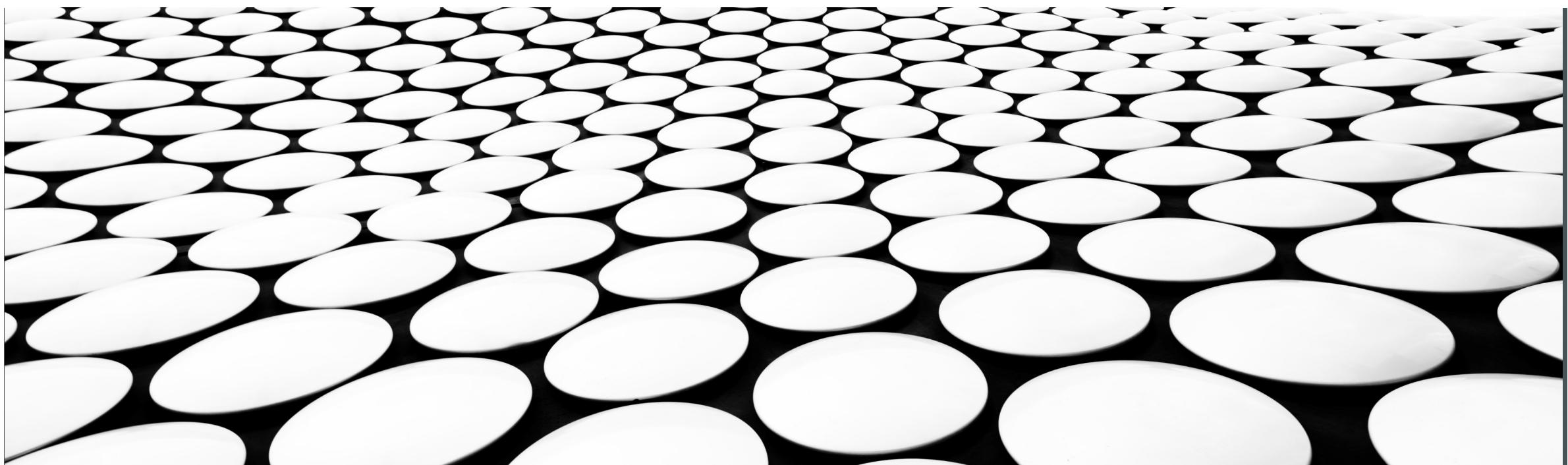
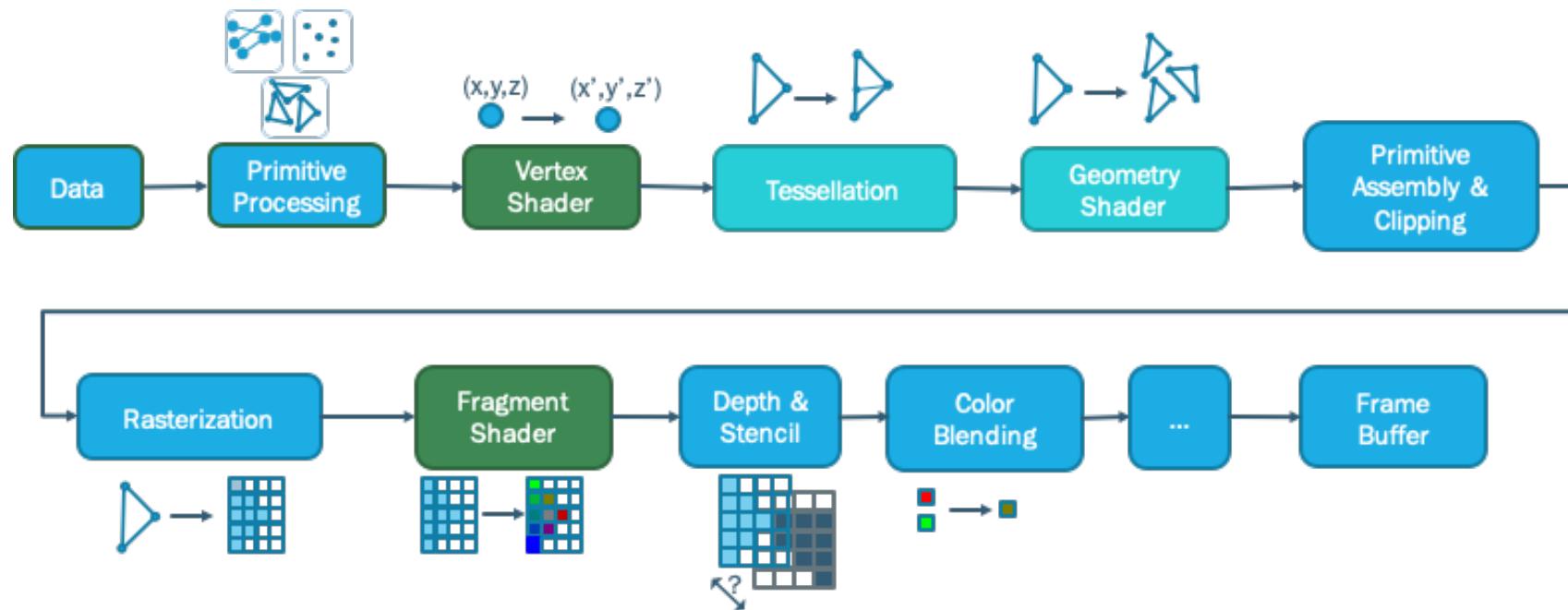

INFO-H-502 – 3D GRAPHICS FOR VR

EXERCISES 03

ELINE SOETENS – DANIELE BONATTO

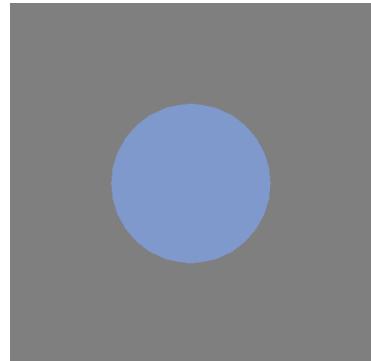


LAST TIME

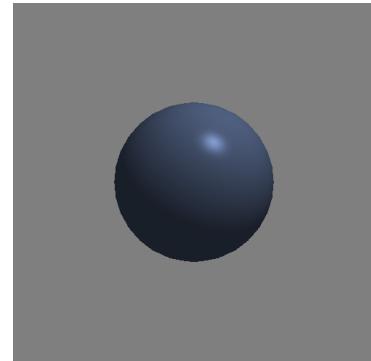


The whole pipeline

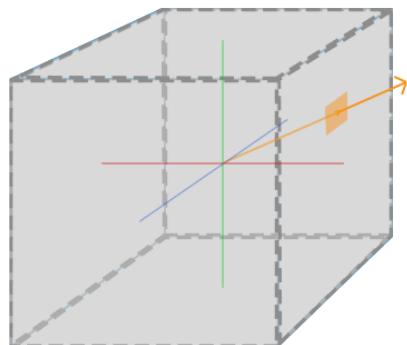
THIS TIME



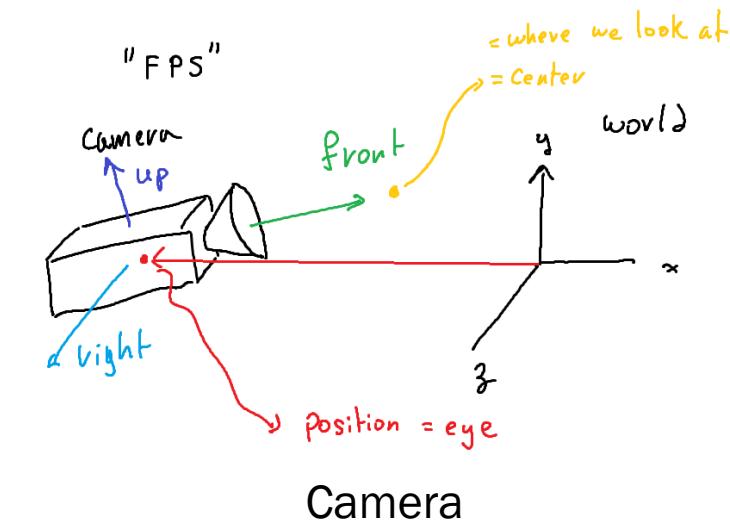
No lights



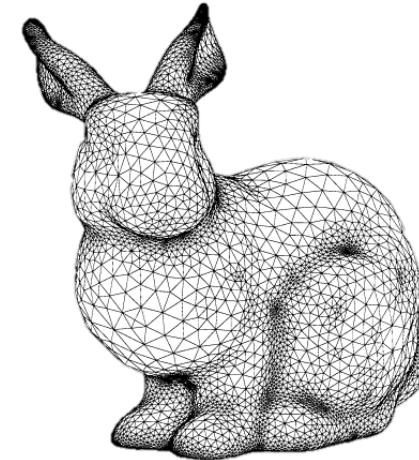
Lights



Cubemap



Camera



Loading a model

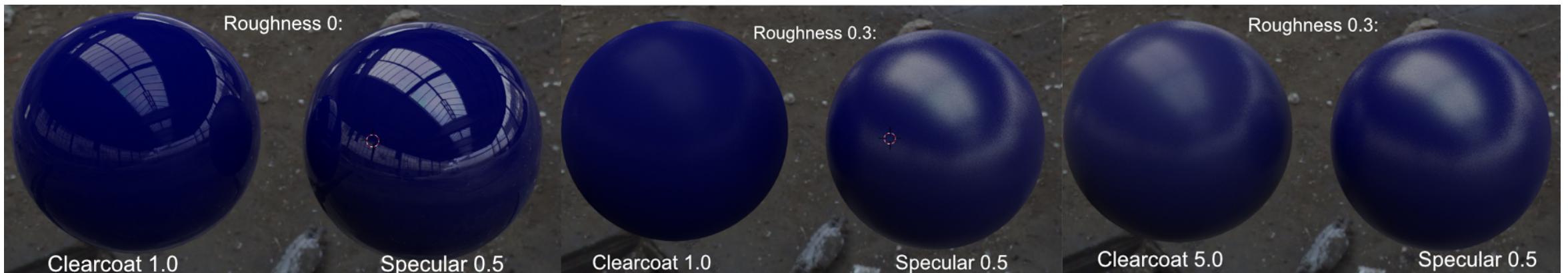
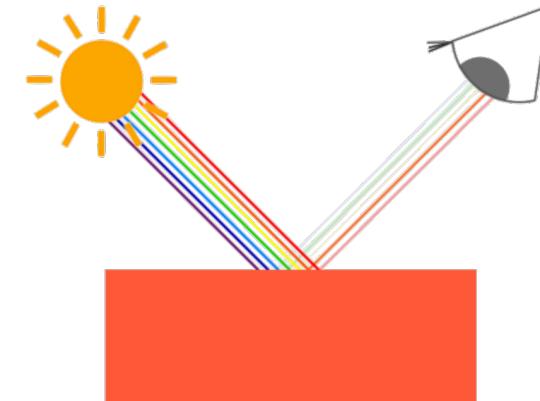
```
MeshVersionFormatted 1
Dimension 3
Vertices
5433
-0.0260146 0.112578 0.0363871 0
-0.0321783 0.174119 -0.00263321 0
-0.080718 0.152855 0.0302446 0
-0.0231294 0.112186 0.0386436 0
0.0164928 0.122293 0.0314234 0
-0.0248365 0.156574 -0.00377469 0
0.0452628 0.0863563 0.0200367 0
-0.071339 0.155715 0.00712639 0
-0.0100758 0.125949 0.0297379 0
-0.0720133 0.154518 0.0303984 0
```

THEORY VS EXERCISES

- In the theoretical lectures: What is light ? How do we get the light equation ?
- Here: How do we implement tge light equation ?
 - Still have 2/3 theory slides here to give some essential notions

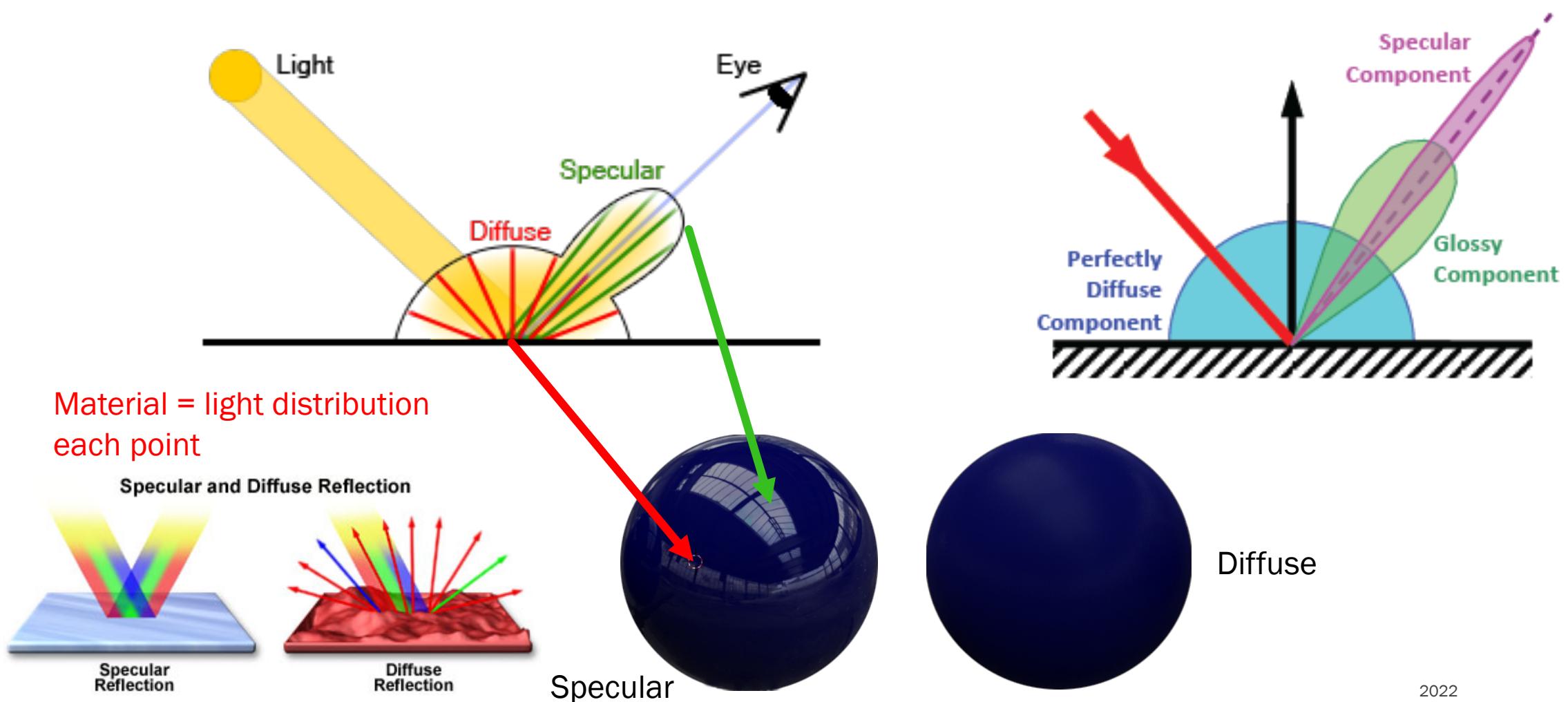
LIGHT – BIDIRECTIONAL REFLECTANCE DISTRIBUTION FUNCTION (BRDF)

- Real life : each object interact in different way with the light
- 3D graphics : materials are modelled



Mix of diffuse and specular lights

LIGHT – BIDIRECTIONAL REFLECTANCE DISTRIBUTION FUNCTION (BRDF)



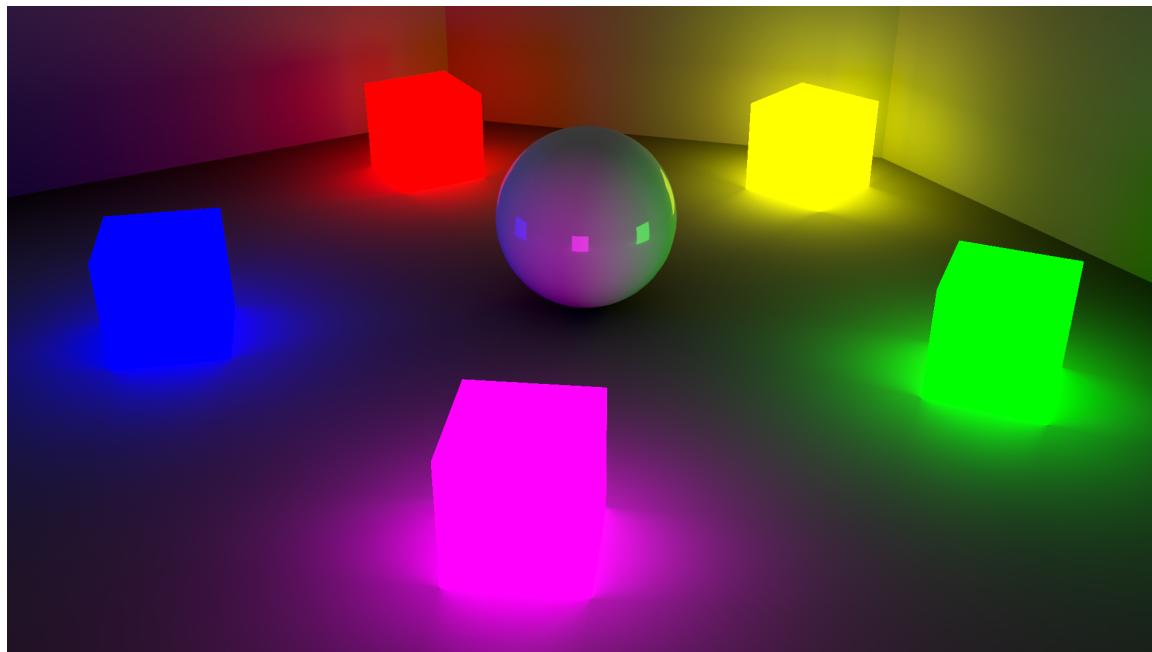
LIGHT - RENDERING EQUATION FOR OPENGL (~BLINN-PHONG MODEL)

- Light equation compute the amount of light outputted $L_o(\vec{x})$ at position/vertex \vec{x} by an object
- Depend on :
 - Light emitted by the object
 - External light sources **S** and their position relative to the object
 - Position of the camera (specular light)

RENDERING EQUATION FOR OPENGL (~BLINN-PHONG MODEL)

- $L_o(\vec{x}) = A + E + \sum_N (diff + spec) \cdot L_i \cdot att_i \cdot \cos \theta_i$

$$(\omega_i \cdot \vec{n}) \equiv$$



$L_o(\vec{x})$ = Output light color at vertex x

A = Ambient light to compensate (lot of simplification to get here)

E = Emitted light from the object

Diff = Diffuse factor of the vertex (to be computed)

Spec = Specular factor of the vertex (to be computed)

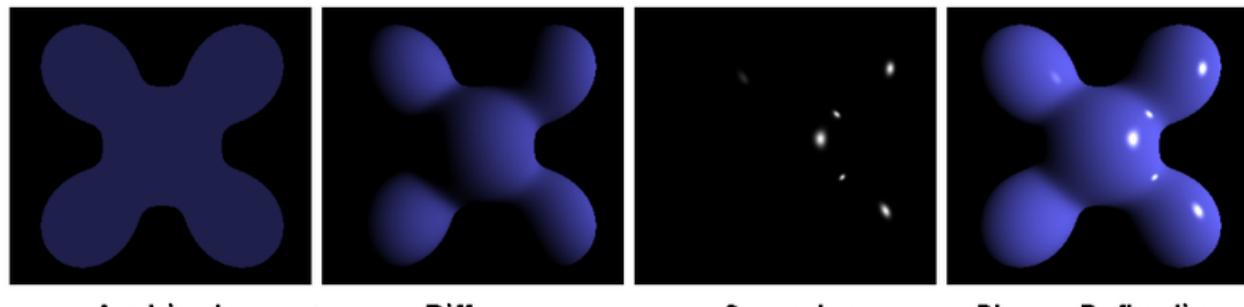
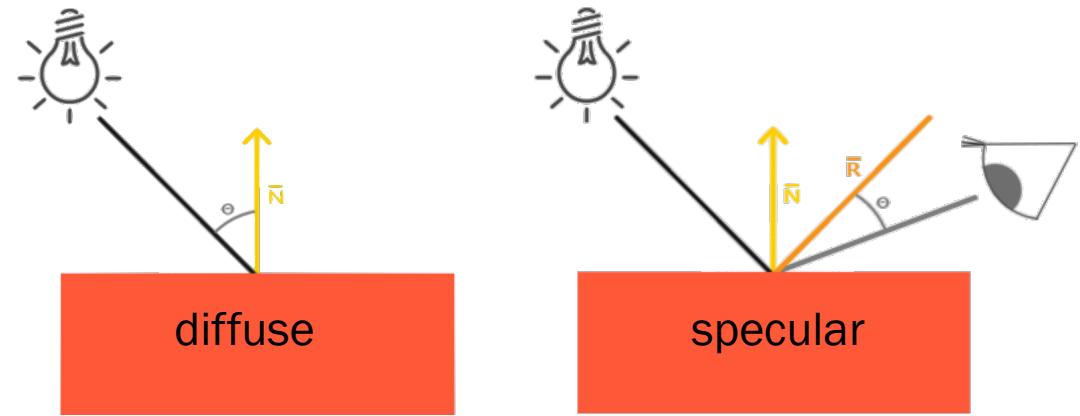
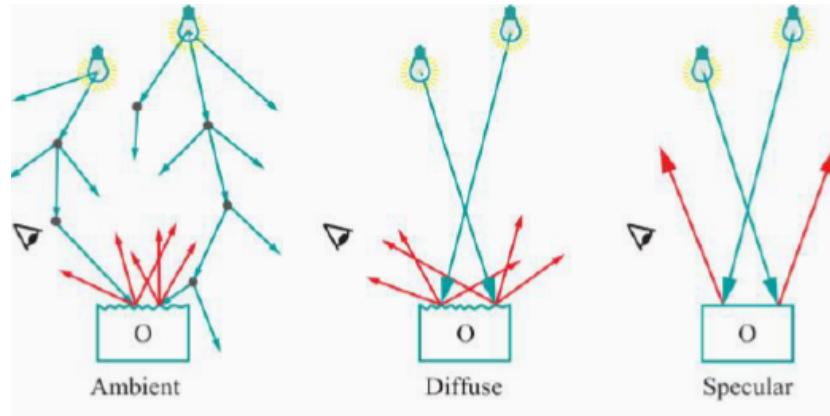
L_i = Constant color for each light i (N lights in the scene)

att_i = Attenuation factor for the light i

θ_i = Angle between the light position and the vertex normal

See theory lessons to know how we get this

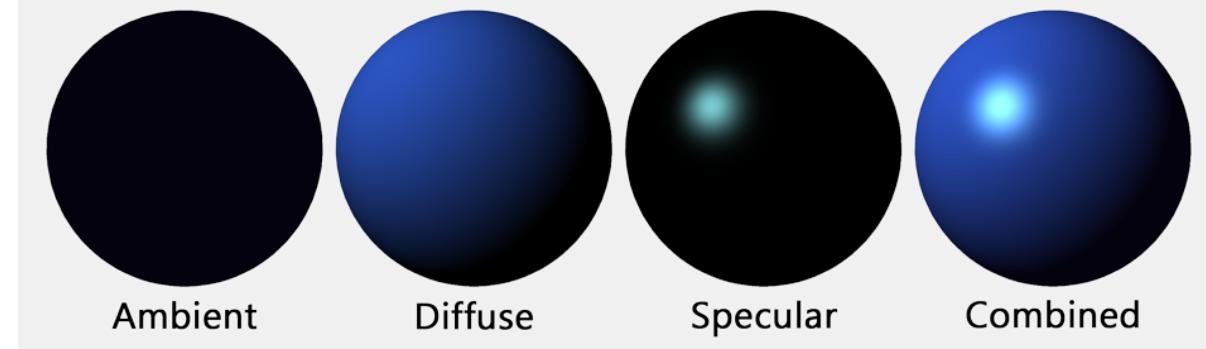
RENDERING EQUATION - COMPONENTS

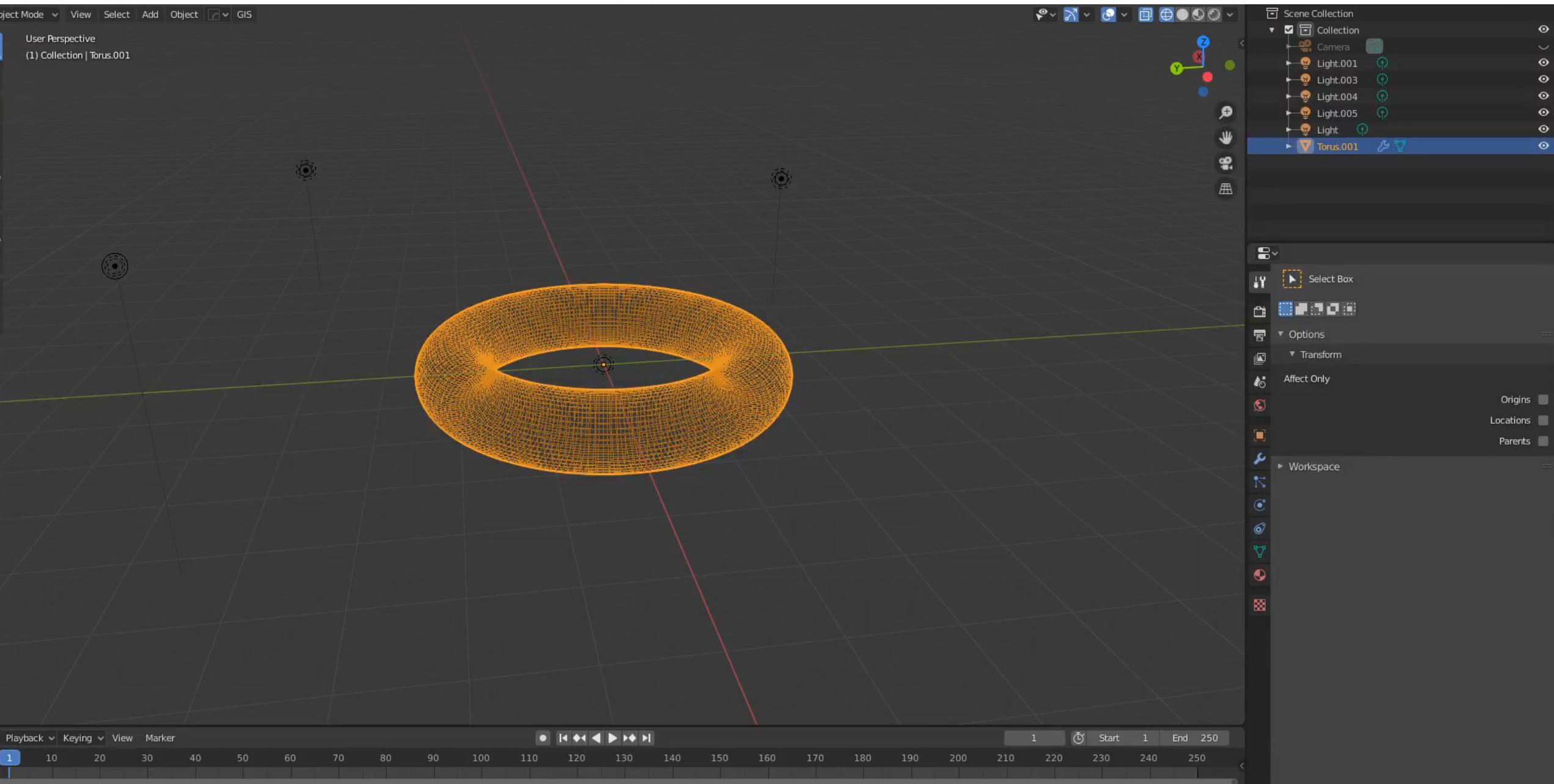


Constant

Angle
light-normal

Reflection
light-eye





Lights + Torus – Material: BRDF

2022

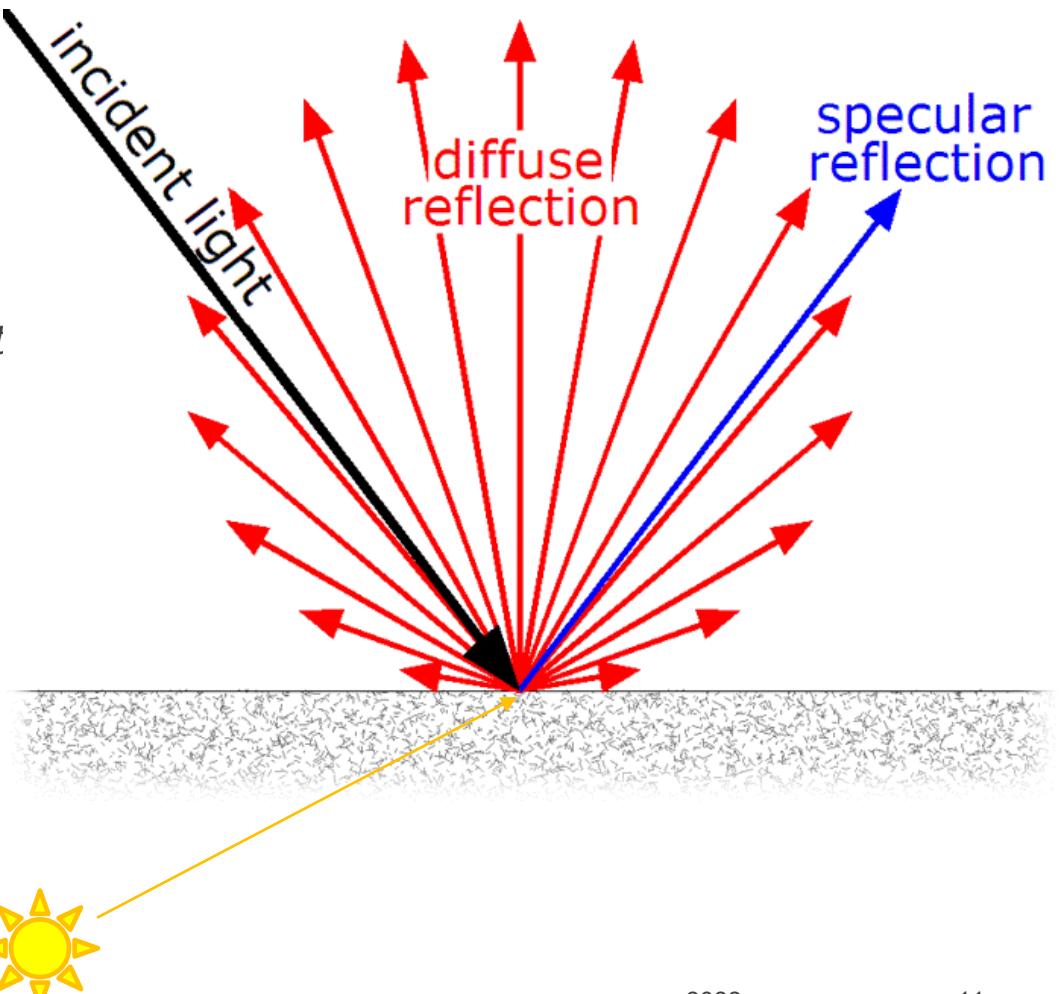
10

DIFFUSE LIGHT

\vec{n} = normal of the surface
 ω_i = direction of the light of the light

Angular dependence

- $L_o(\vec{x}) = A + E + \sum_N (diff + spec) \cdot L_i \cdot att_i \cdot (\omega_i \cdot \vec{n})$
- $L_o(\vec{x}) = A + E + \sum_N (diff \cdot (\omega_i \cdot \vec{n}) + spec \cdot (\omega_i \cdot \vec{n})) \cdot L_i \cdot att$
- Depends on the light position and the normal at a point
 - $diff = 1$ (Empirically!)
 - $(\omega_i \cdot \vec{n}) = \cos \theta = \text{angle between the light and the normal}$
- Need to discard negative values as they do not make sense
 - $\max(\omega_i \cdot \vec{n}, 0)$

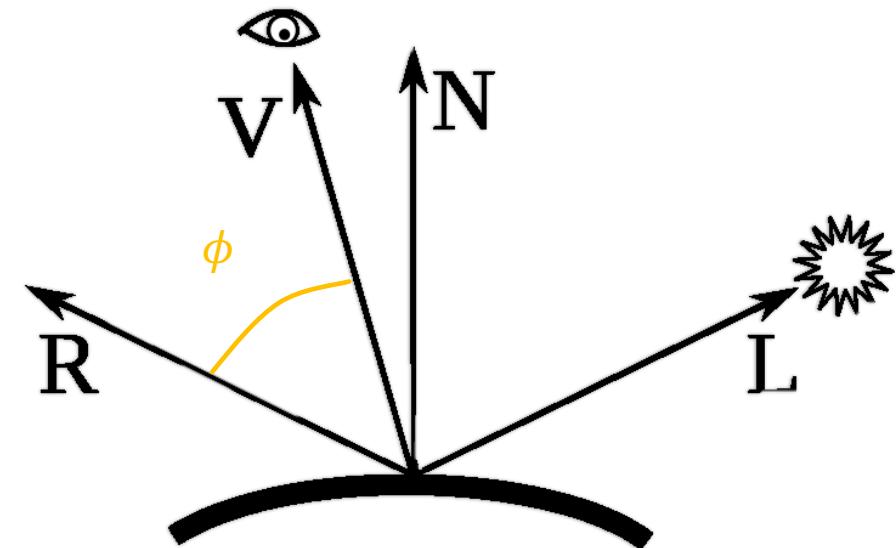


SPECULAR LIGHT (BLINN-PHONG MODEL)

- $L_o(\vec{x}) = A + E + \sum_N ((\omega_i \cdot \vec{n}) + \text{spec.}(\omega_i \cdot \vec{n})).L_i.att_i$
- $L_o(\vec{x}) = A + E + \sum_N ((\omega_i \cdot \vec{n}) + \text{spec.}(\omega_i \cdot \vec{n})).L_i.att_i$
- Depends on the light vector (L) and the View vector (V) (the camera)
 - Compute the reflection vector (R) from L
 - $R = 2(N \cdot L)N - L$
 - $\cos \phi = R \cdot V$
- $\text{spec} = \cos^r \phi$
- **REMARK:** this is a WRONG MODEL! The $(\omega_i \cdot \vec{n})$ factor disappeared! (But nice results), spec should be divided by it!

! dot products !

! Normal vectors !

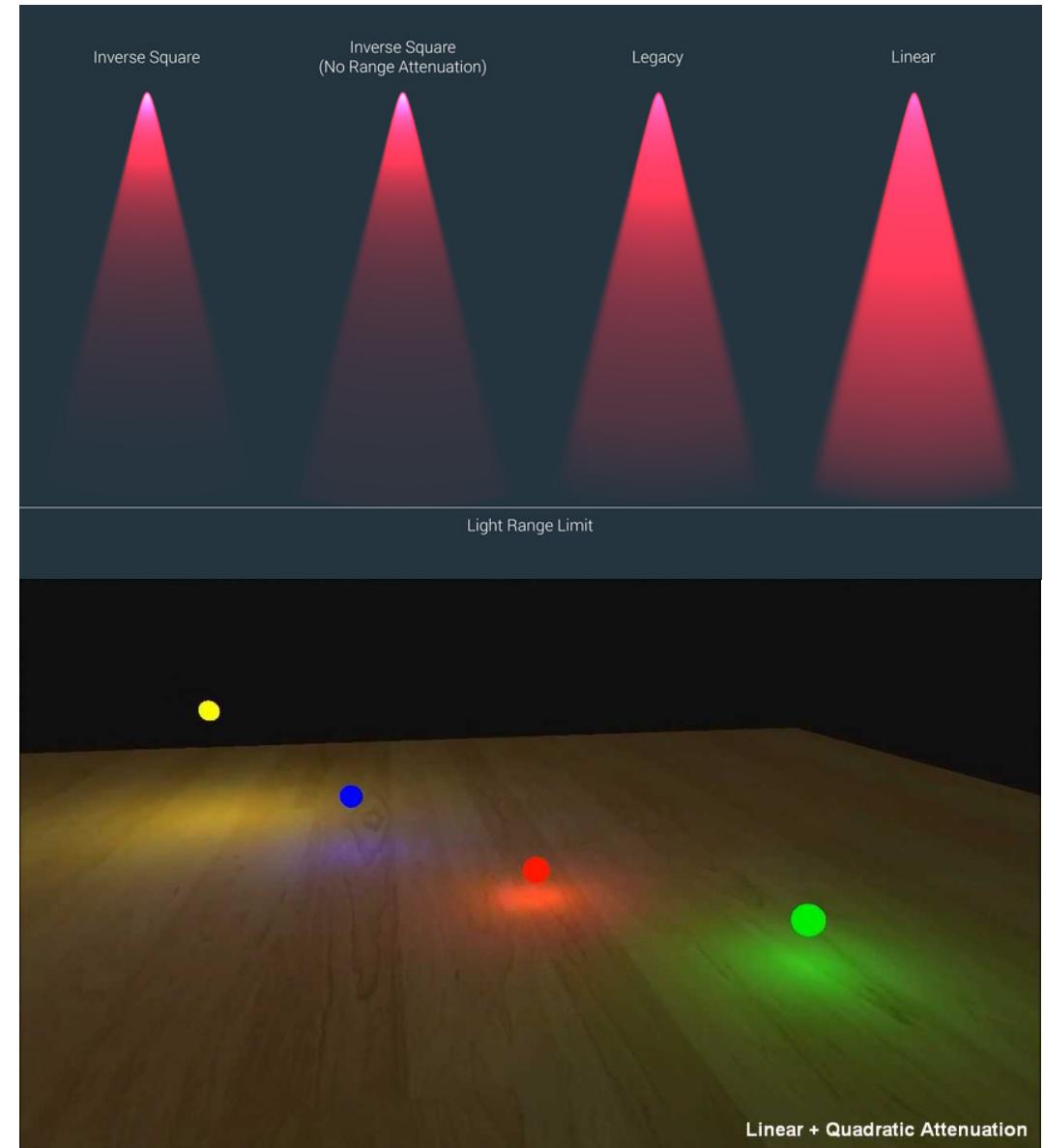


More on the wrong model here:

<https://computergraphics.stackexchange.com/questions/7590/phong-and-the-rendering-equation-whats-with-the-cosine>

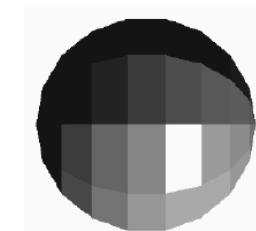
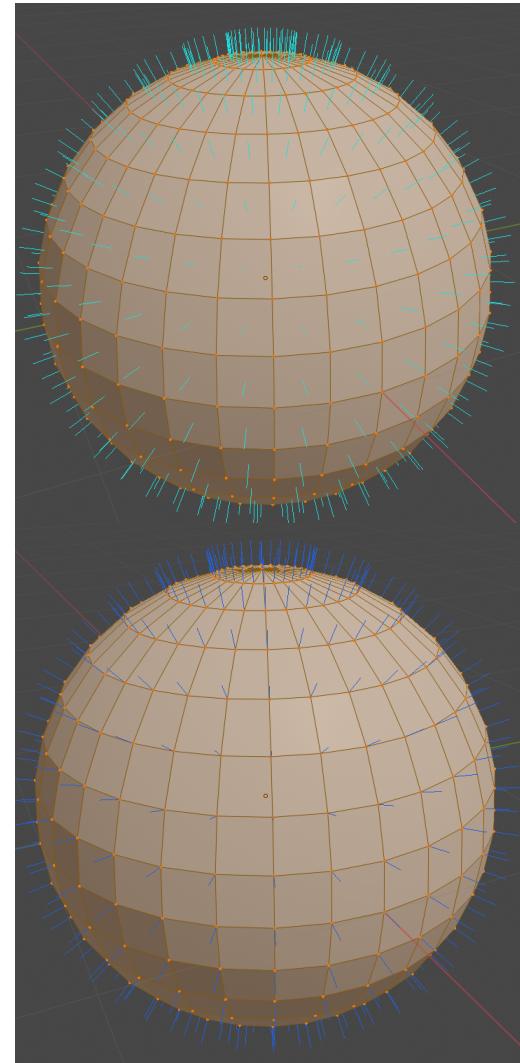
ATTENUATION WITH THE DISTANCE

- $L_o(\vec{x}) = A + E + \sum_N ((\omega_i \cdot \vec{n}) + spec).L_i.att_i$
- $att_i = \frac{1}{k_c^i + k_l^i d^i + k_q^i (d^i)^2}$
 - k_c = Constant attenuation
 - k_l = Linear attenuation
 - k_q = Quadratic attenuation
 - d = Distance between the light source and the fragment

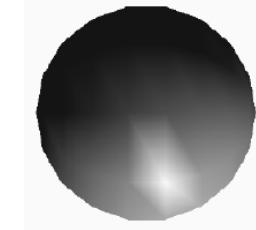


WHERE IS THE RENDERING EQUATION COMPUTED?

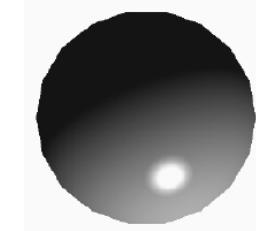
- Flat Shading
 - Every triangle in the scene has one normal
- Gouraud shading
 - Each vertex of each triangle has a normal
 - The **light is computed per vertex** (vertex shader)
 - The **light is interpolated** in the fragment shader
- Phong shading
 - Each vertex of each triangle has a normal
 - The **normals are interpolated** between the vertex and fragment shaders
 - The **light is computed for every fragment** using the normal



Flat

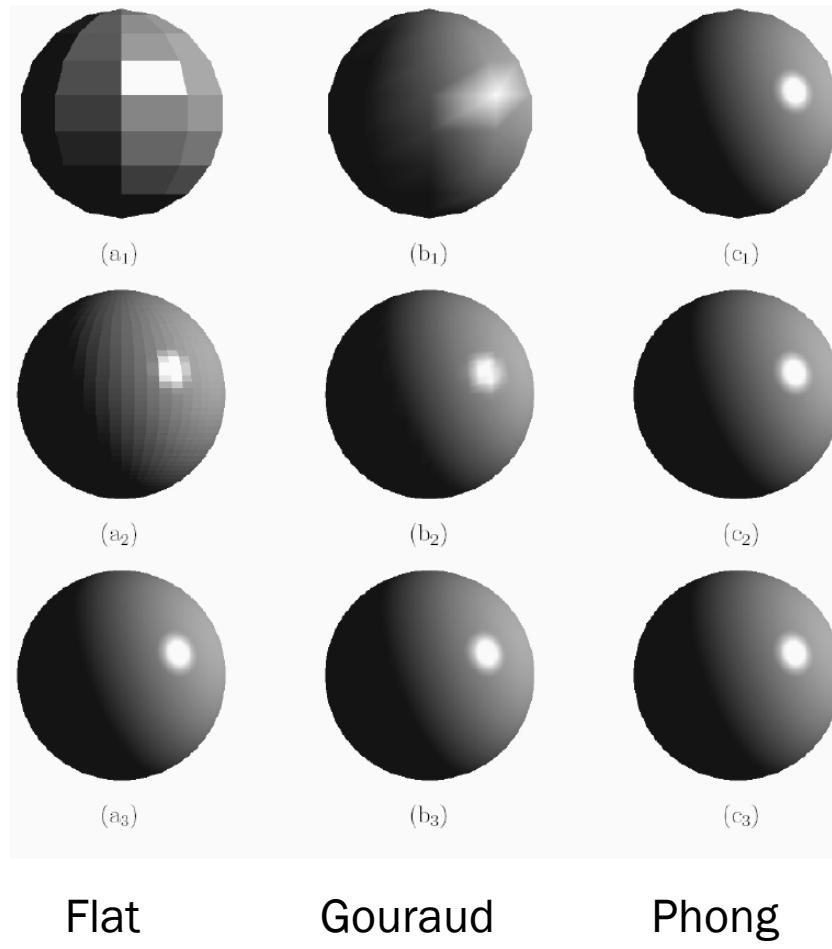


Gouraud



Phong

SHADING AND TRIANGLE COUNT

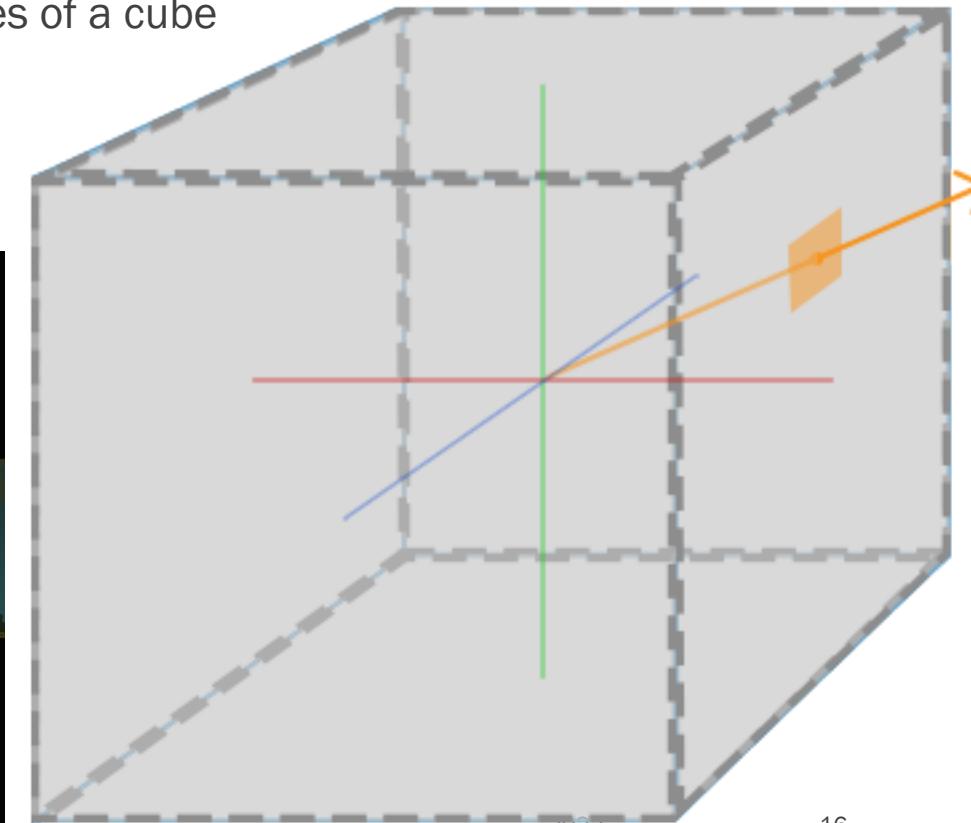
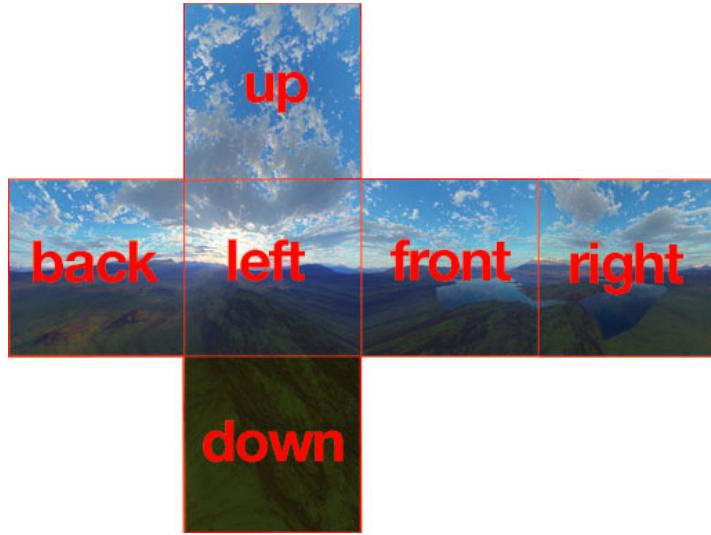


Less triangles

More triangles

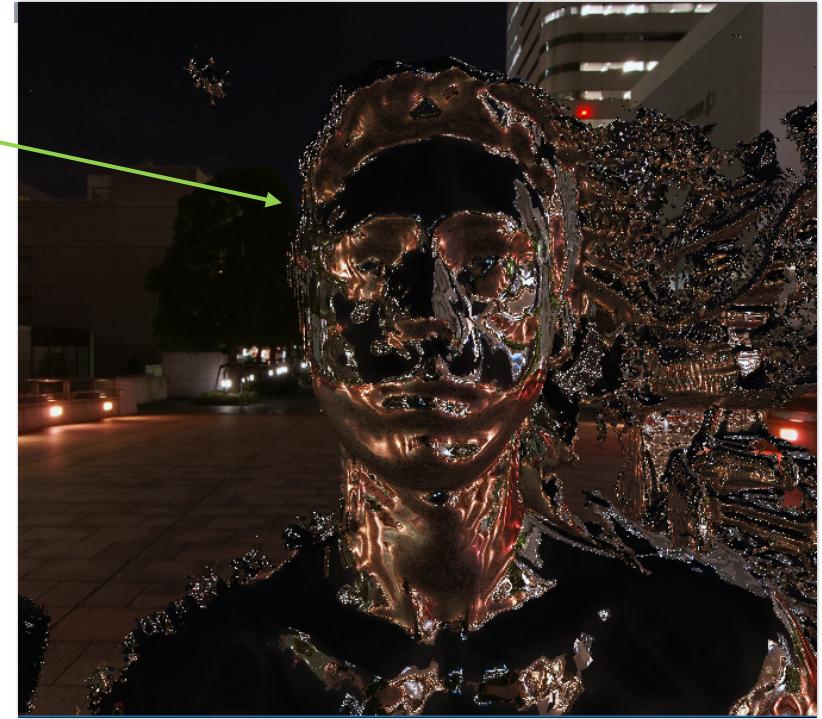
CUBEMAPS

- Cubemap: A texture containing 6 individual 2D textures forming the sides of a cube
- Shaders let you « sample » the cube map using a 3D vector



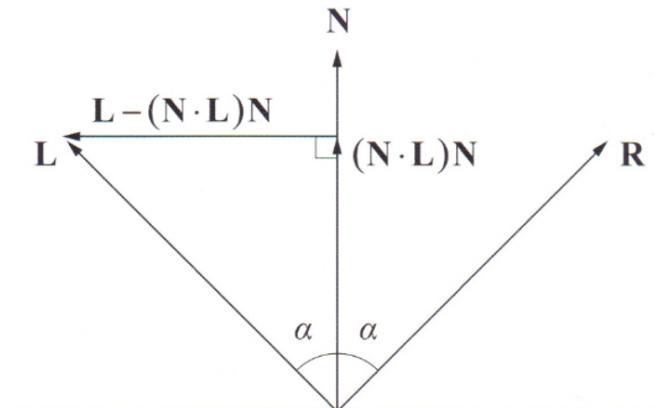
REFLECTION

This is what you should obtain



- Possible because we **sample** our cubemap
- $R = 2(L \cdot N)N - L$
- R is the vector used to sample the cubemap

Use glsl function *reflect*



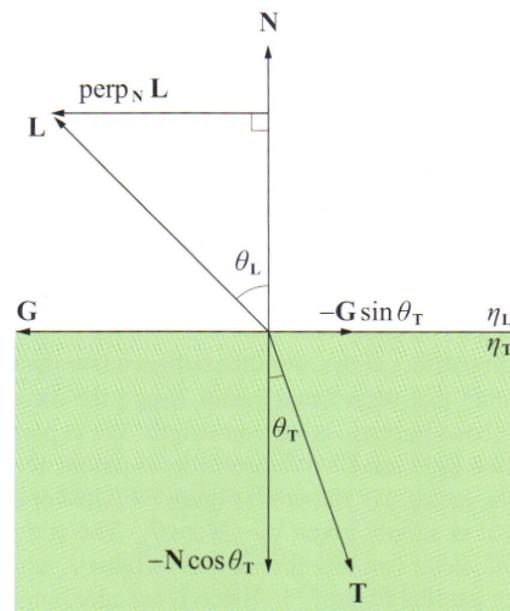
REFRACTION

This is what you should obtain



- Refraction also possible because we sample a cubemap
- $T = \left(\frac{\eta_L}{\eta_T} N \cdot L - \sqrt{1 - \frac{\eta_L^2}{\eta_T^2} [1 - (N \cdot L)^2]} \right) N - \frac{\eta_L}{\eta_T} L$
- Vector used to sample the cubemap

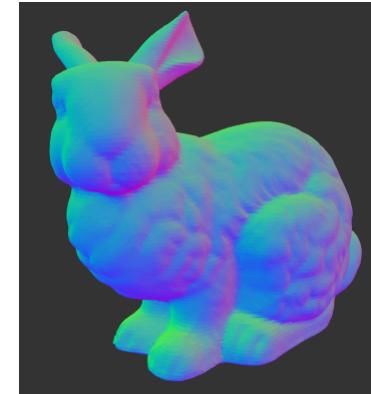
Use glsl function *refract*



| Some η_T : | |
|-----------------|-------|
| Air | 1.00 |
| Water | 1.33 |
| Ice | 1.309 |
| Glass | 1.52 |
| Diamond | 2.42 |

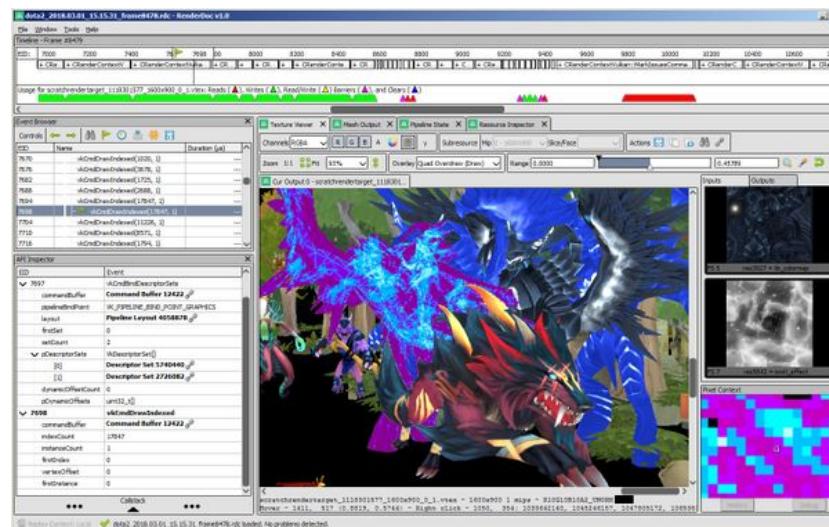
DEBUGGING

- To debug your codes: Output any **vector** you want to check as the **color**!
 - Transformations required between vector spaces, eg: $[-1,1]^3 \mapsto [0,1]^3$
 - If object is black -> the vector contains only zeros

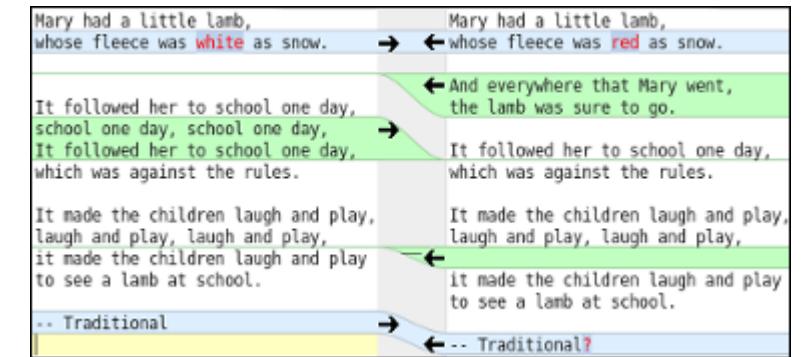


Normals as color

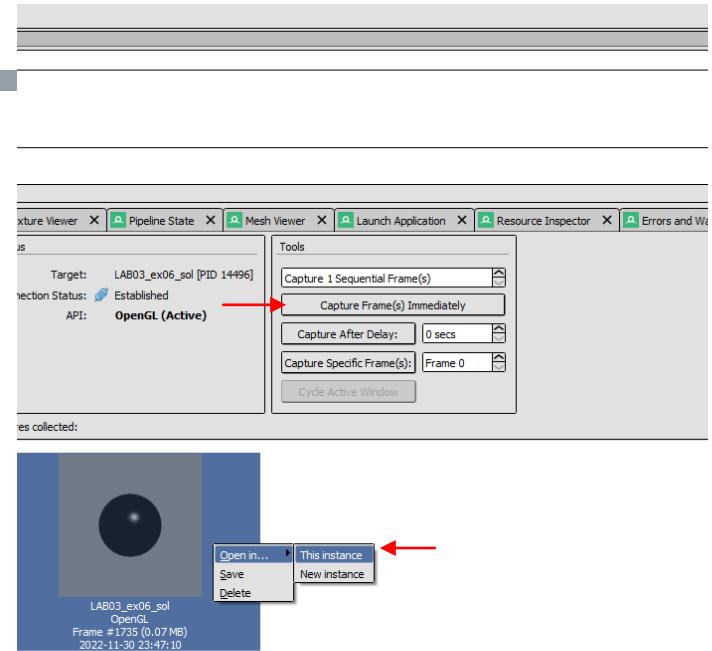
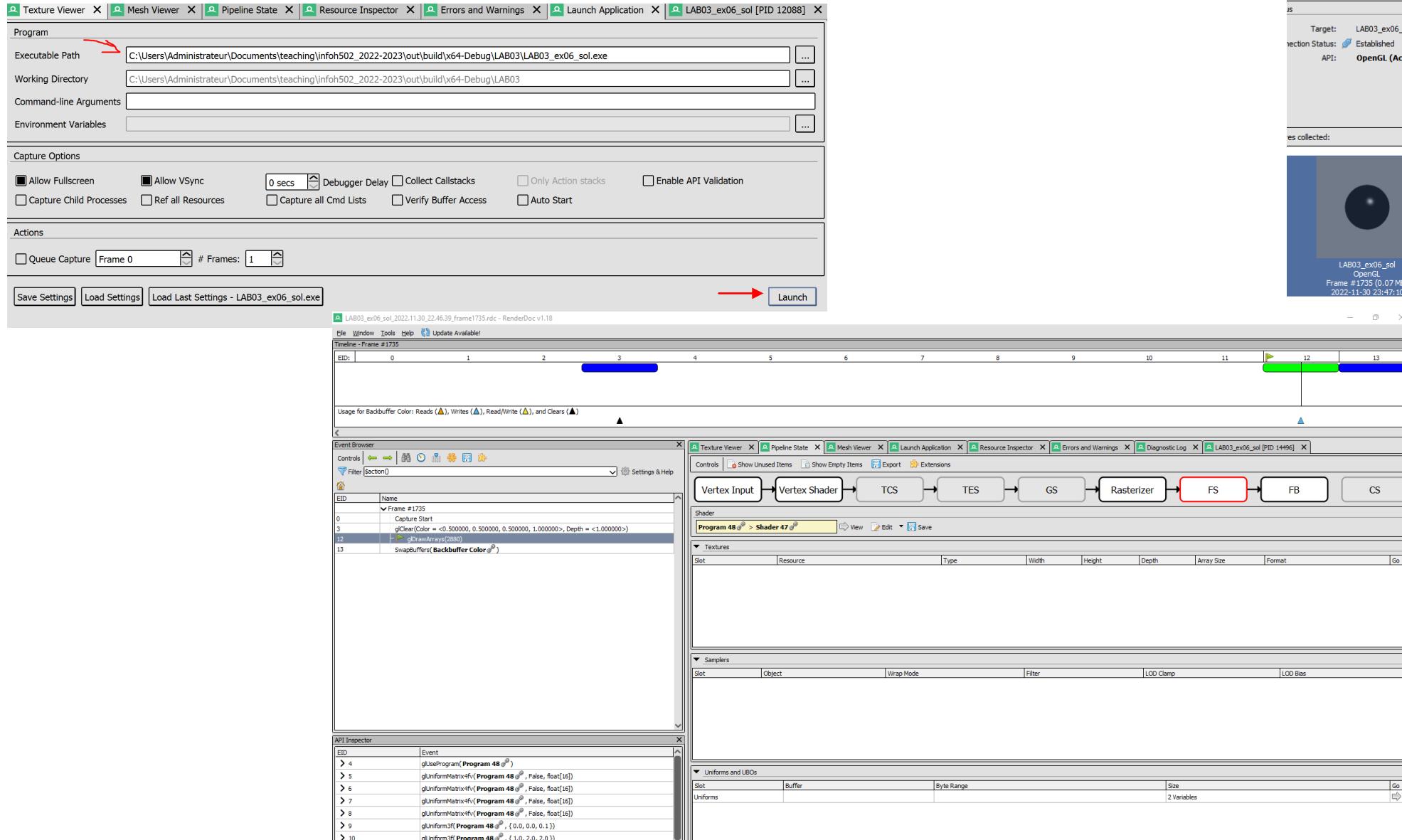
- Use MELD to compare the solutions/your code/exercise code/web code
 - <https://meldmerge.org/>



- (C++) Use RenderDoc !
 - Nice youtube tutorials



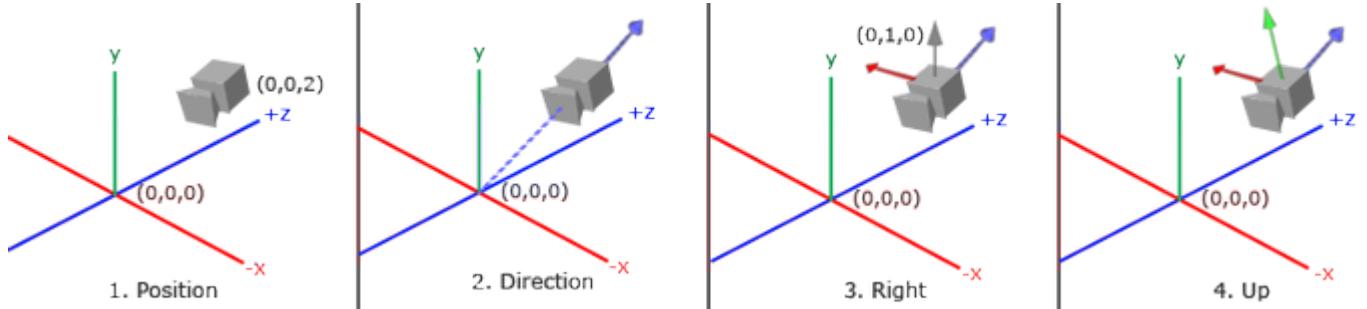
RENDERDOC (LINUX AND WINDOWS ONLY)



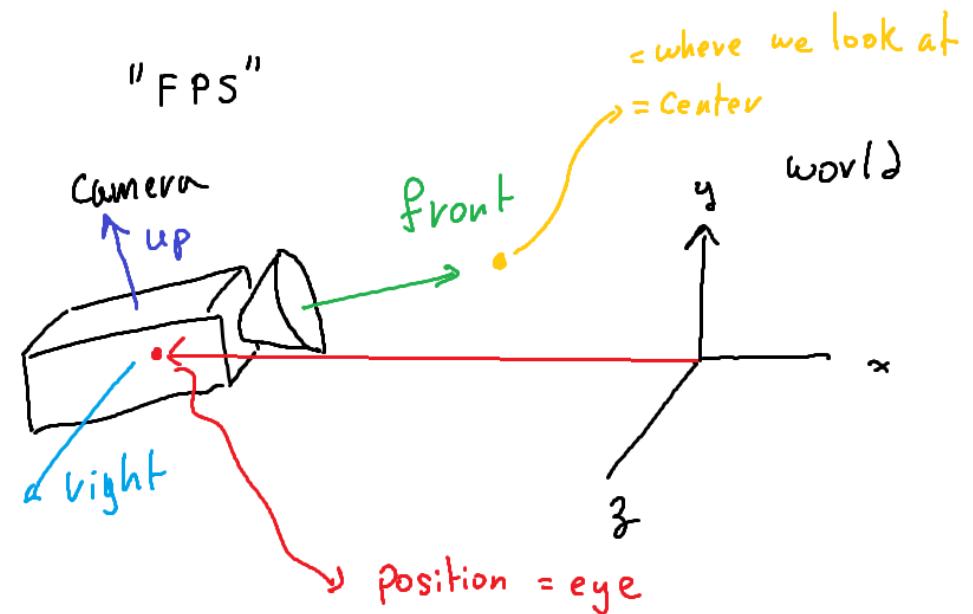
DOWNLOADING LAB 3 FROM GITLAB (WITHOUT CLONING)

- Download the LAB03 folder/ git pull the repository
- Put it at the same place that the LAB01 & LAB02 folders
- Change the CMakeList file directly:
 - In the top-most CMakeList.txt -> Put COMPILE_LAB03 to ON
 - Regenerate Cmake cache
- If the new exercise still don't compile : check the cmake cache file :
 - Microsoft Visual Studio : right click on CMakeList.txt (on top) -> edit cmake setting -> COMPILE_LAB03 to ON
 - VSCode : Ctrl/Cmd + shift + p -> edit cmake cache (ui) -> COMPILE_LAB03 to ON

EXERCISE 01



- Read exercise 01 code to understand how the camera works
 - camera.h
 - What is the difference between the View/Projection matrices AND the provided camera ?



The provided code follows those conventions

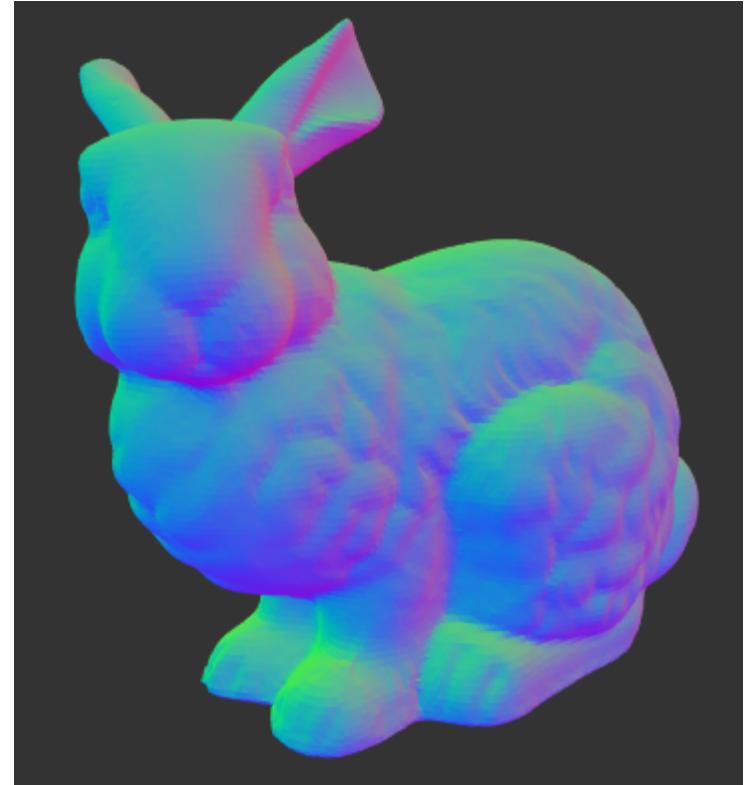
EXERCISE 02

- What does a Wavefront .obj file contain? Open plane.obj or cube.obj and take a look at what's inside.
- Explain (at least to yourself)
 - v = ?
 - vt = ?
 - vn = ?
 - f = ?

```
# Blender v2.81 (sub 16) OBJ File: ''
# www.blender.org
o Cube
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
vt 0.625000 0.500000
vt 0.875000 0.500000
vt 0.875000 0.750000
vt 0.625000 0.750000
vt 0.375000 0.750000
vt 0.625000 1.000000
vt 0.375000 1.000000
vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.250000
vt 0.125000 0.500000
vt 0.375000 0.500000
vt 0.125000 0.750000
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
s off
f 1/1/1 5/2/1 7/3/1 3/4/1
f 4/5/2 3/4/2 7/6/2 8/7/2
f 8/8/3 7/9/3 5/10/3 6/11/3
f 6/12/4 2/13/4 4/5/4 8/14/4
f 2/13/5 1/1/5 3/4/5 4/5/5
f 6/11/6 5/10/6 1/1/6 2/13/6
```

EXERCISE 02

- Make an object reader and load an object you create in Blender
 - You will need to read the content of the file
 - Then stock the relevant information in a way you can use with OpenGL, i.e.
 - Stock vertex position, texture coordinate, normal, ... in a buffer
 - Make sure you can send (and use) this information to the shader
 - Go to [appendix 9](#) to see how to export your blender project as a .obj file with the associated texture
- Display an object and use the normal value to color it

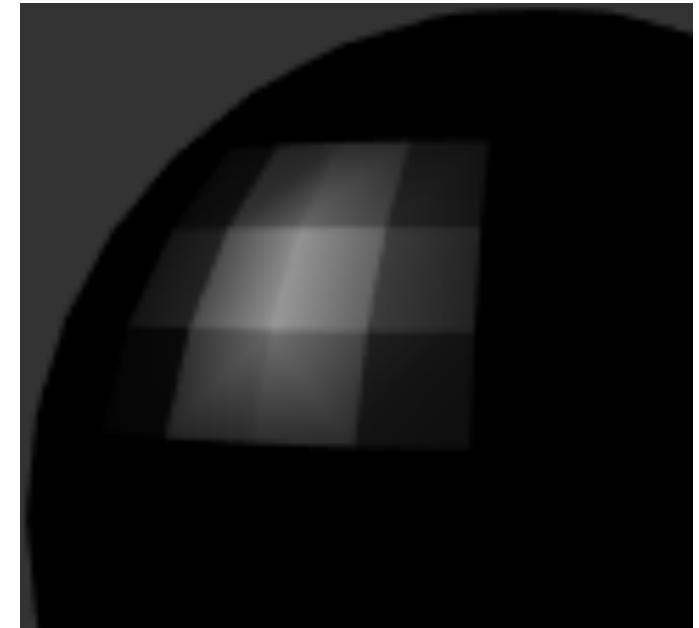


INTERLUDE – A NOTE ON OBJECT FILE

- A lot of formats exist
 - .obj : fairly easy to read, only contains the vertices
 - .mtl : materials file used with .obj, describes light reflecting properties of materials → pretty old but still works well
 - Collada (.dae) : more comprehensive, contains vertices, material properties, and even shaders ! Useful for more advanced objects/effects (animation ...)
 - glTF/gltb: A standard for 3D scenes and models, supported by many software (if you want to read it by hand :
<https://www.willusher.io/graphics/2023/05/16/0-to-gltf-first-mesh> or there is also a header-only library for it
<https://github.com/syoyo/tinygltf>)
- Complex models available online can use other formats
- Existing libraries to load different file formats (assimp, ...), you can use them BUT :
 - You should understand the format you use i.e. what are the data in the file (vertices, texture coordinates, material properties ?) and how are they used in the OpenGL pipeline

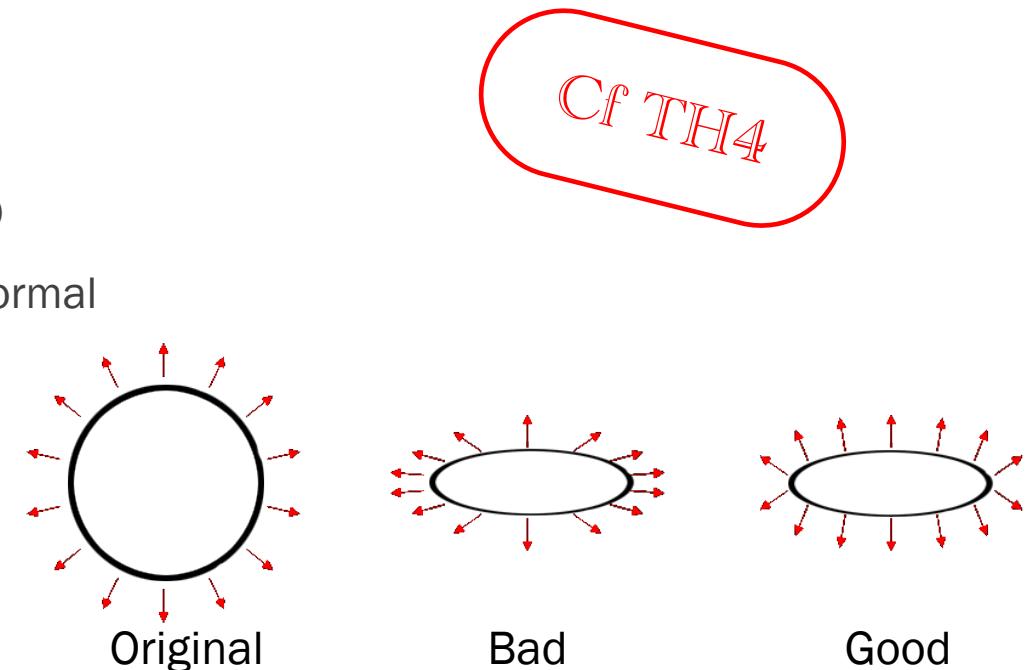
EXERCISE 03

- Implement Gouraud **diffuse** light on a sphere
 - You need to make the sphere and export it from Blender
 - Don't forget to put the light somewhere in your scene (it's a vec3 sent to the shader)
 - Don't forget that normals are NOT vectors but are unitary vectors:
 - They transform in space differently! (M, V, P)
 - See next slide ;)



NORMALS TRANSFORMATION BETWEEN THE MODEL AND WORLD COORDINATES

- A normal \mathbf{n} is a unitary vector
- Any transformation applied on it should preserve $\text{norm}(\mathbf{n}) == 1.0$
- We cannot do $\text{Model} * \mathbf{n}$ as this would result in non-normalized normal
- How does a normal transform?
 - Translation should NOT affect the normal
 - Rotation is applied to normal as with the position
 - Uniform scaling of position does NOT affect the direction of the normal
 - Non-uniform scaling affects the direction of the normal
- How to take all of this into consideration?
 - As usual, linear algebra to the rescue!



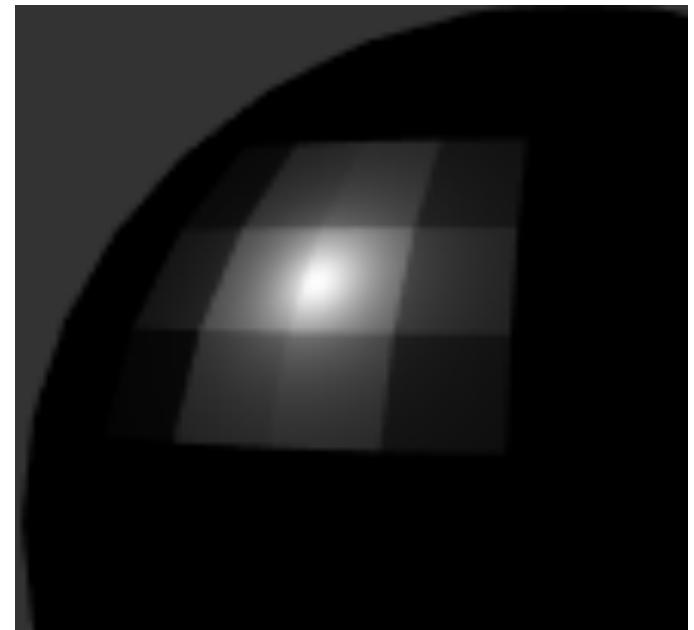
NORMALS TRANSFORMATION BETWEEN THE MODEL AND WORLD COORDINATES - PROOF

Cf TH4

- Let $n \in \mathbb{R}^3$ be a normal at a point $p \in \mathbb{R}^3$ and $M \in \mathbb{R}^{3 \times 3}$ a transformation matrix
- We are looking how n transforms when we transform p .
- Let $W \in \mathbb{R}^{3 \times 3}$ be an unknown transform for n corresponding to $Mp \in \mathbb{R}^3$
 - We have $n \mapsto Wn$ and $p \mapsto Mp$
- Every point define a tangent plane and perpendicularity between p and n must be preserved (ie: $n^T p = 0$)
- By using the transformations we have $n^T p = 0 \Leftrightarrow (Wn)^T (Mp) = 0$
- By using the transpose, we want: $n^T W^T M p = 0$
- One solution is $W^T M = I \Leftrightarrow W^T = M^{-1} \Leftrightarrow W = M^{-T}$ The part that interests you for now
- We found what we were looking for! The transformation of the normal vector using the model matrix!

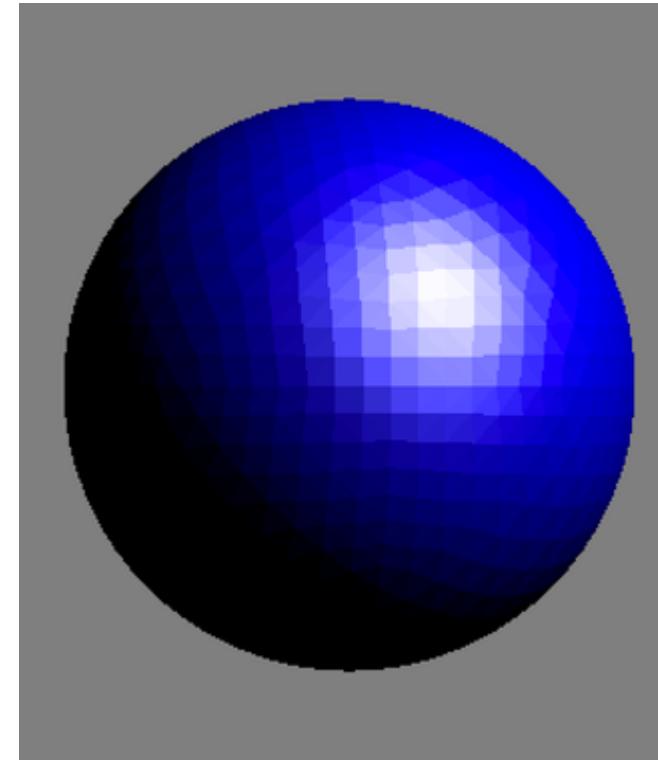
EXERCISE 04

- Implement Phong **diffuse** light on a sphere
- Compare with the previous exercise!
(the light must be placed carefully to see the difference)
- What is the difference between `sphere_coarse.obj` and `sphere_smooth.obj` ?
- Compare Gouraud and Phong: rough idea of how many computations are needed?



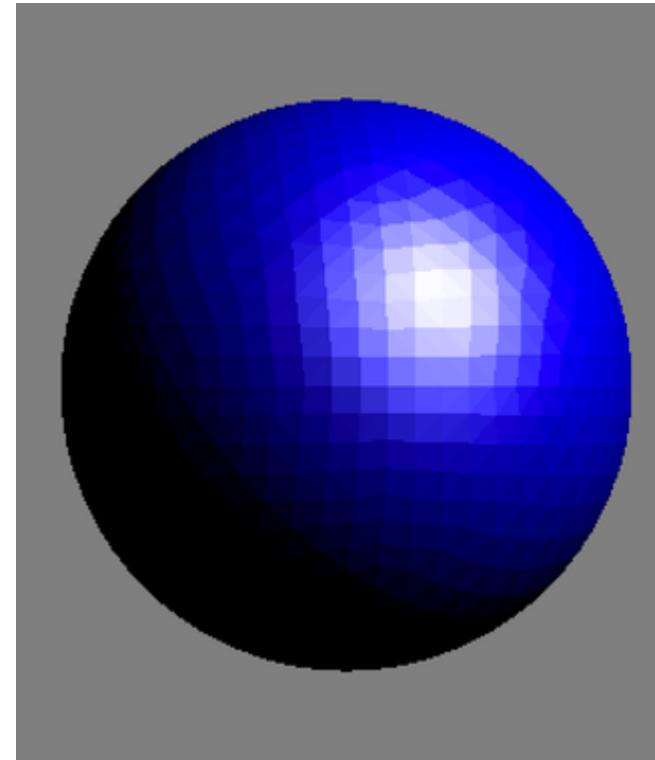
EXERCISE 05

- Implement Gouraud specular light
 - Which vectors do you need?
 - In which shader are those vectors computed?

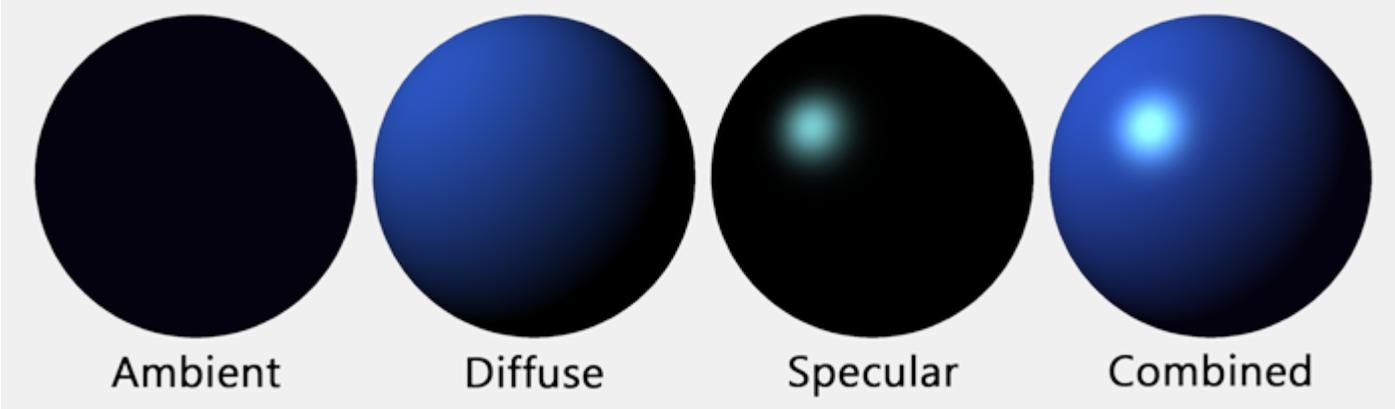


EXERCISE 06

- Implement Phong specular light
 - Which vectors do you need?
 - In which shader are those vectors computed?



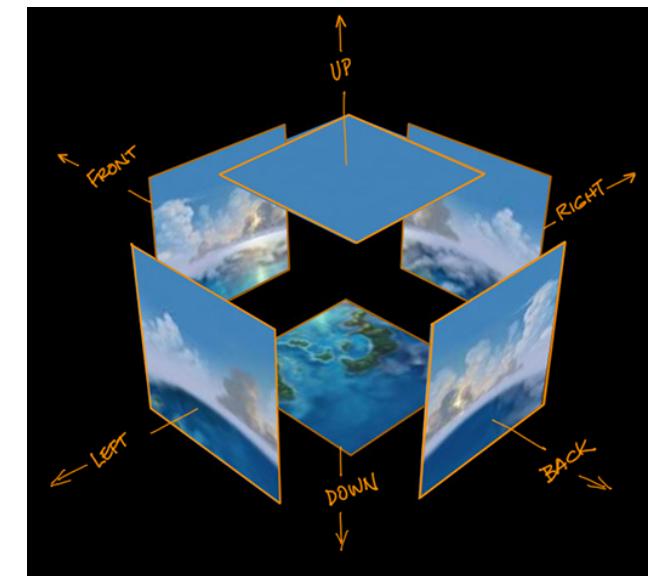
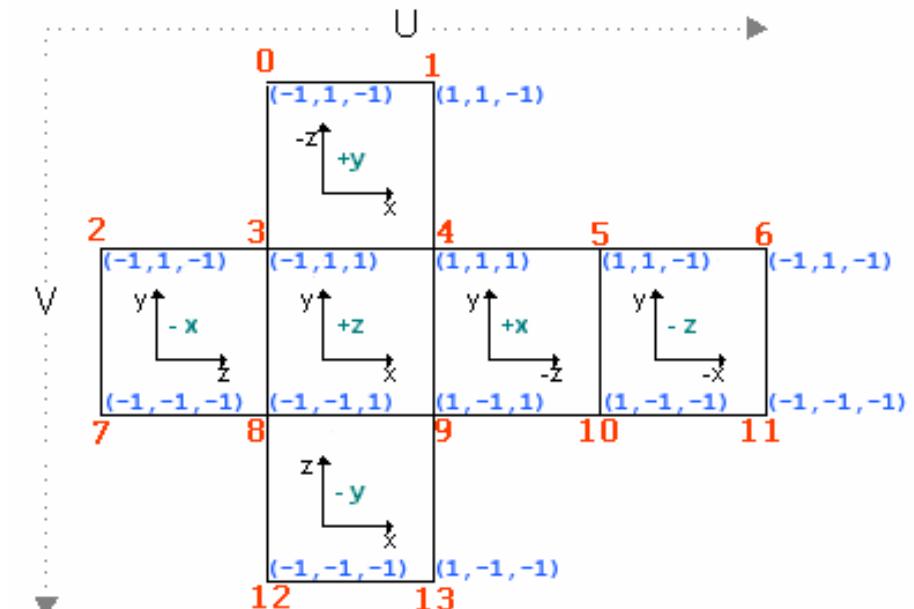
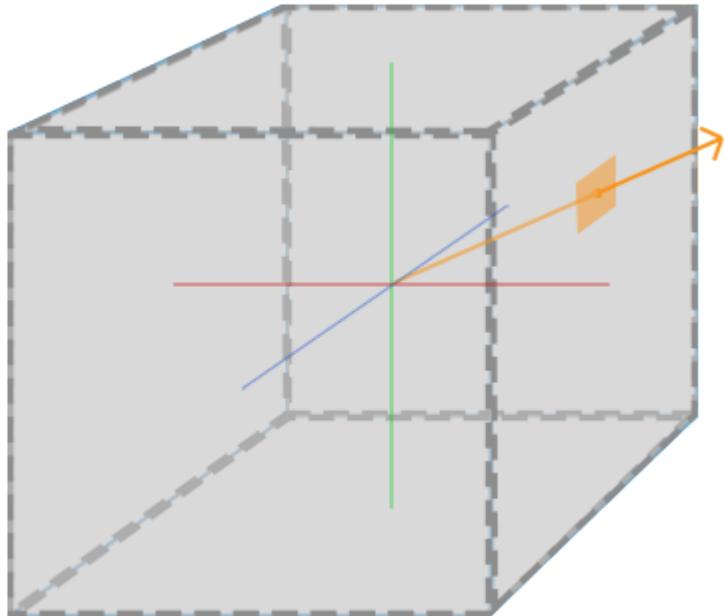
EXERCISE 07



- Implement the light equation on your object
 - Warning: In the solution only one light and a simplified light equation!
 - You can do way better!
- $L_o(\vec{x}) = A + E + \sum_N (diff + spec) \cdot L_i \cdot att_i \cdot \cos \theta$

EXERCISE 08

- Cubemap
 - Find cubemaps, eg: <http://www.humus.name/index.php?page=Textures>
 - 6 textures
 - Or use the one with the exercises LAB3/textures/cubemaps/yokohama3 (from the same website)
 - To sample the cube map use a *samplerCube*
 - A good way to debug cubemaps is to make your own texture with a different color on each face!



EXERCISE 09

- Implement a reflective object



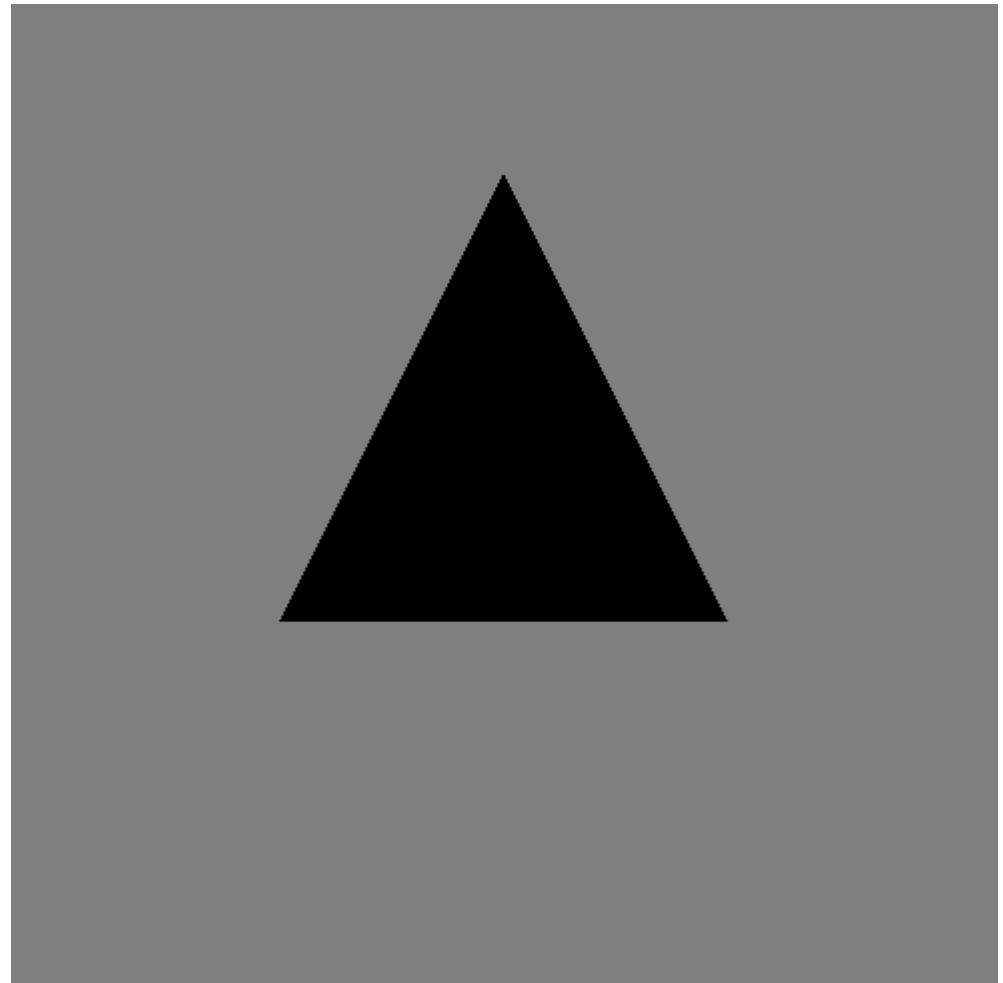
EXERCISE 10

- Implement a refractive object



EXERCISE 11

- Debug the code using Renderdoc (<https://renderdoc.org/>) to identify where is the bug. The goal is to make the texture appears on the triangle.
 - Hint: Appendix 0 may help you

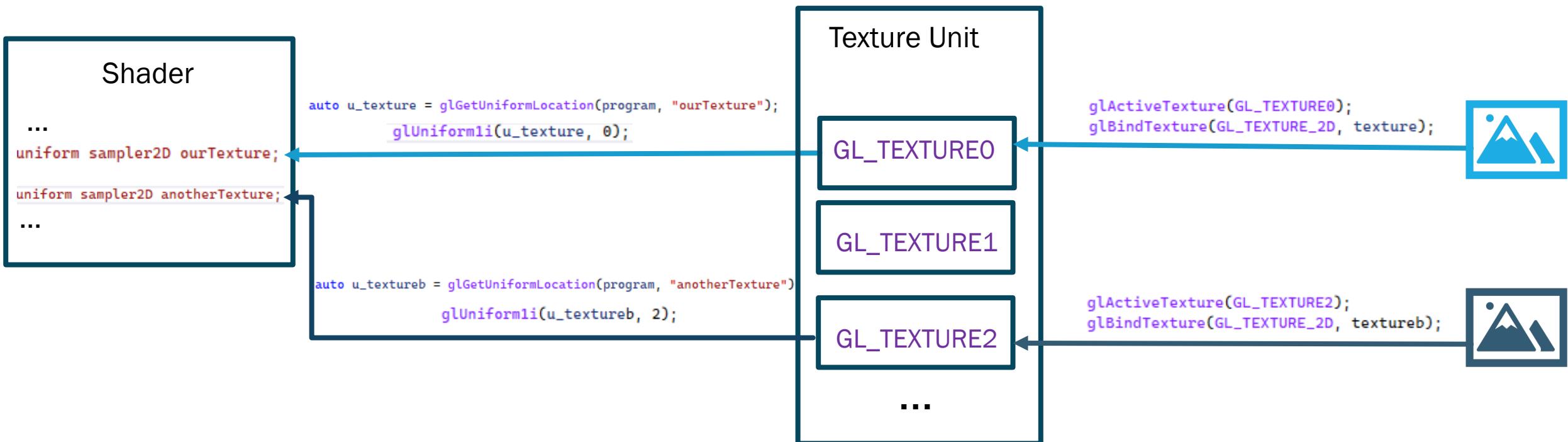


SUPPLEMENTARY EXERCISE FOR THE BRAVE: EQUIRECTANGULAR TO CUBEMAP

- You can find beautiful equirectangular cubemaps here: <https://hdrihaven.com/hdris/>
- But you need to write some code to convert them to a cubemap for OpenGL!
- Resources:
 - <http://paulbourke.net/panorama/cubemaps/>
 - <https://stackoverflow.com/questions/34250742/converting-a-cubemap-into-equirectangular-panorama>



APPENDIX 0: MULTIPLE TEXTURE, TEXTURE UNIT AND TEXTURE



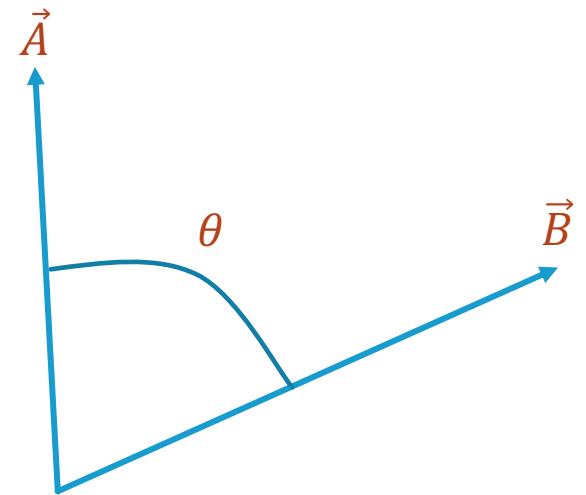
Additional resource:

<https://stackoverflow.com/a/55800428>

[https://www.khronos.org/opengl/wiki/Sampler_\(GLSL\)#Binding_textures_to_samplers](https://www.khronos.org/opengl/wiki/Sampler_(GLSL)#Binding_textures_to_samplers)

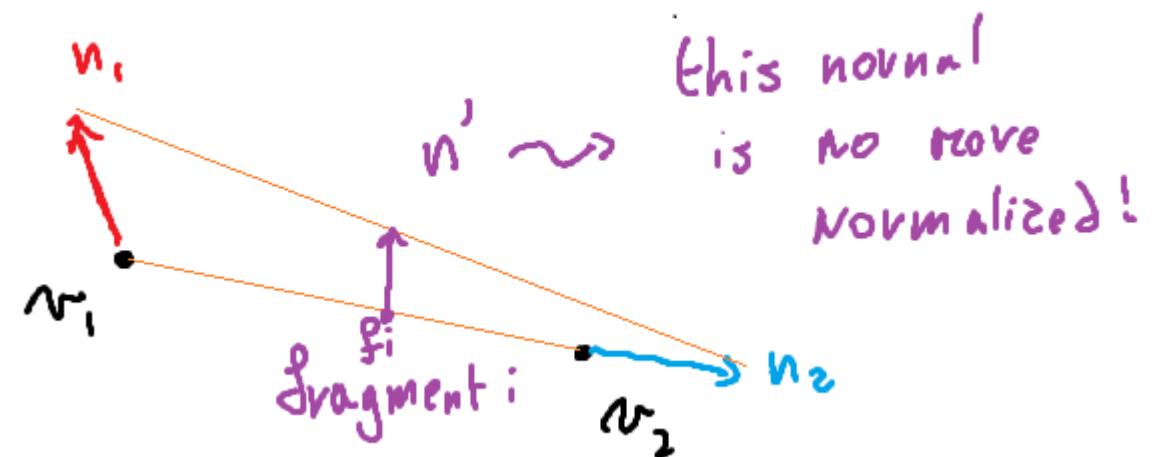
APPENDIX 1: COSINE BETWEEN NORMALIZED VECTORS

- $\text{norm}(\vec{A}) = 1$ and $\text{norm}(\vec{B}) = 1$
- $\text{dot}(\vec{A}, \vec{B}) = \text{norm}(\vec{A}) \cdot \text{norm}(\vec{B}) \cdot \cos \theta = 1 \cdot 1 \cdot \cos \theta = \cos \theta$
- $\Rightarrow \cos \theta = \text{dot}(\vec{A}, \vec{B})$



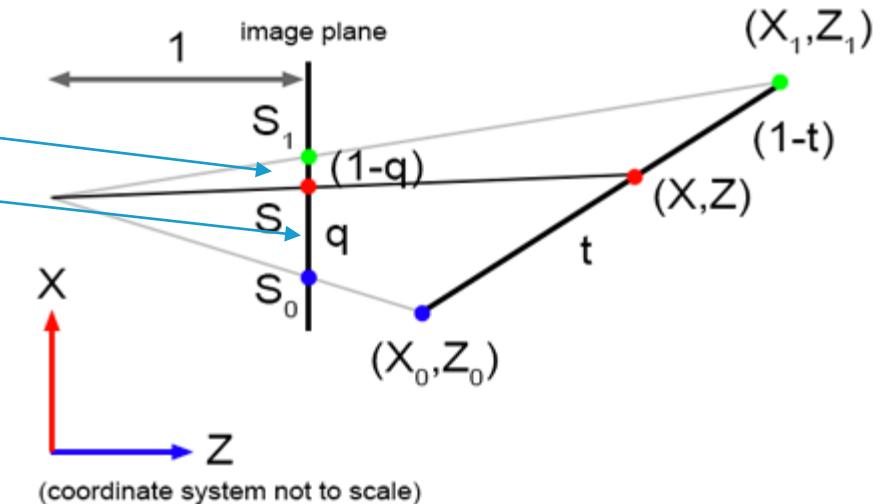
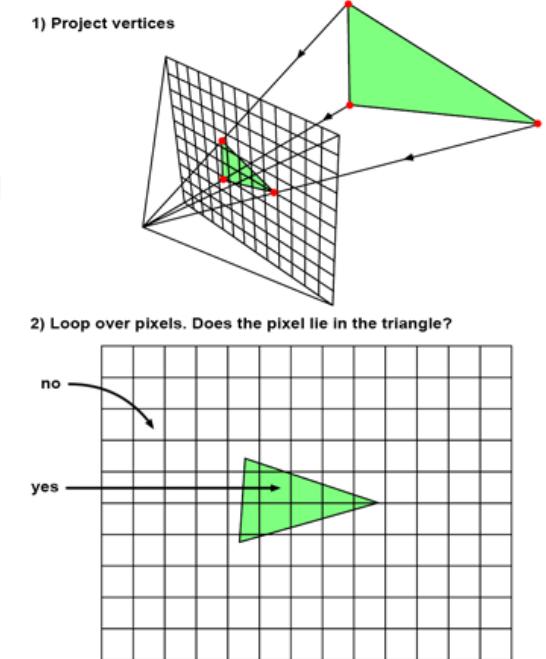
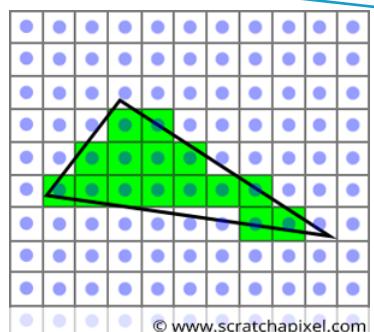
APPENDIX 2: PHONG AND THE NON NORMALIZED NORMALS

- When you interpolate a normal between the vertex and the fragment shader
 - The “normality” condition is not necessarily fulfilled
 - You need to re-normalize your normal when you receive them in the fragment shader!
 - $n = \text{normalize}(n)$



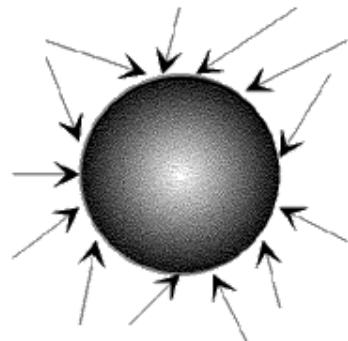
APPENDIX 3: HOW MANY FRAGMENTS - RASTERIZATION

- The screen is a discretized grid, you need to map 3D objects to it
- The projection of 3D elements change using the depth of the points!
 - Less fragments for far away points
 - More fragments for closer points

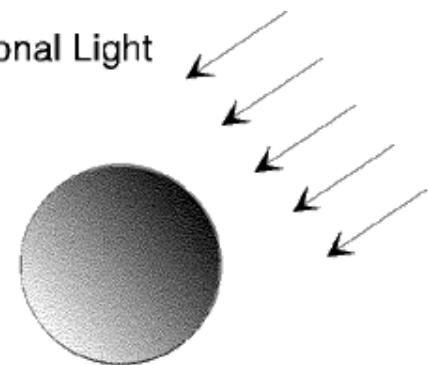


APPENDIX 4 : KIND OF LIGHT SOURCES

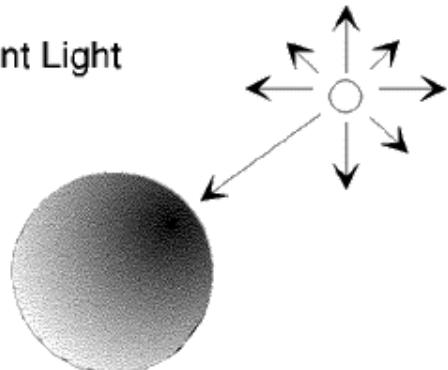
Ambient Light



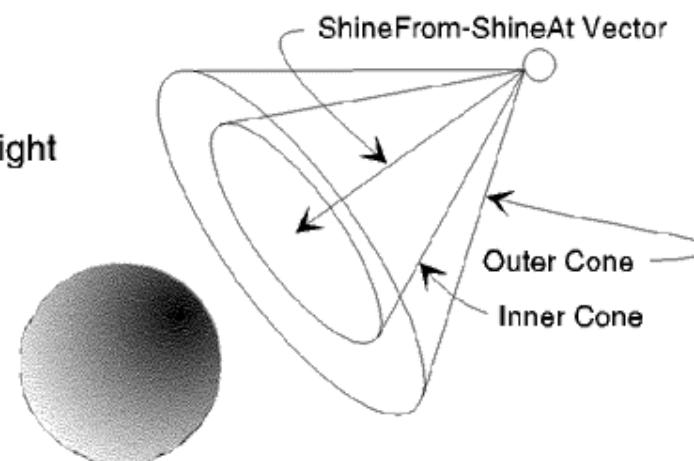
Directional Light



Point Light



Spot Light



3D Lighting: History, Concepts, and Techniques
Arnold Gallardo
2000

Digital Lighting and Rendering
Jeremy Birn
2013

APPENDIX 5 : RENDERING EQUATION NOT ENOUGH

Although the equation is very general, it does not capture every aspect of light reflection. Some missing aspects include the following:

- **Transmission**, which occurs when light is transmitted through the surface, such as when it hits a **glass** object or a **water** surface,
- **Subsurface scattering**, where the spatial locations for incoming and departing light are different. Surfaces rendered without accounting for subsurface scattering may appear unnaturally opaque — however, it is not necessary to account for this if transmission is included in the equation, since that will effectively include also light scattered under the surface,
- **Polarization**, where different light polarizations will sometimes have different reflection distributions, for example when light bounces at a water surface,
- **Phosphorescence**, which occurs when light or other **electromagnetic radiation** is **absorbed** at one moment in time and emitted at a later moment in time, usually with a longer **wavelength** (unless the absorbed electromagnetic radiation is very intense),
- **Interference**, where the wave properties of light are exhibited,
- **Fluorescence**, where the absorbed and emitted light have different **wavelengths**,
- **Non-linear** effects, where very intense light can increase the **energy level** of an **electron** with more energy than that of a single **photon** (this can occur if the electron is hit by two photons at the same time), and **emission** of light with higher frequency than the frequency of the light that hit the surface suddenly becomes possible, and
- **Relativistic Doppler effect**, where light that bounces on an object that is moving in a very high speed will get its wavelength changed; if the light bounces at an object that is moving towards it, the impact will compress the **photons**, so the wavelength will become shorter and the light will be **blueshifted** and the photons will be packed more closely so the photon flux will be increased; if it bounces at an object that is moving away from it, it will be **redshifted** and the photons will be packed more sparsely so the photon flux will be decreased.

For scenes that are either not composed of simple surfaces in a vacuum or for which the travel time for light is an important factor, researchers have generalized the rendering equation to produce a **volume rendering equation**^[5] suitable for **volume rendering** and a **transient rendering equation**^[6] for use with data from a **time-of-flight camera**.

APPENDIX 6 : LIGHT EQUATION – OTHER MODELS

- The Blinn-Phong model we saw is just **one** visually pleasing **model!**
- There is always a tradeoff between visual accuracy and difficulty to compute

- Lambertian:

$$f(\omega_i, x, \omega_o) = \frac{a}{\pi}$$

- Lafourture:

$$f(\omega_i, x, \omega_o) = \frac{a}{\pi} + b(-\omega_i^T A \omega_o)^k$$

- Blinn:

$$f(\omega_i, x, \omega_o) = \frac{a}{\pi} + b \cos^c b \langle \omega_i | \omega_o \rangle$$

- Phong:

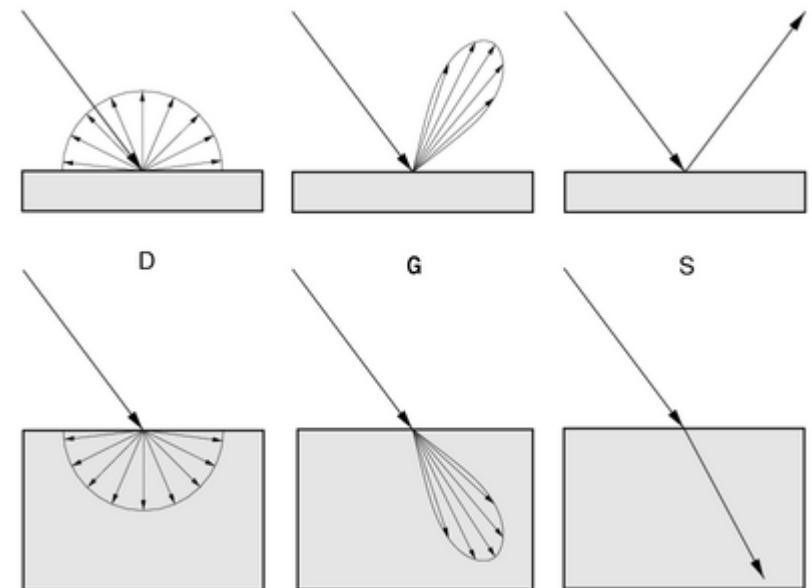
$$f(\omega_i, x, \omega_o) = \frac{a}{\pi} + b \cos^c (2 \langle \omega_i | n \rangle \langle \omega_o | n \rangle - \langle \omega_i | \omega_o \rangle)$$

- Ward:

$$f(\omega_i, x, \omega_o) = \frac{a}{\pi} + \frac{b}{4\pi c^2 \sqrt{\langle \omega_i | n \rangle \langle \omega_o | n \rangle}} \exp\left(-\frac{\tan^2 b \langle \omega_i | \omega_o \rangle}{c^2}\right)$$

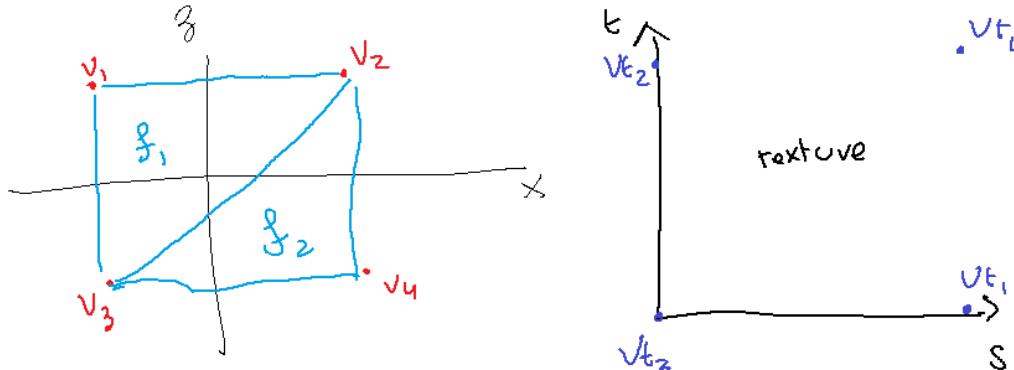
APPENDIX 7 : BRDF NOT ALONE

- BRDF is for Reflectance
- BTDF is for Transmittance (light inside the material!)
- BSDF is both of them

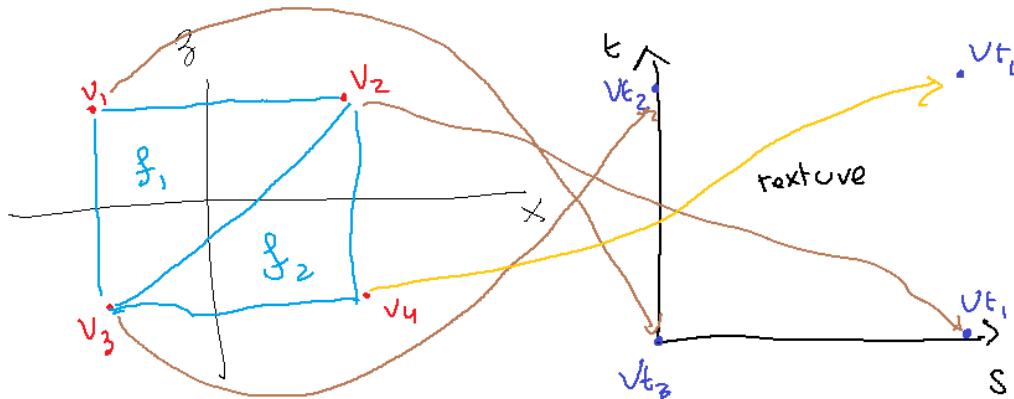


APPENDIX 8 : MATERIAL FILE - SOLUTION

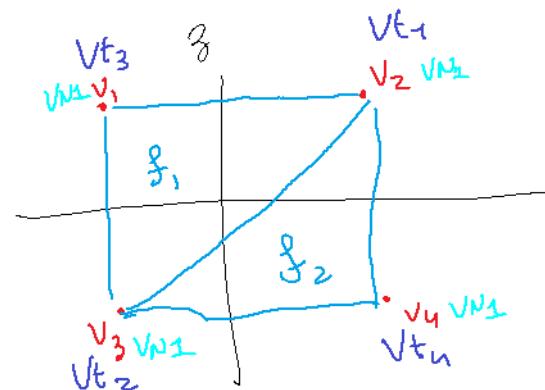
```
plane.obj (D:\VR2020\LAB03\objects) - GVIM1
Fichier Edition Outils Syntaxe Tampons F
# Blender v2.81 (sub 16) OBJ File:
# www.blender.org
v -1.000000 0.000000 1.000000 1
v 1.000000 0.000000 1.000000 2
v -1.000000 0.000000 -1.000000 3
v 1.000000 0.000000 -1.000000 4
vt 1.000000 0.000000 1
vt 0.000000 1.000000 2
vt 0.000000 0.000000 3
vt 1.000000 1.000000 4
vn 0.0000 1.0000 0.0000 1
s off
F 2/1/1 3/2/1 1/3/1
F 2/1/1 3/4/1 3/2/1
```



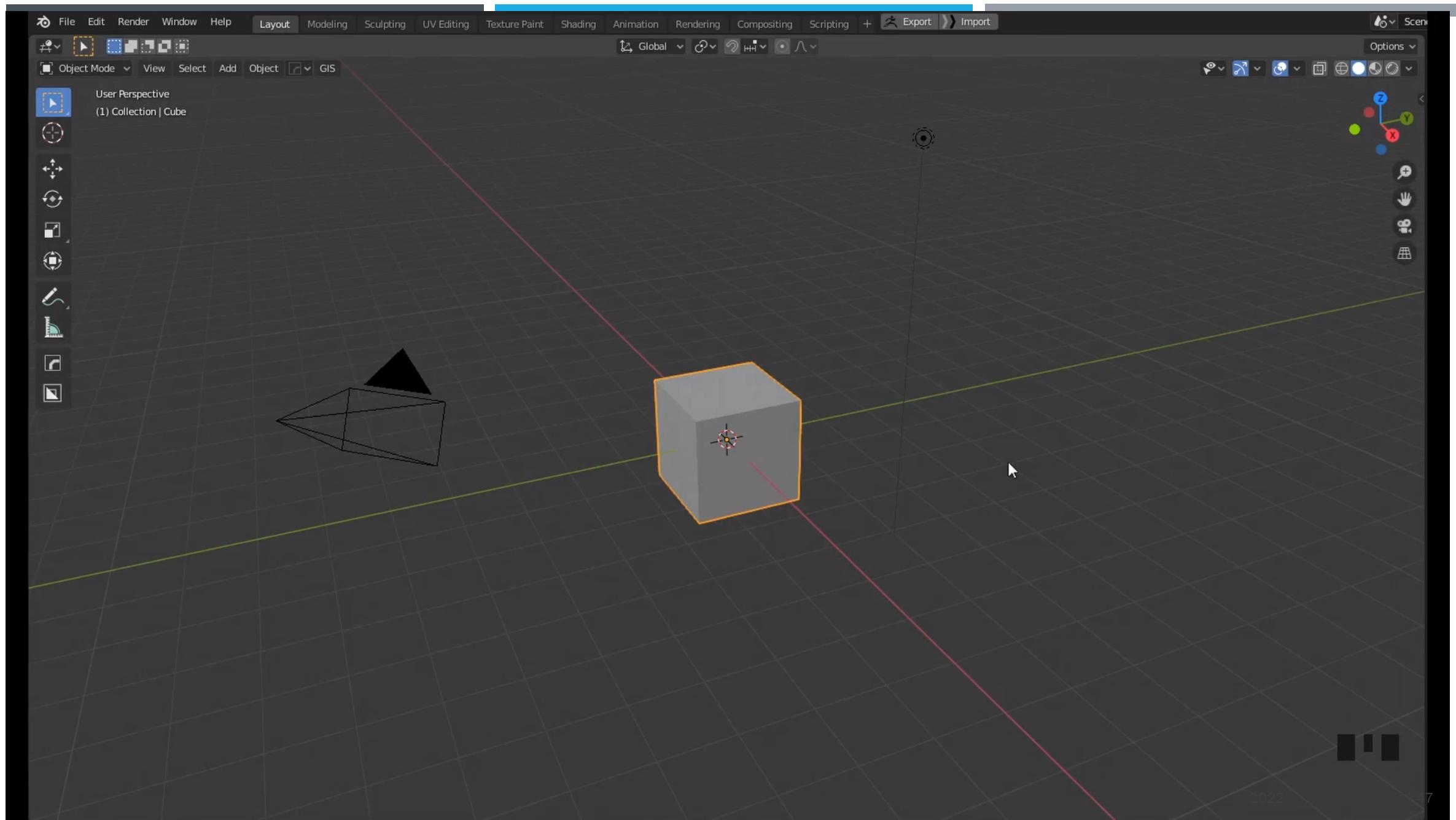
```
plane.obj (D:\VR2020\LAB03\objects) - GVIM1
Fichier Edition Outils Syntaxe Tampons F
# Blender v2.81 (sub 16) OBJ File:
# www.blender.org
v -1.000000 0.000000 1.000000 1
v 1.000000 0.000000 1.000000 2
v -1.000000 0.000000 -1.000000 3
v 1.000000 0.000000 -1.000000 4
vt 1.000000 0.000000 1
vt 0.000000 1.000000 2
vt 0.000000 0.000000 3
vt 1.000000 1.000000 4
vn 0.0000 1.0000 0.0000 1
s off
F 2/1/1 3/2/1 1/3/1
F 2/1/1 3/4/1 3/2/1
```



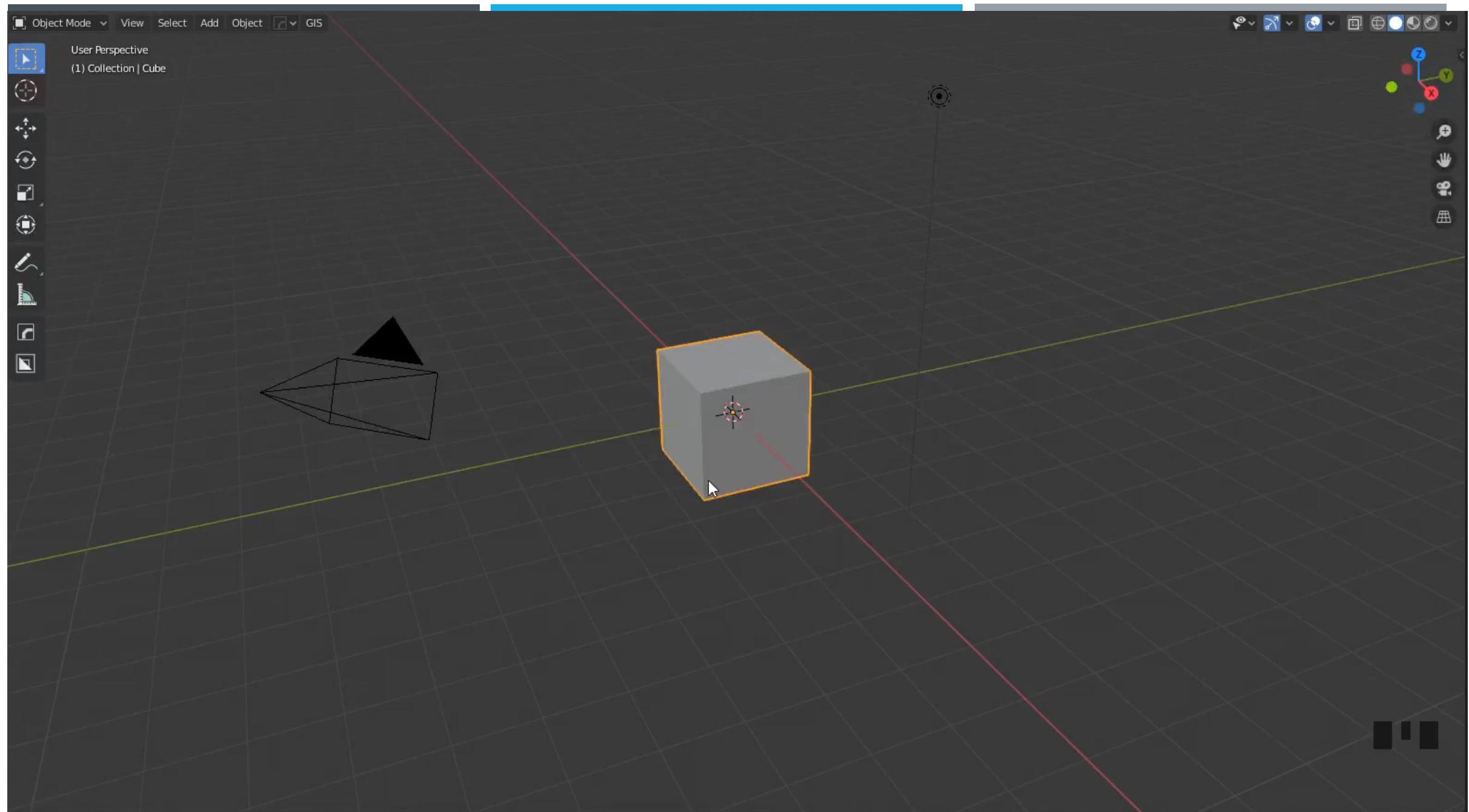
```
plane.obj (D:\VR2020\LAB03\objects) - GVIM1
Fichier Edition Outils Syntaxe Tampons F
# Blender v2.81 (sub 16) OBJ File:
# www.blender.org
v -1.000000 0.000000 1.000000 1
v 1.000000 0.000000 1.000000 2
v -1.000000 0.000000 -1.000000 3
v 1.000000 0.000000 -1.000000 4
vt 1.000000 0.000000 1
vt 0.000000 1.000000 2
vt 0.000000 0.000000 3
vt 1.000000 1.000000 4
vn 0.0000 1.0000 0.0000 1
s off
F 2/1/1 3/2/1 1/3/1
F 2/1/1 3/4/1 3/2/1
```



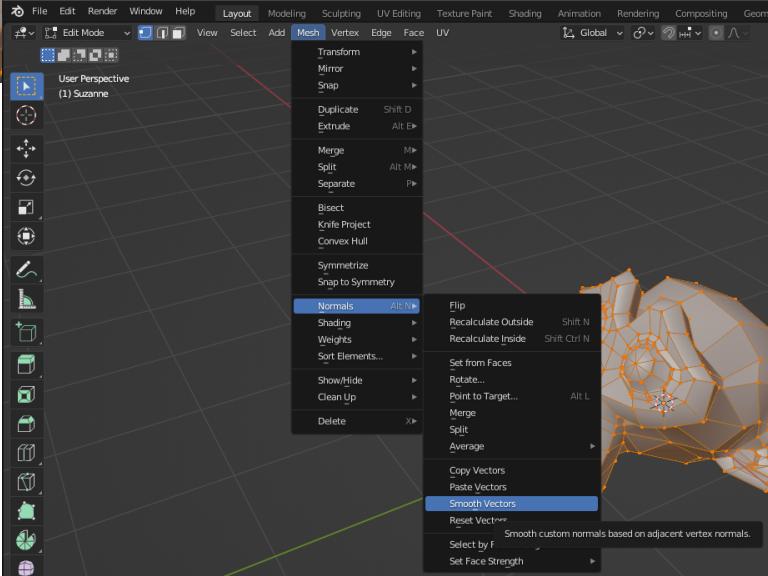
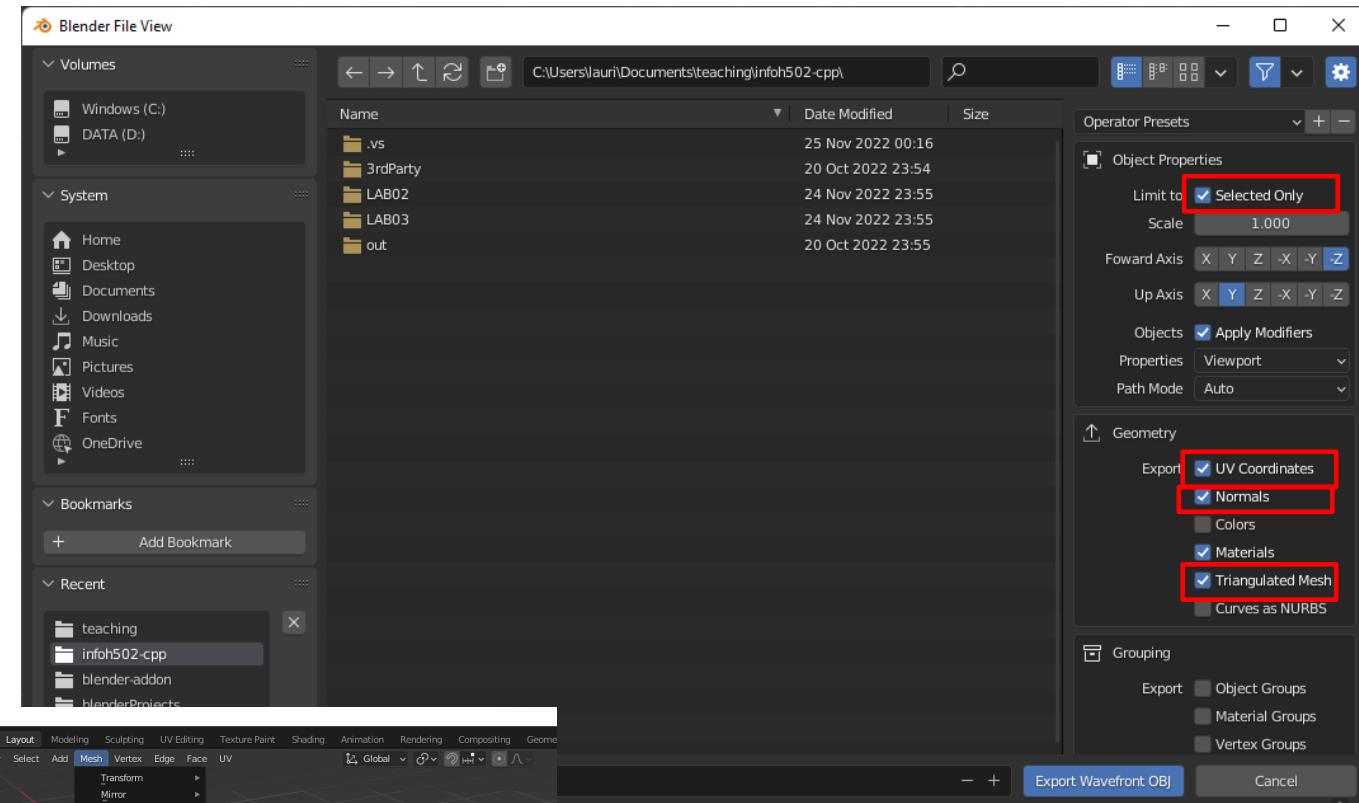
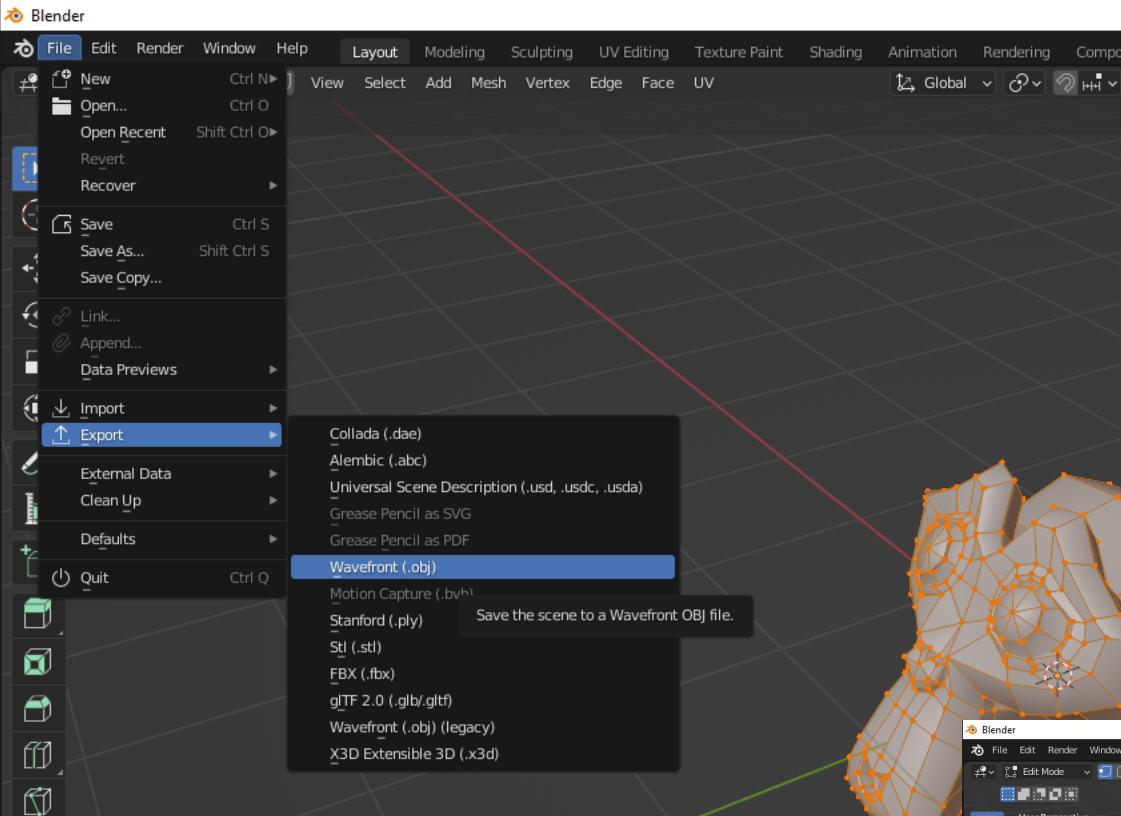
APPENDIX 9: CREATE AN OBJECT WITH BLENDER AND EXPORT IT



APPENDIX 9: SPHERE COARSE VS SMOOTH



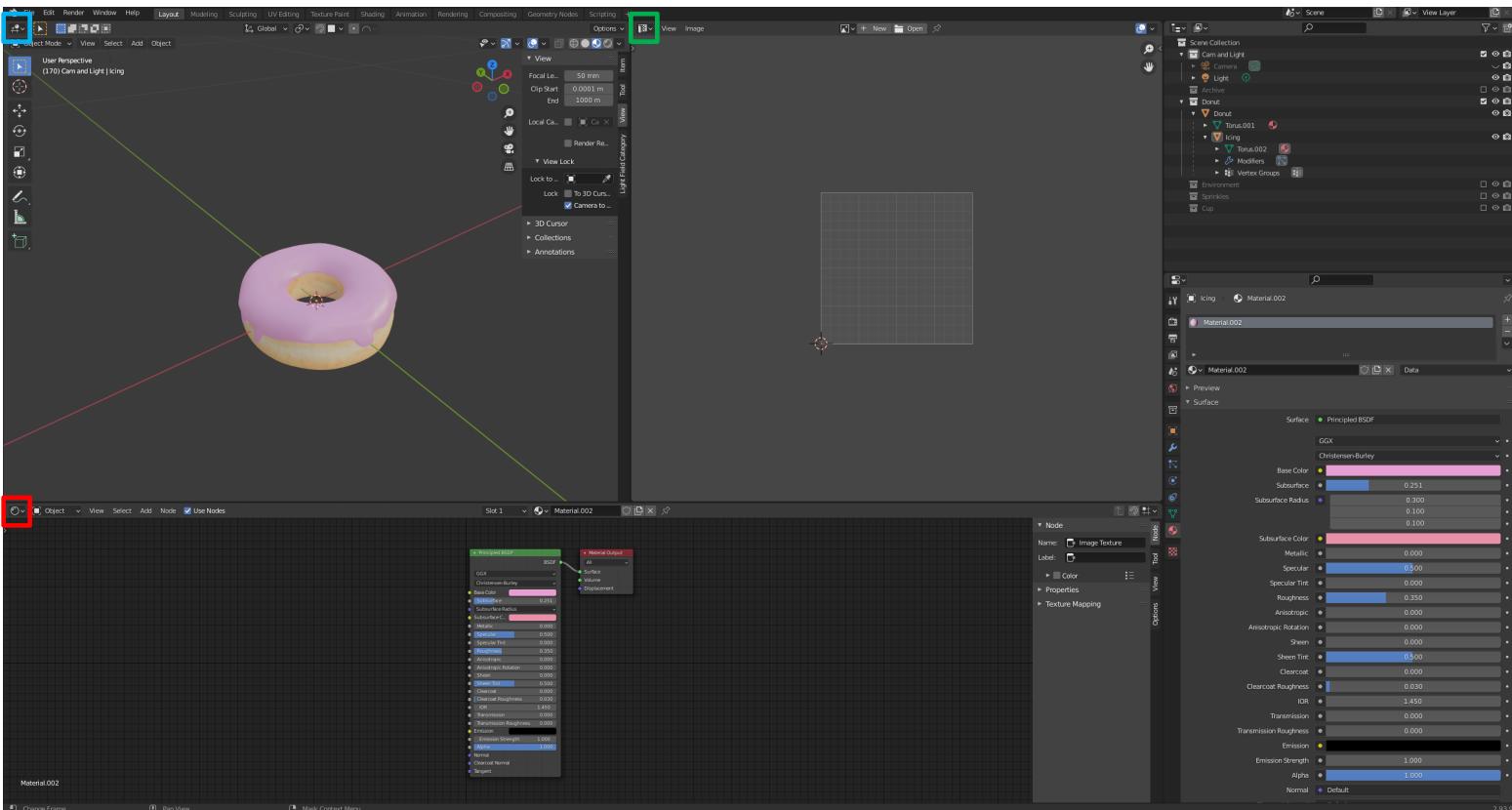
APPENDIX 9: USE BLENDER TO CREATE .OBJ FILES



(optional) if you want vertices to have different normal in a same face

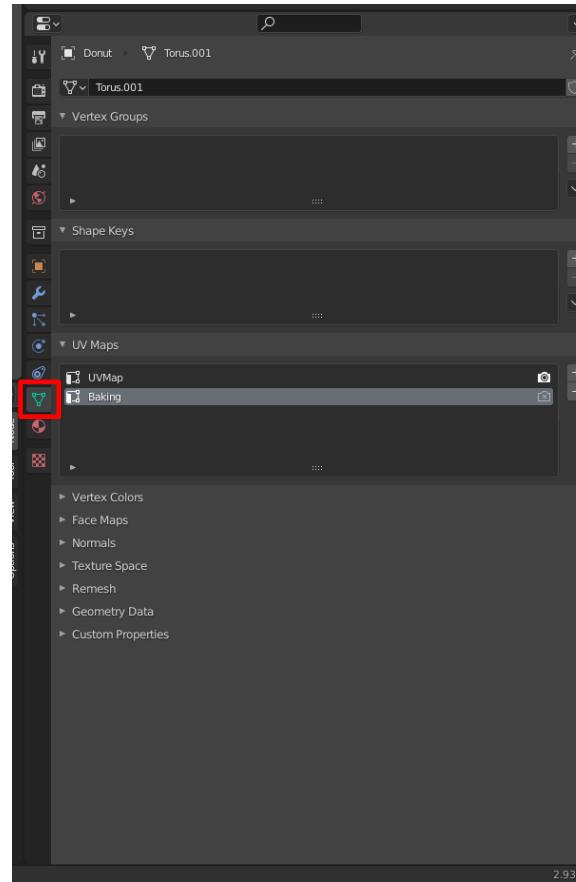
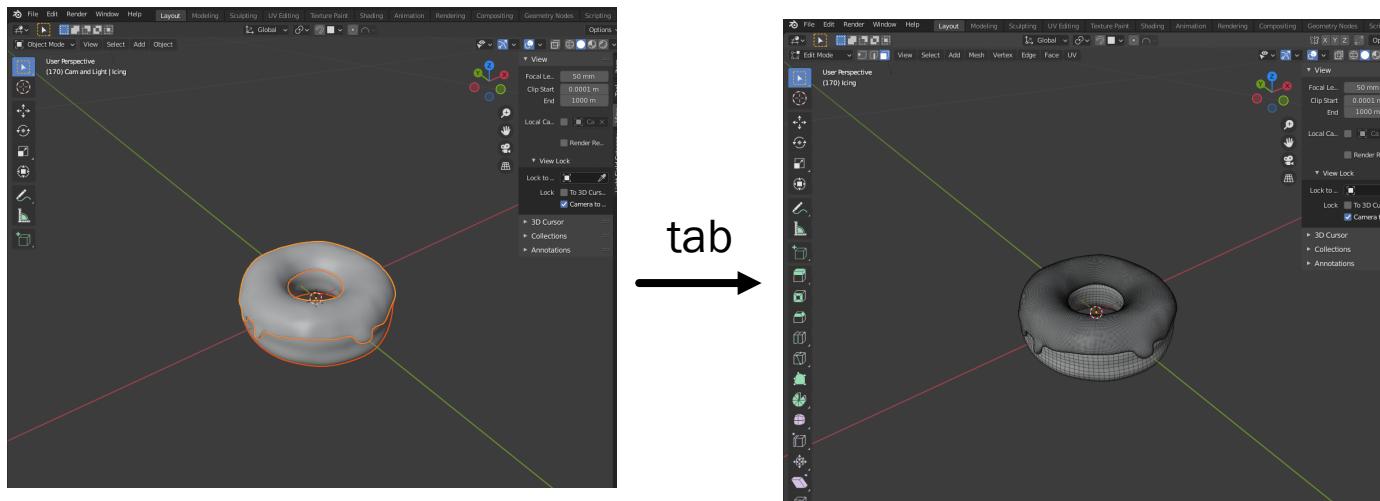
APPENDIX 10 : BAKING TEXTURE (BLENDER 2.9)

- Split your workspace into 3 panels
- **Upper-left** panel shows the 3D viewport, **upper-right** panel shows the uv Editor, **bottom** panel shows the shader editor



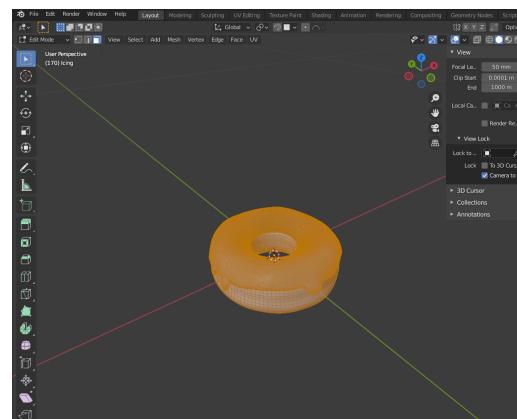
APPENDIX 10 : BAKING TEXTURE

- For each part of your object, go to the **Object Data Properties tab** and create a new uv map called Baking
- Then select all your object and go into edit mode (tab key)

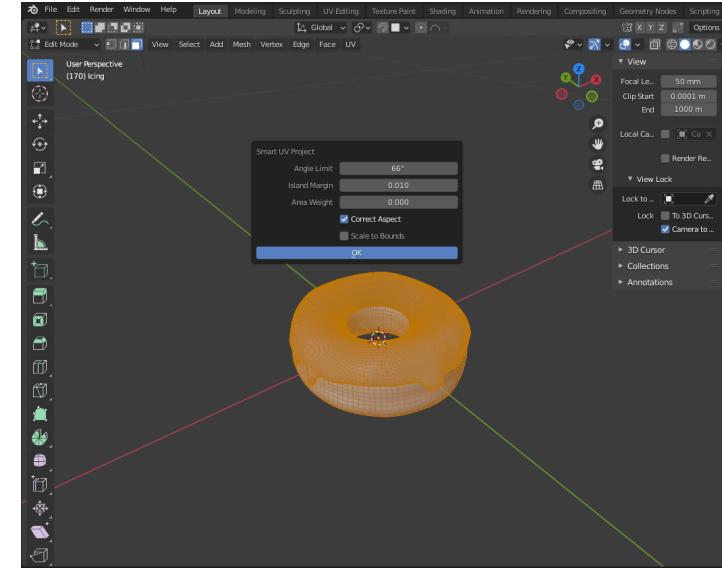
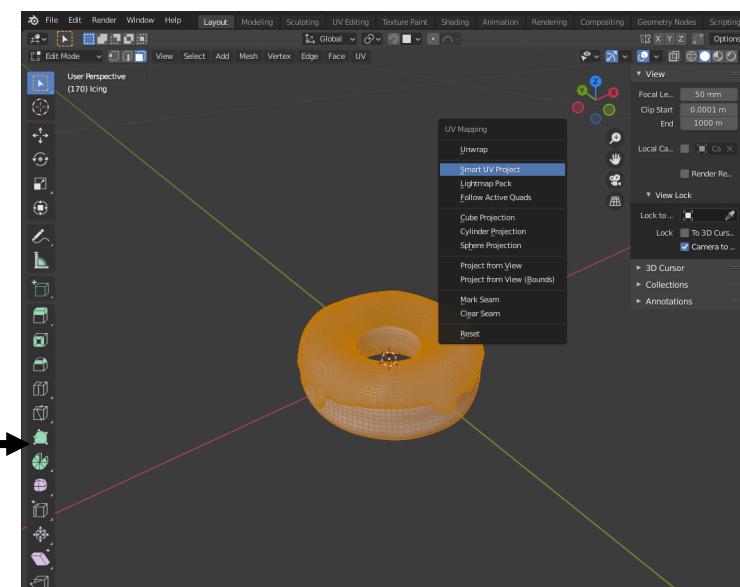


APPENDIX 10: BAKING TEXTURE

- Select all vertices with key « a » then use key « u » to show the unwrapping menu
- Select the smart uv project option
- Change the Island Margin to 0.01 and click OK

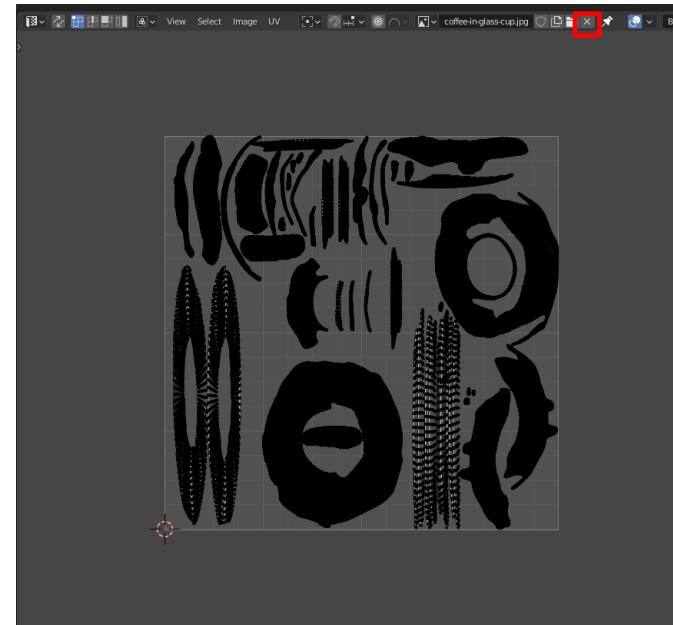
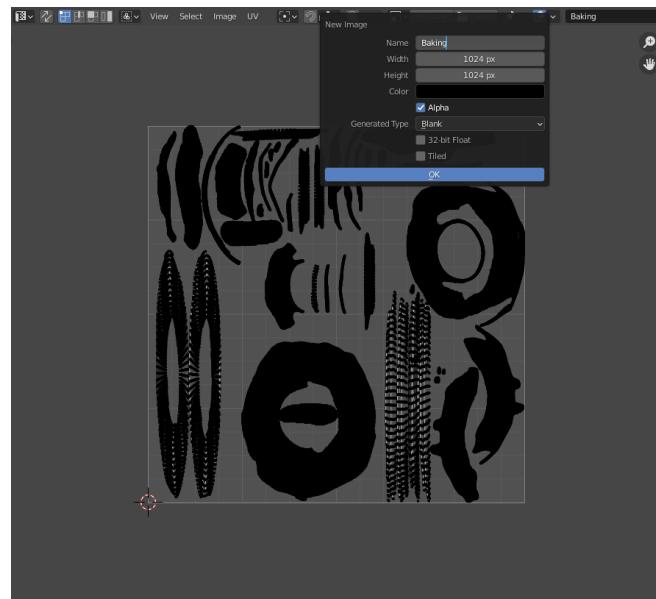


U



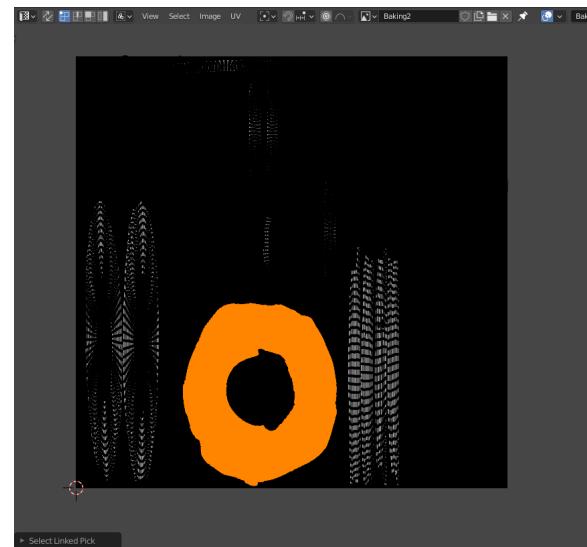
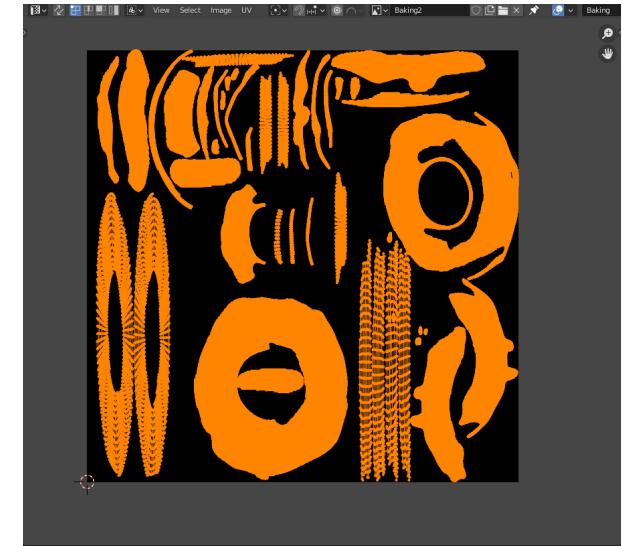
APPENDIX 10: BAKING TEXTURE

- In the uv editor panel you will sell your object « flattened »
- Remove the current texture
- Create a new image with the name baking and the dimension you want



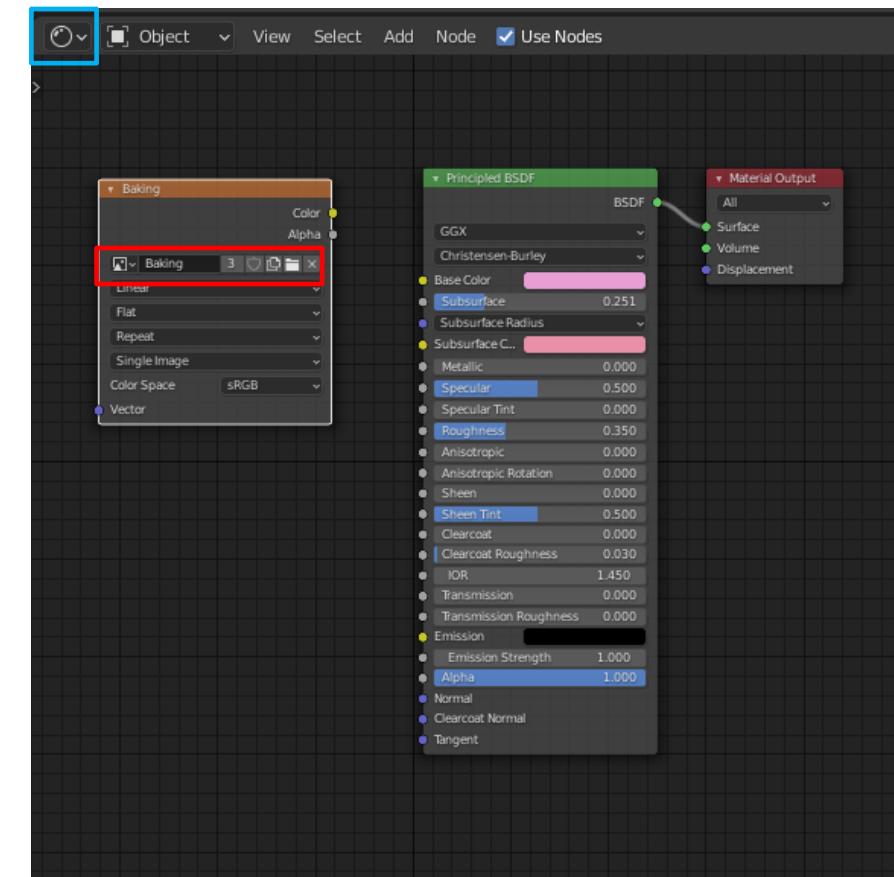
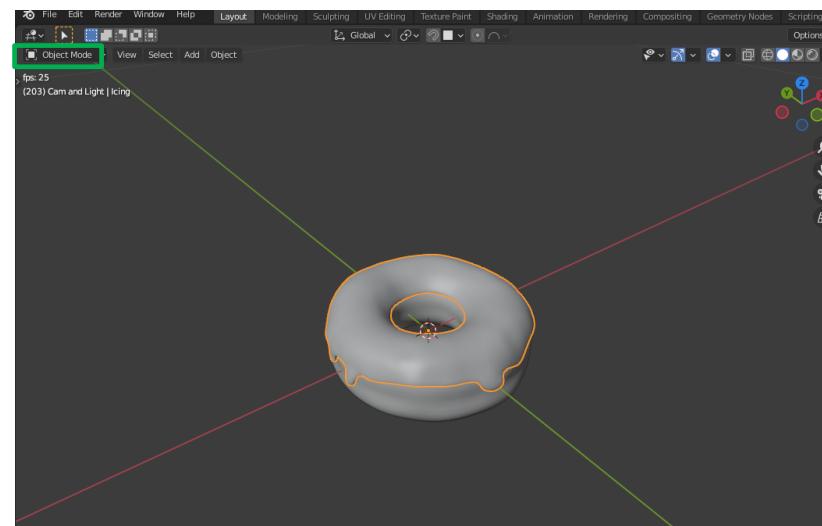
APPENDIX 10: BAKING TEXTURE

- Press « a » to select all points
- Press « l » to select a block of linked vertices
- Move the blocks around (key « g », « r », « s ») to optimize the space, you want to take as much space as possible in the image without having overlapping block



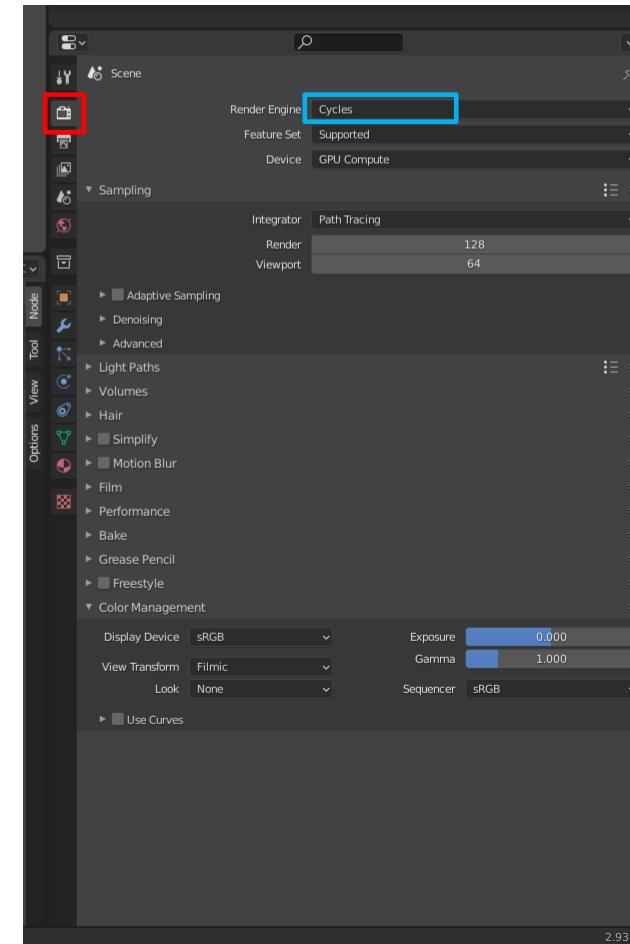
APPENDIX 10: BAKING TEXTURE

- Go back to **object mode** (tab)
- For each material, add a **texture image** node in the **shader editor**
- Use the **baking texture** you have just created
- ! This node must be unconnected to the reste of the nodes presents



APPENDIX 10: BAKING TEXTURE

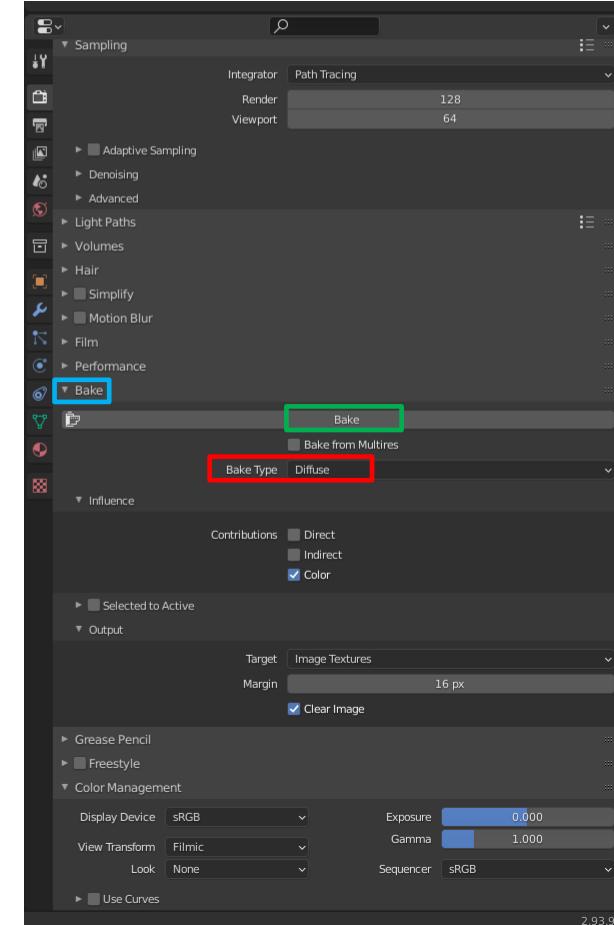
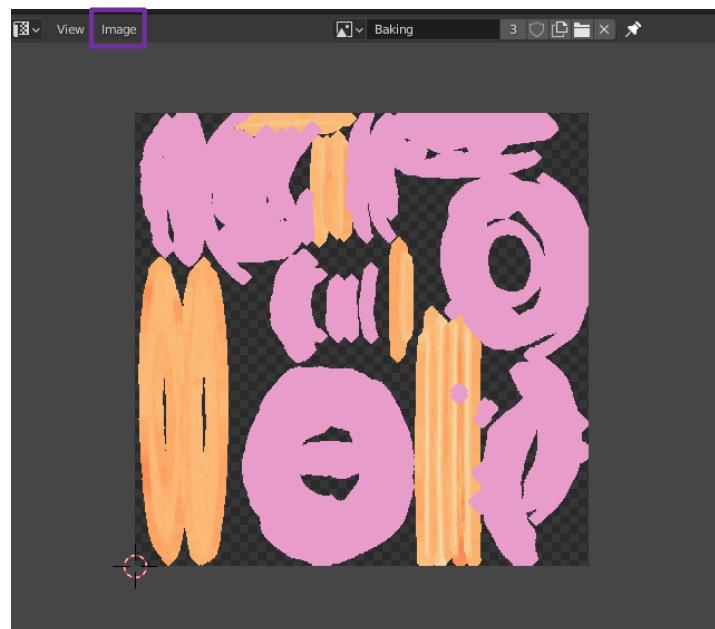
- Make sure all objects that goes together are selected (here the donut and the icing)
- For each object, make sure you added the baking image texture and it is selected (white border)
- For each object make sure the baking uv map is selected
- Go to **render properties** and select the **Cycles** render engine



APPENDIX 10: BAKING TEXTURE

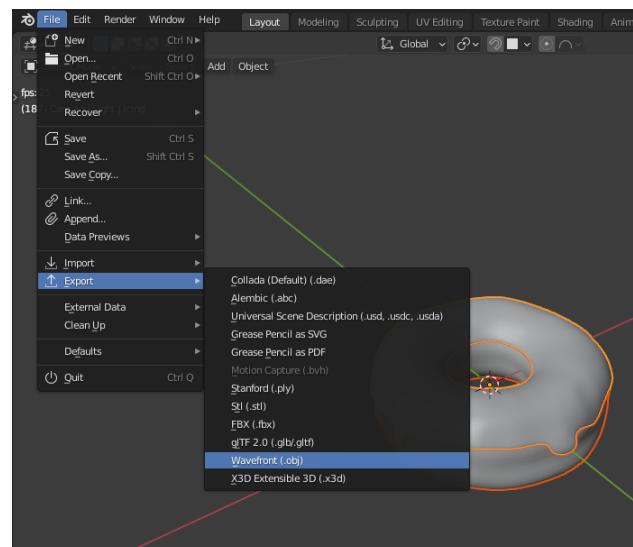
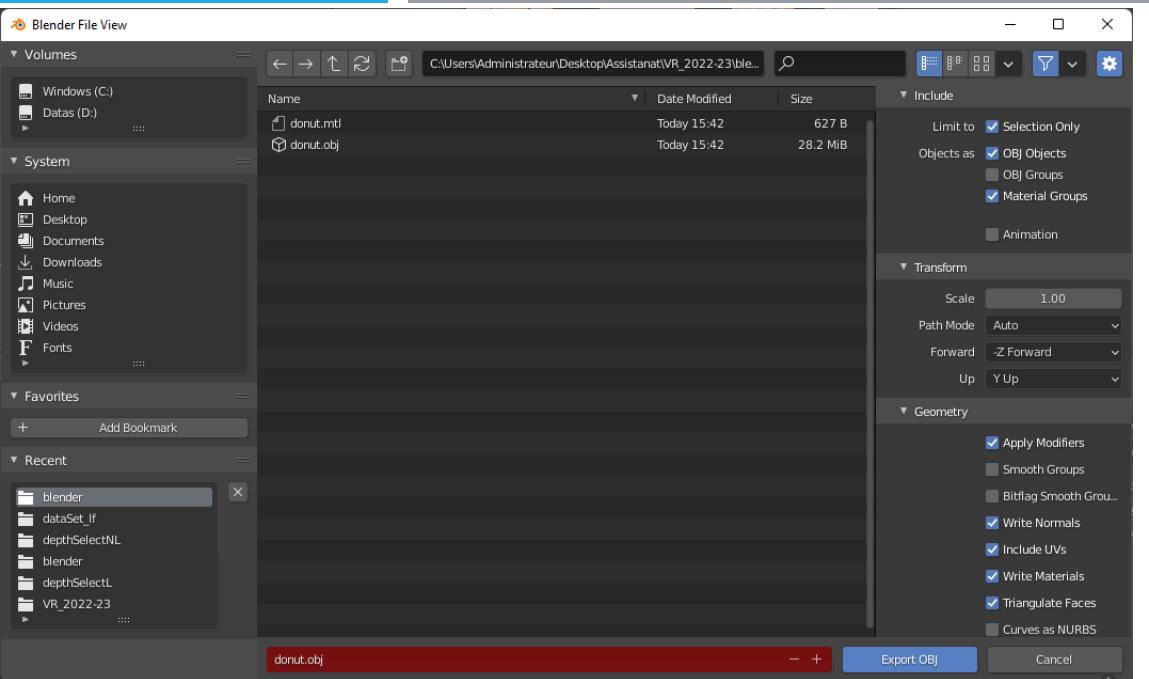
- Go to the **bake** section
- Put bake type as **diffuse** then click on **bake**
- A few moment later you should see your texture image, **save it** as a jpg

Note : here you can see some blocks of vertices overlap, try to avoid this



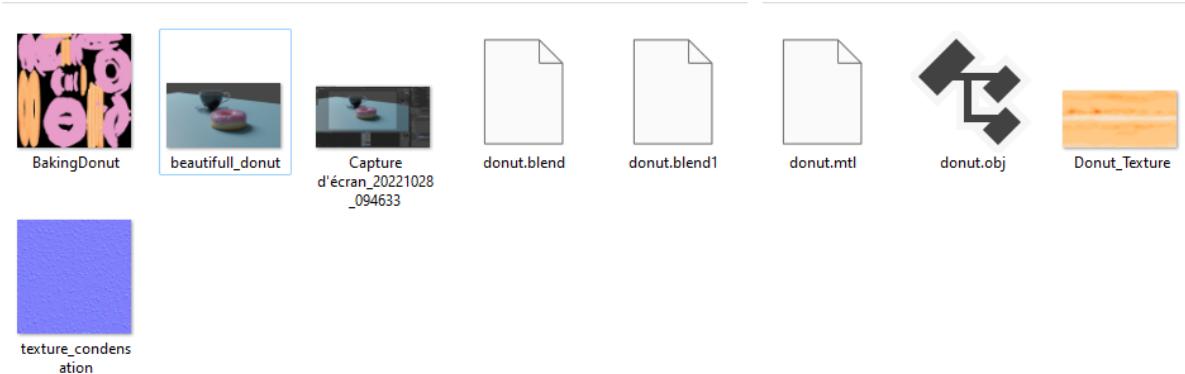
APPENDIX 10: BAKING TEXTURE

- Go to file -> Export -> Wavefront (.obj)
- Select « Material Groups » and « Triangulate Faces » in the option
- Choose the name and export your object



APPENDIX 10: BAKING TEXTURE

- Now you have a .obj file with a texture image that you can use in opengl
- This is a summary, more detail here :
<https://www.youtube.com/watch?v=wG6ON8wZYLc>



APPENDIX 10: DIFFERENCE SMOOTH AND NON SMOOTH SPHERE

- Even with Phong shading, in “non smooth” sphere you see the triangles!
- This happens because the normals on the sphere are not smooth
- Each vertex uses the same normal as the face’s normal
- In blender, when we **smooth** the mesh, what we are doing is to set at each vertex the mean normal between adjacent faces

