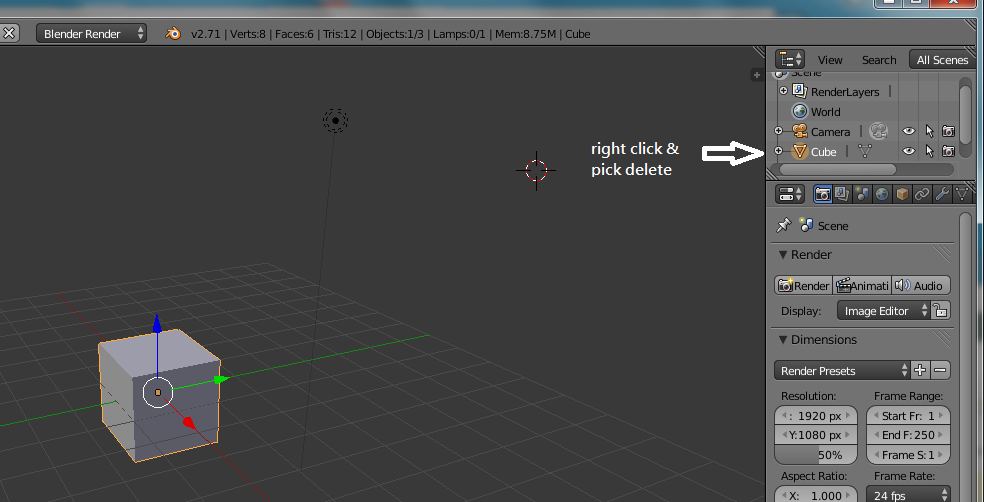
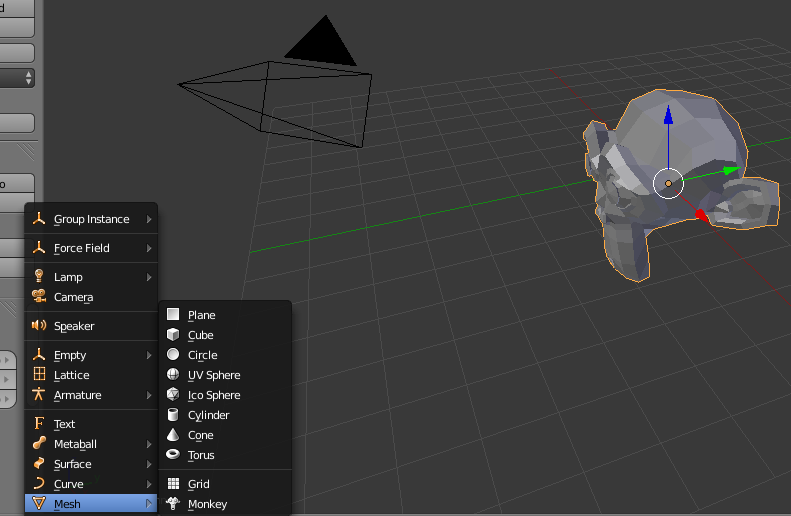
CS3321 3d Transforms

Due: 2/22/18 by midnight

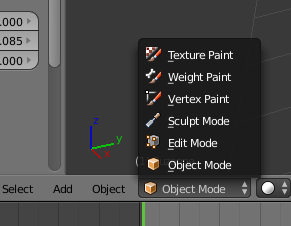
1. Use blender to generate a ply file with a simple 3d object. Choose one of their predefined meshes, but not the circle, plane, mesh, or cube.
2. From a terminal window run blender:

**prompt> blender**

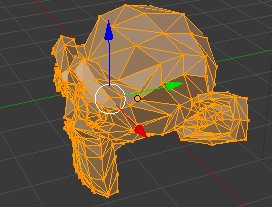
1. Delete the initial cube:

c. pick a mesh from the add menu: 

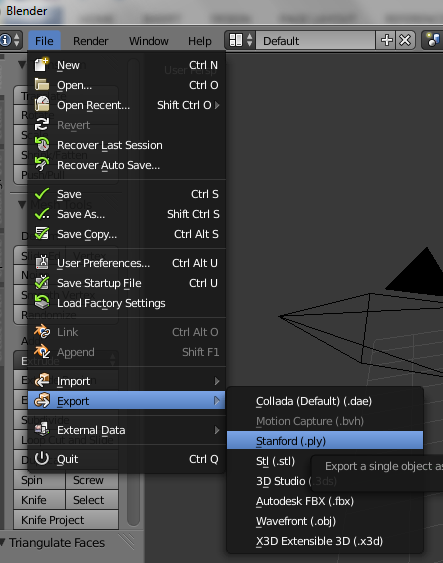
1. Change **Object Mode** to **Edit Mode**



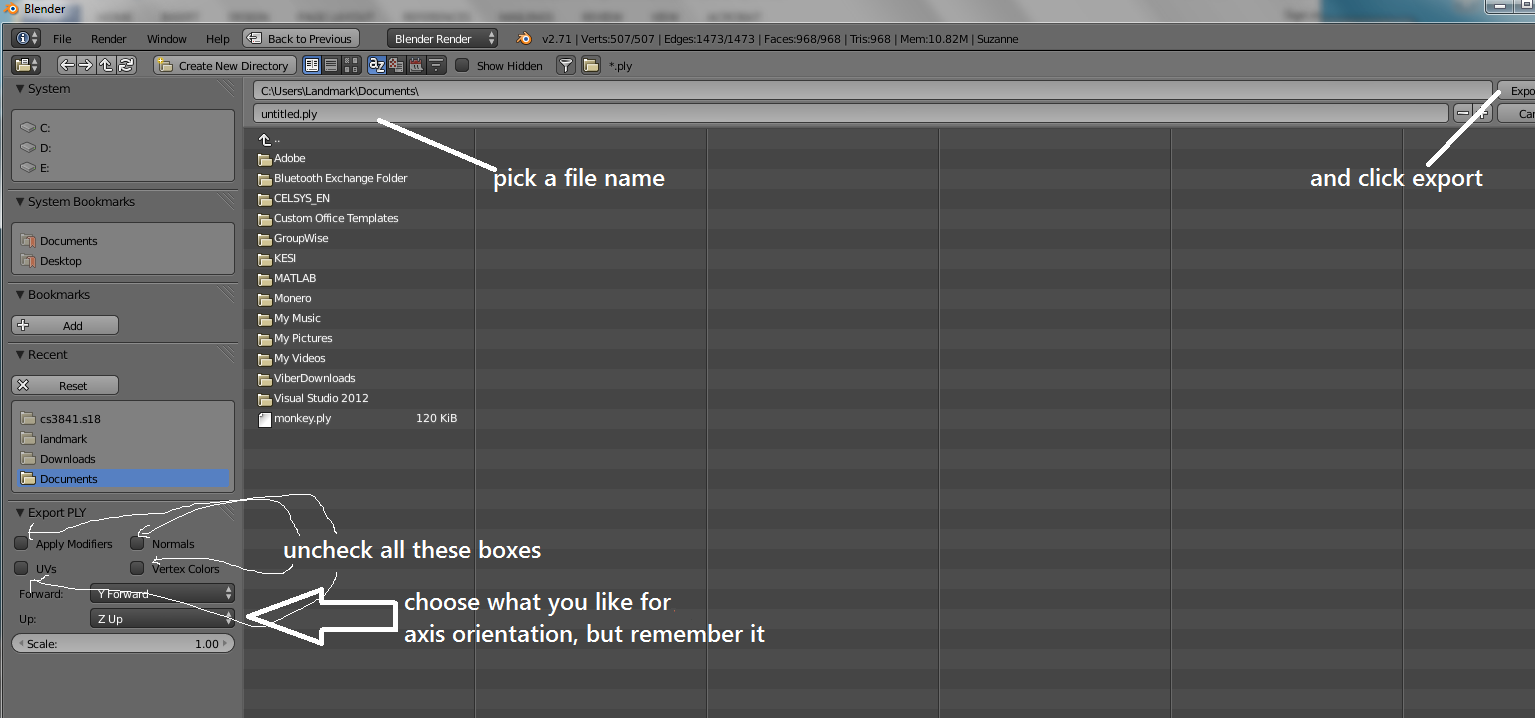
1. Hold the control-key down and press **t**. This will convert it to triangles (and why do we like triangles better?).



1. Export as a ply file:



f, cont..



You should now have a file which looks something like this:

ply

format ascii 1.0

comment Created by Blender 2.71 (sub 0) - www.blender.org, source file: ''

element vertex 507

property float x

property float y

property float z

element face 968

property list uchar uint vertex\_indices

end\_header

6.867248 -3.385576 3.420591

6.929748 -3.307451 3.350278

6.992248 -3.291826 3.498716

5.992248 -3.385576 3.420591

5.960998 -3.377764 3.498716 <<<<<<<<< 507 lines like this

5.867248 -3.291826 3.498716 <<< the x,y,z cords of each vertex

6.976623 -3.198076 3.311216

7.054748 -3.182451 3.498716

5.929748 -3.307451 3.350278

5.804748 -3.182451 3.498716

.

.

.

3 0 1 2

3 3 4 5

3 1 6 7 <<<<<<<<< 968 lines like this

3 8 5 9 <<< one line per triangle, shows

3 10 11 6 <<< which vertices to connect

3 12 8 13

.

.

.

Now write some code to read the ply file, apply matrix transforms, and display your object:

Create a 4x1 matrix class/struct to hold the vertices. The base data type should be **float**, and you must **normalize your data to fit in the unit cube**. For simplicity, you can use a fixed size array which is hardcoded to be the exact size.

Now read in the face data. A hardcoded array size is fine here as well.

Display the object. You have three dimensional data and you have to display it in two dimensions, so for now just drop the z coord (depth). So if for a given vertex you have

[x y z] = [0.13 0.82 0.05] display it using [x’ y’] = [0.13 0.82]. You will of course have to convert to screen coordinates to draw. By display, we mean draw the three lines

v1->v2, v2->v3, v3->v1 which make up the face.

Your program now sits in a read loop catching from the user (you choose the keys):

* one of three rotate commands, counterclockwise about each of the three axis. Each time through the loop, rotate 0.1 radians in the appropriate direction about the appropriate axis.
* one of six translate commands. Plus and minus for each coordinate axis yields six more options. You choose the amount of translation.
* one of two scale commands, uniform scale up by 10%, uniform scale down by 10%

Create 4x4 matrices (or just one if you like) to hold the transforms. After each keystroke, transform all the vertices using a matrix multiply and redisplay your object.

Always keep your data values between 0.0 -> 1.0 inclusive in all three dimensions. Convert to 2d and screen coords as you draw the lines but don’t change the actual data. Use homogeneous coordinates throughout.

This will be the final project using Java. We will use OpenGL going forward but it is not a small topic (esp. the programmable pipeline version), and I want to get the matrix game cemented in. Feel free to write this in C++/C if you like, but note that I will not go over OpenGL in class until next week. If you go this route, take a look at GLM (<**[glm/glm.hpp](https://glm.g-truc.net/0.9.2/api/a00052_source.html)**>) or some similar package.

