14. Shadow Mapping

Advanced
Game Graphic Programming

Prof. HyeongYeop Kang
siamiz@khu.ac.kr

Shadow

Shadows for a realistic scene.

- Virtually all scenes in the real world have shadows, and therefore shadow generation is an indispensable component in computer graphics.
- In addition, shadows help us understand the spatial relationships among objects in a scene.

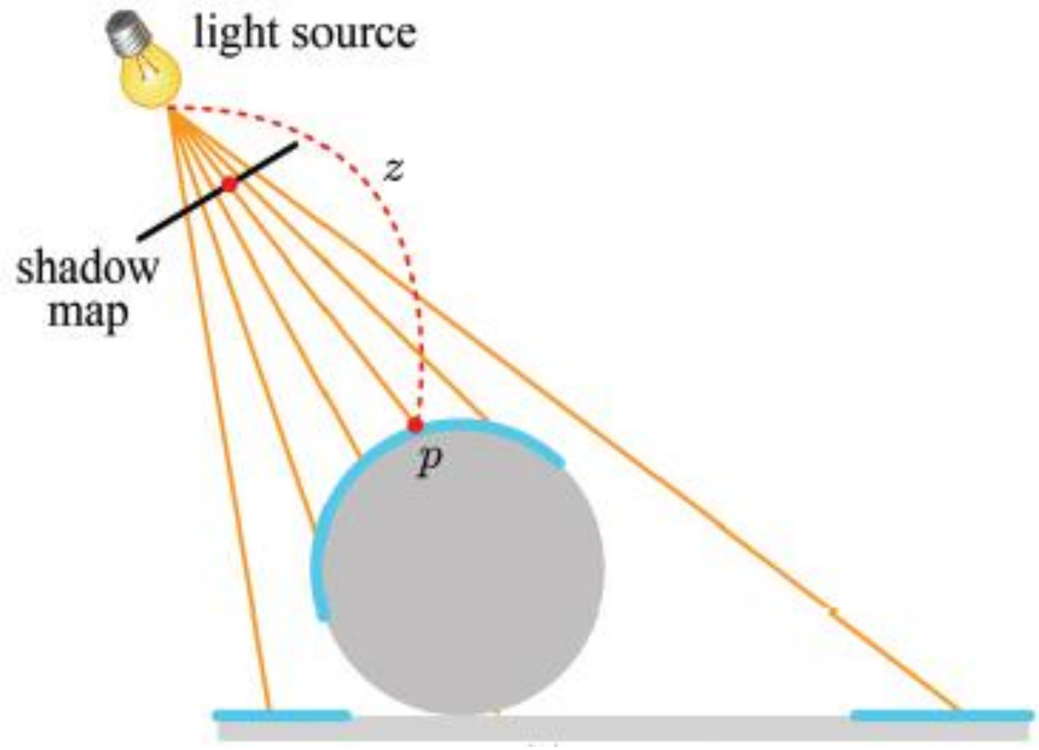
(a)

(b)

Shadow Mapping

Shadow mapping is achieved by two-pass algorithm.

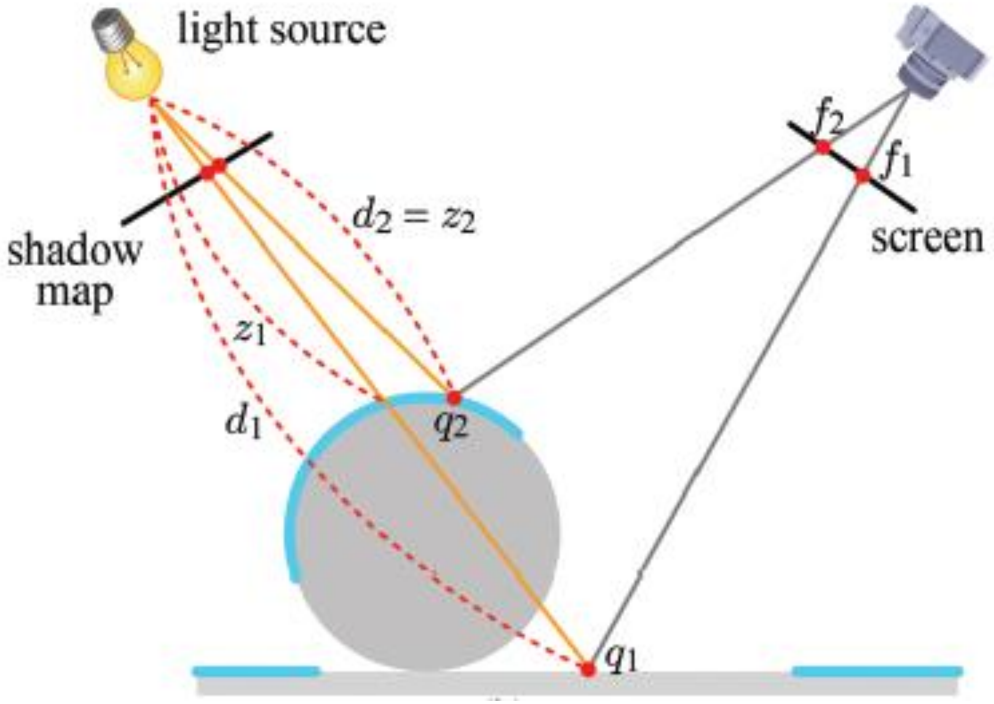
- Pass 1: not real rendering
  - Render the scene from the light source's viewpoint.
  - Store only the depths into the *shadow map*, which is a depth map with respect to the light source.



Shadow Mapping

Shadow mapping is achieved by two-pass algorithm.

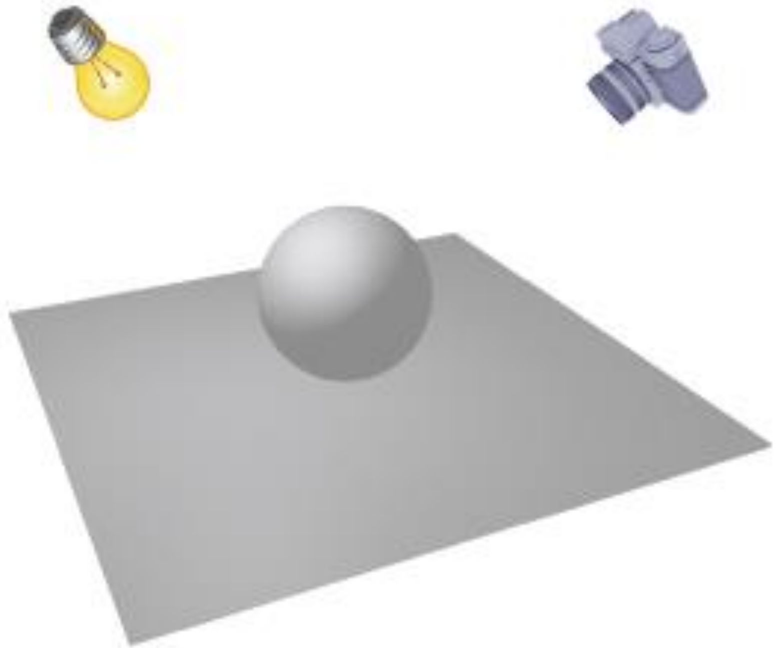
- Pass 2: real rendering
  - While rendering the scene from the camera position, compare each fragment's distance *d* to the light source with the depth *z* in the shadow map.
  - If $d > z$, the pixel is in shadows.
  - Otherwise, the pixel is lit.

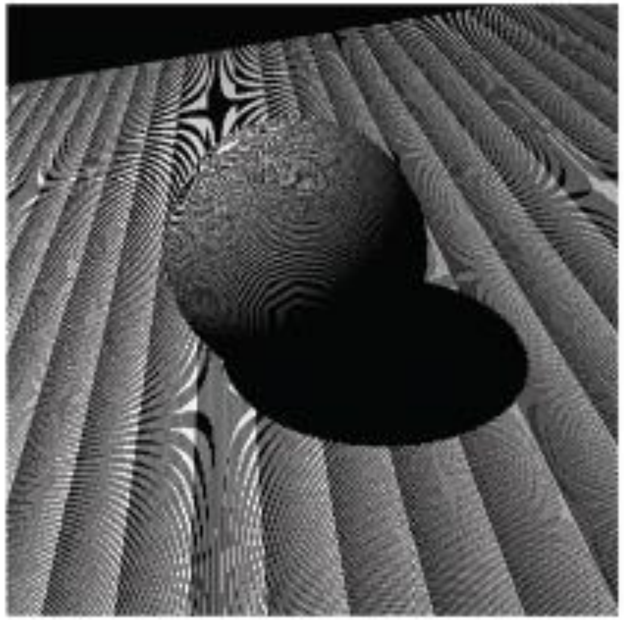


Shadow Mapping

Acne artifact

- A brute-force implementation of shadow mapping suffers from *the surface acne* artifact. The entire scene is decomposed into a number of fractional areas: some are fully lit whereas others are shadowed.
- The shadowed and lit pixels coexist on a surface area that should be entirely lit.

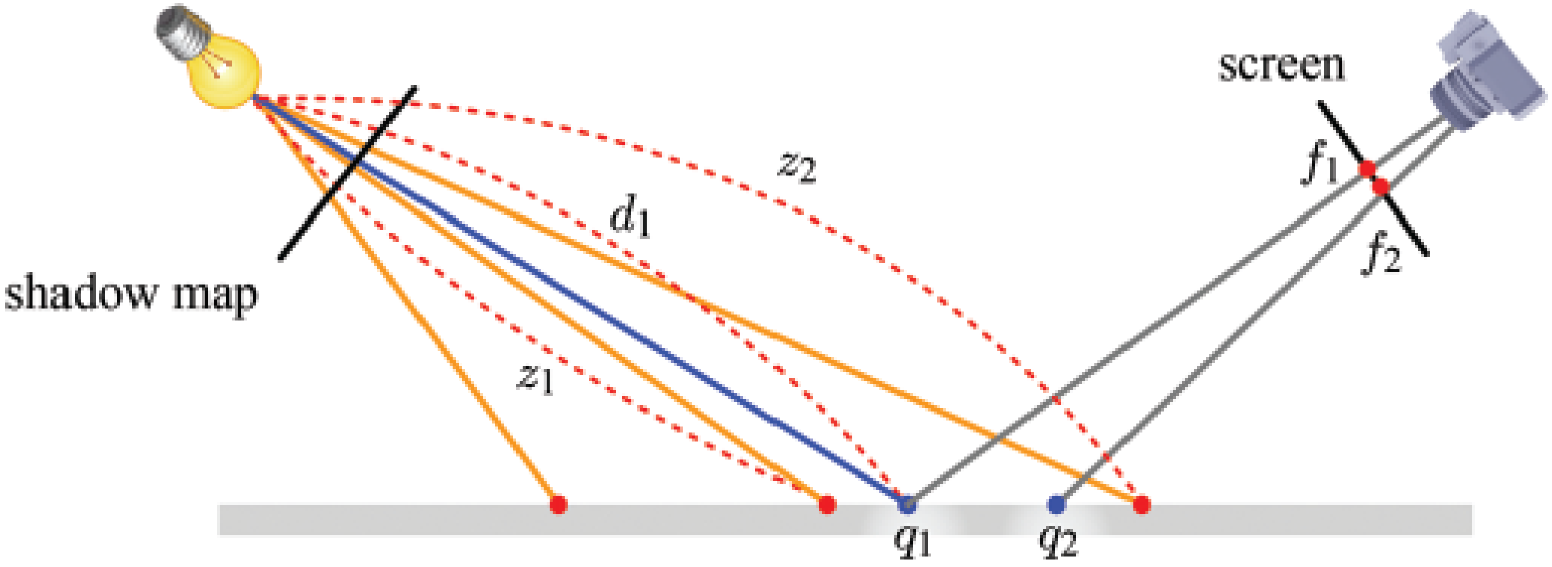




Shadow Mapping

The problem is..

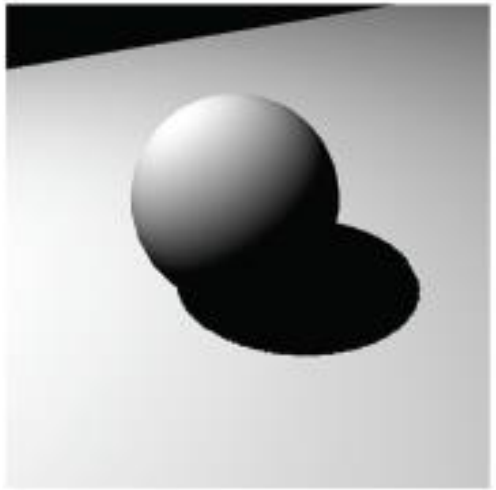
- Note that the scene points sampled at the 2nd pass are usually different from the scene points sampled at the 1st pass.
- Suppose the *nearest point sampling*.
- In the example, q_1 is to be lit, but judged to be in shadows because $d_1 > z_1$. On the other hand, q_2 is to be fully lit. Such a coexistence of shadowed and fully lit pixels leads to the surface acne artifact.



Shadow Mapping

A simple solution to this artifact:

- The bias value is usually fixed after a few trials.



biased shadow mapping

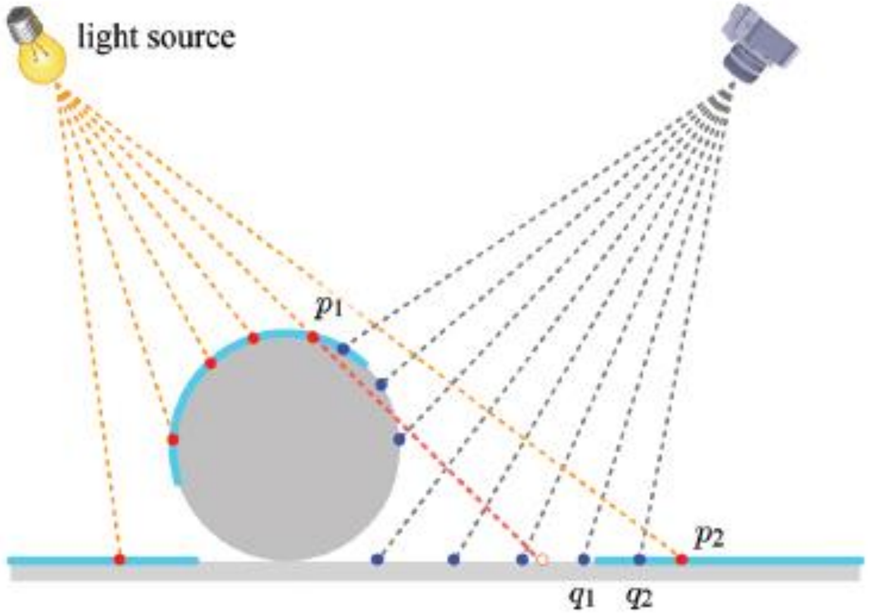
too small a bias

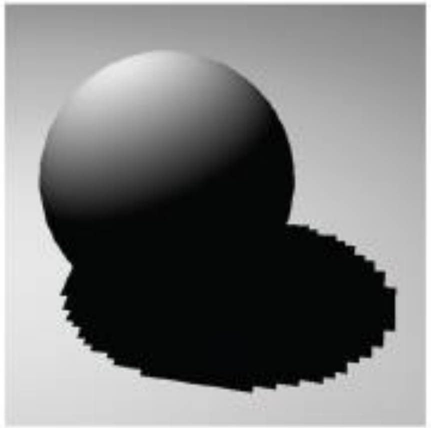
too large a bias

Shadow Map Filtering

Blocky shadow artifact

- If the resolution of a shadow map is not high enough, multiple pixels may be mapped to a single texel of the shadow map.
- This is a *magnification* case. Suppose you choose the nearest point sampling.
- A fragment can be either *fully* lit or shadowed. It has no other option. Then, the shadow boundary usually has a jagged appearance. This is an aliasing artifact.

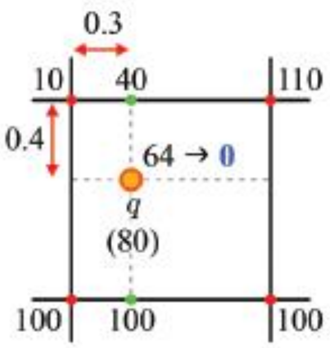


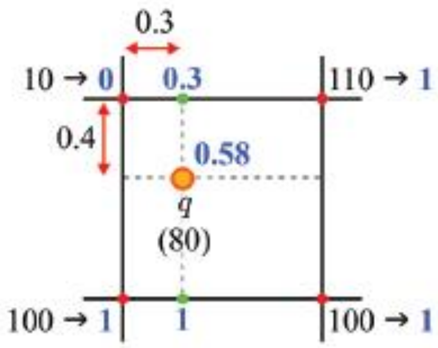


Shadow Map Filtering

Solution to a blocky shadow artifact

- Unfortunately, bilinear interpolation doesn’t help: A pixel is either fully lit or fully shadowed and consequently the shadow quality is not improved at all by choosing bilinear interpolation.
- A solution to this problem is to first determine the *visibilities* of a pixel with respect to the four texels, and then interpolate the visibilities. This value is taken as the “degree of being lit.”
- The technique of taking multiple texels from the shadow map and blending the pixel's visibilities against the texels is named *percentage closer filtering* (PCF).

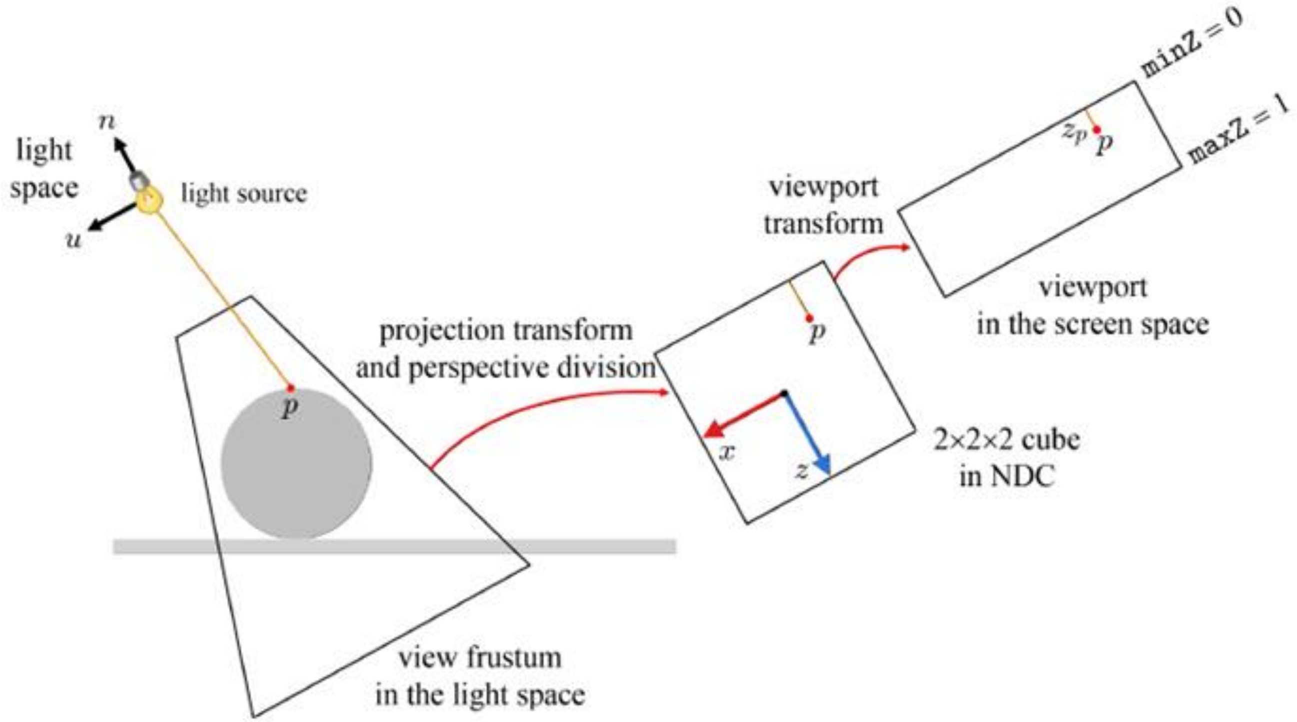




Shadow Mapping

First-pass Implementation

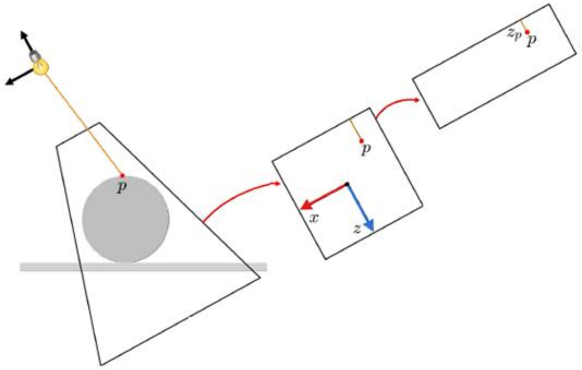
- Let's see the *first-pass* shaders
- The view matrix is defined with **EYE** placed at the light source position. It transforms the world-space vertex into a space, named the *light space*.

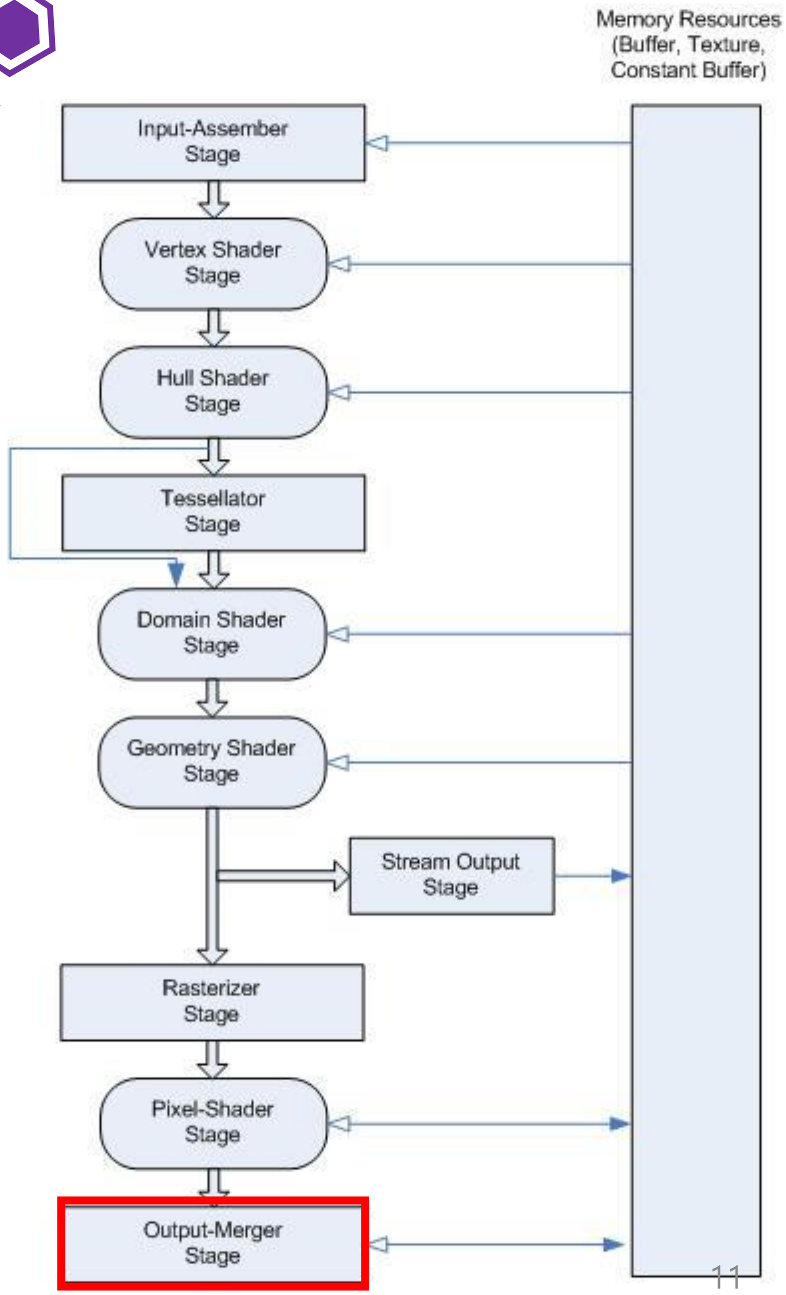


Shadow Mapping

First-pass Implementation

- Note that there are no attributes to interpolate because the vertex shader outputs nothing except the output position. So, nothing to interpolate.
- Not surprisingly, the pixel shader's main function is empty.
- The color of a pixel is not returned, but its screen-space coordinates are passed to the Output-Merger stage so that it goes through *z-buffering*. The z-buffer is updated so as to contain the depth values of the surfaces *visible* from the light source. It is the *shadow map*.





Render-to-Texture

Then the depth map is obtained in the form of a texture. We call this render-to-texture.

```
RenderTextureClass* m_RenderTexture;
m_RenderTexture = new RenderTextureClass;

// Initialize the render to texture object.
result = m_RenderTexture->Initialize(m_D3D->GetDevice(), SHADOWMAP_WIDTH, SHADOWMAP_HEIGHT, SCREEN_DEPTH,
SCREEN_NEAR);

// Set the render target to be the render to texture.
m_RenderTexture->SetRenderTarget(m_D3D->GetDeviceContext());

// Clear the render to texture.
m_RenderTexture->ClearRenderTarget(m_D3D->GetDeviceContext(), 0.0f, 0.0f, 0.0f, 1.0f);
```

Shadow Mapping

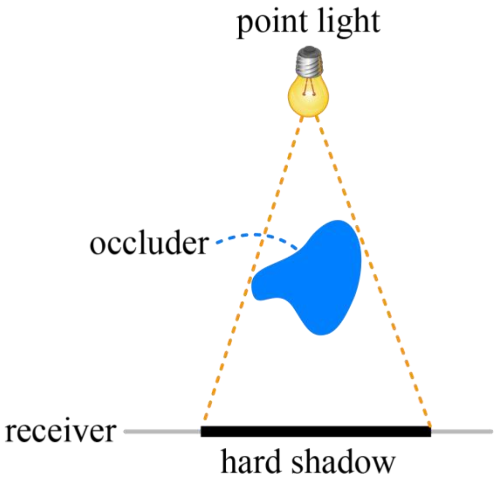
Second-pass Implementation

```
float4 PixelShader(PixelInputType input) : SV_TARGET{
  bias = 0.001f; // Set the bias value for fixing the floating point precision issues.
   // Calculate the projected texture coordinates.
   projectTexCoord.x =  input.lightViewPosition.x / input.lightViewPosition.w / 2.0f + 0.5f;
   projectTexCoord.y = -input.lightViewPosition.y / input.lightViewPosition.w / 2.0f + 0.5f;

    // Determine if the projected coordinates are in the 0 to 1 range.  If so then this pixel is in the view of the light. Saturate() clamps the value within the range of 0 to 1.
    if((saturate(projectTexCoord.x) == projectTexCoord.x) && (saturate(projectTexCoord.y) == projectTexCoord.y))  {
        // Sample the shadow map depth value from the depth texture using the sampler at the projected texture coordinate location.
        depthValue = depthMapTexture.Sample(SampleTypeClamp, projectTexCoord).r;
        lightDepthValue = input.lightViewPosition.z / input.lightViewPosition.w; // Calculate the depth of the light.
        lightDepthValue = lightDepthValue - bias; // Subtract the bias from the lightDepthValue.
        // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
        // If the light is in front of the object then light the pixel, if not then shadow this pixel since an object (occluder) is casting a shadow on it.
        if(lightDepthValue < depthValue){
            // Calculate the amount of light on this pixel.
            lightIntensity = saturate(dot(input.normal, input.lightPos));
            if(lightIntensity > 0.0f)          {
                // Determine the final diffuse color based on the diffuse color and the amount of light intensity.
                color += (diffuseColor * lightIntensity);
                color = saturate(color); // Saturate the final light color.
            }
        }
    }
    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor = shaderTexture.Sample(SampleTypeWrap, input.tex);
    color = color * textureColor; // Combine the light and texture color.
    return color;
}
```

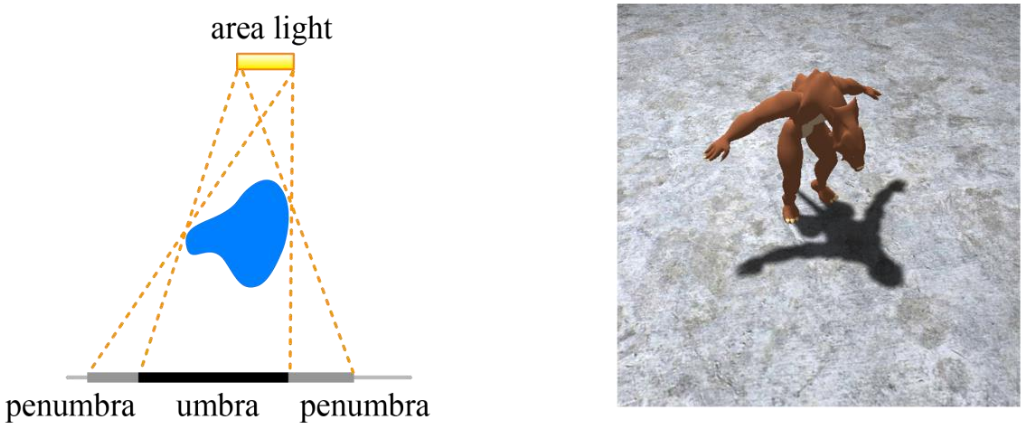
Hard Shadow vs. Soft Shadow

A point light source generates hard shadows.

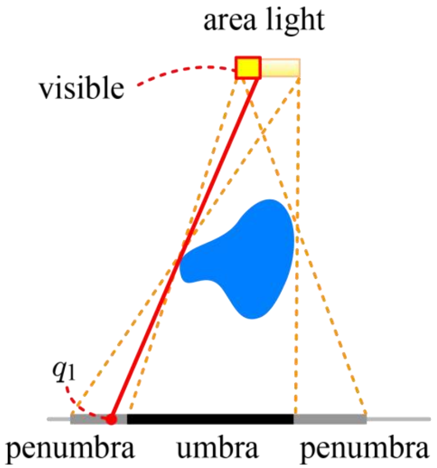


Hard Shadow vs. Soft Shadow

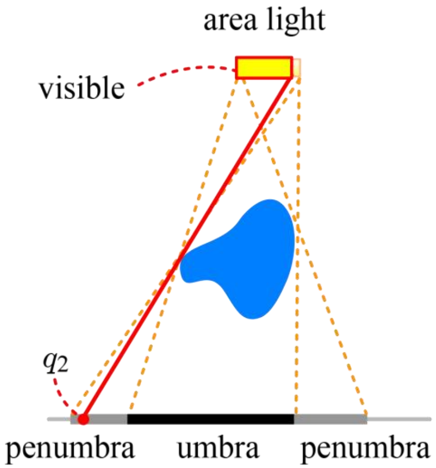
An area or volumetric light source generates soft shadows.



(a)



(b)



(c)

Research directions

Improved anti-aliasing for Euclidean distance transform shadow mapping (EDTSM), 2017[1].

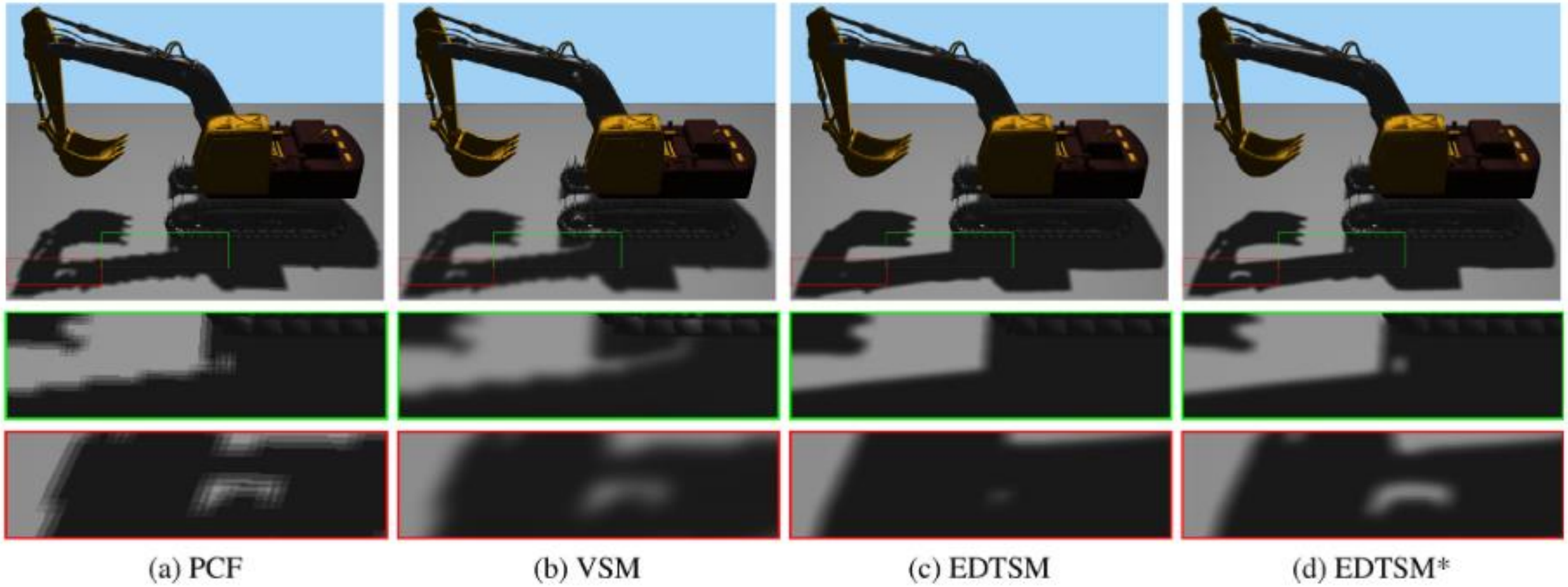
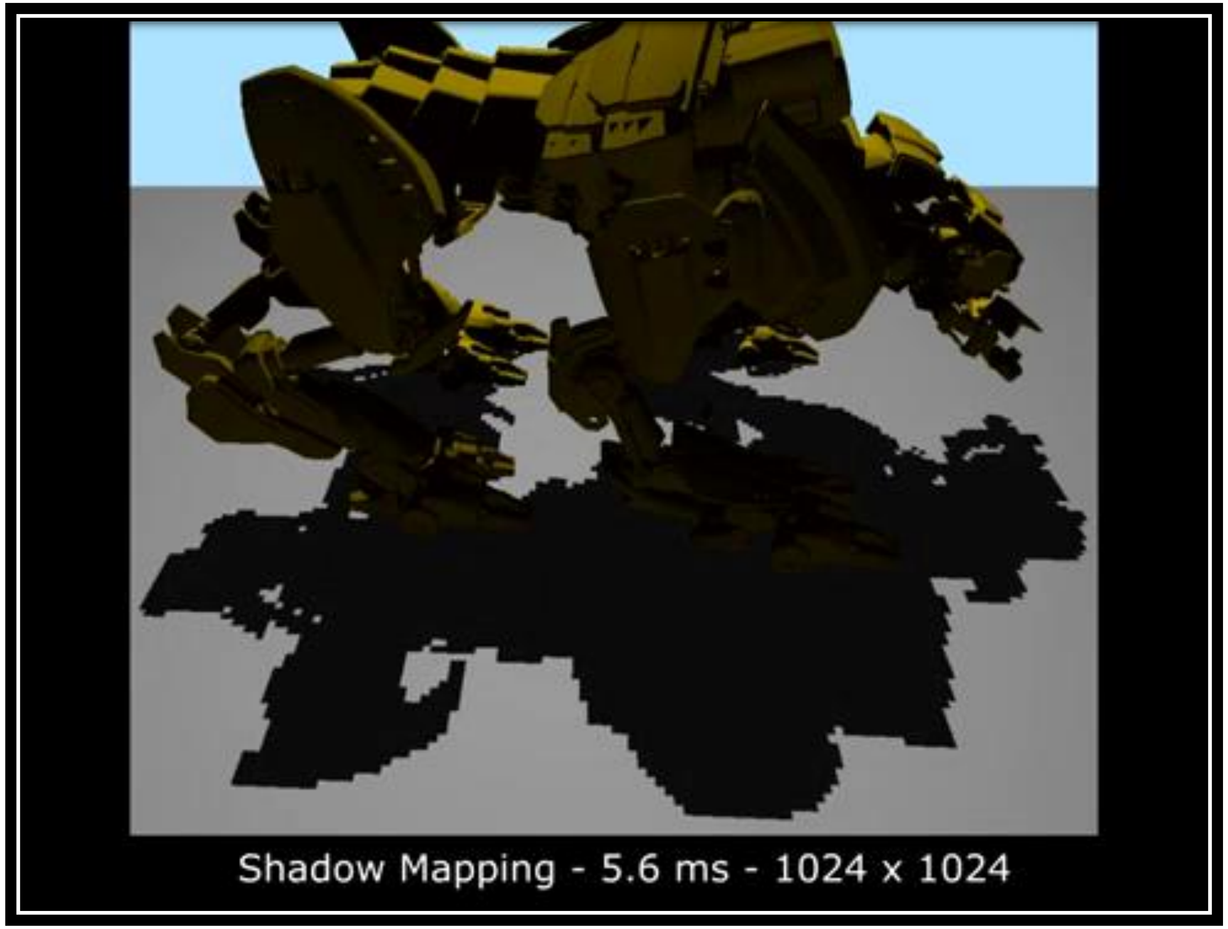


Fig. 1. Fixed-size penumbra produced by different techniques. For a low-order filter size, shadow map filtering techniques, such as PCF, generate shadows with aliasing and banding artifacts (a). Shadow map pre-filtering techniques, such as VSM, are prone to light leaking artifacts (green closeup in (b)). EDTSM suffers from shadow overestimation artifacts (c). The proposed approach (here named EDTSM*) is able to minimize those artifacts efficiently (d). Images were generated for the Excavator model using a 512^2 shadow map resolution. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

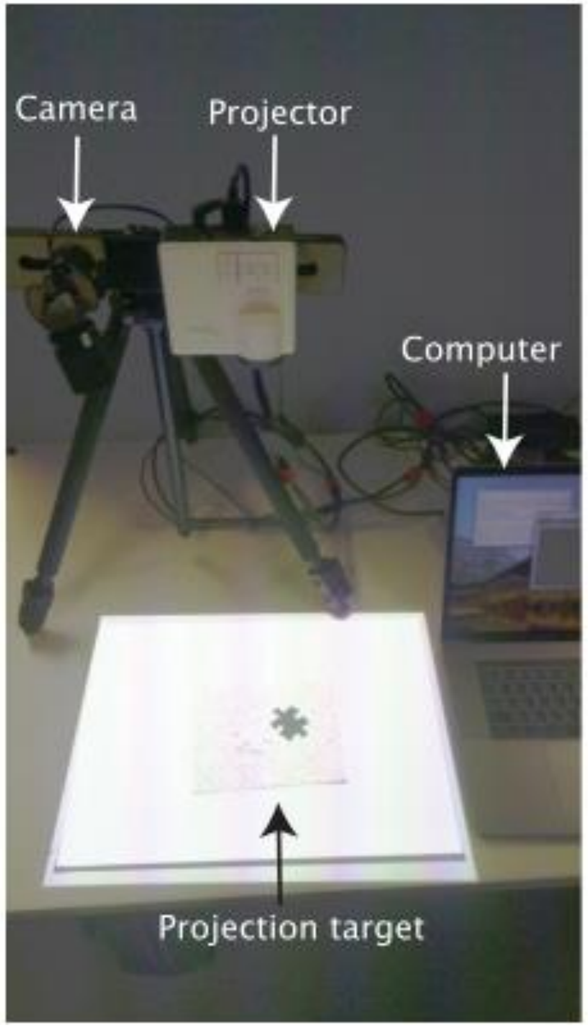
Research directions

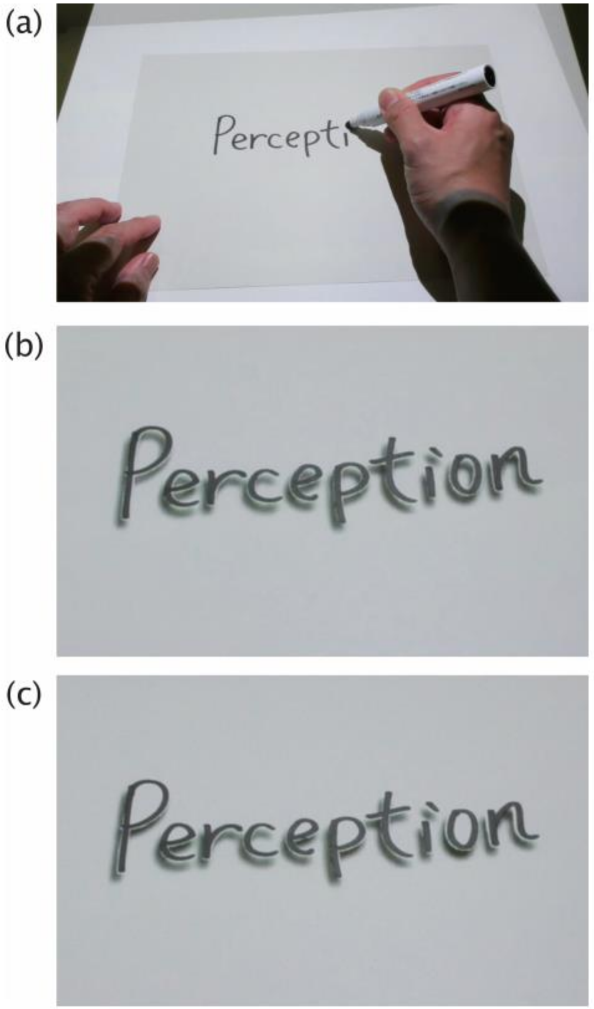
Improved anti-aliasing for Euclidean distance transform shadow mapping (EDTSM), 2017[1].



Research directions

Shadow-based Illusion of Depth and Transparency in Printed Images, 2019[2].





Research directions

ARKit and ARCore.

Reference

[1] https://www.youtube.com/watch?v=V7tH6QXvJic

[2] https://dl.acm.org/doi/10.1145/3342350