec3 B = cross(T, N); // Bittangent vector calculation
mat3 TBN = mat3(T, B, N)
```

5.4 Coordinate system rotation

In the case of symmetric models, it may occur that the direction of the UV coordinate orientation is incorrect, which will also result in a wrong orientation for the tangent vector. Checking this is very simple. The TBN vectors must define a right-handed coordinate system. For example, the cross product of the *n* and *t* vectors should yield an orientation equal to *b*. To mathematically verify this, the scalar product can be used, stating that vectors *A* and *B* have the same orientation if their scalar product is greater than zero. Therefore, *dot(A, B) > 0*. For the TBN vectors, we need to check the result of *dot(cross(n, t), b)*:

```
if (dot(cross(n, t), b) < 0.0f) {
     t = t * -1.0f;
 }
```

By performing this check for each vertex, we can correct any errors arising from incorrect UV orientations.

6. Conclusion

Real-time lighting modeling plays an important role in computer visualization. The rapid advancement of hardware over the years has allowed for the development of numerous different solutions. While only very limited lighting models could be applied in the early years, today we can enhance image quality with various visual effects, creating a more detailed environment. This article focused on per-pixel lighting for multi-object models. To make the surfaces more lifelike, we applied normal mapping. Achieving the final result required the combined application of several techniques. An important aspect is that, in the case of multi-object models, different objects should always be examined together, and vertex attributes should be calculated and transformed accordingly. This helps eliminate visual problems that may arise at the boundaries of the objects. The presented procedures can be applied even for complex models.

References

[1] 3D Head model: https://www.zbrushcentral.com, 2024
[2] Jason Gregory: Game Engine Architecture, A K Peters/CRC Press; 3rd edition, 2018
[3] Tomas Akenine-Moller, Eric Haines, Naty Hoffman: Real-Time Rendering,4th Edition, A K Peters/CRC Press; 2018
[4] Normal Mapping: https://learnopengl.com/Advanced-Lighting/Normal-Mapping, 2024
[5] Matt Pharr, Wenzel Jakob, Greg Humphreys, Physically Based Rendering: From Theory to Implementation, The MIT Press; 4th edition, 2023
[6] Wolfgang Enge, ShaderX7: Advanced Rendering Techniques, Charles River Media; 1st edition, 2009
[7] Kyle Halladay, Practical Shader Development: Vertex and Fragment Shaders for Game Developers, Apress; 1st ed. edition, 2019
[8] David Wolff, OpenGL 4 Shading Language Cookbook - Third Edition: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, Packt Publishing; 3rd ed. edition, 2018
[9] Péter Mileff, Judit Dudra(2022), The Past and the Future of Computer Visualization, Production Systems and Information Engineering, Volume 10, No 1, pp. 16-29., 2022.
[10] Eric Lengyel, Foundations of Game Engine Development, Volume 2: Rendering, Terathon Software LLC, 2019
[11] Frank Luna, Introduction to 3D Game Programming with DirectX 12, Mercury Learning and Information; Illustrated edition, 2016
[12] Miroslav Dimitrijević, Jelena Letić, Ratko Obradovic: LIGHT AND SHADOW IN 3D MODELING, Machine Design, Vol.5(2013) No.3, 2012, DOI: 50709(3):1821-1259
[13] Thorn, A. 3D Lighting and Materials. In: Moving from Unity to Godot. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-5908-5_5, 2020
[14] Stephen Andrilli, David Hecker, Elementary Linear Algebra (Fourth Edition), 2010, https://doi.org/10.1016/B978-0-12-374751-8.00011-1
[15] Tom McReynolds, David Blythe, Advanced Graphics Programming Using OpenGL, Morgan Kaufmann, 2005.