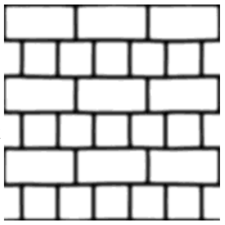
PARALLAX MAPPING

Parallax mapping is a technique similar to normal mapping, but based on different principles.
- Like normal mapping, it is a technique that significantly boosts a textured surface's detail and gives it a sense of depth.
- While also an illusion, parallax mapping is a lot better at conveying a sense of depth. Together with normal mapping, we can achieve incredibly realistic results.

NOTE: Getting an understanding of normal mapping (specifically, tangent space) is strongly advised before learning parallax mapping. Refer to the previous chapter.

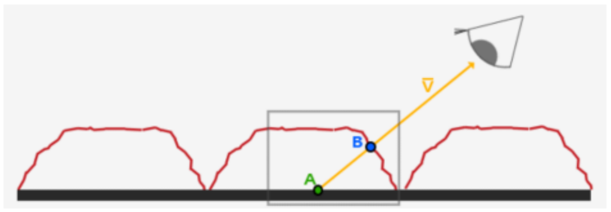
Parallax mapping is closely related to the family of **displacement mapping** techniques that displace or offset vertices based on geometrical information stored inside a texture.
- One way to do this is to take a plane with roughly 1000 vertices and displace each of the vertices based on a value in a texture that tells us the height of the plane at that specific area.
  - Such a texture that contains height values per texel is called a **height map**. An example height map derived from the geometric properties of a simple brick surface looks a bit like the image to the right.
  - When spanned over a plane, each vertex is displaced based on the sampled height value in the height map, transforming a flat plane to a roughly bumpy surface based on a material's geometric properties.
    - Taking a flat plane displaced with the above height map results in the image to the right.



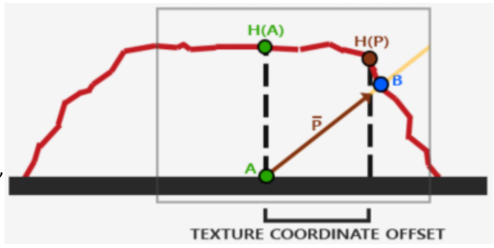
A problem with displacing vertices using the above method is that a plane needs to contain a huge amount of triangles to get a realistic displacement. Otherwise, the displacement looks too blocky.
- As each flat surface may require over 10,000 vertices, this quickly becomes computationally infeasible.

The image to the right actually only consists of two triangles. This is the power of parallax mapping.
- Parallax mapping does not require extra vertex data to convey depth, but (similar to normal mapping) uses a clever technique to trick the user.

The idea behind parallax mapping is to alter the texture coordinates in such a way that it looks like a fragment's surface is higher or lower than it actually is, based on the view direction and a height map. To understand how it works, take a look at the image to the right.
- The rough red line represents the values in the height map as the geometric surface representation of the brick surface.
- The vector $\vec{V}$ represents the surface to view direction.
- If the surface would have actual displacement, the viewer would see the surface at point B. However, as our plane has no actual displacement, the view direction is calculated from point A as we'd expect.
- Parallax mapping aims to offset the texture coordinates at fragment position A in such a way that we get texture coordinates at point B.
- We then use the texture coordinates at point B for all subsequent texture samples, making it look like the viewer is actually looking at point B.

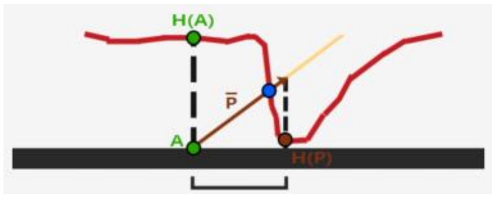


The trick is figuring out how to get the texture coordinates at point B from point A.
- Parallax mapping tries to solve this by scaling the fragment-to-view direction vector V by the height at fragment A.
- So, we're scaling the length of V to be equal to a sampled value from the heightmap $H(A)$ at fragment position A. The image to the right shows this scaled vector $\vec{P}$.
  - We then take this vector $\vec{P}$, we take its vector coordinates that align with the plane as the texture coordiante offset.
    - This works because vector $\vec{P}$ is calculated using a height value from the height map. So, the higher a fragment's height, the more it effectively gets displaced.



The above trick gives good results most of the time, but it is still a really crude approximation to get to point B.
- When heights change rapidly over a surface, the results tend to look unrealistic, since the vector $\vec{P}$ will not end up close to B, as you can see in the image to the right.

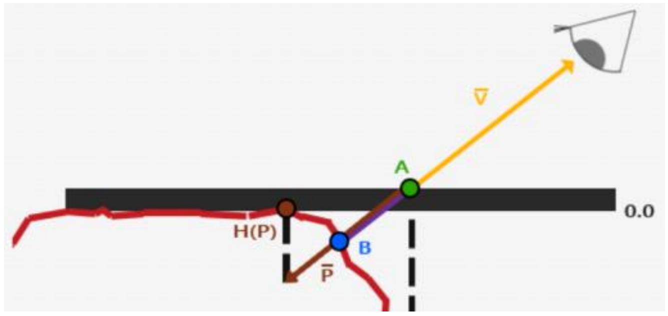
Another issue with parallax mapping is that it's difficult to figure out which coordinates to retrieve from $\vec{P}$ when the surface is arbitrarily rotated in some way.
- We'd rather do this in a different coordinate space where the x and y component of vector $\vec{P}$ always align with the texture's surface. If you've followed along in the normal mapping chapter, you probably know that we can accomplish this in tangent space.
  - By transforming the fragment-to-view direction vector $\vec{V}$ to tangent space, the transformed $\vec{P}$ vector will have its x and y component aligned to the surface's tangent and bitangent vectors.
  - As the tangent and bitangent vectors are pointing in the same direction as the surface's texture coordinates, we can take the x and y components of $\vec{P}$ as the texture coordinate offset, regardless of the surface's orientation.



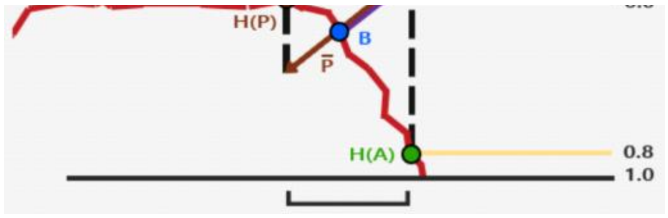
Parallax Mapping

For parallax mapping, we're going to use a simpled 2D plane for which we calculated its tangent and bitangent vectors before sending it to the GPU, similar to what we did in the normal mapping chapter. We are going to attach a diffuse texture, normal map, and displacement map to the plane, as well. We'll be using parallax mapping in conjunction with normal mapping (since the lighting otherwise would not make sense with the displaced texture coordinates, but also because normal maps are typically generated from heightmaps).

Below are all the textures you will need for your parallax (and normal) mapped brick plane.



You may notice that the displacement map is the inverse of the height map shown at the start of the



You may notice that the displacement map is the inverse of the height map shown at the start of the chapter. With parallax mapping, it makes more sense to use the inverse of the height map as it's easier to fake depth than height on flat surfaces. This slightly changed how we perceive parallax mapping, as shown in the image to the right.

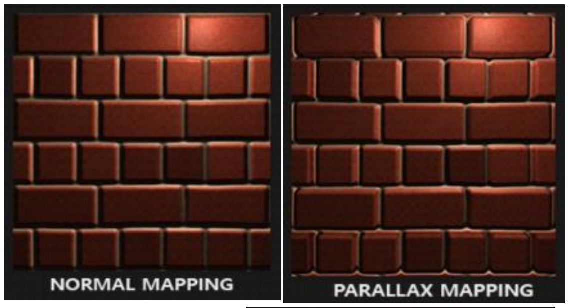
- We again have points A and B, but this time, we obtain vector $\vec{P}$ by subtracting vector $\vec{V}$ from the texture coordinates at point A.
- We can obtain depth values instead of height values by subtracting the sampled height map values from `1.0` in the shaders, or by simply inversing its texture values in image-editing software, as we did with the displacement map linked above.

Parallax mapping is implemented in the fragment shader as the displacment effect is different all over a triangle's surface.

- In the fragment shader, we need to calculate the fragment-to-view direction vector $\vec{V}$, so we need the view position and a fragment position in tangent space.
    - We already have a vertex shader from the previous chapter that sends these vectors in tangent space, so we can just copy it. Here is that vertex shader.

Create a fragment shader that implements parallax mapping.

- Parallax Mapping



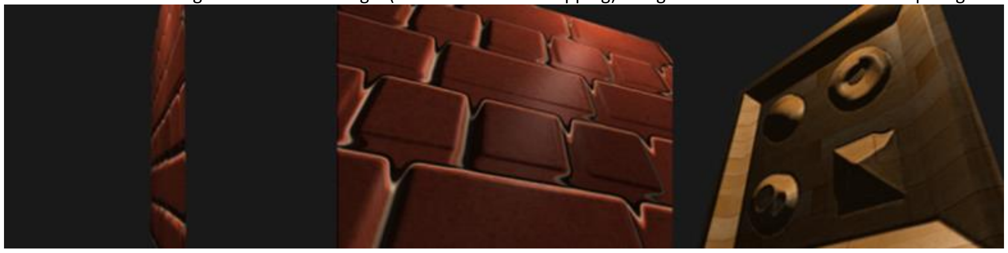
In the images to the right (with the parallax map having a height scaling factor of `0.1`), you can see the difference between normal mapping and parallax mapping combined with normal mapping.

Because parallax mapping tries to simulate depth, it is actually possible to have bricks overlap other bricks based on the direction you view them.

You probably noticed the weird border artifacts at the edge of the parallax mapped plane.

- This happens because, at the edges of the plane, the displaced texture coordinates can oversample outside the range `[0.0, 1.0]`, giving an unrealistic result based on the texture's wrapping model.
- Rather than disallowing repeating textures, we can easily fix this issue by discarding fragments that are sampled outside the default texture coordinate range, like so.
    - **NOTE:** This trick doesn't work on all types of surfaces, but when applied to a plane, it gives great results, as seen in the image to the right.

The effect we've achieved from parallax mapping looks great, is fast, and only requires a single extra texture sample to work. However, it comes with a few issues when being viewed from an angle (similar to normal mapping) and gives incorrect results with steep height changes, as seen below.



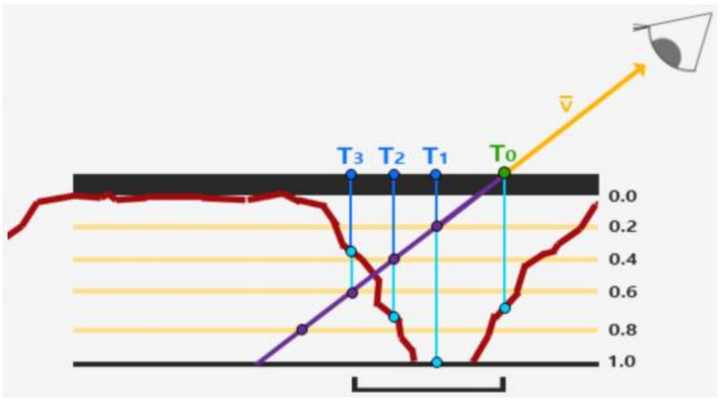
The reason that it doesn't work properly at times is that it's just a crude approximation of displacement mapping.

- There are other tricks that still allow us to get almost perfect results with steep height changes, even when looking at an angle.
- One trick is, instead of taking only one sample, we take multiple samples to find the closest point to B.

Steep Parallax Mapping

Steep Parallax Mapping is an extension on top of parallax mapping that takes multiple samples to better pinpoint vector $\vec{P}$ to B.

- This gives much better results, even with steep height changes, as the accuracy of the technique is improved by the number of samples.
- This technique divides the total depth range into multiple layers of the same height/depth. For each of these layers, we sample the height map, shifting the texture coordinates along the direction of $\vec{P}$, until we find a sampled depth value that is less than the depth value of the current layer.

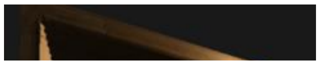


Take a look at the image to the right.

- We traverse the depth layers from the top down and, for each layer, we compare its depth value to the depth value stored in the height map.
    - If the layer's depth value is less than the depth map's value, it means this layer's part of vector $\vec{P}$ is not below the surface.
    - We continue this process until the layer's depth is higher than the value stored in the depth map.
        - This point is then below the (displaced) geometric surface.
- In the image, we can see that the depth map value at the second layer ($D(2) = 0.73$) is lower than the second layer's depth value (`0.4`), so we continue. In the next iteration, the layer's depth value (`0.6`) is higher than the depth map's sampled depth value ($D(3) = 0.37$). We can thus assume vector $\vec{P}$ at the third layer to be the most viable position of the displaced geometry.
    - We then take the texture coordinate offset T_3 from vector $\vec{P_3}$ to displace the fragment's texture coordinates. You can see how the accuracy increases with more depth layers.

Add steep parallax mapping functionality to the ParallaxMapping function.

- Steep Parallax Mapping



With ~10 samples, the brick surface begins to look much better when being viewed from an angle, but doesn't completely demonstrate the power of steep parallax mapping.

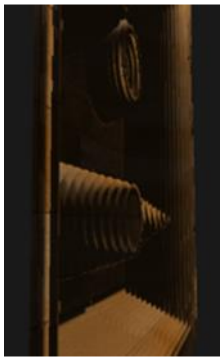
- [Steep Parallax Mapping]

With ~10 samples, the brick surface begins to look much better when being viewed from an angle, but doesn't completely demonstrate the power of steep parallax mapping.

- We'll use the toy box textures you saw earlier to test our steep parallax mapping implementation.
  - The textures for the toy box are provided below.

We can improve our steep parallax mapping algorithm by exploiting one of parallax mapping's properties.

- When looking straight onto a surface, there isn't much texture displacement going on, while there is a lot of displacement when looking at a surface from an angle.
- By taking less samples when looking straight at a surface and more samples when looking at an angle, we only sample the necessary amount.
- This can be accomplished [like so].



Steep parallax mapping also comes with its problems, though.

- Because the technique is based on a finite number of samples, we get aliasing effects and the clear distinctions between layers can be easily spotted, as seen in the image to the right.
  - We can reduce the issue by taking a larger number of samples, but this quickly very low performance.
  - There are several approaches that aim to fix this issue by not taking the first position that''s below the (displaced) surface, but by interpolating between the position's two closest depth layers to find a much closer match to B.

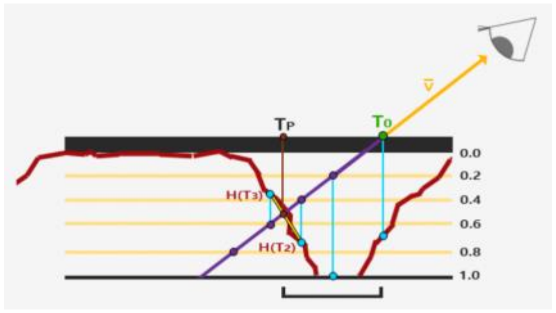
The two most popular approaches to solving the aliasing effect seen in the image to the right are **Relief Parallax Mapping** and **Parallax Occlusion Mapping**.

- Relief parallax mapping gives the most accurate results.
- Parallax occlusion mapping is more performant and gives similar results, making it the preferred approach.

Parallax Occlusion Mapping

Parallax occlusion mapping is similar to steep parallax mapping, but instead of taking the texture coordinates of the first depth layer after a collision, we're going to linearly interpolate between the depth layer after and before the collision.

- We base the weight of the linear interpolation on how far the surface's height is from the depth layer's value of both layers.
- Take a look at the image to the right.
  - It's largely similar to steep parallax mapping with, as an extra step, the linear interpolation between the two depth layers' texture coordinate surrounding the intersected point.
  - While this is an approximation, it is still significantly more accurate than steep parallax mapping.



Extend your ParallaxMapping function to include parallax occlusion mapping.

- [Parallax Occlusion Mapping]

While there are still some artifacts and aliasing issues, the improvement of parallax occlusion mapping over steep parallax mapping is still great, and the issues only really arise when heavily zoomed in or looking at very steep angles.

Most often, parallax mapping is used on floor or wall-like surfaces where it's not as easy to determine the surface's outline and the viewing angle is most often roughly perpendicular to the surface. This way, the artifacts of parallax mapping aren't as noticeable and make it an incredibly interesting technique for boosting your objects' details.

Vertex Shader for Tangent Space Coordinates

Tuesday, June 14, 2022 2:23 PM

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;

out VS_OUT {
   vec3 fragPos; // Added
   vec2 texCoords;
   vec3 tangentLightPos;
   vec3 tangentViewPos;
   vec3 tangentFragPos;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   vs_out.fragPos = vec3(model * vec4(aPos, 1.0));
   vs_out.texCoords = aTexCoords;

   // Gram-Schmidt Process
   mat3 normalMatrix = mat3(transpose(inverse(model))); // Multiplied with tbn vectors to prevent skewed vectors when vertices are scaled
   vec3 t = normalMatrix * aTangent;
   vec3 n = normalMatrix * aNormal;
   // Re-orthogonalizes t with respect to n
   t = normalize(t - dot(t, n) * n);
   // Calculates the perpendicular vector b using the cross product of n and t
   vec3 b = normalMatrix * cross(n, t);
   mat3 tbn = mat3(t, b, n);

   vs_out.tangentLightPos = tbn * lightPos;
   vs_out.tangentViewPos = tbn * viewPos;
   vs_out.tangentFragPos = tbn * vec3(model * vec4(aPos, 1.0));
}
```

The only difference in this vertex shader from the one used in the previous chapter is that we're now using an output interface block to send data to the fragment shader (which now includes `fragPos`) and we multiply the normal, tangent, and bitangent vectors by the normal matrix.

Parallax Mapping

Tuesday, June 14, 2022 2:34 PM

```
#version 330 core

out vec4 FragColor;

in VS_OUT {
   vec3 fragPos;
   vec2 texCoords;
   vec3 tangentLightPos;
   vec3 tangentViewPos;
   vec3 tangentFragPos;
} fs_in;

struct Material {
   sampler2D diffuse;
   sampler2D normal;
   sampler2D height;

   float shininess;
};

struct PointLight {
   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
};

uniform float heightScale;
uniform Material material;
uniform PointLight pointLight;

vec3 CalcPointLight(PointLight light, vec3 viewDir, vec3 normal, vec2 texCoords);
vec2 ParallaxMapping(vec3 viewDir);

void main() {
   vec3 viewDir = normalize(fs_in.tangentViewPos - fs_in.tangentFragPos);

   // Offsets texture coordinates with Parallax Mapping
   vec2 texCoords = ParallaxMapping(viewDir);

   // Samples normal map texture with new texture coords
   vec3 normal = texture(material.normal, texCoords).rgb;
   normal = normalize(normal * 2.0 - 1.0);

   // Proceed with lighting code
   ...
}

// Performs Parallax Mapping on the texture coordinates using the fragment-to-view vector V in tangent space and returns the displaced
// texture coordinates.
vec2 ParallaxMapping(vec3 viewDir) {
   // Samples the height map for the height of the current fragment
   float height = texture(material.height, fs_in.texCoords).r;
   // Calculates P, which is scaled by heightScale
   vec2 p = viewDir.xy / viewDir.z * (height * heightScale);
   return fs_in.texCoords - p;
}

vec3 CalcPointLight(PointLight light, vec3 viewDir, vec3 normal, vec2 texCoords) {
   // Update this code to use the new texture coords for sampling the diffuse map (note: the function signature changed)
   ...
}
```

This fragment shader is basically just the normal mapping fragment shader with parallax mapping implemented as well.

What is interesting to note here is the division of `viewDir.xy` by `viewDir.z`.

- As the `viewDir` vector is normalized already, `viewDir.z` will be somewhere in the range `[0.0, 1.0]`. When `viewDir` is largely parallel to the surface, its z component is close to `0.0` and the division returns a much larger vector $\vec{P}$ compared to when `viewDir` is largely perpendicular to the surface.
- We're adjusting the size of $\vec{P}$ in such a way that it offsets the texture coordinates at a larger scale when looking at a surface from an angle compared to when looking at it from the top; this gives more realistic results at angles.
- Some prefer to leave the division by `viewDir.z` out of the equation as default parallax mapping could produce undesirable results at angles,
  - The technique is then called **Parallax Mapping with Offset Limiting**.

Fixing Oversampling at Edges

Wednesday, June 15, 2022 5:13 PM

```
void main() {
    ...
    vec2 texCoords = ParallaxMapping(viewDir);
    // Discards fragments that are sample outside the range [0.0, 1.0] to prevent border artifacts
    if (texCoords.x < 0.0 || texCoords.x > 1.0 || texCoords.y < 0.0 || texCoords.y > 1.0) {
        discard;
    }
    ...
}
```

Steep Parallax Mapping

Wednesday, June 15, 2022 5:56 PM

```
vec2 ParallaxMapping(vec3 viewDir) {
   // Steep Parallax Mapping
   // Number of depth layers
   const float numLayers = 10;
   // Calculates the size of each layer
   float layerDepth = 1.0 / numLayers;
   // Depth of the current layer
   float currentLayerDepth = 0.0;
   // The amount to shift the texture coordinates by per layer (from vector P)
   vec2 P = viewDir.xy / viewDir.z * heightScale;
   vec2 deltaTexCoords = P / numLayers;

   // Gets the initial values
   vec2 currentTexCoords = fs_in.texCoords;
   float currentDepthMapValue = texture(material.height, currentTexCoords).r;

   while (currentLayerDepth < currentDepthMapValue) {
      // Shifts the texture coordinates along the direction of P
      currentTexCoords -= deltaTexCoords;
      // Gets the depth map value at the current texture coordinates
      currentDepthMapValue = texture(material.height, currentTexCoords).r;
      // Gets the depth of the next layer
      currentLayerDepth += layerDepth;
   }

   return currentTexCoords;
}
```

Controlling Sampling

Wednesday, June 15, 2022 6:27 PM

```
// Minimum and maximum number layers to sample (for optimization)
const float minLayers = 8.0;
const float maxLayers = 32.0;
// Number of depth layers
float numLayers = mix(maxLayers, minLayers, max(dot(vec3(0.0, 0.0, 1.0), viewDir), 0.0));
```

With the `numLayers` calculation, we take the dot product of `viewDir` and the positive z direction vector to get the angle between them (with a minimum angle of `0.0`). This angle is then used by the mix function to generate an interpolated value that is between the `maxLayers` and `minLayers` values.

- The lower the angle, the less samples that will be used.
- The higher the angle, the more samples that will be used.

Parallax Occlusion Mapping

Wednesday, June 15, 2022 7:00 PM

```
// Steep parallax mapping code here
[...]

// Gets the texture coordinates before collision (reverse operations)
vec2 prevTexCoords = currentTexCoords + deltaTexCoords;

// Gets the depth after and before collision for linear interpolation
float afterDepth = currentDepthMapValue - currentLayerDepth;
float beforeDepth = texture(material.height, prevTexCoords).r - currentLayerDepth + layerDepth;

// Interpolates between the after and before depths to determine what texture coords to use
float weight = afterDepth / (afterDepth - beforeDepth);
vec2 finalTexCoords = prevTexCoords * weight + currentTexCoords * (1.0 - weight);

return finalTexCoords;
```