Computer Graphics II
– Parallax Mapping

Kai Lawonn

Introduction

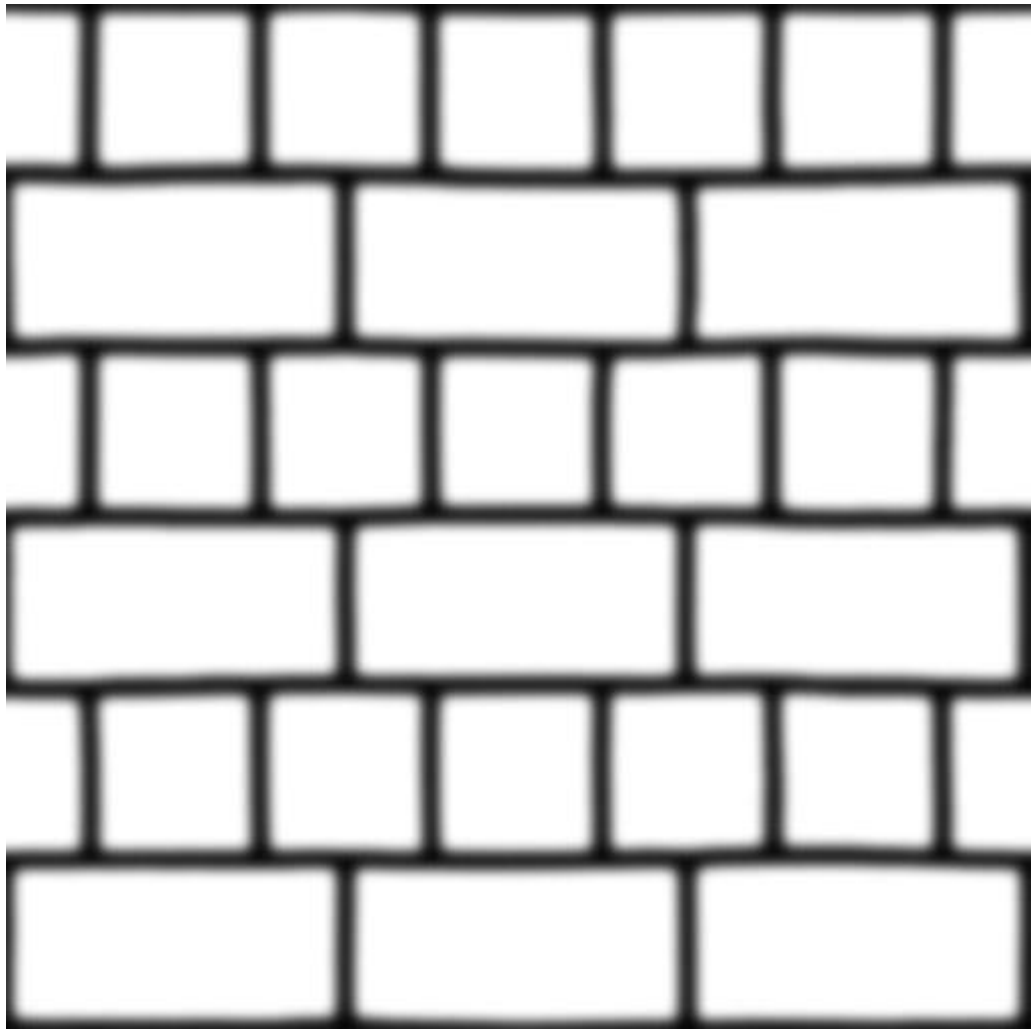
- Parallax mapping is similar to normal mapping, a technique that boosts a textured surface's detail and gives it a sense of depth
- While also an illusion, parallax mapping is better in conveying a sense of depth (with normal mapping realistic results)
- Parallax mapping is a logical follow-up of normal mapping

Introduction

- Parallax mapping is a displacement mapping techniques (displace vertices based on geom. information stored inside a texture)
- One way to do this is, take a plane with roughly 1000 vertices and displace each of these vertices based on a value in a texture that tells us the height of the plane at a specific area
- Such a texture containing height values per texel is a height map

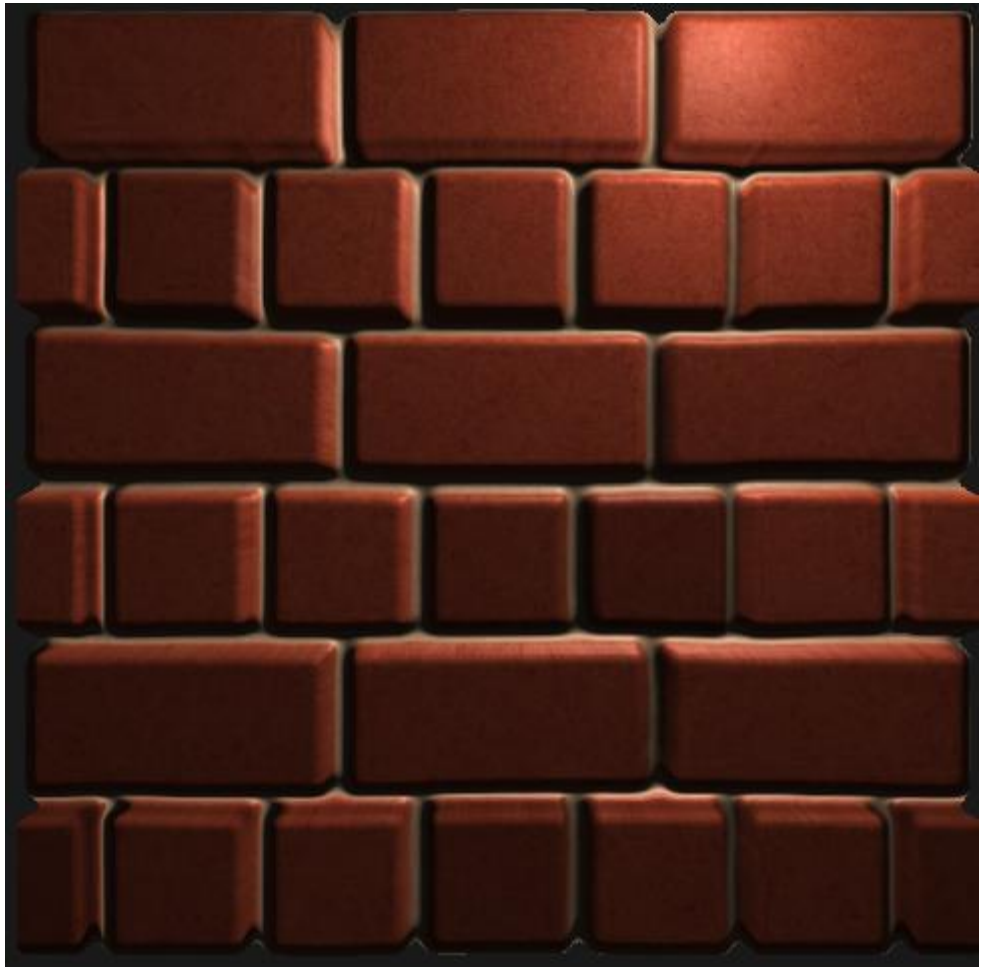
Introduction

- An example heightmap derived from the geometric properties of a simple brick surface looks a bit like this:



Introduction

- Spanned over a plane each vertex is displaced based on the height value in the heightmap
- Flat plane → rough bumpy surface based on a material's geometric properties
- E.g., flat plane displaced with the previous heightmap:

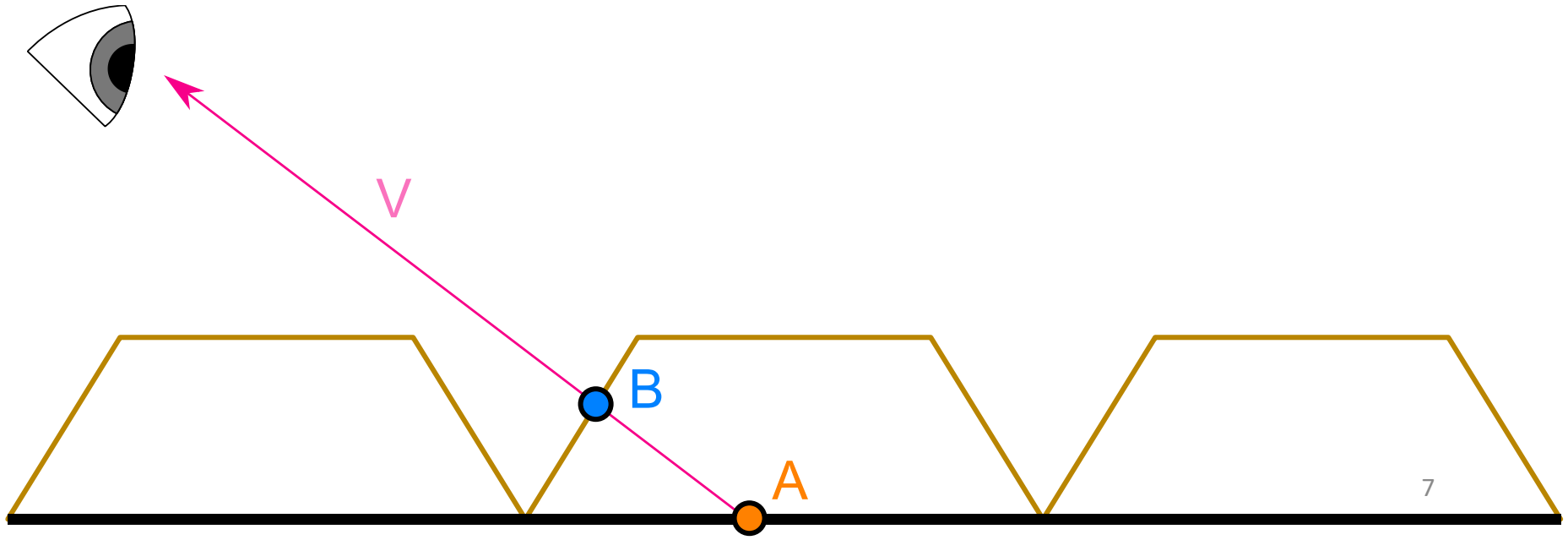


Introduction

- Problem: needs a large amount of triangles to get a realistic displacement otherwise the displacement looks too blocky
- Each flat surface may require 1000 vertices → computationally infeasible
- Achieve similar realism without the need of extra vertices?
- In fact, the previous displaced surface rendered with only 6 vertices
- Rendered with parallax mapping, a displacement mapping technique that does not require extra vertex data to convey depth

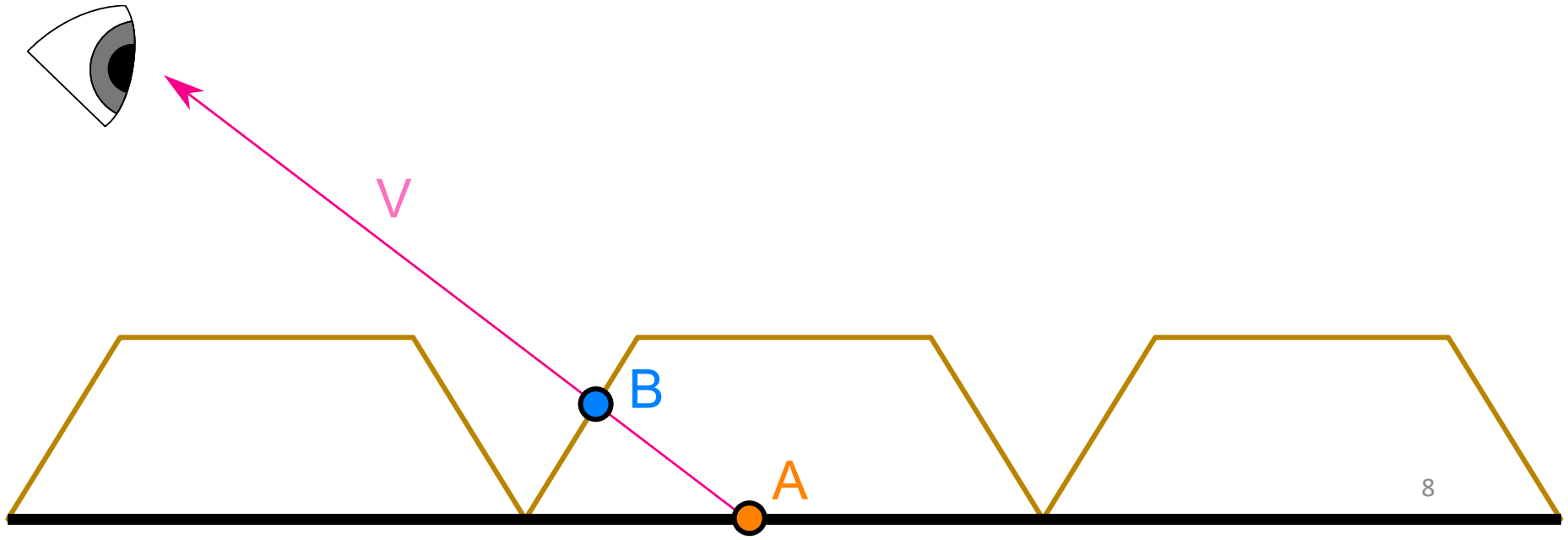
Introduction

- Idea is to alter the texture coordinates that it looks like a fragment's surface is higher or lower based on the view direction and a heightmap
- To understand how it works, take a look at the following image of our brick surface:



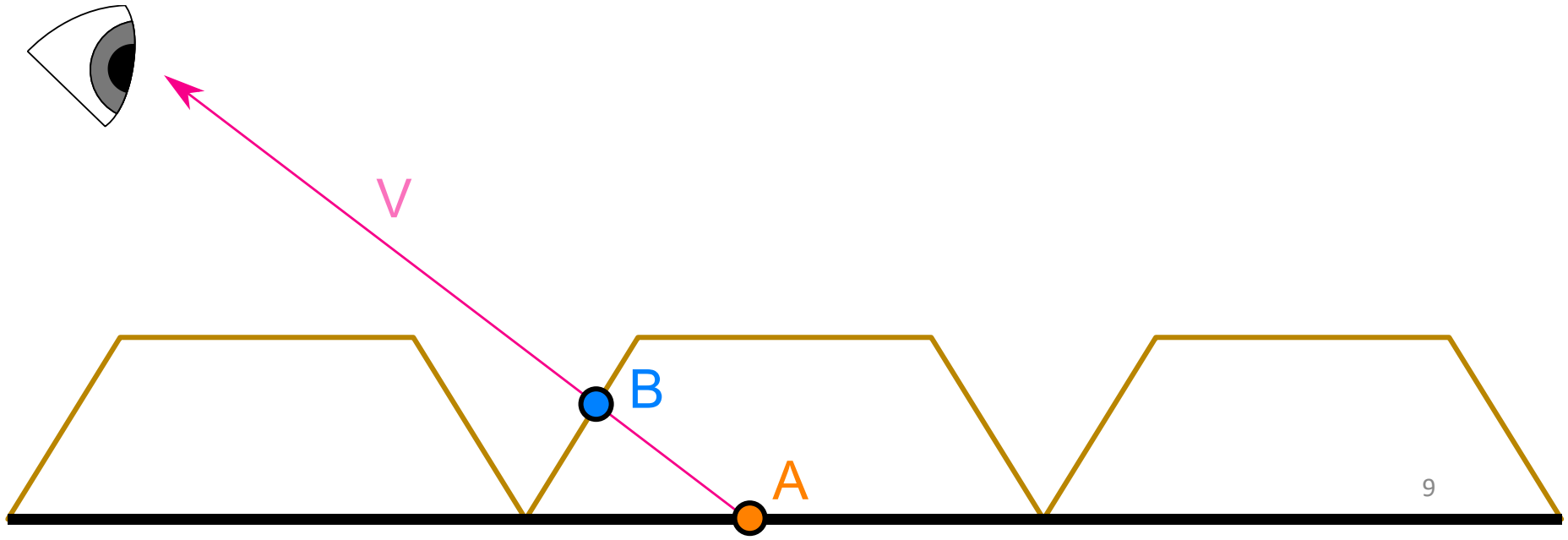
Introduction

- Brown line values in the heightmap (geometric surface representation of the brick surface); V view direction (viewDir)
- With displacement, viewer sees surface at point B; plane has no actual displacement → view direction hits plane at point A



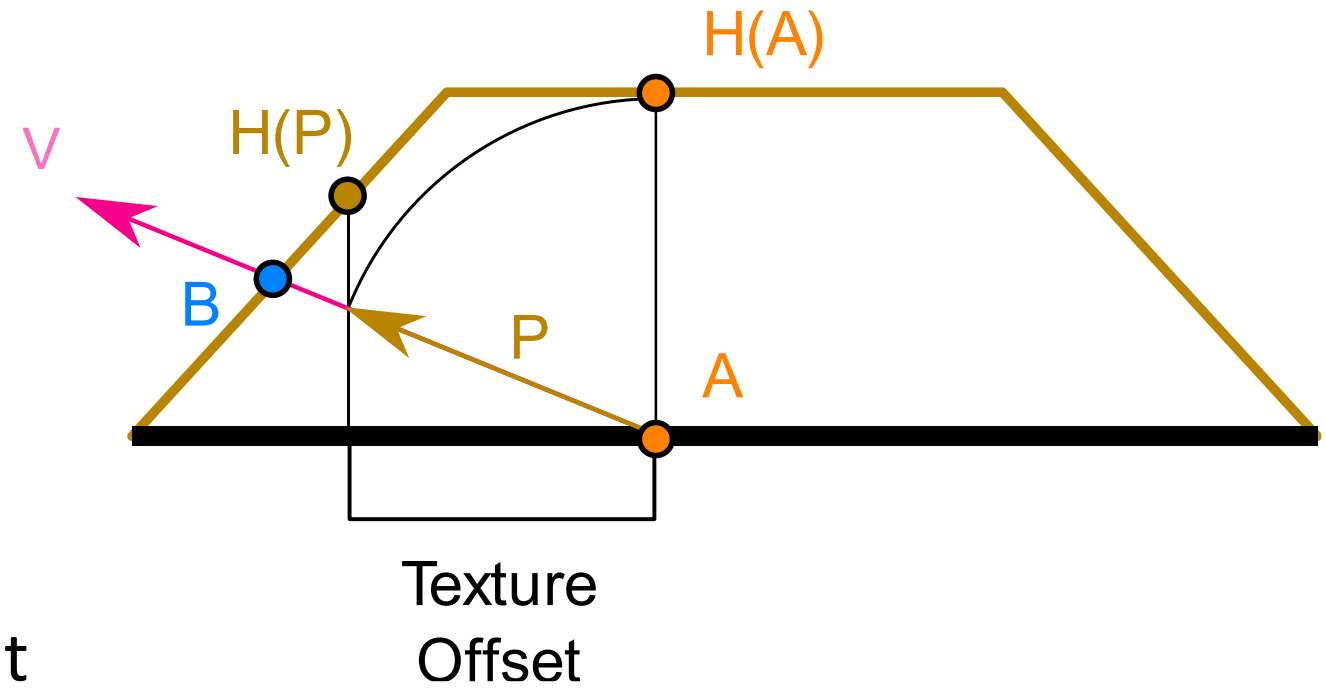
Introduction

- Parallax mapping aims to offset the texture coordinates at fragment position A such 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



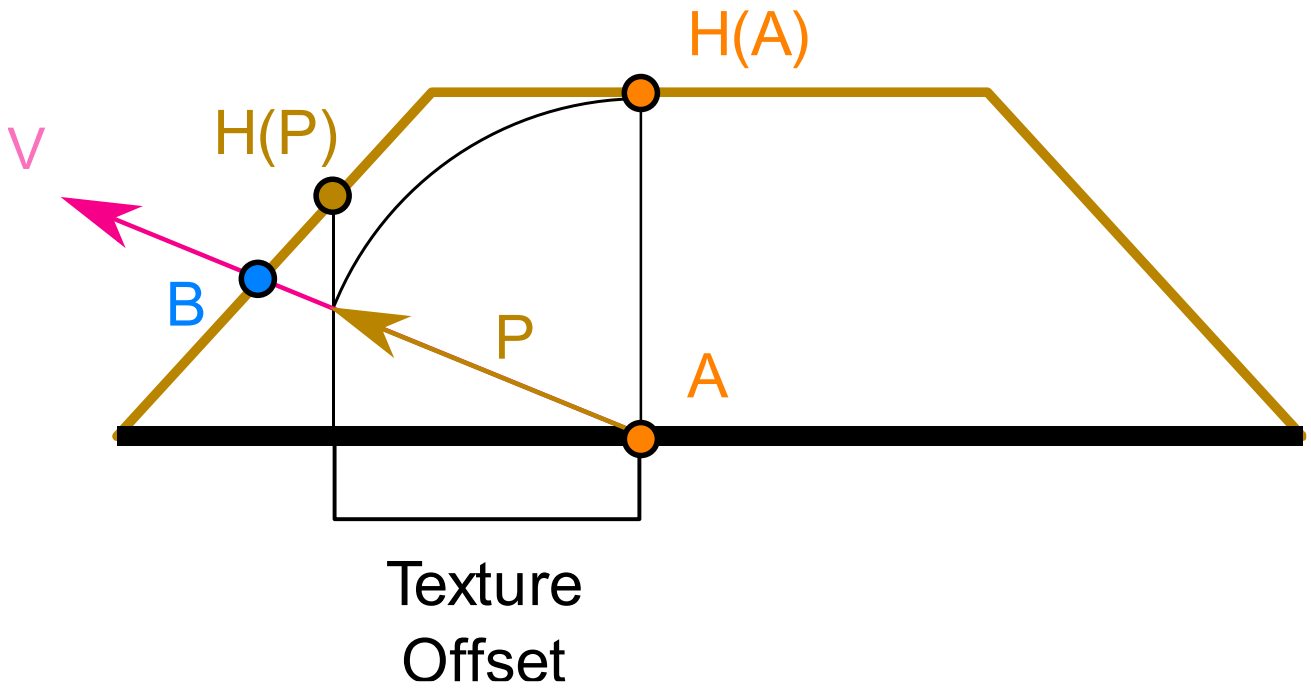
Introduction

- How to get texture coordinates at point B from point A?
- Parallax mapping scales the fragment-to-view direction vector V by the height at fragment A
- Scale the length of V to be equal to a sampled value from the heightmap H(A) at fragment position A



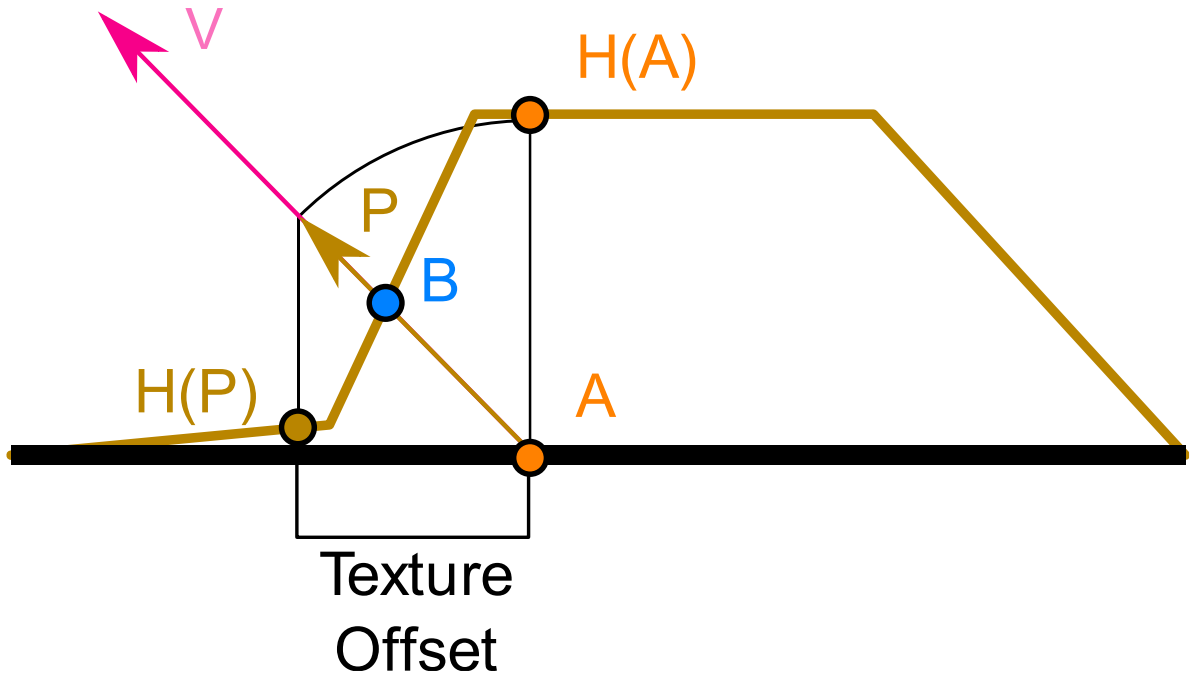
Introduction

- Take vector P and take its vector coordinates that align with the plane as the texture offset
- Works because vector P is calculated using a height value from the heightmap → the higher a fragment's height, the more it effectively gets displaced



Introduction

- This gives good results most of the time, but it is a crude approximation to get B
- When heights change rapidly the results tend to look unrealistic (P will not end up close to B):



Introduction

- Another issue: difficult to figure out which coordinates to retrieve from P when the surface is arbitrarily rotated
- Parallax mapping in a different coordinate space where the x and y component of vector P always aligns with the texture's surface
- → Parallax mapping in tangent space

Introduction

- Transforming the fragment-to-view direction V to tangent space, the transformed P's x and y component aligned to the surface's tangent and bitangent vectors
- Tangent and bitangent vectors are pointing in same direction as surface's texture coordinates, can take x and y components of P as the texture coordinate offset (regardless of the surface's direction)

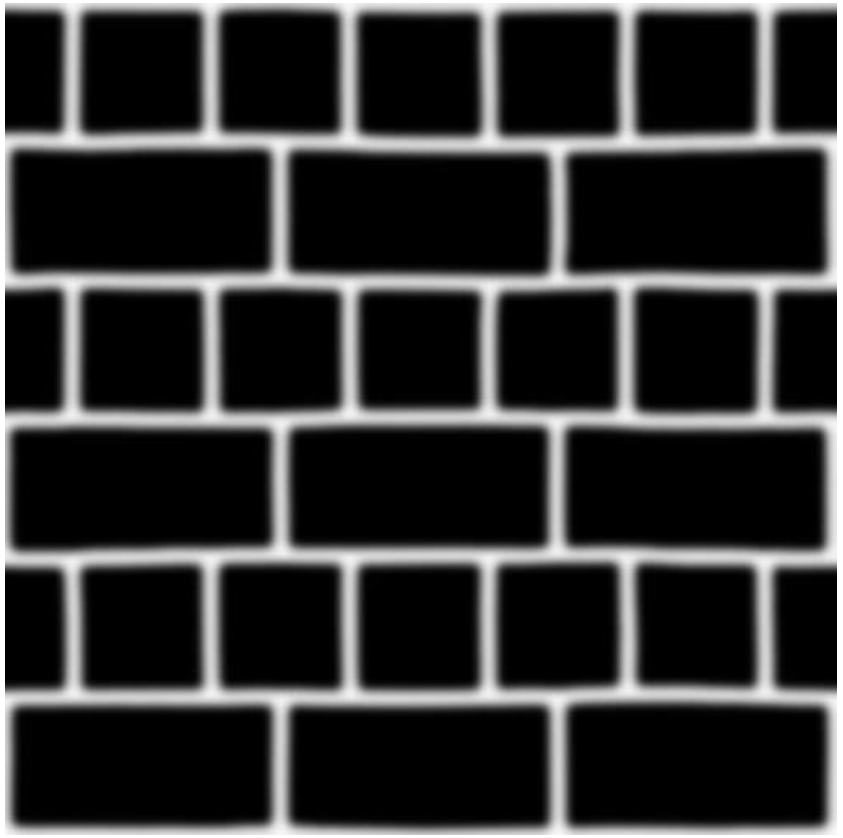
Parallax Mapping

Parallax Mapping

- Use a simple 2D plane and calculate its tangent and bitangent vectors (before sending it to the GPU, see normal mapping lecture)
- Attach a diffuse texture, a normal map and a displacement map on the plane
- Here, use parallax mapping with additional normal mapping (displacement illusion breaks when the lighting does not match)
- Normal maps are often generated from heightmaps, using a normal map together with the heightmap makes sure the lighting is in place with the displacement

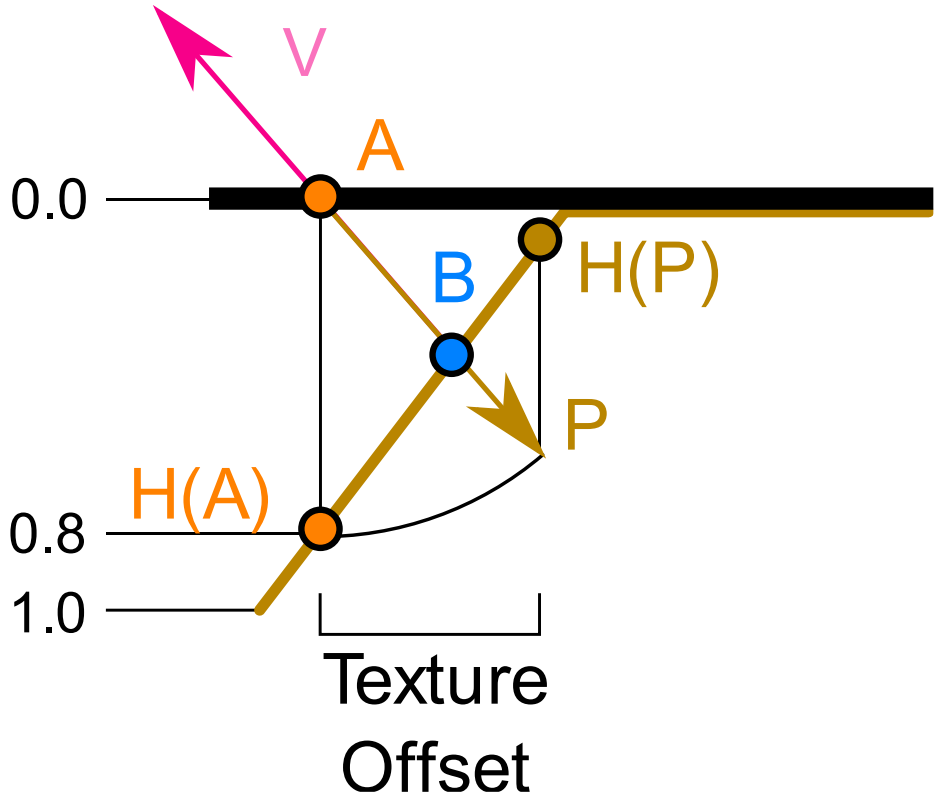
Parallax Mapping

- Actually, displacement map is the inverse of the heightmap shown at the beginning
- Parallax mapping makes more sense to use the inverse of the heightmap (also known as a depthmap)
- Easier to fake depth than height on flat surfaces



Parallax Mapping

- This changes how we perceive parallax mapping:
- Given points A and B, but now obtain P by subtracting V from texture coordinates at A
- Obtain depth values instead of height values (subtract heightmap values from 1.0), or inverse texture values in image-editing software



Parallax Mapping

- Parallax mapping implemented in fragment shader (displacement effect differs all over a triangle's surface)
- Need to calculate the fragment-to-view direction vector V → need the view position and a fragment position in tangent space
- Copy of that normal mapping vertex shader:

```
#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;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;
```

Parallax Mapping

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

uniform vec3 lightPos;
uniform vec3 viewPos;

void main()
{
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.TexCoords = aTexCoords;

    vec3 T = normalize(mat3(model) * aTangent);
    vec3 B = normalize(mat3(model) * aBitangent);
    vec3 N = normalize(mat3(model) * aNormal);
    mat3 TBN = transpose(mat3(T, B, N));

    vs_out.TangentLightPos = TBN * lightPos;
    vs_out.TangentViewPos  = TBN * viewPos;
    vs_out.TangentFragPos  = TBN * vs_out.FragPos;

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Parallax Mapping

- Within the fragment shader, implement the parallax mapping logic:

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} fs_in;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D depthMap;

uniform float heightScale;
```

Parallax Mapping

- Within the fragment shader, implement the parallax mapping logic:

```
…

vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir);

void main()
{
    vec3 viewDir = normalize(fs_in.TangentViewPos - fs_in.TangentFragPos);
    vec2 texCoords = fs_in.TexCoords;

    vec3 normal = texture(normalMap, texCoords).rgb;
    normal = normalize(normal * 2.0 - 1.0);
    vec3 diffuse = texture(diffuseMap, texCoords).rgb;
    …
}
```

Parallax Mapping

- The ParallaxMapping function:

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
   float height = texture(depthMap, texCoords).r;
   vec2 p = viewDir.xy / viewDir.z * (height * heightScale);
   return texCoords - p;
}
```

Parallax Mapping

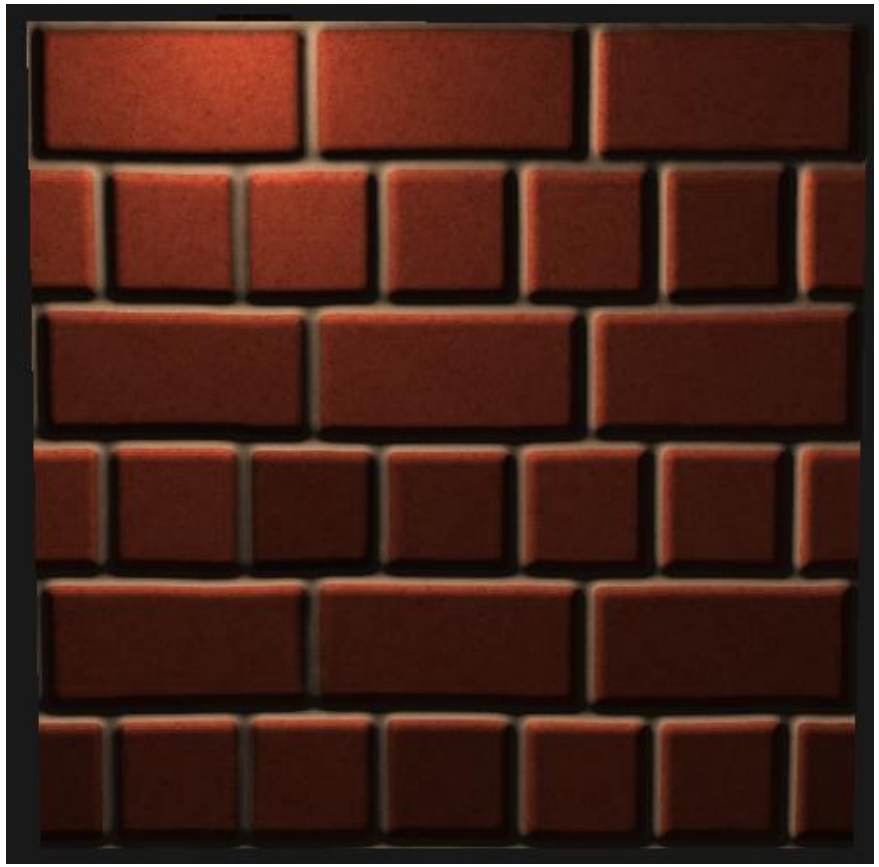
- Note, division of viewDir.xy by viewDir.z: viewDir vector is normalized → viewDir.z in $[0,1]$
- When viewDir is largely parallel to the surface → z close to 0.0 (division returns larger vector P compared to when viewDir is largely perpendicular to the surface)
- So increasing P that it offsets the texture coordinates at a larger scale when looking at a surface from an angle compared to when looking at it from the top → more realistic results at angles

Parallax Mapping

- Some prefer to leave the division out (normal Parallax Mapping could produce undesirable results at angles)
- Technique is then called Parallax Mapping with Offset Limiting
- Choosing which technique to pick is usually a matter of personal preference

F5…

- … bumpy (left heightScale=0, right 0.1)



Parallax Mapping

- Border artifacts at the edge of the parallax mapped plane
- Edges of the plane, the displaced texture coordinates oversample outside [0, 1] giveing unrealistic results based on the texture's wrapping mode(s)
- Discard the fragment whenever it samples outside the default texture coordinate range:

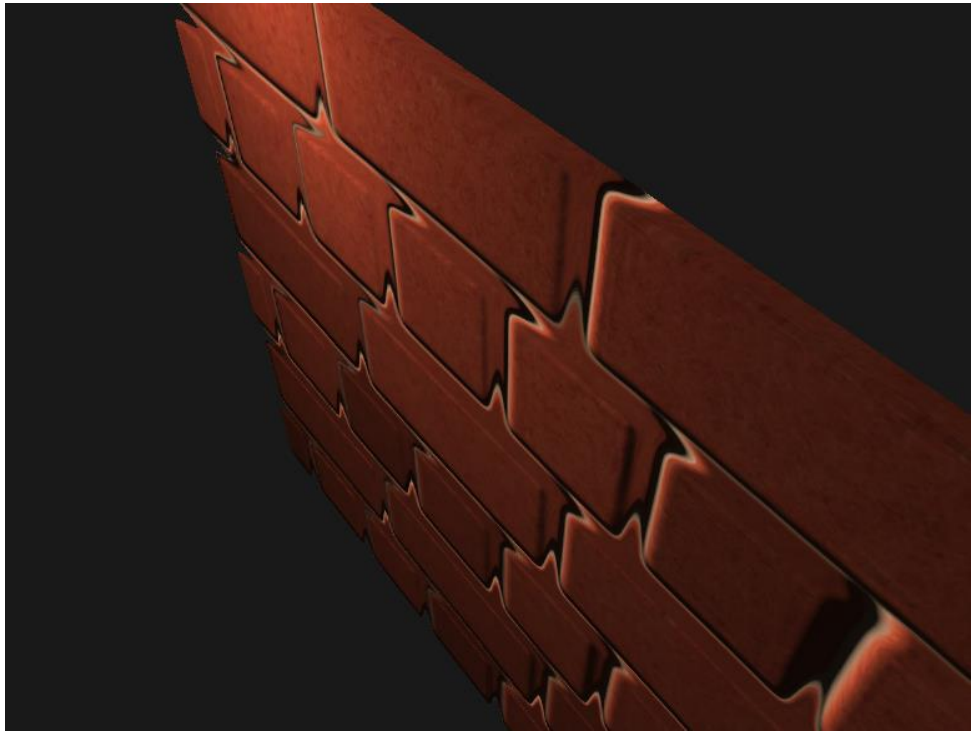
```
texCoords = ParallaxMapping(fs_in.TexCoords,  viewDir);
if(texCoords.x > 1.0 || texCoords.y > 1.0 || texCoords.x < 0.0 || texCoords.y
                                                                  < 0.0)
        discard;
```

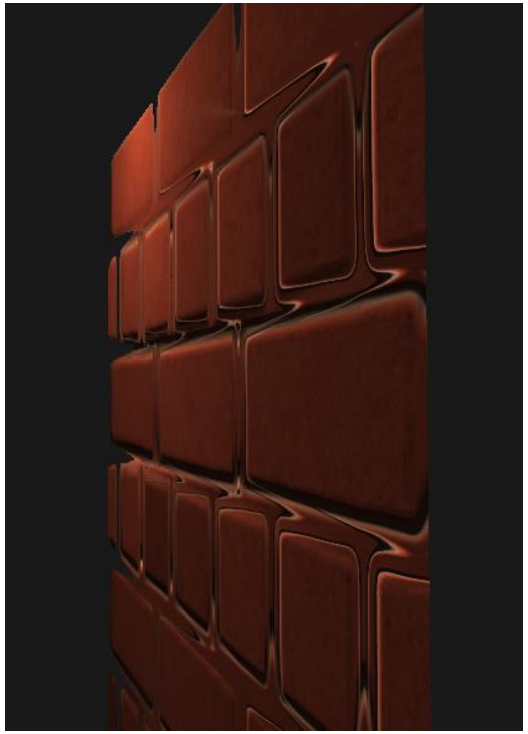
F5…

- …better (left old, right new)

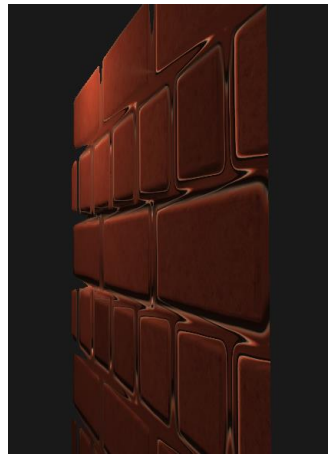
Note

- Looks great, but breaks down when looking at it from an angle (similar to normal mapping) and gives incorrect results with steep height changes:





Note



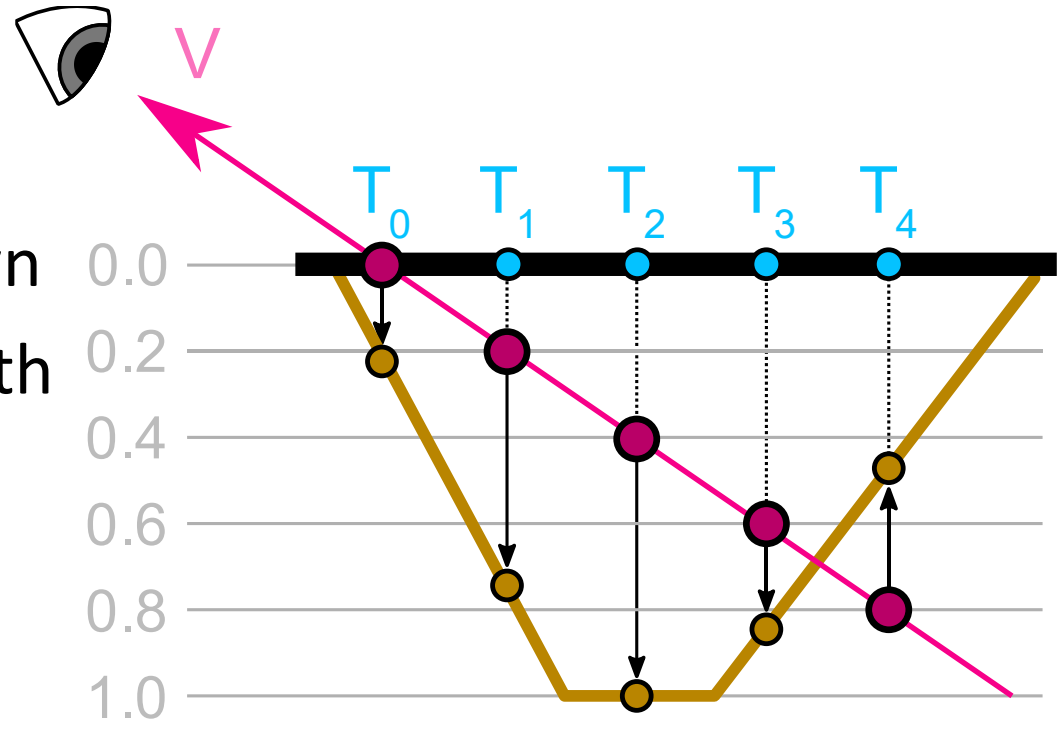
- Reason: just a crude approximation of displacement mapping
- Trick: steep height changes (works even when looking at an angle)
- E.g., what if we instead of one sample, take multiple samples to find the closest point to B?

Steep Parallax Mapping

Introduction

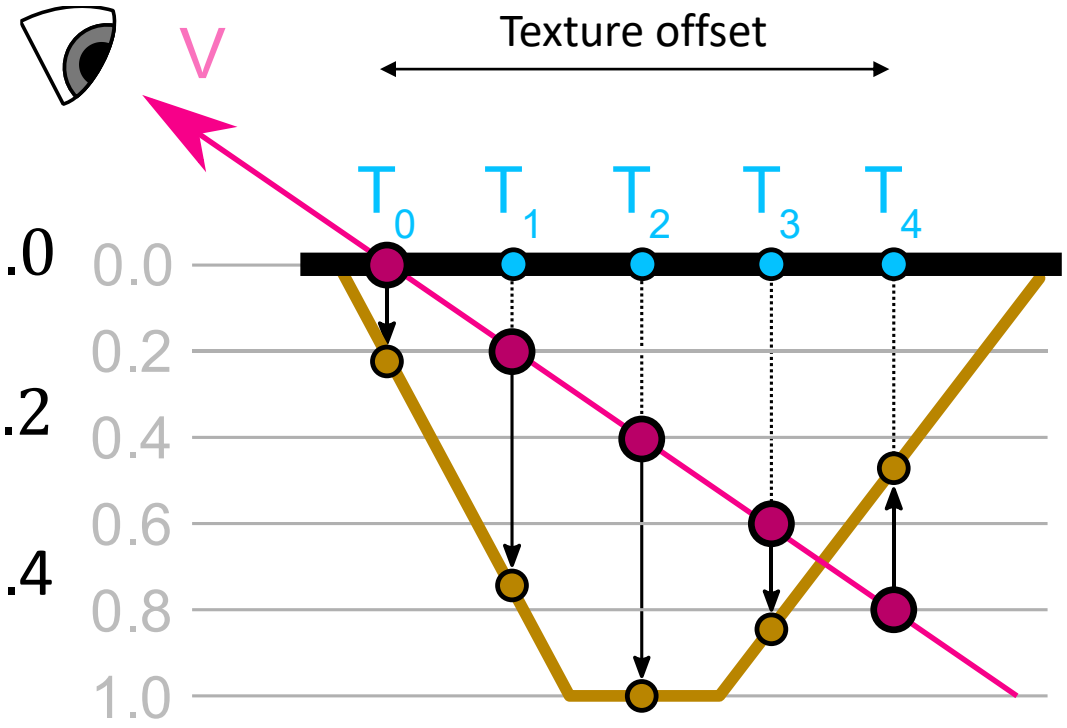
- Steep Parallax Mapping an extension on top of Parallax Mapping
- Uses same principles, but instead of 1 sample it takes multiple samples to better pinpoint vector P to B
- Better results, even with steep height changes as the accuracy of the technique is improved by the number of samples
- Steep Parallax Mapping divides total depth range into multiple layers of the same height/depth
- Each layer sample the depthmap shifting the texture coordinates along the direction of P until we find a sampled depth value that is below the depth value of the current layer

Steep Parallax Mapping



- Traverse the depth layers from top down
- Each layer compare depth value to depth value stored in depthmap
- If layer's depth value is less than the depthmap's value → this layer's part of vector P is not below the surface
- Continue until layer's depth is higher than the value stored in the depthmap: this point is then below the (displaced) geometric surface

Steep Parallax Mapping



- Depthmap value at T_0: $D(T_0) = 0.22 > 0.0$
  → continue
- Depthmap value at T_1: $D(T_1) = 0.75 > 0.2$
  → continue
- Depthmap value at T_2: $D(T_2) = 1.00 > 0.4$
  → continue
  …
- Depthmap value at T_4: $D(T_4) = 0.47 < 0.8$
  → Vector P at T_4 most viable position of the displaced geometry
- Take the texture coordinate offset to displace the fragment's texture coordinates

Steep Parallax Mapping

- To implement this, change the ParallaxMapping function:

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    // number of depth layers
    const float numLayers = 10;
    // calculate the size of each layer
    float layerDepth = 1.0 / numLayers;
    // depth of current layer
    float currentLayerDepth = 0.0;
    // the amount to shift the texture coordinates per layer (from vector P)
    vec2 P = viewDir.xy * height_scale;
    vec2 deltaTexCoords = P / numLayers;

    [...]
}
```

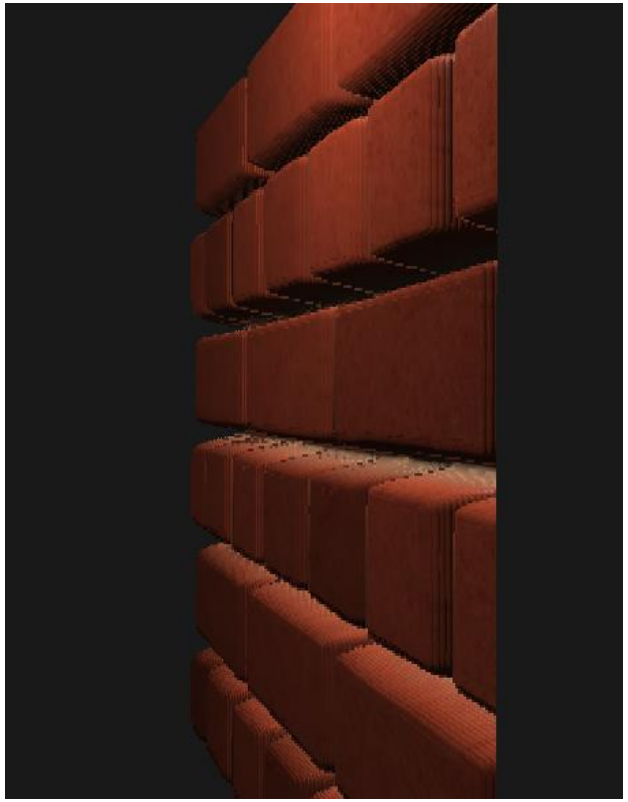
Steep Parallax Mapping

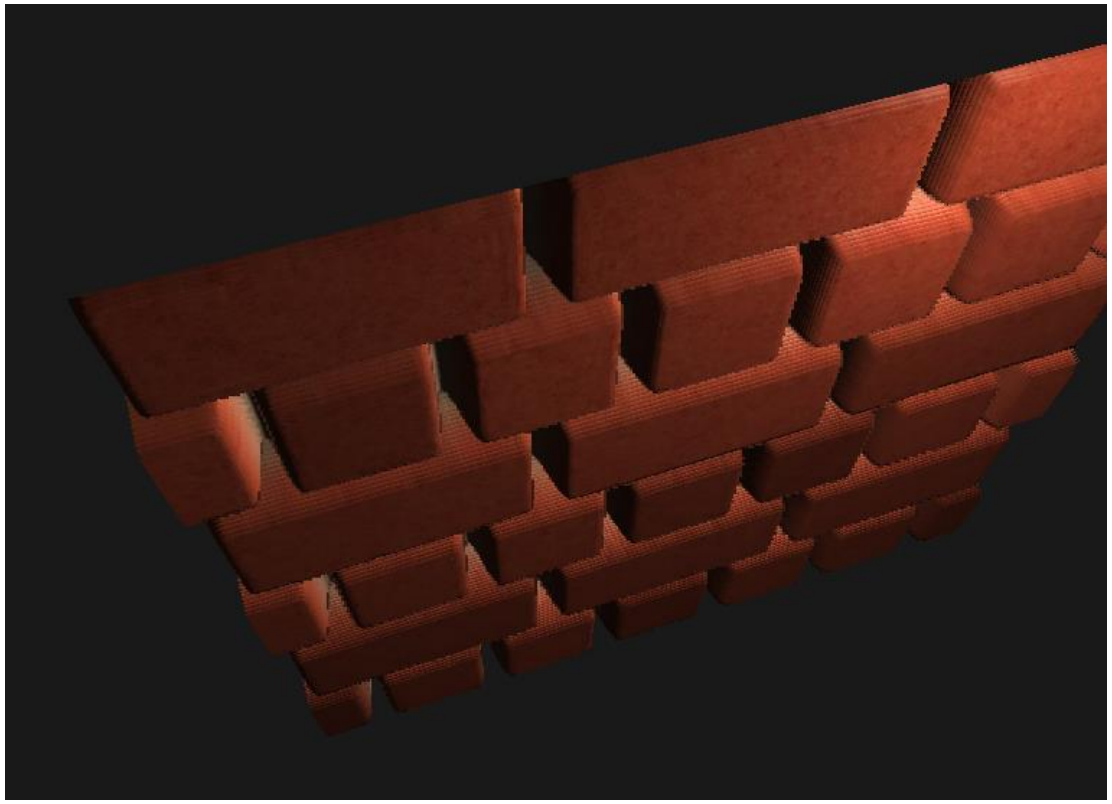
- Then iterate through all the layers, starting from the top, until we find a depthmap value less than the layer's depth value:

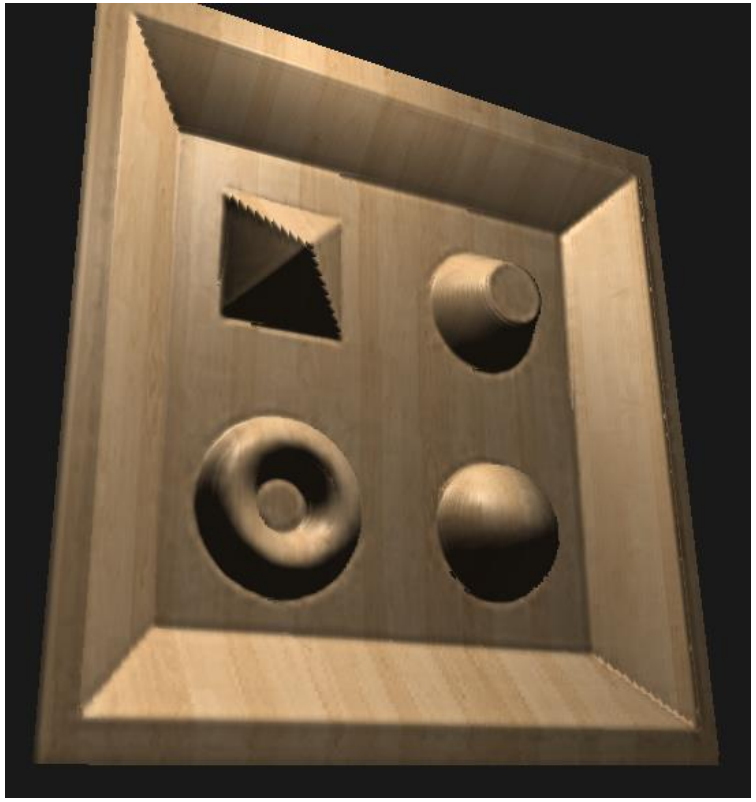
```
// get initial values
vec2  currentTexCoords     = texCoords;
float currentDepthMapValue = texture(depthMap, currentTexCoords).r;

while(currentLayerDepth < currentDepthMapValue)
{
        // shift texture coordinates along direction of P
        currentTexCoords -= deltaTexCoords;
        // get depthmap value at current texture coordinates
        currentDepthMapValue = texture(depthMap, currentTexCoords).r;
        // get depth of next layer
        currentLayerDepth += layerDepth;
}
return currentTexCoords;
```

F5…

- … much better!







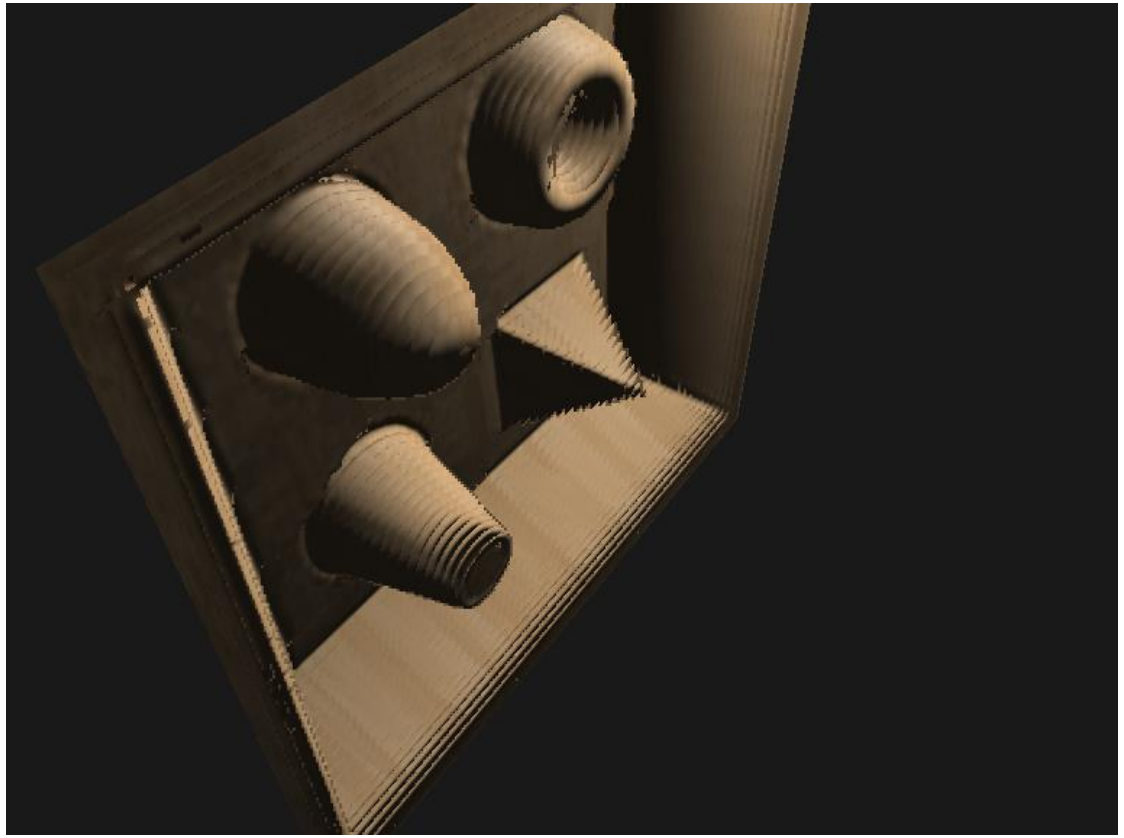
Steep Parallax Mapping

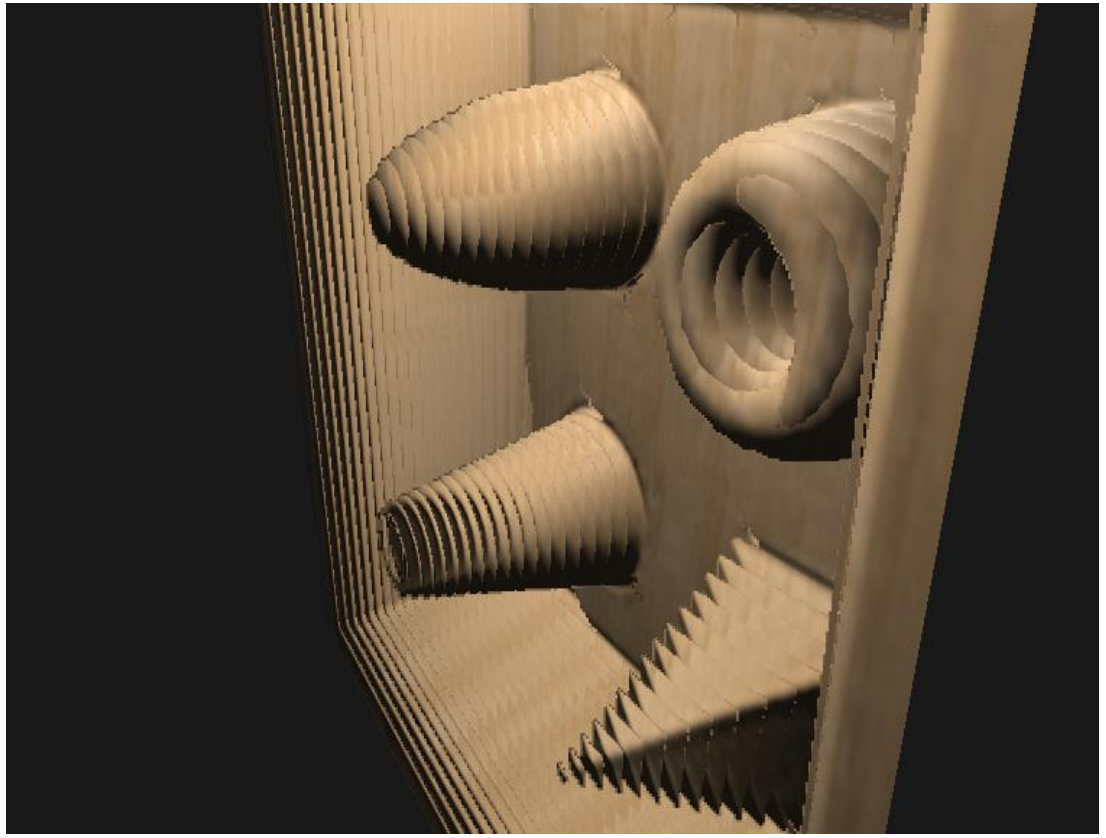
- Improve by exploiting one of Parallax Mapping's properties
- When looking straight onto a surface not much texture displacement while there is a lot when looking from an angle
- Take less samples when looking straight at a surface and more samples when looking at an angle (sample the necessary amount):

```
const float minLayers = 8;
const float maxLayers = 32;
float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0.0, 0.0, 1.0),
                                                    viewDir)));
```

Steep Parallax Mapping

- Problems: technique is based on a finite number of samples → aliasing effects distinctions between layers can easily be spotted:





Steep Parallax Mapping

- Reduce it by taking a larger number of samples → performance
- Several approaches to fix this by interpolating between the position's two closest depth layers to find a much closer match to B
- Popular approaches are called *Relief Parallax Mapping* and *Parallax Occlusion Mapping*
- Relief Parallax Mapping most accurate results, but is also more performance heavy
- Parallax Occlusion Mapping gives almost the same results and more efficient → preferred approach

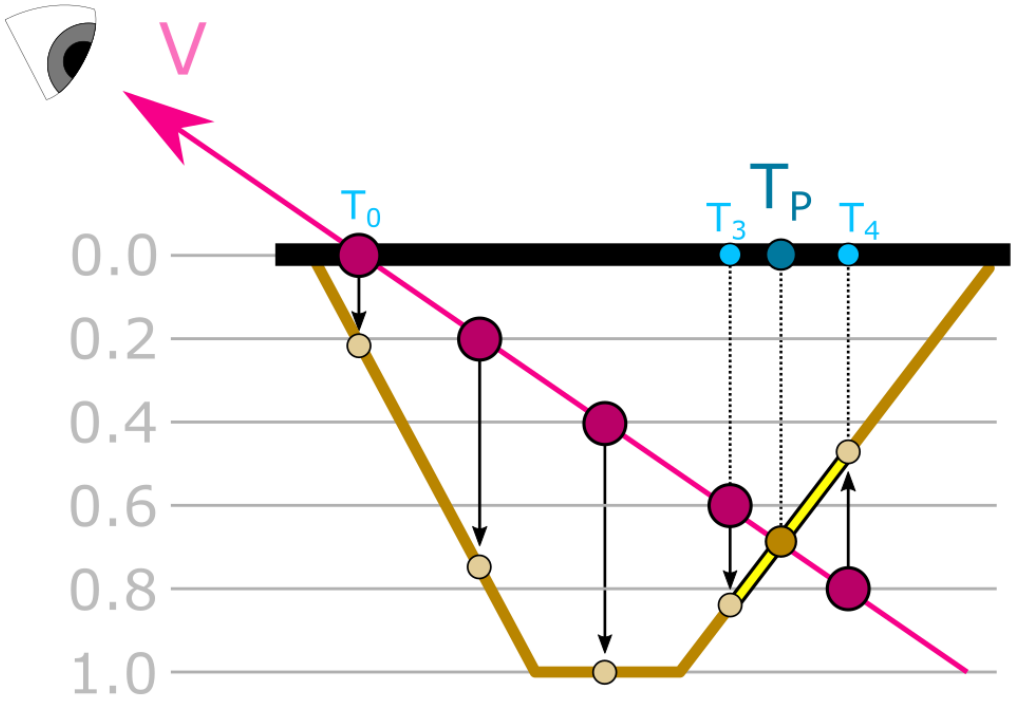
Parallax Occlusion Mapping

Introduction

- Parallax Occlusion Mapping based on same principles as Steep Parallax Mapping, but instead of taking the texture coordinates of the first depth layer after a collision going to linearly interpolate between the depth layer after and before the collision
- Base the weight of the linear interpolation on how far the surface's height is from the depth layer's value of both layers

Parallax Occlusion Mapping

- Take a look at the following picture:
- Similar to Steep Parallax Mapping with extra: linear interpolation between the two depth layers' texture coordinates surrounding the intersected point
- Again, an approximation, but more accurate



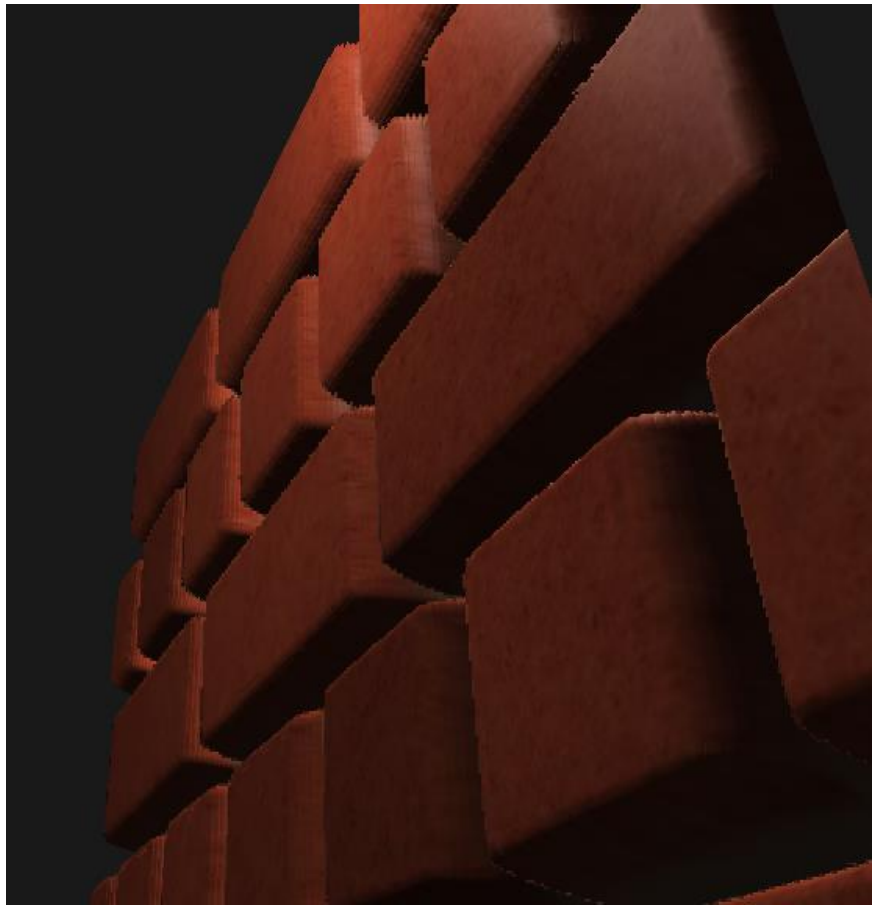
Parallax Occlusion Mapping

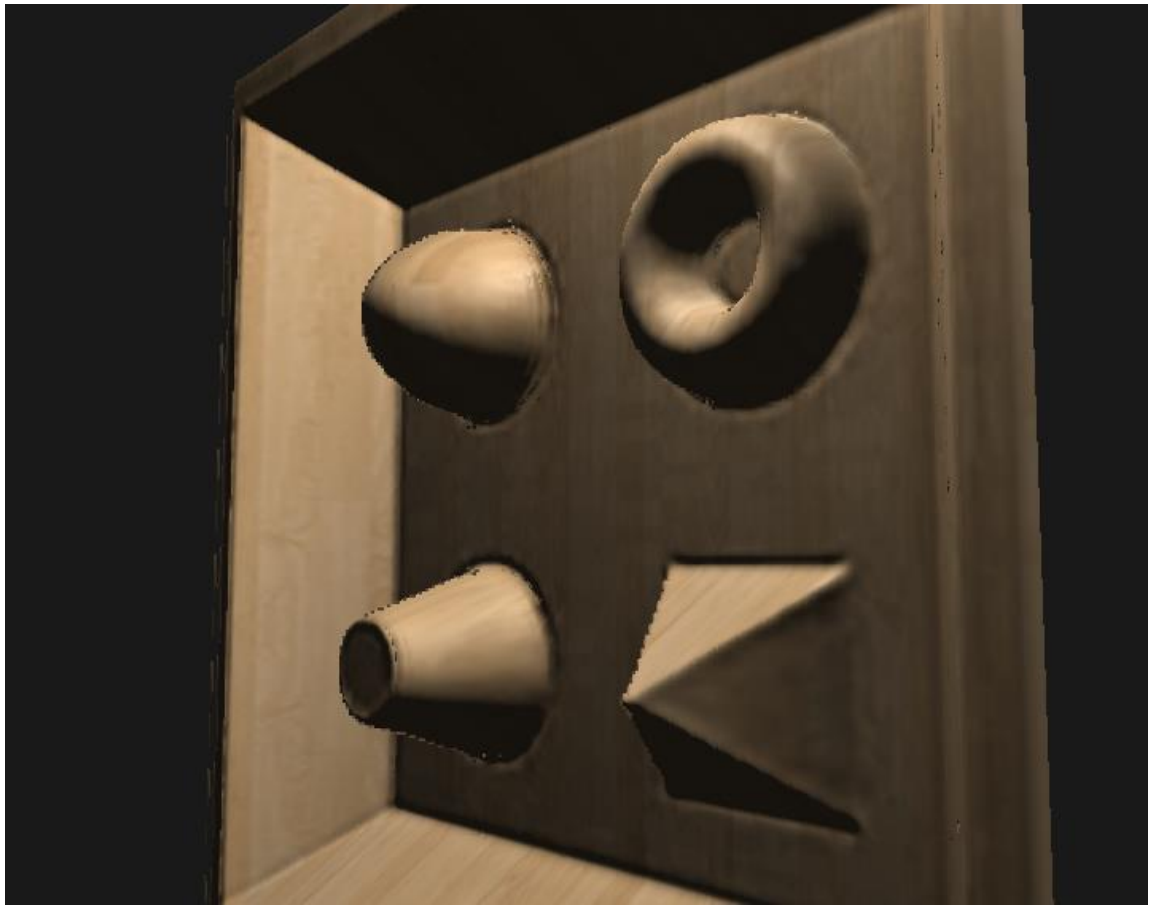
- Code for Parallax Occlusion Mapping is an extension on top of Steep Parallax Mapping:

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

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

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

// interpolation of texture coordinates
float weight = afterDepth / (afterDepth - beforeDepth);
vec2 finalTexCoords = prevTexCoords * weight + currentTexCoords * (1.0 - weight);

return finalTexCoords;
```

F5…

- … less aliasing!





Questions???