Creating a Skybox in Python

Justin Schirle

Abstract

Implementing a skybox into a program can be useful for adding an aesthetic appeal by creating a 3D enviroment which gives the illusion of infinite space. It can be done fairly simply in OpenGL by creating a cube which is unaffected by depth and translations, and by applying textures to each face of the cube.

1 Loading Textures

Before rendering the skybox, one must load the textures which will be later used. For this, and throughout, I will assume that you have the PIL Imaging library in addition to the OpenGL libraries. The first step is importing these libraries:

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys
from PIL import Image
```

We now begin loading the images for our textures, and then binding them. This should all be done during the start of the program, so that they are only loaded once. Otherwise the program will load the images for each frame which would make the program run extremely slowly. For a typical PySkel SZG program, this is done in onWindowStartGL. The portion run in the start will be a function `loadskybox()`.

```
def loadskybox():
```

All the commands will be within this function until otherwise noted. First we load our images.

```
im1 = Image.open("file.jpg").rotate(180).transpose(Image.FLIP_LEFT_RIGHT).resize((512,512))
texture1 = im1.tostring()
```

We use the command `Image.open` from the PIL library to open and read the file. Be sure to specify the correct path to your file. The rotate and transpose

are used here to correct the orientation of the image. If we were to simply load the image and then redisplay it, it would appear upside-down. We send the data to a string - a usable format - using `im.tostring()`. This process should be repeated for each of the 6 images that will be used for the cube faces. For the back and left faces, you should not use `transpose(Image.FLIP_LEFT_RIGHT)`, because our view of them from inside the cube flips them naturally. This will make more sense when you create your box. Finally, we resize the image to 512x512 pixels. These dimensions can be changed, however later it will be necessary for them to be a power of 2 (256x256, 512x512, 1024x1024,...) If a high resolution is used, it may not run on some graphics cards. This can be easily fixed by lowering the resolution.

2 Creating Textures

Now that the images are loaded, we must bind them to textures so that they can be used later. We begin by generating a texture:

```
glGenTextures(1)
```

The 1 means that it will generate one texture. The first time `glGenTextures(1)` is used, the output will be 1. Each time after that, it will output an integer which is one greater than the previous time. So the second time it will output a value of 2. It should be noted that if `glGenTextures` is used in a Python shell, it will output a large integer. This is not how it behaves if it is used in code. Next we bind the textures:

```
glBindTexture(GL_TEXTURE_2D, 1)
```

This binds a 2D texture to 1, the texture we generated above. For the second texture, you will use `glBindTexture(GL_TEXTURE_2D, 2)` and so on. Next we define the parameters of the texture:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE,
texture1)
```

The `glPixelStorei` sets how the pixels will be stored for the 2D texture. The `GL_CLAMP` means that the textures will be clamped at the edges on both the sides and top/bottom. (`GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T`), as opposed to having the texture repeated to fill the space, which would be accomplished with `GL_REPEAT`. I do not really understand entirely what the final

two parameters do. One could also use GL_LINEAR instead of GL_NEAREST, but I am not sure of the difference. Finally, the nine arguments of the glTexImage2D. The first is the target texture and should be left as GL_TEXTURE_2D. The second is the detail level, and should be left at 0. The third is the color type of the texture, in this case just GL_RGB. The fourth and fifth are the width and height. These need to be a power of 2, and should be the same size as your resized images from before. The sixth argument is the border type, which must be either 0 or 1. The seventh is the pixel format of the images loaded as textures, usually GL_RGB. The eighth is the data type of the image, which should be GL_UNSIGNED_BYTE. The final part of the glTexImage2D is a pointer to your image data. This is the string of image data from before, which in this case is texture1. Once you have done this for your first texture, repeat it for the other 5 textures of the skybox. The only thing that needs to be changed each time is the number in glBindTexture, and the last entry of glTexImage2D.

3 Rendering

Once the textures have been loaded, we are able to draw our skybox. We begin by defining a new function:

```
def drawskybox():
```

Everything following is within this function. This should be called to draw the skybox, and it is important that this is done before all other objects are drawn. This is so that the skybox is in the background and does not cover other things up. For a typical PySkel SZG program, this should be called in onDraw. First, certain parameters must be changed to use the textures:

```
glEnable(GL_TEXTURE_2D)
glDisable(GL_DEPTH_TEST)
```

This enables 2D textures, and turns off the depth test. This means the skybox will be unaffected by translation (but rotation still affects it). To render the skybox, we will draw 6 quadrilaterals.

```
glColor3f(1,1,1) # front face
glBindTexture(GL_TEXTURE_2D, 1)
glBegin(GL_QUADS)
glTexCoord2f(0, 0)
glVertex3f(-10.0, -10.0, -10.0)
glTexCoord2f(1, 0)
glVertex3f(10.0, -10.0, -10.0)
glTexCoord2f(1, 1)
glVertex3f(10.0, 10.0, -10.0)
glTexCoord2f(0, 1)
```

```
glVertex3f(-10.0, 10.0, -10.0)
glEnd()
```

We first define the color to be white. This the textures are drawn using all colors. Before drawing the quadrilaterals, we must bind the texture we want to use. In this case, we are using texture 1. `glBindTexture()` cannot be used between `glBegin()` and `glEnd()`. We then establish the coordinates of the texture using `glTexCoord2f` and the vertices. The vertices might need to be changed to fit your needs. For subsequent faces, simply bind a different texture and change the vertices. The same texture coordinates are used each time. After each of the faces has been drawn, do the following:

```
glBindTexture(GL_TEXTURE_2D, 0)
glEnable(GL_DEPTH_TEST)
```

The 0 texture is the null texture. If we do not bind it, all subsequent objects will be rendering with colors of the last texture used, and they will not look correct. We then enable the depth test and continue with the program.