Lab 3 - 3D polygonal meshes: interactive scene navigation

Laura Mazzuca

07/09/2021

0 Assignments

1. Load and view 3D polygonal meshes
   (a) create new material
   (b) create Wave Motion shader
   (c) create Toon shader
2. Interactive scene navigation
   (a) Horizontal pan of the camera
3. Scene objects transformations
   (a) handle single objects transformations in modifyModelMatrix() func wrt either WCS or OCS, depending on user's choice
   (b) light object translation

1 Load and view 3D polygonal meshes

1.1 Create new material

The new material created is called "Yellow Rubber". To define a new material, we follow the Phong model so four components have to be defined: ambient, diffuse, specular and shininess. Since yellow is, in RGB, (1.0, 1.0, 0.0), we define:

- **ambient** as (0.1f,0.1f,0.015f), which is the low-light color;
- **diffuse** as (1.0f,1.0f,0.0f), which represents the base object color;
- **specular** as (0.7f,0.7f,0.04f), which represents the highlight color;
- **shininess** as 10.0f, which represents the surface smoothness.

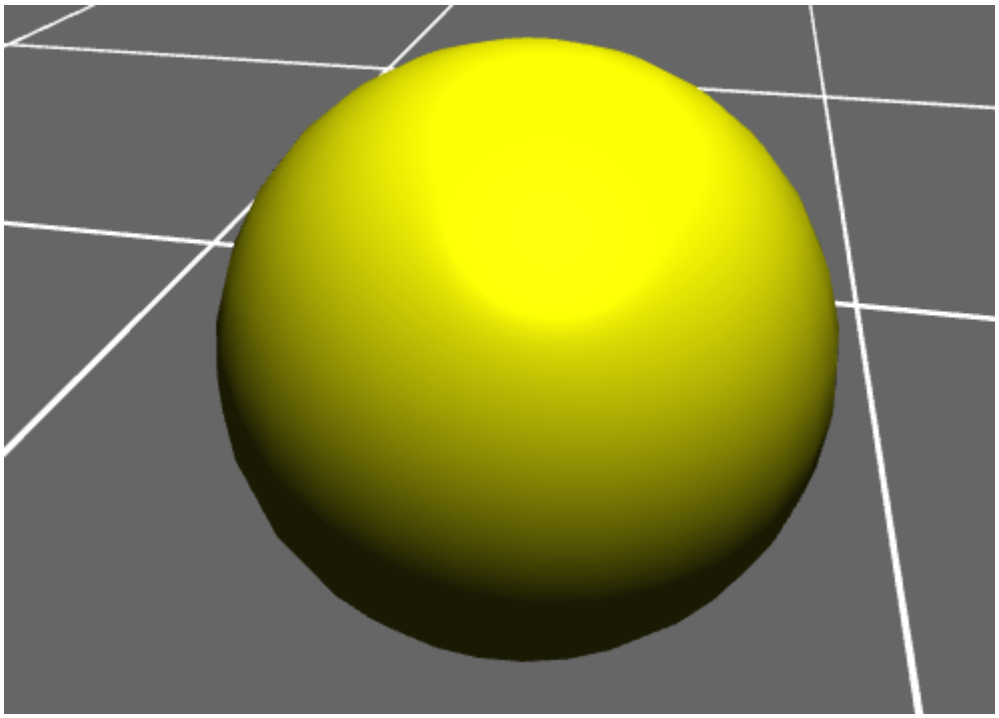

Figure 1: The Yellow Rubber material on the smooth ball object.

1.2 Create Wave Motion shader

The Wave Motion shader was implemented in the **v_wave.glsl** by duplicating the aPos vertex position and assigning it a y value computed with the function:

$v.y = aPos.y + h * sin(a * time + 10.0 * aPos.x) * sin(a * time + 10.0 * aPos.z)$;

Here, s, h and $time$ are parameters passed runtime by the program using the $glUniform()$ primitives.

The Phong shading algorithm was also added to enable the material's properties to modify the look of the mesh.

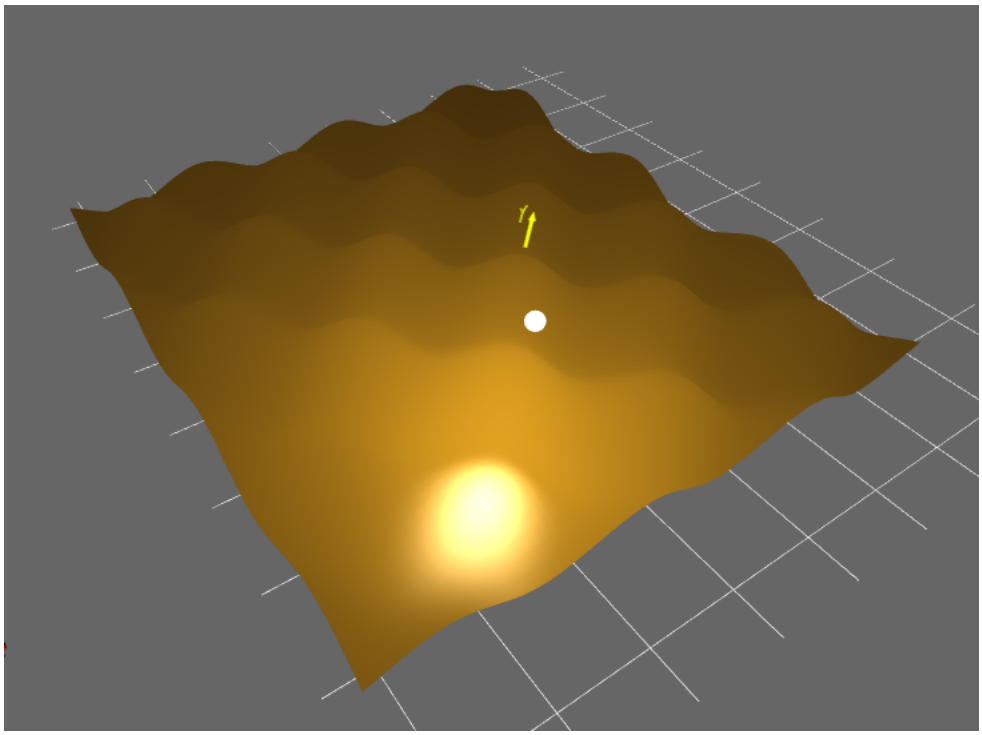

Figure 2: The plane rendered with the Wave Phong shader.

1.3 Create Toon shader

The Toon shader divides the mesh into areas depending on how high the intensity of light hitting it is and assigns an homogeneous color to each interval of intensity defined. It's implemented in the vertex shader v_toon.glsl and fragment shader f_toon.glsl.

v_toon.glsl computes the normalized vectors E,L and N to pass to the fragment shader. E represents the direction from the eye to the object's position, L the direction from the light to the object and N is the normalized normal of the current vertex.

It also receives informations from the program about the material so that it can forward the material.diffuse property (i.e. the base color of the mash) to the fragment shader.

Then, **f_toon.glsl** computes the intensity of the light hitting the mesh by making a dot product between the normalized L and normalized N. The resulting intensity i is then checked to decided which coefficient should be multiplied by the diffuse color d to obtain a brighter or darker shade:

- i>0.95 very bright, $d* = 1.5$
- 0.5<i ≤0.95 normal color, $d* = 1.0$
- 0.25<i ≤0.5 slightly dark shade, $d* = 0.75$
- 0.15<i ≤0.25 dark shade, $d* = 0.5$

- $0.05 < i \leq 0.15$ very dark shade, $d* = 0.25$
- $i \leq 0.05$ darkest shade (black) $d* = 0.0$

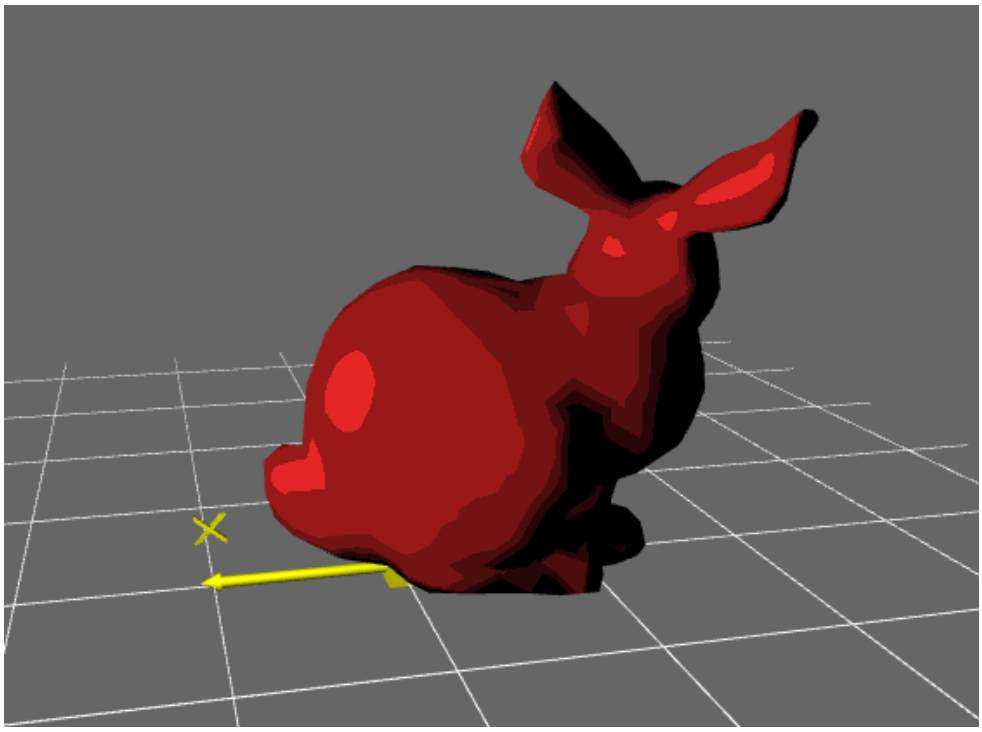

Figure 3: The Bunny rendered with Toon Shader.

2 Interactive scene navigation

2.1 Horizontal pan of the camera

The horizontal pan was implemented in the functions $moveCameraRight()$ and $moveCameraLeft()$. Also, a $CAMERA\_TRASLATION\_SPEED\_H$ was defined.

The computation of the translation is done by computing the *slide_vector* and then multiplying by the speed. This result is either added, if moving right, or subtracted, if moving left, to the position and the target variables of the $ViewSetup$ structure.

3 Scene objects transformations

3.1 Single object transformations in WCS or OCS

Both options were implemented in the $modifyModelMtrix()$.

For **OCS** transformations, since there can only be one modification at the time of the model, the model matrix is always multiplied by all of the transformations matrices this way we don't need to check which one is currently happening.

For **WCS** we do the same, but first multiply the Model Matrix by its inverse to move to the coordinate system in WCS, then apply the transformations, and finally multiply again by the Model matrix to get back to the OCS.

3.2 Light transformations

The light object can be modified just like the others, but the important thing that was done here was update the *light.position* property. This way, it can be updated in the shaders making them reactive to the light's transformations.