Assignment 4 - INF01009
Applying Textures and Texture filtering to Arbitrary Texture-enabled Models

Guilherme G. Haetinger

2021-10-21

---

1 Introduction

Textures have many uses in computer graphics. They can be used not only for holding images, but also to hold data of any sort.

In most cases, it is necessary that these textures can have data queried in continuous coordinates. Furthermore, these queries can result artifacts also known as aliasing, which has been the subject of many works and algorithms, **two** of which we'll be covering in this report.

2 New File Structure

In this assignment, we were introduced a new 3D Model representation file. In this format, we get an indicator of whether the model has texture coordinates or not. My new implementation checks if the model is receptive to texture and constructs the *UI* without the related buttons/options.

3 Setting up Texture attributes

There are essentially two attributes related to the textures:

- The texture coordinates, which will be taken care *per* vertex. They will be set inside a vertex attribute buffer for **OpenGL** and will be fed as an attribute to the **Close2GL** vertices. In the latter, we must highlight that it is also divided by the perspective correction;
- The texture filtering mode, which will be set as an option in our `Options` class, processed both by **OpenGL**, setting the texture parameters to either:
  - `GL_NEAREST`: Nearest Neighbor;
  - `GL_LINEAR`: Bilinear Interpolation;
  - `GL_MIPMAP`: MipMap Pyramid with Trilinear Interpolation. It is important to point out that we need to ask **OpenGL** to generate the MipMapped layers.

4 Close2GL Texture Filtering

In this section, we'll go over the three asked features for transforming continuous texture coordinates resulted from the scan conversion process to color.

4.1 Nearest Neighbor

This is the most simple way of fetching the texture color. For a given continuous coordinate c, we calculate the discrete coordinate $c_{nn} = \lfloor c \rceil$. This coordinate will be use to query a color from the texture.

What would happen if this is oversampled (more pixels than coordinates)? The answer is quite simple: We would have chunks of sequential pixels with the same color, which would highlight even more harsh color changes. Check out fig. 1.

What would happen if this is undersampled (more coordinates than coordinates)? The image would turn pixelated and, depending on the amount of detail, can become difficult to understand. Check out section 4.3.

4.2 Bilinear Interpolation

Bilinear interpolation helps us generate smooth gradients for oversampled textures, removing harsh borders. The process is quite simple, for continuous coordinate C:

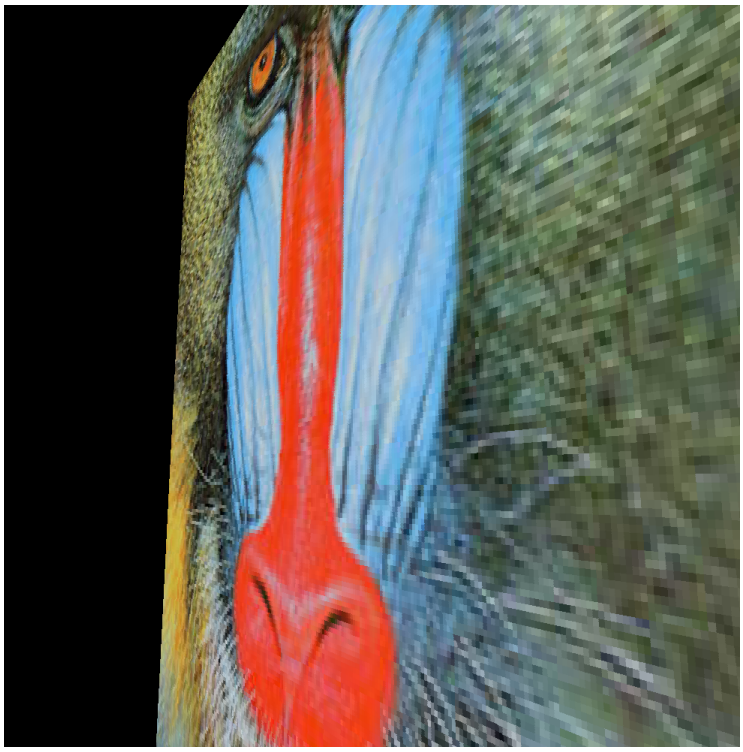


Figure 1: Nearest Neighbor algorithm's harsh edges on top of the Mandrill image.

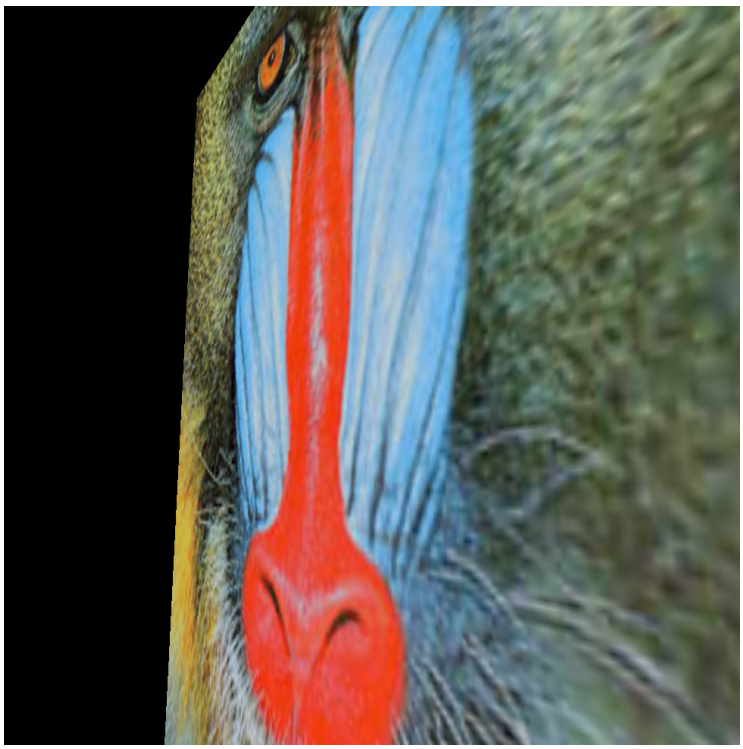


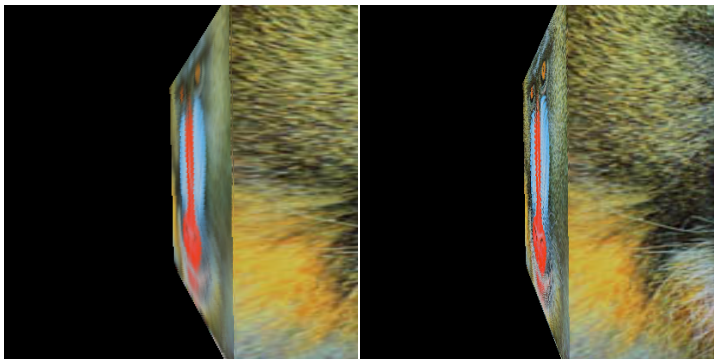
Figure 2: Bilinear algorithm's smooth edges on top of the Mandrill image.

$$a = \begin{bmatrix} \lfloor c.x \rfloor \\ \lfloor c.y \rfloor \end{bmatrix}, b = \begin{bmatrix} \lceil c.x \rceil \\ \lfloor c.y \rfloor \end{bmatrix}, c = \begin{bmatrix} \lfloor c.x \rfloor \\ \lfloor c.y \rfloor \end{bmatrix}, d = \begin{bmatrix} \lceil c.x \rceil \\ \lceil c.y \rceil \end{bmatrix}$$
$$ab = (C.x - a.x) * a.color + (1 - (C.x - a.x)) * b.color$$
$$cd = (C.x - c.x) * c.color + (1 - (C.x - c.x)) * d.color$$
$$color = (C.y - a.y) * ab + (1 - (C.x - a.x)) * cd$$

By the end, we'll have a very smooth surface not very prone to harsh color changes when oversampled fig. 2. It still suffers from aliasing when undersampled, however!

Figure 3: Mipmap angled result (a) compared to Bilinear angled result (b). It is clear the Mipmapped image is clearer even from an angle.

(a) (b)



4.3 MipMapped Trilinear Interpolation

Mipmapping can be done by constructing a pyramid of images, each level having a degree of *blurriness* generated by calculating a mean of the closest 4 points. We, then, use levels of lower resolution to render points further away in the perspective deform, *i.e.* points that ocuppy space that would be mapped to more than one pixel (undersampling). By dynamically changing the level of the texture fed to the viewport, we get a much moother perspective result. We do so by applying **Trilinear Interpolation**, which is basically doing the Bilinear interpolation on two mip levels and interpolating between them.

Figure 4: Zoomed out textures to compare how they behave while undersampled. It's clear (c) holds the advantage.

(a) Nearest Neighbor (b) Bilinear (c) Mipmapped

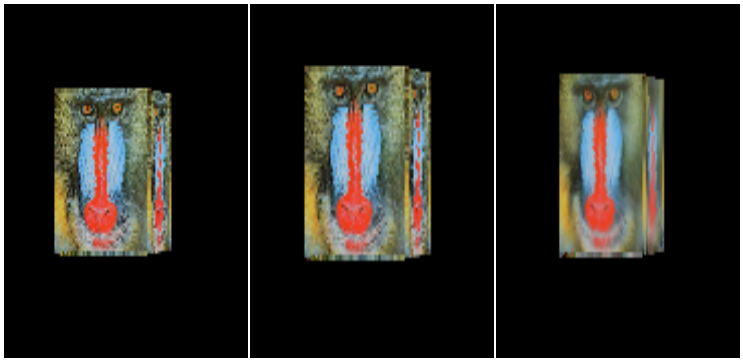


Figure 5: All lighting models required. Obs.: The cube might not have been the best model to show these off.

(a) Gourad (-Specularity) (b) Gourad (c) Phong

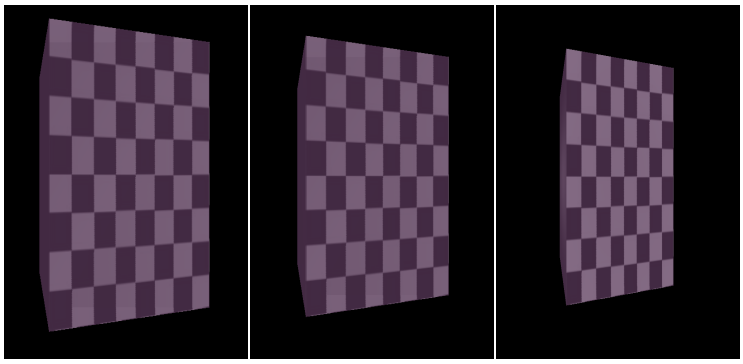
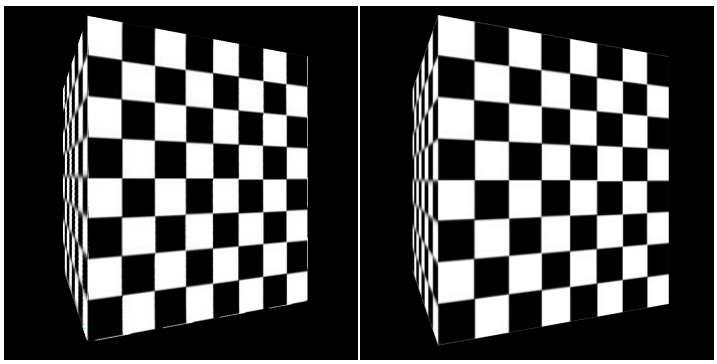


Figure 6: Close2GL mipmapping Vs. OpenGL's Mipmap result.

(a) Close2GL (b) OpenGL



5 Texture Modulate Shading

We add the texture process organically, meaning the texture color will replace the original object color from the previous assignments. However, this makes us keep track and perspective divide attributes such as the diffuse coefficient and the specular component for every vertex in the Gouraud model.

6 Conclusions

This was definitely the easiest project of them all. There was definitely a bit of a learning curve both on the mipmapping attribute definition and on the calculation of the Pyramid. I really liked the results. I've also liked how my result compares to **OpenGL**'s (section 6).

There are somethings I might need to fix regarding edge cases. I've also bumped the speed of the program to over 30FPS using Phong shading!