CS559 Lecture 19-20: More Texture

Idea - break the "week's" material into smaller chunks
(rather than 2 big ones)

Outline:

1. Texture Basics Review
2. Fake Normals, Normal Maps, Bump Maps
3. Multi-Texture, Light Maps, AO Maps
4. Environment Maps
5. Shadow Maps

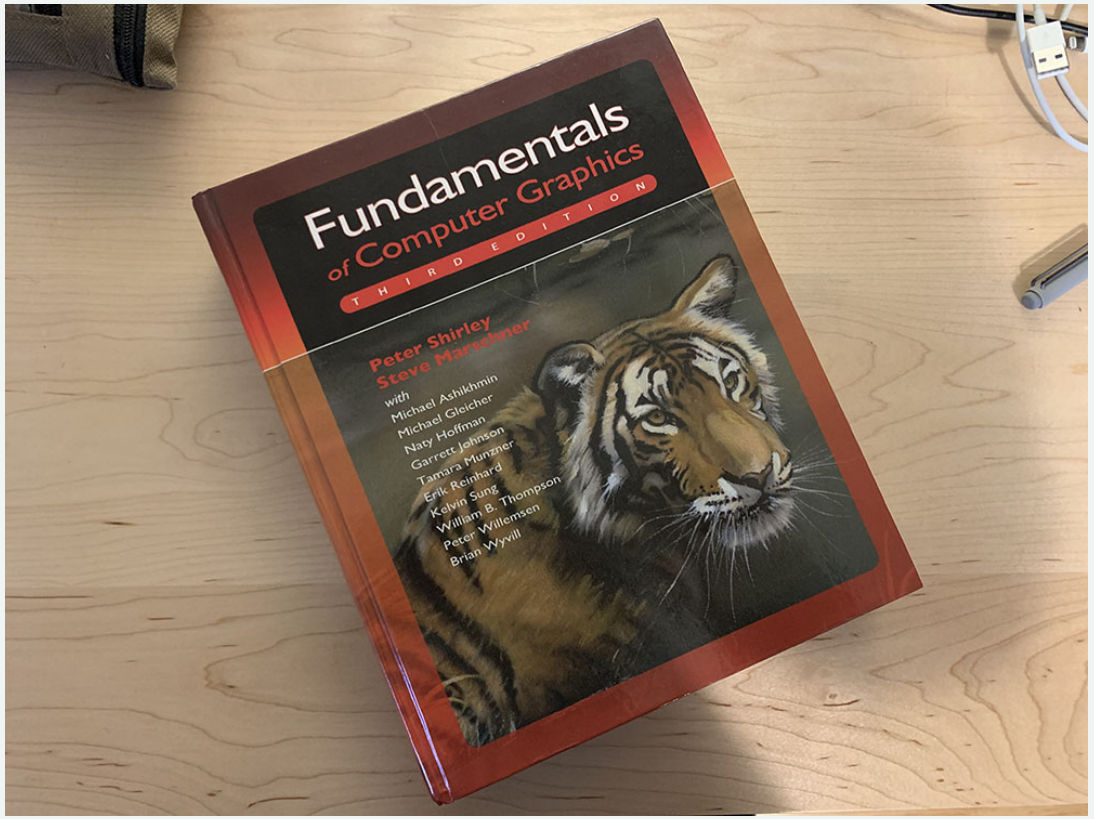
CS559 Lecture 19-20: More Texture

Part 1: Basic Texture Review

Motivation and Review

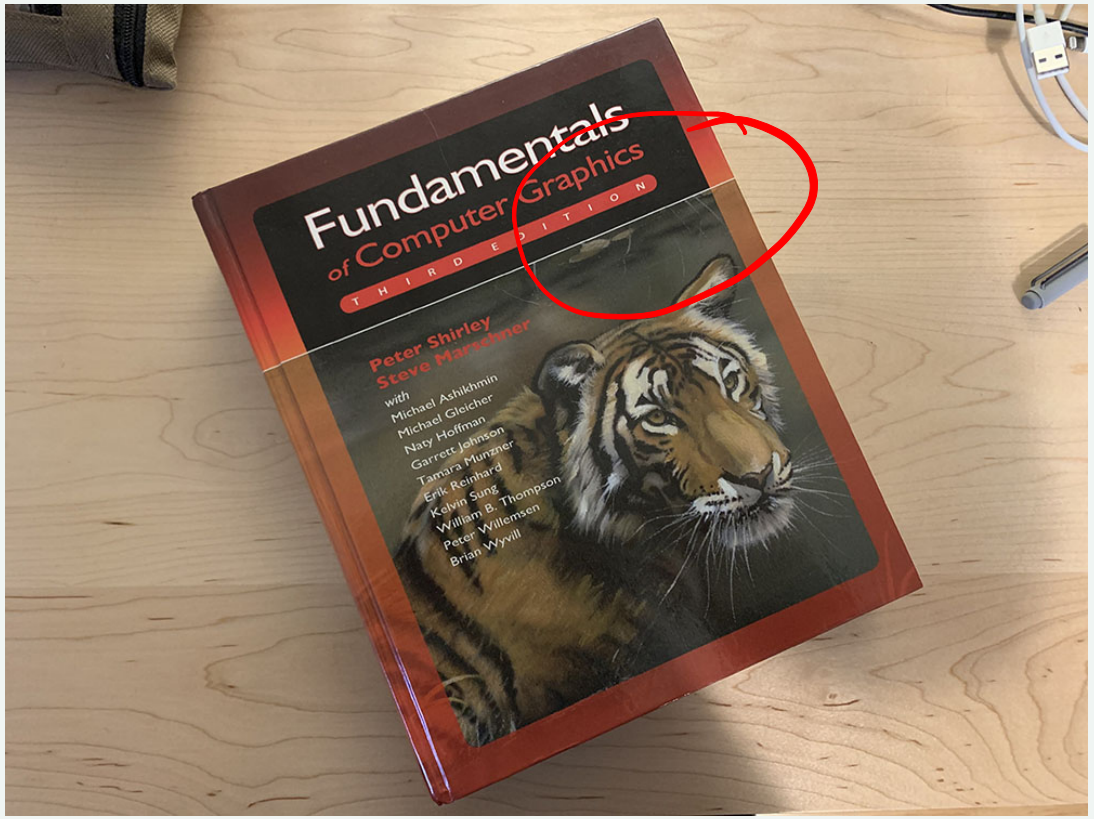
Why Basic Textures?

Because real objects are interesting

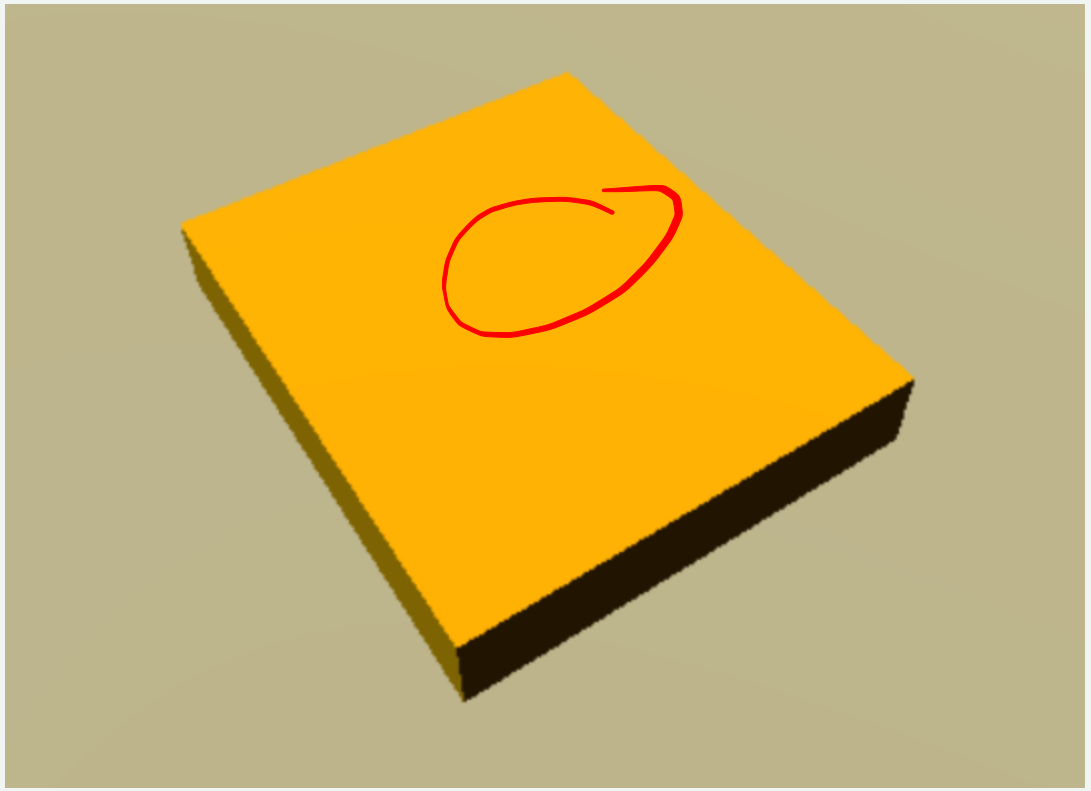


Why Basic Textures?

Real objects are interesting



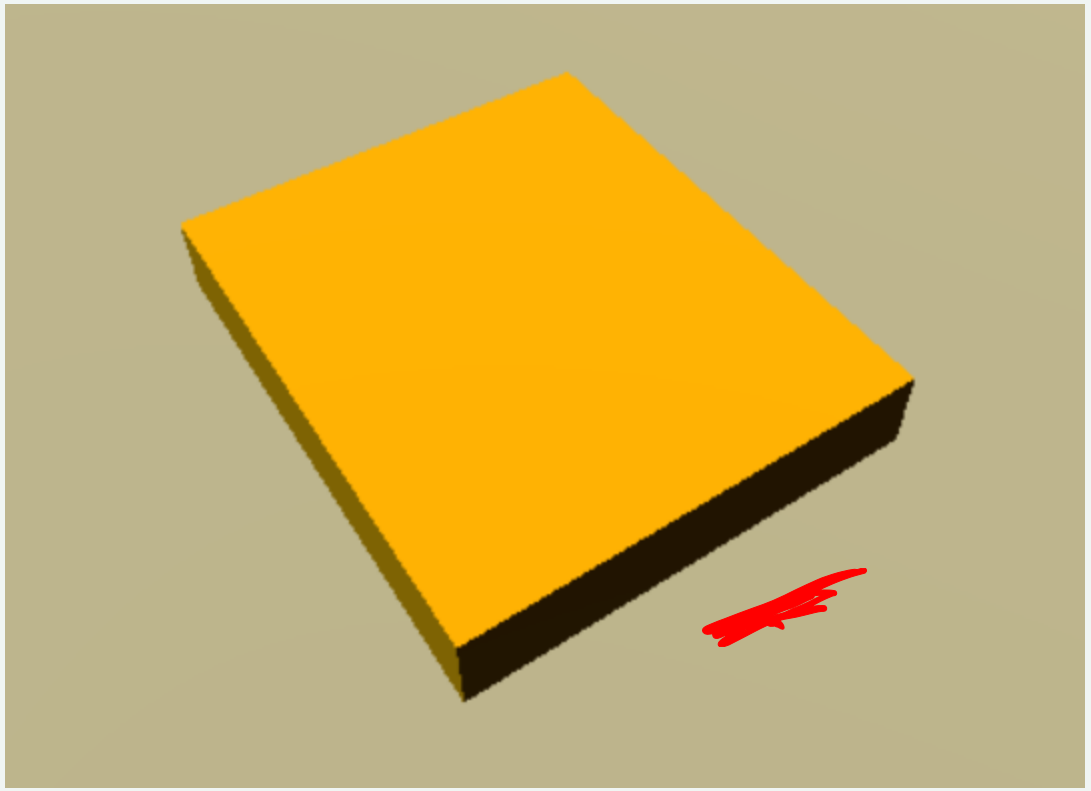
Computer Graphics can be boring...



Why Basic Textures?

Even Colors can Help

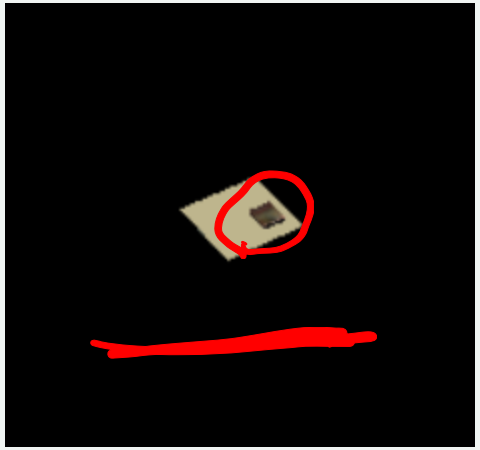
Computer Graphics can be boring...



But Why Textures?

Even Colors can Help

- Easy to get image
- Hard to model detals
- Easy to make simple geometry
- Proper sampling

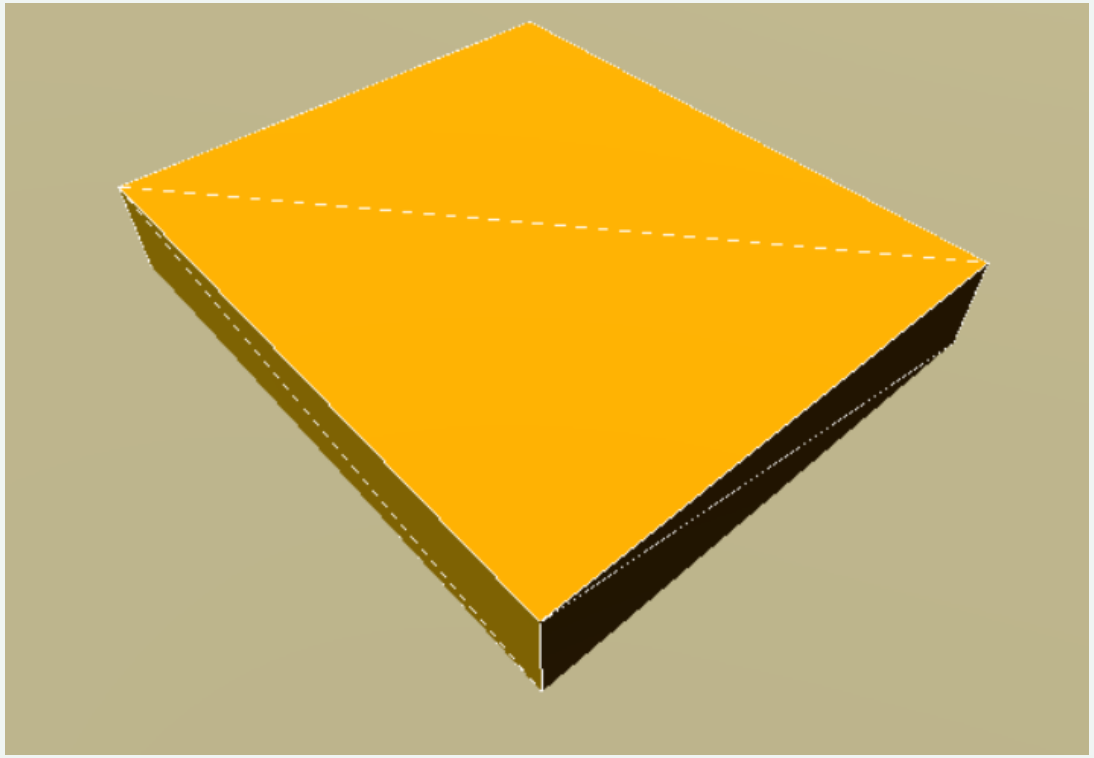


How To Do Basic Textures?

1. Make Some Geometry
2. Get a Picture
3. Get the picture in the right form
4. Assign UV values to vertices
5. Enable Texturing

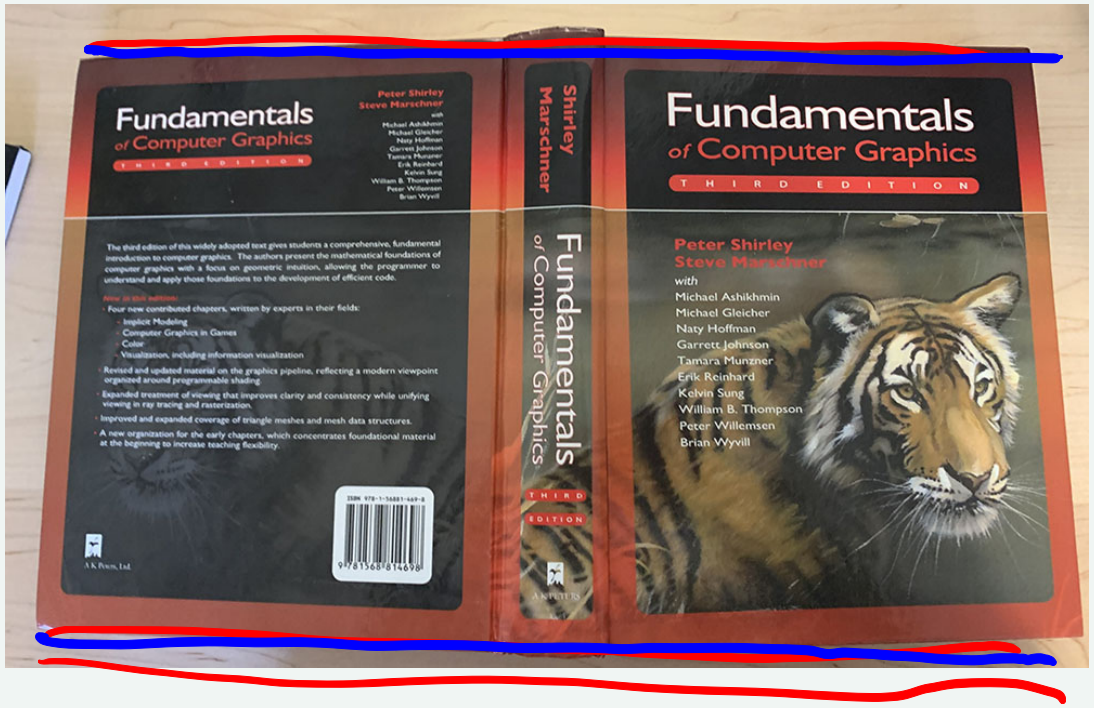
Geometry

1. **Make Some Geometry**
2. Get a Picture
3. Get the picture in the right form
4. Assign UV values to vertices
5. Enable Texturing



A Picture

1. Make Some Geometry
2. **Get a Picture**
3. Get the picture in the right form
4. Assign UV values to vertices
5. Enable Texturing

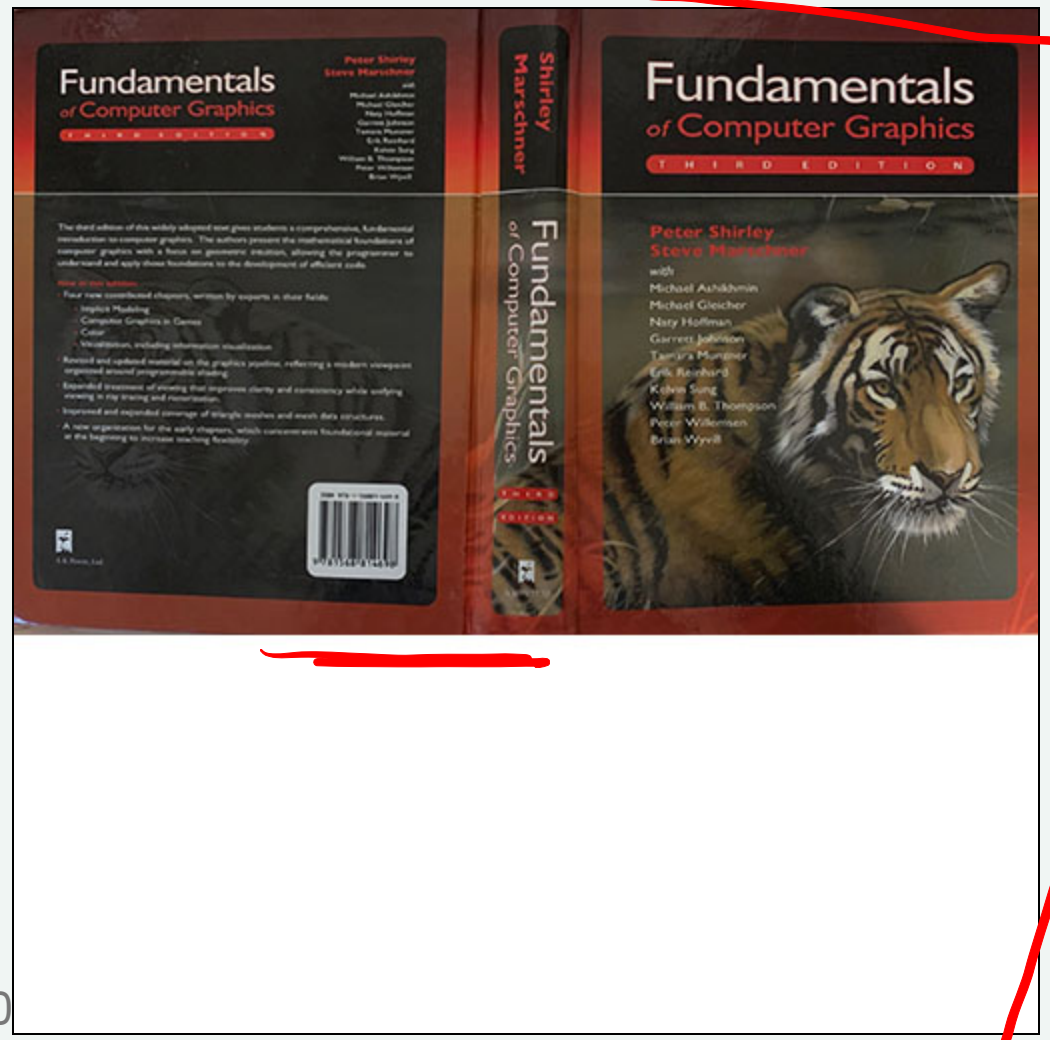


Can paint it yourself

Need to get things to match simple geometry

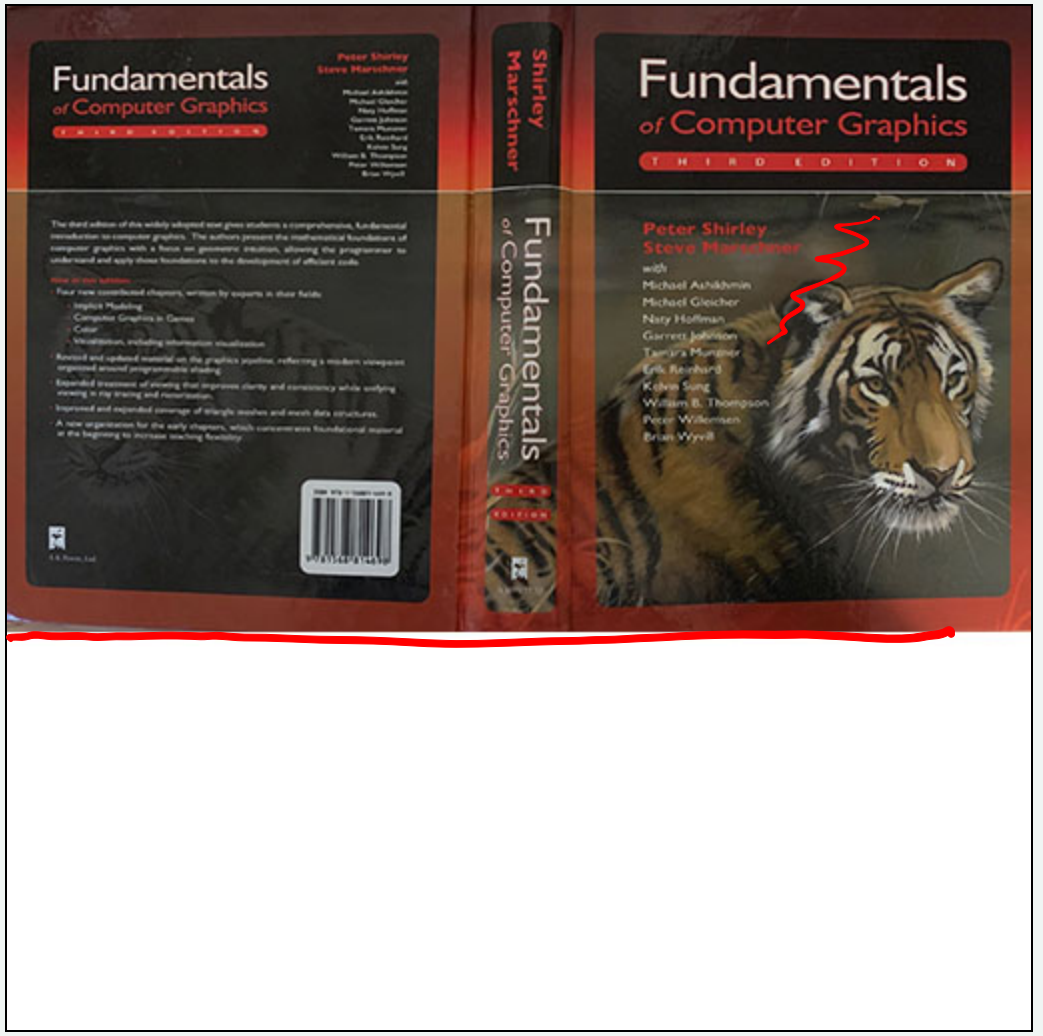
Process the Picture

1. Make Some Geometry
2. Get a Picture
3. **Get the picture in the right form**
4. Assign UV values to vertices
5. Enable Texturing



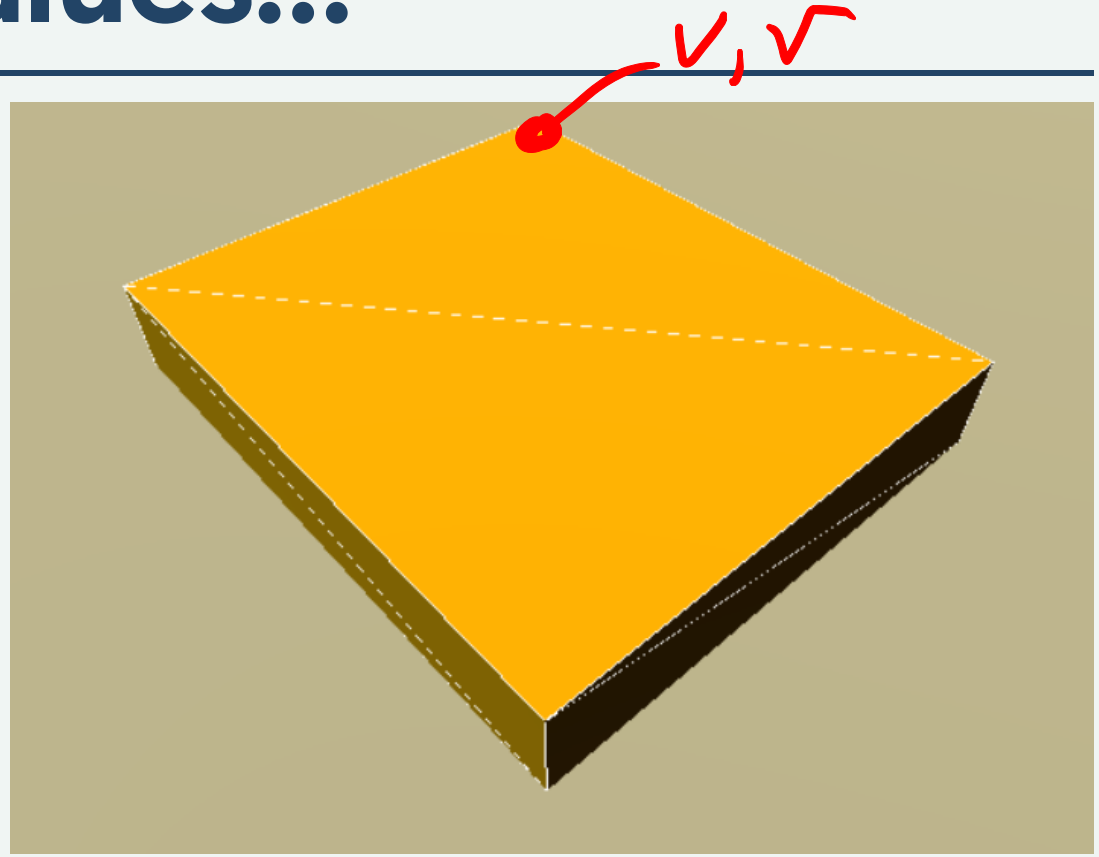
What do we need from a texture?

1. Square
2. Matches Simple Geometry
3. Minimal lighting
4. Put lots of parts in one image

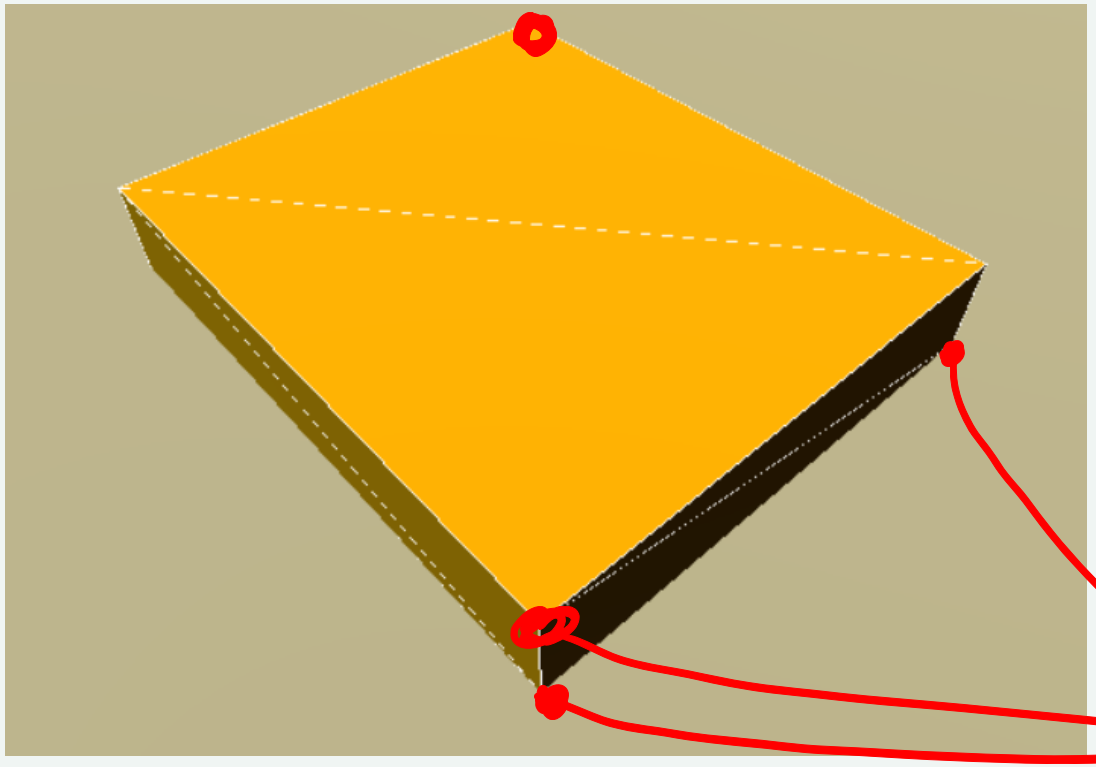


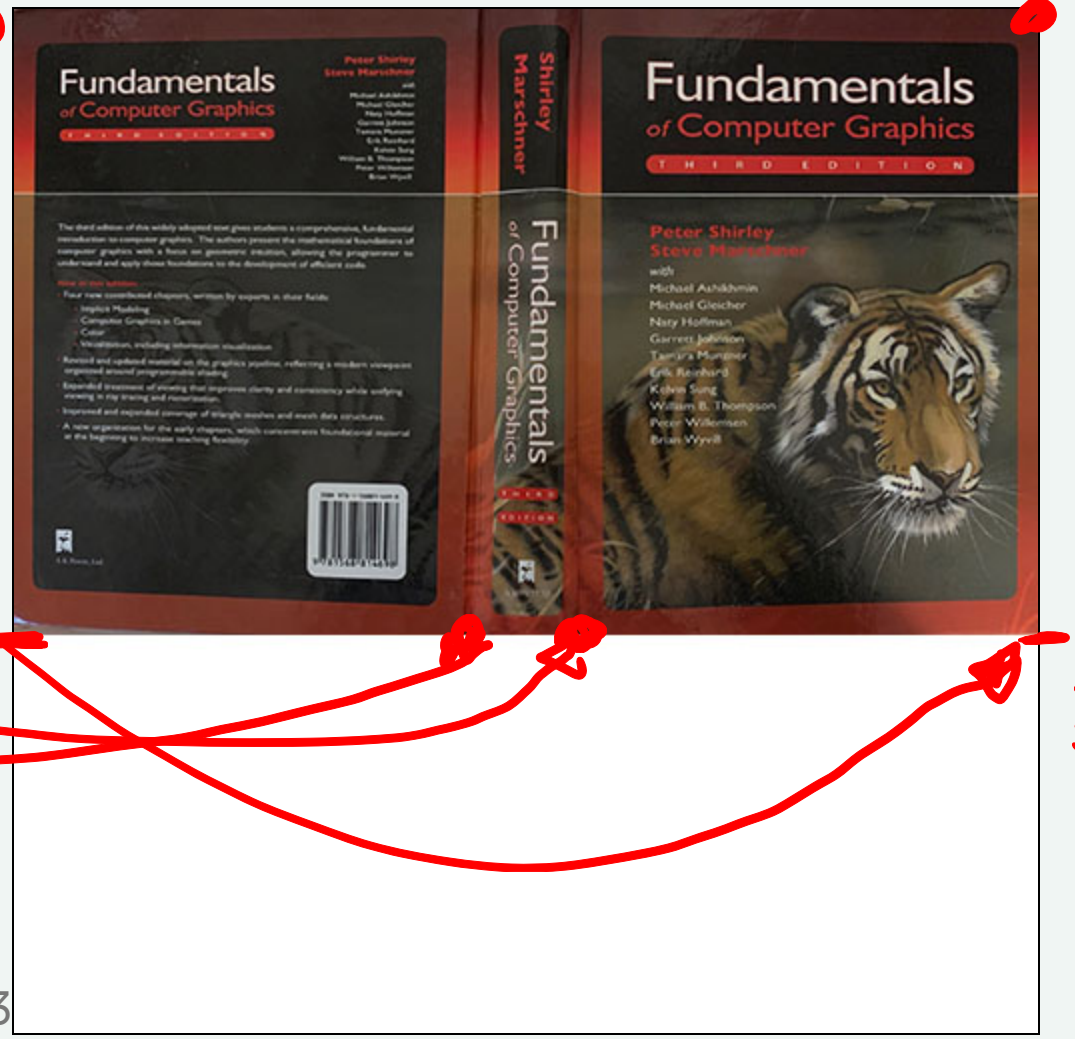
Getting those UV Values...

1. Make Some Geometry
2. Get a Picture
3. Get the picture in the right form
4. **Assign UV values to vertices**
5. Enable Texturing



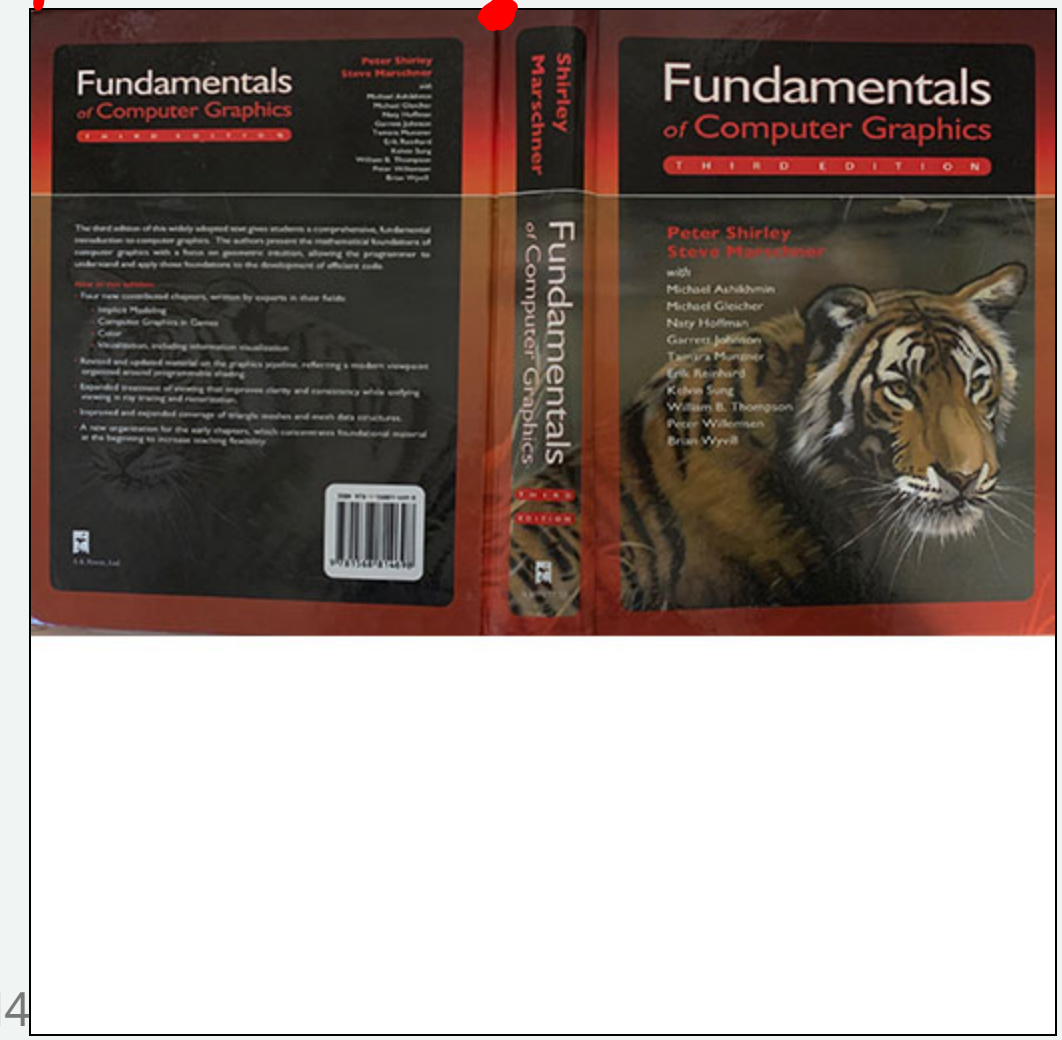
Finding UVs




Fundamentals
of Computer Graphics
THIRD EDITION
Peter Shirley
Steve Marschner
with
Michael Ashikhmin
Michael Gleicher
Naty Hoffman
Garrett Johnson
Tamura Munzner
Erik Reinhard
Kelvin Sung
William B. Thompson
Peter Willemsen
Brian Wyvill
Shirley
Marschner
Fundamentals
of Computer Graphics

Assign UV values to vertices

```
const vertexUVs = [
    // bottom (back of book)
    new T.Vector2(232/512,0),
    new T.Vector2(0       ,0),
    new T.Vector2(0,       311/512),
    new T.Vector2(232/512,311/512),
    // top (front of book)
    new T.Vector2(282/512, 0),
    new T.Vector2(512/512, 0),
    new T.Vector2(512/512,311/512),
    new T.Vector2(282/512,311/512),
]
```



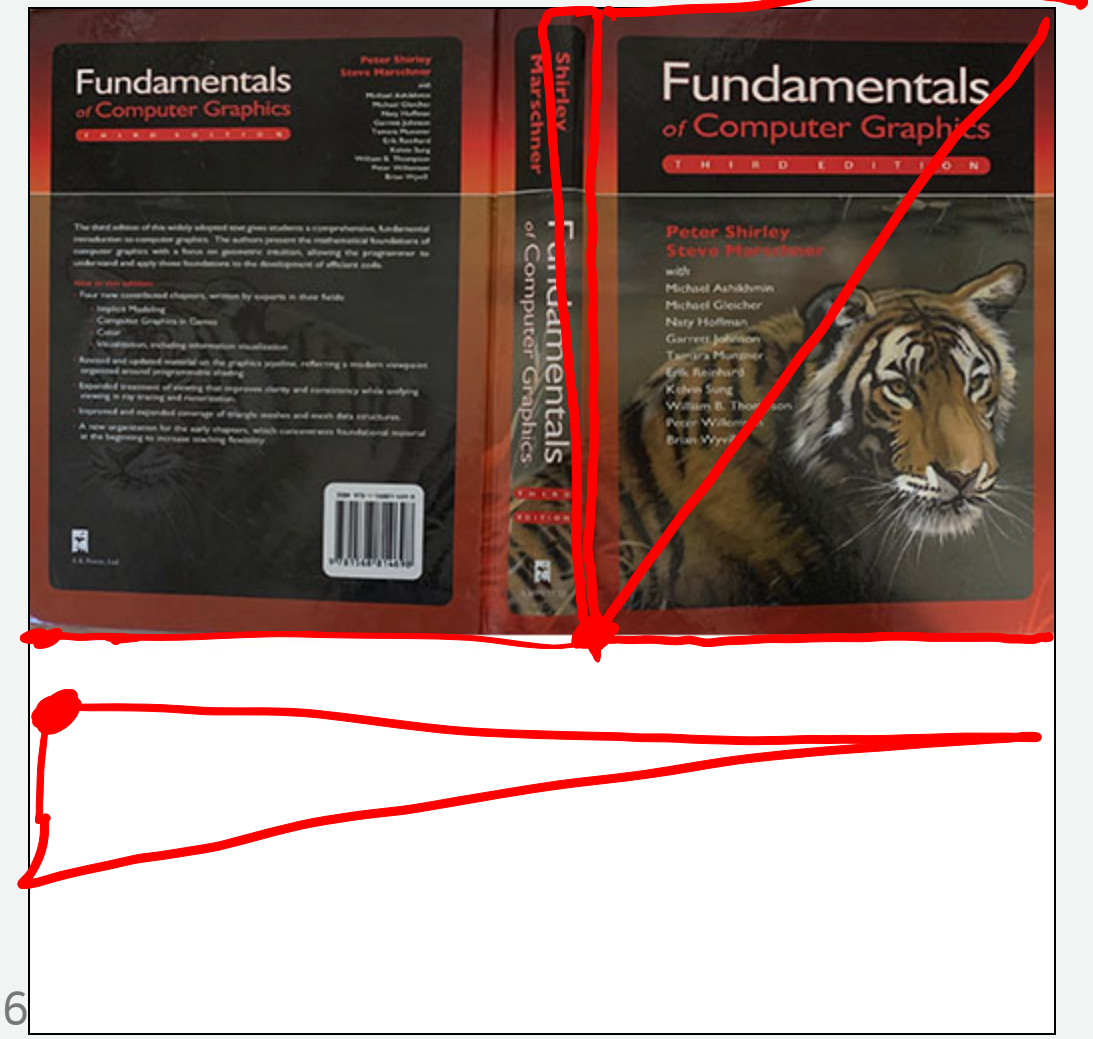
Put into the (weird) THREE structures

```
const vertexUVs = [
    // bottom (back of book)
    new T.Vector2(232/512,0),
    new T.Vector2(0      ,0),
    new T.Vector2(0,       311/512),
    new T.Vector2(232/512,311/512),
    // top (front of book)
    new T.Vector2(282/512, 0),
    new T.Vector2(512/512, 0),
    new T.Vector2(512/512,311/512),
    new T.Vector2(282/512,311/512),
]
```

```
let face1V = [vertexUVs[0],
              vertexUVs[1],
              vertexUVs[2]
             ];
// ...
let faceVs = [face1V, face2V, ...];

//
geom.faceVertexUvs = [faceVs];
```

Why Per Face? - Vertex Splitting!

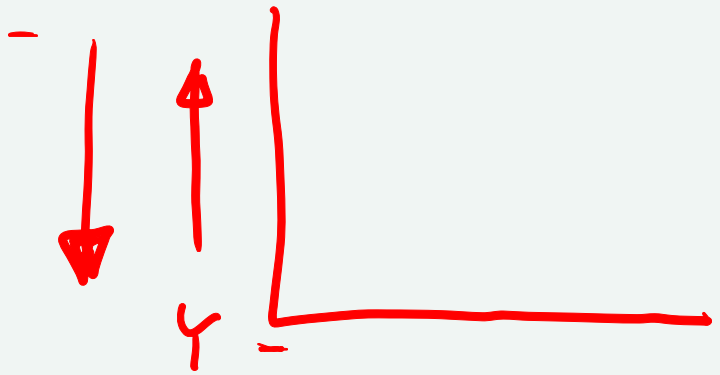


Put it together...

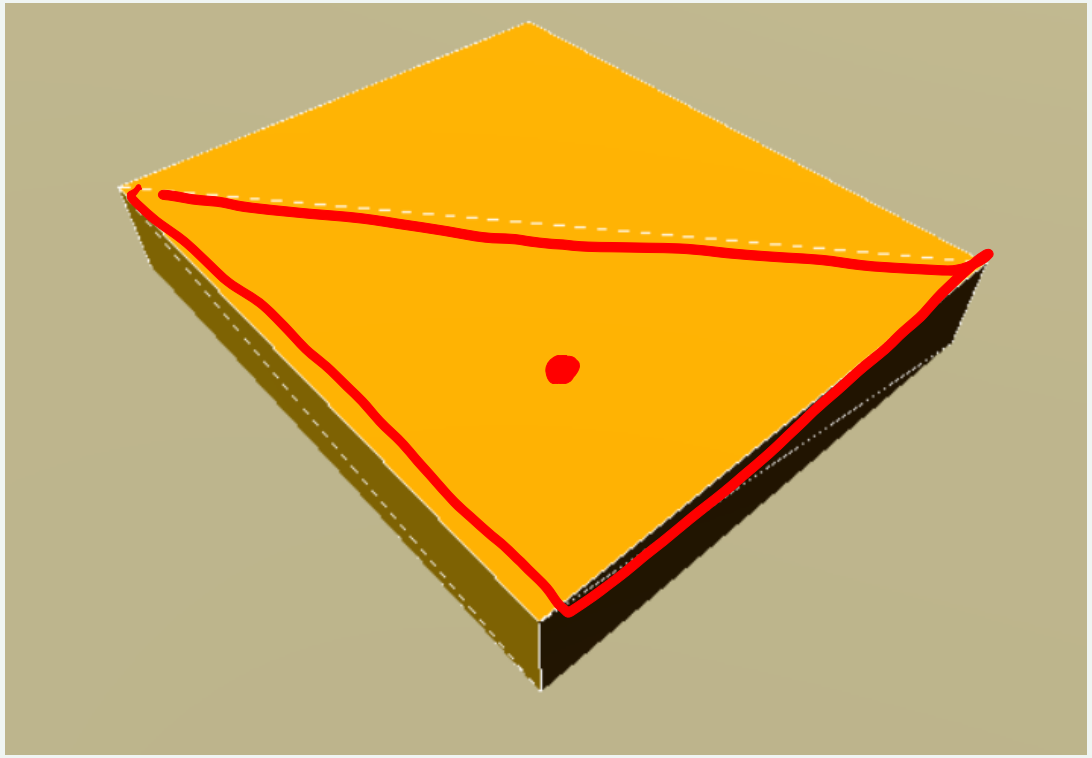
1. Make Some Geometry
2. Get a Picture
3. Get the picture in the right form
4. Assign UV values to vertices
5. **Enable Texturing**

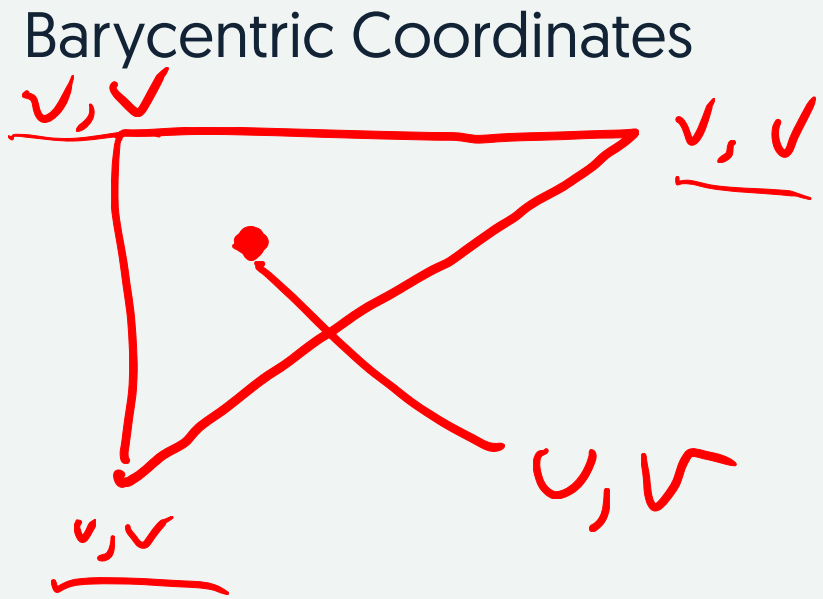
```
// load in the cover texture
let fcg = new T.TextureLoader().load("fcg-texture.jpg");
fcg.flipY = false;
```

```
let mat = new T.MeshStandardMaterial(
    {color:"white", map:fcg}
);
```

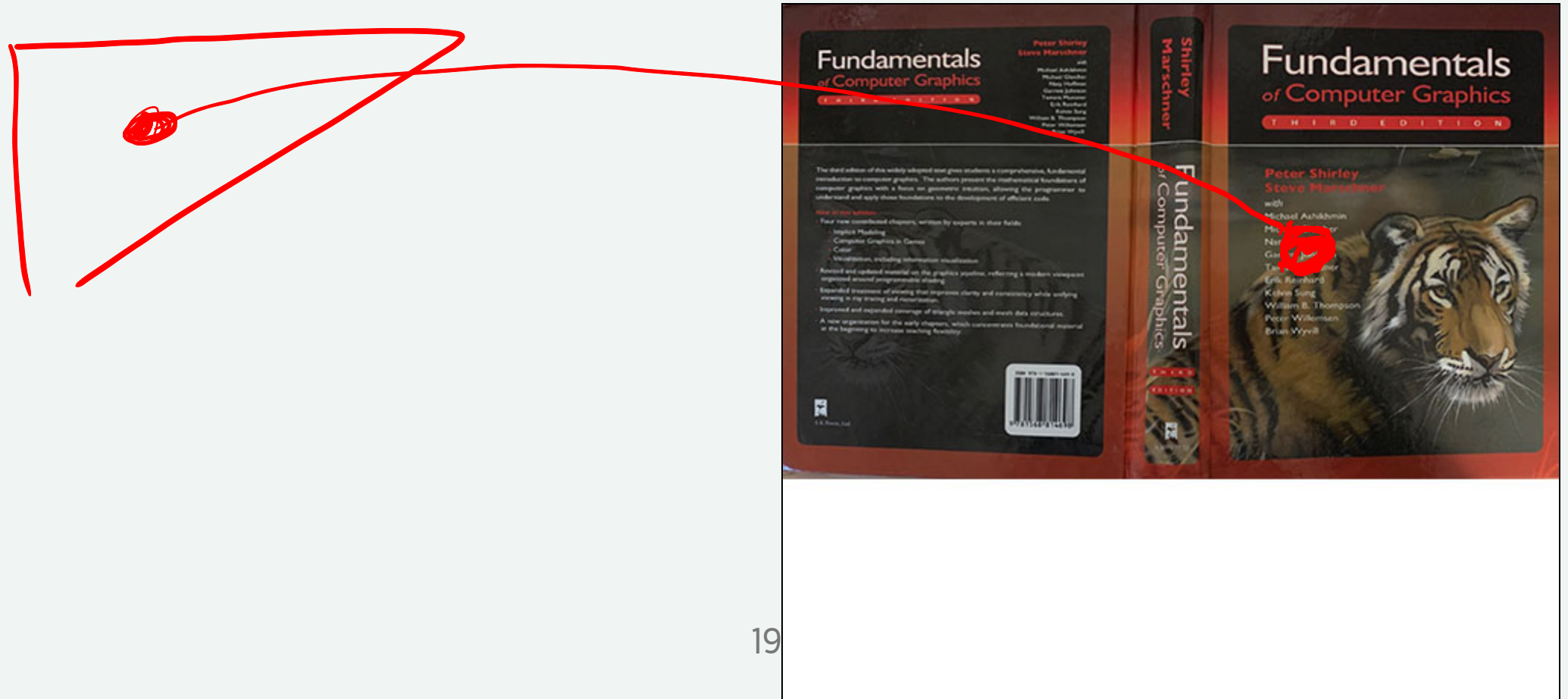


What the hardware does...
1. UV coordinates per pixel





What the hardware does...
2. Texture Lookup

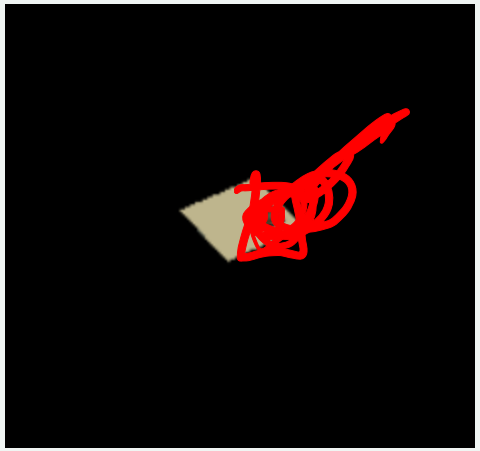


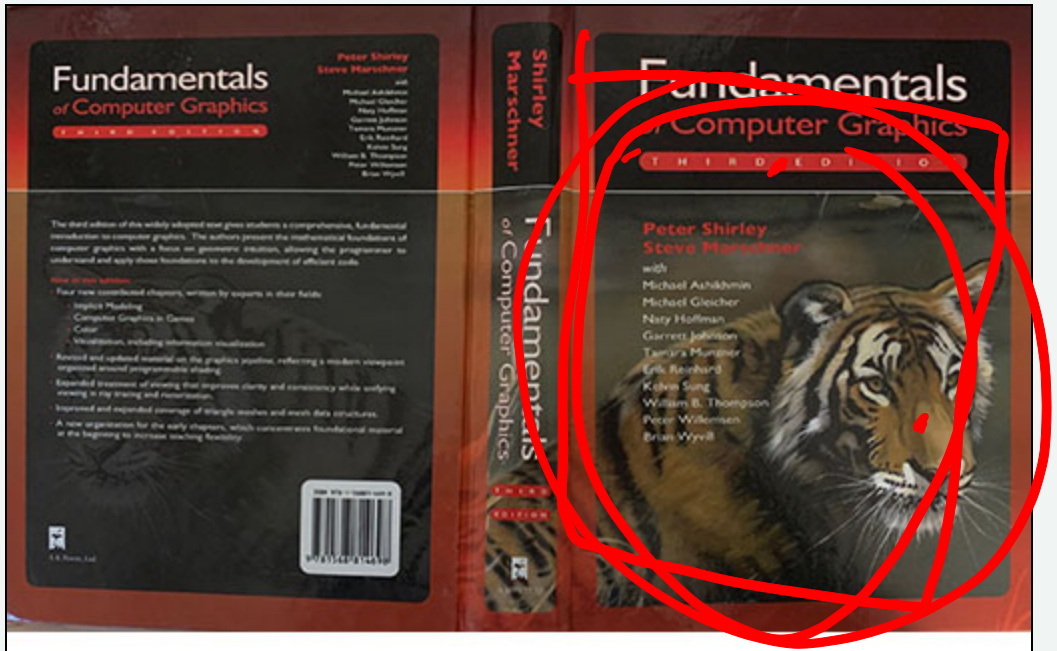
What the hardware does...
3. Texture Filtering

Each pixel maps to many texels

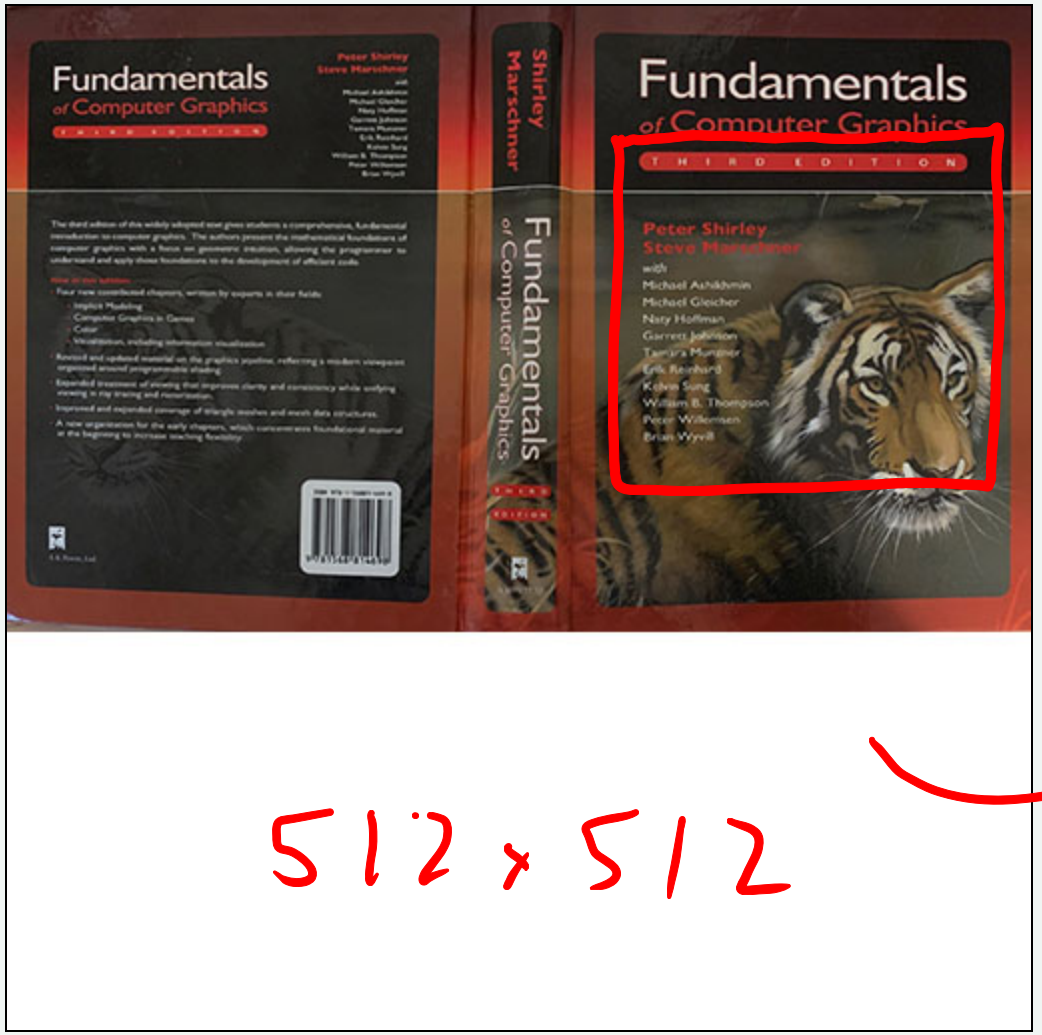
Can't pick one!

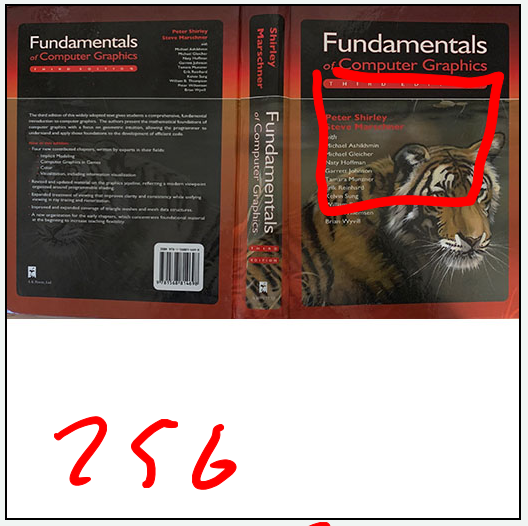
Average region together!

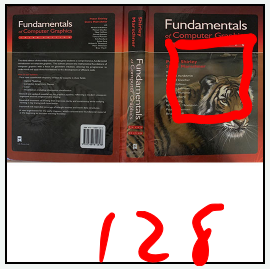




Filtering Fast... Mip Maps







Once you have the color...

Use as the material color (for lighting)

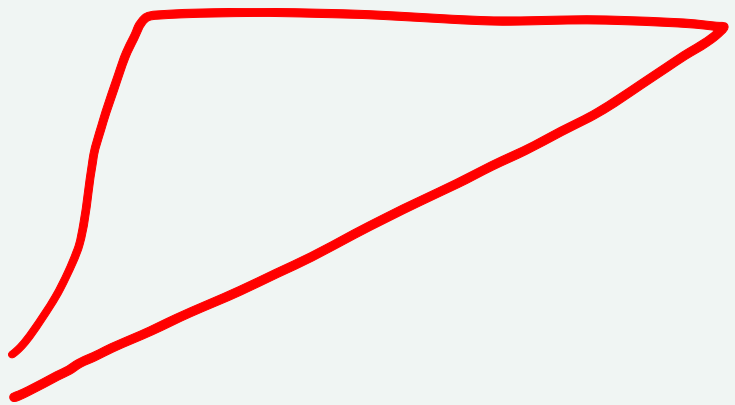
CS559 Lecture 19-20: More Texture

Part 2 - Fake Normals

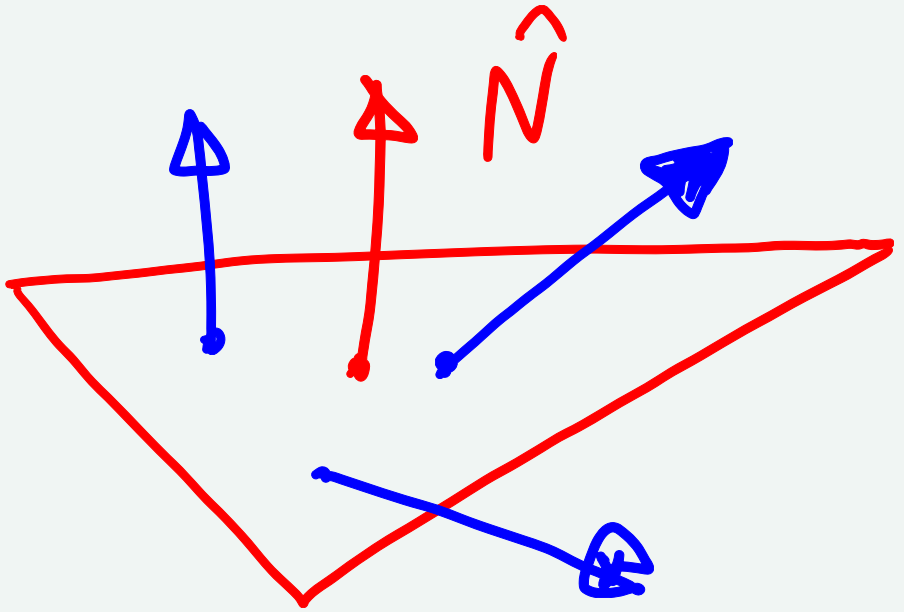
Bump Mapping

How do we get things to not be flat?

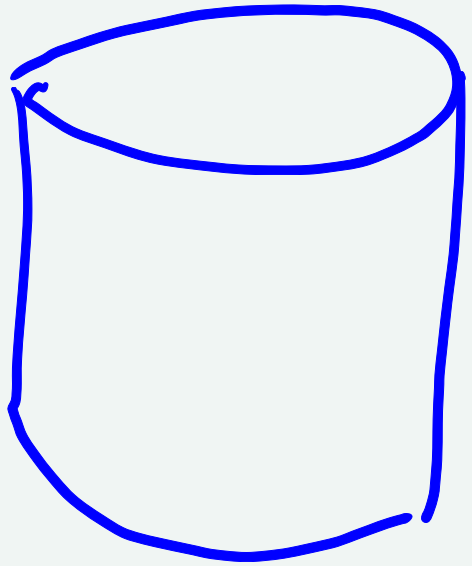
1. Make lots of triangles
2. Fake it with texture

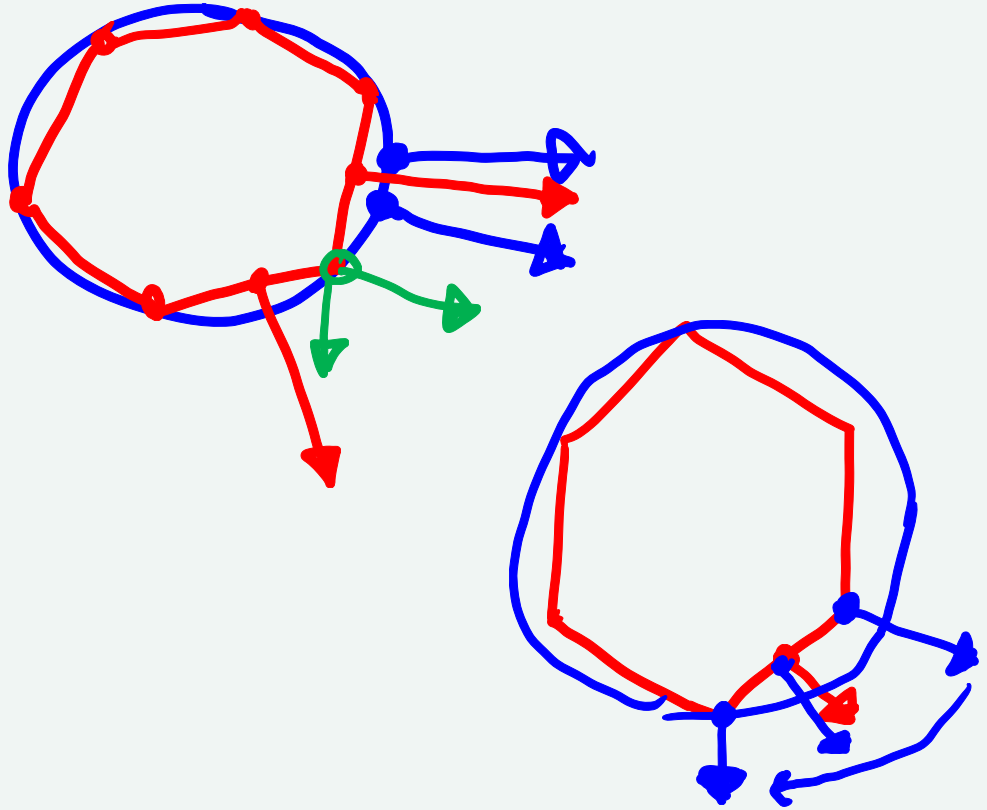


The Real Normal to a Triangle
The Fake Normal to a Triangle

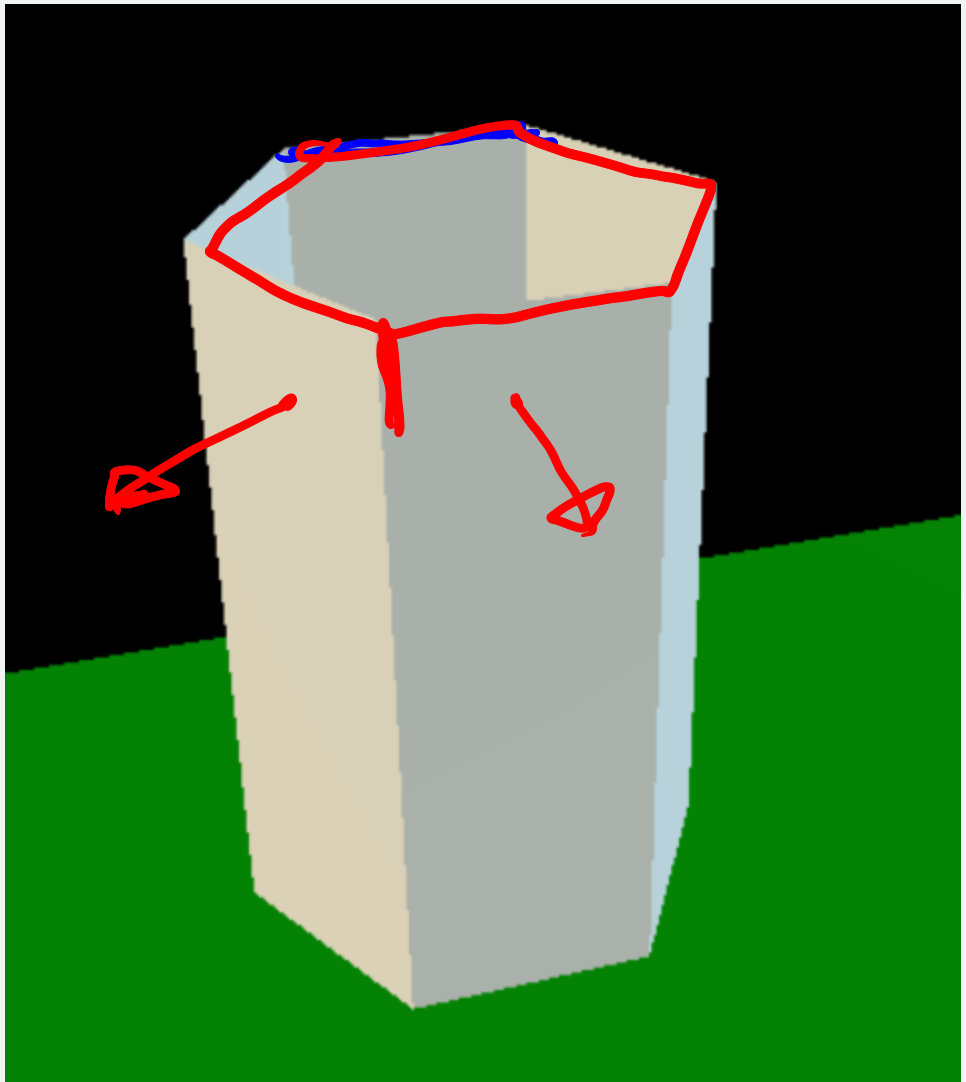


Why Fake Normals 1: Faking a Smooth Surface

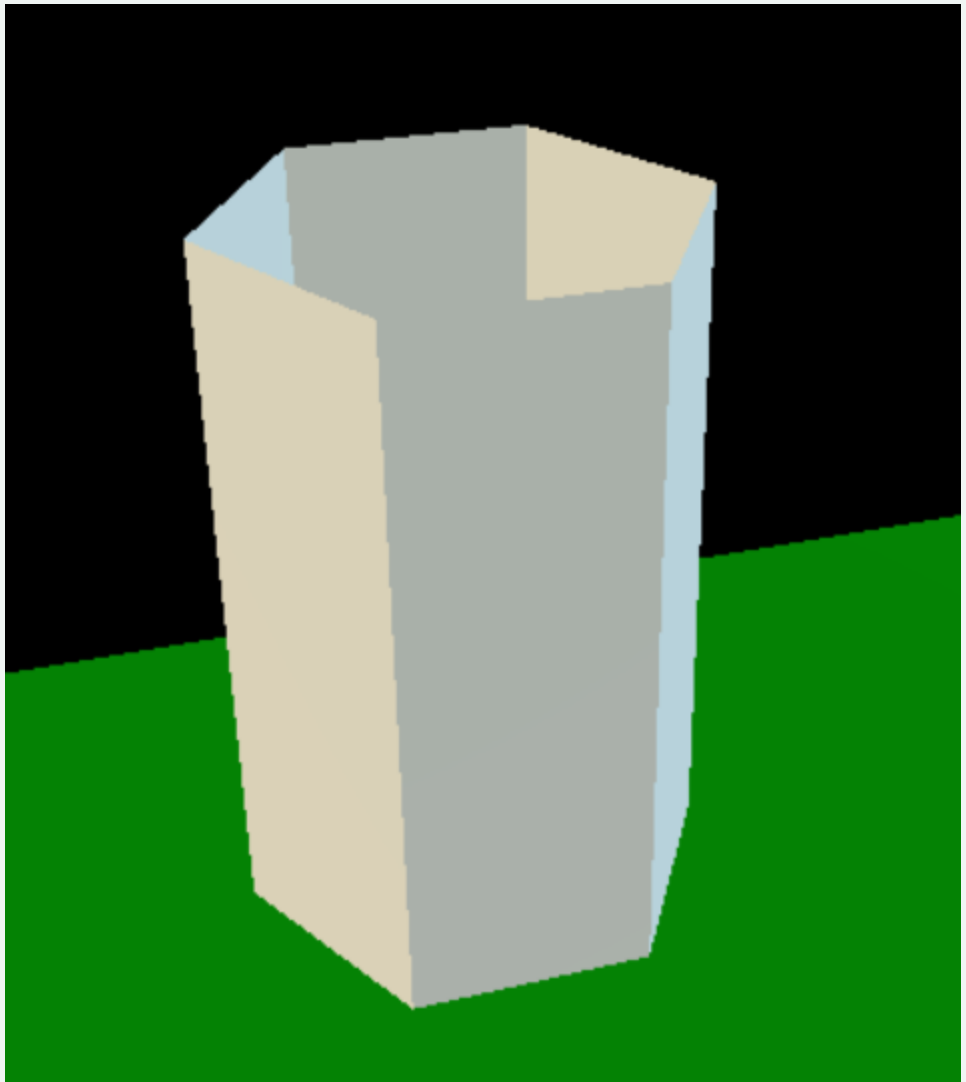


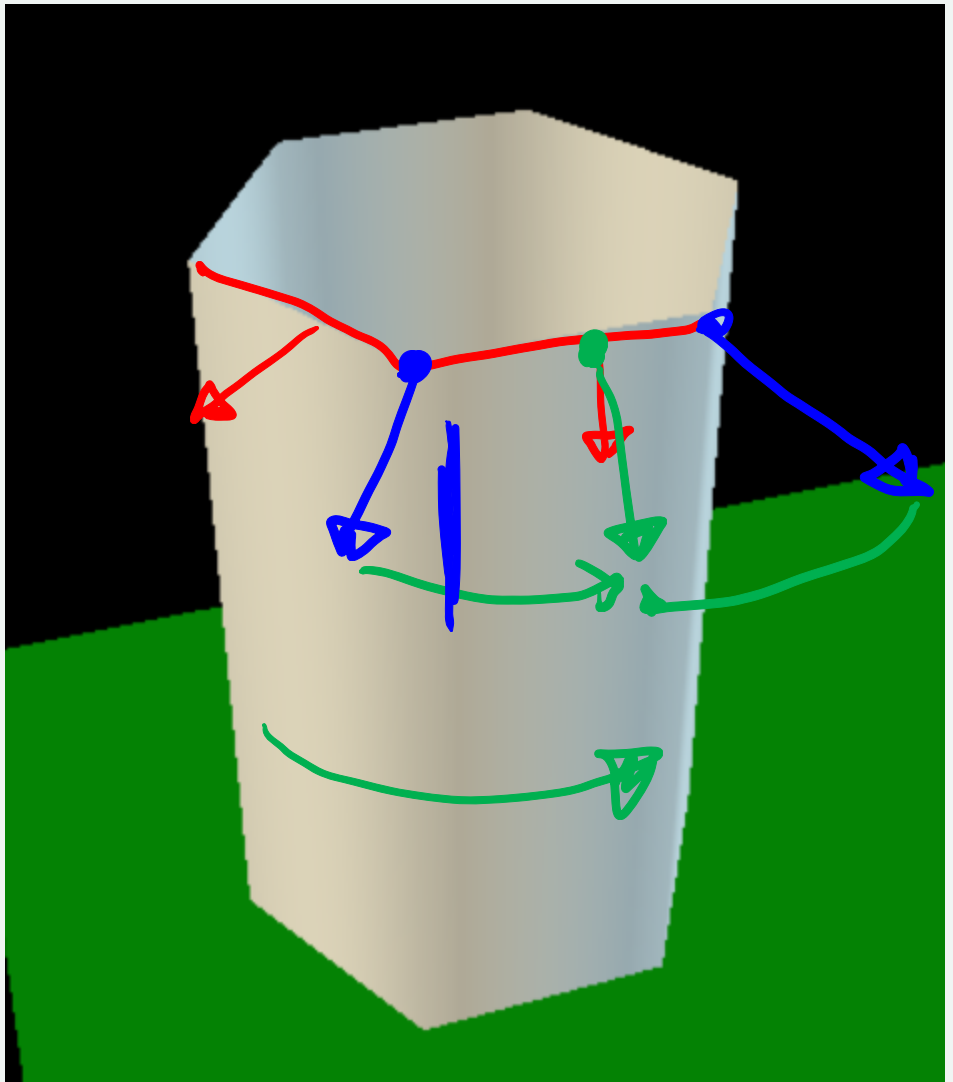


A "Cylinder" (6 sides)



A "Cylinder" (6 sides)



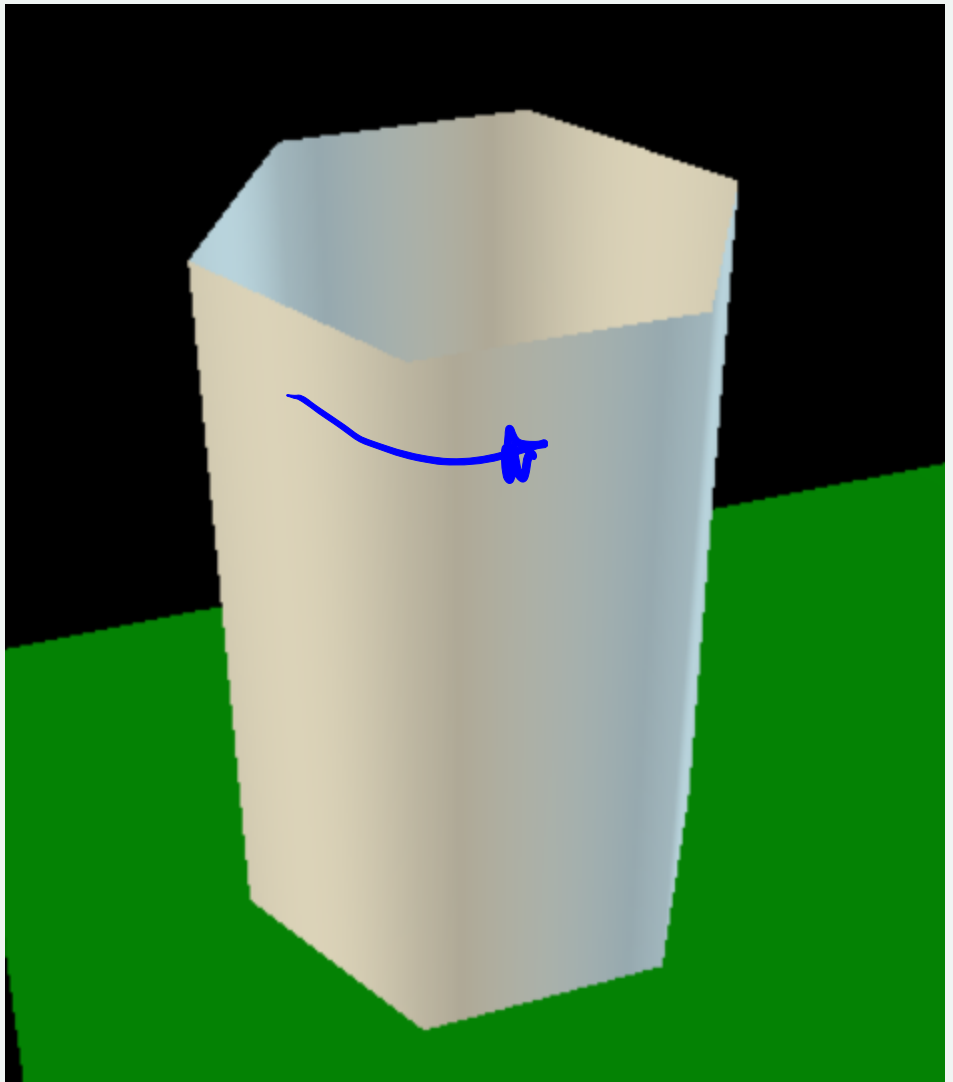


Smooth Shading

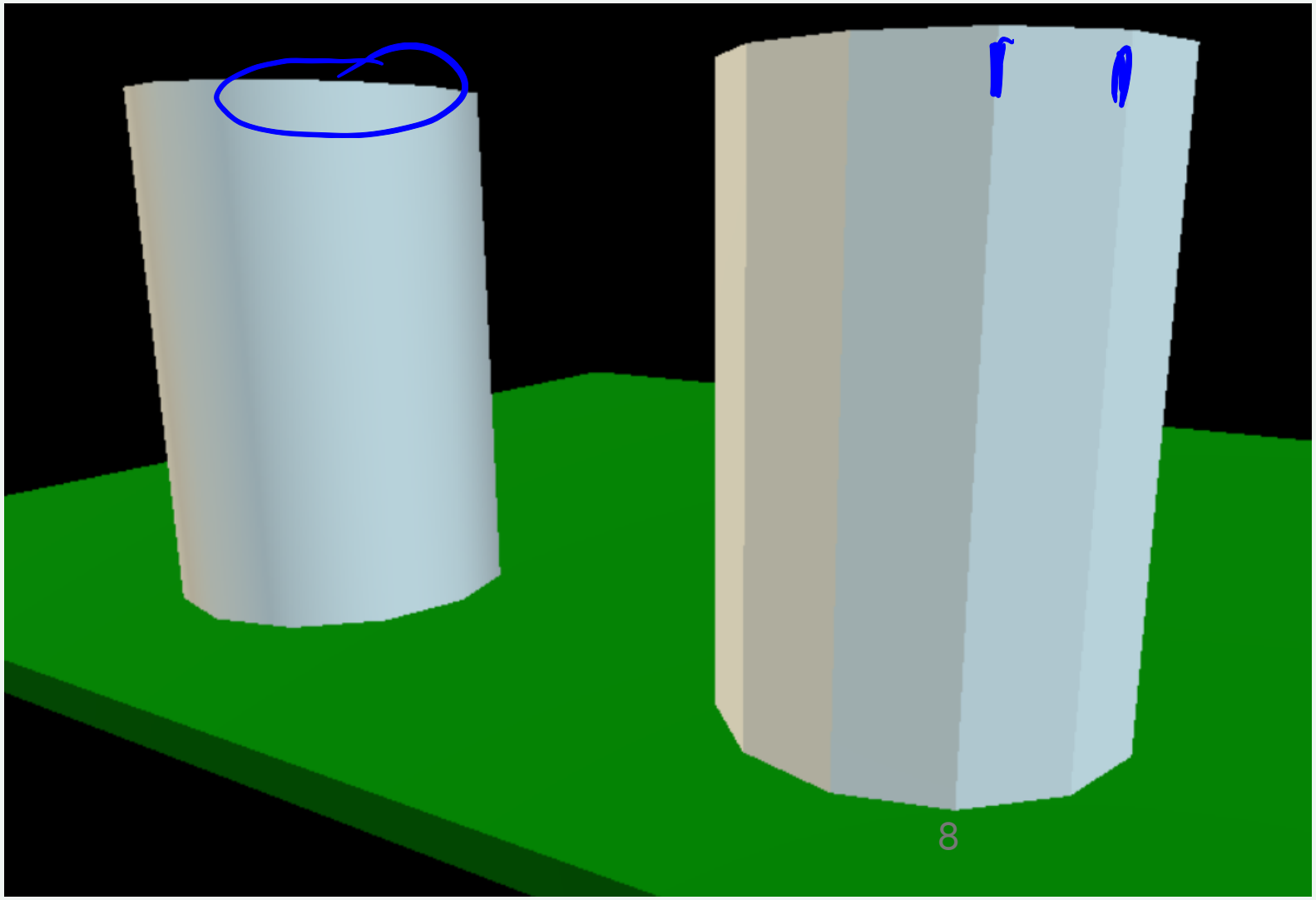
Still a 6 sided cylinder

Only changing the **lighting**

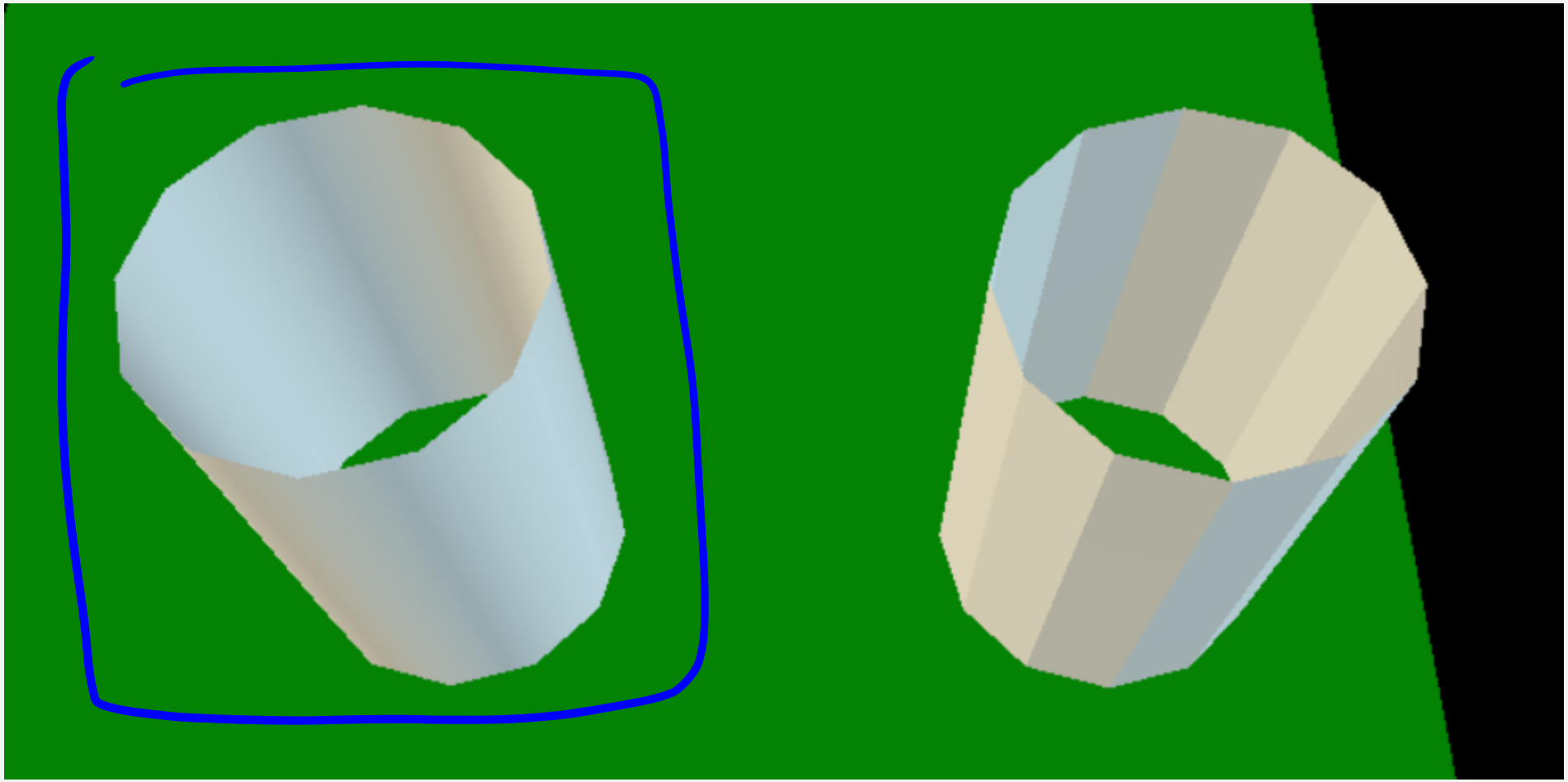
Change lighting by changing **normals**



Even better (12 sides)



Really 12 sides



Faking Shapes with Normals

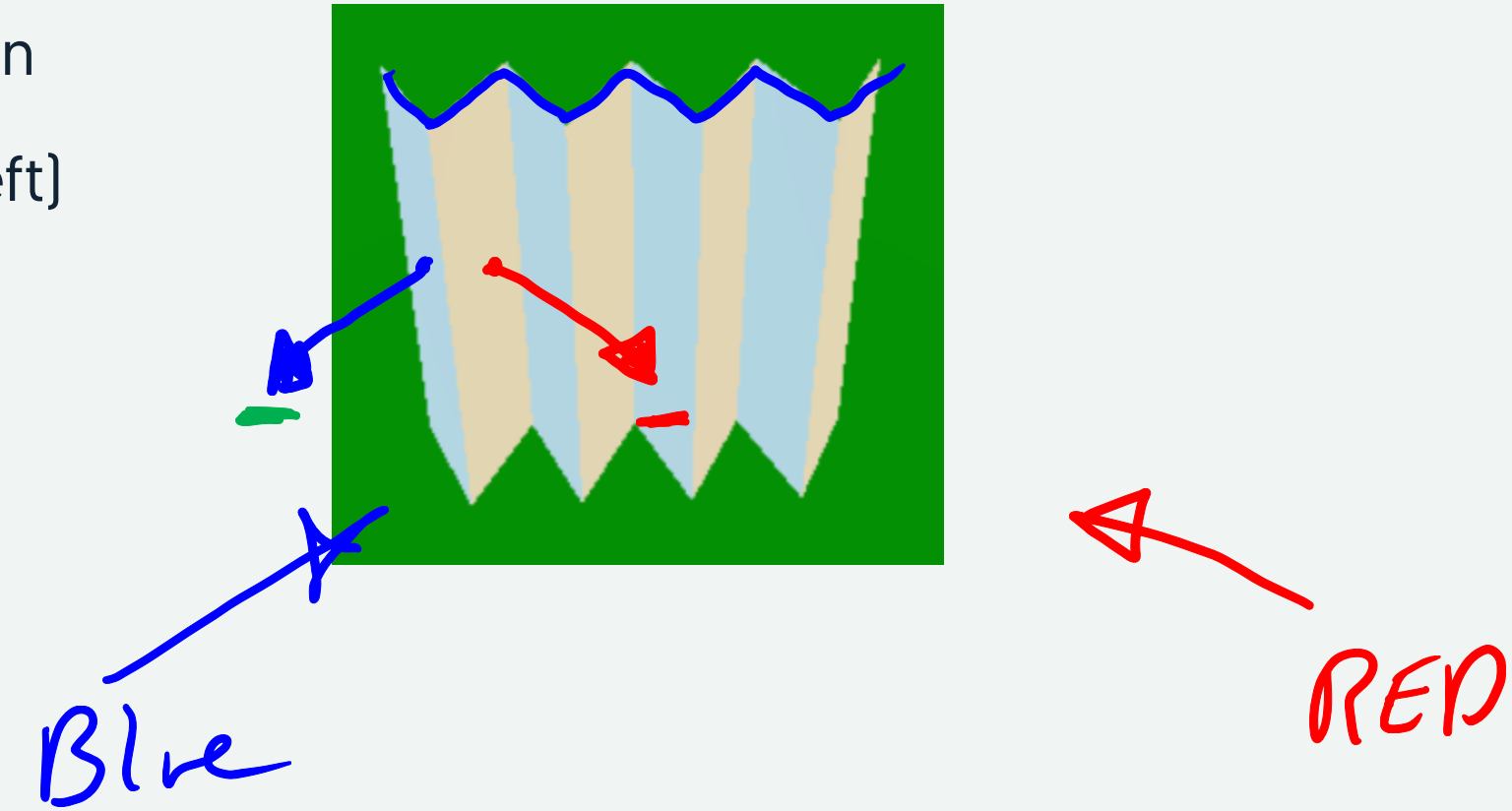
We don't have to make the right shape...

We just have to make it look right (with lighting)

Toy example: Wavy surface

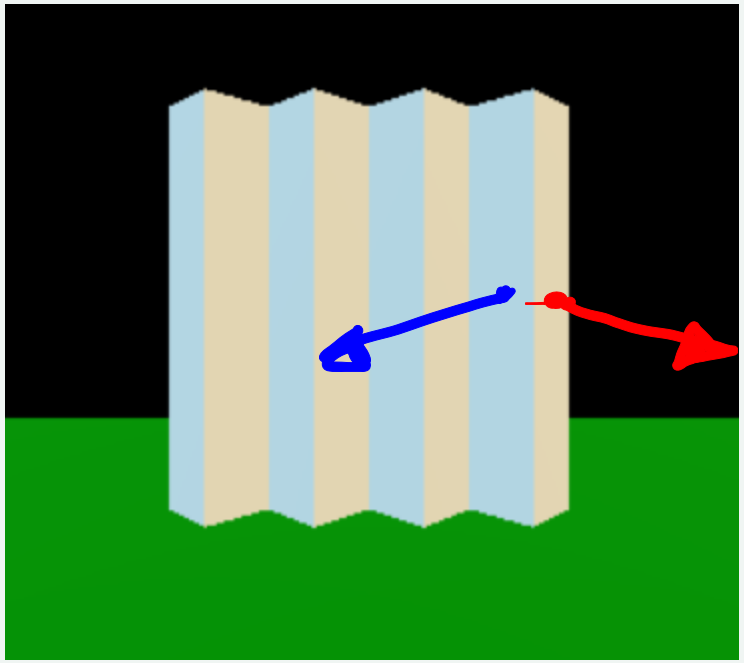
16 triangles - accordion pattern

extreme lighting (blue from left)



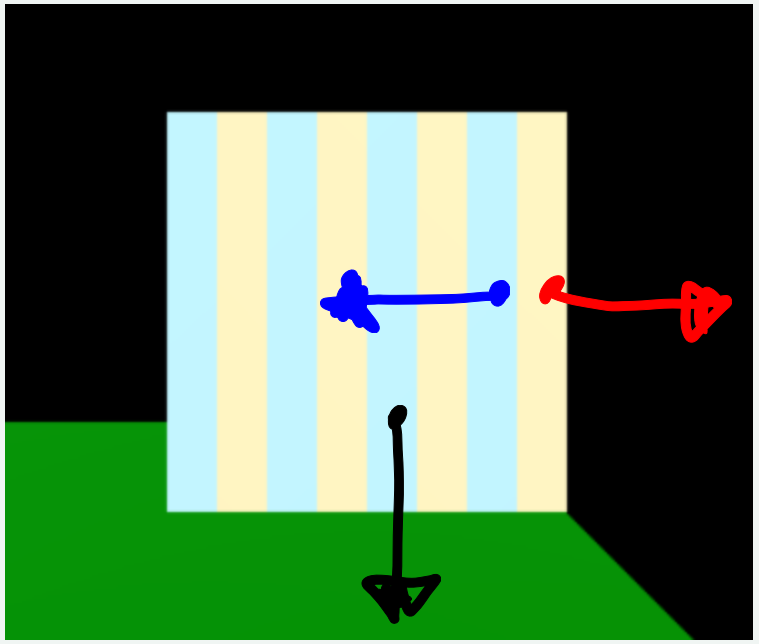
Toy example: Wavy surface

head on



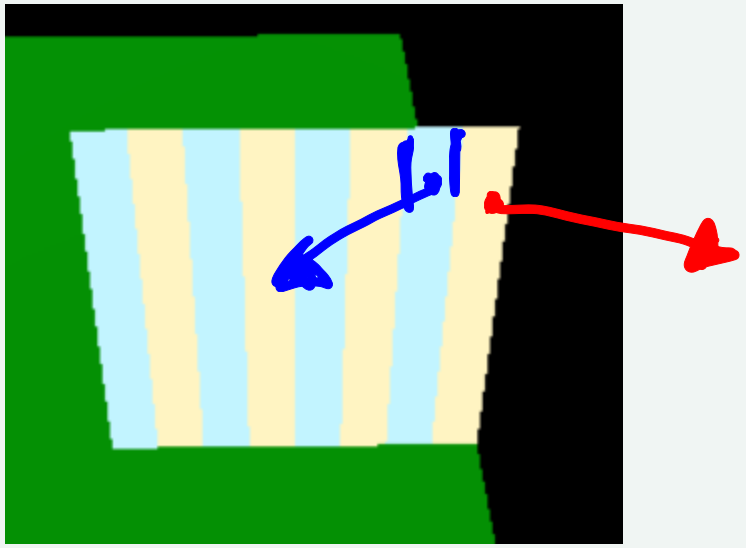
Fake Wavy surface

head on

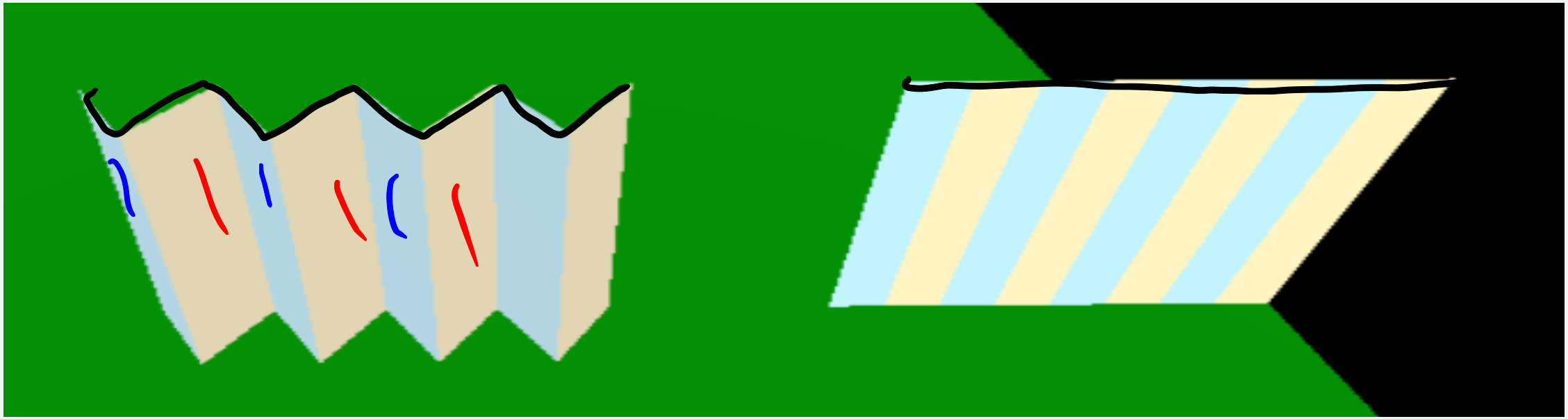


Fake?

It's flat - we just changed the normals



Real vs. Fake



Not so bad if you don't look at the edge...

Normal Maps

Idea: texture lookup of the normal vector!

R = x direction

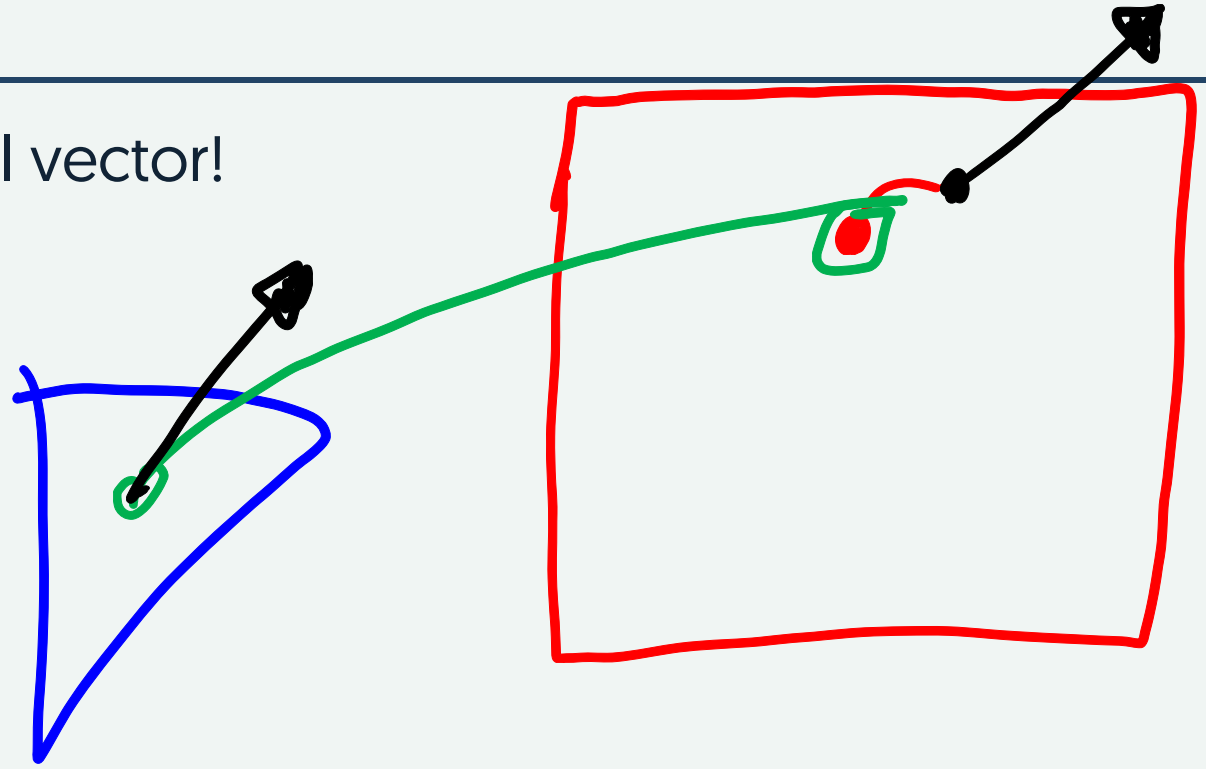
G = y direction

B = z direction

middle value (128/256) = 0

need to renormalize

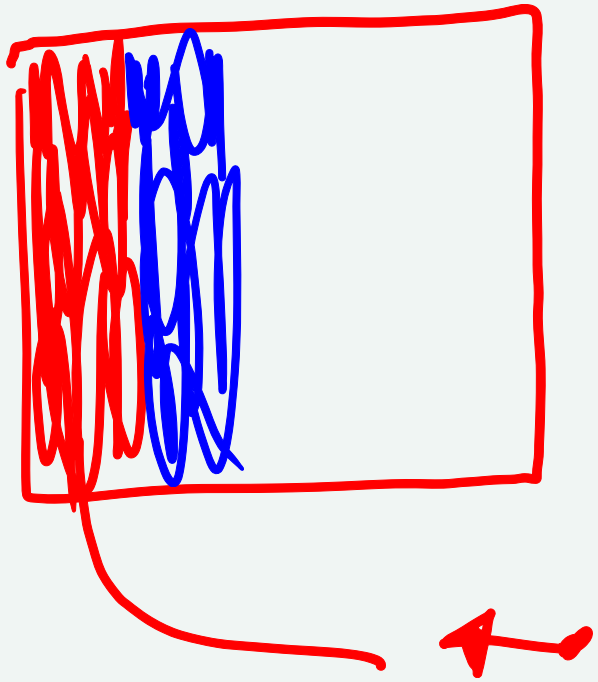
vector relative to real normal

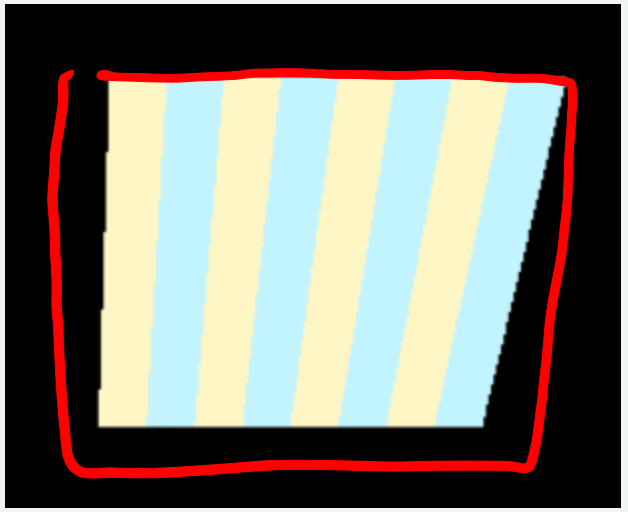


Wavy with Normal Map

One square (2 triangles)

Simple to paint pattern



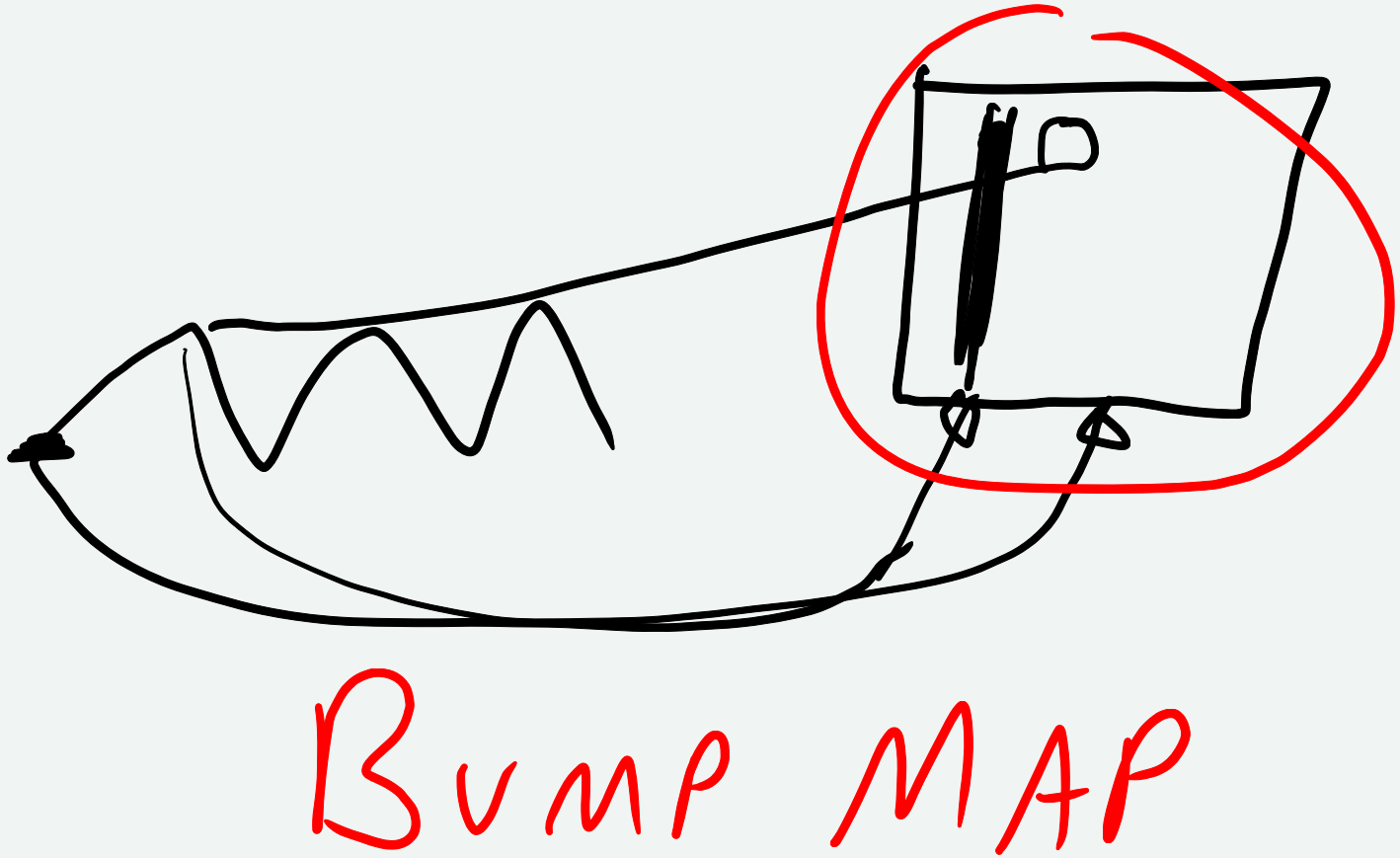


It's a pain to do XYZ

hard to think about

paint 1 dimension instead?

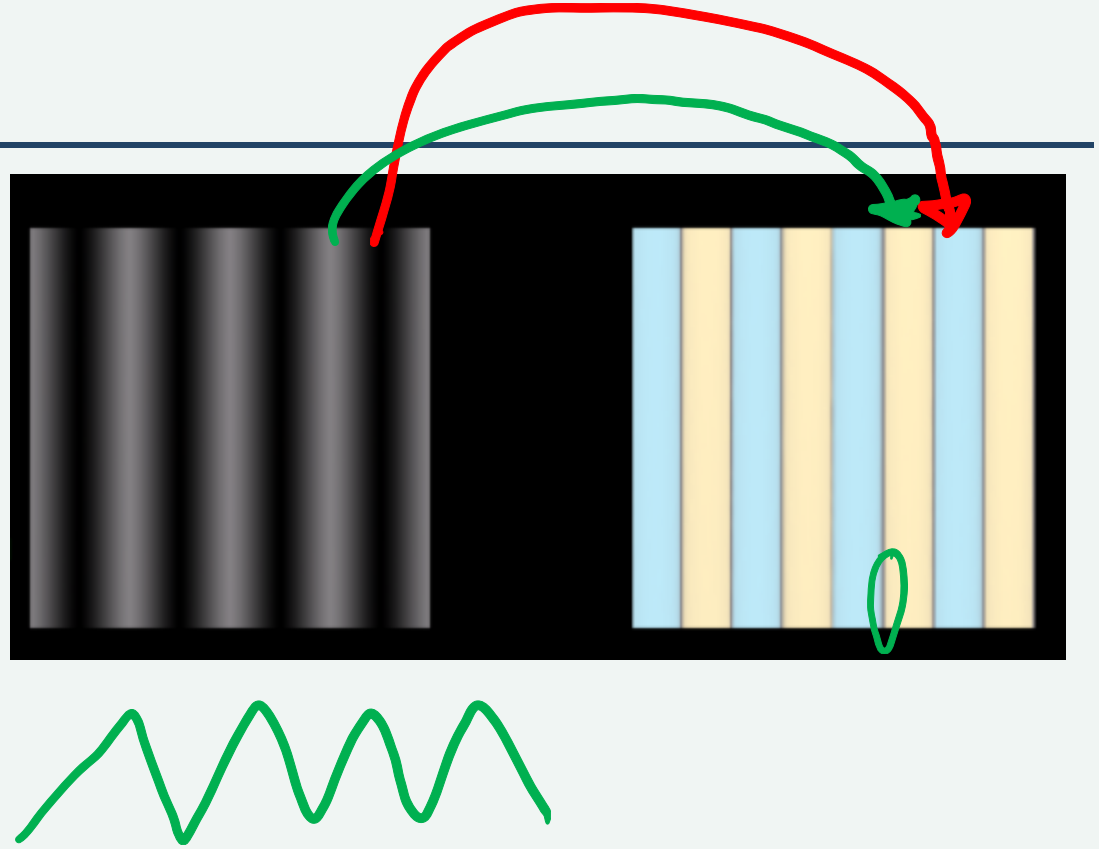
height field



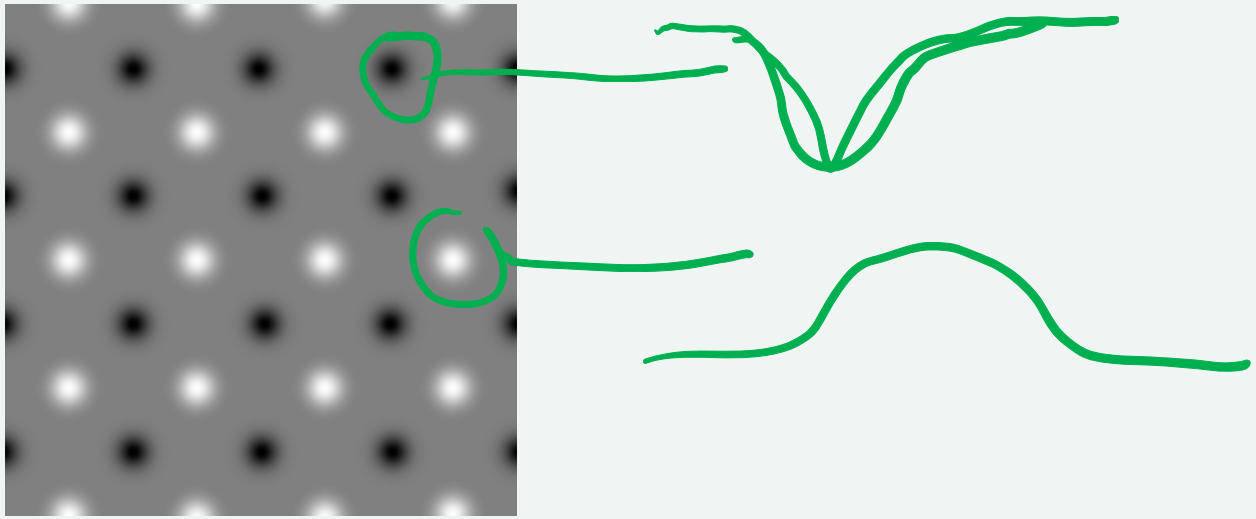
Bump Map

One square (2 triangles)

Simple to paint pattern



A more realistic example

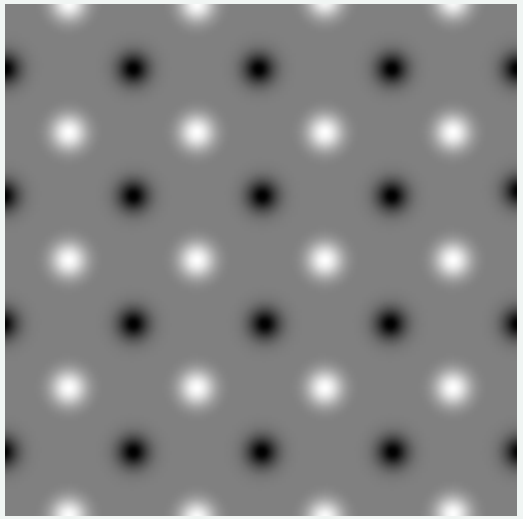


Gray = middle, black=down, white=up

(it's all relative)

A more realistic example

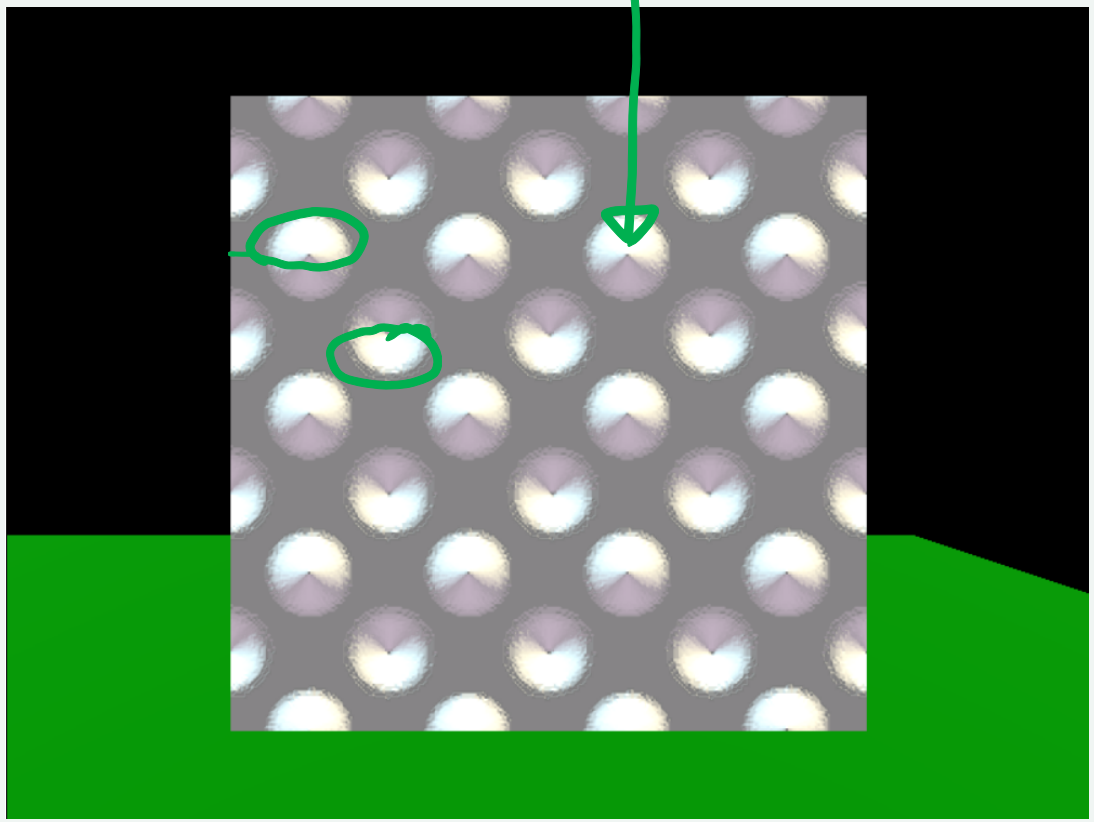
white



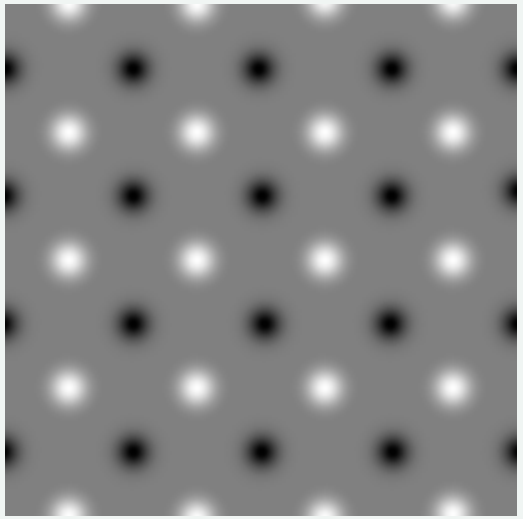
Bright from above

Dark from Below

Lighting not reflection!



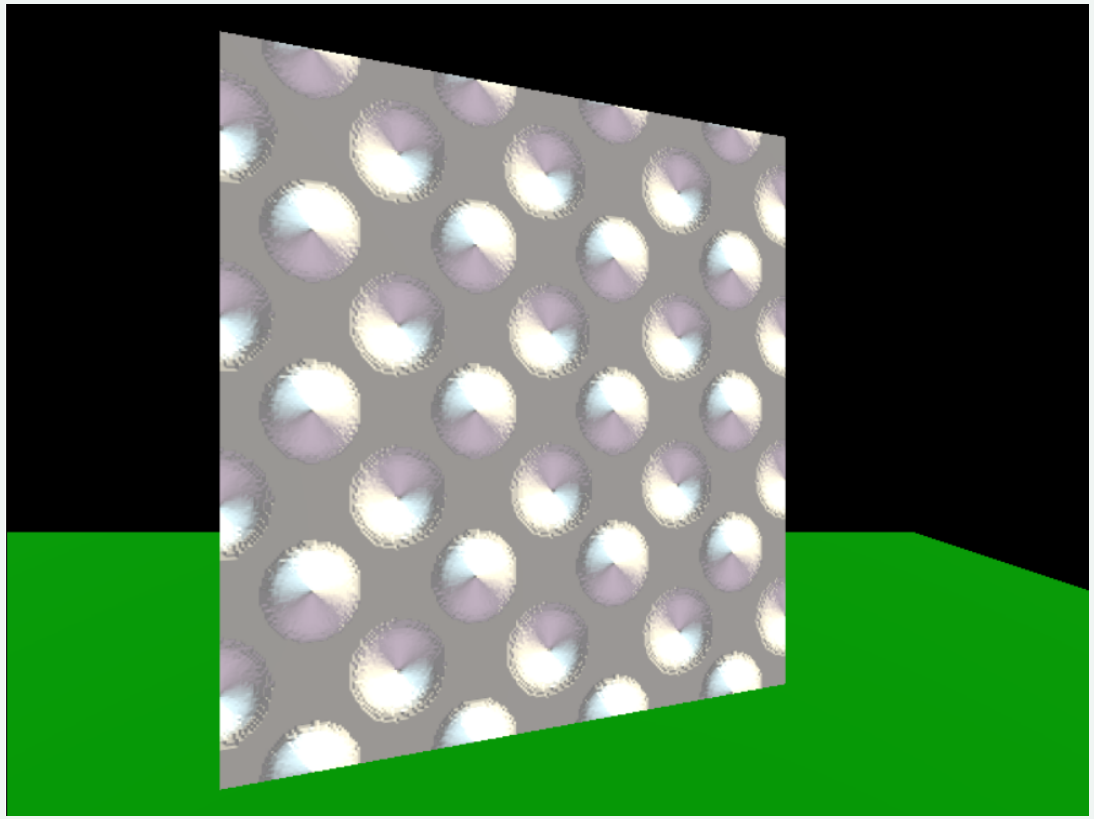
A more realistic example



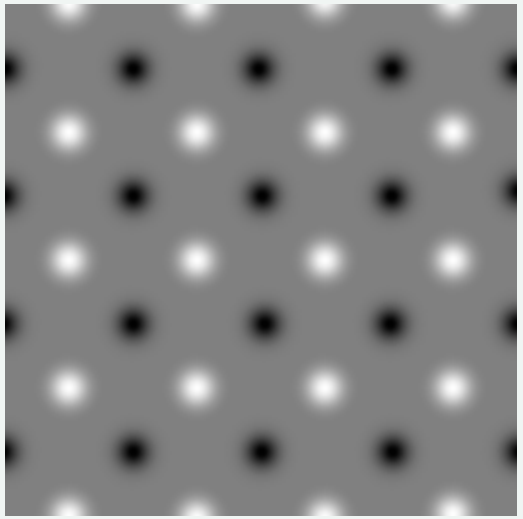
Bright from above

Dark from Below

Lighting not reflection!

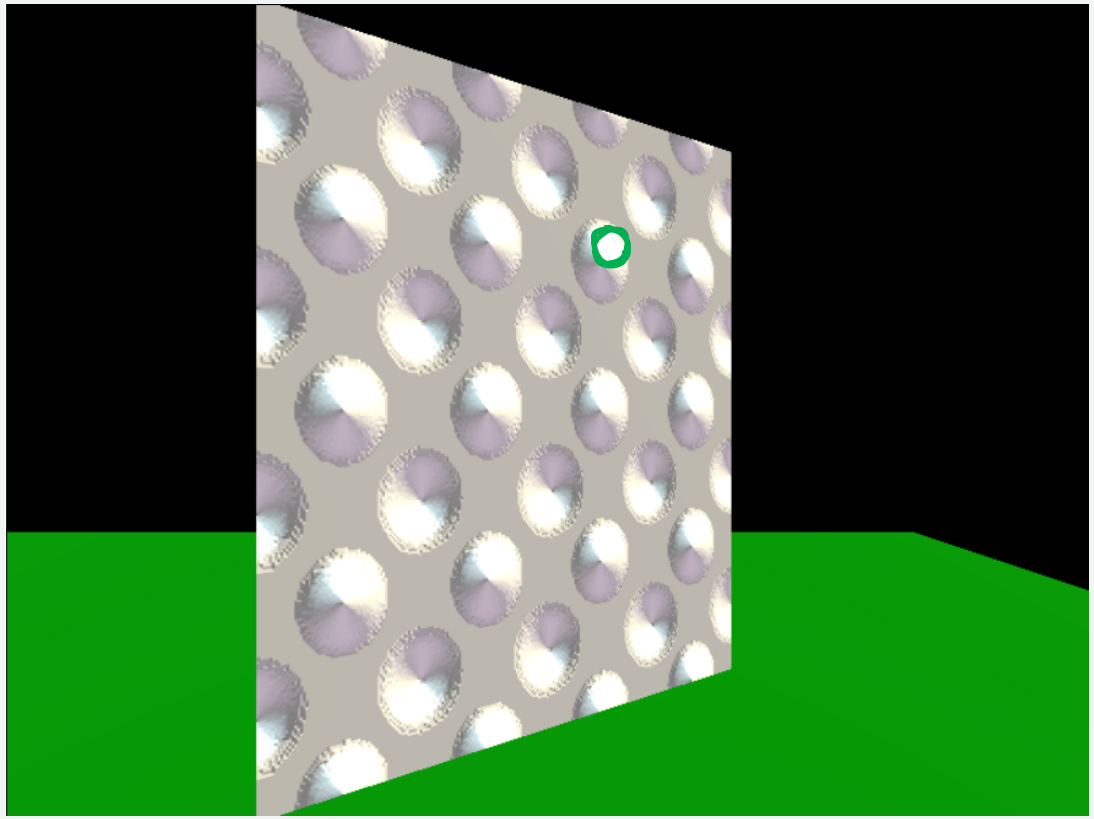


A more realistic example

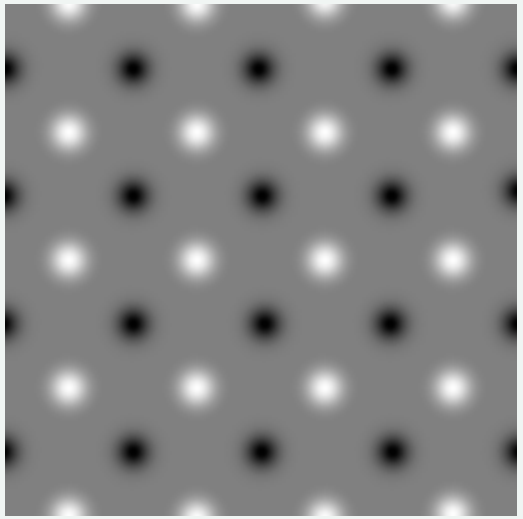


In motion, it can be convincing

(if you don't look too close)



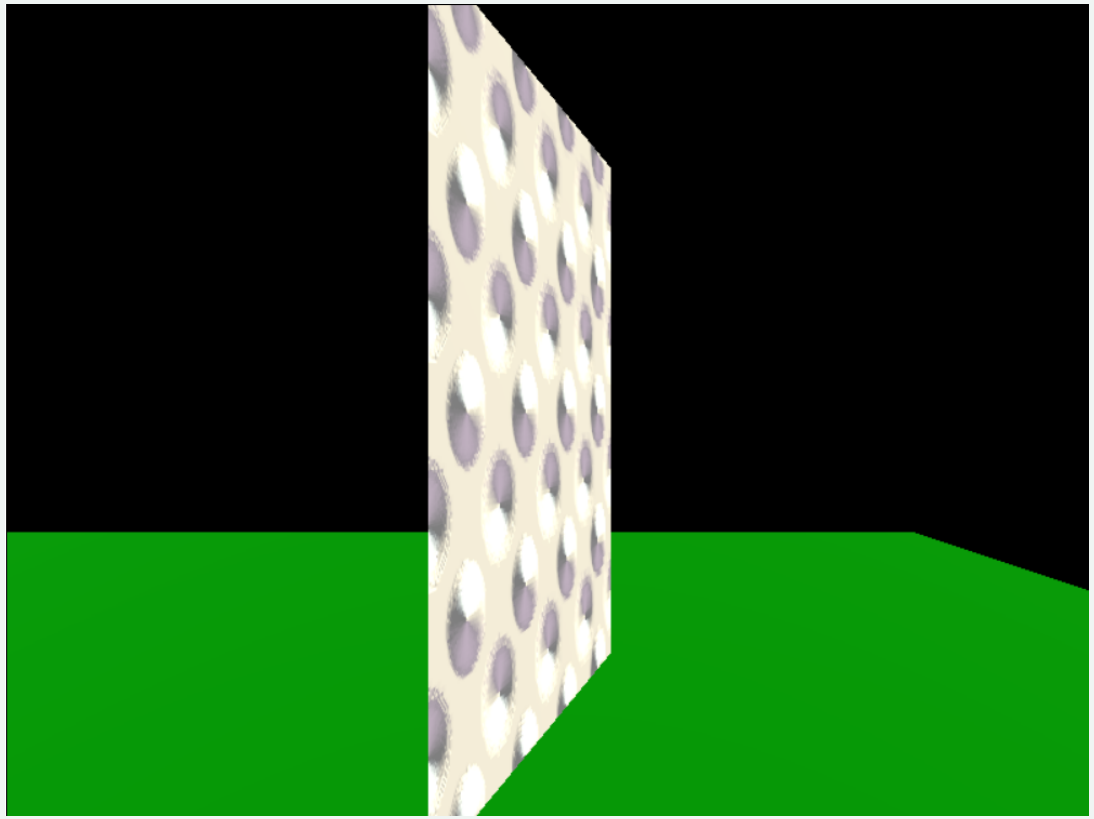
Where it breaks



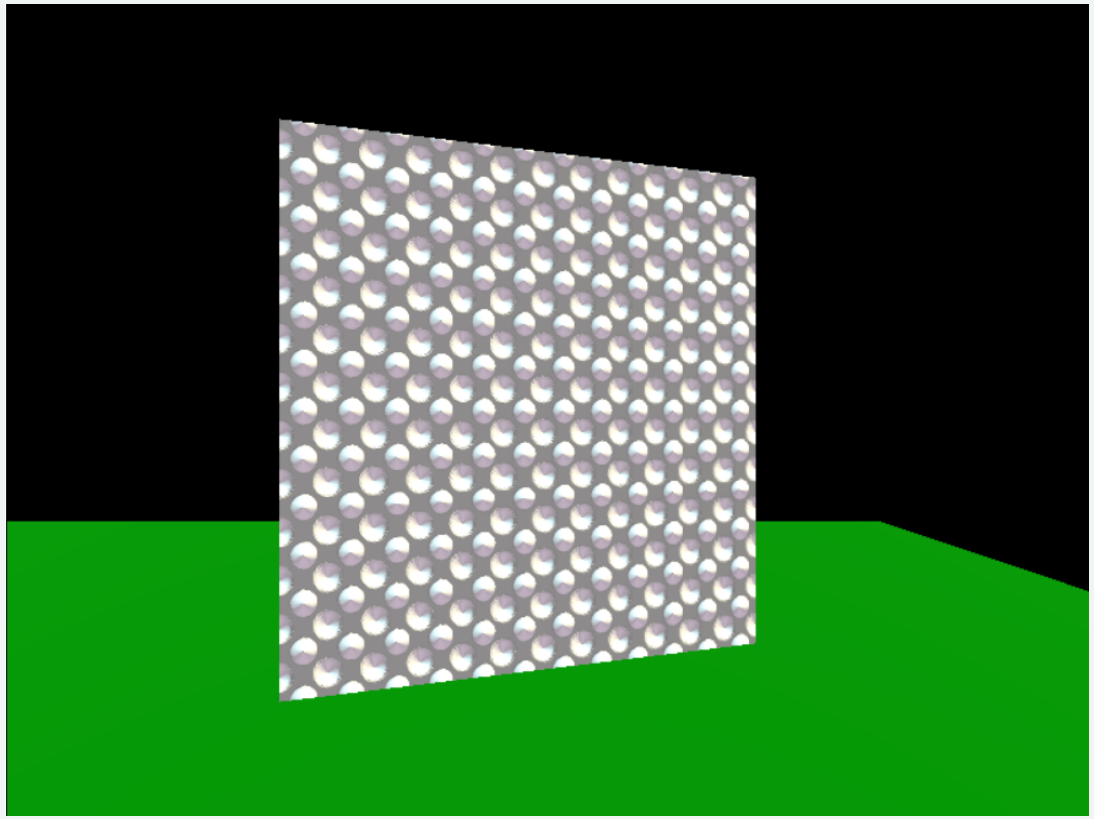
Doesn't change the shape!

It's still flat

Doesn't change the silhouette



Works better if subtle/small/moving



Easy in THREE.js!

```
let bumps = new T.TextureLoader().load("dots-bump.png");
let mat = new T.MeshStandardMaterial({bumpMap:bumps});
```

Normal Maps and Bump Maps

Good

Easy to specify surface details

Doesn't actually change shape

Gets basic lighting effects

Works with lighting

Easy in THREE

Bad

Doesn't change side view

Doesn't cause occlusions

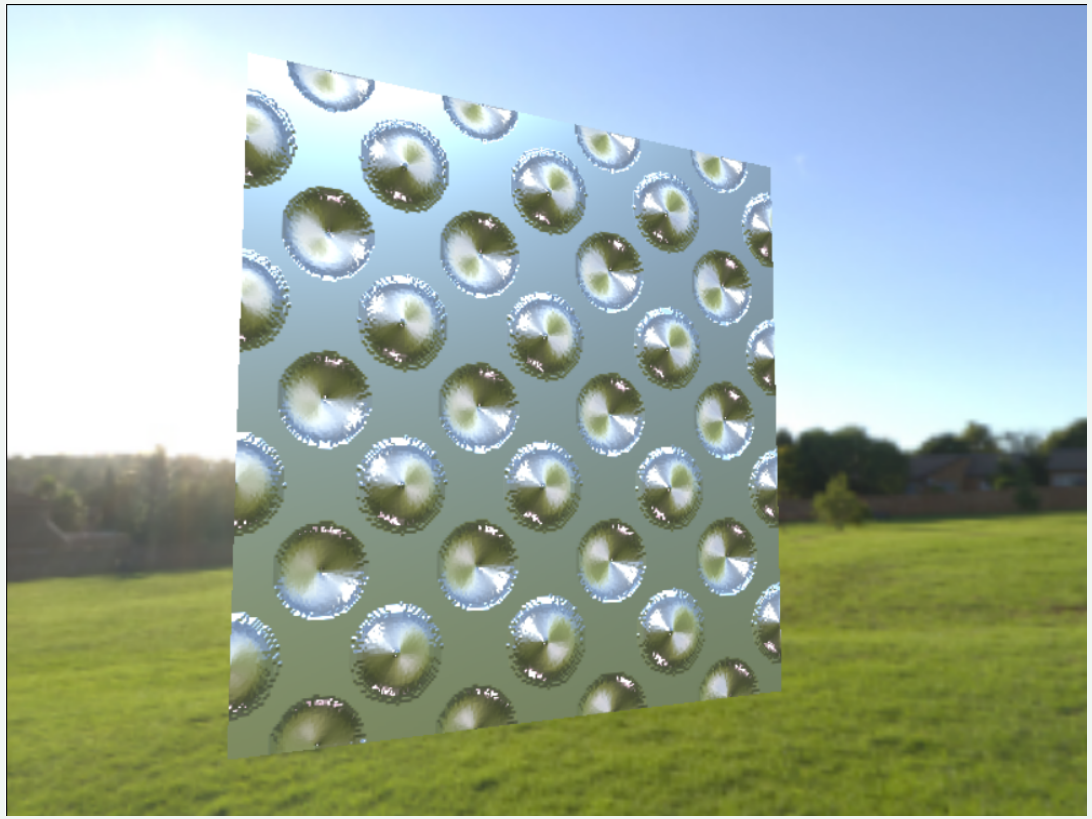
Doesn't work for big effects

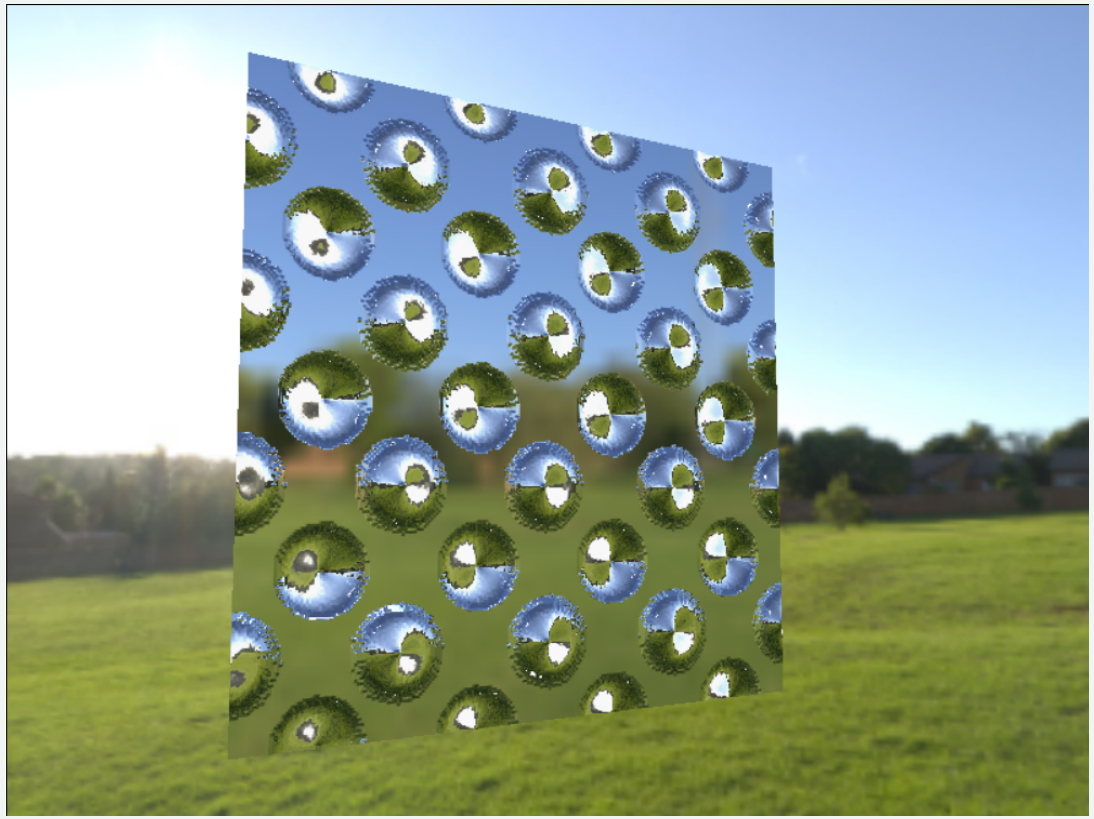
Doesn't cause shadows

Need something else for reflection

(wait a bit)

Coming Attraction...

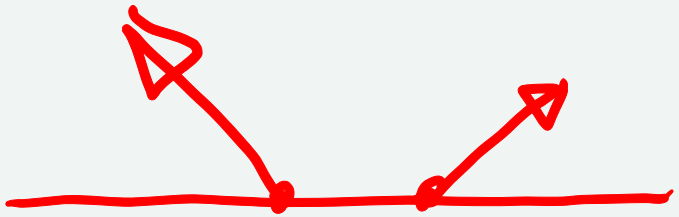


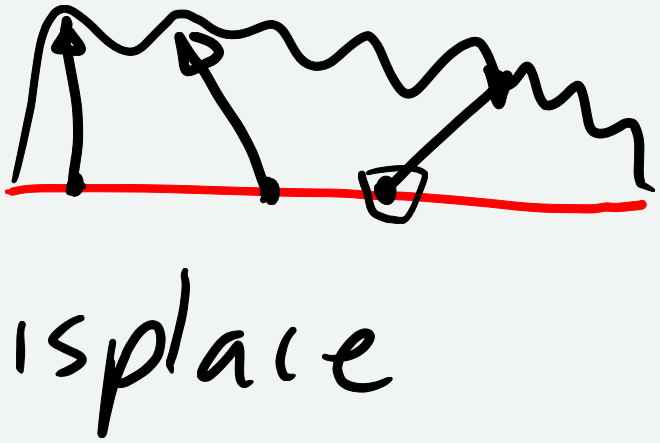


Change geometry?

RGB -> XYZ - could be a displacement

Displacement Maps!

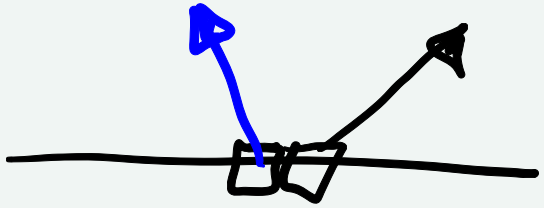




Why not Displacement Maps?

Much harder to implement

- actually moves pixels (can't do per-pixel)
- may cause gaps
- doesn't fit hardware model



Summary

- Fake Normals to get Smooth Surfaces
- Fake Normals to get Bumpy Surfaces
  - Bump Maps
  - Normal Maps
- Normal/Bump Maps are not ~~Displacement Maps~~

CS559 Lecture 19-20: More Texture

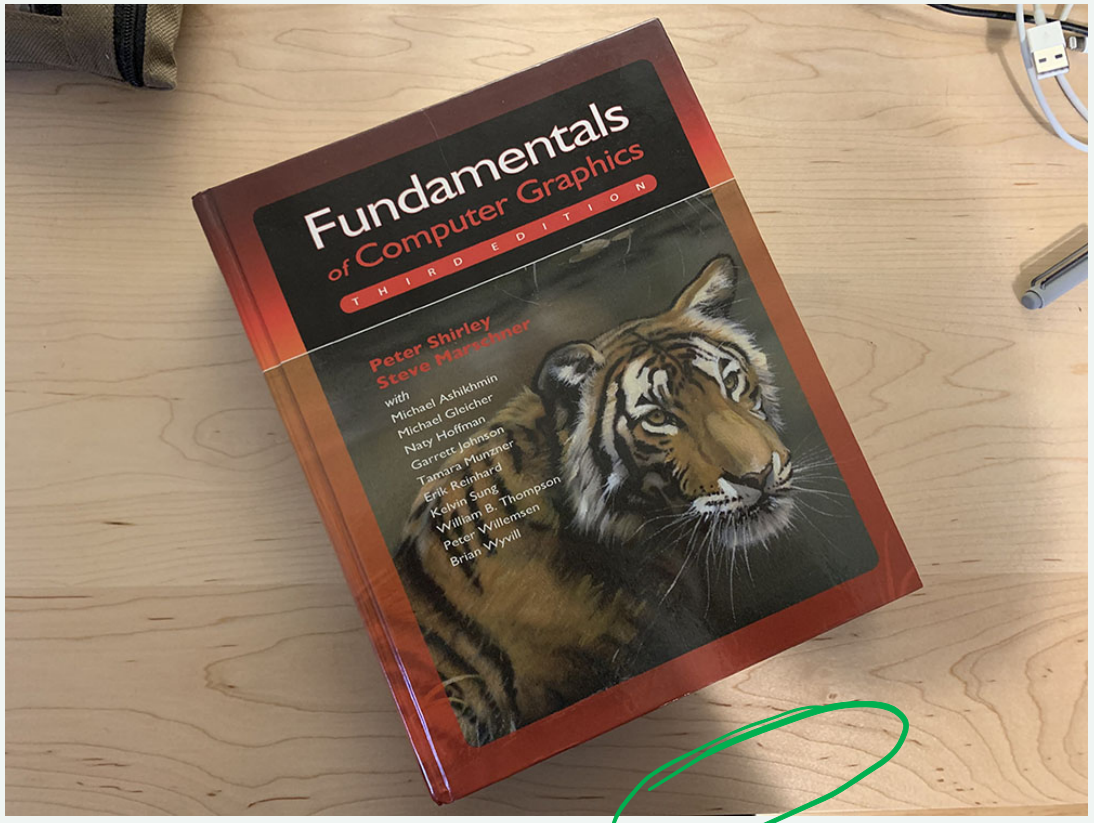
Part 3 - Other Things to Do With Textures

A few smaller topics...

- dealing with patterns
- layering
- "baked in" lighting
- ambient occlusion
- solid tectures

Still more to do...

Real objects are interesting



Still need the wood, the lights, ...

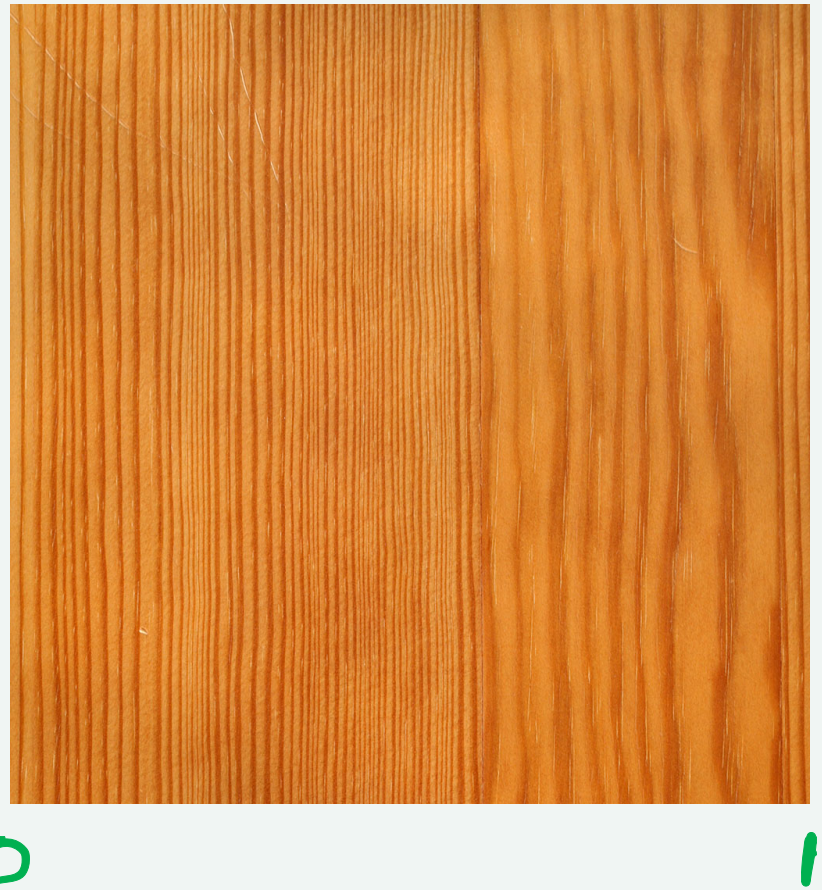
Let's make woodgrain!

Find a texture on the web:

https://freestocktextures.com/texture/wood-board-wood-grain,78.html

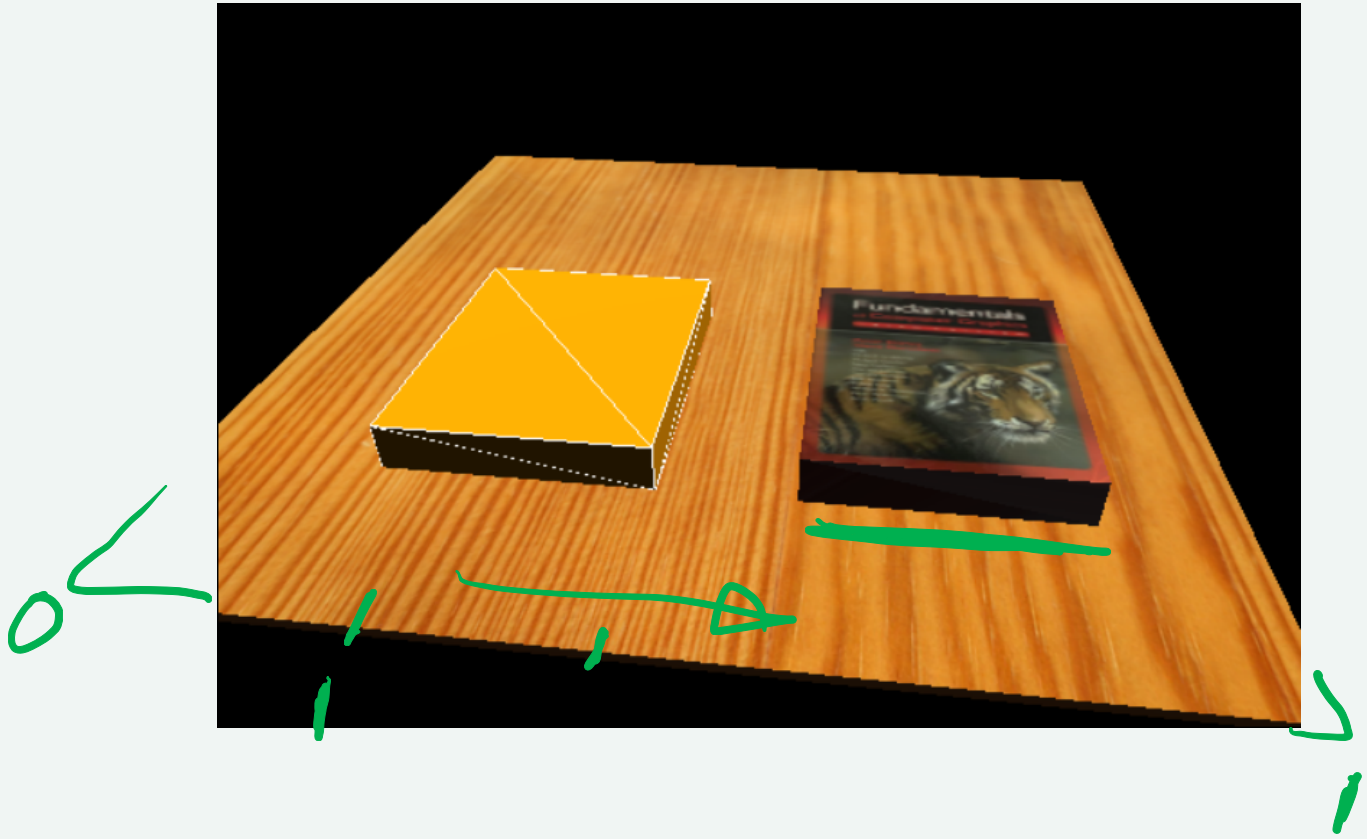
A usable texture?

Needs to be a square



Apply it to the table...

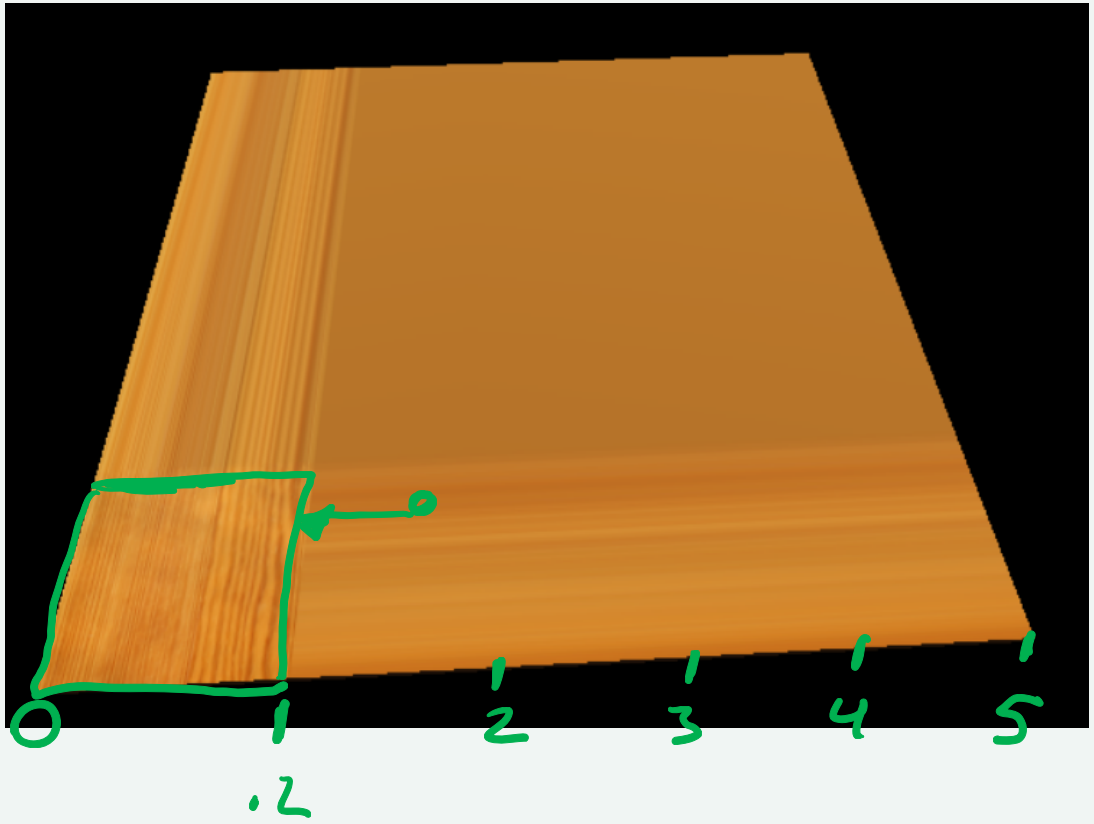
Giant Wood Grain



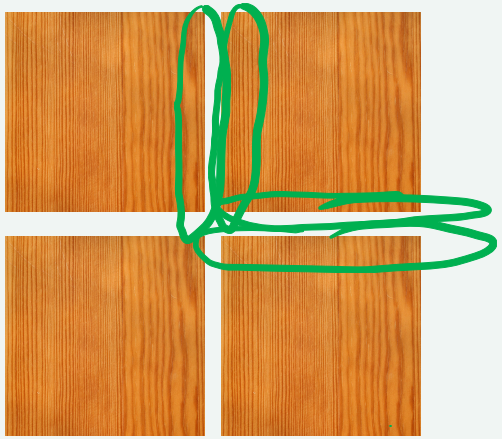
Scaling

U,V values beyond 1 

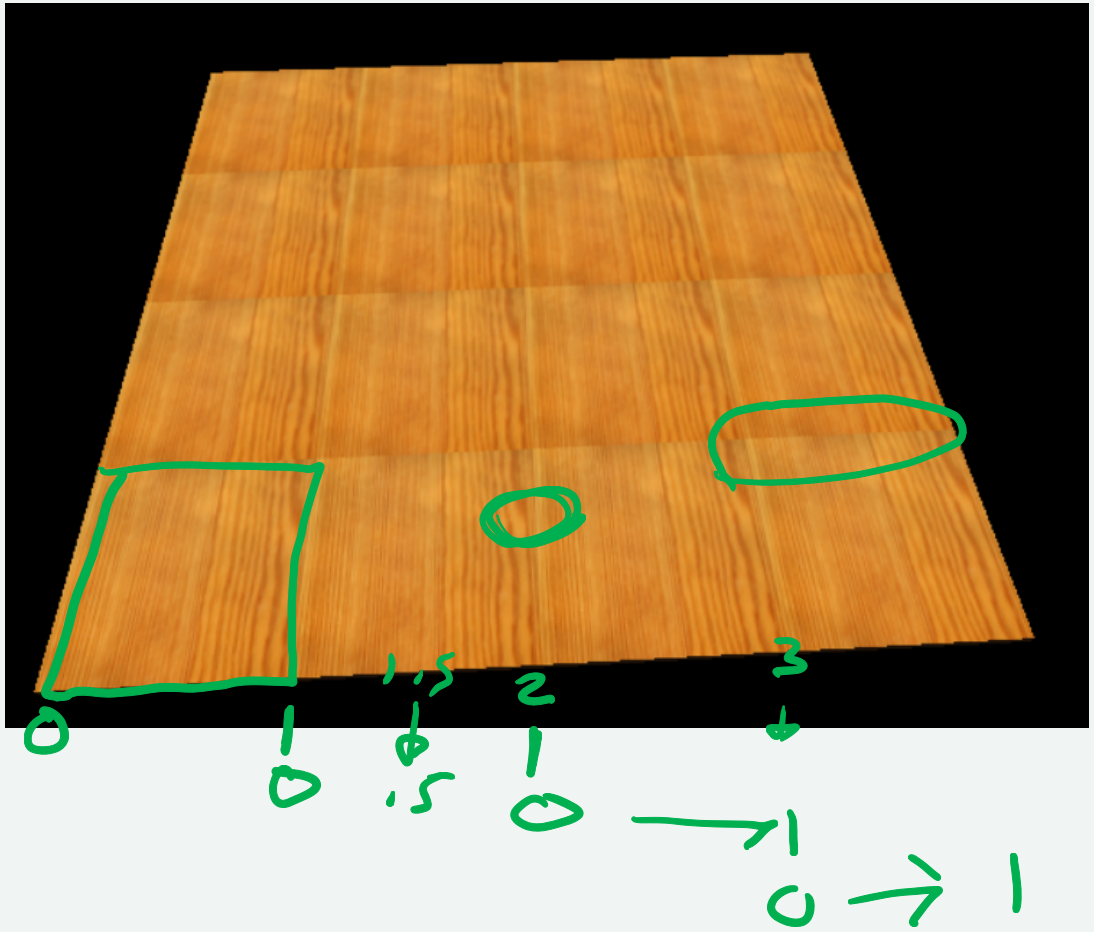
Clamping



Repeat (tiles)

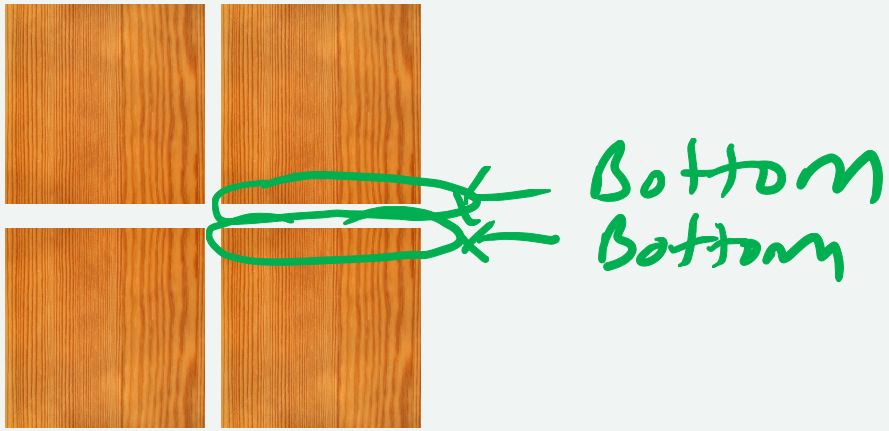


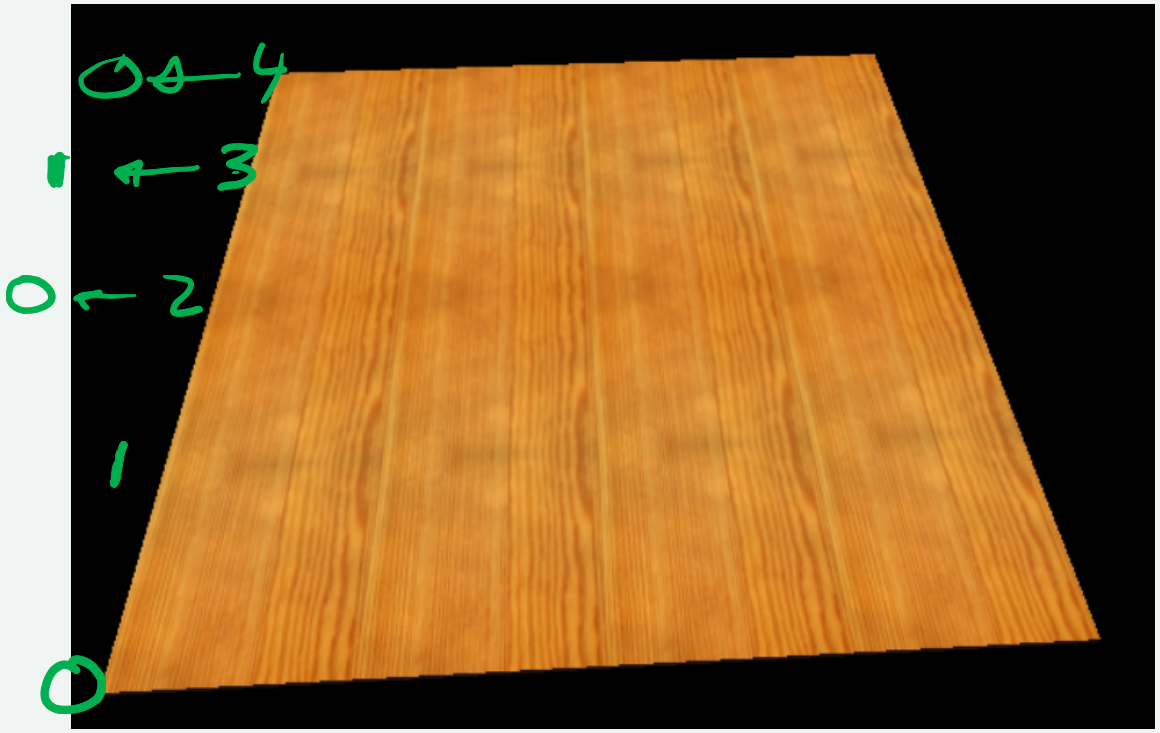
The edges need to fit together



Mirror (tiles)

Mirror Repeating

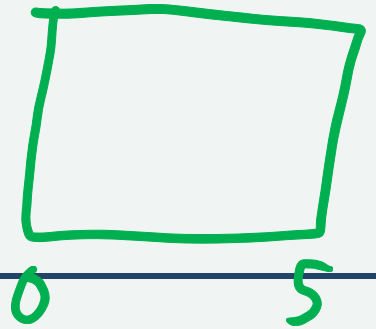




This only flipped Y

Sometimes called **bookmatching**

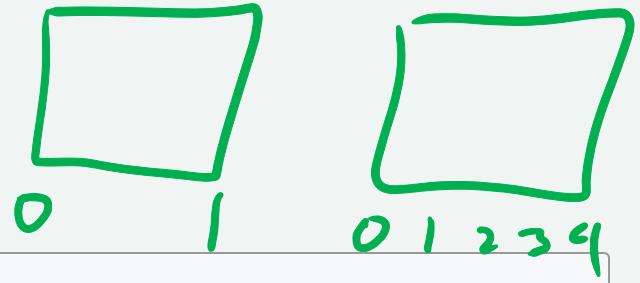
In THREE.JS



Of course, they make it easy!

You can specify UV values that you like on objects.

Or (if you are stuck with primitives with U,V in [0,1])




```
texture.repeat.set(x,y);

texture.wrapS = T.RepeatWrapping; // or T.ClampToEdgeWrapping
texture.wrapT = T.MirroredRepeatWrapping;
texture.needsUpdate = true;
```

S = V

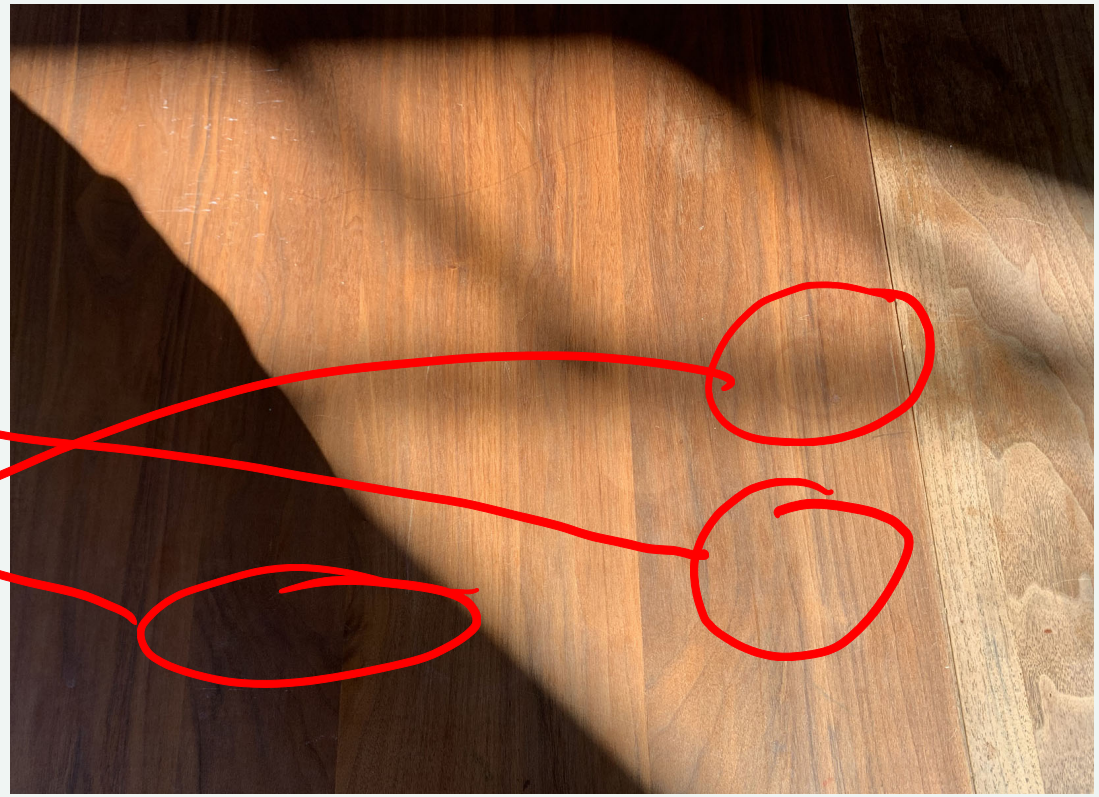
T = V

But I want my walnut table!

A Real Photograph can have:

- Not aligned correctly

- Highlights (lighting)
- Shadows (lighting)
- Dirt / Imperfections



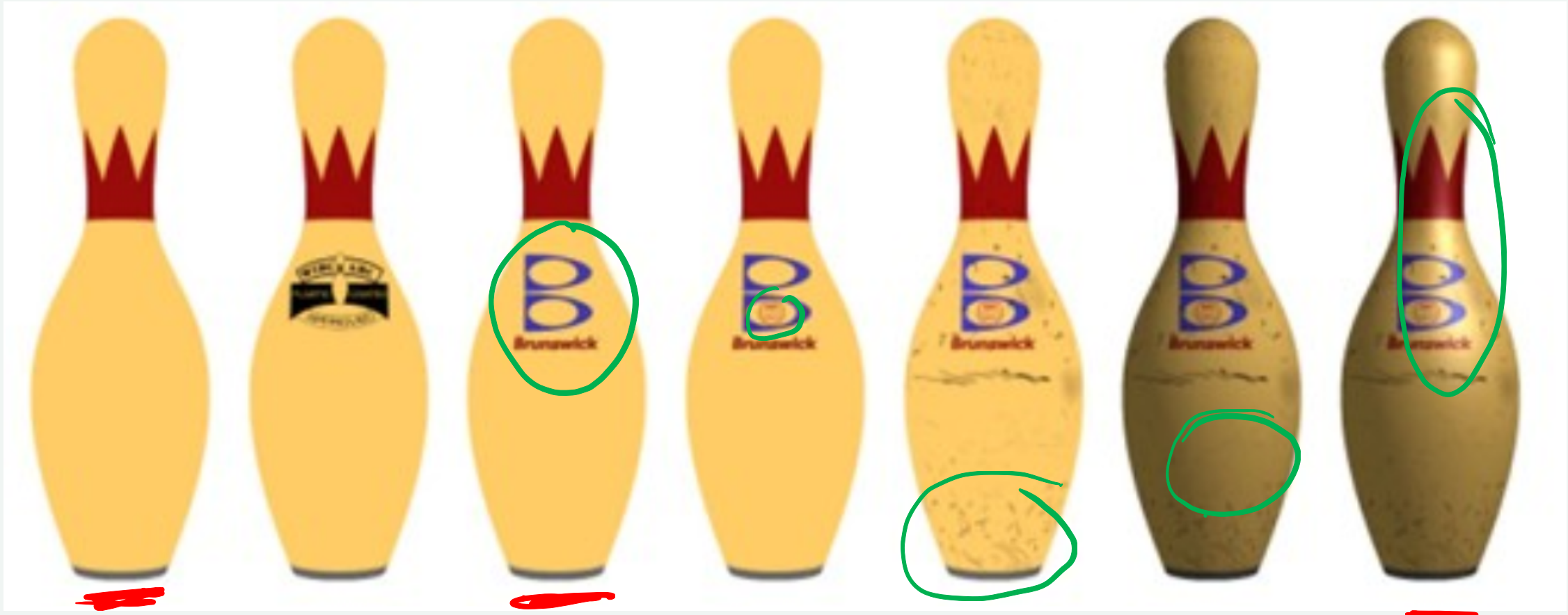
Maybe this is a feature?

Use textures to get the complexity of the world!

- Dirt and small details

- Capture Lighting Effects that we can't easily make

Layered Textures (Multi-Texture)



(old pixar example)

Combine...

Use multiple textures

1. need different U,V values for each one
2. need to blend colors together
3. need to choose textures that work together

In THREE #1 is what layers are for, #2 is not built in

Light Maps

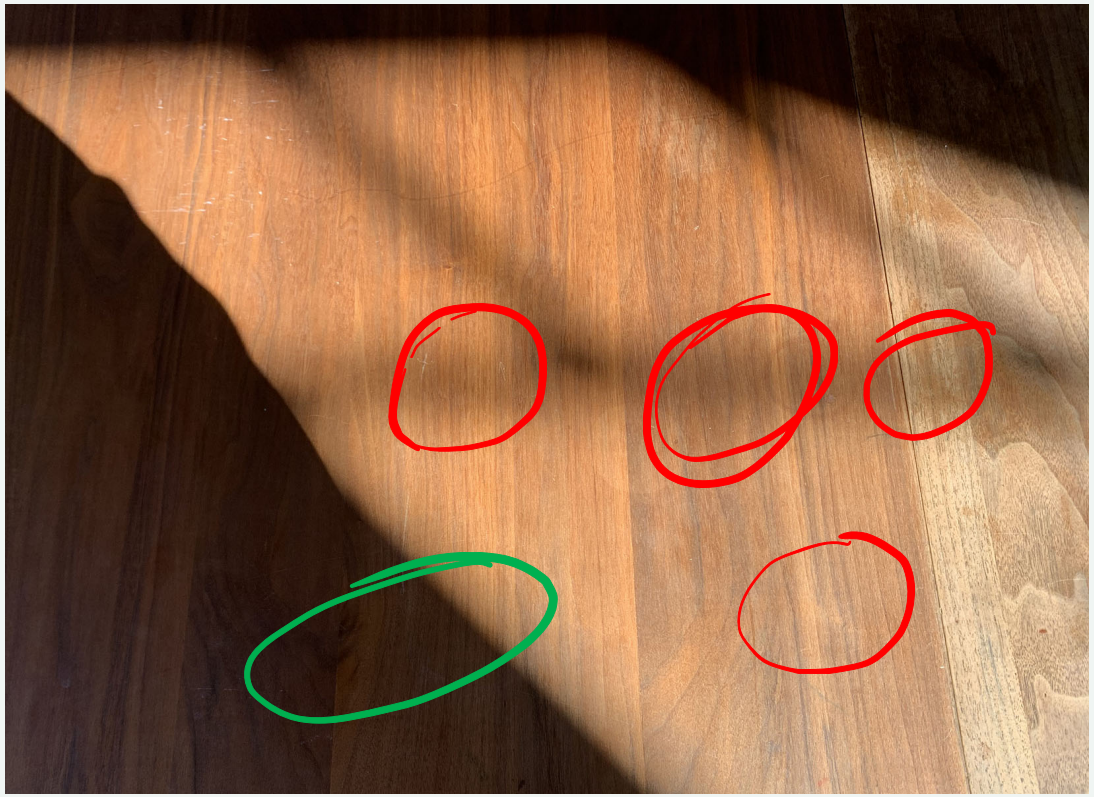
Put lighting into the texture

Good:

- can be as fancy as you like
- pre-computed!

Bad:

- lighting must be known ahead
- can't change
  - camera moves
  - objects move



A Special Kind of Pre-Computed Light

Self shadowing

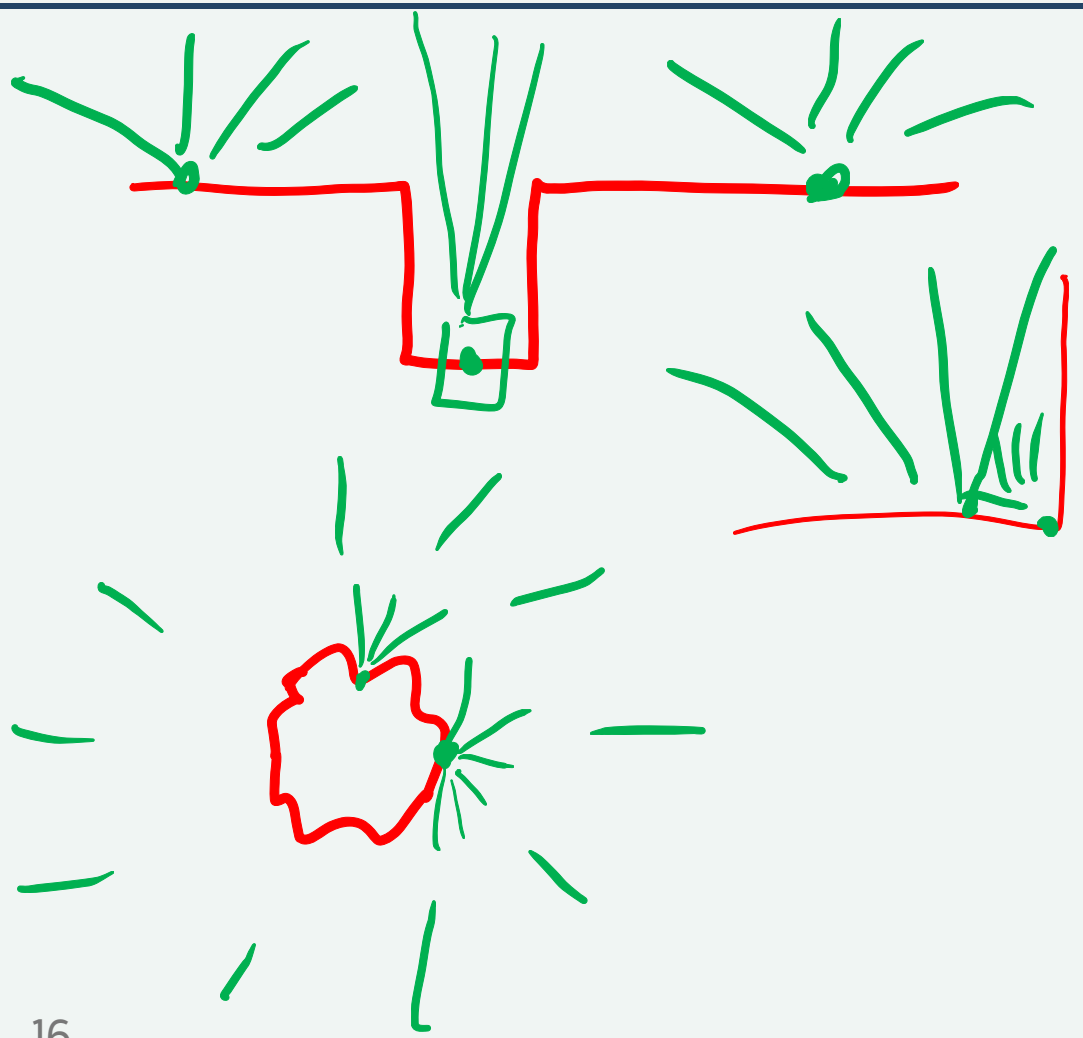
- important for conveying shape

Pretend light comes from all directions

- like ambient lighting

Amount each point is "visible"

Corners and crevices are dark

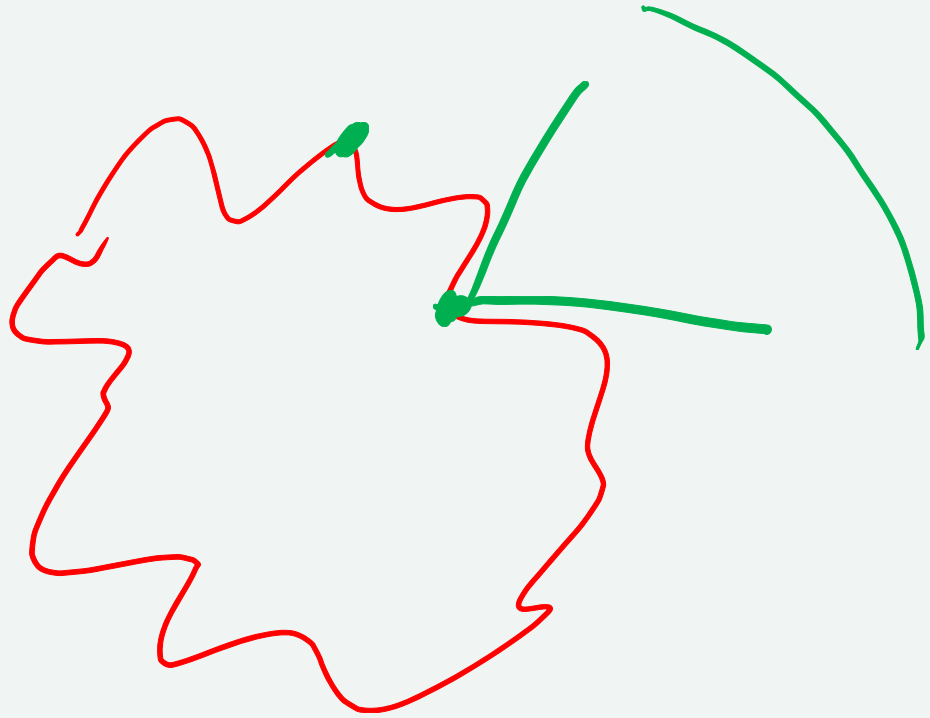


Ambient Occlusion Shading

Pre-compute for all points

- use special tools
- or clever hacks

Used like a light-map



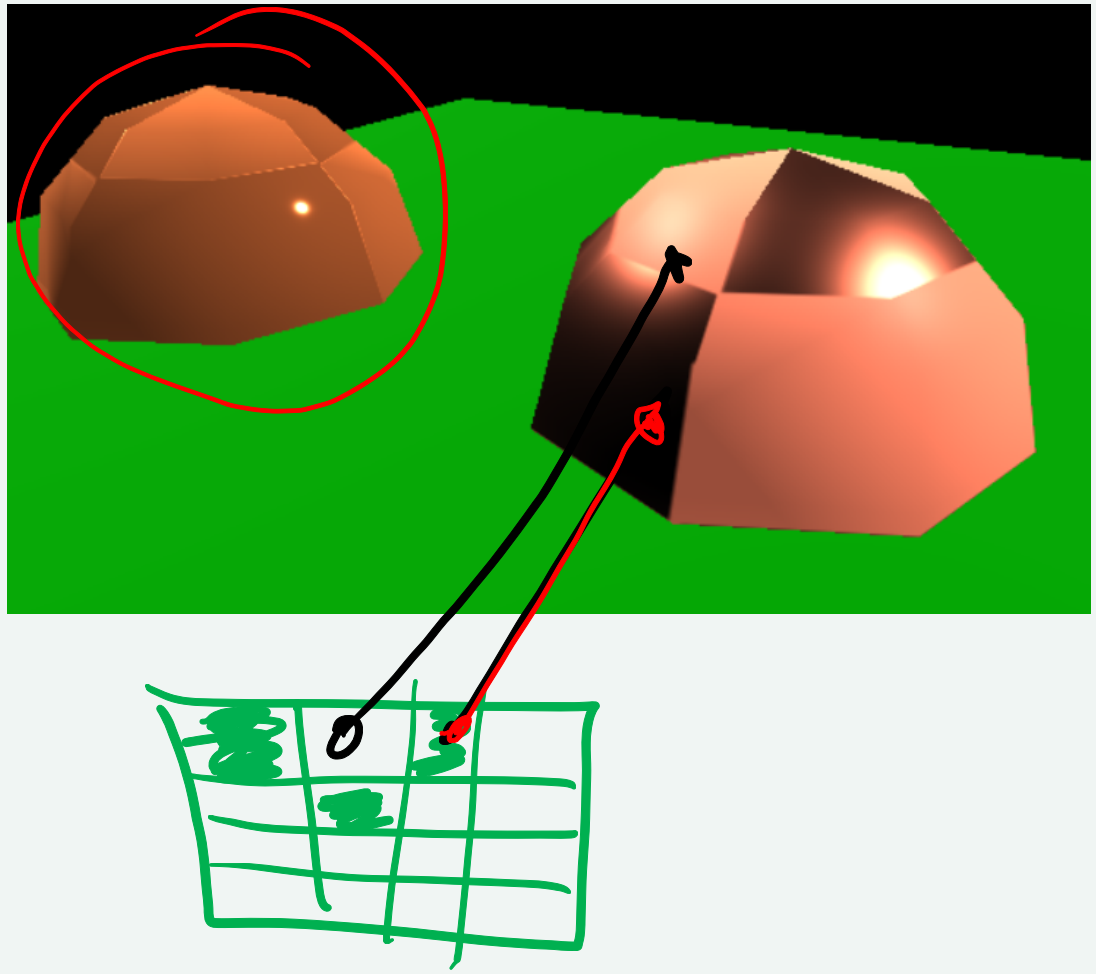
https://vr.arvilab.com/blog/ambient-occlusion

What to change with texture?

- Colors
- Normals

- Other Material properties

Material Property Maps

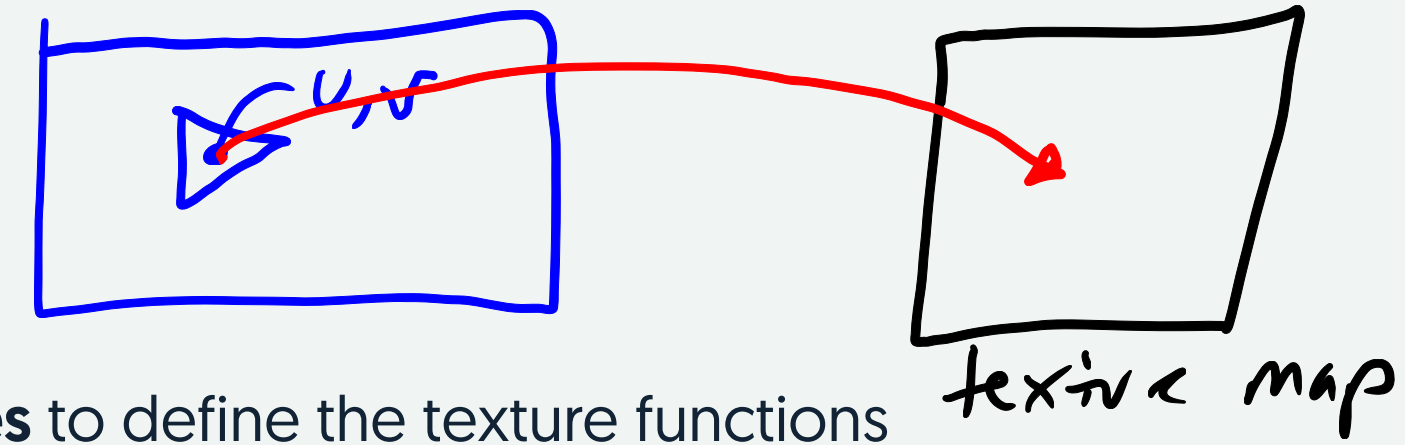


Thinking about texture...

Each pixel has a (u,v)

Some function

(u,v) -> (r,g,b)



We can write **procedural textures** to define the texture functions (coming in a few weeks)

function

Solid Textures

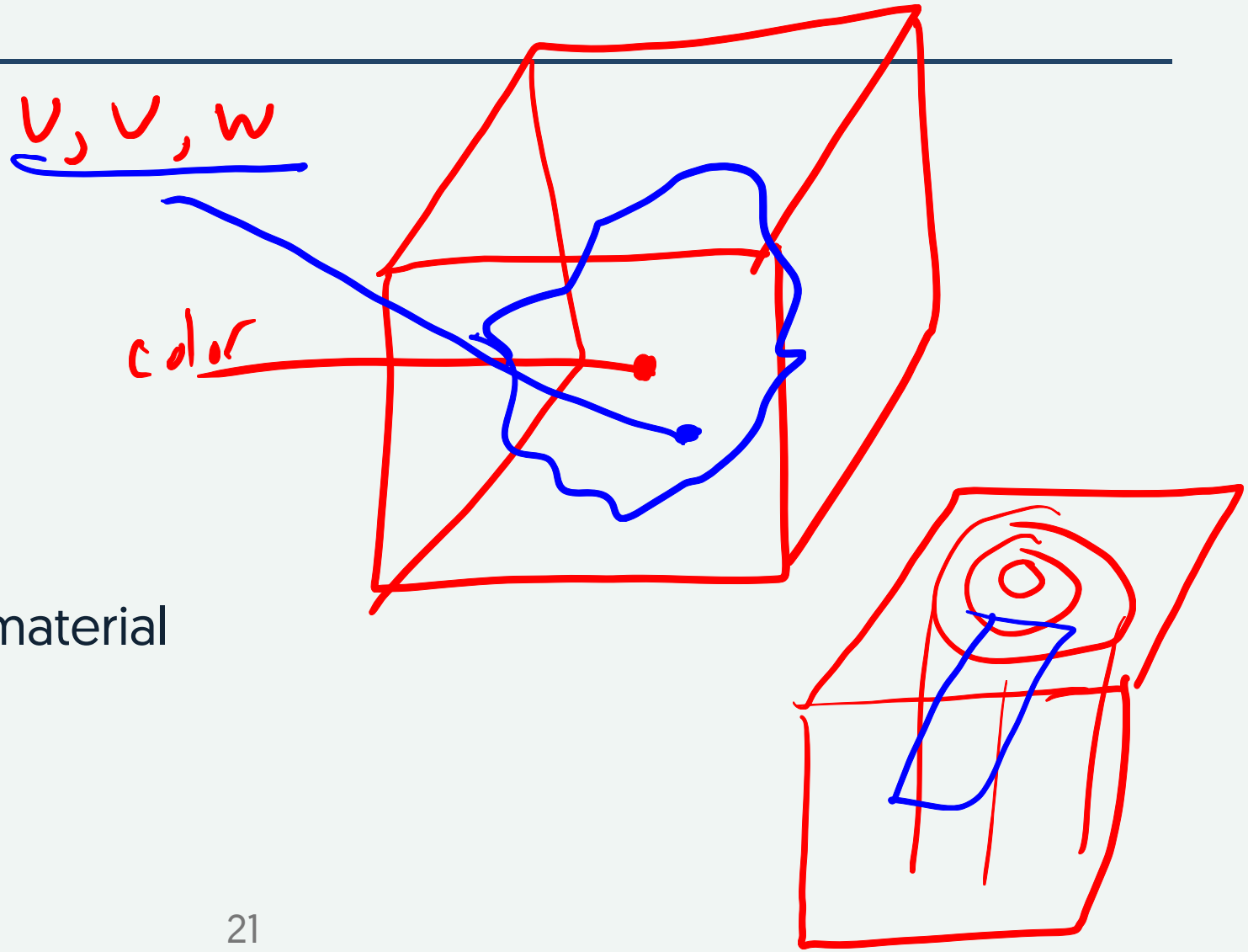
Points have 3D coordinates

Look up values in 3D

Useful for 3D materials

- wood
- stone

Like carving the object out of material



Summary

- Texture Scaling, Wrapping, Wrapping Modes
- Layer Textures for Other Effects
- Light Maps for pre-computed "baked in" lights
- Ambient Occlusion to get cool effects
- Procedural and Solid Textures in the future

Next: using other ways to generate coordinates to get lighting

CS559 Lecture 19-20: More Texture

Part 4:
SkyBoxes and Environment Maps

How do we fake lighting?

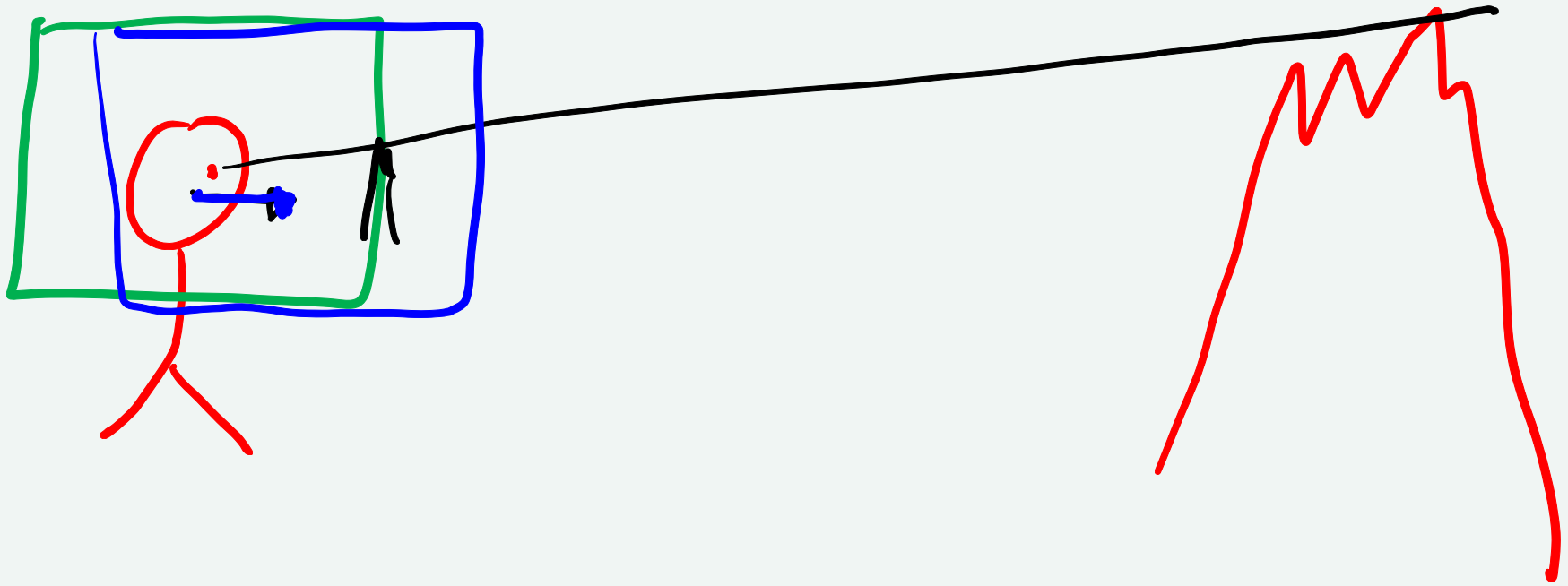
- Fake stuff far away
- Fake reflections by pretending things are far away

Do we have to draw everything?

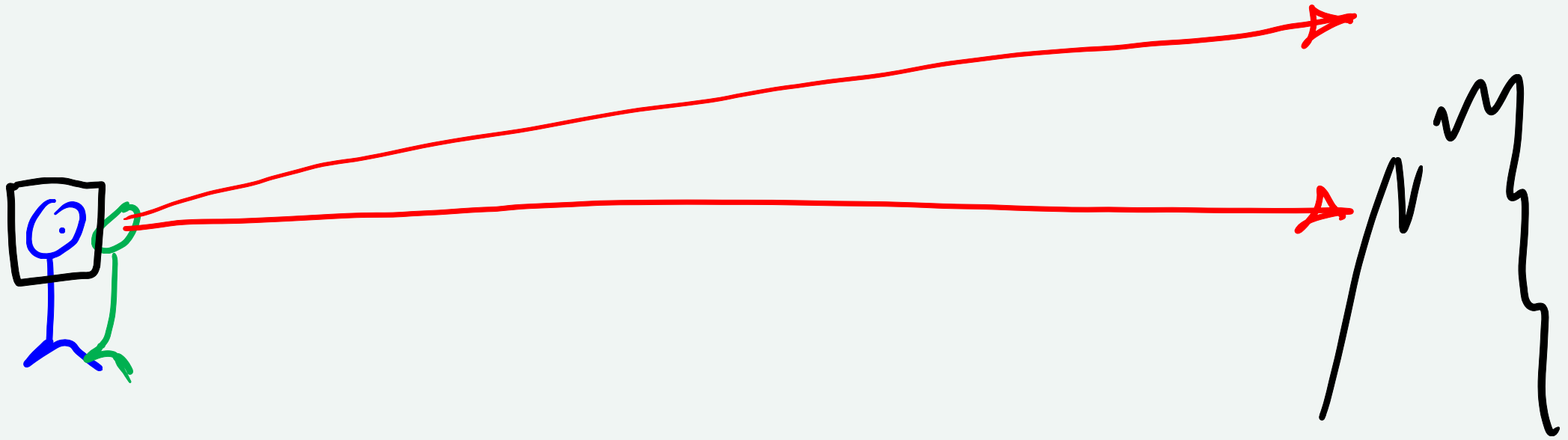
Why not use a (pre-computed) picture?

Far Away and Not Changing

Could draw a box around the viewer



Positions do not matter
Orientations do



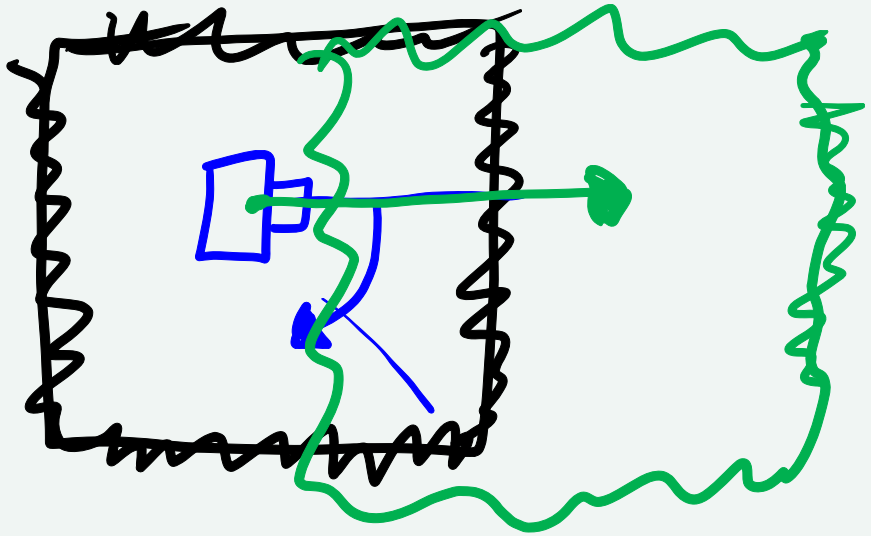
Sky Box

A Box for the "Background"

Always stays centered at the camera

The box moves with the camera)

(beware fake sky boxes)



In THREE.js

Sky Box is built in! (no excuse for fake!)

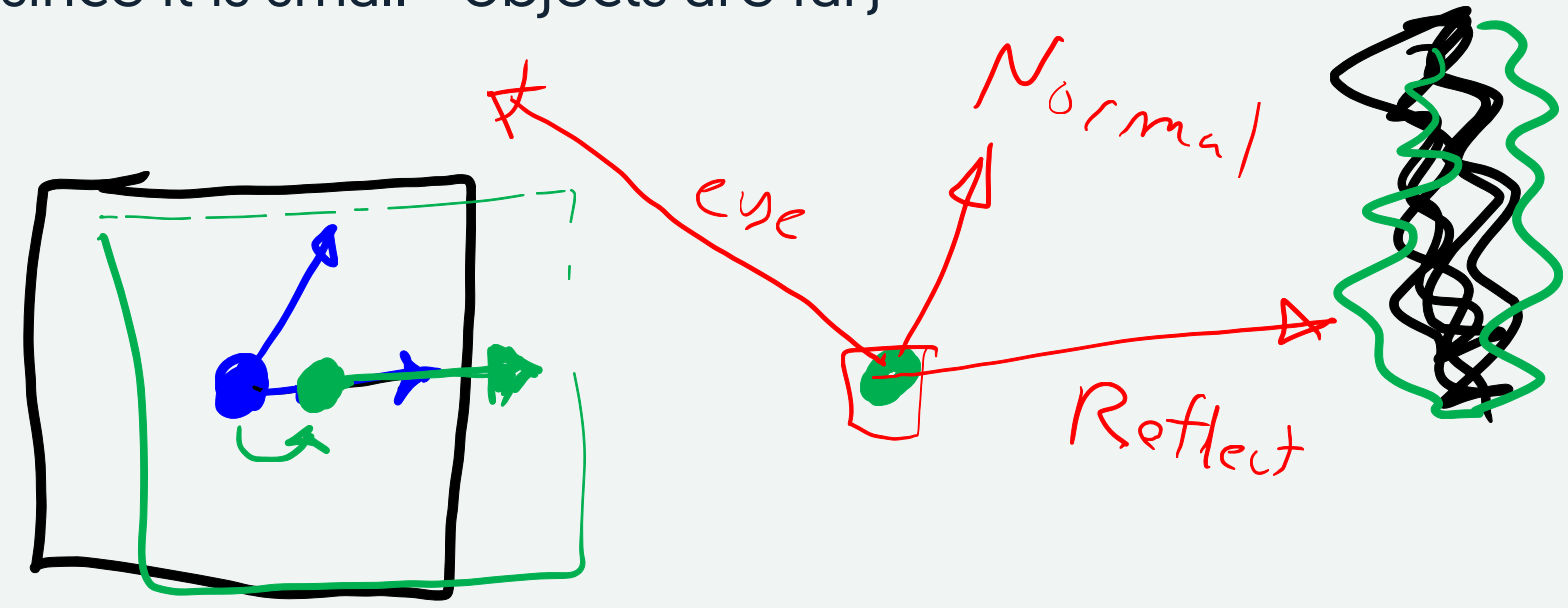
- Cube Texture (6 images)
- `scene.background` = some cube texture

How does this help with reflections?

Assume the objects being looked at are far away (a Sky Box)

Viewing direction matters

Point Position doesn't (since it is small - objects are far)

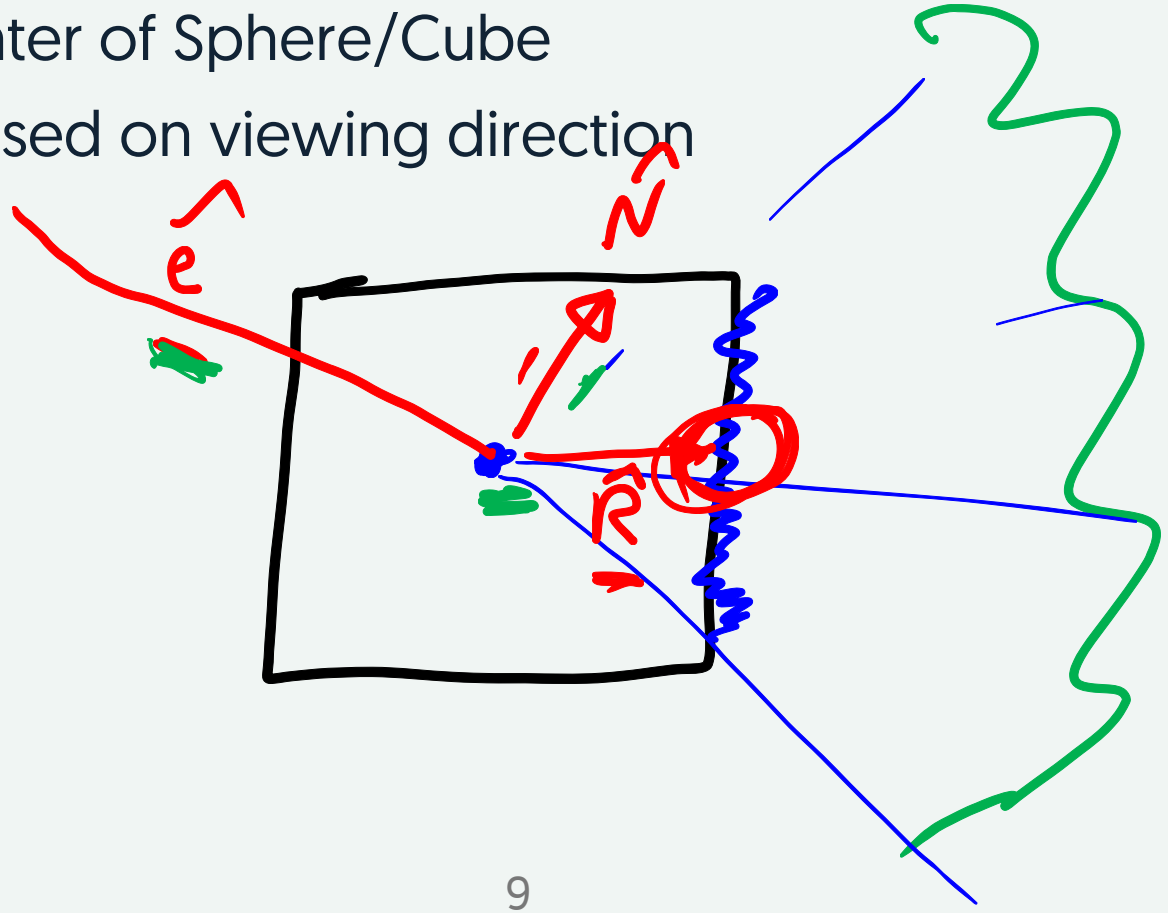


Environment Mapping

A Cube or Sphere texture map (around object)

Assume point is at center of Sphere/Cube

Lookup direction is based on viewing direction



How to store the maps

Sphere Maps

(Equi-Rectangular)

Cube Maps

Representing the Environment

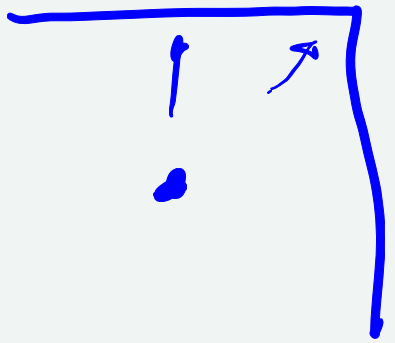
Sphere Maps

- Sampling issues at pole
- Single Images
- Capture in 1 Photograph

Tools to convert images between forms

Cube Maps

- Sampling Issues in Corners
- Images are human viewable
- Maps nicely to graphics hardware



When do Environment Maps Work?

Require assumptions:

- Small Object / Far Environment
- Eye is far (only direction matters)
- Position doesn't matter

Small curvy objects - good

Large flat objects - bad

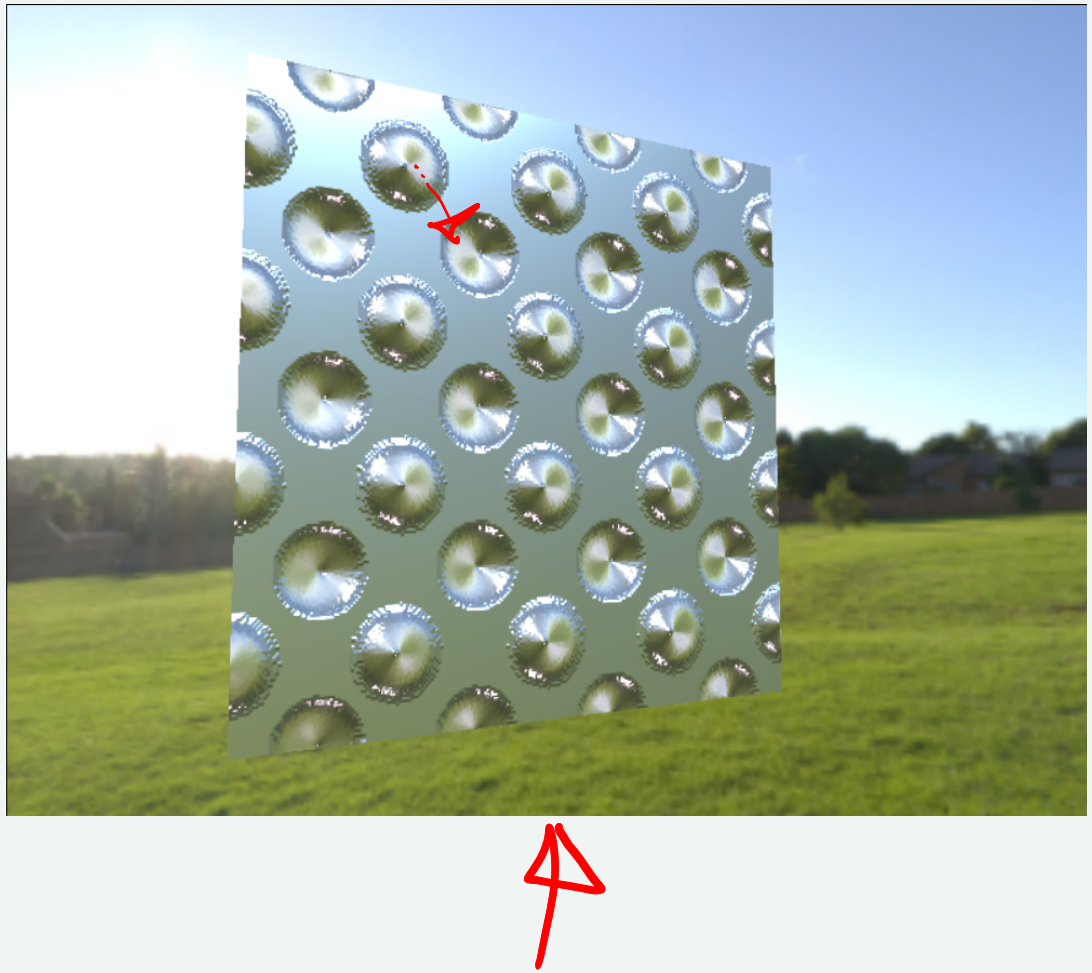
Very easy in THREE

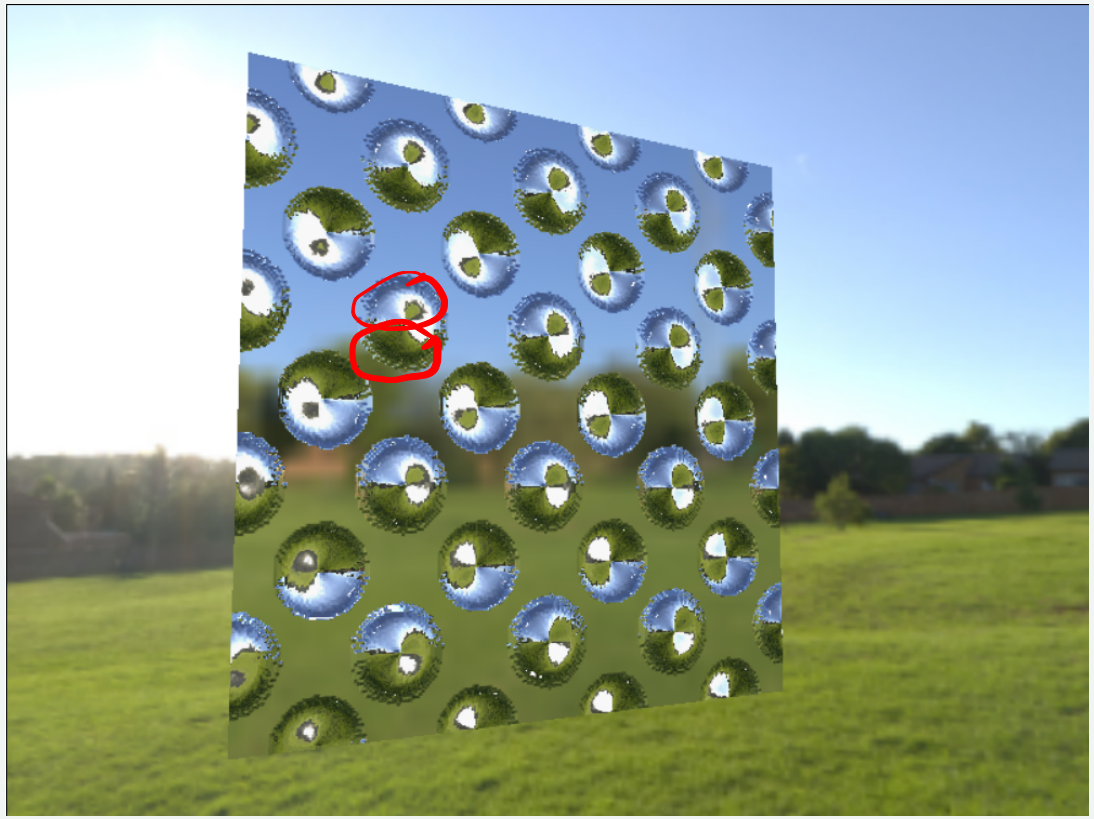
```
// assuming that cubeTexture is a Cubic Texture Map
let mat = new T.MeshBasicMaterial({ envMap: cubeTexture });
```

A lot goes on...

- U,V computed from view direction
- Lookup into cube texture map

Works well with bump maps!



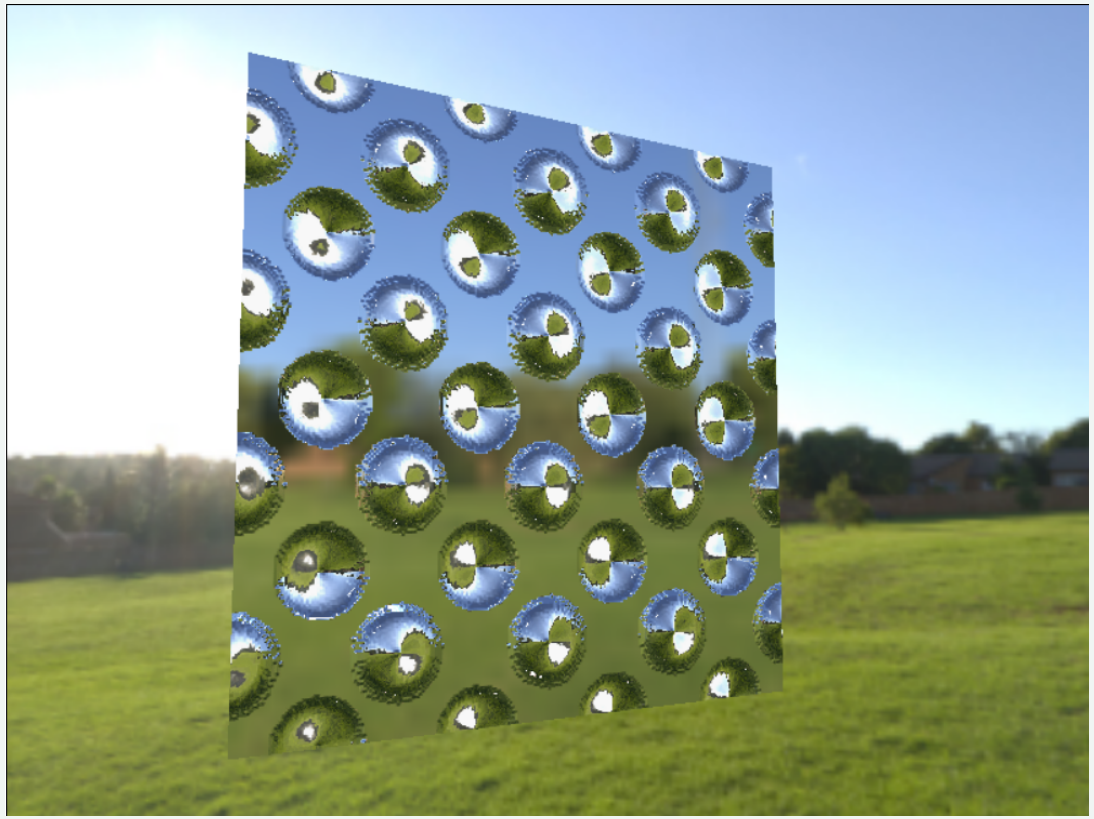


Requires a Shiny Material

Using THREE.js `MeshStandardMaterial`

- metalness 1.0
- Roughness 0.0

(try the demo)

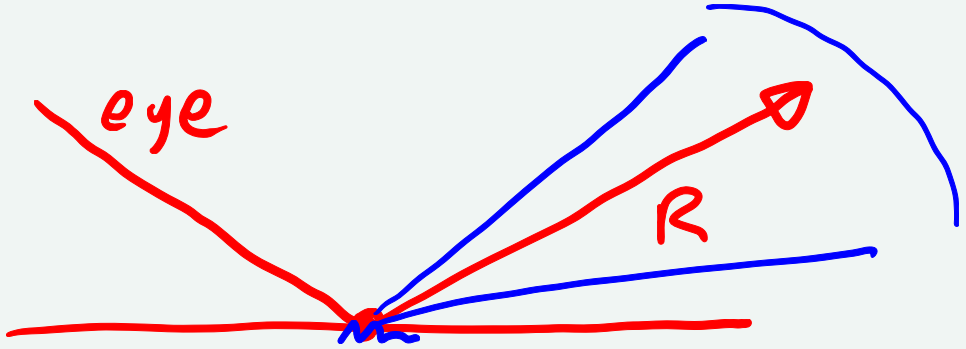


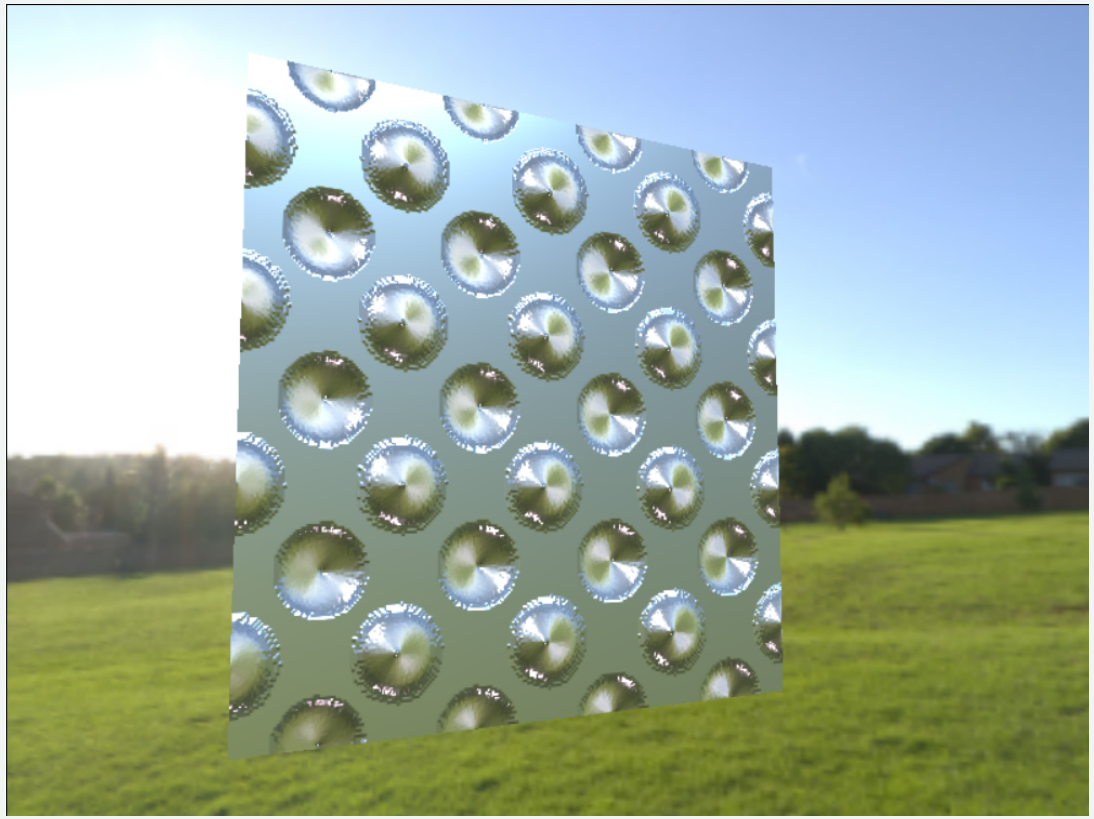
Less "reflective"

How much to mix in "regular lighting"

How much sharp are reflections

Rough materials start to look **diffuse**



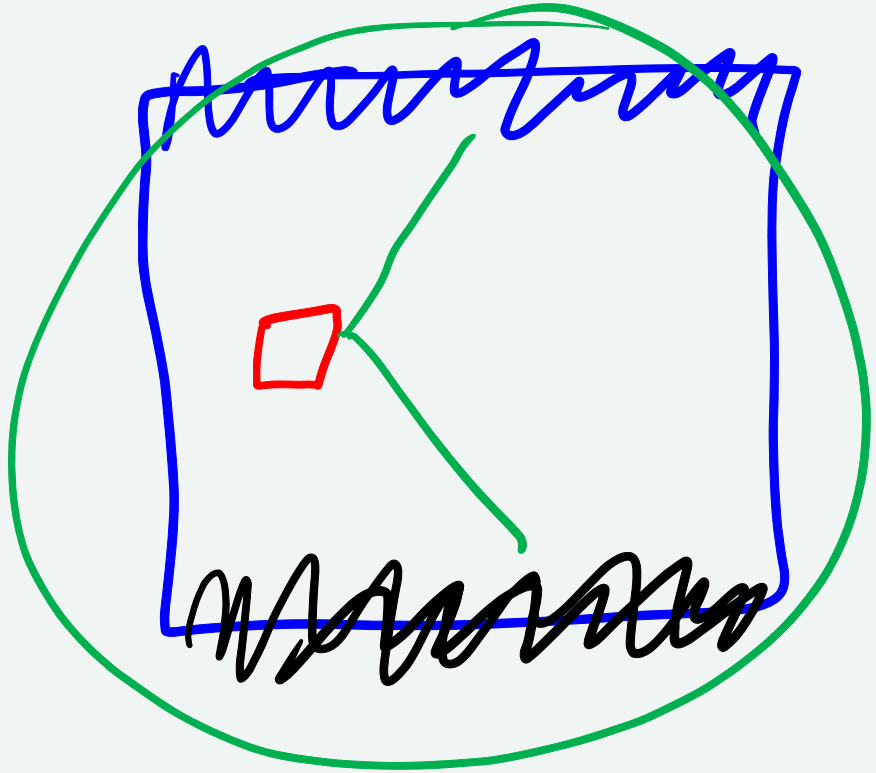


Environment Maps for Lighting

The environment map is light!

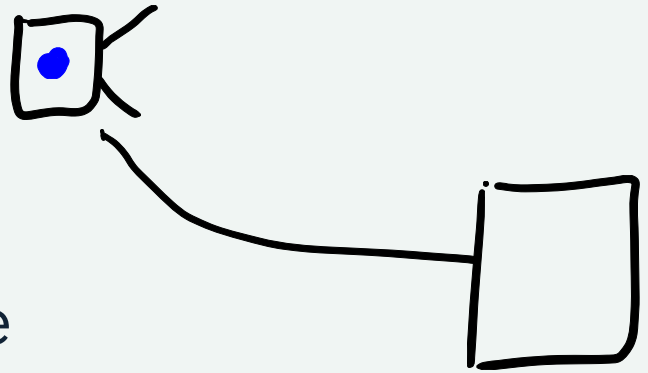
Use the real scene to create brightness!

Not limited to a small set of light sources!



But the environment map is static?

Often, pre-compute the environment map ahead of time (or use a photo)



But: we can draw it ourselves!

- draw the scene with the camera where the object will be
- use a special "cube camera" to take a picture in 6 directions
- take this picture before making the "real" picture
- use this picture as the texture
- **Dynamic Environment Map**

1st picture as texture when drawing

Multi-pass rendering (draw the scene multiple times)

CS559 Lecture 19-20: More Texture

Part 5: Shadow Maps

Shadow Maps

Note: This is online one way to do shadows

The details on how it works change

Idea: Can the Light See the Object?

We know how to make a picture of what the **camera** sees

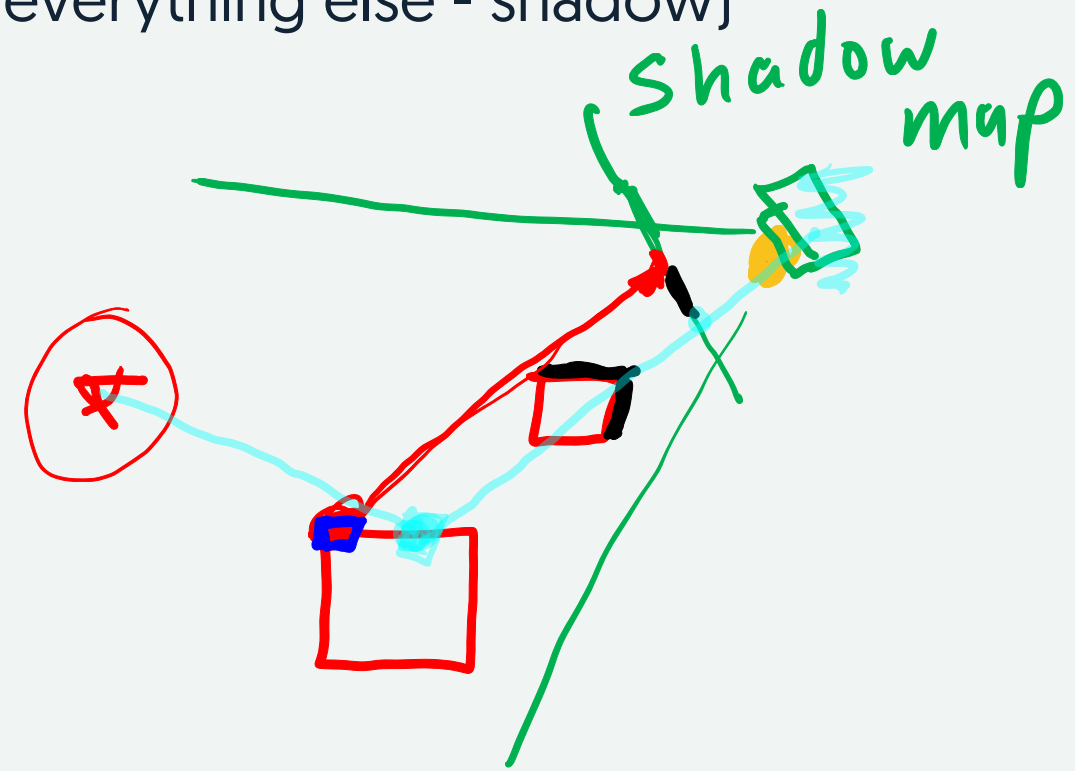
(warning - we didn't discuss how this works yet)

Use the same method to ask what the **light** sees

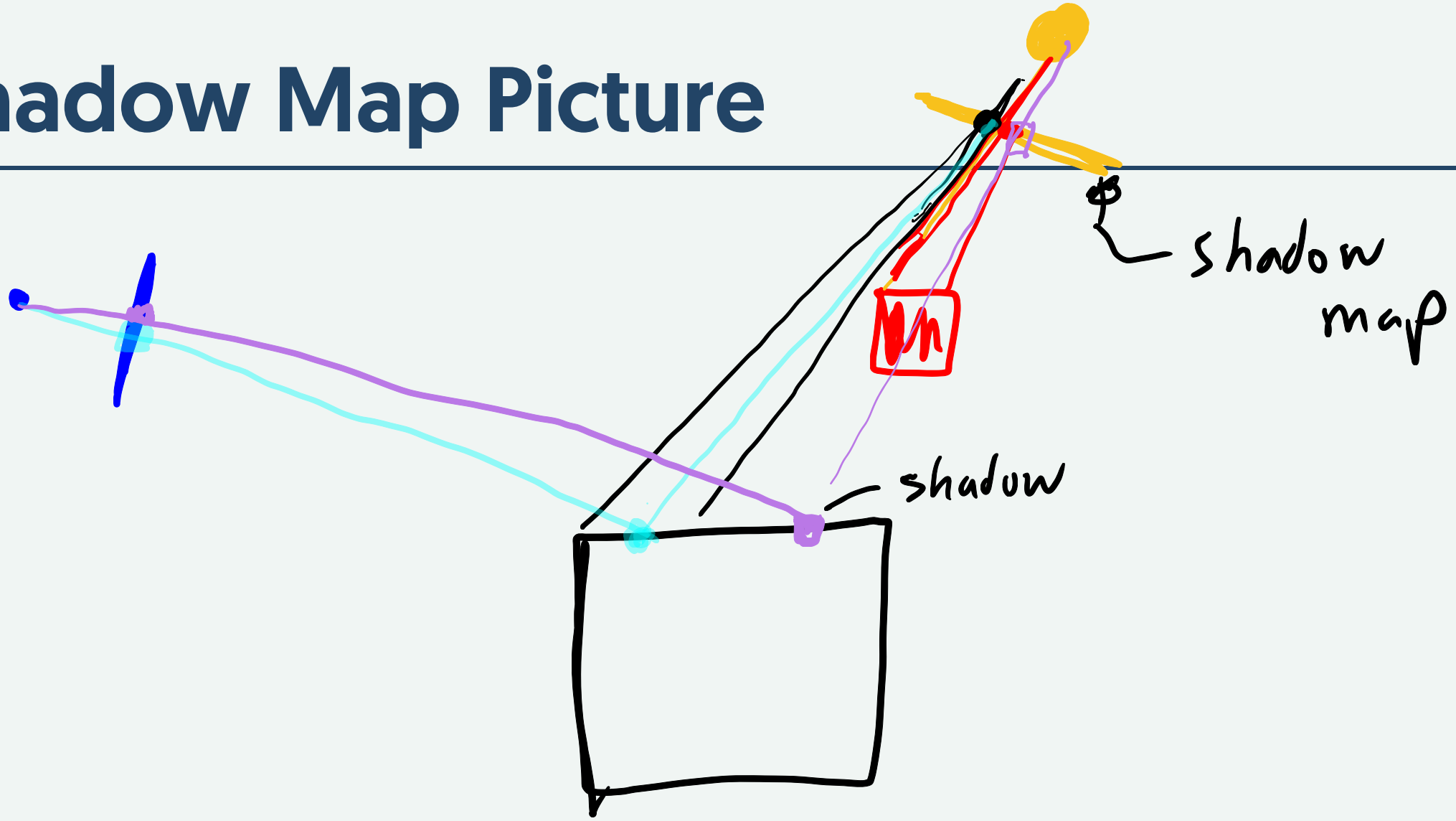
- put the "camera" where the light source is

Shadow mapping

1. Take a picture from the light position
    - camera at light source
    - what objects are visible to the light (everything else - shadow)
    - use this picture as the **shadow map**
2. Draw the "regular picture"
    - for each pixel on an object
    - see if pixel is visible in shadow map



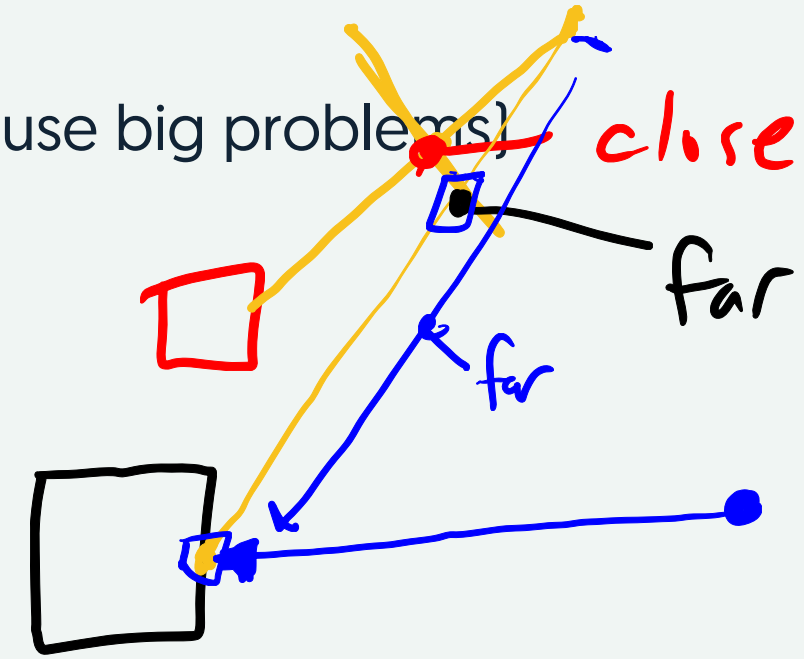
Shadow Map Picture



Shadow Map Test

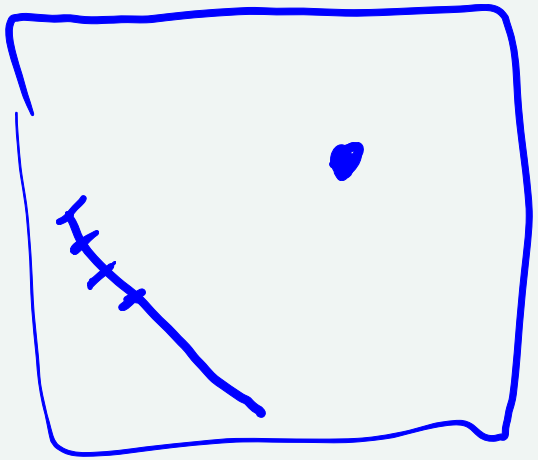
How do we know if the pixel is the same pixel in the shadow map?

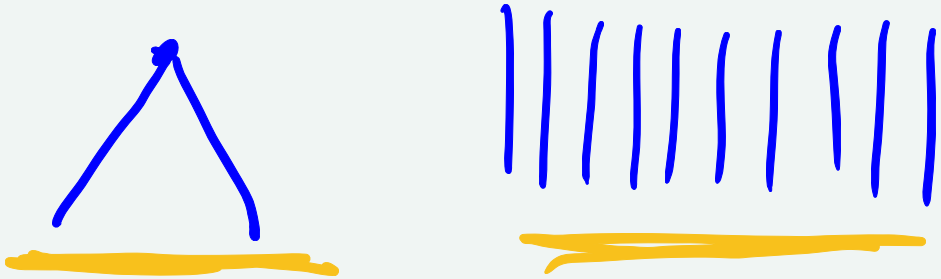
- check color (not a good idea)
- check depth (most common approach)
  - can be problematic (small errors cause big problems)



Shadow Map Resolution

- Size of Shadow Map Matters
  - spotlight (can be small)
  - directional light source (area)
  - point light? (needs to be a cube/sphere)





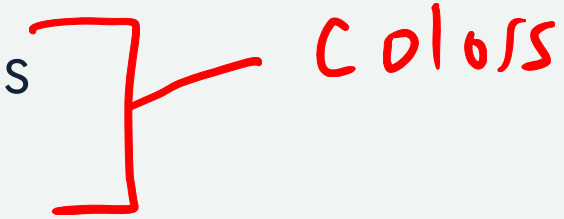
Shadow Maps in THREE

Of course it makes it easy!

- Tell the lights to cast shadows
  - they will make shadow maps
- Tell the objects to **cast** shadows
  - they will be rendered in all passes
- Tell the objects to **receive** shadows
  - their shaders will access the shadow maps
- Tell the renderer to do shadows
  - it will set up the multiple passes

Summary: Advanced Texture Hacks

- Normal and Bump Maps for surface details
- Layered Textures to mix effects
- Lightmaps / Ambient occlusion for pre-computed lighting
- Environment Maps for Reflections (and lighting)
- Shadow Maps for Shadows



Hacks?

Why Hacks? Can't we do something more principled?

We use hacks because they are implemented efficiently in the **graphics hardware**.

(guess what we learn about next)