Vulkan Lab 6: Bump Mapping Techniques (Vulkan 1.3)

1 INTRODUCTION

In the previous lab, we added realism by wrapping 2D images (textures) onto our 3D models. However, the surfaces are still perfectly flat, and the lighting doesn't interact with any small-scale bumps or grooves. **Bump Mapping** is a powerful set of techniques that solves this by faking fine-grained surface detail without adding *any* extra geometry.

This lab will start from **Normal Mapping**, the most common form of bump mapping and then further explore other bump mapping methods, such as height mapping, procedural bump mapping and ray-tracing based parallax mapping. We will simulate intricate surface details like bumps, cracks, and rivets by replacing the smooth vertex normal with a detailed normal calculated from a texture carrying special information defined based on tangent space. This will require us to learn about a new coordinate system—**Tangent Space**—which is the foundation for most advanced shading techniques.

2 LEARNING OBJECTIVES

Upon successful completion of this lab, you will be able to:

- **Understand Tangent Space:** Explain the purpose of the Tangent-Binormal-Normal (TBN) coordinate system and its role in shading.
- **Generate TBN Vectors:** Extend your Vertex struct and input layout to include tangent and binormal vectors.
- **Implement Normal Mapping:** Load a texture representing spatial geometric information, bind it to a shader, and use it to replace the vertex normal in lighting calculations.
- **Perform Tangent-Space Lighting:** Transform light and view directions into tangent space within the vertex shader to correctly perform lighting calculations in the fragment shader.
- **Explore Different Bump Mapping Techniques:** Differentiate between Normal Mapping, Height Mapping, and Parallax Mapping as methods for creating detailed surface illusions.

3 CORE CONCEPTS

- **The Illusion of Detail:** A lit surface's appearance is determined by its normal vector. By using a texture to store spatial information for every fragment/*pixel* instead of every *vertex*, we can create the illusion of a highly complex, bumpy surface on a very simple, low-poly model.

- **Tangent Space (TBN):** The texture used for implementing bump mapping is defined for flat-surface, which is not defined based on world-space. For a non-planar geometric object, it is defined based on local geometric primitives, varying from vertex to vertex. This locale coordinate system is called Tangent Space.
  - **Normal (N):** The familiar vertex normal, pointing "out" from the surface.
  - **Tangent (T):** A vector that is tangent to the surface and points in the direction of the U texture coordinate.
  - **Binormal (or Binormal) (B):** A vector that is tangent to the surface and points in the direction of the V texture coordinate.
  - **TBN Matrix:** These three vectors form a 3x3 matrix. In the vertex shader, we use this matrix to transform our light and view directions from world-space into tangent-space.
- **Bump Mapping Workflow:**

1. **C++:** Calculate T, B, and N vectors for each vertex. Add tangent and binormal to the Vertex struct.
2. **C++:** Load textures: colour texture (Albedo), normal texture, height texture.
3. **C++:** Bind both textures to Vulkan graphics pipeline.
4. **Vertex Shader:** Create the TBN matrix. Transform necessary geometric elements required for implementing per-fragment lighting from world-space into tangent-space.
5. **Fragment Shader:** Calculate the normal from the provided spatial texture maps and apply it in your lighting light equation.

- **Height Maps vs. Parallax:**
  - **Height Map:** A grayscale texture where white = high and black = low. It just stores displacement.
  - **Parallax Mapping:** A more advanced technique that uses a height map. It displaces the *texture coordinates* based on the view angle, creating a 3D illusion with apparent depth.

4 LAB EXERCISES

EXERCISE 1: PREPARING FOR TANGENT SPACE

1. **Goal:** Update the application to support tangent and binormal vectors, which are required for normal mapping.
2. **Implementation:**

- **Modify Vertex Struct:** Add tangent and binormal to your C++ Vertex struct.

```
struct Vertex {
   glm::vec3 pos;
   glm::vec3 color;
   glm::vec3 normal;
   glm::vec2 texCoord;
   glm::vec3 tangent;   // New
   glm::vec3 binormal; // New
};
```

- **Update Vertex Input Description:** Add two new VkVertexInputAttributeDescription entries for tangent and binormal. Remember to update all location and offset values.
- **Update Vertex Data:** Calculating tangents for a complex mesh is hard, but for our simple quad, it's easy. Update your Quad_vertices (or equivalent) with the TBN vectors. For a flat quad on the XY plane with Normal = (0,0,1), the Tangent will point along the x-axis and the Binormal along the y-axis.

```
const std::vector<Vertex> Quad_vertices = {
   //        pos                color              normal             texCoord      tangent          binormal
   {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 1.0f, 0.0f}},
   {{ 0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 1.0f, 0.0f}},
   ... ...

  ... ...
};
```

- **Get Textures:** In addition to a colour texture, you need a normal map for normal mapping, a height map for height mapping and parallax mapping. Normal maps are typically purple-ish images.

EXERCISE 2: NORMAL MAPPING IMPLEMENTATION

1. **Goal:** Implement the full normal mapping pipeline to render a bumpy-looking surface.
2. **Implementation:**

- **C++:**

  Create a set of VkImage/VkImageView/VkSampler for the colour texture and the normal texture map.

- **Vertex Shader (shader.vert):**
  - Make a copy of the vertex shader for your per-fragment lighting implemented in Lab 5.
  - Edit the vertex shader by adding new inputs for inTangent and inBinormal.

```
... ...

... ...

layout(location = 4) in vec3 inTangent;
layout(location = 5) in vec3 inBinormal;

layout(location = 0) out ... ...
... ...
layout(location = 3) out vec2 fragTexCoord;
layout(location = 4) out vec3 fragLightPos_tangent;
layout(location = 5) out vec3 fragViewPos_tangent;
layout(location = 6) out vec3 fragPos_tangent;
```

  - In main(), calculate the TBN matrix and transform the directions:

```
void main() {
   // ... (standard MVP for gl_Position)
 ... ...

 ... ...

   // Create the TBN matrix (transforms from world-space to tangent-space)

  Mat4 ModelMatrix_TInv= transpose(inverse(ubo.model);
   vec3 T = normalize(mat3(ModelMatrix_TInv) * inTangent);
   vec3 B = normalize(mat3(ModelMatrix_TInv) * inBinormal);
   vec3 N = normalize(mat3(ModelMatrix_TInv) * inNormal);
   mat3 TBN = transpose(mat3(T, B, N)); // Use transpose to invert

   // Get world-space light and view positions
   vec3 lightPos_world = ubo.lightPos;
   vec3 viewPos_world = ubo.viewPos;
   vec3 fragPos_world = (ubo.model * vec4(inPosition, 1.0)).xyz;

   // Transform light and view POSITIONS to tangent space
   fragLightPos_tangent = TBN * lightPos_world;
   fragViewPos_tangent = TBN * viewPos_world;
   fragPos_tangent =TBN *fragPos_world;
}
```

- **Fragment Shader (shader.frag):**
  - Add the new in variables and the new sampler2D for the normal map.

```
layout(binding = 1) uniform sampler2D colSampler;
layout(binding = 2) uniform sampler2D normalSampler;

layout(location = 3) in vec2 fragTexCoord;
layout(location = 4) in vec3 fragLightPos_tangent;
layout(location = 5) in vec3 fragViewPos_tangent;
```

  - In main(), perform all lighting in tangent space.

```
void main() {
  // Get base color
  vec3 albedo = texture(colSampler, fragTexCoord).rgb;

  // Get normal from normal map
  vec3 N_tangent = texture(normalSampler, fragTexCoord).rgb;
  N_tangent = normalize(N_tangent * 2.0 - 1.0);

  // Implement lighting in tangent space based on normal N_tangent
  vec3 ambient = ... ...;
  vec3 diffuse = ... ...;
  vec3 specular = ... ...;

  vec3 result = ambient + diffuse + specular;
  outColor = vec4(result, 1.0);
}
```

3. **Expected Outcome:** The rendered quad (or cube) will now appear with a bumpy surface. If you are using the provided coin textures, you should create a visual effect shown in the left figure. Consider how to modify the normal to make the bump looks more significant shown in the right figure.

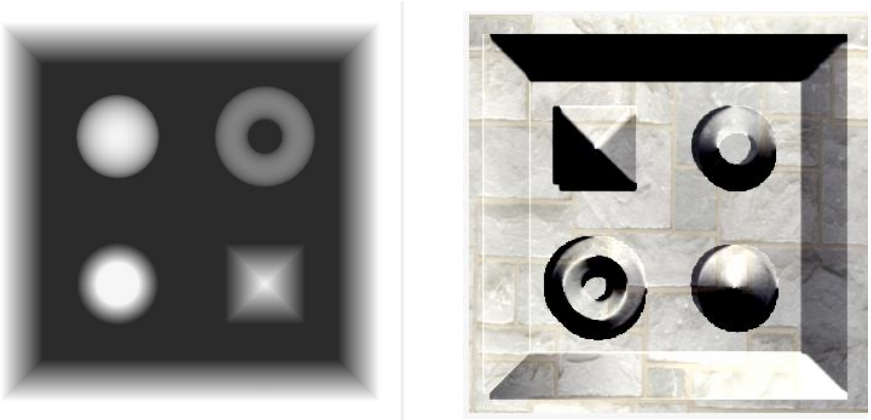
EXERCISE 3: HEIGHT MAP BUMPY EFFECT

1. **Goal:** Create a similar bumpy effect using only a grayscale height map.
2. **Implementation:**

- **C++:** Load a height map (e.g., bricks_height.png) instead of a normal map.
- **Fragment Shader:** You do not have a normal map to sample. Instead, you must calculate the normal. This is commonly done using the dFdx and dFdy functions, which compute derivatives ("slopes") of a value.

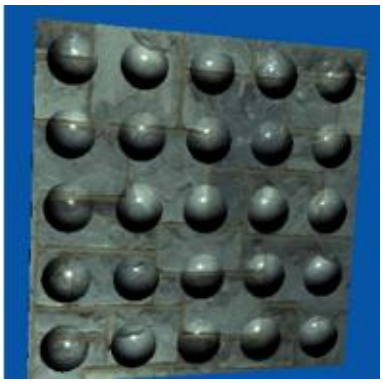
```
// modify the fragment shader for normal mapping by calculating the normal from the
hight //map in the following way:

// Reconstruct the normal from the height map
float h = texture(heightSampler, inTexCoord).r;

// (1). Calculate derivatives of world position w.r.t. screen space
float dFx= dFdx(h);
float dFy= dFdy(h);

float bumpHeight = ... ; //a value between 0.1 and 1
vec3 Normal = normalize( vec3(-dFx, -dFy, bumpHeight));

// Perform lighting in WORLD space using this new ' Normal '
// ...
```

3. **Expected Outcome:** A bumpy surface appearance is created based on the height map without using a normal map. If you are using the height map shown on the left, you should be able to generate the visual effect shown on the right.



EXERCISE 4: PROCEDURAL NORMAL MAPPING

1. **Goal:** Create a bumpy or patterned effect (like waves, bumps) without using either normal map or height map, by calculating a normal procedurally.
2. **Implementation:**
   - **Fragment Shader:** In your fragment shader, invent a normal. For example, the procedural bump effect shown in my lecture note shown below:



3. **Expected Outcome:** A strange, synthetic, but perfectly lit pattern on your object.

EXERCISE 5: RAY MARCHING -BASED PARALLAX MAPPING (OPTIONAL)

1. **Goal:** Use a height map to create a more convincing 3D illusion by shifting texture coordinates.
2. **Implementation:**
   - **C++:** Load a colour texture, a normal map, and a **height map**.
   - **Fragment Shader:**

- This technique *modifies the texture coordinates* before you sample.
- Take the viewDir_tangent you calculated.
- Sample the height from the height map: float height = texture(heightSampler, fragTexCoord).r;
- Displace the texture coordinates based on the height and view angle following the details provided in my lecture note.
  … …
- Use these displacedUVs to sample your color and normal maps to calculate light at each fragment.

4. **Expected Outcome:** A much more realistic 3D effect. As you rotate the object, the "higher" parts of the texture (e.g., bricks) will appear to move over and occlude the "lower" parts (e.g., mortar).

5 FURTHER EXPLORATION (OPTIONAL)

The simple procedural bump pattern created in Exercise 4 is just the beginning of procedural generation. The key to creating natural-looking procedural effects is to use **noise functions**.

- **Implement Noise:** Find a GLSL implementation of a 2D or 3D noise function, such as **Perlin noise** or **Simplex noise**. These functions create smooth, organic, pseudo-random patterns.
- **Fractal Noise (fBm):** Create a "fractal noise" (fBm) function by adding multiple "octaves" of your noise function together. Each octave has a higher frequency and lower amplitude than the last. This is the standard way to create natural-looking textures like clouds, marble, or terrain.

By combining procedural noise with the lighting and parallax techniques from this lab, you can create infinitely detailed, dynamic, and visually stunning effects that use almost no texture memory.