HomeWork 2

Rasterization

Fall 2012

Submitted By -

Nitin Agrahara Ravikumar

73714398

The First Part: Rasterization

As required by the homework, the display function of the smooth program is hijacked and a custom display method id used. In our case the method is called Rasterization ().
The steps involved in the custom display process are as follow.

- Obtain the vertices of each of the triangles one by one.
- Obtain the coordinates of the triangle by using the Bresenhams line algorithm.
- Update the line buffer with these values. (Line buffer is the size of the screen)
- Now, scan each pixel column and rows wise. For each row, start the scan from both ends so as to determine the start and end points of the scan in the triangle.
- Once obtained, fill all the pixels between those pixels to an arbitrary color.
- Do this for all triangles in the scene.

End results are shown in the snapshot.

The Second Part: Z Buffer and Shading

When we use gluProject to obtain the real world coordinates of the three vertices of a triangle, we also get the apparent depth values of the vertices. What we do for the z buffer calculation is that once we know that a pixel lies within a triangle (from the Rasterization part) we calculate is apparent i.e. Z` values using Barycentric interpolation. We store the values in alpha, beta and gamma. Now, for each pixel that is to be colored, we check the calculated depth value with that of the existing z' stored for that location in the Z buffer (it is also the size of the window.) If the current depth is less than that of the existing value in the z buffer, we overwrite it and replace the color for that pixel with the new way. This way we ensure we are showing only that which is in the front.

The result of the same can be viewed in the snapshots.

For the flat shading, we assume that the camera and light source are constant. Thus the bisector value remains constant as well. We use this bisector value, plus the normal to the surface of the current triangle. This is because the normal to every point on a triangle is same. Thus we get the normal per triangle, and calculate the shading of the triangle once. And whenever a pixel in a triangle is to be colored, we add the pre calculated shading value to it. And thus obtain flat shading. The result can be viewed in the snapshots.

The Third Part: Bonus - Gouroud Shading.

Since we calculated and stored the Barycentric values foe each of the triangles, according to the Gouroud shading technique, all we have to do is calculate the shading of each of the vertices of a triangle, and use the alpha, beta and gamma values to get the shading of the current pixel. We can obtain the normals of the vertices by accessing the data structure provided with the program. Use the same shading functions described earlier and interpolate the color and the check the z buffer as described before.
The results are shown in the snapshots.

Observations

Issues encountered:

- Improper pixel calculation in Bresenhams algorithm
- Improper start and end point of a triangle in a scan line.

Fixes:

Handling all cases while drawing a line (+ve / -ve slopes)

With the initial coding techniques employed, the Rasterization was extremely slow especially for meshes with large number of triangles.
Adding the flat shading and later Gouroud shading just worsened the situation.

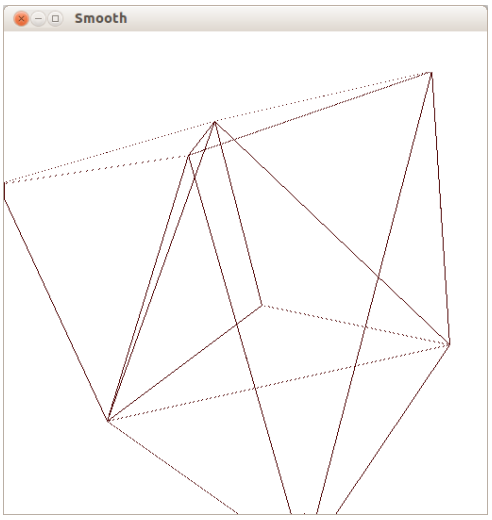
Several corrective measures were employed. Which are described as follows:

- Removing unnecessary for loop and replacing with memset
- Per triangle flat shading calculations instead of per pixel
- Integrating the bounding box strategy (the biggest improvement)

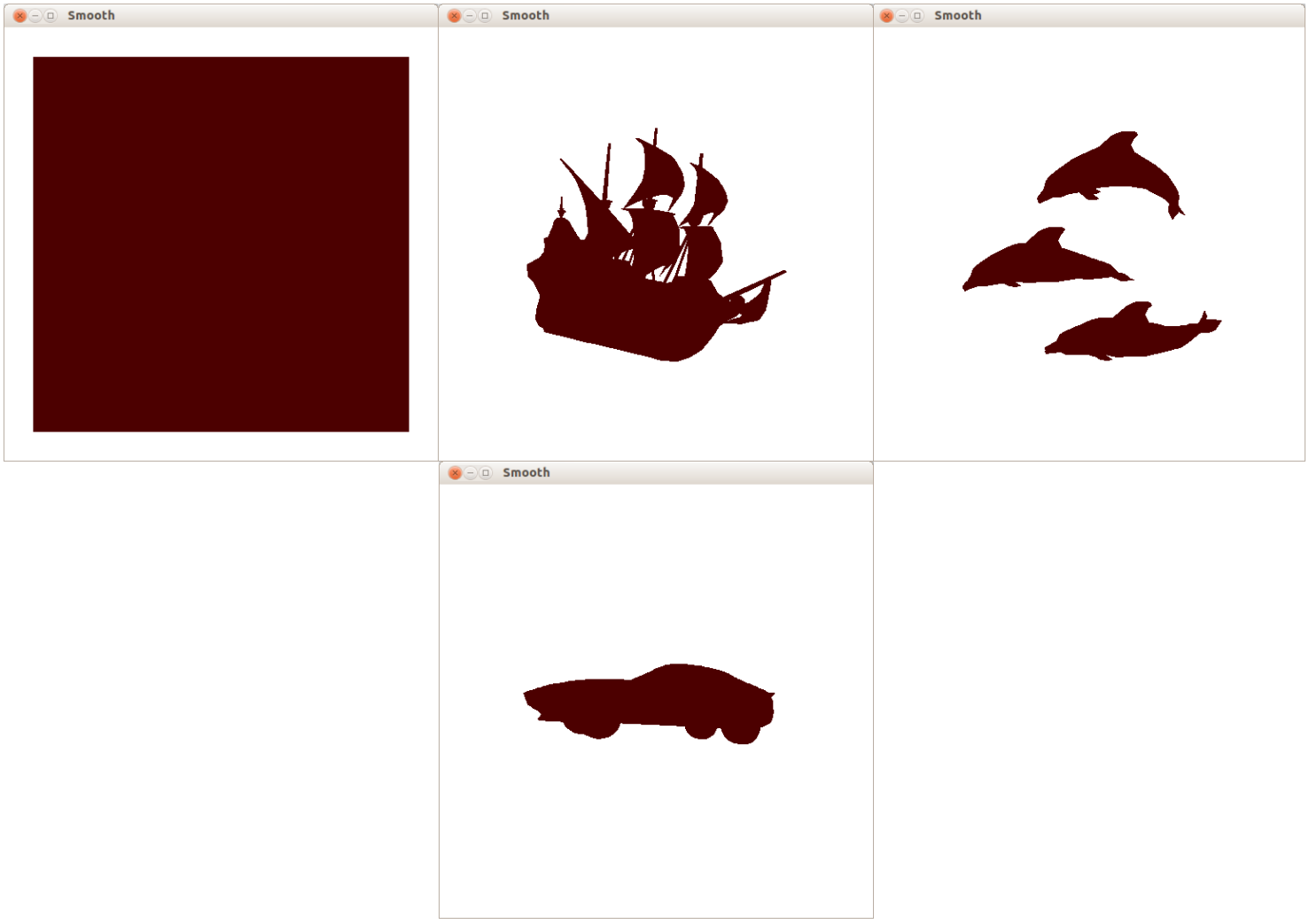
Still, even with these improvements, the pipeline coded is far less efficient than the once implemented by openGL. This is because of one simple fact. Open GL uses techniques programmed into the hardware. So essentially openGL is doing an operation in one cycle, which we are doing in several lines of code, which has to be interpreted and then executed.

The problem of color peaking at the edges in Gouroud is clearly seen as shown in the snapshots.

Snapshots

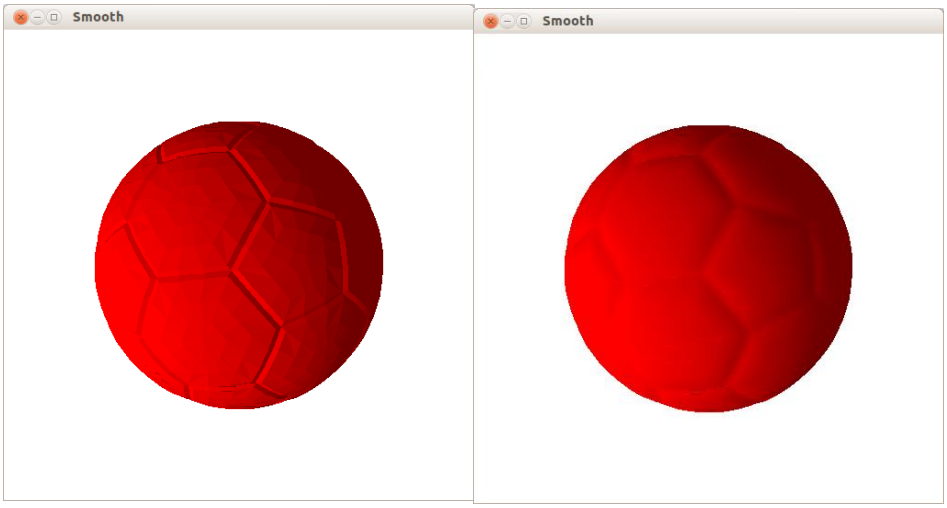


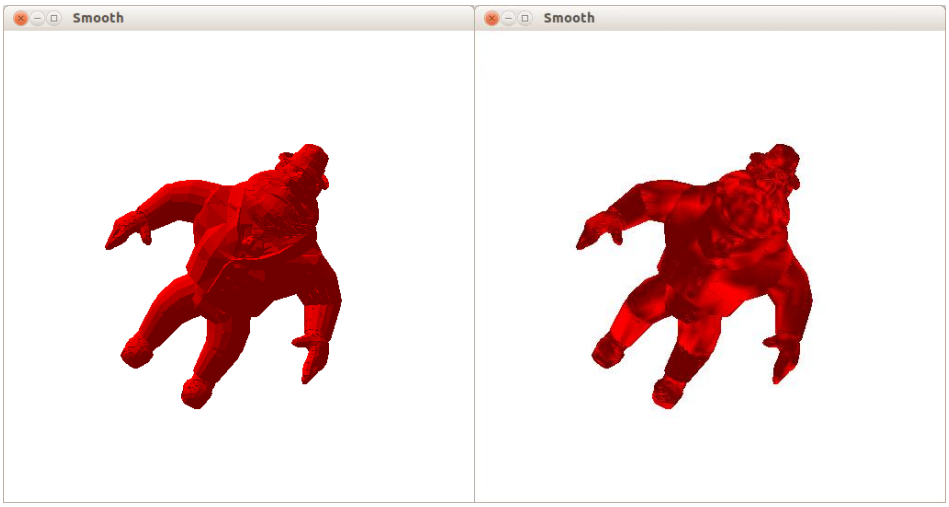
Only the triangles being rendered

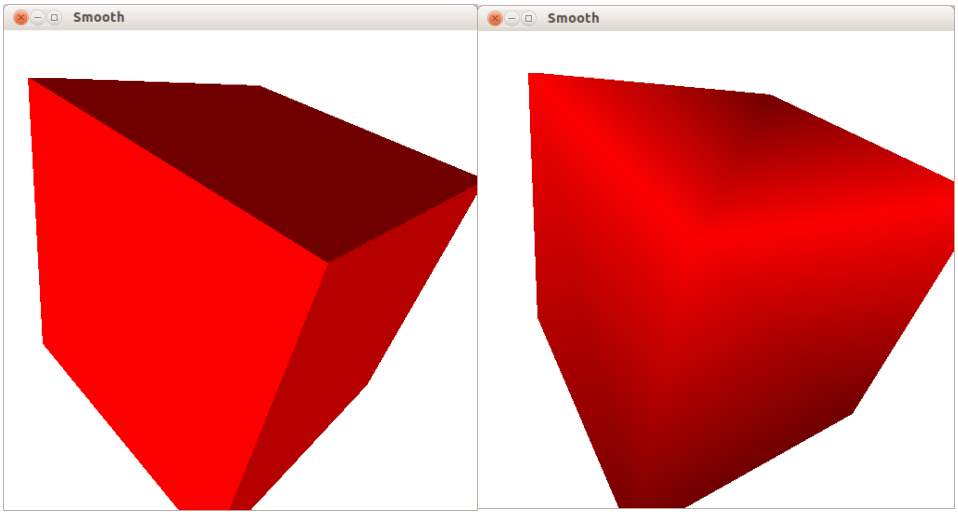


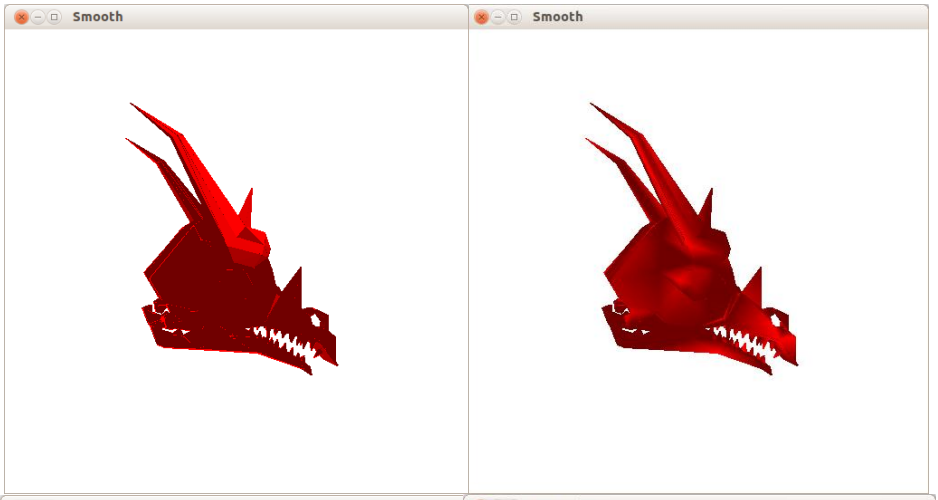
Snapshots of images rendered without using z buffer or shading

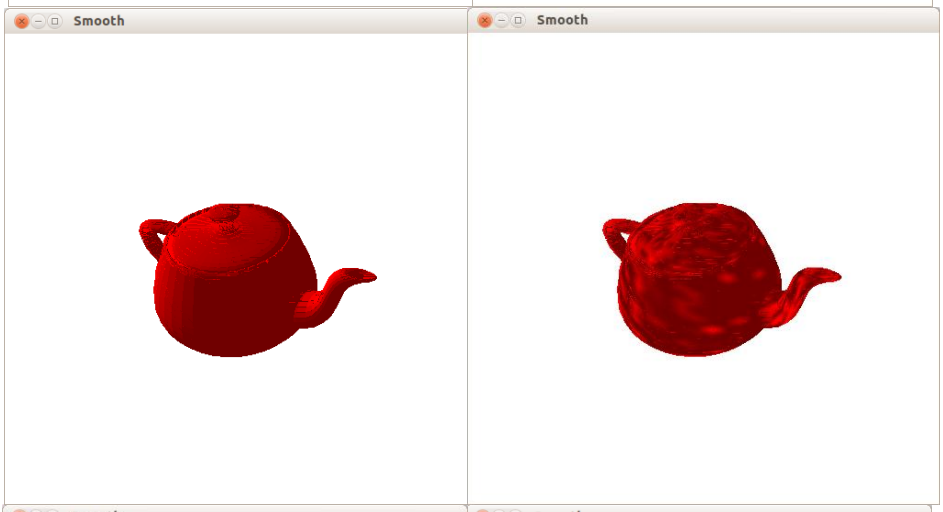
Multiple examples of Flat and Gouroud Shading

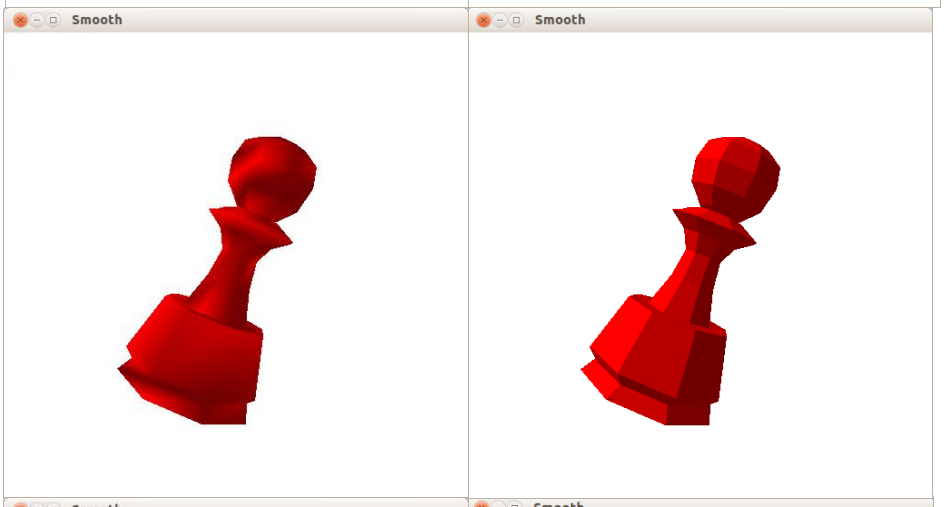


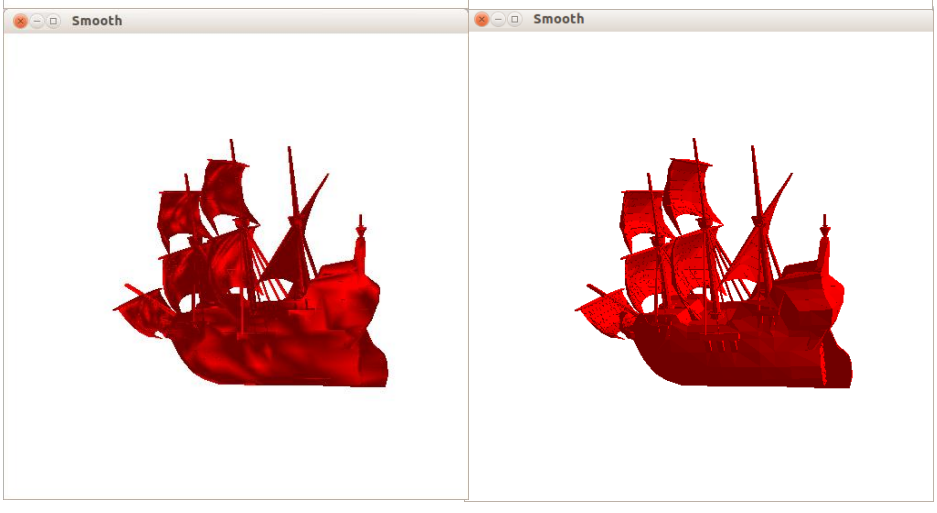





Smooth
Smooth

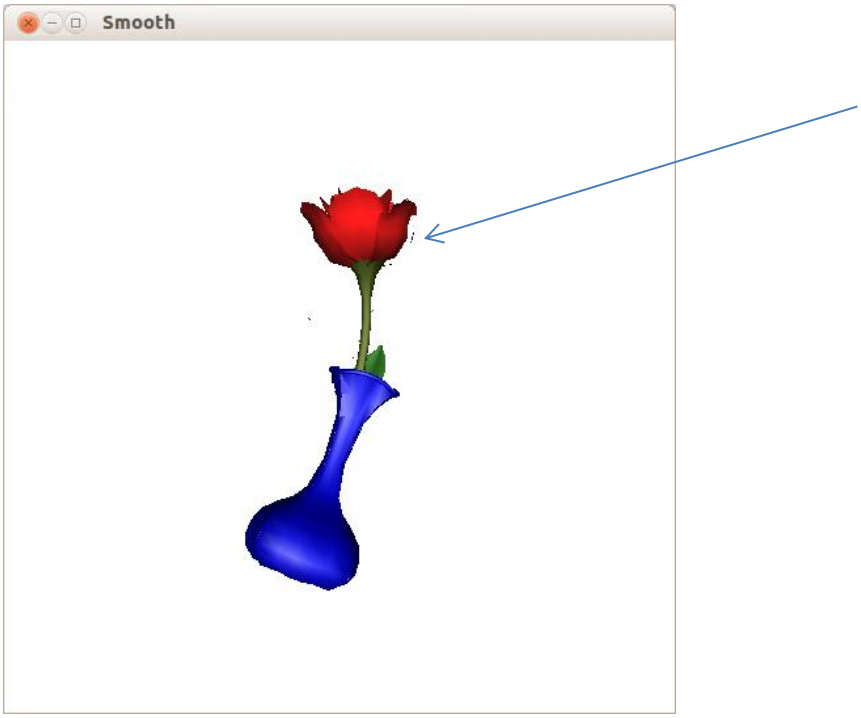

Smooth
Smooth

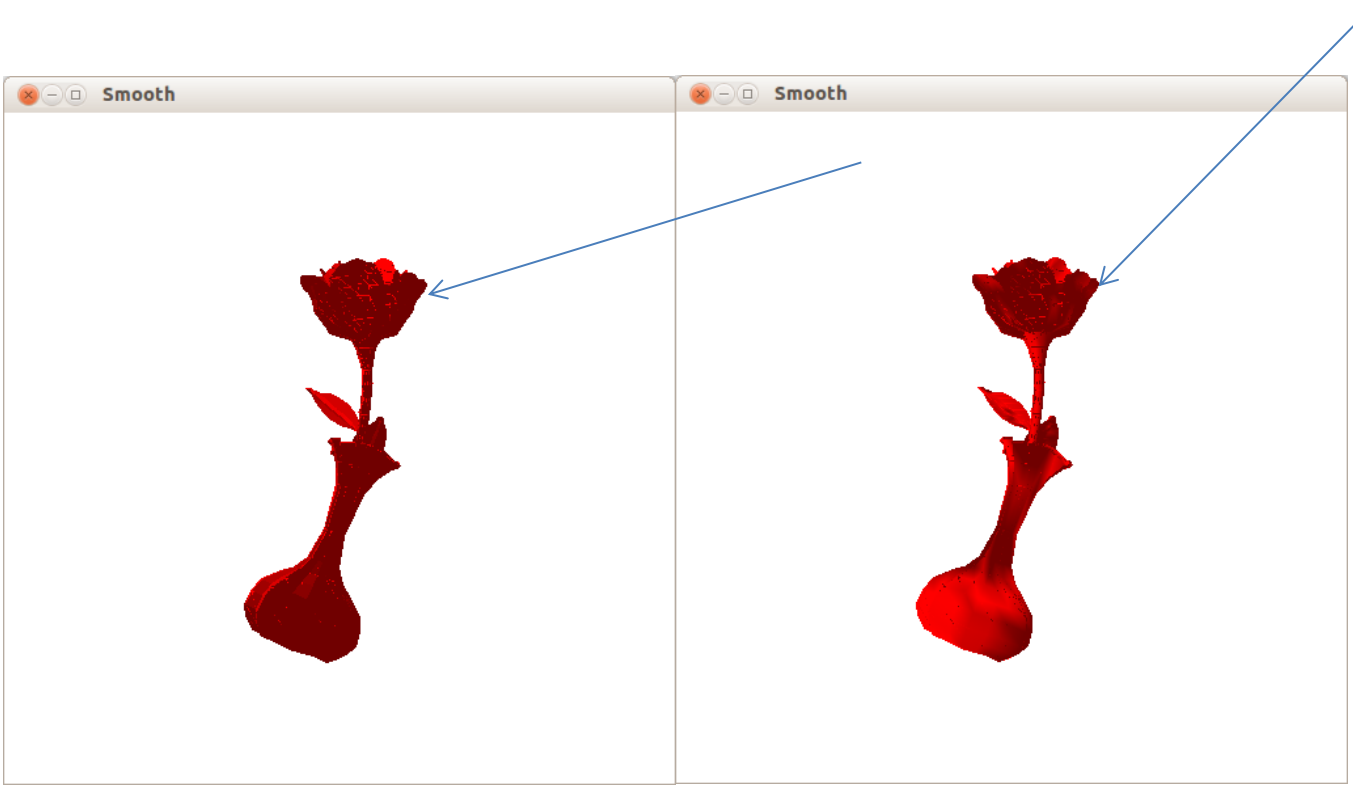

Smooth
Smooth


Smooth
Smooth

Abnormal behavior:

As shown in the images below, the pipeline employed by the openGL does not render one of the petals of the rose whereas the implemented by our Rasterization () function does. This petal is indicated below.





About the Submission

The code is kept in C and the filename remain unaltered. A makefile has been provided which creates the binary "**smooth**"

Two extra modes have been added:

'**y**' – To switch between original rendering and the rendering we have implemented.
'**f**' – To switch between Flat and Gouroud shading.

Acknowledgements

The homework is an extension of the original program "smooth" written by Nate Robbins and all the code used for reading the object files and corresponding data (vertices/normal/colors) is attributed to him.

http://user.xmission.com/~nate/smooth.html