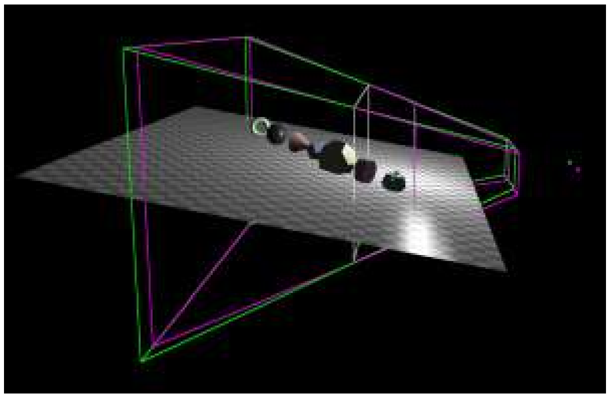
COMP 557 - Fall 2018 - Assignment 2
Perspective Projection Frustums, Depth of Field, and Anaglyphs

Available: 23:30 Wednesday 26 September

Due: 23:30 Wednesday 10 October

Getting Started



In this assignment you will use projection frustums to draw a scene with blur due to depth of field, and to draw anaglyphs that allow for 3D viewing with inexpensive colour filter glasses.

Provided Code

Download the sample code from MyCourses. It is a working program that will display a small test scene. You will need to add the jogl, vecmath, and mintools jars to your project. The provided code contains a main class, a simple scene class, and simple class for generating Possoin disk sample points within a unit disc.

The sample code has several drawing modes which you change by entering numbers. They are as follows:

1. The world view (default) where you can use the primary trackball to see the scene, the location of your frustums (as shown top right of this assignment).
2. The view from the eye that lies on the world z axis.
3. The blurred depth of field view from the eye that lies on the world z axis.
4. The view from the left eye.
5. The view from the right eye.
6. An anaglyph showing both left and right eyes in the same image.
7. The combination of a depth of field blur for each eye show as an anaglyph.

The first mode is the only mode that partialy exists in the sample code. For the other viewing modes a secondary trackball is enabled to allow you to rotate the scene.

You need to implement frustum visualization in mode 1. Likewise, you will need to implement the other viewing modes in the objectives listed below.

To help with testing your assignment at the different steps, you will be using *Parameters* for various values, such as the eye position, and the near and far planes (and many other values). The provided code uses classes from the mintools jar to create a window with controls for each of the parameters. When you call the getValue method of a given parameter, you get the current value which is set in the interface. In this assignment we use double, int, and boolean valued parameters.

You will use the following OpenGL calls to complete the assignment:

- ***glFrustum***

  You will need to set up several projection frustums by specifying the left right top and bottom positions of the near plane, and the position of near and far planes.

- ***glAccum***

  For depth of field, you will render the scene multiple times with slightly different viewing frustums, and blend the results. The accumulation buffer is a convenient mechanism for donig this. If you're blending N *images together, use glAccum(GL.GL_LOAD, 1f/N) to load the first image into the accumulation buffer, and then* *gl.glAccum(GL.GL_ACCUM, 1f/N) to add the contribution of the other renders. Remember to use glClear before drawing each image, and use* gl.glAccum(GL.GL_RETURN, 1) to copy the result back into the frame buffer once you're finished all rendering passes.

If you are using Macintosh, you'll need to use the provided Java implementation of the accumulation buffer. Create an instance of the provided Accum class, and use methods glAccumLoadZero, glAccum, and glAccumReturn.

- ***glColorMask***

  When drawing an anaglyph, you will want to draw one image for the left eye, and a different image for the right eye. With *glColorMask* you can enable or disable writing to different components of the frame buffer. There are four components: red, green, blue, and alpha. The alpha channel is used for transparency and should always be enabled. Thus, to draw only the red channel, call *gl.glColorMask(true, false, false, true)* before rendering the scene. To draw an image into the green and blue chanells (cyan), then call *gl.glColorMask(false, true, true, true)*, clear the buffer with *glClear*, and draw the scene again with the projection matrix for the other eye. Different glasses will have different filters so you'll have to choose your colour masks according to those that you're using (e.g., red/blue, red/cyan, or green/magenta, where magenta is blue and red combined, and cyan is blue and green combined).

  Note that you will want to clear the depth and colour buffers before drawing with *glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)*. Also note that your colour mask call sets the opengl state, which affects all calls, such as glClear!

- ***glGetFloatv***

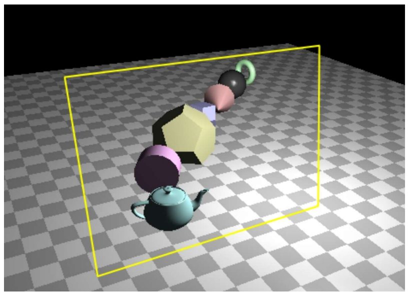
  You need to draw frustums in this assignment, and since you will likely find it easier to use *glFrustum* to set up the projection matrix, you will also want to use *glGetFloatv* (with either *GL_MODELVIEW_MATRIX* or *GL_PROJECTION_MATRIX*) to get a copy of this matrix in a Java array. You can use the mintools *FlatMatrix4f* (or *FlatMatrix4d* if you use *glGetFloatd*) to help with converting OpenGL matrices to vecmath matrices. Note the asArray() and reconstitute() methods of the flat matrix classes. You will use the inverse of your projection matrix, combined with *glutWireCube(2)* to draw the frustum.

Steps and Objectives

In each step, you should be able to visually verify that your implementation is working correctly. Note that you will probably want to spend some time working with pen and paper to decide exactly what needs to be computed for each step before you actually start writing code!

1. ***Draw Screen Window Rectangle and Eye Point (1 mark)***

   Draw a yellow rectangle in the xy plane centered at the world origin in immediate mode using glBegin(GL_LINE_LOOP), providing the 4 points with glVertex, and finishing with glEnd(). The x and y dimension should be equal to the width and height of the display window in meters (that is, the physical dimension of this window as it appears on the screen you are currently using). Getting the screen dimensions approximately correct is important for correct 3D viewing, so you'll want to measure the width of your screen with a ruler and set the variable *metersPerPixel* appropriately (see the commented code).

   Draw a white sphere at the position of the eye using glutSolidSphere.
   Note that with lighting disabled, it should just look like a white circle, i.e., no shading. The radius should be 0.0125 meters to approximate the size of the human eye. Call *getValue* on the eyeZPosition member variable to find out where to draw the eye in the world coordinate system.

2. ***Draw Frustum (2 marks)***

   In this step, you will want to create a projection matrix using glFrustum that will be needed to project points onto the screen. The *near* and *far* parameters should be used to compute the near and far plane positions. Notice that these near and far plane position parameters are specified with respect to the world origin! So you will need to add the eye position to compute the value to pass to glFrustum. The motivation for defining these parameters in world space is that they do not need to change when you move the eye. That is, when in mode 1, world viewing mode, you can easily change the eye distance while leaving the near and far planes fixed with respect to the screen rectangle, and the objects of your scene can thus remain between the two clipping planes.

Note that since you must provide left, right, top and bottom of the near plane, you will need to compute appropriate values to take into account the position of the near plane with respect to the screen rectangle (i.e., the screen rectangle should exactly fit inside your frustum).

Use the technique described in the assignment introduction to draw a white wire frustum when *drawCenterEyeFrustum* is true. For instance, you might first do a glPushMatrix, load the identity, set up your frustum, then extract it with getDoublev, and then pop your changes off the stack; then, invert your matrix, and put the inverse on the stack along with any necessary translation to accommodate the eye position, and draw a wire cube size 2 to show the region that will be projected to normalized device coordinates.

You should also now implement viewing mode 2 which shows the result of using this frustum. The easiest thing to do is overwrite the opengl projection and modelview matrices to be your projection and viewing transformation (i.e., the translation to the eye position). Use glMatrixMode to change between modifing the projection matrix and the modelview matrix.

3. ***Focal Plane Rectangle (1 mark)***

   Draw a grey rectangle to represent the focal plane. The focal plane is in the xy plane, and at a displacement in z from the origin specified by the *focalPlaneZPosition* parameter. You need to compute the left, right, top, and bottom coordinates of the focal plane so that it exactly fits inside your frustum. Try moving your focal plane to different distances. Note that you should be able to place it beyond your far plane, or in front of your near plane.

4. ***Eye Position Offset (2 marks)***

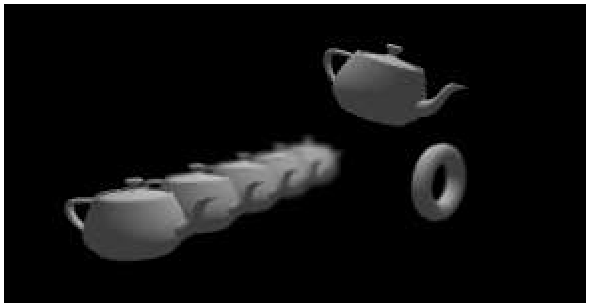
   In this step, you will use the x and y offset parameters *eyeXOffset* and *eyeYOffset* to create a frustum at an eye position that is displaced from its position on the z axis. The idea is to explore the set of frustums that keep the focal plane rectangle fixed while using different eye positions.

   In the next step, you will average together images created from many nearby eye positions while keeping the focal plane fixed in order to create an image with a depth of field effect. However, the purpose of this step is to make sure that your frustum is correct before proceeding.

   Adapt your eye and frustum drawing code from previous steps to account for the displaced eye position. Compute left, right, top, and bottom values to pass to glFrustum to create the correct frustum which always contains the focal plane. Test your code by moving the focal plane and the eye offsets, and convince yourself that it is doing the right thing!

5. ***Depth of Field Blur (1 mark)***

   Implement viewing mode 3 by averaging multiple renders of the scene with the accumulation buffer. Use the *samples* parameter to get an integer number of samples, and use the provided FastPoissonDisk class to obtain different samples (which you must then scale appropriately for the current *apertureSize*). Note that you could ultimately use diffrent sets of samples to create different bokeh effects.

   Additional reading can be found in the paper SIGGRAPH 1990 paper The Accumulation Buffer: Hardware Support for High-Quality Rendering by Haeberli and Akeley.

6. ***Draw Left and Right Eyes (1 mark)***

   This step can be done independently of steps 3, 4, and 5. For this objective you must create frustums for left and right eyes. You should display both eye positions and their frustums in the same way as in step 2, but only if *drawEyeFrustums* is true.

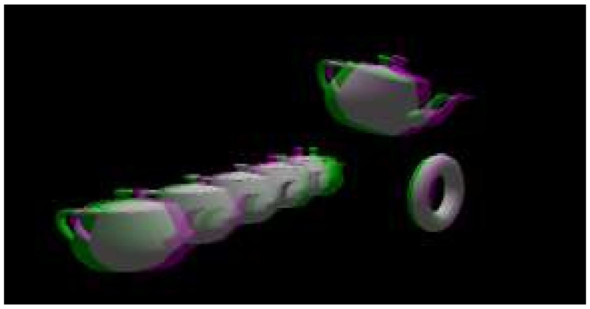
   The frustums should be colour coded according to the glasses that you are using, but you probably have red-cyan glasses, in which case your left eye is red, and you should draw the left eye frustum and left eye position in red. Note that you should glDisable(GL_LIGHTING) before drawing in solid colours, and glEnable(GL_LIGHTING) when you are done.

   The frustums should both contain the screen rectangle, thus you'll need to compute correct left, right, bottom, and top values for each. The *eyeDisparity* parameter should be used to set the distance between the eyes (default of 63 mm), so the left eye should be half this distance in the negative x direction, while the right eye should be half this distance in the positive x direction (that is, the center point between the eyes lies on the z axis of the world and the centre eye). Both eyes are viewing in the negative world z direction (that is, they are NOT looking at the origin of the world).

Note that if we were able to track the eyes, then we might want to use the known position of each eye in setting up the frustum! However, we will just assume that the viewer is sitting exactly in front of the window on screen at a distance from the screen specified by *eyeZPosition*.

Test that your frustums are correct by adjusting the near plane, and by viewing the resulting images with viewing mode 4 and 5.

7. ***Anaglyph viewing (1 mark)***



   For mode 6, combine the images you render for mode 4 and 5 by using the appropriate glColorMask calls. Test your image using colour glasses. You should be able to make objects appear both behind and in front of the screen. Be sure you do not have your left and right eyes reversed! The *eyeDisparity* parameter value should always be a positive number. Note that it may be hard to fuse the anaglyphs when using the proper values because the colour glasses do not completely block the view of the opposite eye. You may find it easier to fuse the anaglyphs and have a better 3D effect by using an artificially reduced disparity.

8. ***Anaglyph viewing with Defocus (1 mark)***

   Combine your efforts in the previous step to create anaglyphs and side-by-side pairs where the left and right images have blur due to depth of field. The trick is to define your frustums correctly. The offsets you use to create the blurred image need to keep the focal plane rectangle fixed, but this focal plane rectangle is different for each eye as it is defined by the eye position and the screen rectangle position!

Finished?

Great! Be sure your name and student number is in the window title, in your readme.txt (should you have additional information to submit), and in the top comments section of each of your source files.

Submit your source code as a zip file via MyCourses. Be sure to check that your submission is correct by downloading your submission from MyCourses. You cannot receive any marks for assignments with missing or corrupt files!

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own.