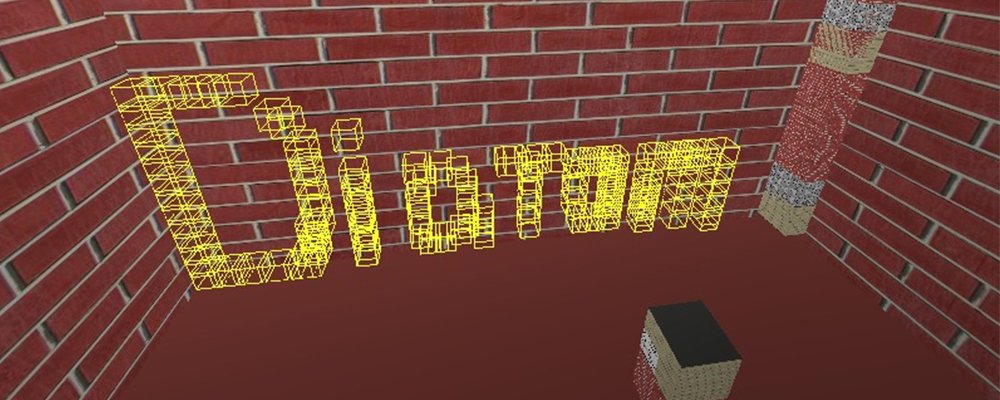
**Different sides of Ruby development: OpenGL**

*OpenGL is common everywhere and learning it once, you will be able to build projects with it on any language. The same is with Ruby.*

***Alexander Kovtunov***

When I’m trying to take a break from web development, I like to build something new and uncommon in Ruby programming. Ruby is typically known for web services, APIs and other web-related development. But with some time, you can build many other types of software.



After creating, I started researching Ruby applications that use 3D graphics. I found a nice library, [OpenGL for Ruby](https://github.com/larskanis/opengl), and created a few samples with it. At first, it can be difficult to understand everything, but let’s see what can we do!

To start working with OpenGL for Ruby, we only need two gems:

*Gemfile.rb:*

source "https://rubygems.org"

gem "gosu"

gem "opengl"

I chose [gosu](https://github.com/tianon/gosu) because it provides good functionality to work with windows, players and partial rendering of window objects.

**LET’S CREATE SOMETHING SIMPLE!**

Sometimes it’s hard to imagine how everything should look in 3D, so for the first step, I chose to draw an axis instead of a simple cube.

This is a good way to see how our coordinates lie on the three-dimensional grid. Also, we will set the colour of the lines to define where the x, y and z coordinates are.

I suggest that you separate objects into different classes that will stay under the *objects/* directory. So *media/axis.rb* will contain:

**class** Axis

attr\_reader :x\_width, :y\_width, :z\_width

**def** initialize(x\_width, y\_width, z\_width)

@x\_width = x\_width

@y\_width = y\_width

@z\_width = z\_width

**end**

**def** self.draw(x\_width, y\_width, z\_width)

object = **new**(x\_width, y\_width, z\_width)

object.draw

**end**

**def** draw

glBegin(GL\_LINES)

glColor3d(1, 0, 0)

glVertex3d(0, 0, 0)

glVertex3d(x\_width, 0, 0)

glColor3d(0, 1, 0)

glVertex3d(0, 0, 0)

glVertex3d(0, y\_width, 0)

glColor3d(0, 0, 1)

glVertex3d(0, 0, 0)

glVertex3d(0, 0, z\_width)

glEnd

**end**

**end**

All the OpenGL magic happens in the draw action. *glBegin* and *glEnd* works like a ruby block where in argument you need to send the type of object that you want to draw in the argument. You can find all the types in the [official documentation](https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glBegin.xml).

Inside this block, you set the lines and their colour. This sets the colour for all the lines that you will draw after the method *glColor3d*. There are lots of different implementations of the *glColor* method that differ only by the type of argument.*glColor3d* takes arguments in *GLdouble*, *glColor3f*, *GLfloat*, etc. *glColor3d* takes three arguments – red, green and blue values. *glColor4\** can also take an alpha value.

To draw a line, you need to set two points, and OpenGL will draw a line between them. To do this, you need to call *glVertex*. You can pass two to four arguments of different types, depending on the method you are trying to call. In our case, we are sending the x, y and z coordinates of our future points. You can set as many lines in your *glBegin…glEnd* block as you wish. In my example, I’m drawing lines from the beginning of the axis *(x = 0, y = 0, z = 0)* to *(x = x\_width, y = y\_width, z = z\_width)* in pixels.

require 'opengl'

require 'glu'

require 'gosu'

**Dir**["objects/\*"].**each** {|file| require\_relative file }

include Gl, Glu

**class** Window &lt; Gosu::Window

**def** initialize

**super** 800, 600

**self**.caption = "Diatom's OpenGL Tutorial"

**end**

**def** update

**end**

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

Axis.draw(100, 100, 100)

**end**

**end**

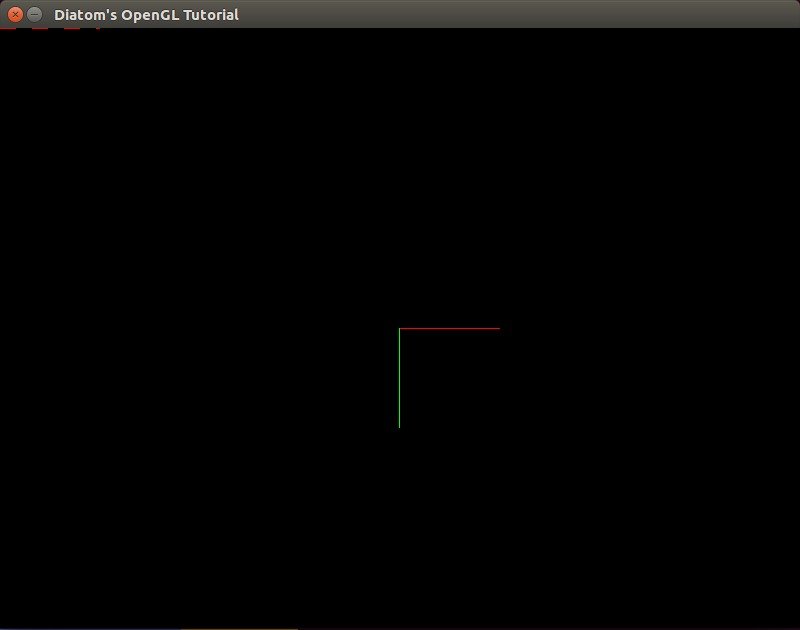
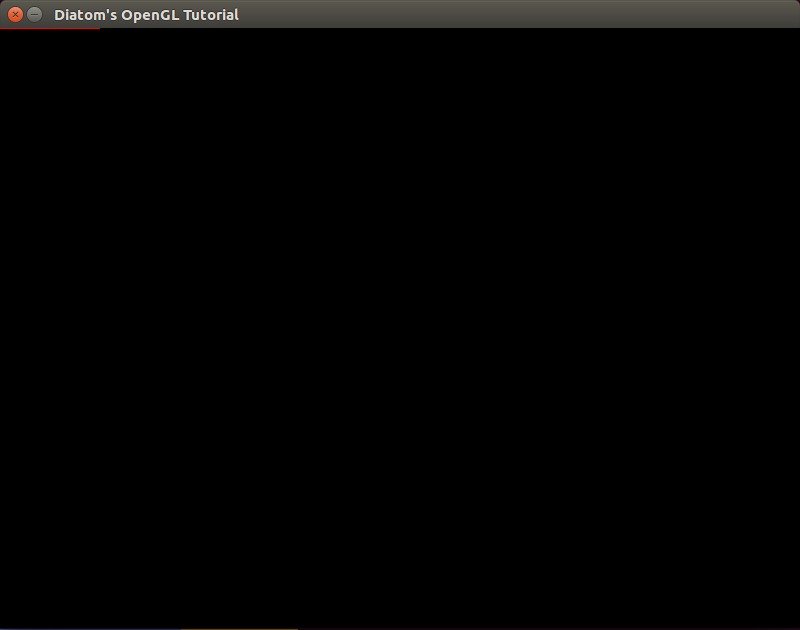
**def** button\_down(id)

exit **if** id == Gosu::KbEscape

**end**

**end**

Window.**new**.show

*glClear* clears buffers to preset values. After that we call *Axis.draw* to draw our axis.  
  
As you can see, we can’t find an *y* axis because there is a frame border in its place. So if you want, you can set start of the line from (x = 1) instead of (x = 0).  
  
Here we see a problem: what if we want to draw set different objects, but not from the top left corner? What if we want to move to the centre of the axes?

**MOVING THE CENTRE OF THE AXES**

To move to the centre of the axes, we can use the *glTranslate* method. It multiplies the current matrix by a translation matrix and produces a translation by x, y, z. All the objects after the *glTranslate* method will be drawn depending on those coordinates. To work with both translated and normal coordinates, we need to use the *glPushMatrix* and *glPopMatrix* methods, which will save the previous matrix and restore it. *glTranslate* produces different methods – *glTranslated(x, y, z), glTranslatef(x, y, z)* etc. – where x, y and z are the coordinates of a translation vector.

Let’s update our code using this method.

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glPushMatrix

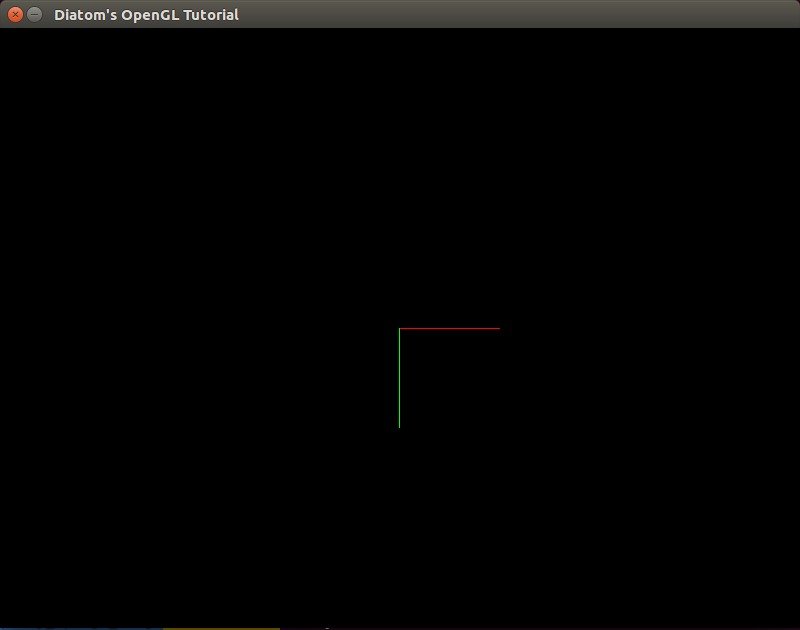
glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

glPopMatrix

**end**

**end**



As we can now see, our axis moved to the centre of our screen. To see the difference and how *glPushMatrix* and *glPopMatrix* work, let’s add the previous axis too.

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glPushMatrix

glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

glPopMatrix

glEnable(GL\_LINE\_STIPPLE)

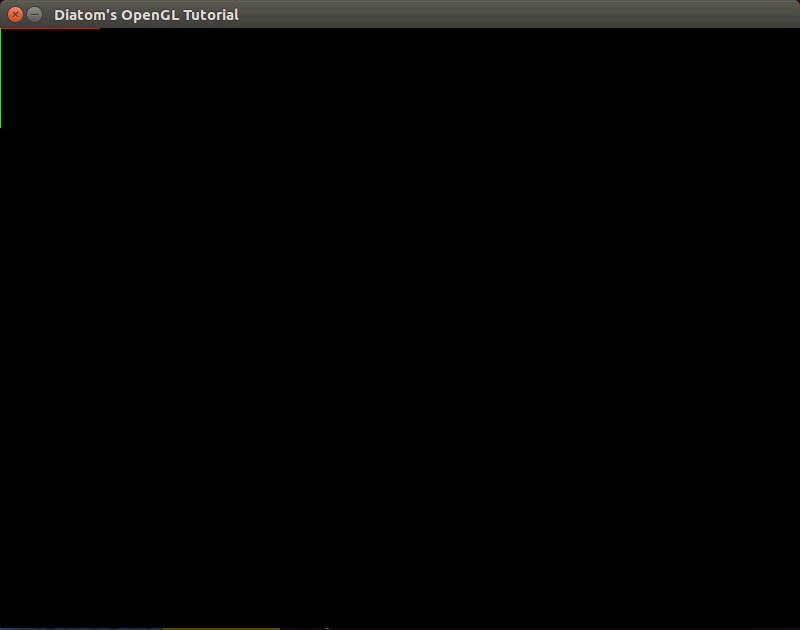
glLineStipple(2,0x00FF)

Axis.draw(100, 100, 100)

glDisable(GL\_LINE\_STIPPLE)

**end**

**end**



As you can see, the previous axis was drawn on the (0, 0, 0) coordinates with the stippled lines without any change to *axis.rb* or its coordinates.

It seems like we forgot about the z coordinate that we can’t see on our flat screen. To take a look at it, we need to use a viewpoint.

**GETTING INTO THE REAL 3D WORLD**

At first, I suggest that you build another object on our scene – a cube. It will help us to see the difference between the angles of our viewpoint.

To draw it, we can use *GL\_QUADS*. Each of the four points defines a Quad, so to make a cube we need six Quads = 24 points.

*objects/cube.rb:*

require 'pry'

**class** Cube

attr\_reader :x\_width, :y\_width, :z\_width, :mode, :mode\_face

**def** initialize(x\_width, y\_width, z\_width, mode\_face, mode)

@x\_width = x\_width

@y\_width = y\_width

@z\_width = z\_width

@mode = mode

@mode\_face = mode\_face

**end**

**def** self.draw(x\_width, y\_width, z\_width, mode\_face = GL\_FRONT\_AND\_BACK, mode = GL\_FILL)

object = **new**(x\_width, y\_width, z\_width, mode\_face, mode)

object.draw

**end**

**def** draw

glPolygonMode(mode\_face, mode);

glBegin(GL\_QUADS)

#yx

glColor3d(1, 0, 0)

glVertex3d(0, 0, 0)

glVertex3d(x\_width, 0, 0)

glVertex3d(x\_width, y\_width, 0)

glVertex3d(0, y\_width, 0)

#yz

glColor3d(0, 1, 0)

glVertex3d(0, y\_width, 0)

glVertex3d(0, 0, 0)

glVertex3d(0, 0, z\_width)

glVertex3d(0, y\_width, z\_width)

#zx

glColor3d(0, 0, 1)

glVertex3d(0, 0, z\_width)

glVertex3d(0, 0, 0)

glVertex3d(x\_width, 0, 0)

glVertex3d(x\_width, 0, z\_width)

#z -&gt; yx

glColor3d(1, 0, 0)

glVertex3d(0, 0, z\_width)

glVertex3d(x\_width, 0, z\_width)

glVertex3d(x\_width, y\_width, z\_width)

glVertex3d(0, y\_width, z\_width)

#x -&gt; yz

glColor3d(0, 1, 0)

glVertex3d(x\_width, y\_width, 0)

glVertex3d(x\_width, 0, 0)

glVertex3d(x\_width, 0, z\_width)

glVertex3d(x\_width, y\_width, z\_width)

#y -&gt; zx

glColor3d(0, 0, 1)

glVertex3d(0, y\_width, z\_width)

glVertex3d(0, y\_width, 0)

glVertex3d(x\_width, y\_width, 0)

glVertex3d(x\_width, y\_width, z\_width)

glEnd

**end**

**end**

I’ve added two new variables – arguments for *glPolygonMode* that set a rasterization mode. The arguments are face (front, back) and mode (*GL\_POINT, GL\_LINE, GL\_FILL*).

Let’s also update our axes and move code that defines their line types inside.

*objects/axis.rb*:

**class** Axis

attr\_reader :x\_width, :y\_width, :z\_width, :stipple

**def** initialize(x\_width, y\_width, z\_width, stipple = false)

@x\_width = x\_width

@y\_width = y\_width

@z\_width = z\_width

@stipple = stipple

**end**

**def** self.draw(x\_width, y\_width, z\_width, stipple = false)

object = **new**(x\_width, y\_width, z\_width, stipple)

object.draw

**end**

**def** draw

**if** stipple

glEnable(GL\_LINE\_STIPPLE)

glLineStipple(2,0x00FF)

**end**

glBegin(GL\_LINES)

...

glEnd

glDisable(GL\_LINE\_STIPPLE) **if** stipple

**end**

**end**

Let’s update our code to use our new cubes.

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glPushMatrix

glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE)

glPopMatrix

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE)

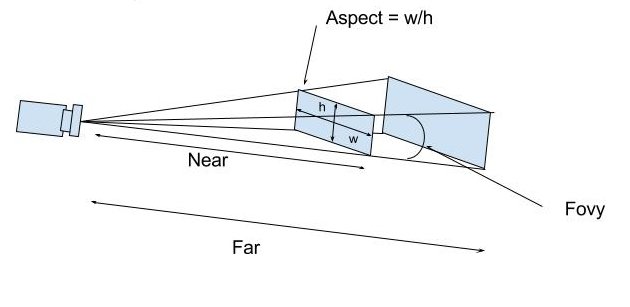
Axis.draw(100, 100, 100, **true**)

**end**

**end**



Looks clear, doesn’t it?

To add perspective, we need to use the *gluPerspective* method, which is used to set up a perspective projection. It’s done once to set up how the scene will be rendered. If *gluPerspective* is used, the perspective correction will happen while rendering. The arguments are the angle of view, aspect ratio, z-near and z-far. 

So, let’s add perspective to our application.

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

gluPerspective(130, width.to\_f/height, 0, 500)

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

glPushMatrix

glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE)

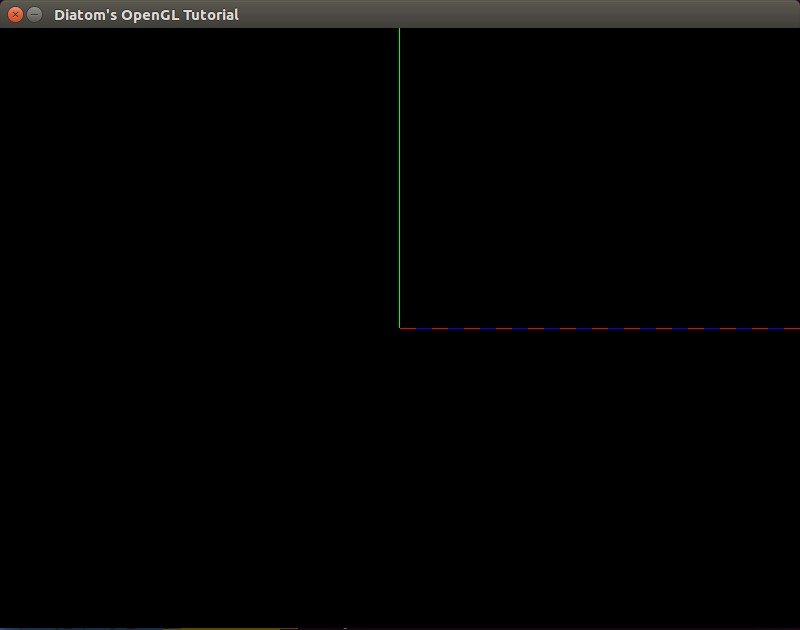
glPopMatrix

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE)

Axis.draw(100, 100, 100, **true**)

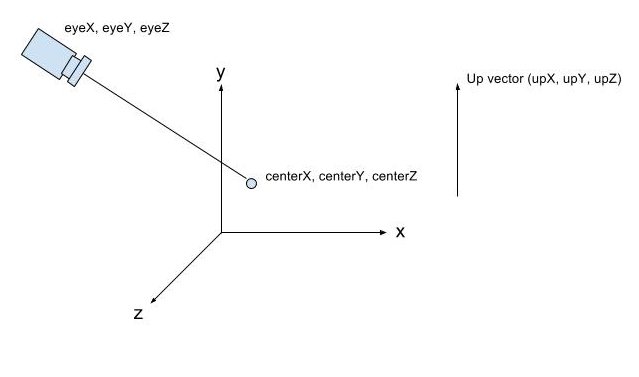
**end**

**end**



We can see that our field of view has been updated, and we can see only part of our (0, 0, 0) cube and axis. To see what *gluPerspective* really changes, let’s add a viewpoint.

To turn the player, OpenGL developers usually build a matrix of the view, load it and use later to turn the entire scene around the player. To simplify our code, we will create a ‘camera’ object that will help us look at the whole scene. To do this, we will use *gluLookAt*.



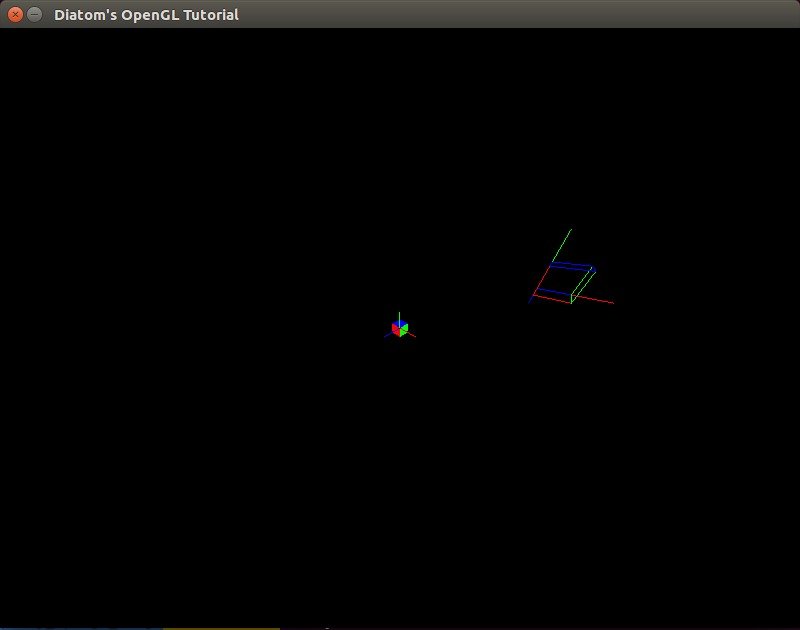
If we don’t set *gluLookAt*, OpenGL will automatically set it to (x = 0, y = 0, z = 0). Looking at our screen, that will be equal to *gluLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0)*. So, let’s set our *gluLookAt* properly, so we can see our scene.

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

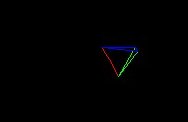
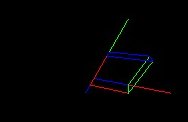
<b>gluLookAt(400, 400, 400, 0, 0, 0, 0, 1, 0)</b>

*gluLookAt* sets our camera to point (x = 400, y = 400, z = 400) and looks at point (x = 0, y = 0, z = 0), three other coordinates of `up` vector that we will not touch for now.



Let’s play with gluPerspective a bit now.

*gluPerspective(130, width.to\_f/height, 0.001, 225)*



What if we want to move the camera and see the other sides of our scene? Here gosu helps with its built-in I\O mechanisms.

**MOVING THE CAMERA**

As you noticed, we can change all of the arguments of the *gluLookAt* method. To move our camera, we need to bind those changes to the keyboard.

I will not explain the difficult math behind the matrixes and real turning of cameras as it was made in games\applications through formulas of radiuses and angles. I want to move our camera only through axes. I know that it’s not the best solution, but it’s the easiest for understanding.

Let’s move our *gluLookAt* method inside the new camera object that will be in objects/camera.rb

**class** Camera

attr\_reader :camera\_position, :look\_at\_object\_position, :window, :speed

**def** initialize(window)

@window = window

@camera\_position = Vector3.**new**(400, 400, 400)

@look\_at\_object\_position = Vector3.**new**(0, 0, 0)

@speed = 10

**end**

**def** capture

font = Gosu::Font.**new**(window, Gosu::default\_font\_name, 20)

button\_down

font.draw("Camera position: #{camera\_position.x}:#{camera\_position.y}:#{camera\_position.z}", 10, 10, 3.0, 1.0, 1.0, 0xffffffff)

font.draw("Look at position: #{look\_at\_object\_position.x}:#{look\_at\_object\_position.y}:#{look\_at\_object\_position.z}", 10, 25, 3.0, 1.0, 1.0, 0xffffffff)

font.draw("Your speed: #{speed}", 10, 40, 3.0, 1.0, 1.0, 0xffffffff)

gluLookAt(camera\_position.x, camera\_position.y, camera\_position.z, look\_at\_object\_position.x, look\_at\_object\_position.y, look\_at\_object\_position.z, 0, 1, 0)

**end**

**def** button\_down

@camera\_position.x += speed **if** Gosu::button\_down?(Gosu::KbRight) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@camera\_position.x -= speed **if** Gosu::button\_down?(Gosu::KbLeft) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@camera\_position.y += speed **if** Gosu::button\_down?(Gosu::KbUp) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@camera\_position.y -= speed **if** Gosu::button\_down?(Gosu::KbDown) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@camera\_position.z += speed **if** Gosu::button\_down?(Gosu::KB\_NUMPAD\_8) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@camera\_position.z -= speed **if** Gosu::button\_down?(Gosu::KB\_NUMPAD\_2) &amp;&amp; !Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.x += speed **if** Gosu::button\_down?(Gosu::KbRight) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.x -= speed **if** Gosu::button\_down?(Gosu::KbLeft) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.y += speed **if** Gosu::button\_down?(Gosu::KbUp) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.y -= speed **if** Gosu::button\_down?(Gosu::KbDown) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.z += speed **if** Gosu::button\_down?(Gosu::KB\_NUMPAD\_8) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

@look\_at\_object\_position.z -= speed **if** Gosu::button\_down?(Gosu::KB\_NUMPAD\_2) &amp;&amp; Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

**end**

**end**

To understand how everything is working, I bind each axis to a key. Here is the schema to move the camera:

UP, DOWN - by Y axis

LEFT, RIGHT - by X axis

NUM8, NUM2 - by Z axis

To move the target point, you need to hold shift, clicking on the same buttons.

I can display the current coordinates of points and the current camera speed using gosu’s text output methods. To do this, we need to pass a ‘window’ object into our class and work with it. For camera position, I wrote a Vector3 class that contains x, y and z coordinates:  
objects/vector3.rb:

Vector3 = **Struct**.**new**(:x, :y, :z)

After we build our camera class, let’s use it inside run.rb.

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

gluPerspective(90, width.to\_f/height, 0, 500)

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

@camera.capture

glPushMatrix

glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE)

glPopMatrix

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_FILL)

Axis.draw(100, 100, 100, **true**)

**end**

**end**

‘Fly’ a bit on our scene to see how *gluLookAt* works.

I suggest rewriting the keyboard settings a bit to make them more useful. Also, I want to be able to increase our camera’s speed. For this I want to rewrite the camera object:  
objects/camera.rb

**def** button\_down

check\_speed

**case**

**when** Gosu::button\_down?(Gosu::KbD)

@camera\_position.x += speed

@look\_at\_object\_position.x += speed

**when** Gosu::button\_down?(Gosu::KbA)

@camera\_position.x -= speed

@look\_at\_object\_position.x -= speed

**when** Gosu::button\_down?(Gosu::KbW)

@camera\_position.z -= speed

@look\_at\_object\_position.z -= speed

**when** Gosu::button\_down?(Gosu::KbS)

@camera\_position.z += speed

@look\_at\_object\_position.z += speed

**when** Gosu::button\_down?(Gosu::KbUp)

@look\_at\_object\_position.y += speed

**when** Gosu::button\_down?(Gosu::KbDown)

@look\_at\_object\_position.y -= speed

**when** Gosu::button\_down?(Gosu::KbRight)

@look\_at\_object\_position.x += speed

**when** Gosu::button\_down?(Gosu::KbLeft)

@look\_at\_object\_position.x -= speed

**end**

**end**

**def** check\_speed

@speed = 10

@speed = 20 **if** Gosu::button\_down?(Gosu::KB\_LEFT\_SHIFT)

**end**

Let’s use the W and S keys to move the camera on the z axis, A and D to move the camera on the x axis, UP and DOWN to move the target of view on the y axis and LEFT and RIGHT to move the target of view on the x axis. Holding shift will increase our speed.

**LET’S PLAY!**

Let’s add a cube object filled with lines to our pointer. To do this, we need to update our cube code to receive the current coordinates and build a cube depending on them.

objects/cube.rb

require 'pry'

**class** Cube

attr\_reader :position, :x\_width, :y\_width, :z\_width, :mode, :mode\_face

**def** initialize(x\_width, y\_width, z\_width, mode\_face, mode, x = 0, y = 0, z = 0)

@x\_width = x\_width

@y\_width = y\_width

@z\_width = z\_width

@mode = mode

@mode\_face = mode\_face

@position = Vector3.**new**(x, y, z)

**end**

**def** self.draw(x\_width, y\_width, z\_width, mode\_face = GL\_FRONT\_AND\_BACK, mode = GL\_FILL, x = 0, y = 0, z = 0)

object = **new**(x\_width, y\_width, z\_width, mode\_face, mode, x, y, z)

object.draw

**end**

**def** draw

glPolygonMode(mode\_face, mode);

glBegin(GL\_QUADS)

#yx

glColor3d(1, 0, 0)

glVertex3d(position.x, position.y, position.z)

glVertex3d(x\_width + position.x, position.y, position.z)

glVertex3d(x\_width + position.x, y\_width + position.y, position.z)

glVertex3d(position.x, y\_width + position.y, position.z)

#yz

glColor3d(0, 1, 0)

glVertex3d(position.x, y\_width + position.y, position.z)

glVertex3d(position.x, position.y, position.z)

glVertex3d(position.x, position.y, z\_width + position.z)

glVertex3d(position.x, y\_width + position.y, z\_width + position.z)

#zx

glColor3d(0, 0, 1)

glVertex3d(position.x, position.y, z\_width + position.z)

glVertex3d(position.x, position.y, position.z)

glVertex3d(x\_width + position.x, position.y, position.z)

glVertex3d(x\_width + position.x, position.y, z\_width + position.z)

#z -&gt; yx

glColor3d(1, 0, 0)

glVertex3d(position.x, position.y, z\_width + position.z)

glVertex3d(x\_width + position.x, position.y, z\_width + position.z)

glVertex3d(x\_width + position.x, y\_width + position.y, z\_width + position.z)

glVertex3d(position.x, y\_width + position.y, z\_width + position.z)

#x -&gt; yz

glColor3d(0, 1, 0)

glVertex3d(x\_width + position.x, y\_width + position.y, position.z)

glVertex3d(x\_width + position.x, position.y, position.z)

glVertex3d(x\_width + position.x, position.y, z\_width + position.z)

glVertex3d(x\_width + position.x, y\_width + position.y, z\_width + position.z)

#y -&gt; zx

glColor3d(0, 0, 1)

glVertex3d(position.x, y\_width + position.y, z\_width + position.z)

glVertex3d(position.x, y\_width + position.y, position.z)

glVertex3d(x\_width + position.x, y\_width + position.y, position.z)

glVertex3d(x\_width + position.x, y\_width + position.y, z\_width + position.z)

glColor3d(1, 1, 1) #return back primary color

glEnd

**end**

**end**

Now we can update our main window’s method draw with this code

glPushMatrix

glTranslated(width/2, height/2, 0)

Axis.draw(100, 100, 100)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_FILL)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE, @camera.look\_at\_object\_position.x, @camera.look\_at\_object\_position.y, @camera.look\_at\_object\_position.z)

glPopMatrix

At the end, you should see this view:



**CUBE CREATOR**

What if we want to place some new cubes? Easy!

Let’s change our run.rb code to

require 'opengl'

require 'glu'

require 'gosu'

**Dir**["objects/\*"].**each** {|file| require\_relative file }

include Gl, Glu

**class** Window &lt; Gosu::Window

<b>attr\_accessor :scene\_objects, :translation</b>

**def** initialize

**super** 800, 600

**self**.caption = "Diatom's OpenGL Tutorial"

@camera = Camera.**new**(**self**)

<b>@scene\_objects = []

@translation = Vector3.**new**(width/2, height/2,0)</b>

**end**

**def** update

**end**

**def** draw

gl **do**

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

gluPerspective(90, width.to\_f/height, 0, 500)

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

@camera.capture

<b>@scene\_objects &lt;&lt; Axis.**new**(100, 100, 100)

@scene\_objects &lt;&lt; Cube.**new**(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_FILL)</b>

glPushMatrix

<b>glTranslated(translation.x, translation.y, translation.z)

@scene\_objects.**each do** |object|

object.draw

**end**</b>

glPopMatrix

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_LINE, @camera.look\_at\_object\_position.x, @camera.look\_at\_object\_position.y, @camera.look\_at\_object\_position.z)

Cube.draw(50, 50, 50, GL\_FRONT\_AND\_BACK, GL\_FILL)

Axis.draw(100, 100, 100, **true**)

**end**

**end**

**def** button\_down(id)

exit **if** id == Gosu::KbEscape

<b>add\_cube **if** id == Gosu::KbSpace</b>

**end**

<b>

**def** add\_cube

@scene\_objects &lt;&lt; Cube.**new**(

50, 50, 50,

GL\_FRONT\_AND\_BACK, GL\_FILL,

@camera.look\_at\_object\_position.x - translation.x, @camera.look\_at\_object\_position.y - translation.y, @camera.look\_at\_object\_position.z - translation.z

)

**end**

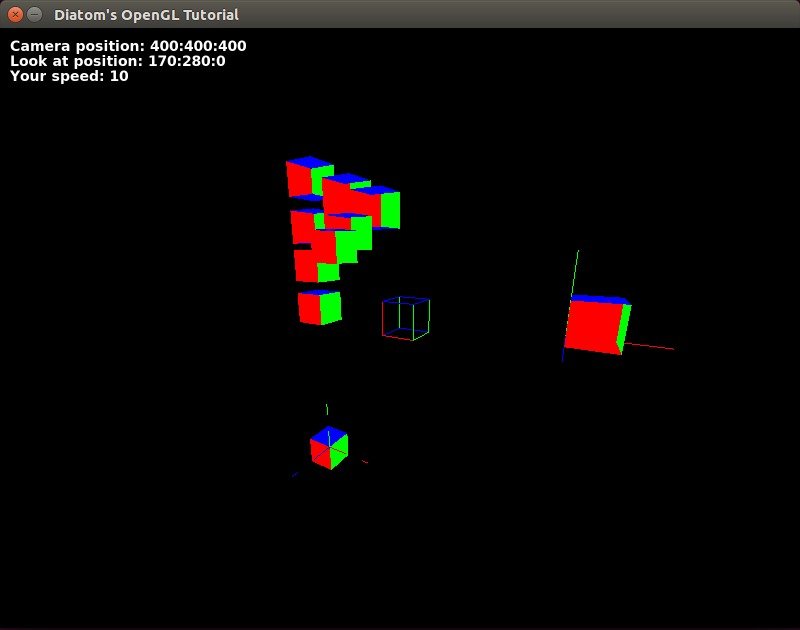
</b>

**end**

Window.**new**.show

I have highlighted the changes. As you will notice, I moved the translation vector to the variable of Vector3 class, moved all objects that will be translated and drawn on the scene to the array @scene\_objects and then placed them on the scene with a loop.

To add a new cube, I can just create a new cube object and place it in the array. Coordinates should depend on the translation vector to place everything on valid places.



Let’s add some textures. We can create a gosu image object from file and then get texture information from it with the *gl\_tex\_info* method. To build it, let’s add a class ‘texture.’

**class** Texture

attr\_accessor :info

**def** initialize(window, filepath)

@image = Gosu::Image.**new**(window, filepath, {tileable: **true**, retro: **false**})

@info = @image.gl\_tex\_info

# gl\_tex\_info can return nil if the image was too large to fit onto

# a single OpenGL texture and was internally split up.

**end**

**end**

Then we will need to update our cube draw method with:

texture\_info = texture.info

glEnable(GL\_TEXTURE\_2D) # enables two-dimensional texturing to perform

glBindTexture(GL\_TEXTURE\_2D, texture\_info.tex\_name) # bing named texture to a target

glPixelStorei(GL\_UNPACK\_ALIGNMENT,1)

draw\_textured\_cube

glDisable(GL\_TEXTURE\_2D)

where specify the texture’s edge that will be assigned to that point. Basically, the texture’s left bottom coordinates are (0, 0) and top right coordinates are (1, 1), and we stretch it between the edges of our cube. Code sample for one side:

glTexCoord2d(0,0); glVertex3d(position.x, position.y, position.z)

glTexCoord2d(1,0); glVertex3d(x\_width + position.x, position.y, position.z)

glTexCoord2d(1,1); glVertex3d(x\_width + position.x, y\_width + position.y, position.z)

glTexCoord2d(0,1); glVertex3d(position.x, y\_width + position.y, position.z)

By adding some lights with

glEnable(GL\_LIGHTING)

glLightfv(GL\_LIGHT0, GL\_AMBIENT, [0.5, 0.5, 0.5, 1])

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, [1, 1, 1, 1])

glLightfv(GL\_LIGHT0, GL\_POSITION, [1, 1, 1,1])

glLightfv(GL\_LIGHT1, GL\_AMBIENT, [0.5, 0.5, 0.5, 1])

glLightfv(GL\_LIGHT1, GL\_DIFFUSE, [1, 1, 1, 1])

glLightfv(GL\_LIGHT1, GL\_POSITION, [100, 100, 100,1])

glEnable(GL\_LIGHT0)

depth test to depth buffer comparisons

glEnable(GL\_DEPTH\_TEST);

glDepthMask(GL\_TRUE);

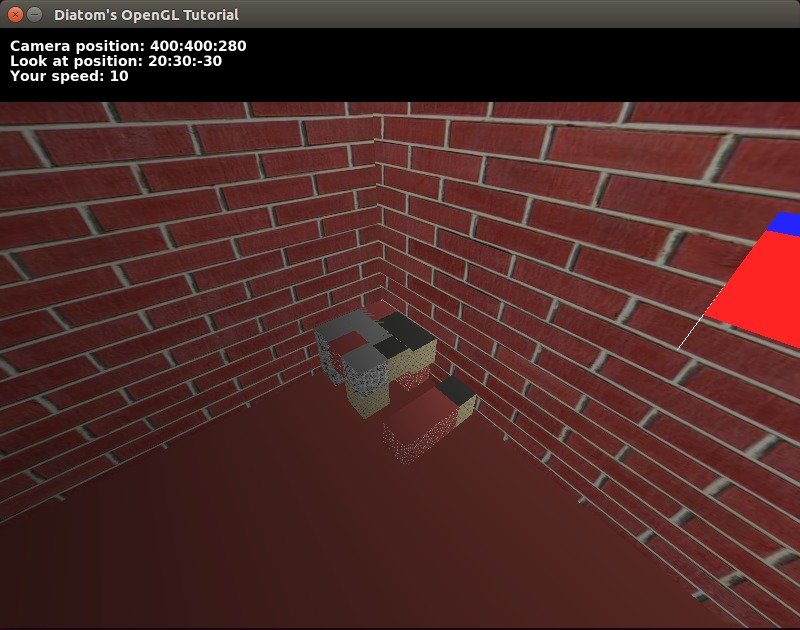
glDepthFunc(GL\_LEQUAL);

and textures with

glEnable(GL\_TEXTURE\_2D) # enables two-dimensional texturing to perform

glBindTexture(GL\_TEXTURE\_2D, texture\_info.tex\_name) # bing named texture to a target

glPixelStorei(GL\_UNPACK\_ALIGNMENT,1)

we can build nice textured scene with real lighting.  


With some additional work, we can achieve a switch of material for blocks that we are trying to build and other nice things.

Video: [YouTube](https://youtu.be/aKyR4a5SGbo)  
All the code for that can be found at [our repository on GitHub](https://github.com/DiatomEnterprises/ruby_opengl)

OpenGL is common everywhere and learning it once, you will be able to build projects with it on any language. The same is with Ruby. Just imagine something cool, and we can build it with Ruby!