

1 Introduction

As usual, zip the project directories into one zip file. Upload the zip file to the Homework #7 page of the MyClasses site for this class. These exercises are extensions of the texture and lighting example.

2 Exercises #1: Picture Cube

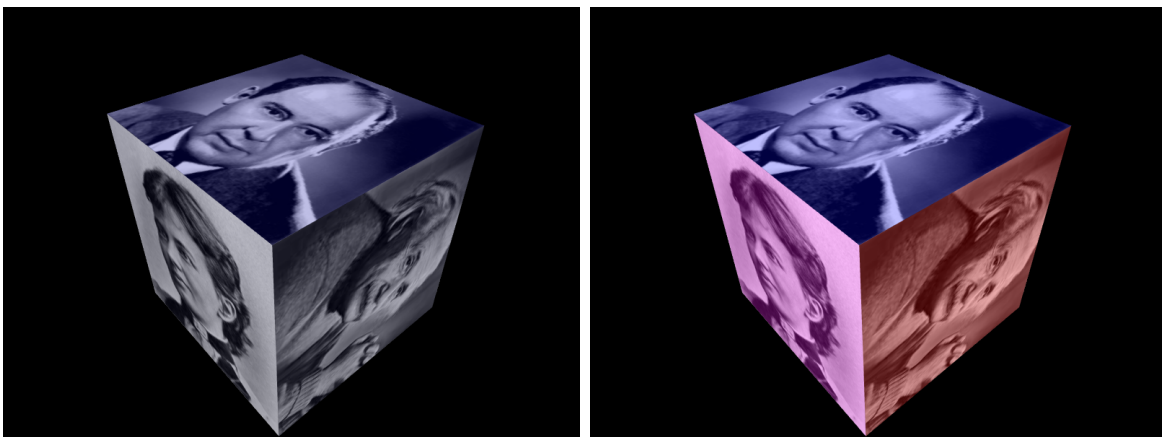
The first exercise was inspired by the old acrylic picture cubes that came out in the 1970s. Create a program that will display a single picture cube with 6 famous faces on each face of the cube. The image textures are in the included zip file.

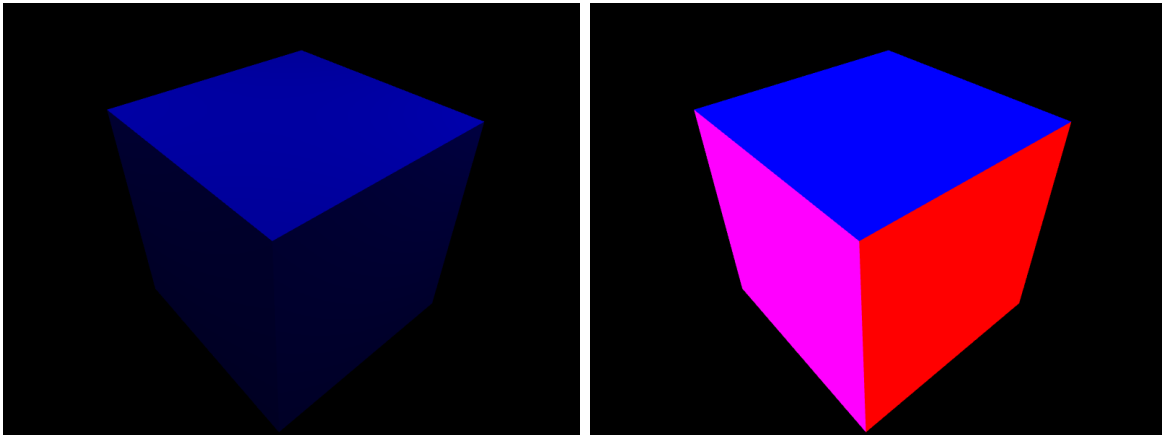


The program should have the same 3-D user interface as we have been working with, support for spherical and YPR cameras, keyboard and mouse navigation, etc. In addition, implement two more options.

- T: toggles the textures on and off.
- Y: toggles the lights on and off. That is, the use of lighting, not just the graphing of the light spheres.

The top two images the textures are on, the image in the top left the lights are on (with a blue plastic material) and the top right the lights are off. For the bottom two images the textures are off bottom left the lights are on and the bottom right the lights are off.





For this, update the cube object but rename it `PictureCube`. Remove the outline mode from the cube, so you will only need one EBO. The `PictureCube` constructor should take in 6 texture image IDs. Note that it should not be doing any image loading, that is to be done in the graphics engine. The graphics engine is to take care of texture activation as well. You will use one active texture and change the image using the binding of the different texture ids. The binding changes are to be done in the draw function of the `PictureCube` object.

A couple notes here about the code. First, while looking up some other things I happened across a faster way to load in our texture images. The bottleneck in the process is the numpy conversion to an array, the rest is fairly quick.

In our loading code we have the line

```
img_data = np.array(list(teximg.getdata()), np.int8)
```

which is very slow. Replace it with the line

```
img_data = np.asarray(teximg)
```

and the conversion is much faster. With only a few images this is not a problem but if you were loading more images there is a definite lag-time for the start of the program.

Also, when we draw the cube with an indexing array we usually use the command, which starts at the beginning of the data and draws triangles using the 36 indicies in the array.

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, None)
```

For this exercise you will want to draw each side with a different image, so you will bind the image and draw the corresponding side. To draw just a portion of the cube you can replace `None` with a pointer to the starting position of the data, and of course the second parameter is the number of vertices to draw. So if you have defined

```
self.ustrsize = ctypes.sizeof(ctypes.c_uint)
```

in the constructor then the command,

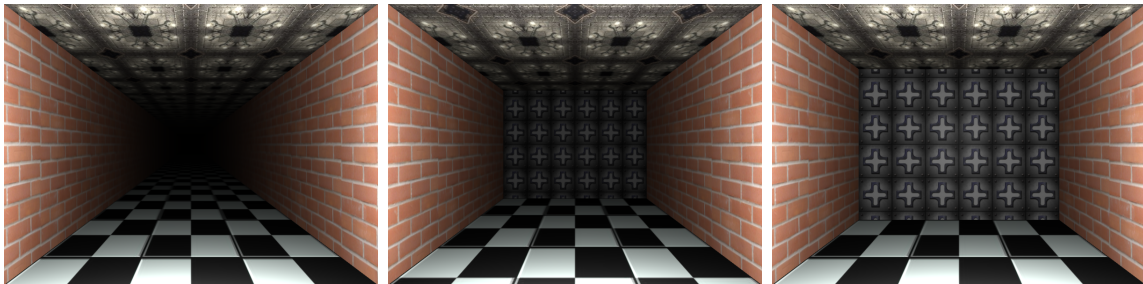
```
glDrawElements(GL_TRIANGLES, n, GL_UNSIGNED_INT, ctypes.c_void_p(
    self.ustrsize * s))
```

will draw triangles starting at the s index in the vertex array and use n vertices in the draw.

3 Exercises #2: Spooky

This exercise was inspired by a final project of one of my students several years ago. He created a maze game where you were in a hallway (dungeon like but simple and sparse) and you could only see a few feet in front of you. There were a lot of dead-end hallways and you had to navigate the maze by remembering the turns you made.

This program is a hallway with different textures on the walls, floor, and ceiling. The only user interface, besides the usual screen shot and mode change, is the up and down arrow keys. The up key moves you forward and the down key backward. Make sure that you stop before you hit the end of the hallway but can see the dead end. When moving backward make sure that you stay inside the hall and not move outside the model. There should be only one light source which is at the position of the camera, to simulate the effect of a candle or lantern being carried by the person. You will probably need to alter the calculations in the shader since the way we wrote the shader, attenuation is applied to the lighting only, and you will need it applied to the textures as well.



4 Challenge Exercise: Optional for Extra Credit

One question that came up in class was the limitations on texture loading, active textures, and texture ID generation. While under most circumstances we will not run into any of the hardware limitations for textures, there are of course limitations. These limitations are primarily driven by the limited amount of texture memory on your graphics card, while substantial still finite. There are also limitations on the size of a texture (in pixels), again substantial but finite.

When these limitations are hit we can enlist the CPU to offload the storage requirements as long as we can send image data from machine RAM to the graphics card quickly enough as to not slow the graphics processor below our 60 FPS mark. The computer memory is far larger and can hold more data, the trick is the continual transfer of images and the bus speed to the card.

For this exercise we will be playing a video on our various objects, like the plane, sphere,

torus, ..., and of course the teapot. OpenGL does not have facilities for playing a video as a texture so we will need to animate the textures by changing the image of the texture while the program is running. Included in the texture image zip file is a directory containing 3014 jpg images that make up a 2.5 minute trailer for the movie The Matrix, at 20 frames per second. The resolution was reduced so that the image sizes are small and hence can be transferred to graphics card memory quickly. If you want to do this yourself on another video you can use a number of software packages, I used ffmpeg, a commandline tool that is cross platform.

Using our conventional method of putting each texture with its own texture ID and loading the texture to the graphics card failed at around 2200 images. This was on the laptop I use at home which has an Intel Corporation HD Graphics 620 card.

Here is how we are going to share the load between the CPU and the GPU. We will go into more detail in a moment but the general idea is to load all the images from the hard drive, convert them to arrays with numpy, and store them in a list on the CPU side. Then upload each image while the program is running at 20 FPS, although our rendering framerate will be 60 FPS or higher. As usual, the graphics engine will do all the work. Have the video play on all of our standard objects we have been using in the examples, the array of cubes, single cube, sphere, torus, trefoil, tessellated plane, height map, teapot, and the simple plane. None of the objects will need to be altered.

Here are some details that may help.

- In the constructor of the graphics engine, load in all of the images from the trailer and store them in a list. Specifically, for each image, load the image, convert to RGBA, flip, and convert to an array with `img_data = np.asarray(teximg)`. Then create a list of the converted image along with the image height and width, finally append this list to a list of images in Python. Even with the increased speed of the asarray conversion this will still take around 15–30 seconds. You may want to have a counter printing out to the console so you know how far the process has progressed.
- Generate one texture ID for the video and make sure it is active.
- Load the ID to the texture sampler2D in the shader.
- In the update function in the graphics engine, link a counter to the Python timer so that the counter increments at 20 FPS no matter what the rendering framerate is. If the counter increases, extract the next image information from the list you created. Recall that this is a list of the image array, the height and the width. Now do the remainder of the image loading. Bind the texture to the video texture ID you created, use `glTexImage2D` to load the image to the card, mipmap and set the texture parameters. In the process above make sure it only happens at 20 FPS and not on each frame. If you set this up to load on each frame you will be doing 3 or more times the amount of work you need to do.
- As this is running you may notice that the video seems pixelated. This is due to the resolution reduction in the creation of the images, not your min and mag filters.

