Shadow Mapping

A common solution for real-time shadows

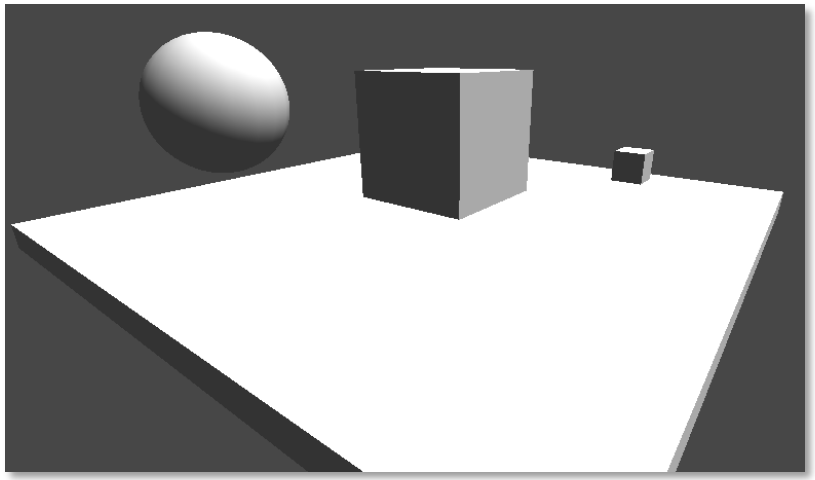
Types of shadows

- Real-time
  - Shadows that are calculated every frame
  - Correct even if the objects or lights move

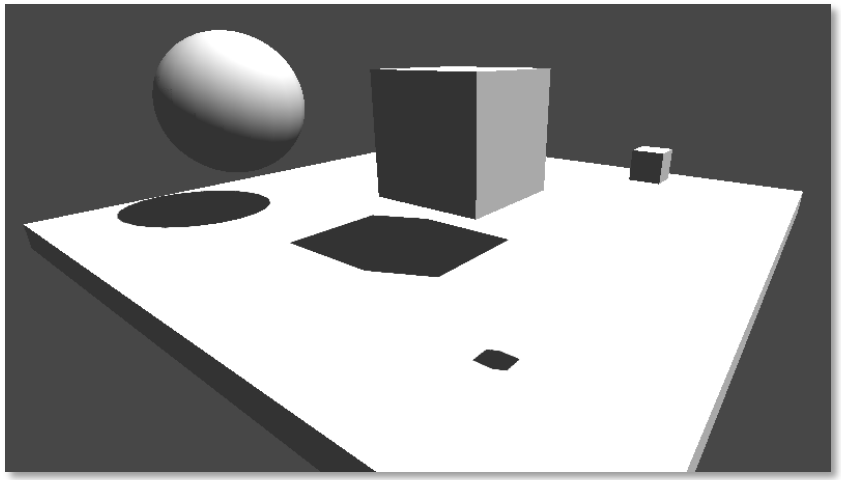
- Pre-calculated (baked)
  - Shadows that are created at design or build time
  - Only work if light source & objects don't move
  - Usually "baked" into textures or mesh vertices
  - Less costly at run time though

Why shadows?

- Realism is a big reason
- Also great for determining relative distances between objects



Where are the objects
in relation to the ground?



Relative distances
are far more obvious

Where are *our* shadows?

- Our objects don't cast shadows
- Basic surface illumination is N·L
  - Normal vs. Light direction
  - Just two vectors
- No "scene" information available in our pixel shader
  - "Which objects might block light?"
  - "Is there a surface between this pixel and the light?"

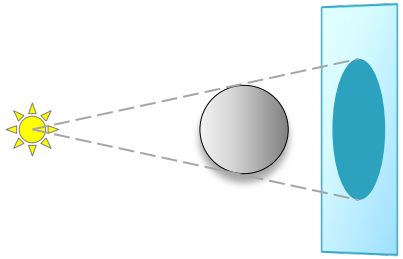
Shadows at first glance

- How could we determine if the current pixel is in shadow?
- Loop through all potential light blockers and check?
  - So…every triangle of every object in the scene?
  - To see if it is between this pixel and the light?
- Ain't nobody got time for that
  - We don't have access to that data in our shader
  - And it would be *incredibly slow* if we did this *for every pixel*

How else could we handle shadows?

- Two problems

- 1. Need information about “light blockers”
  - What are the closest surfaces to the light?

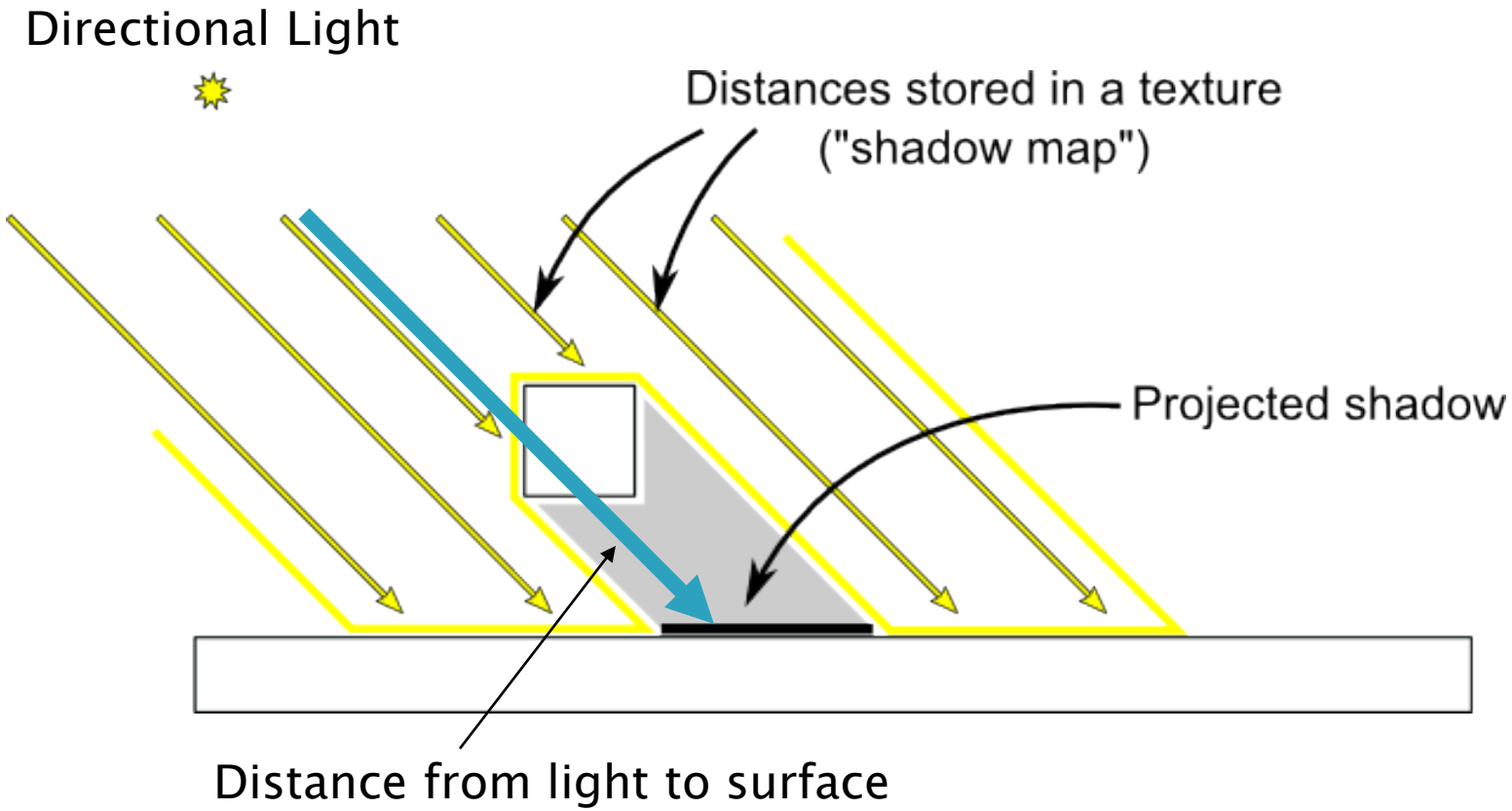


- 2. Needs to be quickly accessible in a pixel shader
  - What kind of resource can hold LOTS of data?
  - And supports arbitrary access in a shader?

Shadow mapping

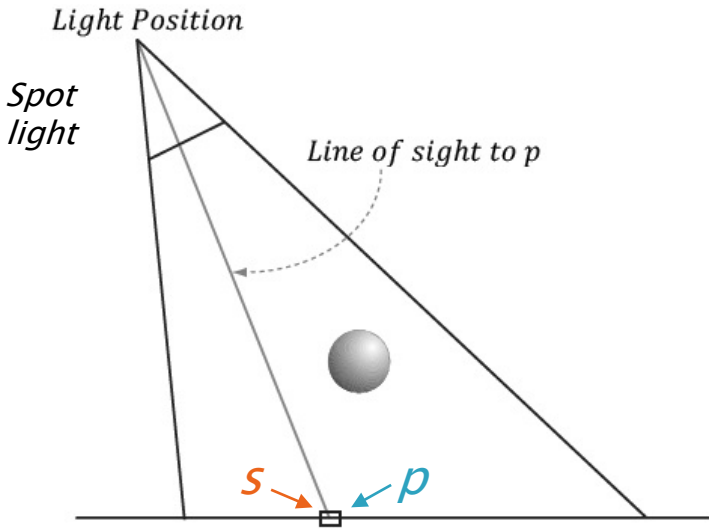
- A texture-based approach to shadows
- Stores “shadow information” in a texture
  - What is “shadow information”?
  - Distance of each surface *from the light*
- That texture is called a “shadow map”
  - Used when drawing the scene
  - Allows us to determine which pixels are “in shadow”

Shadow mapping – Directional light example

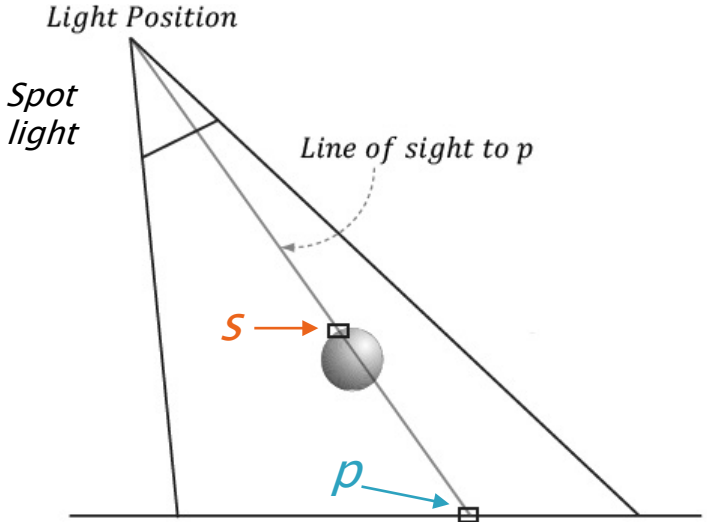


Shadow mapping – Spot light example

- For each rendered pixel *p*, compare:
  - Distance from light source to *p (calculated)*
  - Distance to closest surface *s* along same direction (from shadow map)



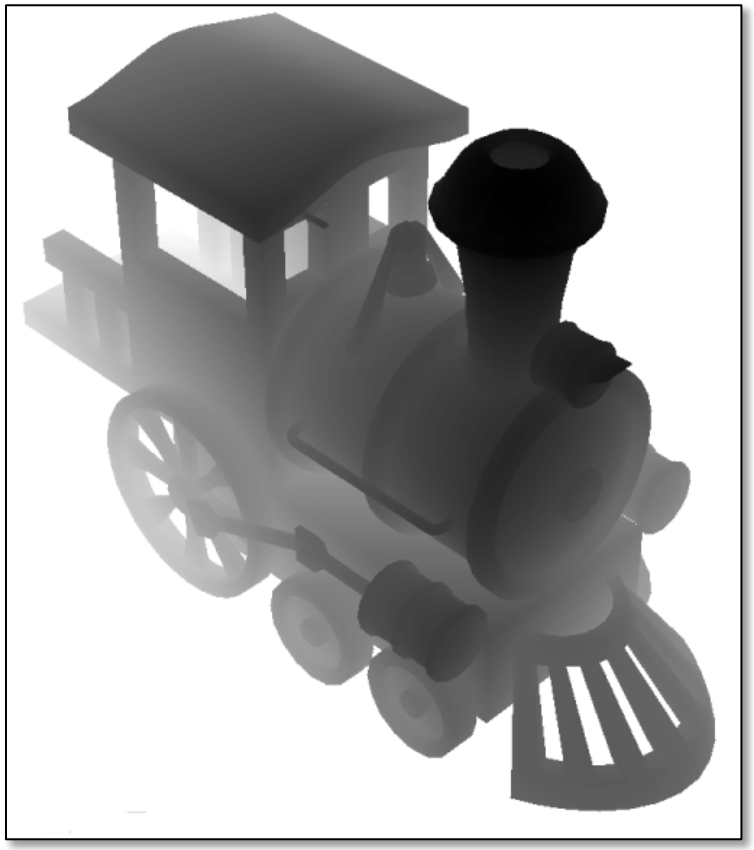
Lit pixel (*p* == *s*)



Shadowed pixel (p > s)

What does a shadow map look like?

- Darker
  - Closer to 0.0
  - Closer to the light
- Brighter
  - Closer to 1.0
  - Further away from light
- Looks suspiciously like a depth buffer 🤔



Basic rendering steps

void Draw()
{

- Render scene entities to **shadow map**
  - From light's point of view
  - (Do this once for *each* light that casts shadows!)

- Render scene entities to **screen**
  - Pass in shadow map
  - Perform per-pixel shadow calculation(s)

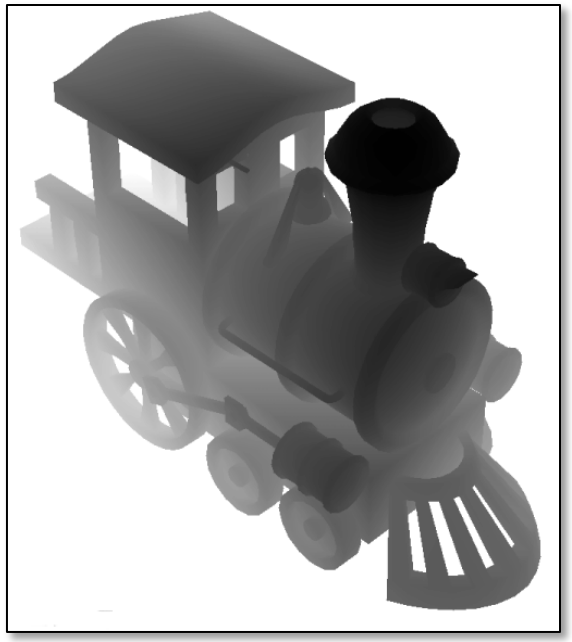
- Render **sky**
- Render **sorted transparent objects**

}

Shadow Mapping

Implementation

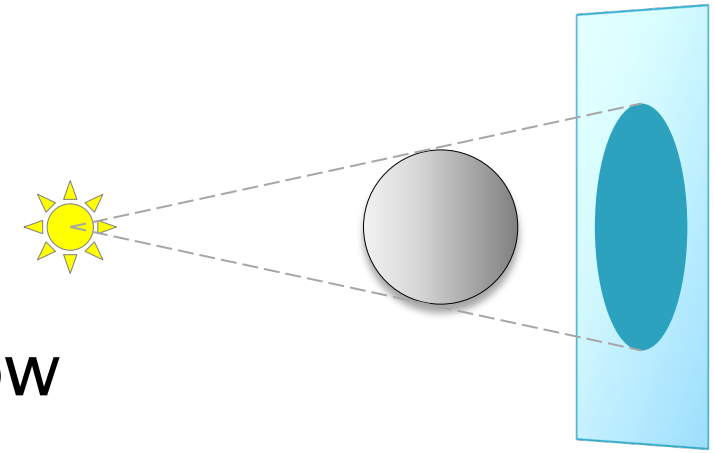
Shadow map == depth buffer



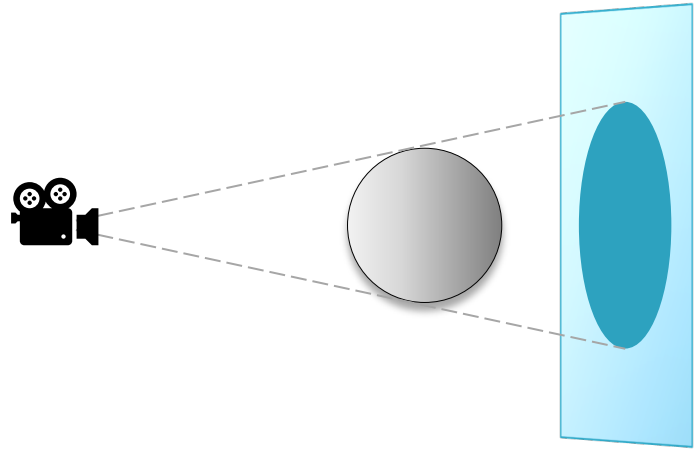
- We need depth of each pixel
  - From the light's point of view
- Sounds like a depth buffer!
  - Stores depth of each pixel
  - For every object we render
- Need a new depth buffer for *each* light that casts shadows
  - Generally you'll just have one main shadow-casting light
  - Usually the main light source (the sun)

What's in a Shadow Map?

- Everything the light can "see" is lit
- Everything the light can't "see" is in shadow



- How do we know what the light "sees"?
- Render the scene!
  - From the light's point of view
  - As if the light was a "camera"

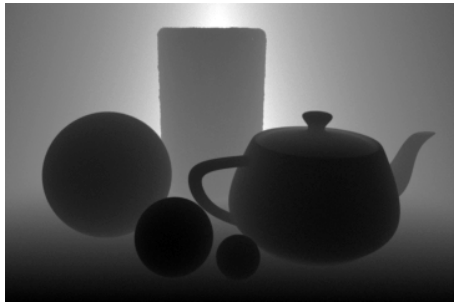


Rendering a shadow map

- Make a “camera” at light’s position
  - View & proj matrices
  - To match light’s pos & dir

- Render all objects
  - Or just the ones the light could “see”

- Put results in a new depth buffer
  - NOT the screen!
  - Just need *depth* of each pixel
  - No color, shading, etc.



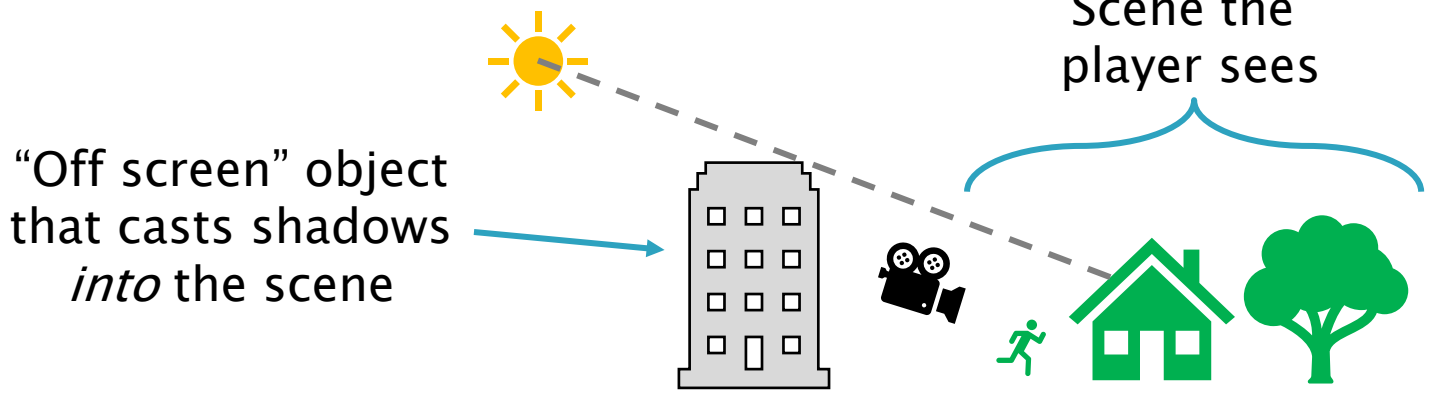
Creating shadow map depth buffer

- Requires a Texture2D
  - Square texture – Power of 2 for size (for performance)
  - Bind flags: DEPTH_STENCIL and SHADER_RESOURCE
  - Format is R32_TYPELESS
- Also requires a Depth Stencil View
  - For using this texture as a depth buffer
  - Format is D32_FLOAT
- And a Shader Resource View
  - For sampling from the texture in a pixel shader
  - Format is R32_FLOAT

Ok, I have a depth buffer. Now what?

- Render everything to it!
  - From the light's point of view
  - Before rendering *anything* to the screen
  - Kind of like a "pre-process"

- What's "everything"?
  - Anything that could cast shadows
  - Even things "off screen"

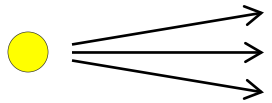


What's the "light's point of view"?

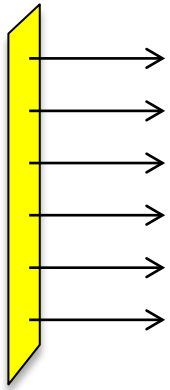
- How the scene looks from the light's position & orientation
  - Normally handled by a "camera"
  - Really just need the matrices: view and projection
- Create the matrices as if we had a "camera" there
  - Same position
  - Facing the same direction
- Note:
  - If the light moves/rotates, update the light's view matrix accordingly

Shadow map view matrix?

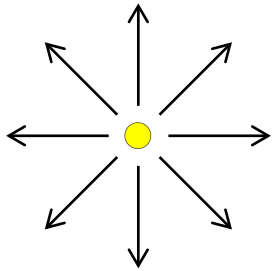
- Spot Lights
  - Have both a position & direction
  - Usually easiest to wrap your head around



- Directional Lights
  - Only have a direction – no inherent "position"
  - Figure out a useful position based on your scene

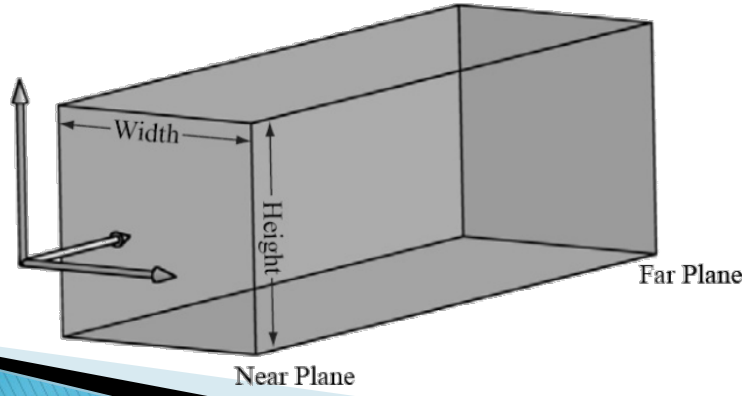


- Point Lights (Advanced)
  - Only have a position – emit light in **all** directions
  - Need to render to a *shadow cube map*
  - Render 6 shadow maps: one in each of 6 directions

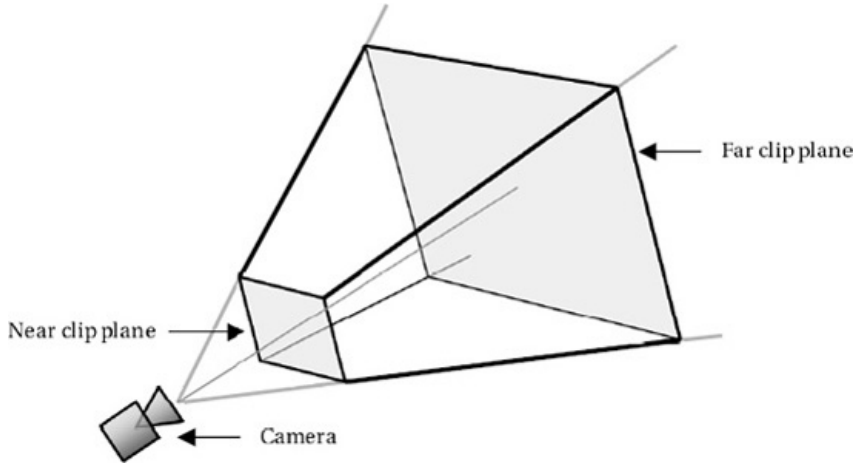


Shadow map projection matrix?

- Projection type should match light type
  - Directional Light?
    - Orthographic projection
    - All rays are parallel
  - `XMMatrixOrthographicLH()`
    - Width & Height in world units
    - Near & Far distances too



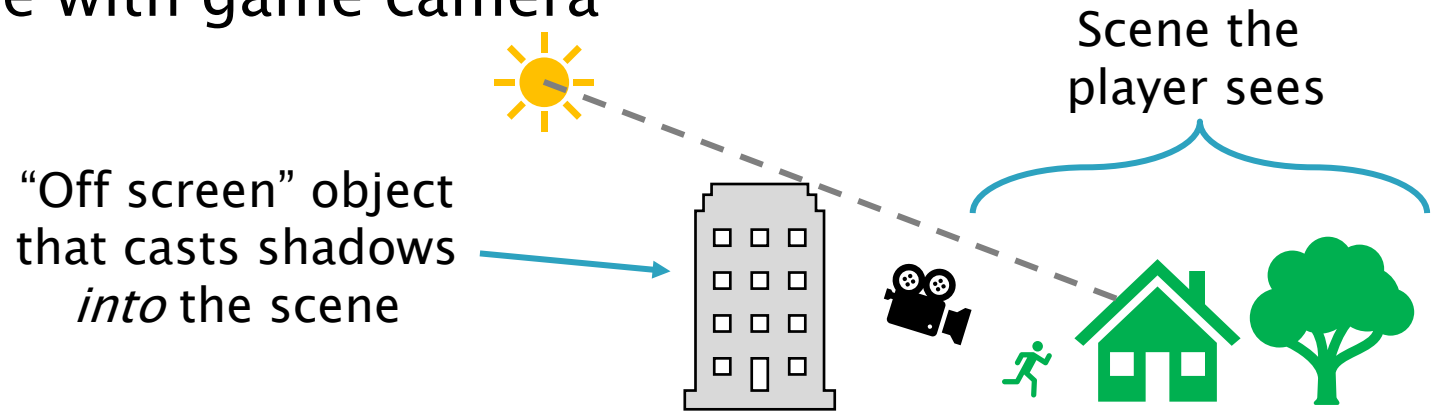
  - Spot Light?
    - Perspective projection
    - Rays converge at camera
  - `XMMatrixPerspectiveFovLH()`
    - Field of view should match light
    - Aspect ratio is 1:1



Shadow map projection options

- Big enough to encompass entire game world?
  - Pro: View & Projection don't need to move
  - Pro: Make them once and forget about them
  - Con: Precision and resolution issues

- Or just encompass the camera's frustum?
  - Only render shadows for what you currently see
  - Shadow View should move with game camera
  - Still need to consider off screen occluders



Shadows from directional lights

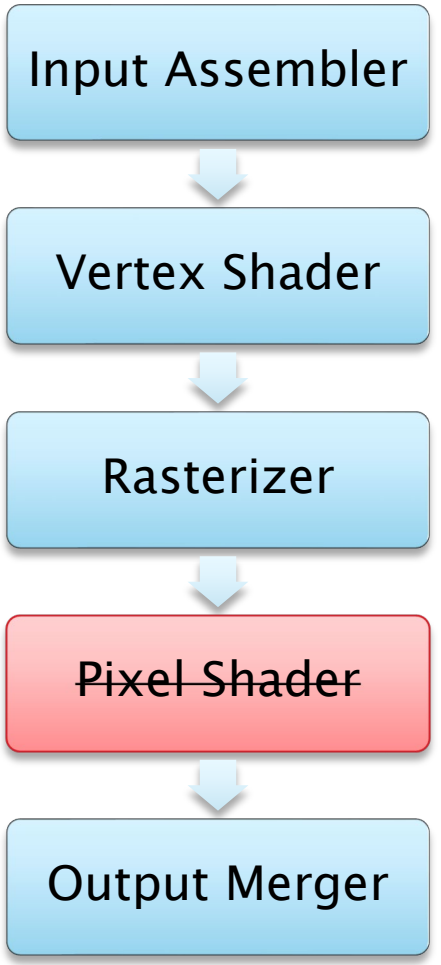
- If you only do one shadow map, ***do this one***
  - Usually represents main light source
  - Most outdoor shadows come from the sun
- What's the "position" of a directional light?
  - Depends on your scene
- Pick a position and "back up" along light's dir
  - Start at center of "game world"?
  - Start at center of player's view frustum?

How do I “render to a shadow map”?

- Change the current **render targets**
  - Set the new depth buffer (your shadow map)
  - Set a null render target
- Set a **viewport** that matches shadow map size
- Set a simplified **vertex shader**
- Set a null **pixel shader**!

Shadow map render – Shaders?

- Use a simplified Vertex Shader
  - Output exactly one thing: Position
  - No UV's, normals, etc.
  - World matrix is from the current entity
  - View & projection are *light's point of view*
- No pixel shader though!
  - Rasterization will still put data into depth buffer
  - `deviceContext->PSSetShader(0, 0, 0);`



Side note: No pixel shader?

- For opaque objects, nope!

- Alpha cutout?
  - You WILL need a shader that selectively discards
  - Based on alpha value of surface texture
  - Still no color necessary

- Actual transparency/translucency
  - Doesn't work with basic shadow mapping
  - See 5.8 in SC 2 Rendering article for advanced technique

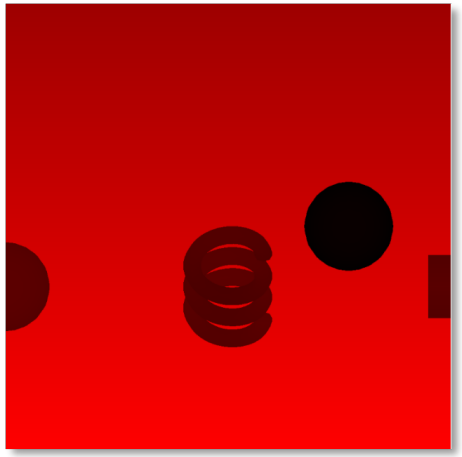
Shadow map render – Finishing up

- Render all entities in your scene
  - Don't use their materials though
  - Simply use special "shadow map VS"
  - Pass in entity's *world*, but light's *view* & *projection*

- Afterwards: Reset for "normal" rendering
  - Original render target & depth buffer
  - Original viewport
  - Default rasterizer state (just set to null)

Options for checking progress

- Draw shadow map to the screen
  - Use SpriteBatch to draw the texture



- Use the Graphics Debugger to look at it
  - Look for R32_TYPELESS texture object

Graphics Object Table

376: obj:1->Present(0,0)=S_OK

Name	Type	Active	Size	Format	Width	Height
rockNormals.jpg	D3D11 Shader Resource View			R8G8B8A8_UNORM	1024	1024
SunnyCubeMap.dds	D3D11 Shader Resource View			B8G8R8A8_UNORM	1024	1024
obj:6	D3D11 Texture2D		1,920,000	D24_UNORM_S8_UINT	800	600
obj:30	D3D11 Texture2D		33,554,424	B8G8R8A8_UNORM	1024	1024
obj:35	D3D11 Texture2D		4,194,304	R32_TYPELESS	1024	1024
WICTextureLoader(1)	D3D11 Texture2D		5,592,404	B8G8R8A8_UNORM	1024	1024

I have a shadow map! Now what?

- Render entities to the screen w/ updated shaders
- Updated Vertex Shader
  - Pass in light's view & projection
  - In addition to the game camera's view & projection
- Updated Pixel Shader
  - Pass in the shadow map as an SRV
  - Also pass in a special sampler for the shadow map

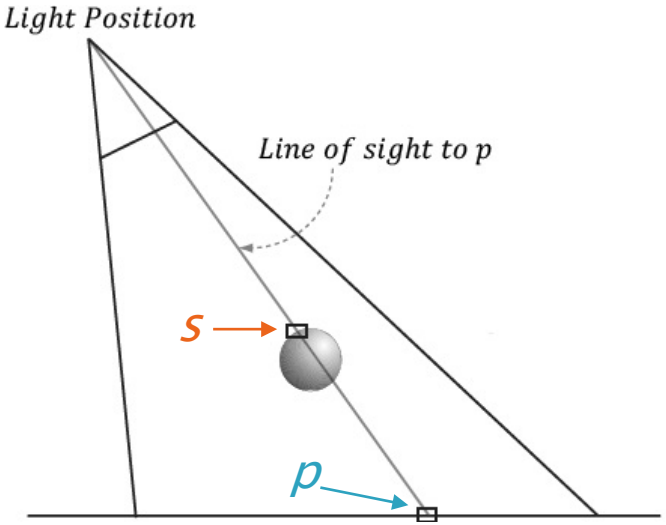
Render with shadows – VS

- Calculate object's screen position as usual
- Also calculate it's "shadow" position
  - world * lightView * lightProj
  - Result is float4 (similar to screen position)
- This is object's position "from the light's PoV"
  - We'll use this to know where to sample shadow map
  - And how far away pixels are from the light
  - Pass this to PS as another part of VS output

Render objects with shadows – PS

- New input:
  - Position from light's PoV (float4 from VS)
  - Shadow map itself (Texture2D)
  - SamplerState for shadow map (more on this soon)

- Several steps to determine shadows:
  - Calculate pixel's depth (distance) from light (P)
  - Calculate pixel's UV position on shadow map
  - Sample shadow map at correct UV (S)
  - Compare depths P and S



Shadowed pixel (p > s)

Render objects with shadows – PS code

```
// Calculate depth (distance) from light
// - Doing the perspective divide ourselves
float lightDepth = in.shPos.z / in.shPos.w;

// Adjust [-1 to 1] range to be [0 to 1] for UV's
float2 shadowUV = in.shPos.xy / in.shPos.w * 0.5f + 0.5f;
shadowUV.y = 1.0f - shadowUV.y; // Flip Y for sampling

// Read shadow map for closest surface (red channel)
float shDepth = shadowMap.Sample(shadowSamp, shadowUV).r;
```

Render with shadows – Last step

- Have to do a comparison of some type
  - Could branch
  - Could use HLSL's step() function
  - Could have Sample() handle it for us (more soon)

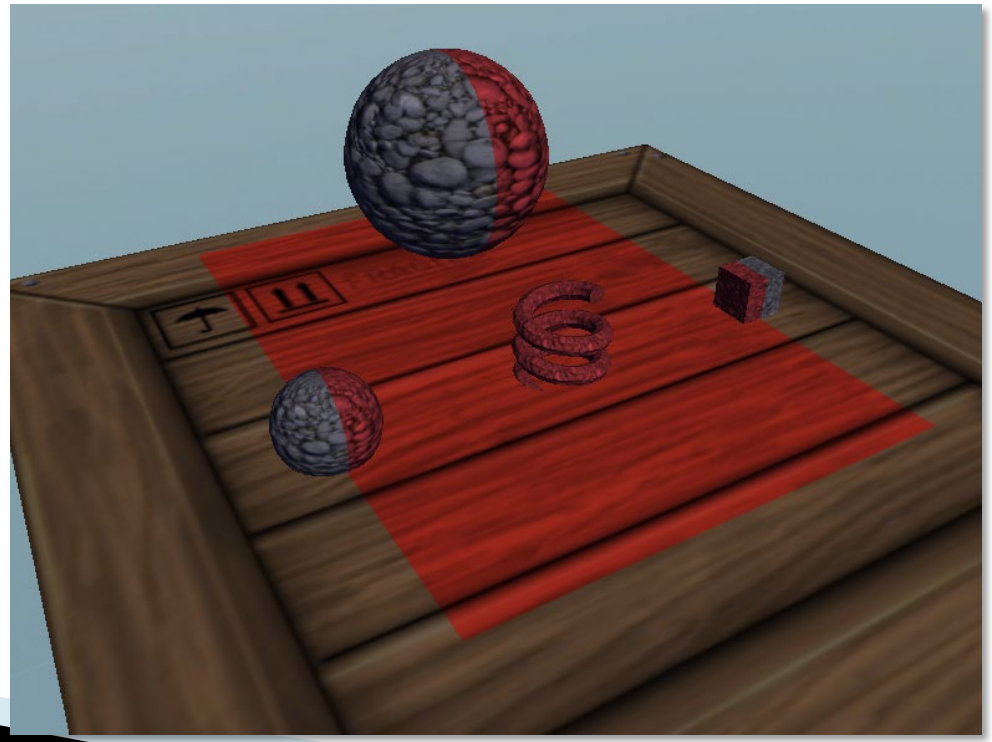
- You'll have a "shadow or not" value
  - Use this to cut the directional light contribution
  - Keep ambient light
  - Keep other lights which aren't casting shadows

Shadow Mapping

Issues & Improvements

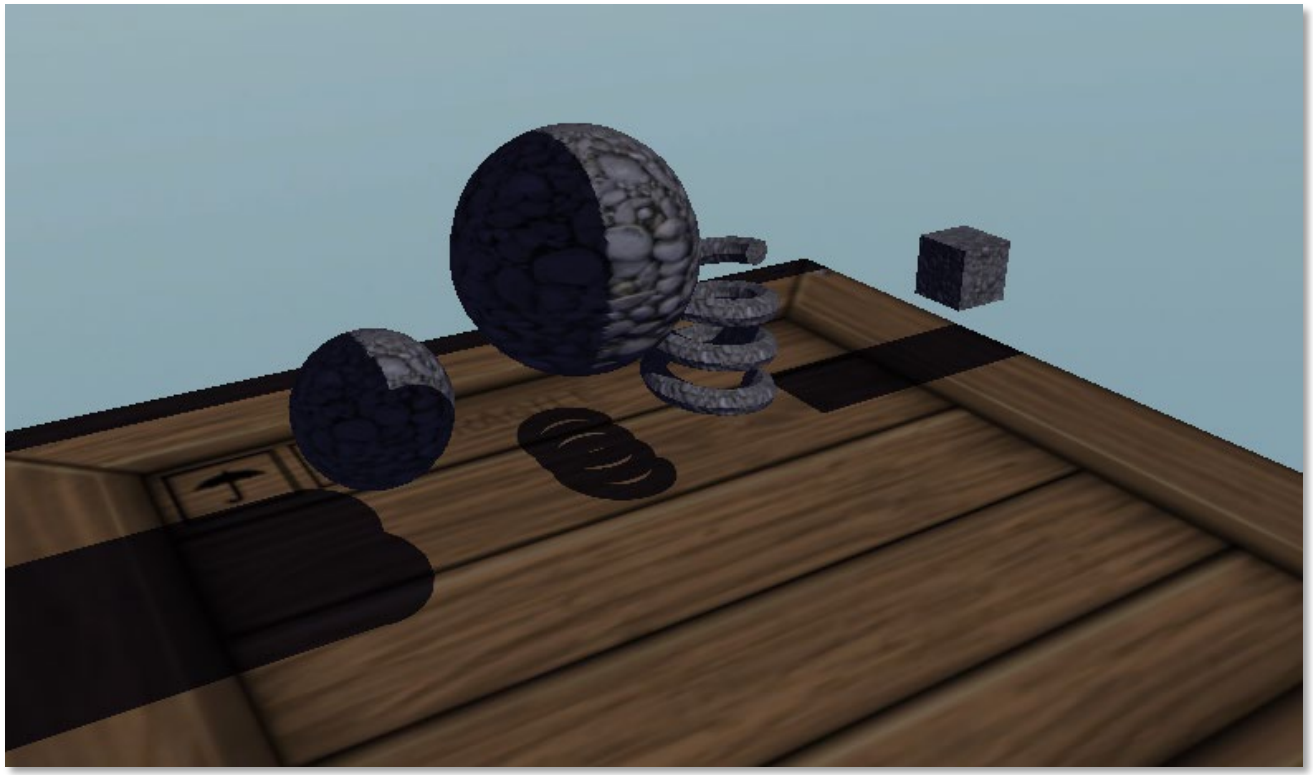
Shadow map coverage

- May only cover a portion of your game world
  - Usually fine if it covers what the player sees
  - Shadow UV coords may go outside the 0-1 range



Shadow map sampler state

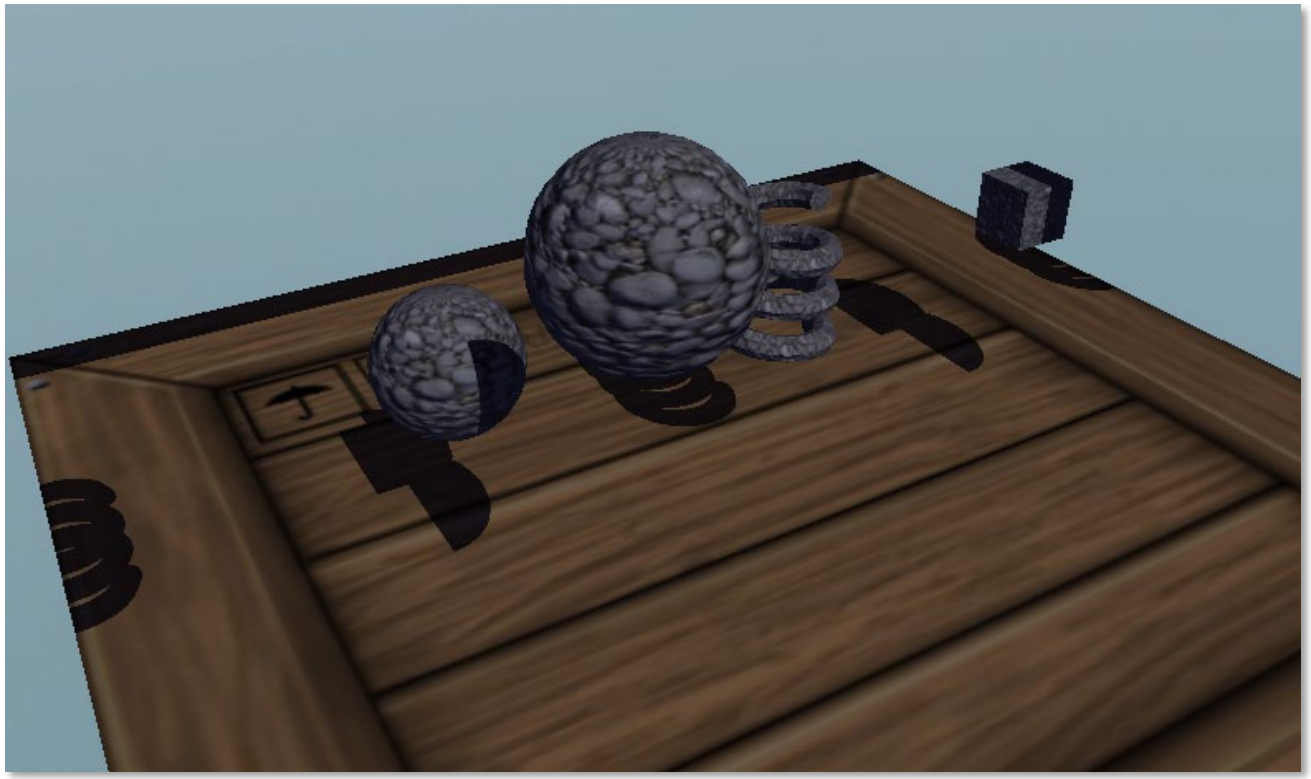
- Which UV address mode?



Clamp – Shadows Go Forever

Shadow map sampler state

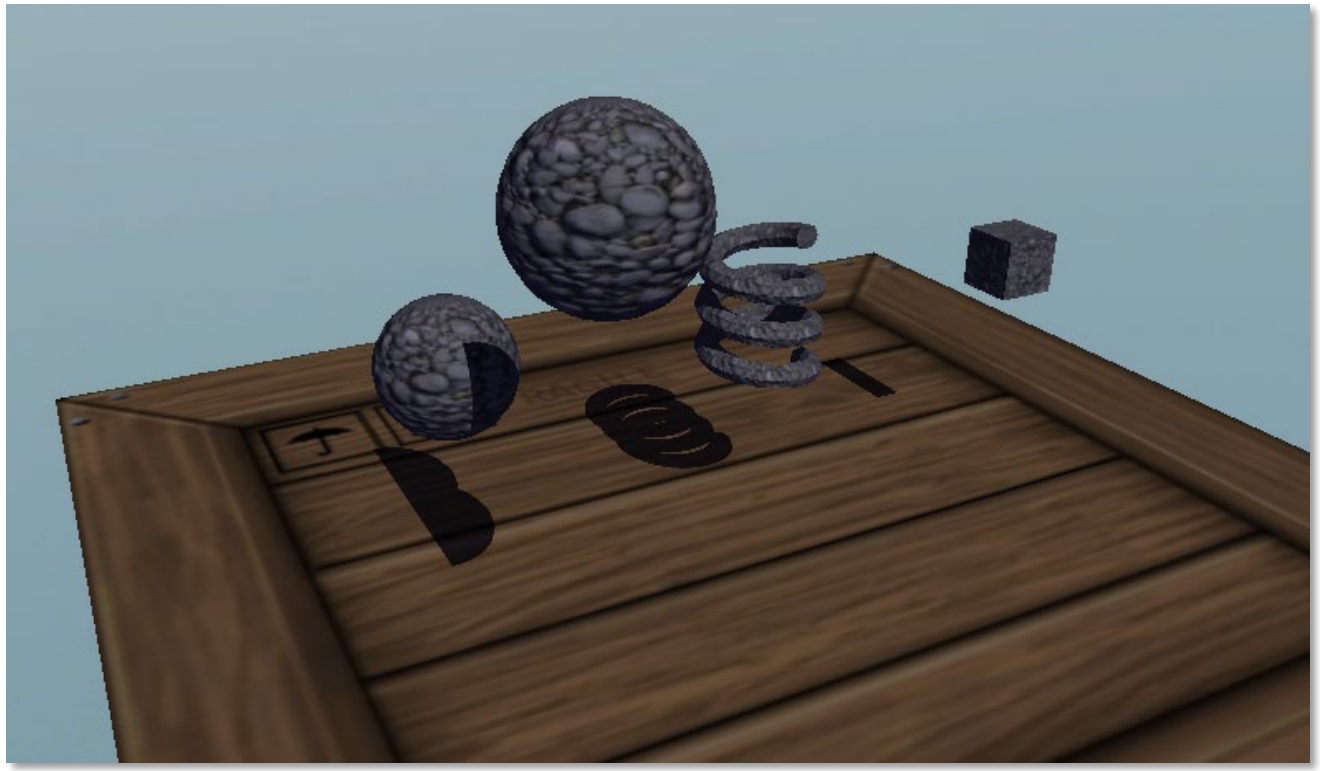
- What about “wrap” mode?



Wrap – Shadows Repeat

Shadow map sampler state

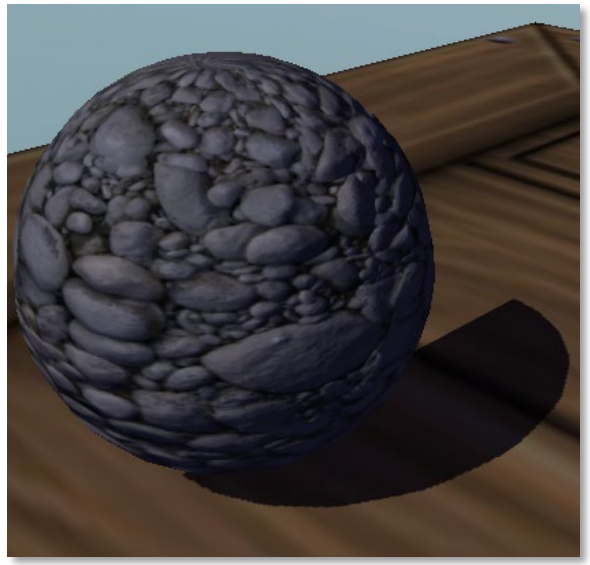
- Let's try "border"



Border – Shadows End!

Shadow Sampler – Border!

- Border addressing
  - Uses a solid color outside [0-1]
  - Use 1.0's (means "far away")
  - No shadows outside light's PoV



- Least offensive option
  - Hopefully player never sees this edge
  - Could "fade" shadows as UV nears 0 or 1

What else can the sampler do?

- Samplers have a “comparison” mode
  - Will sample a texture and compare against a value
  - Gives back a value representing comparison results
- Can even compare neighboring pixels
  - Gives back linear interpolation of results
  - Useful for soft shadows!
- Change sampler description as follows:
  - `sampDesc.Filter = D3D11_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;`
  - `sampDesc.ComparisonFunc = D3D11_COMPARISON_LESS;`

Sample & compare in HLSL

- Pixel Shader changes:
  - “SamplerState” to “SamplerComparisonState”
  - Replace current sample with:

```
float shadowAmount = shadowMap.SampleCmpLevelZero(
    shadowSamp, shadowUV, depthFromLight);
```

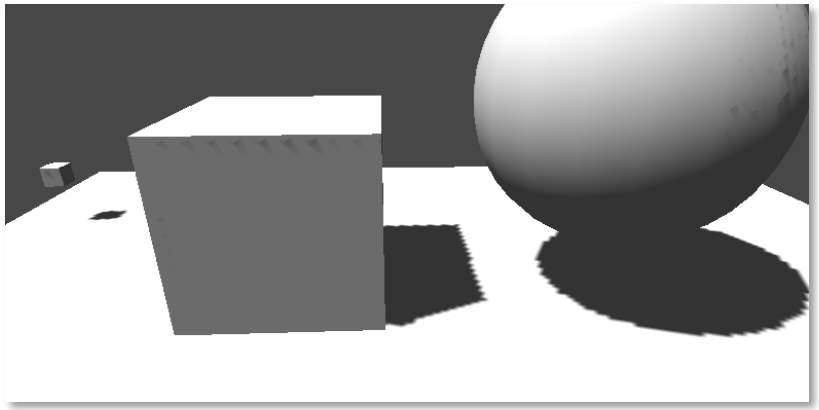
- SampleCmpLevelZero()
  - Takes a sampler, UV and *value to compare*
  - Reads first component in texture
  - Applied comparison function to both values
  - *LevelZero* part refers to always using lowest mip

Shadow mapping issues

- Shadow mapping is an imperfect technique
  - We can overcome some of the issues
  - And it's still fast enough for real-time

- Potential issues:
  - Resolution
  - Aliasing
  - "Hard" shadow edges

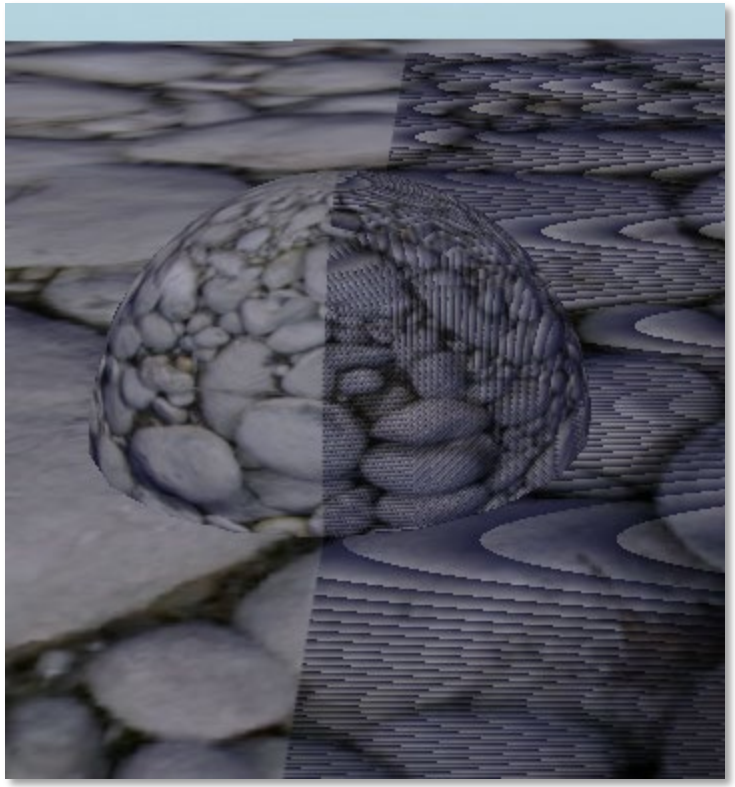
Shadow map resolution

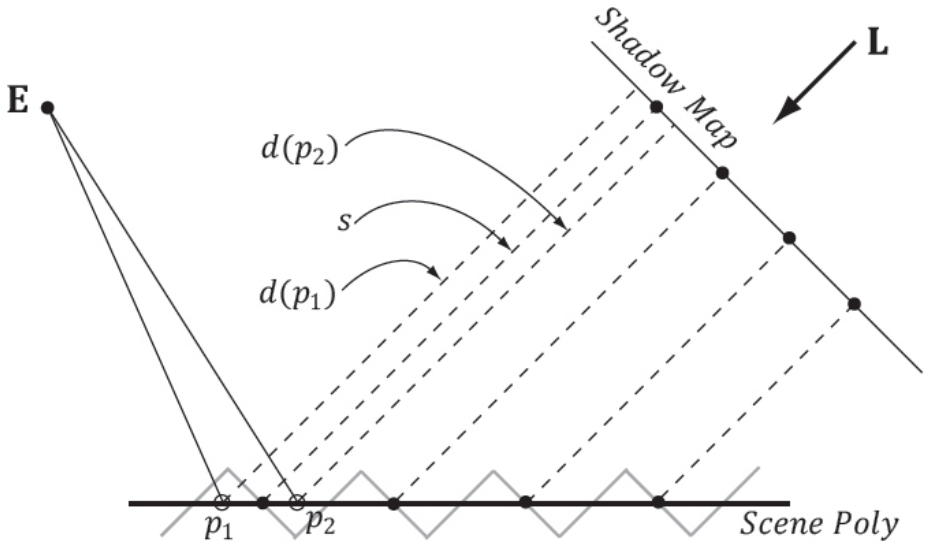


Low Res
- Smaller
- Faster to render
- Rough edges
- Unwanted artifacts

High Res
- Larger
- Slower to render
- Smoother edges
- Artifacts go away

Aliasing – Shadow acne

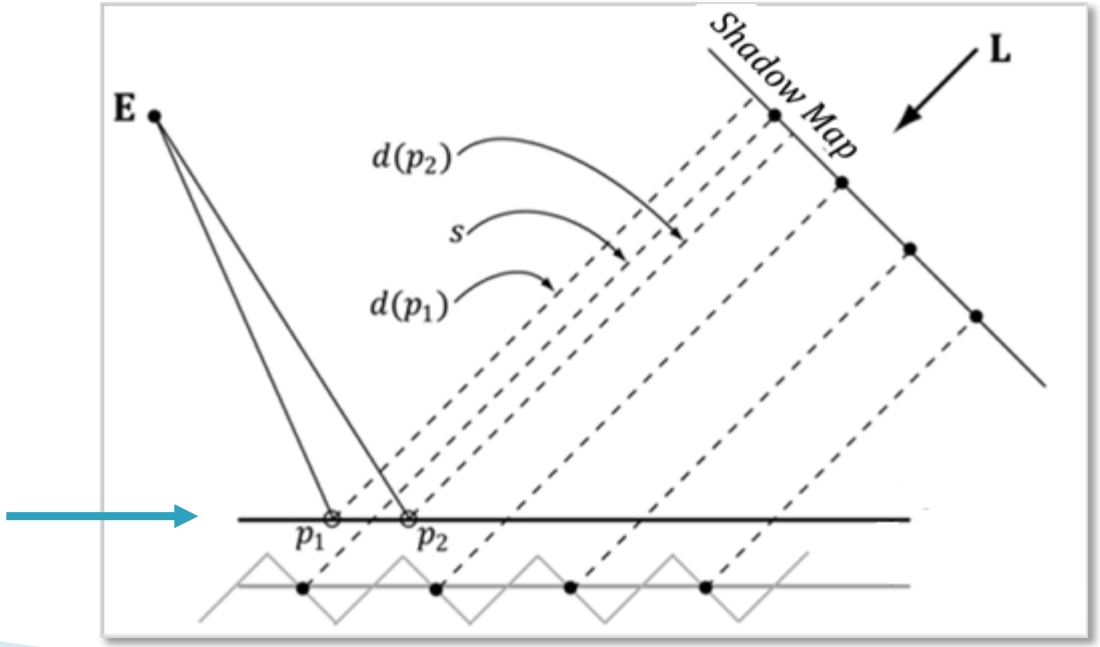




Aliasing & bias

- Shadow “acne” solution: Add a bias amount

- Assume the polygon is slightly offset from it’s actual position to overcome inconsistency

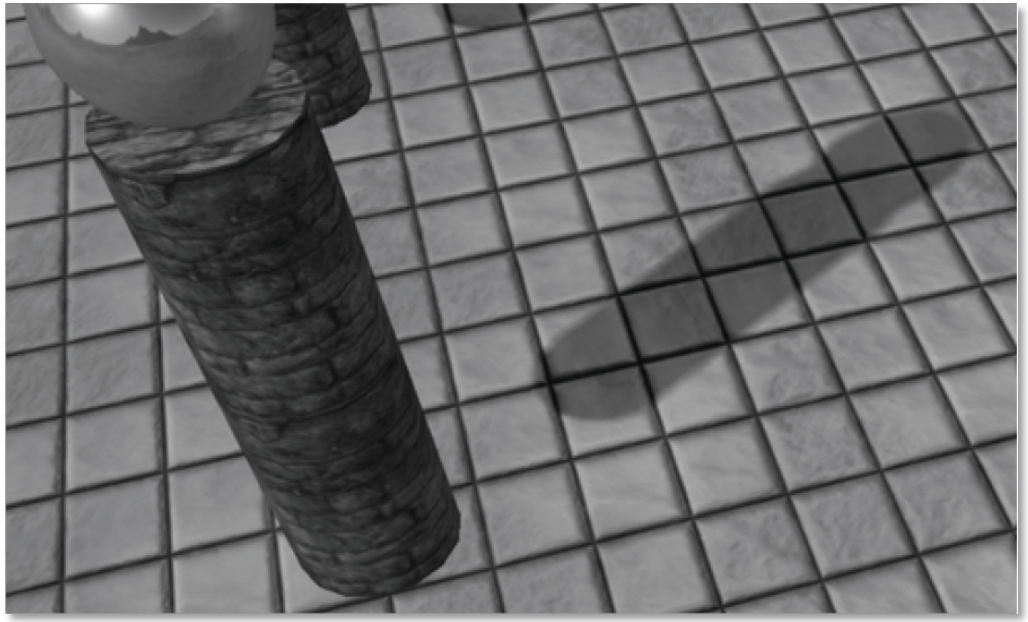


Bias & rasterizer

- Rasterizer can apply a bias for you
  - And take into account the slope of the polygon
  - Use it during initial shadow map render
- Create a rasterizer state with defaults, and:
  - `rsDesc.DepthBias = 1000;`
  - `rsDesc.DepthBiasClamp = 0.0f;`
  - `rsDesc.SlopeScaledDepthBias = 1.0f;`
- *DepthBias* member above is an int
  - It's multiplied by smallest possible positive value which fits into the depth buffer (based on format)

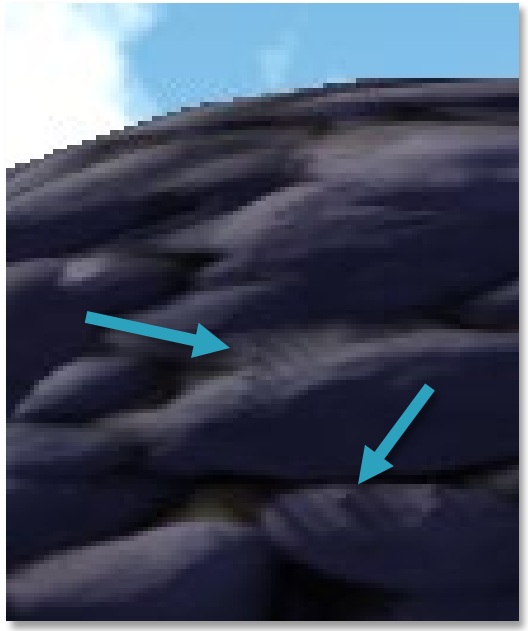
Bias issues

- Of course too much bias leads to issues too
  - Large bias may lead to disconnected shadows
  - “Peter panning”

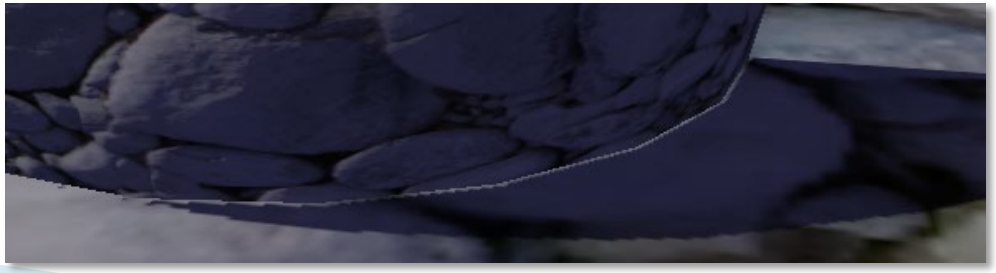


Self shadowing

- Can lead to aliasing
  - Front side casts shadows on back
  - Especially bad on surfaces perpendicular to light direction



- Render back sides of objects
  - Instead of front sides
  - Back sides now cast shadows
  - But can “disconnect” shadows



DirectX requirements recap

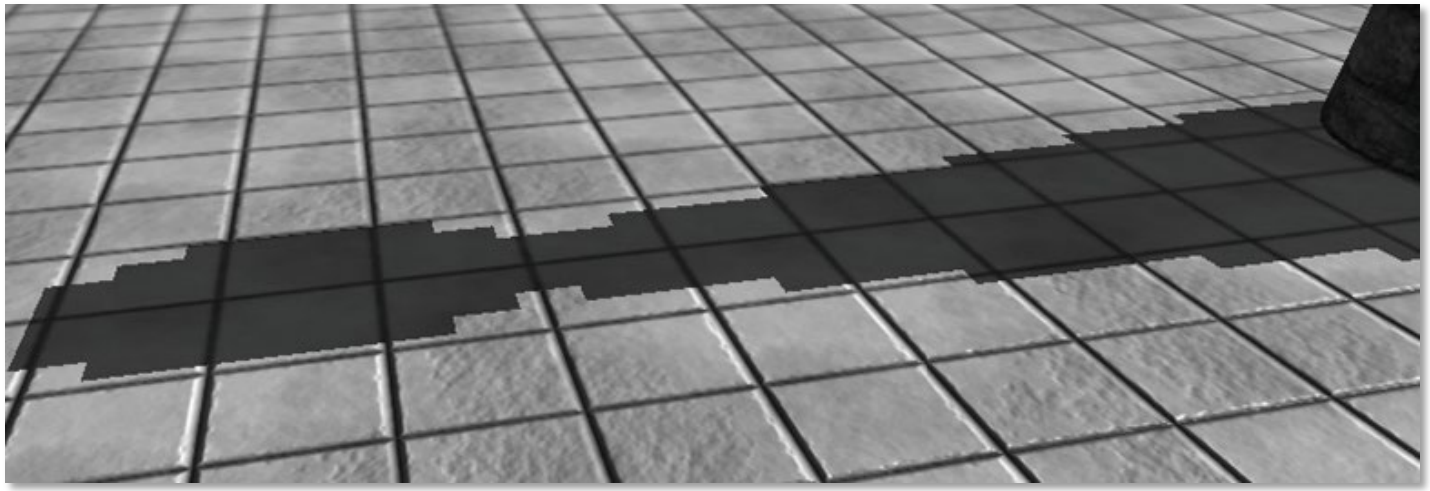
- Texture2D (the shadow map itself)
  - Depth Stencil View
  - Shader Resource View
- Comparison Sampler State – To simplify comparing depths
- Rasterizer State – To apply depth bias
- Viewport – Must match shadow map resolution
- New vertex shader – For *creating* shadow map
- Light's PoV view matrix
- Light's PoV projection matrix
- Updated shaders – For applying the shadow map

Shadow Mapping

Advanced Topics

Hard shadows

- Default implementation has hard edges
  - Regardless of the resolution
  - Example here is especially low-res to show issues



Can’t just blur it

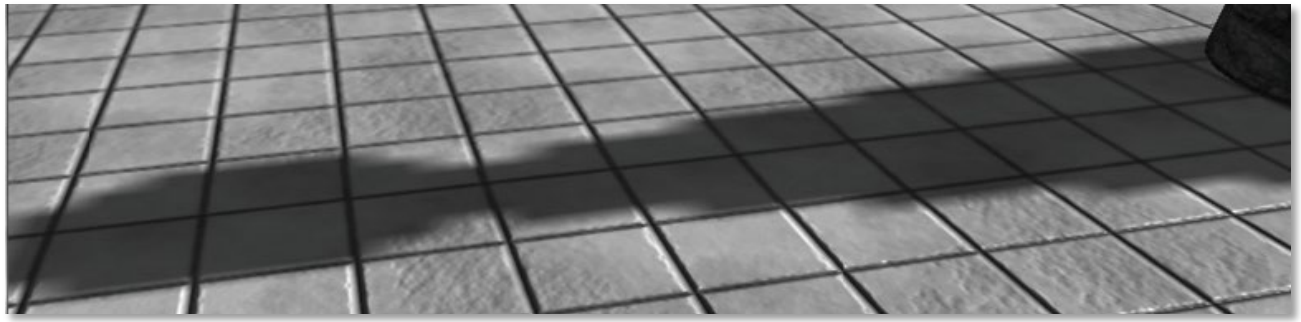
- Shadow map isn’t “textured” onto surface
  - It’s just a bunch of depth values
  - “Blurring” depths just jitters the edges

- There are shadow techniques that use a blur
  - Variance Shadow Maps
  - Have issues of their own →

Common soft shadows – PCF

- Percentage Closer Filtering
  - Take extra, near-by samples
  - What percentage of near-by pixels are in shadow?
  - Use that percentage as a shadow fall-off

- SampleCmpLevelZero() does this!
  - Still need multiple samples for very soft shadows



Advanced techniques

- Problem: Shadow Map Coverage

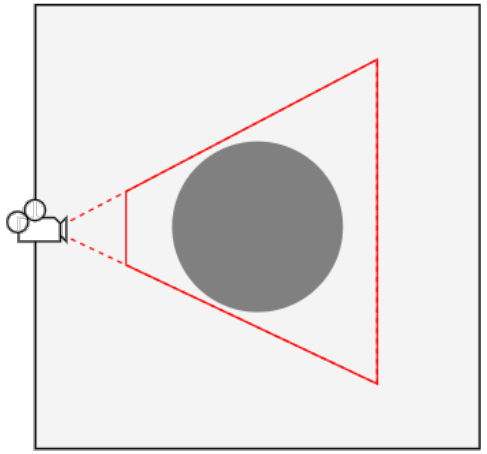
- Small shadow map
  - Doesn't give enough detail close to the camera
  - DOES have enough detail for distant objects

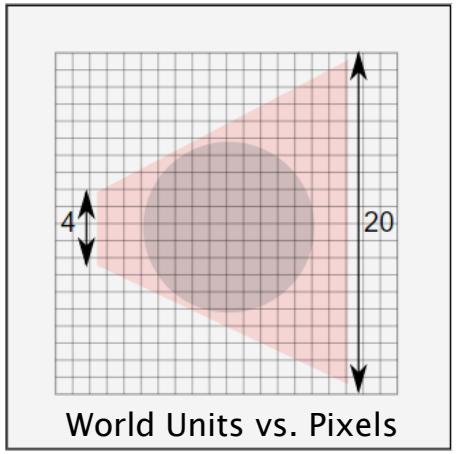
- Larger shadow map
  - Gives more detail up close
  - But much of that "detail" is lost (skipped) far away
  - Takes much longer to render

Shadow map coverage

- How much area does a single pixel represent?
  - Pixels of distant objects represent a **large** area
  - Pixels of closer objects represent **smaller** areas

- Shadow maps have a set number of pixels
  - Each represents a static area





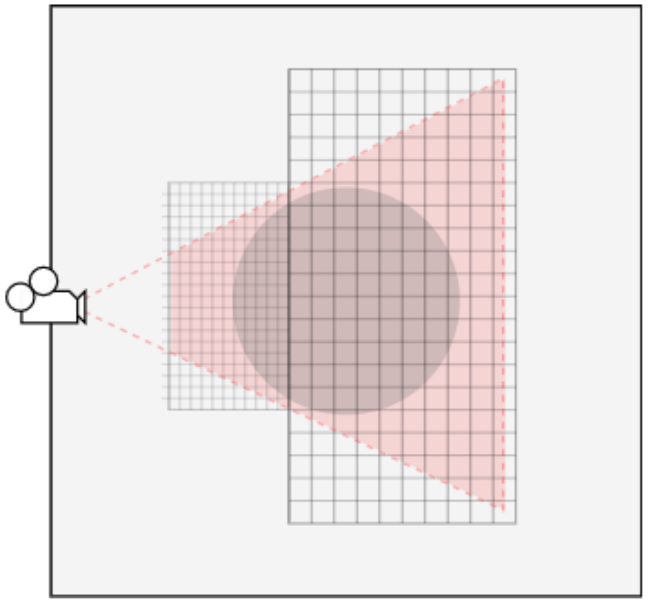
World Units vs. Pixels

Shadow map coverage

- Should we use a super high-res shadow map?
  - So that we get more detail everywhere?
- No! Excessive and expensive!
  - Closer objects need higher shadow detail
  - Objects in distance need less shadow detail
- How do we handle both cases?
  - Render 2 (or more) shadow maps PER light!
  - And align them based on distance

Cascaded shadow maps

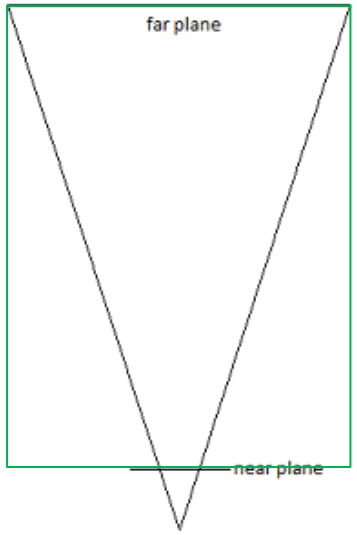
- Several shadow maps
  - To more closely match each pixel's world area

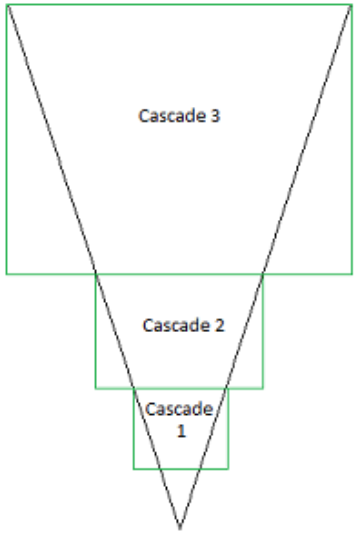


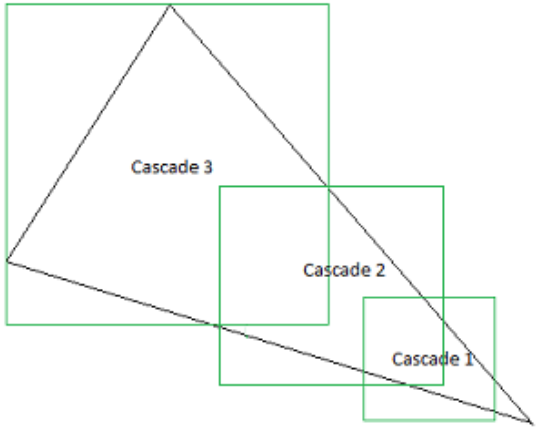
- Requires multiple shadow maps per light
  - Faster than one single MASSIVE shadow map
  - More accurate than one small shadow map

CSM – Example

- Different ways of setting up cascades

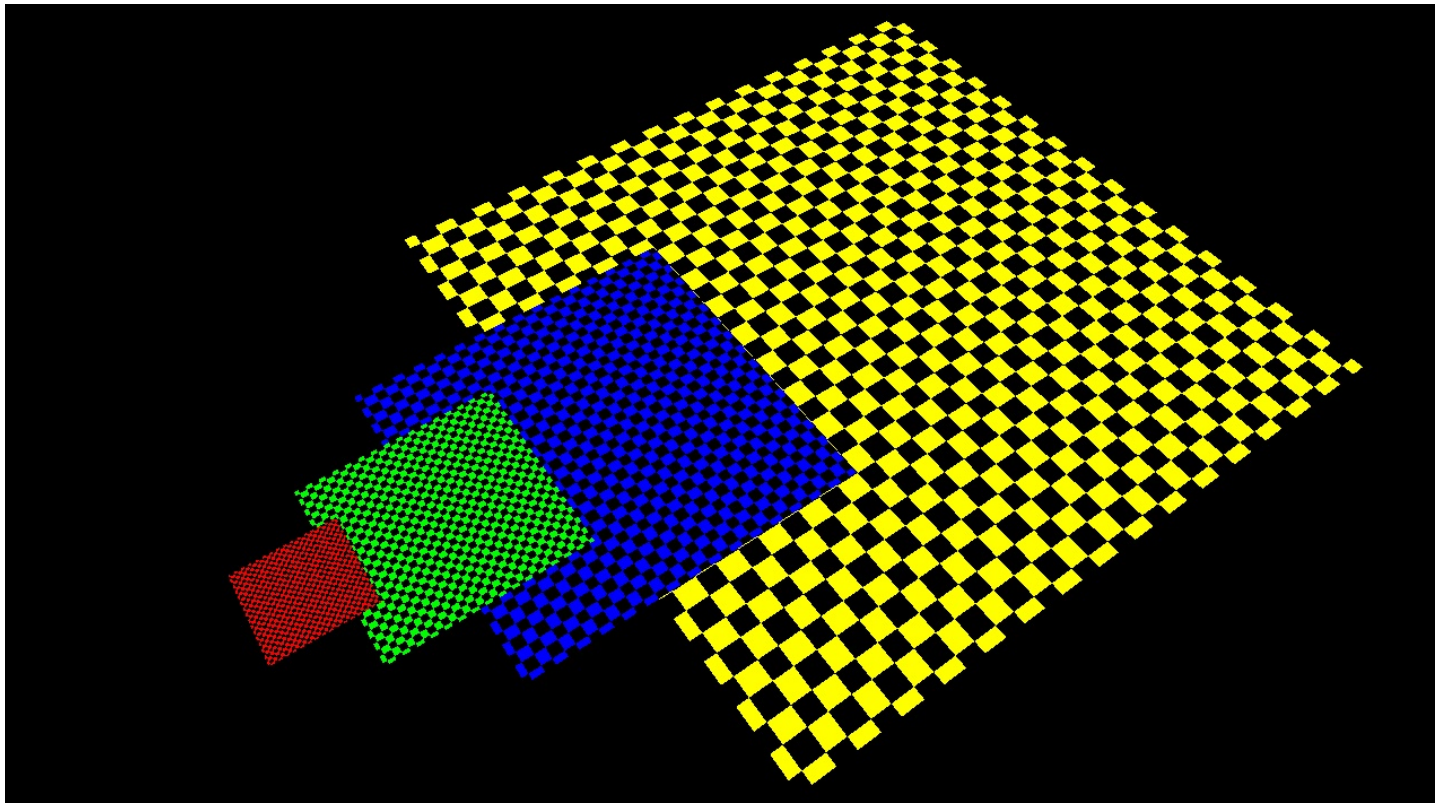






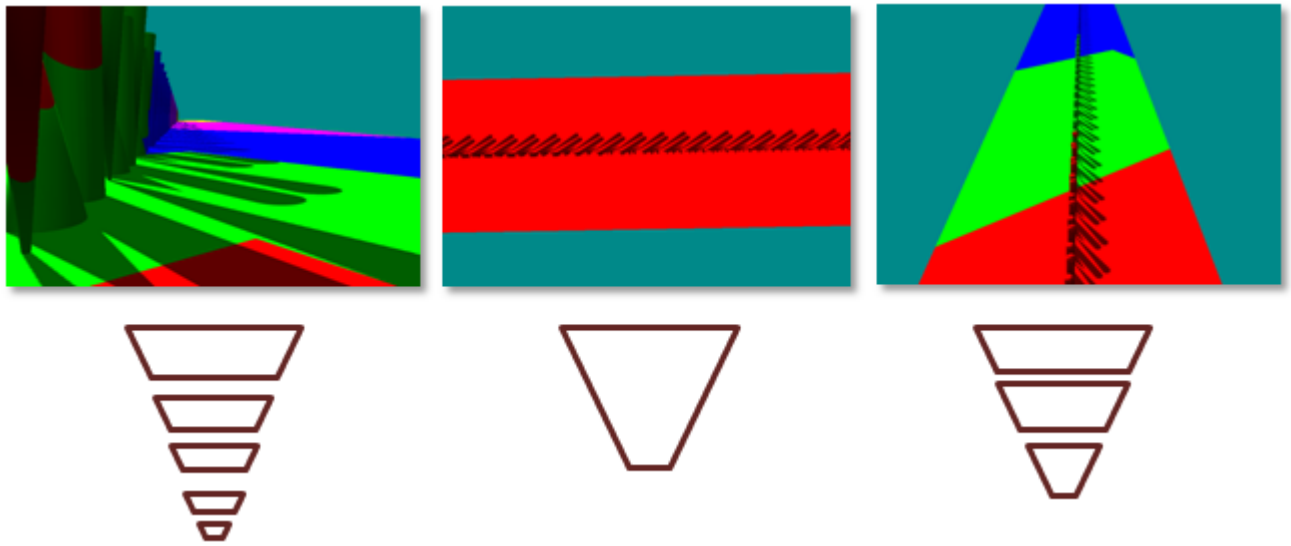
CSM – Example

- Each cascade has different coverage



Cascaded Shadow Maps

- Number of cascades depend on the scene and camera placement:



- Images from Microsoft article on CSMs