Real-time Shadows

Javi Agenjo 2020

Source code of the Framework

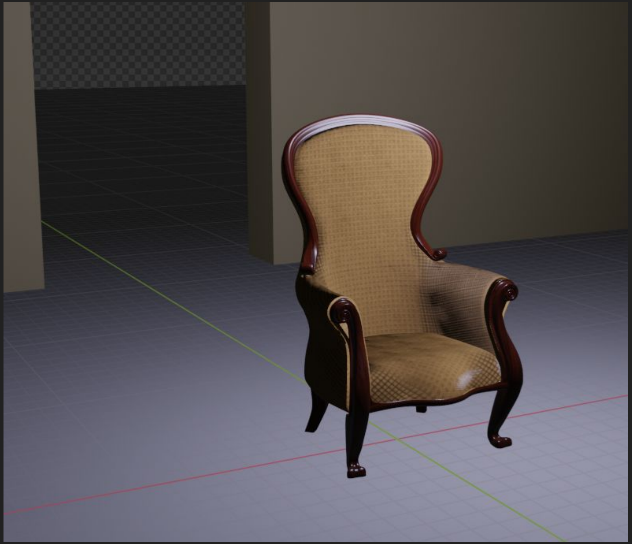
Introduction

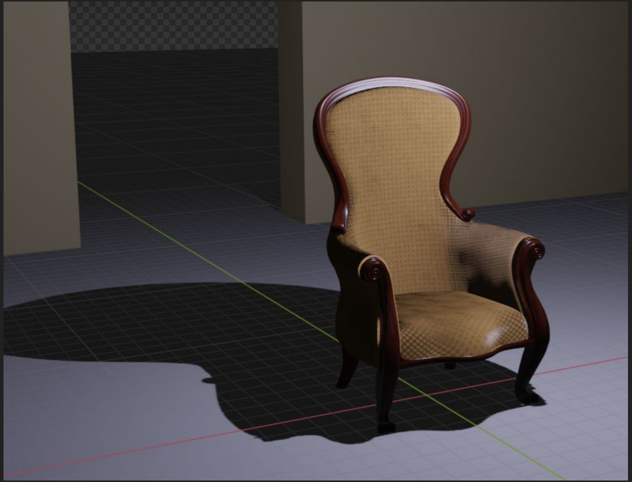
In the previous presentation we evaluated the amount of light reflected by an object just by taking into account the orientation of the pixel and the direction of the light.

But in that approach we are ignoring an important concept, occlusions.

Between the pixel position and the light position could be an object blocking the light path, and will be very noticeable.

In this presentation we will talk about projected shadows.

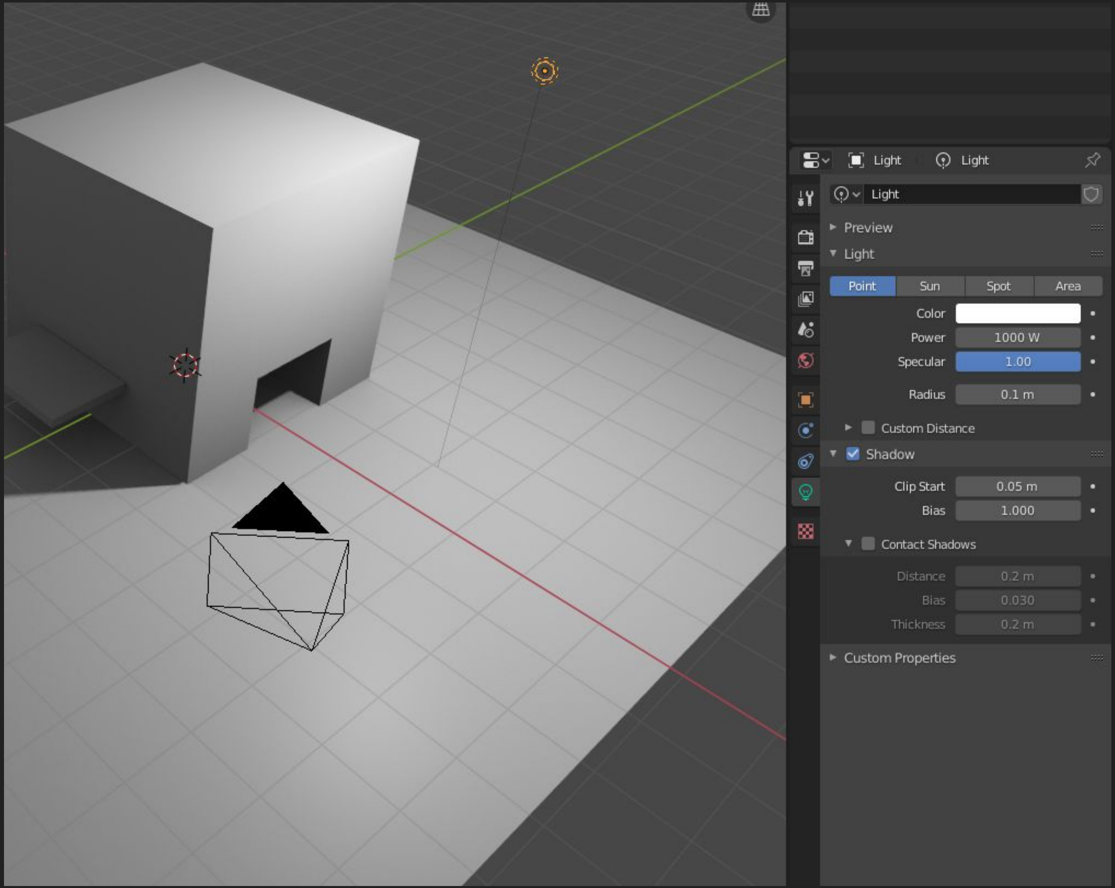




Blender

During this presentation we will be talking about shadows and their properties.

If you want to test many of the concepts explained here, I recommend you try [Blender 2.8]() using the Eevee renderer so you can see how changing some parameters affect the final image.

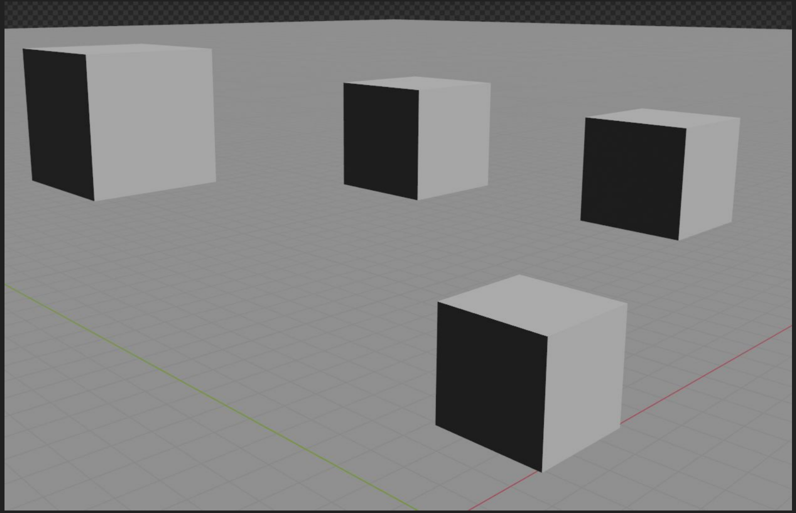

Light | Light
Light
Preview
Light
Point | Sun | Spot | Area
Color
Power 1000 W
Specular 1.00
Radius 0.1 m
Custom Distance
Shadow
Clip Start 0.05 m
Bias 1.000
Contact Shadows
Distance 0.2 m
Bias 0.030
Thickness 0.2 m
Custom Properties

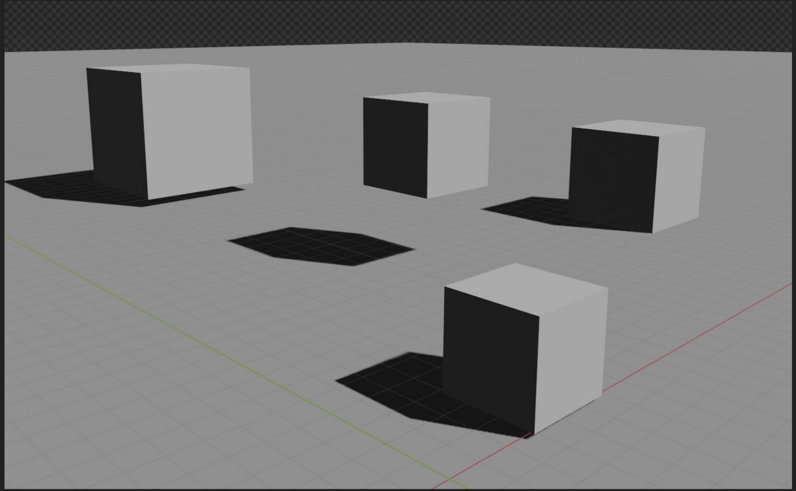
Shadows

Shadows are very important in a scene because, besides making the render more realistic, without a shadow objects seem to be floating over the floor and it is hard to know its relative size and distance.

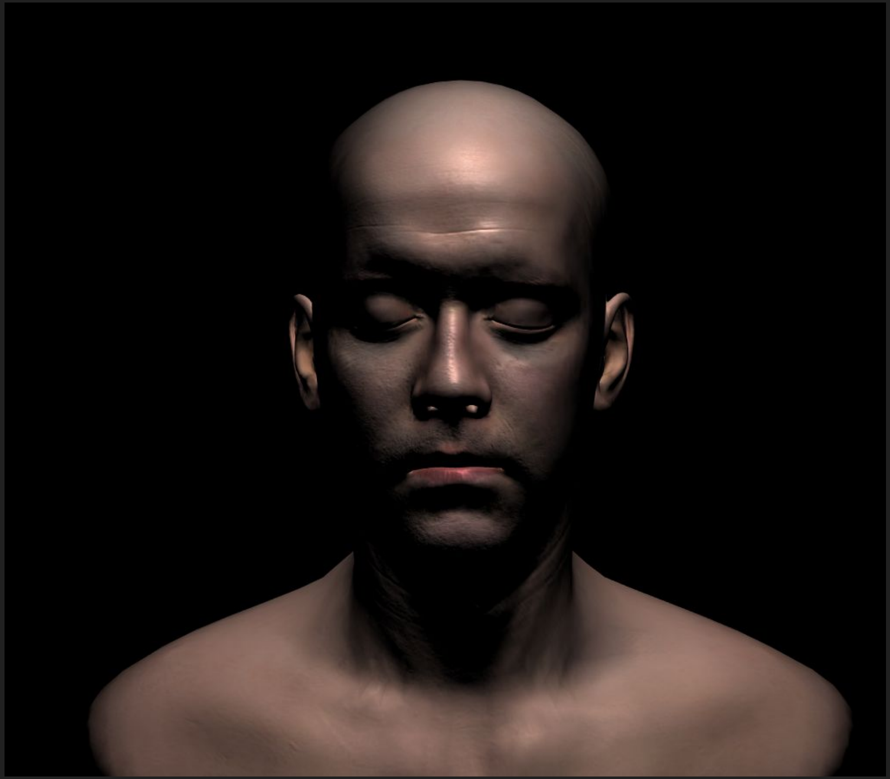
Also shadows can create very dramatic effects in scenes with low ambient light.

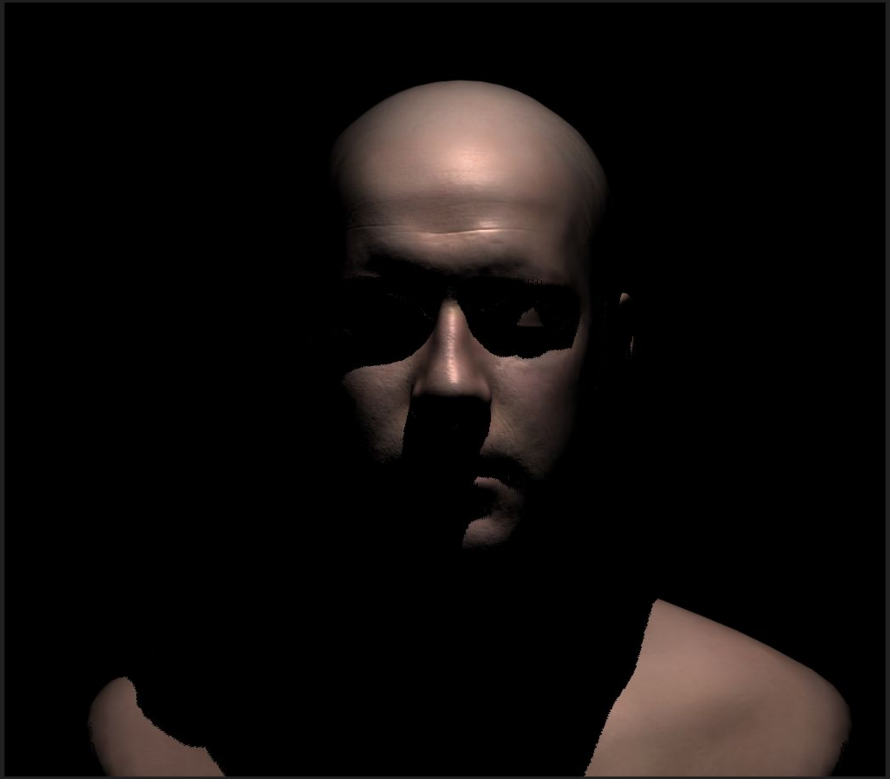
Sadly coding shadows using the GPU is not very straightforward.





Shadows to improve dramatism


Only Phong

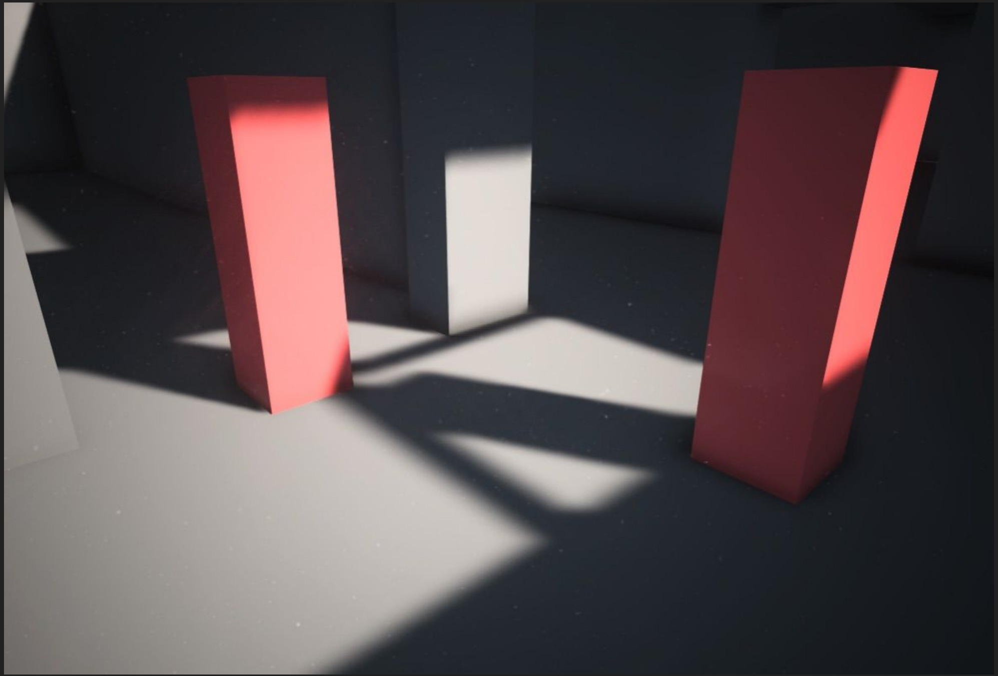

Phong + Shadows

Penumbra

It will also be interesting if we could compute the amount of penumbra in our shadows.

Shadows tend to look blurred the further they are from the light source (relative to the light source size).

This give us more information about the depth of the scene.

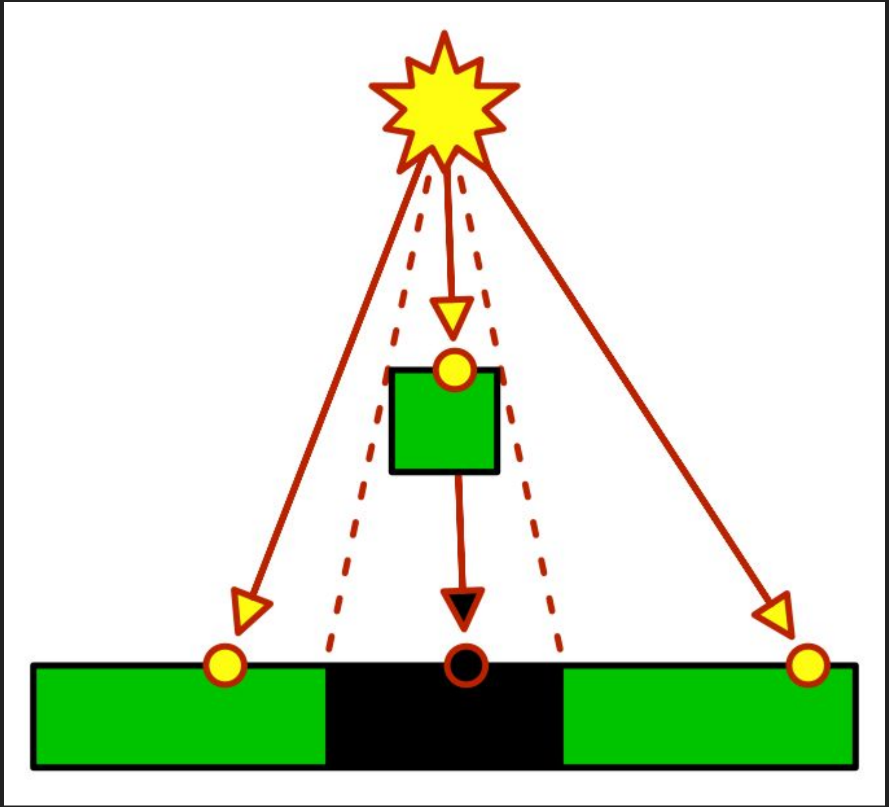


Light occlusion

What we need is to have a way from our shader to know if there is any object between the rendered point in world space and the light position.

Let's call it a shadow_factor, that tells how much of the original light emitted from the light source reached our pixel.

Then we multiply the final light contribution by this factor.

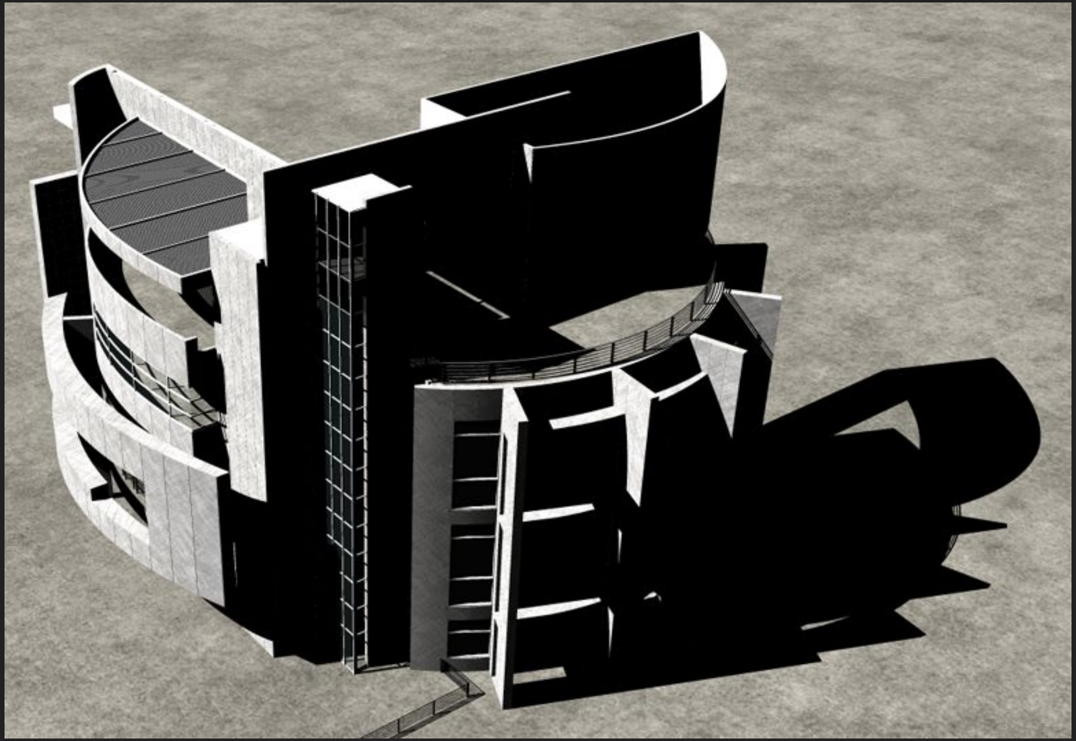


Raytraced shadows

When using ray-tracers shadows are a problem very easy to solve. We cast a ray from the shaded point to the light position and check if that ray collides with any object of our scene.

This give a very accurate and sharp shadow.

But requires a complex scene representation in memory and lots of resources to check intersections with all triangles.



Raster and Scene

When working with raster the problem is that GPUs work by drawing triangles one by one as we request. GPUs don’t store information about the structure of the scene, they only store the rendered framebuffer (and depth-buffer) from our point of view.

When we are rendering there is no way for the GPU to know if you are going to render another triangle later that could occlude the light of the current pixel.

So throwing rays into the scene is not supported unless we construct in VRAM a representation of the scene.

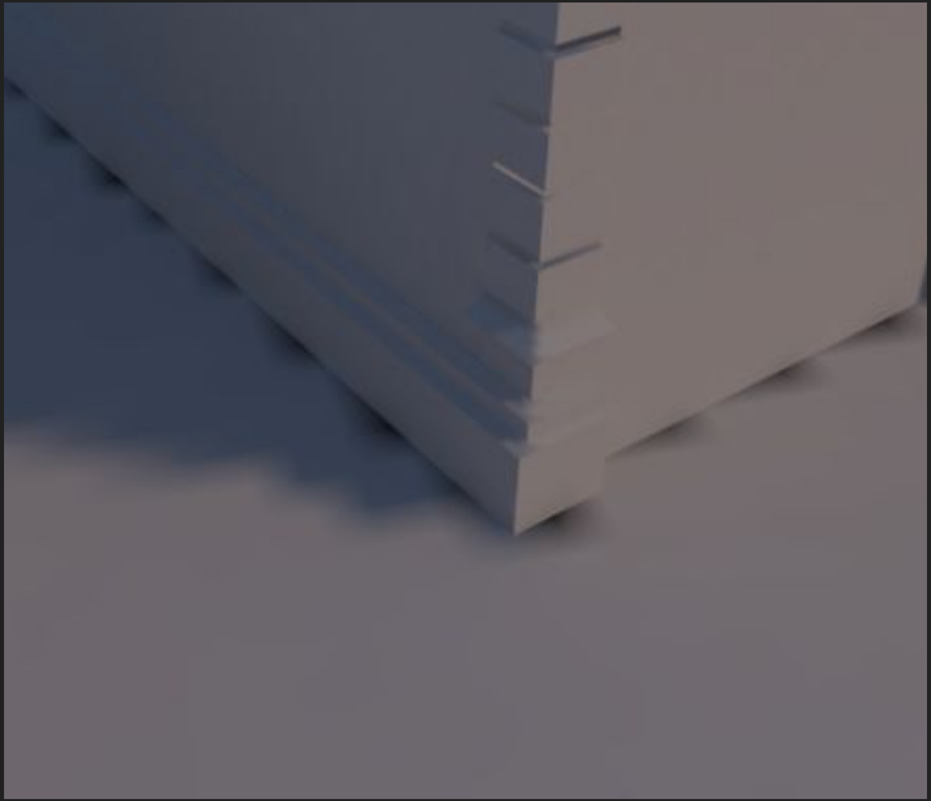
Baked shadows

As it mentioned in previous classes, for many years games relied in lightmaps (precomputed textures that contain the illumination information for the scene).

This textures were computed using ray-tracing methods that could take several minutes, and were stored in disk.

The problem with baked illumination is that if an object moves it won’t cast shadows.

Also if there is not enough resolution the shadows look pixelated, and you need to have special texture coordinates for every object so all the surface has its own part of the texture.

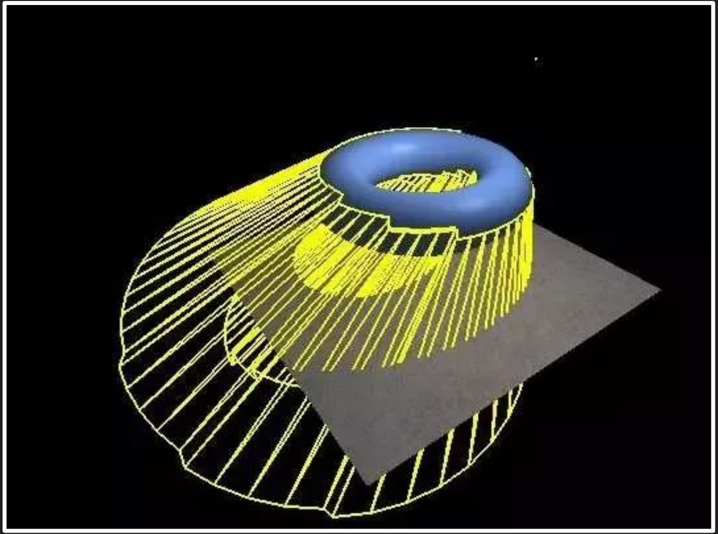


Stencil Shadows

The first real-time solution for shadows using the GPU came at the beginning of 2000s with the introduction of stencil shadows (also known as shadow volume).

The idea of the algorithm was to create a mesh from the CPU extruding the edges of the mesh seen from the light position, and render this mesh using the stencil buffer to mark areas inside this projected volume.

It was tricky to code and not very GPU friendly but impressive at the time.

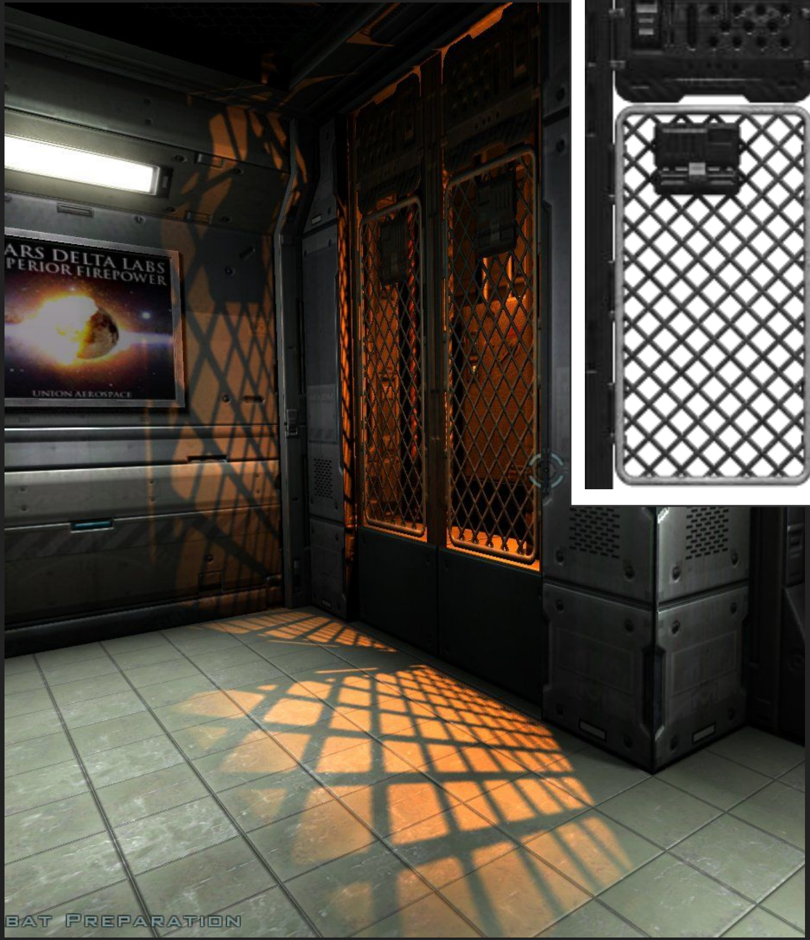


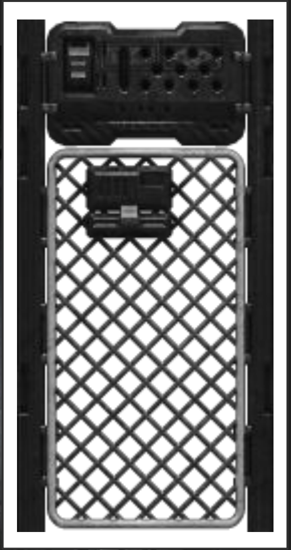
Fake Shadows

Another trick used by games is to fake shadows by projecting an image over the scene.

This is easy to do while computing the pixel illumination by projecting the pixel 3D position into a plane and using its coordinates as UVs to fetch from a texture.

It wont react to changes in the scene (occlusion by objects) but it is a cheap solution to give dramatism to scenes.





Rethinking the problem

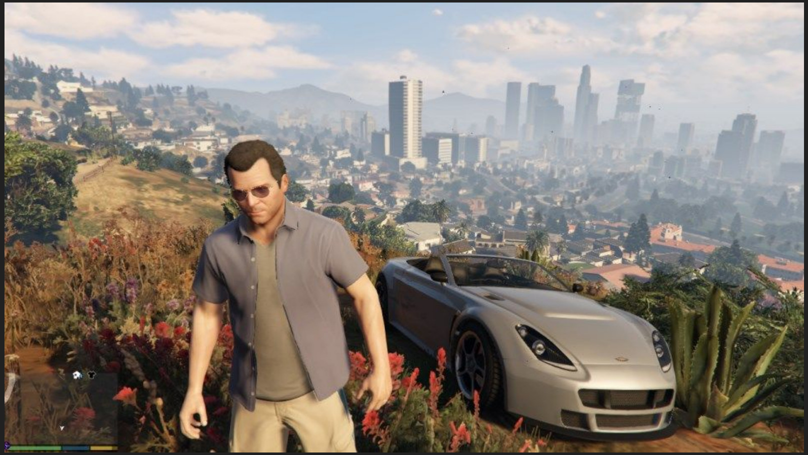
But we want to have true real-time shadows using the GPU, so let's rephrase the problem again.

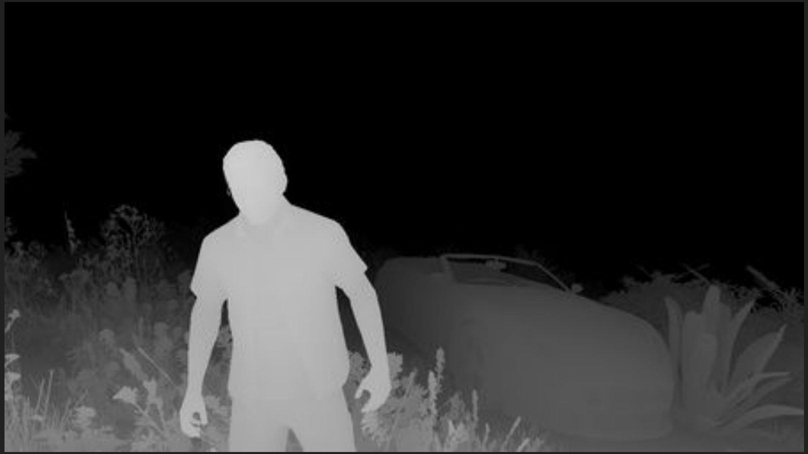
We want to have a way from the shader to query the GPU to know if there is another object closer to the light that the point we are illuminating.

On the other hand, we know that shaders are good reading from textures.

If there was a way to have a texture with the distance from every point to the light… It turns out GPUs have an efficient way to store distances. It is called the depth buffer, the same depth buffer we use to compute occlusions when rendering our scene.

Let's see how can we put all together.





Shadow mapping

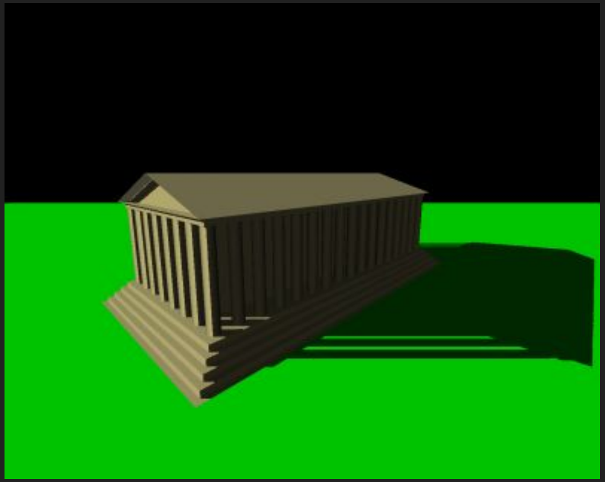
Shadow mapping

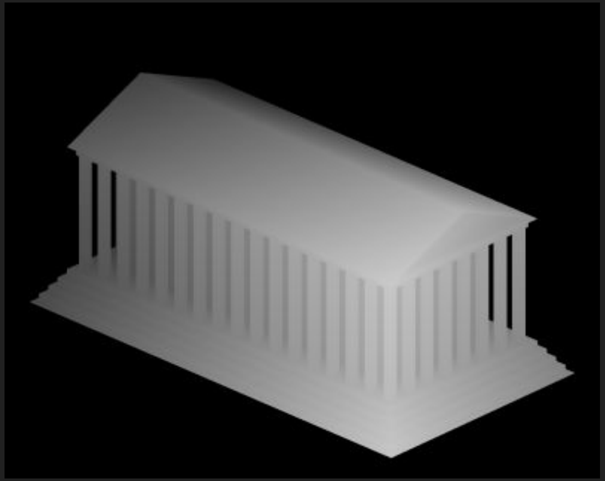
The concept of shadow mapping was introduced already in the 70s although it took almost two decades to have a working realtime version.

The idea is to store in a texture the distance from the light to the scene and use it to compute occlusions.

For that purpose we need to create a Shadowmap using our rasterizer.

Let's see what a shadowmap is and how can we create them.





Shadowmaps

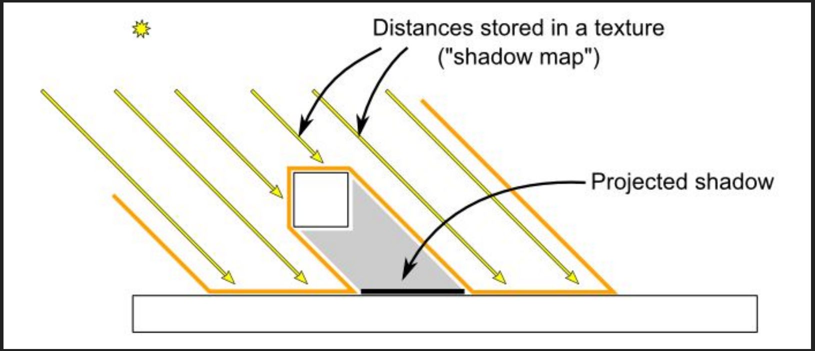
A shadowmap is a texture that stores the closest distance to the light position for an area of the scene.

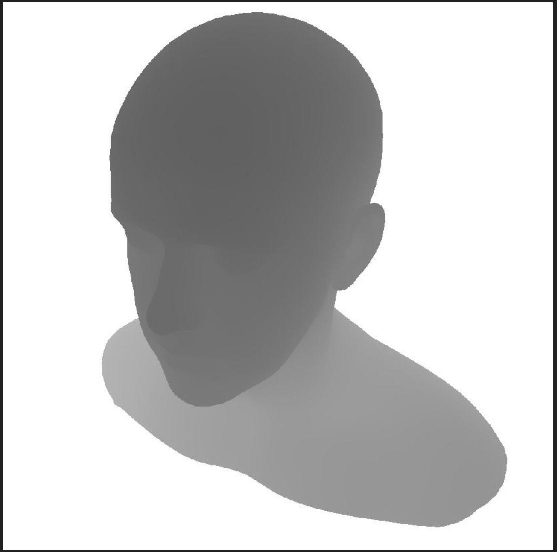
It is computed using the GPU by rendering the whole scene from the point of view of the light, and stored as a regular depth-buffer but inside a texture.

Because it is a texture it can be passed to the shader during the illumination pass and used to see if a given position in our 3D scene was occluded by another object.

Fetching the value is very cheap (one texture fetch).

One way to think of shadow maps is as a raytrace cache. You want to cache the results of the "is this point occluded" query, and the way you do it is by building a shadow map from the light's point of view.





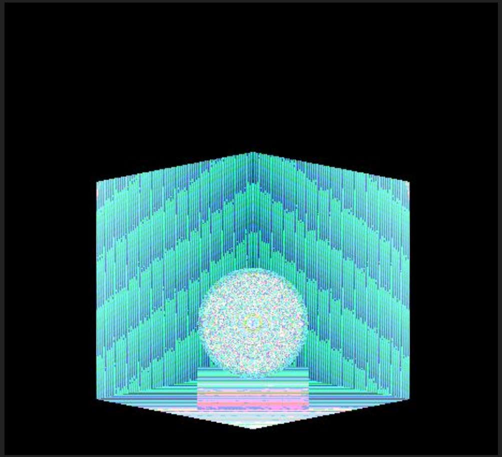
Shadowmap texture

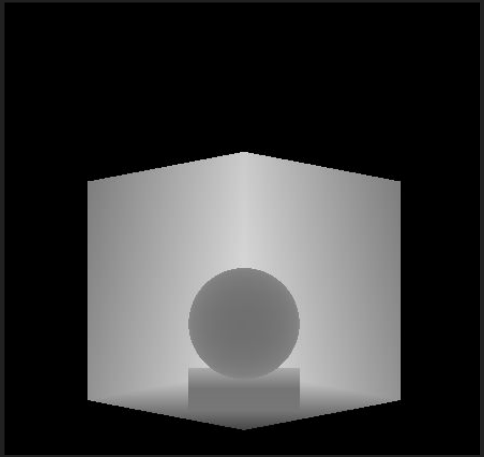
To store the shadowmap we can use any sort of texture, as long as we can store the distance for every pixel.

We could use RGBA and encode the distance in the four channels (32bits in total) using a special fragment shader, or we could use a single channel texture of type Float32 (which are not always supported).

But usually the easiest way is to use a DEPTH texture as they are more fit for this task. DEPTH textures are usually of 24bits but they could be of 32bits.

For the dimensions of the texture, that's our decision but it is good to work with square textures as they cover a more regular area of our scene.





Encoding depth as color

In case we want to encode the depth as a color, here is a function that separates the depth in 4 values so it can be stored as RGBA.

And the inverse function.

Remember that you do not need to do this unless your platform does not support rendering to depth textures.

```
//coverts a value from [0..1] into RGBA
vec4 PackDepth32( in float depth )
{
    depth *= (256.0*256.0*256.0 - 1.0) / (256.0*256.0*256.0);
    vec4 encode = fract( depth * vec4(1.0, 256.0, 256.0*256.0, 256.0*256.0*256.0) );
    return vec4( encode.xyz - encode.yzw / 256.0, encode.w ) + 1.0/512.0;
}

//coverts a RGBA into a float [0..1]
float UnpackDepth32( in vec4 pack )
{
    float depth = dot( pack, 1.0/vec4(1.0, 256.0, 256.0*256.0, 256.0*256.0*256.0) );
    return depth * (256.0*256.0*256.0) / (256.0*256.0*256.0 - 1.0);
}
```

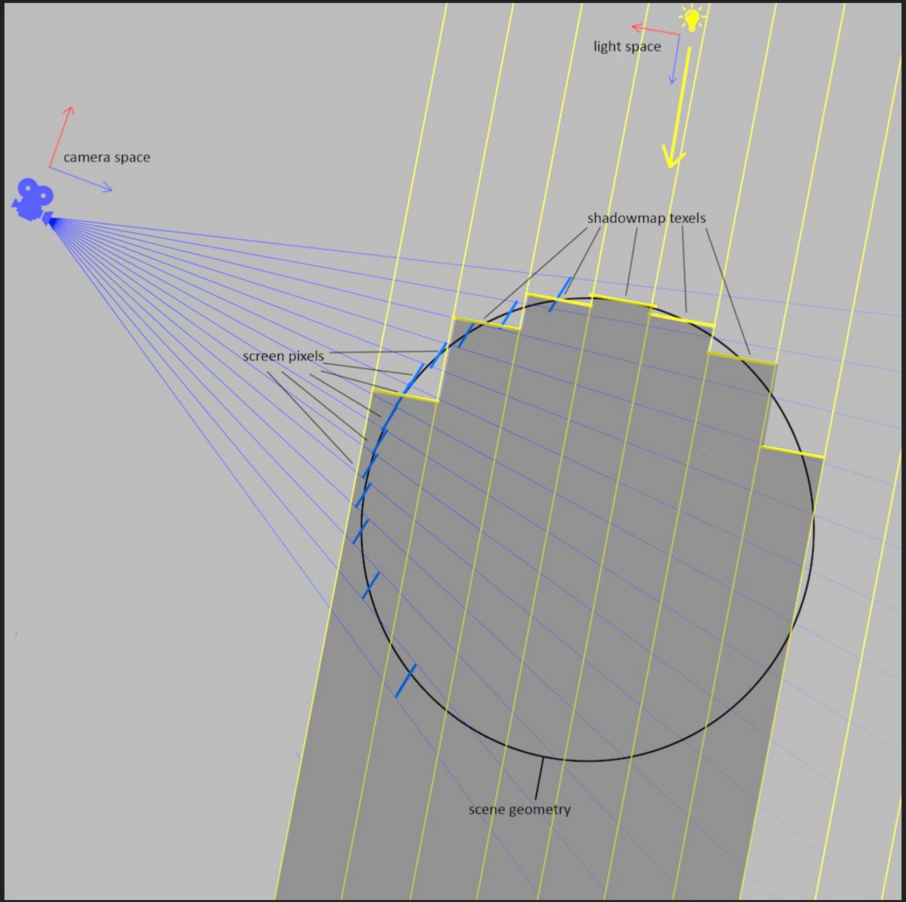
Source

Shadow Texels

It is important to remark that every pixel of the shadowmap represents a finite area of our scene.

Anything smaller than that area won't have an accurate representation inside our shadowmap.

We can increase it by increasing the resolution of the shadowmap but at the cost of more memory and more computation time as more pixels need to be filled.



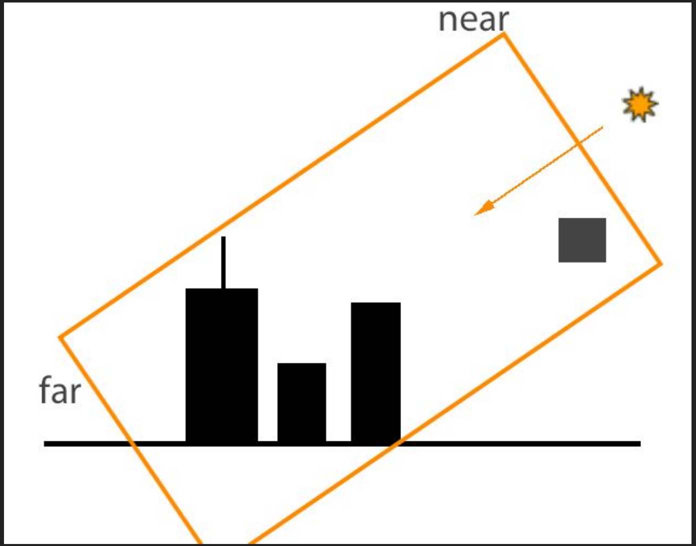
Shadow view

To compute the shadowmap camera for directional and spot light is easy, we can use the regular methods from our camera class.

We use lookAt to construct the view matrix, passing the light_pos and the light_target (and some up) to position in the right place.

For the projection matrix, in the case of spotlights we will use the setPerspective method (taking into account the spot angle as FOV). For directional lights we will use the setOrthographic method, passing a factor related to the amount of world we want to cover, the bigger the factor more area will cover but with less precision.

We try to set as near and far distance something that matches our scene boundary.



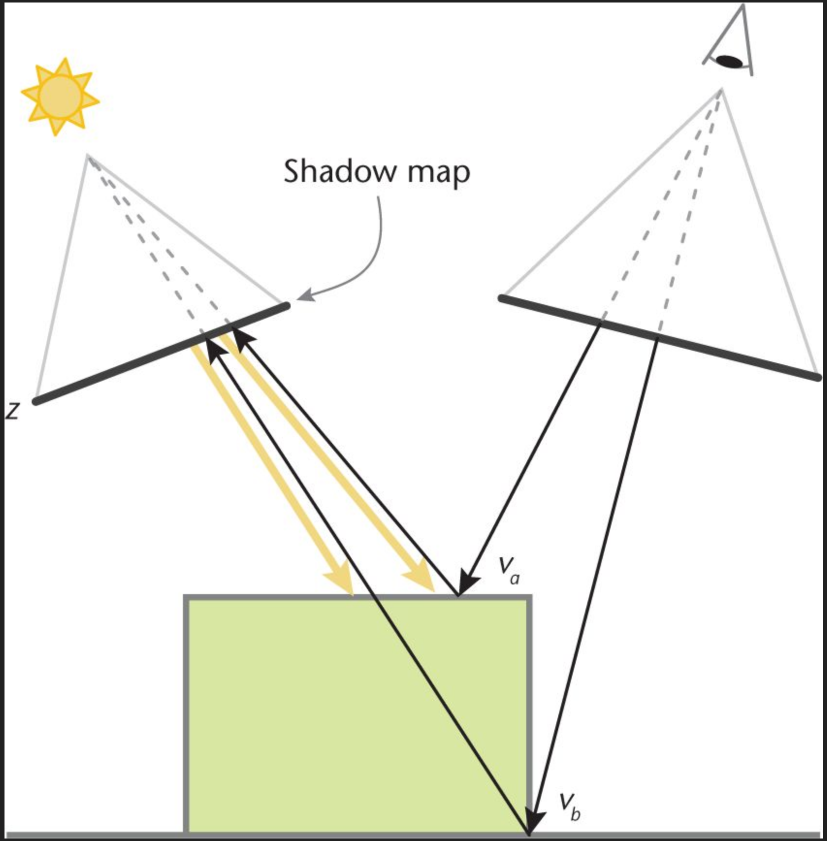
```
//example, assuming the light is not
vertical
camera->lookAt(
        ligh->position,
        light->position + light->light_dir,
        Vector3(0,1,0));
```

Reprojecting to shadowmap

Once the shadowmap is generated, and during the illumination pass, given a 3D position, we need to know in which position of the shadowmap this point will lay.

To do so we must reproject this 3D position using the same viewprojection matrix used to generate the shadowmaps. This will give us a coordinate in clip-space [-1..+1] that we must convert to UV space [0..1] to fetch the depth value from the shadowmap.

So we need to pass to the shader not only the camera viewprojection matrix, but also the shadowmap camera viewprojection matrix.



Comparing depths

Once we reproject the 3D position using the shadowmap view_projection matrix we can fetch inside the depth buffer the depth and compare it with the relative depth to the light source.

We do not compare using the distance because the depth buffer does not store distances, it stores a normalized value between near and far.

Also remember that when projecting a point using a projection matrix the result is in homogeneous coordinates, and we need to divide by the resulting .w to pass to clip-space coordinates.

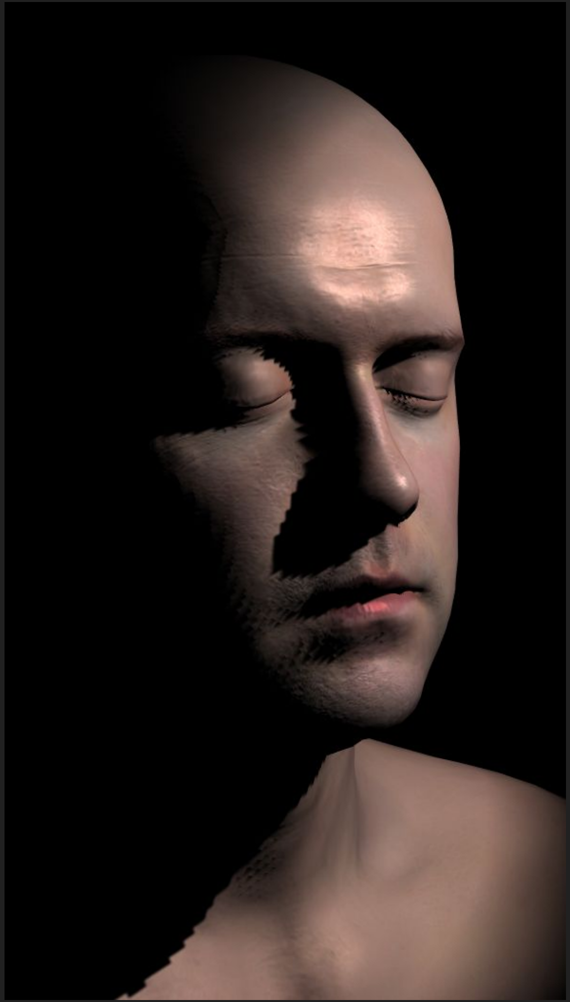
```
//project our 3D position to the shadowmap
vec4 proj_pos = shadow_viewproj * pos;

//from homogeneous space to clip space
vec2 shadow_uv = proj_pos.xy / proj_pos.w;

//from clip space to uv space
shadow_uv = shadow_uv * 0.5 + vec2(0.5);

//get point depth (from -1 to 1)
float real_depth = proj_pos.z / proj_pos.w;

//normalize from [-1..+1] to [0..+1]
real_depth = real_depth * 0.5 + 0.5;

//read depth from depth buffer in [0..+1]
float shadow_depth = texture( shadowmap,
                              shadow_uv).x;

//compute final shadow factor by comparing
float shadow_factor = 1.0;
if( shadow_depth < real_depth )
     shadow_factor = 0.0;
```

Problems

There are several problems using this approach:

- We need to render the whole scene one extra time for every light, that could have a huge performance cost in complex scenes.
- We can solve the shadowmap with one render pass for directional and for spot lights, but for point lights we can't just do a single render.
- Textures have a limited resolution, what happens when there is not enough pixel density per unit of our scene space?
- In directional lights, how can we cover the whole world?
- Textures also have a limited precision per pixel component, what if we do not have enough bits?
- Why store the same resolution for areas that are far away from the camera where it won't be seen?

We will analyze this problems one by one but first let's check how can we code it using the GPU.



Generating the shadowmaps

Rendering the whole scene from every light view require lots of resources. Games usually use lots of tricks when rendering the shadowmaps:

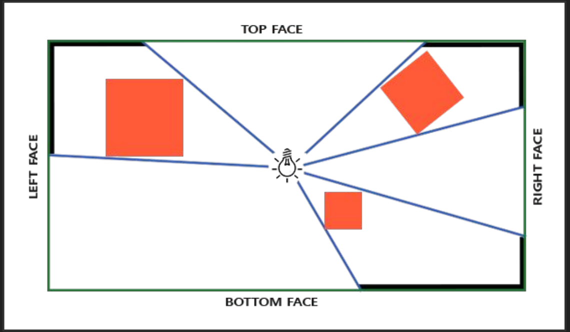
- Only update shadowmaps of lights that are inside the camera view
- Set camera far as max dist of light or lower.
- Only update when something changed inside
- Avoid reupdating the shadowmap every frame (maybe precompute them on startup)
- Skip small objects (they won't cast shadows)
- Use simplified versions of the meshes
- Use very simple shader

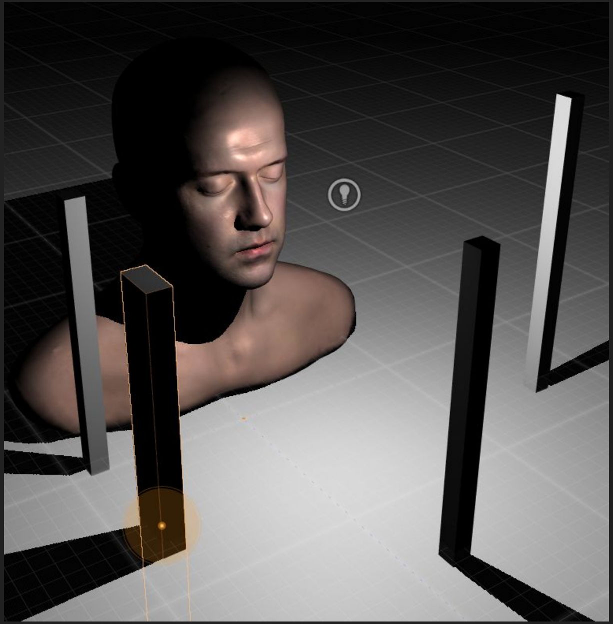
And any trick that make sense.

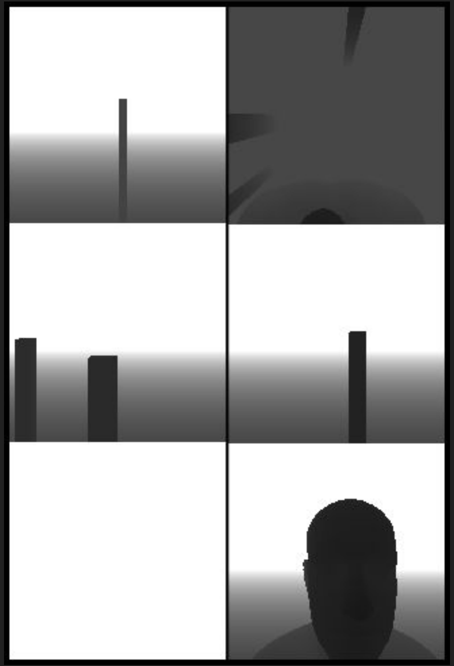
Point lights

For point lights it is more tricky as light travels in any direction and we cannot do a simple render to capture depth information in all directions (that would require a special spherical rendering method that GPUs do not support).

One solution is to do six render passes and store them as a cubemap, but this is very costly.







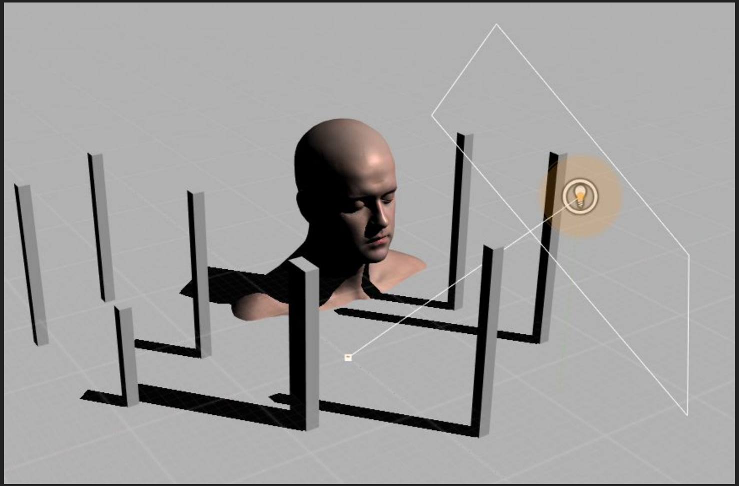
Resolution problems

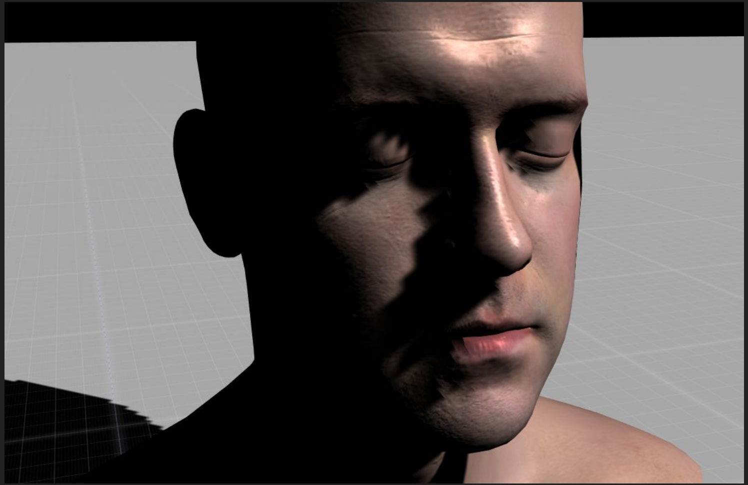
And when we store information in a texture there is a resolution limit.

The wider is the camera view to cover more area or our world, less pixel density in our shadowmap per world unit, which means we don't have much precision when fetching depth in our shadowmap.

This can produce pixelated shadows.

And increasing the resolution of our shadowmap could have a big impact in performance and memory consumption.





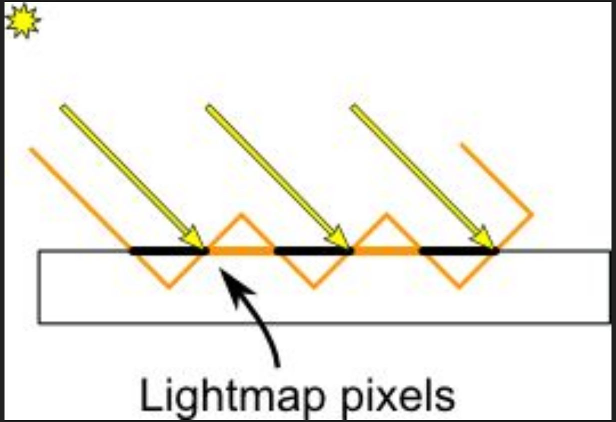
Precision problems

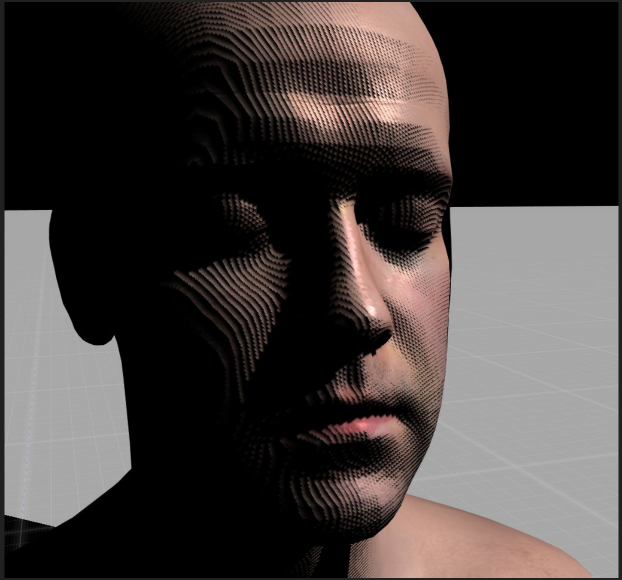
Besides having pixelated shadows, there is also another problem due to the limited shadowmap resolution and precision (as depth is stored using a 24bits per pixel).

Values that are stored in the depthbuffer will be quantized to the nearest precision, and that introduces an error in our shadowmap on top of the resolution error.

This could result in strange artifacts called shadow acne.

There are ways to paliate this problem but it is always present in some degree.





Shadowmap constant texel density

Another problem is that every pixel in a shadow map covers roughly the same volume of our scene.

That is not ideal as we are wasting resources if areas that are far away in our view store the same amount of data in the shadowmap that areas that are close to the camera.

It would be nice to have more resolution in our shadowmap the closer we are to the camera, and less the farther it is.

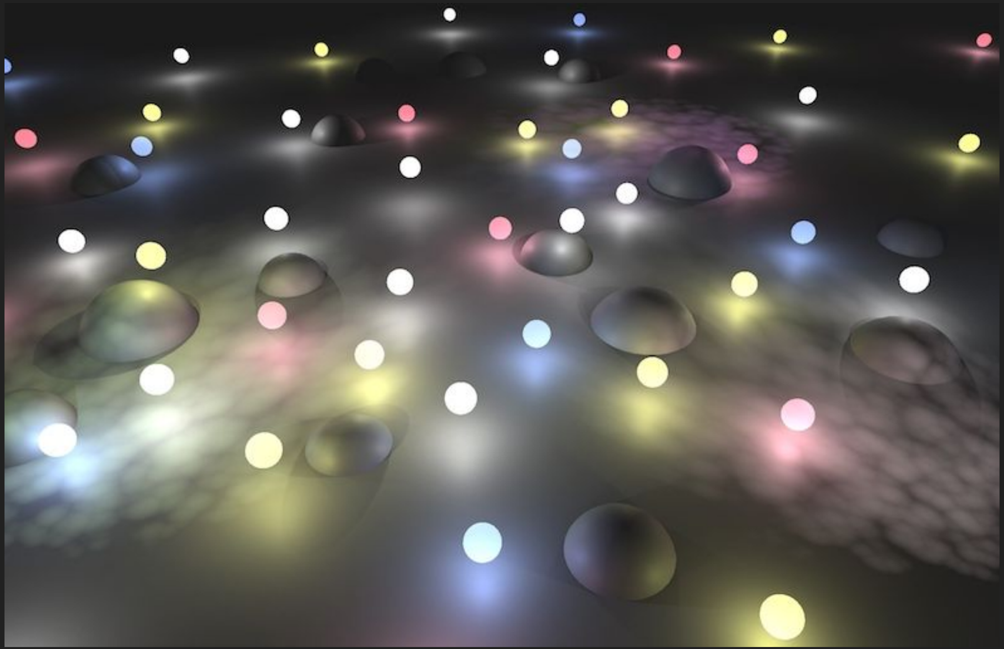
Sadly GPUs do not support variable resolution density textures, and although there are some tricks [to distort the projection] that work in some situations, we need to find a more generic solution.

Shader Texture limit

GPUs also have a max number of textures that can be binded.

This is a problem when we have lots of lights in our scene using shadowmaps.

We need a way to avoid this limitation.



Shadows in the GPU

Depth Buffer Precision

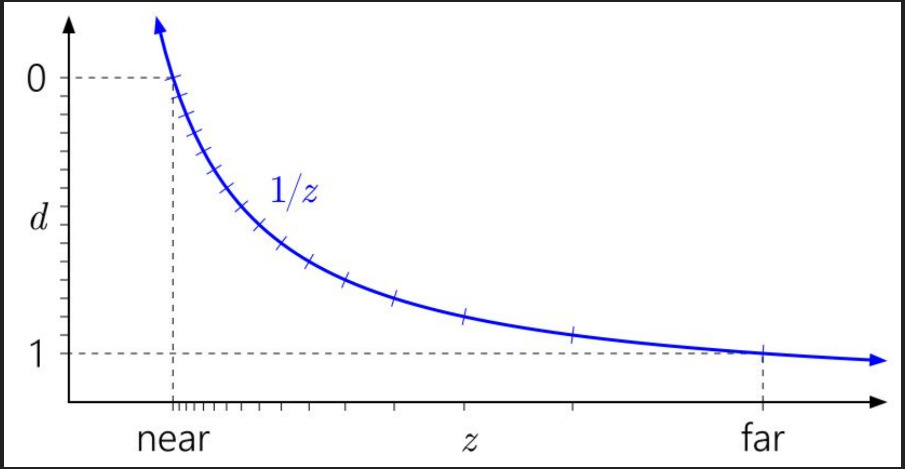
GPUs store the depth of every pixel using a 24 bits value.

But the bits are not always distributed evenly among the distance range.

In perspective projections GPUs store 1/z, which means more precision in the areas closer to the camera, and less precision in the areas far from the camera.

It is important to take this into account because when manipulating or visualizing depth values we cannot assume depth is linear.

Article about depth precision



```
//you can linearize any Depth value of a perspective
projection with this formula (z from -1..+1)
linear_z = n * (z + 1.0) / (f + n - z * (f - n));

n = near_plane distance
f = far_plane distance
z = depth value from the depth map * 2 - 1
```

Render to a Depth Buffer

The depth buffer cannot be accessed directly from our shader unless it has been stored inside a texture.

Gladly GPUs allow to do that using Frame Buffer Objects. A FBO allows to render inside a texture.

Before rendering we must set up our FBO and enable it. We clean only the Depth Buffer as there is no need to use the Color Buffer.

We can disable writing to the color as it wont be used, this speeds up a little the rendering.

After rendering the whole scene we disable it to render back to the screen.

```
//first time we create the FBO
if (!light->shadow_fbo)
{
	light->shadow_fbo = new FBO();
	light->shadow_fbo->setDepthOnly(1024, 1024);
}

//enable it to render inside the texture
light->shadow_fbo->bind();

//you can disable writing to the color buffer to
speed up the rendering as we do not need it
glColorMask(false,false,false,false);

//clear the depth buffer only (don't care of color)
glClear( GL_DEPTH_BUFFER_BIT );

//whatever we render here will be stored inside a
texture, we don't need to do anything fanzy
//...

//disable it to render back to the screen
light->shadow_fbo->unbind();

//allow to render back to the color buffer
glColorMask(true,true,true,true);
```

Rendering the scene inside the shadowmap

When rendering the scene inside the shadowmap remember that we do not care about the color, so the fragment shader should be as simple as:

```
FragColor = vec4(1.0);
```

Also remember to skip all the multipass and blending. Shadowmaps do not care about the number of lights or transparencies. Remember to do not render the objects that shouldn't cast shadows (like transparent objects).

We just want to render a flat mesh per object that cast shadows so we can get the depth of every pixel.

I use a rendering_shadowmap flag to control this special case in my rendering pipeline. Also check

Reading the Shadowmap

Once the shadowmap has been created, we can continue rendering the final pass as we did before.

But now for lights with shadows enabled, we will also pass to the shader the texture and the shadowmap viewprojection matrix as any other relevant information.

```
//get the depth texture from the FBO
Texture* shadowmap = light->shadow_fbo->depth_texture;

//pass it to the shader in slot 8
shader->setTexture("shadowmap", shadowmap, 8 );

//also get the viewprojection from the light
Matrix44 shadow_proj = light->camera->viewprojection_matrix;

//pass it to the shader
shader->setUniform("u_shadow_viewproj", shadow_proj );
```

Shadow Acne and Bias

To solve the shadowmap acne we will displace the real_depth by a constant value.

We call this value the shadow_bias. It must be very small (0.0001) as depth goes from 0 to 1.

Keep in mind that we apply the shadow bias before dividing the real_depth by the W. We do this to avoid different bias depending on the near-far range of our scene.

```
//we will also need the shadow bias
shader->setUniform("u_shadow_bias", light->bias );



//applying it in the shader
float real_depth = (proj_pos.z - u_shadow_bias)
/ proj_pos.w;
```

Bias in linear space or non-linear space

About the bias, there is the question if we must convert the Z values of perspective projections to linear space (for our pixel and the depth stored inside the depth texture) before comparing it.

In both cases the result should be the same as the conversion does not change the sign, with one exception:

If we compare it in linear space, in further away pixels the difference could be smaller than the precision of the depth texture which would cause shadow acne.

So by comparing it in the non-linear space we are using a variable size bias, as the further it is the bigger it seem, which could make sense for perspective projections (spotlights).

Final shader

Now we can compute the shadow factor taking into account the info.

We could place all this code inside a function so we can fetch more than one sample to do an average.

```
//project our 3D position to the shadowmap
vec4 proj_pos = u_shadow_viewproj * vec4(pos,1.0);

//from homogeneus space to clip space
vec2 shadow_uv = proj_pos.xy / proj_pos.w;

//from clip space to uv space
shadow_uv = shadow_uv * 0.5 + vec2(0.5);

//get point depth [-1 .. +1] in non-linear space
float real_depth = (proj_pos.z - u_shadow_bias) /
proj_pos.w;

//normalize from [-1..+1] to [0..+1] still non-linear
real_depth = real_depth * 0.5 + 0.5;

//read depth from depth buffer in [0..+1] non-linear
float shadow_depth = texture( shadowmap, shadow_uv).x;

//compute final shadow factor by comparing
float shadow_factor = 1.0;

//we can compare them, even if they are not linear
if( shadow_depth < real_depth )
     shadow_factor = 0.0;
```

Special cases: outside of the shadowmap

What happens when the point we are evaluating its shadow factor falls outside of the shadowmap frustum? We do not have its depth.

We must decide if we want to be in darkness or not.

In the case of Directional we will set the shadow_factor to 1.0 (no shadow), in other light types we must set it to 0.0 (full shadow).

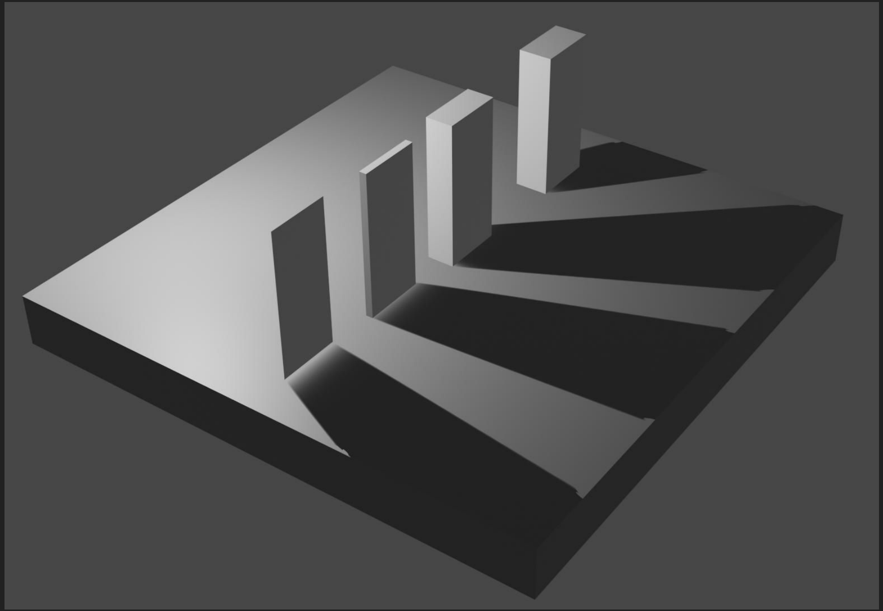
```
//for directional lights


//it is outside on the sides
if(   shadow_uv.x < 0.0 || shadow_uv.x > 1.0 ||
      shadow_uv.y < 0.0 || shadow_uv.y > 1.0)
            return 1.0;

//it is before near or behind far plane
if(real_depth < 0.0 || real_depth > 1.0)
      return 1.0;
```

Advanced Shadows

Light Bleeding

The problem is that if the depth in the shadowmap is too close to the opposite side, in could trespass due to the bias and produce bright corners in areas that shouldn't be bright (inside of rooms, etc).

This is called Light Bleeding or Light Leaking and happens if the walls are thinner than the bias.



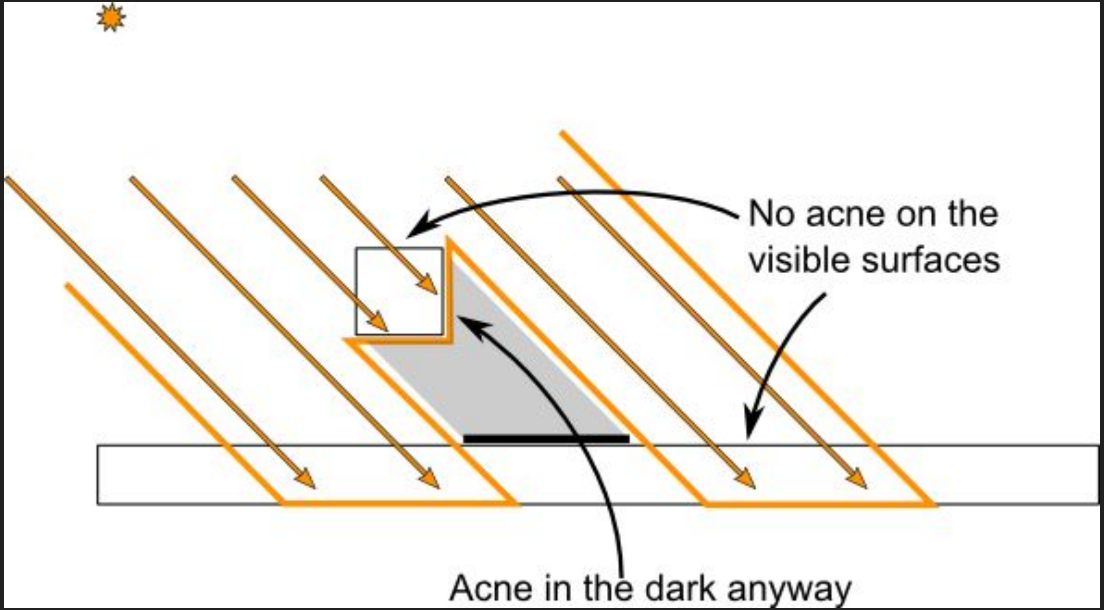
Culling the outside faces

One trick to reduce the shadow acne and the bleeding is if instead of rendering the front face triangles we render the back face (the ones not facing to camera).

This works because we do not care about shadows inside the objects.

This way the shadowmap won't have values too close to the surface that could produce artifacts when comparing the depths.

Remember to leave it back to its original value after generating the shadowmaps.



```
glEnable( GL_CULL_FACE );
glFrontFace( GL_CW ); //instead of GL_CCW
```

Precomputed Shadowmaps

We could generate all shadowmaps at the beginning of our application to avoid recomputing them every frame.

But that would imply that moving objects won't have shadows.

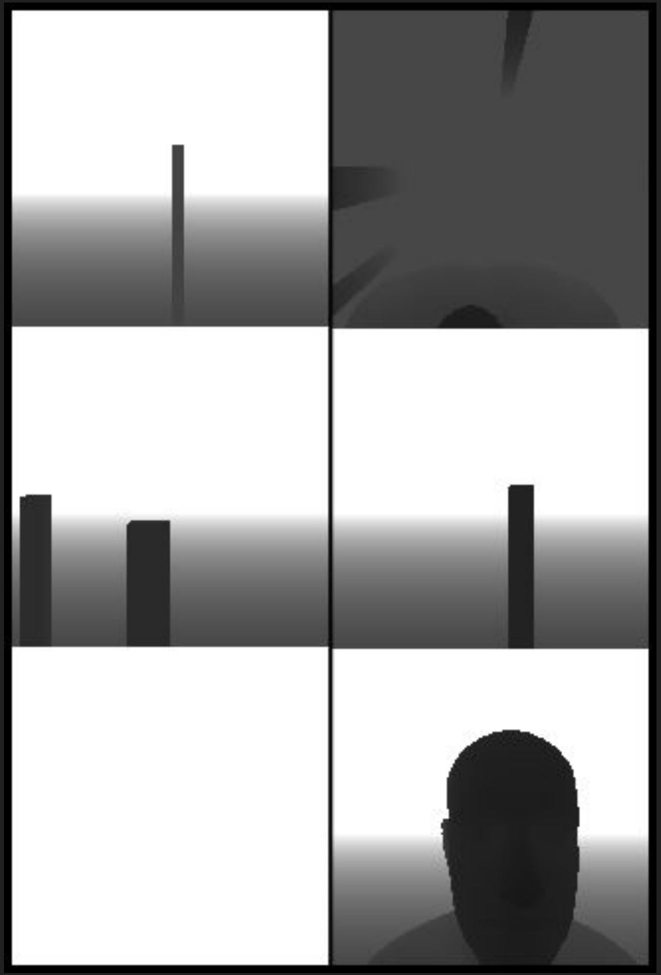
This can be fixed if instead of render the static and the dynamic objects in the shadowmap, we clear the shadowmap writing the content of a precomputed shadormap that contains the static objects and just render the dynamic ones.

Point light shadows

For point lights we can do the same process that we follow for spot and directional but store every side of the view in a different region of the texture to form six sides as in a cube.

Then from our shader depending on the direction we must compute to which of the six faces the vector collides and also compute the UV vector.

When rendering every face of the view, remember that the glClear method in openGL clears ALL the buffer, not just the area inside the selected viewport. You must do it once (or use glScissor)



Point lights depth comparison

The problem when using a multi view rendering is that we cannot project our pixel using the viewprojection as there are several.

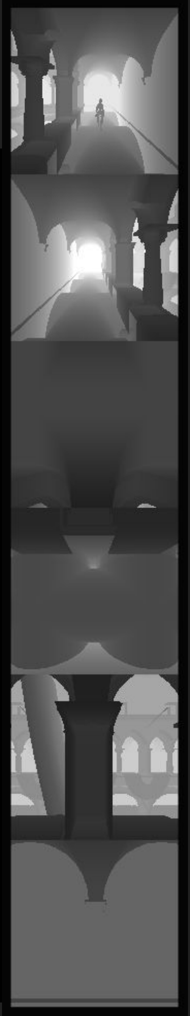
The best solution is to read from the texture and transform it to distance so we can compare it with the distance between the light and the point.

vec3 to tex2D

Here there is an example of how to convert a 3D vector to a 2D coordinate in a texture that contains the 6 sides of a cubemap arranged vertically inside a 2D texture, in the order defined by opengl for cubemaps which are:

- POSX
- NEGX
- POSY
- NEGY
- POSZ
- NEGZ

```
vec2 vec3ToCubemap2D( vec3 v )
{
	vec3 abs_ = abs(v);
	//Get the largest component
	float max_ = max(max(abs_.x, abs_.y), abs_.z);
	//1.0 for the largest component, 0.0 for the others
	vec3 weights = step(max_, abs_);
	//0 or 1
	float sign_ = dot(weights, sign(v)) * 0.5 + 0.5;
	float sc = dot(weights, mix(vec3(v.z,v.x,-v.x), vec3(-v.z,
v.x, v.x), sign_));
	float tc = dot(weights, mix(vec3(-v.y, -v.z, -v.y),
vec3(-v.y, v.z, -v.y), sign_));
	vec2 uv = (vec2(sc, tc) / max_) * 0.5 + 0.5;
	// Offset into the right region of the texture
	float offsetY = dot(weights, vec3(1.0, 3.0, 5.0)) - sign_;
	uv.y = (uv.y + offsetY) / 6.0;
	return uv;
}
```



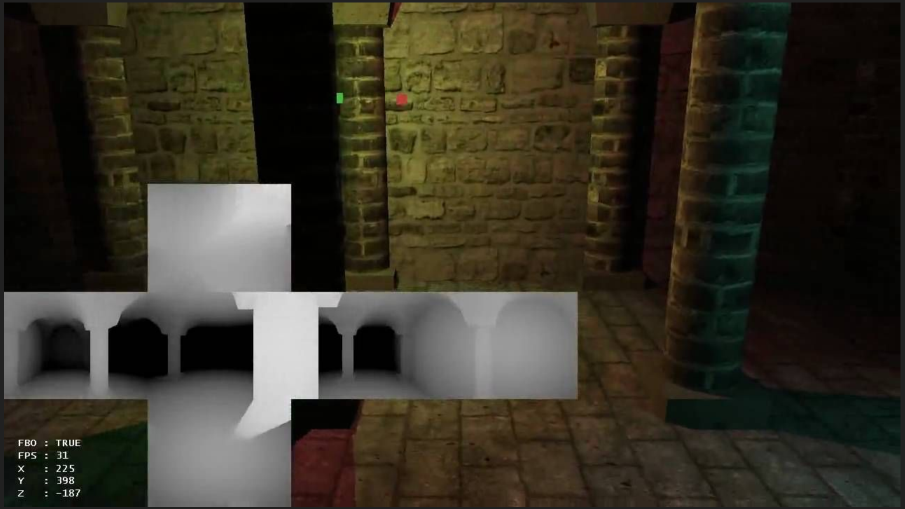
Shadow Cubemaps

We could also use directly a Cubemap to store the six sides of a point light shadowmap.

The only problem is that the shader must be different as cubemap textures use a different type of sampler (textureCube) than regular 2D textures (texture2D).

Also, some devices do not handle well rendering to a depth texture inside a cubemap.

But using cubemaps we can ignore the problem of converting our direction vector to appropiate face and uv as passsing the L vector is enough.



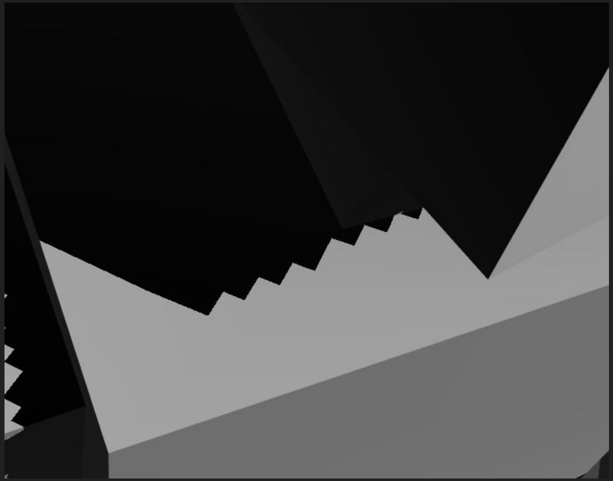
```
//we use the direction of the light as uv
float depth = textureCube( shadowmap, L ).x;
```

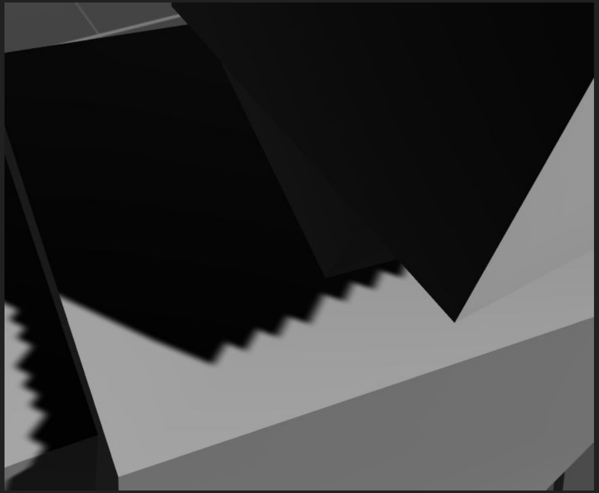
Bilinear Filtering

One problem with our current solution is that shadows do not transition smoothly.

One solution could be to compute the shadow value in the four pixels of the shadowmap and do a bilinear interpolation. This is not equivalent to do one single fetch with a depth that has been bilinear-interpolated (that would give still a hard shadow but more curvy).

GPUs have already a function to perform bilinear filtering for shadowsmaps using a different type of sampler called sampler2DShadow although it is not straightforward.





Percentage-Closer Filtering

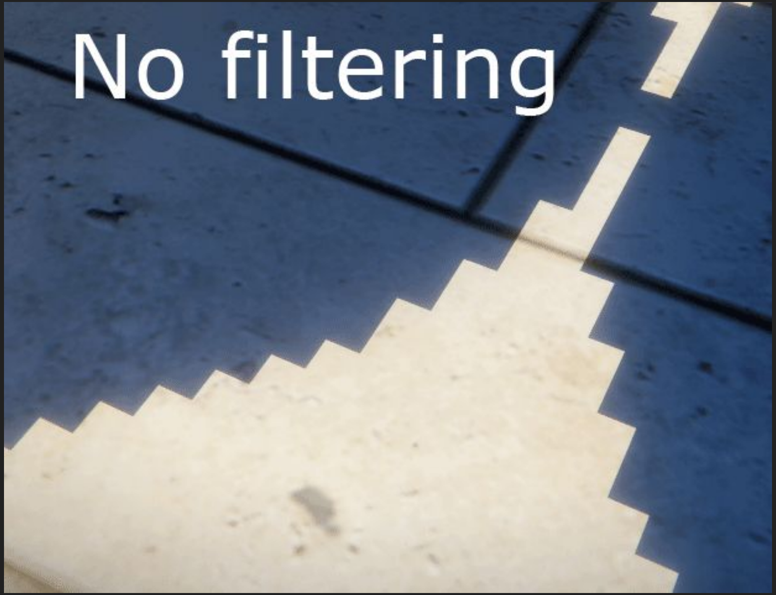
Another solution is to do several shadow computations with small offsets and do the average of the result.

This is called Percentage Close Filter (PCF) and it is easy to implement (although increases the shader work-load).

Just add to your testShadow function a vec2 parameter with an offset when reading the shadowmap pixel to allow to read nearby pixels, and do several calls to it with different offsets.

You can use a regular offset or use some noise.

Article



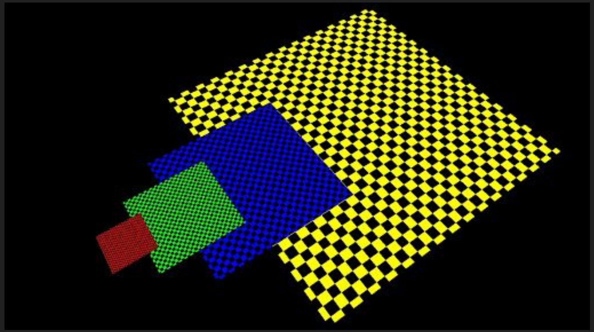
Cascaded Shadowmaps

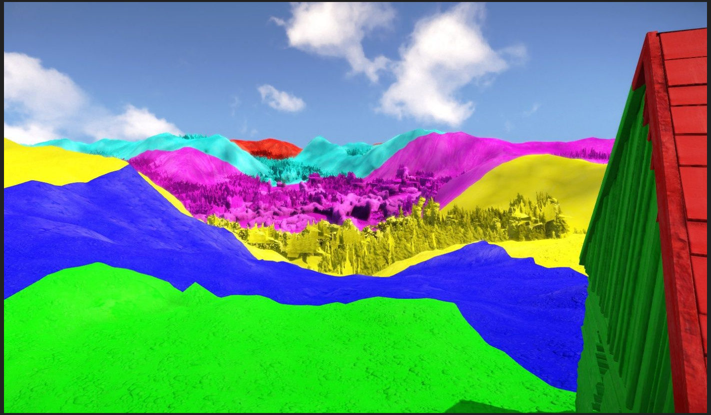
To solve the problem of having the same resolution for the whole shadowmap there is a very common trick: To use several shadowmaps, each with a bigger region than the previous one. They are called cascades.

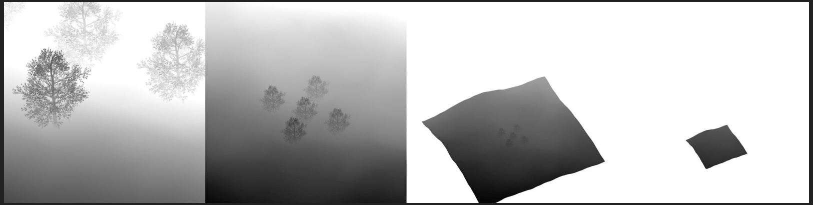
Then depending on the pixel we read from one shadowmap or the other.

This is easy to implement as we pass to the shader four shadowmaps (or one with all four packed in different areas) and four viewproj matrices and check if the points fall inside the smallest one, if not test the next one, and so.

More info here.





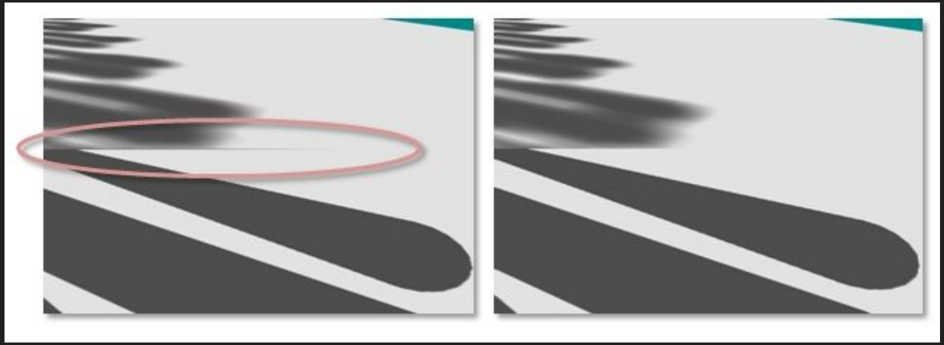


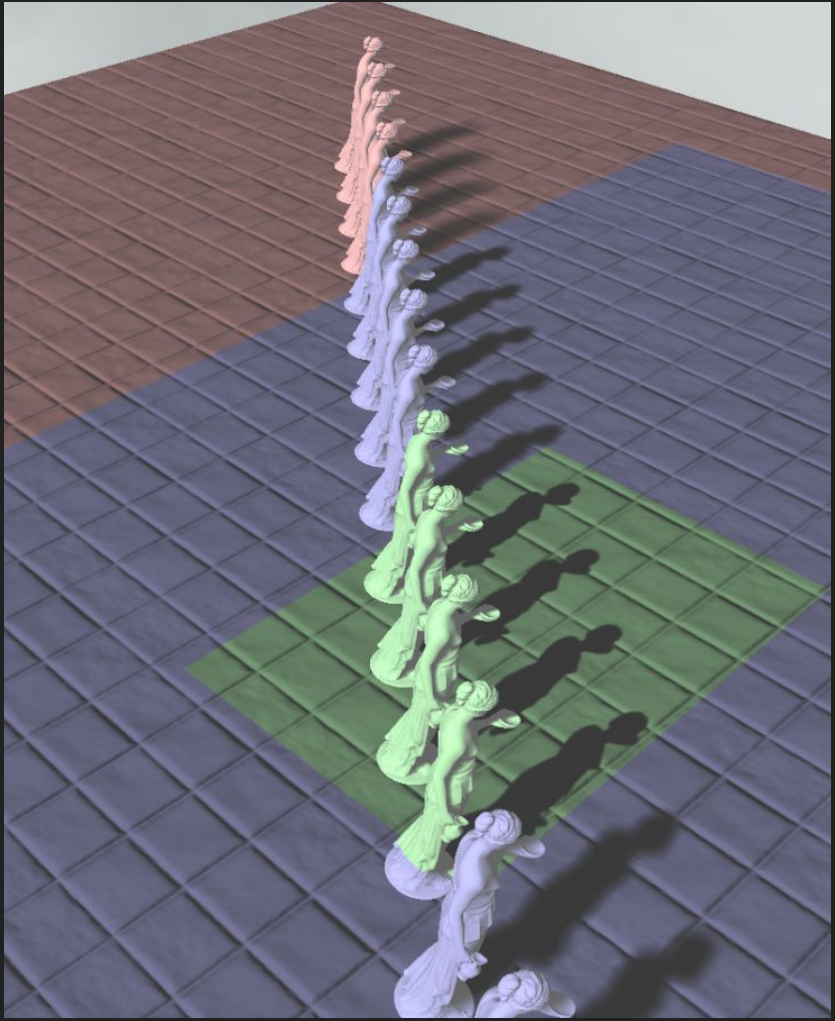
Cascades

Here you can see how the shadows look more sharp in the cascade that has more density of pixels per unit of space.

Also you can see how the seams between cascades are visible.

This can be mitigated using a linear interpolation between cascades.





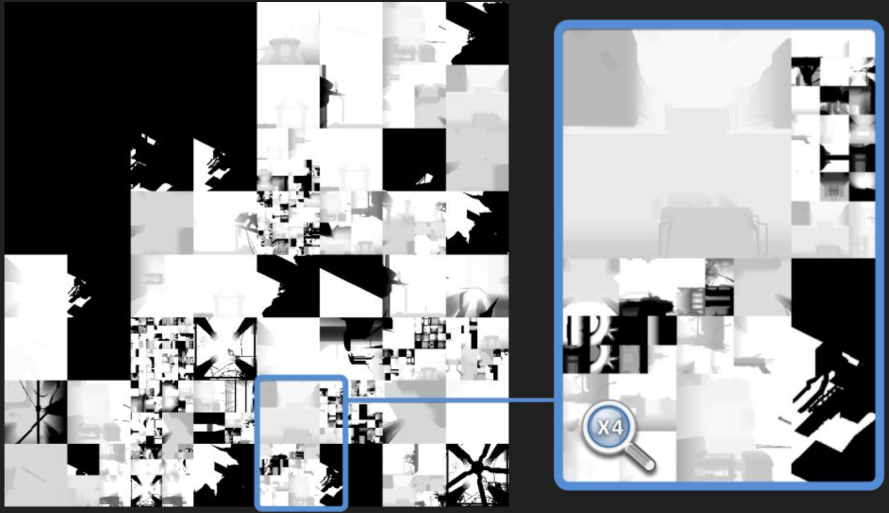
Packing Shadowmaps

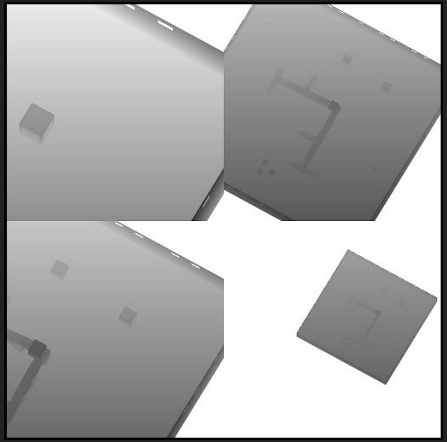
One common trick when using shadowmaps is to pack them all in a single texture (a shadow atlas).

We can pass to the shader the rectangle coordinates of that light inside the shadowmap texture and adapt the UVs according to it.

This works better as shaders have a limited amount of textures they can use, and uses a constant amount of memory for all shadowmaps.

To render inside an area of the shadowmap is as simple as changing the glViewport, but remember that glClear clears the whole buffer, not just the viewport area!





Paraboloid Shadowmaps

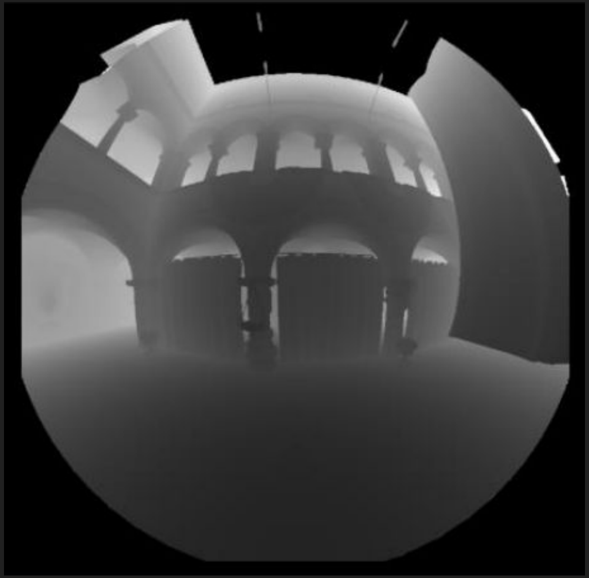
There is a way to render a view close to a semi-sphere in one single pass.

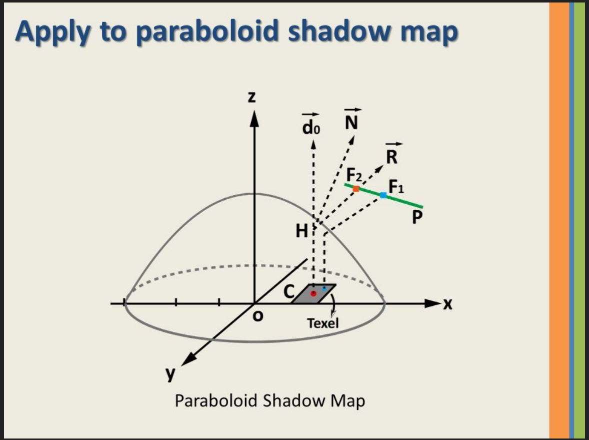
It implies of using the vertex shader to deform vertices as if they were in a curved space. It is called paraboloid rendering.

It is easy to implement as it is just a mathematical function applied to every vertex of the scene. Here is an implementation.

This could reduce point light shadowmaps to two passes (dual-paraboloid shadowmap) instead of six, and it is easy to pack both views in one single texture.

The only problem is that it produces some artifacts in some areas like the equator of the paraboloids, and in scenes with big triangles they can produce errors due to poor tessellation.



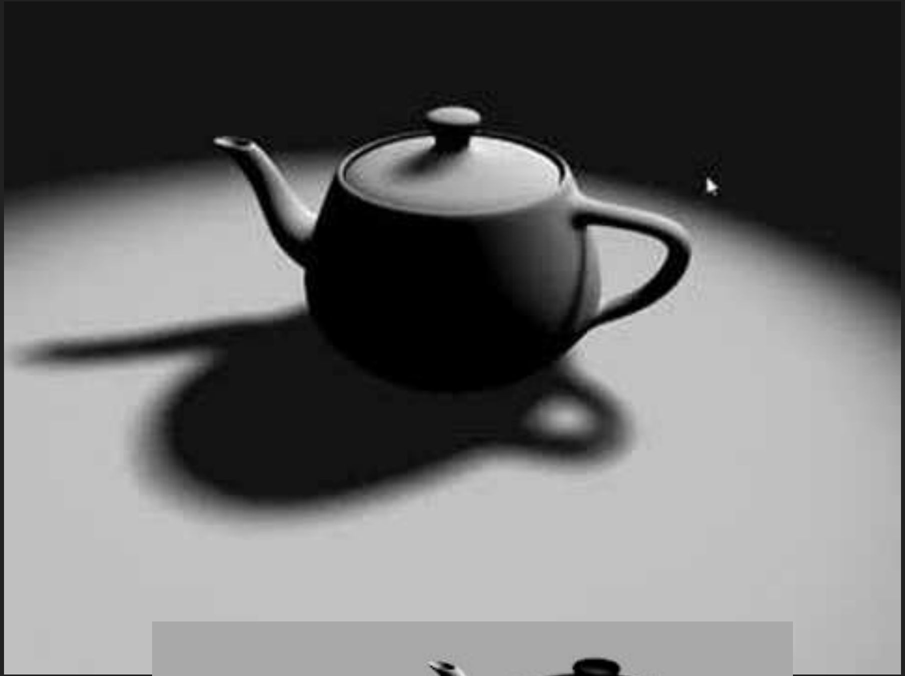


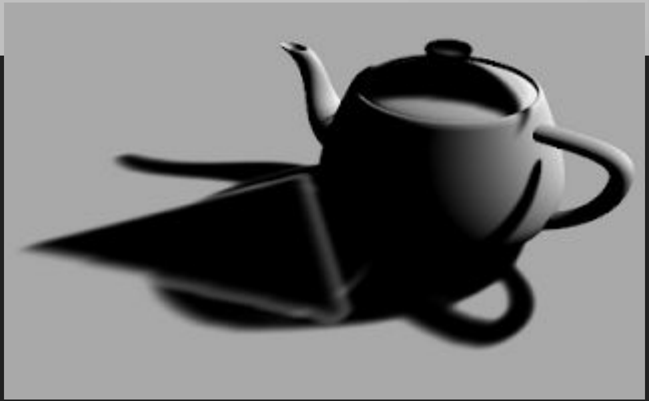
Variance Shadowmap

To obtain an smoother shadowmap there are more complex algorithms like Variance Shadowmaps that store not just the shadowmap but also a blurred version of it.

The way to compute the final shadow factor changes but the results are better.

There are some situation that could produce some ugly artifacts.



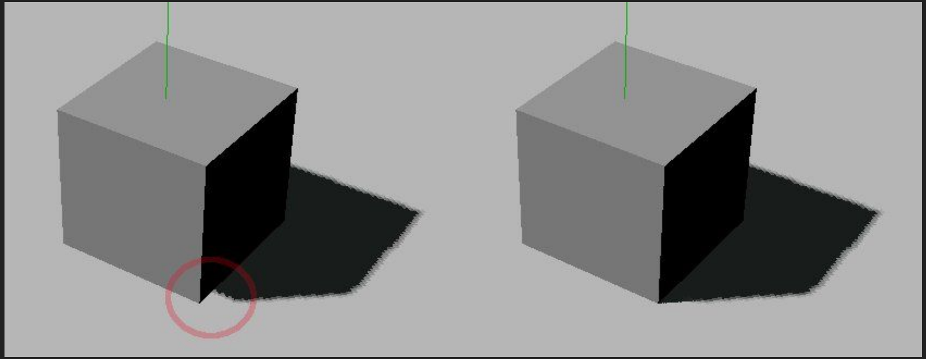


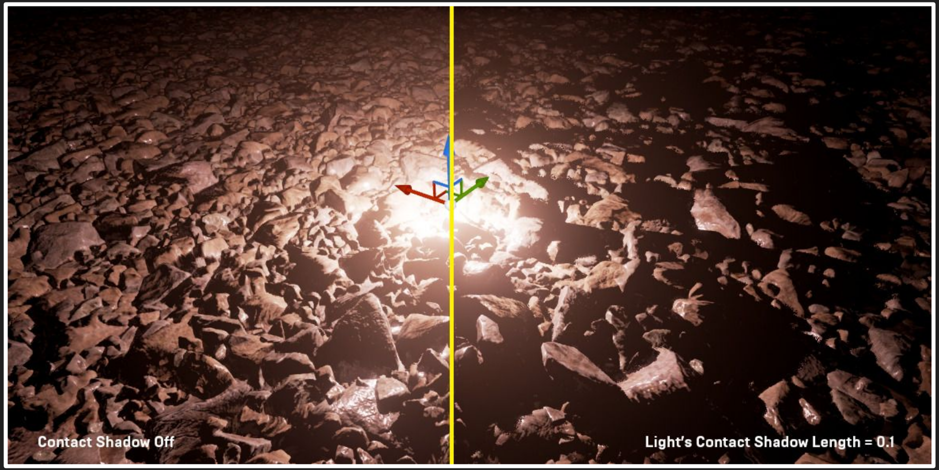
Contact shadows

As we saw, due to the bias and the shadowmap resolution quite often shadows are not touching the object where they should (an effect known as peterpaning). Also objects smaller than one texel of the shadowmap won't have shadows.

A solution used by recent engines is to use the scene depth-buffer (not the shadowmap) to search for pixels that are close to the current pixel in the direction of the light and that could occlude this shadow. For this they use ray-marching over the depth buffer.

This approach require that you first render the whole scene depth buffer, and it has a penalty as doing raymaching is costly, but gives great results.



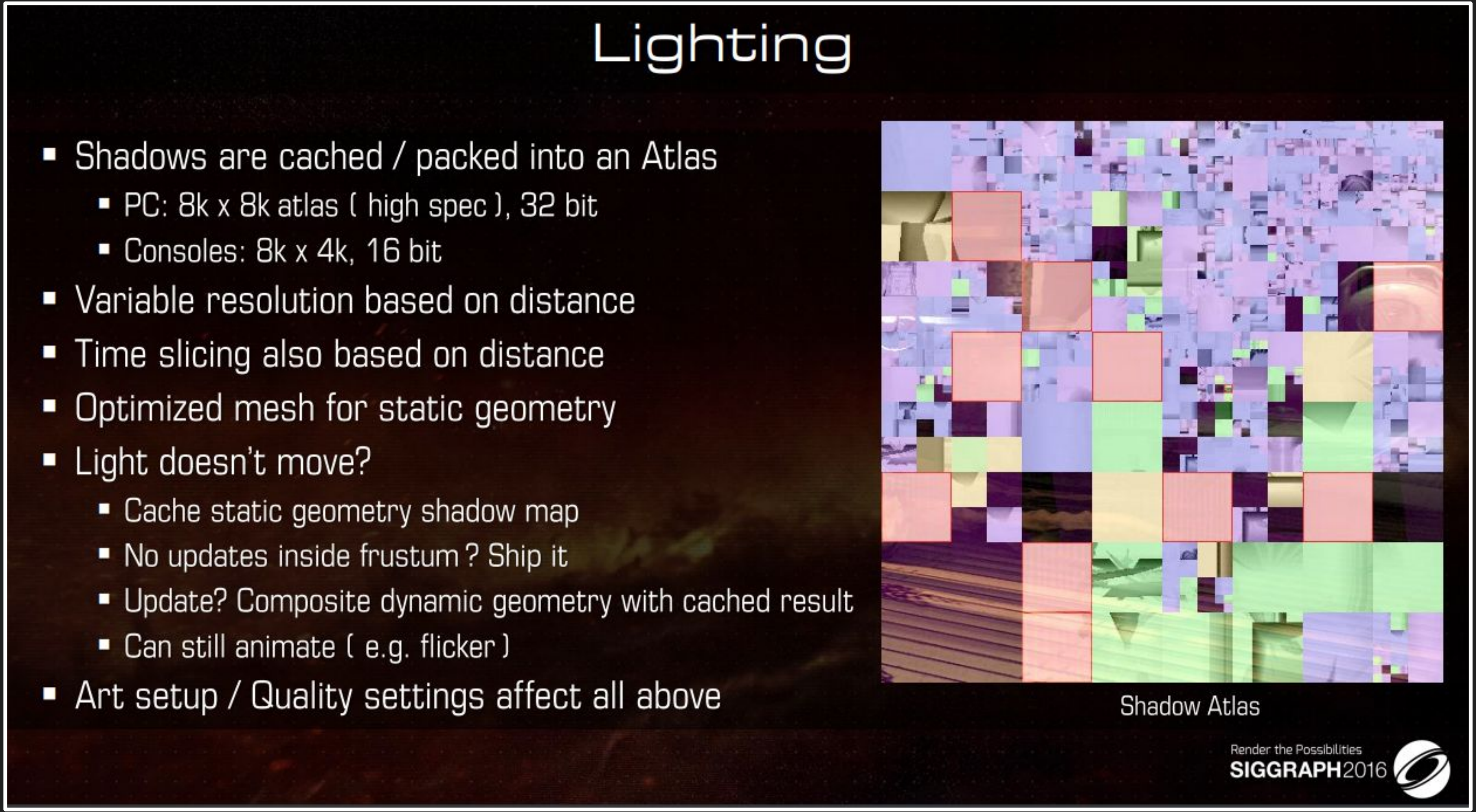


Raytraced Shadows

Current RTX cards support ray-tracing and some games are starting to use them to cast perfect shadows.

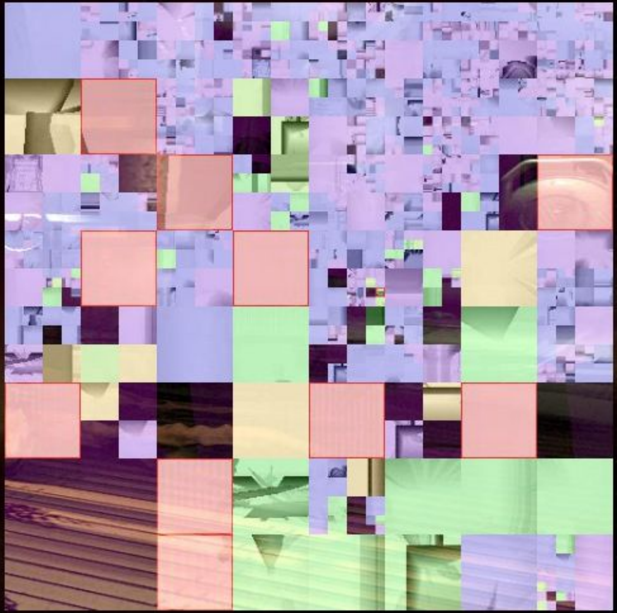
They have many benefits, they are simpler to code, they don't look pixelated and they can have translucency.

But the cost is very expensive computationaly and economically.



Lighting

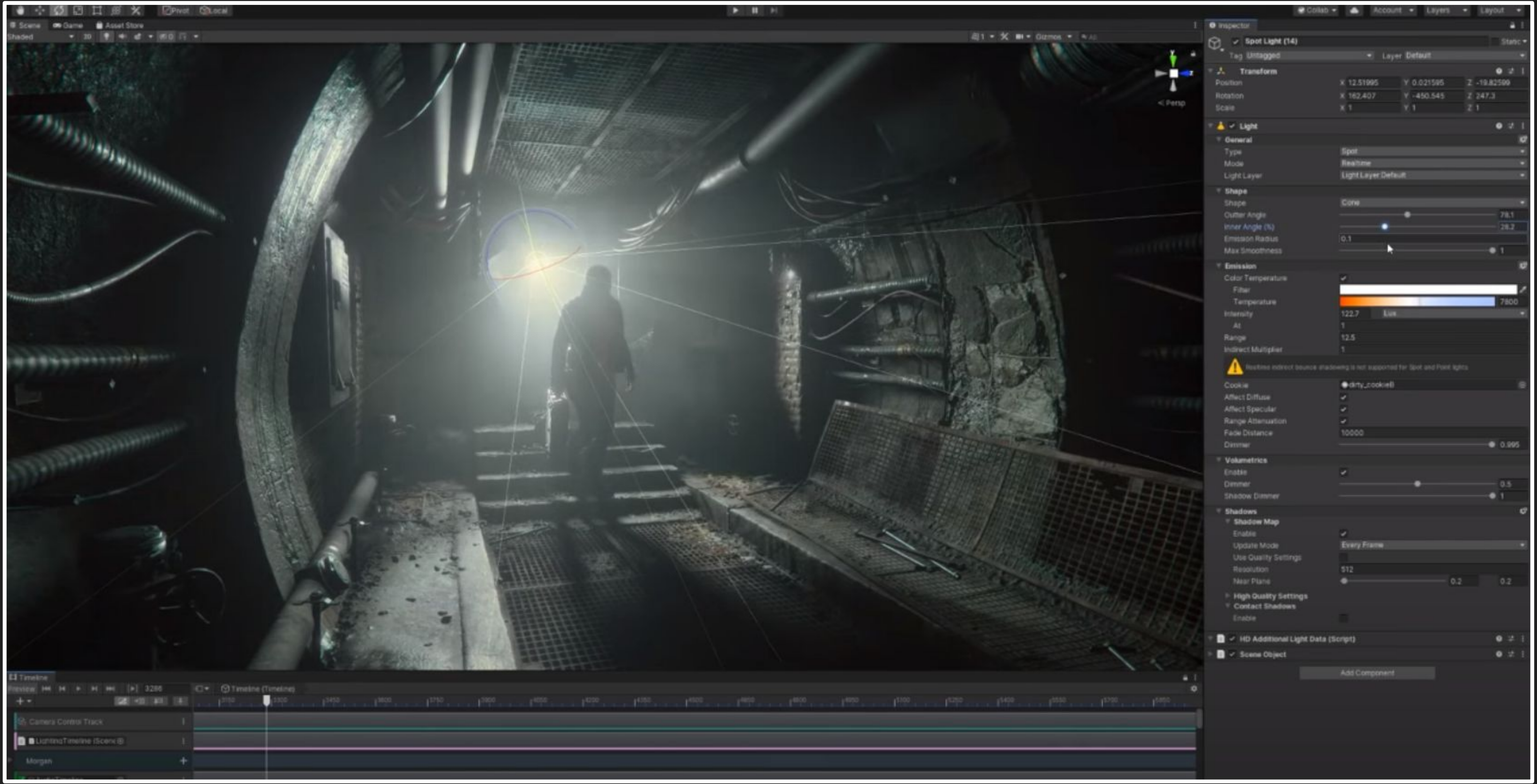
- Shadows are cached / packed into an Atlas
  - PC: 8k x 8k atlas (high spec), 32 bit
  - Consoles: 8k x 4k, 16 bit
- Variable resolution based on distance
- Time slicing also based on distance
- Optimized mesh for static geometry
- Light doesn't move?
  - Cache static geometry shadow map
  - No updates inside frustum ? Ship it
  - Update? Composite dynamic geometry with cached result
  - Can still animate (e.g. flicker)
- Art setup / Quality settings affect all above


Shadow Atlas

Render the Possibilities
SIGGRAPH2016

[Slide 22 from Doom 16 Presentation]

The Heretic: Example of good use of shadows



```
Spot Light (14)                                  Static
Tag Untagged            Layer Default
Transform
Position    X 12.51995   Y 0.021595   Z -19.82599
Rotation    X 162.407    Y -450.545   Z 247.3
Scale       X 1          Y 1          Z 1
Light
  General
    Type                    Spot
    Mode                    Realtime
    Light Layer             Light Layer Default
  Shape
    Shape                   Cone
    Outter Angle            78.1
    Inner Angle (%)         28.2
    Emission Radius         0.1
    Max Smoothness          1
  Emission
    Color Temperature       ✓
      Filter
      Temperature           7800
    Intensity               122.7   Lux
      At                    1
    Range                   12.5
    Indirect Multiplier     1
    Realtime indirect bounce shadowing is not supported for Spot and Point lights
    Cookie                  dirty_cookie8
    Affect Diffuse          ✓
    Affect Specular         ✓
    Range Attenuation       ✓
    Fade Distance           10000
    Dimmer                  0.995
  Volumetrics
    Enable                  ✓
    Dimmer                  0.5
    Shadow Dimmer           1
  Shadows
    Shadow Map
      Enable                ✓
      Update Mode           Every Frame
      Use Quality Settings
      Resolution            512
      Near Plane            0.2    0.2
    High Quality Settings
    Contact Shadows
      Enable
HD Additional Light Data (Script)
Scene Object
```

References

Source code of the Framework

Shadowmapping tutorial

Point Light Shadows

Doom 16: Devil is in the details (Slide 22)

Common techniques to improve shadow maps

Variance Shadow Maps

Thesis about illumination

Linearize Depthmap