CARGANDO CAPAS DE POSTGIS EN EL VISOR DE PYQGIS

Germán Alonso Carrillo Romero
geotux_tuxman@linuxmail.org
http://geotux.tuxfamily.org

Introducción

La *API* de *QGIS* permite entre otras cosas, el cargue de capas de *PostGIS*. A continuación presento una forma de hacerlo en el visor de *PyQGIS*, implementando un formulario que almacena los parámetros de la conexión para facilitar el cargue.

A diferencia de la *API* de *MapWinGIS*, *QGIS* provee clases para cargar capas de *PostGIS,* una característica indispensable en una librería de desarrollo de aplicaciones espaciales.

Según la documentación de *PyQGIS*, la forma de cargar las capas de *PostGIS* es la siguiente:

```
uri = QgsDataSourceURI()
uri.setConnection(host, puerto, nombre_db, usuario, password)
uri.setDataSource(esquema, tabla, columna_geometria, clausula_WHERE)
vlayer = QgsVectorLayer(uri.uri(), nombre_capa, "postgres")
```

Sin embargo, para este ejercicio, debemos tener en cuenta lo siguiente:

- El usuario debe poder ingresar los parámetros en un formulario.
- Los parámetros que ingrese el usuario deben permanecer en el formulario durante la sesión de trabajo.
- El usuario debe poder seleccionar la capa a agregar de una lista desplegable.
- El usuario no debe preocuparse por conocer el esquema y el nombre de la columna geometría de la capa a agregar.
- Se debe informar la geometría de la capa a agregar.

Para lograr los objetivos, el formulario de cargue debe lucir así:

Figura 1. Formulario para cargar capas de PostGIS.

Para ello tendremos una clase que controle el formulario y una que nos permita acceder a la información de la capas geográficas disponibles en la base de datos de PostGIS.

Para empezar

Para empezar es importante haber realizado el visor de shapefiles de un *blog* anterior. Una vez terminado, debemos tener una carpeta llamada *visor_shapefiles_qgis* con dos archivos principales: *VisorShapefiles.py* y *visor_shapefiles.ui* Usaremos estos dos archivos como base para agregar el cargue de capas de *PostGIS* al visor.

Pasos a seguir

Debemos seguir los siguientes pasos para construir la nueva funcionalidad:

1. Agregar a la aplicación los recursos necesarios. Esto es, las imágenes que usaremos en el formulario.
2. Instalar *QtSql*. *QtSql* es parte de *Qt* y permite el acceso a bases de datos.
3. Agregar el módulo *conexion_postgis*. Este módulo nos permitirá conectarnos a la base de datos de *PostGIS.*
4. Agregar el módulo *cargue_postgis*.
5. Implementar la nueva funcionalidad en la clase *VisorShapefiles*.

1. Agregar a la aplicación los recursos necesarios

Los recursos son archivos (cursores, imágenes, íconos, etc.) que la aplicación necesita en un momento dado. *Qt* convierte dichos archivos a un formato binario que es independiente de la plataforma y que queda embebido en la aplicación.

Para este ejercicio necesitamos las imágenes que representarán las geometrías (*punto.png*, *linea.png* y *poligono.png*) y la que irá en el botón de conexión (*conectar.png*). En este enlace descargamos el archivo comprimido que contiene las imágenes y lo descomprimimos en la carpeta de la aplicación.

Para agregar los recursos utilizamos ***Qt Designer*.** Abrimos el programa y cargamos el archivo *visor_shapefiles.ui* Buscamos la ventana de recursos (*Resource Browser*). Si no se encuentra visible la habilitamos en el menú *View*.

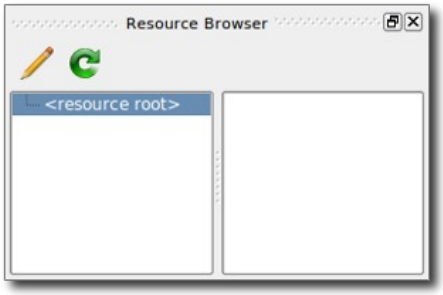

Figura 2. Ventana de recursos de *Qt Designer*.

Hacemos *click* al botón *Edit Resources* y en la ventana que se abre creamos un nuevo archivo de recursos, lo ubicamos en la carpeta de la aplicación y lo llamamos “recursos”, así:

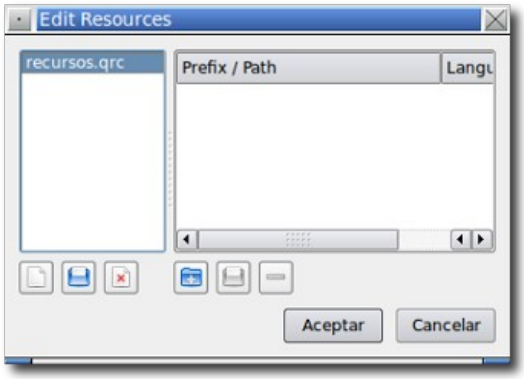

Figura 3. Ventana Editar Recursos.

Agregamos un nuevo prefijo haciendo *click* en el botón *Add Prefix* y lo nombramos “imgs”. Damos *click* al botón *Add Files* y seleccionamos los cuatro archivos *png* que se encuentran en la carpeta *imagenes.* Debemos tener algo como esto:

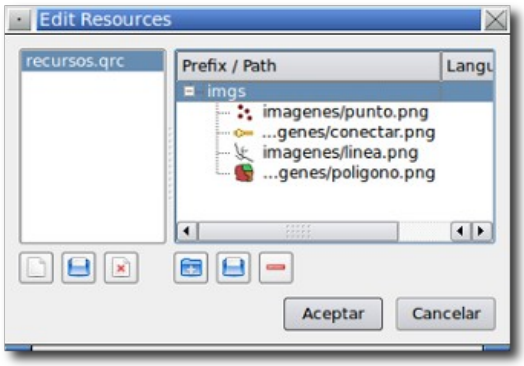

Figura 4. Archivos *png* cargados.

Damos *click* en *Aceptar* y cerramos *Qt Designer*. De esta forma hemos generado un archivo *recursos.qrc* que no es más que un *XML*, para generar el binario que la aplicación utilizará ejecutamos desde la terminal el siguiente comando, ubicados en la carpeta *visor_shapefiles_qgis*.

```
pyrcc4 -o recursos.py recursos.qrc
```

Con esto generamos el archivo *recursos.py* que más adelante incluiremos en la aplicación como múdulo.

2. Instalar QtSql

QtSql es un módulo de *Qt* que permite realizar conexiones a bases de datos para ejecutar consultas.

Para instalar *QtSql* en *Ubuntu* 9.04 ejecutamos el siguiente comando con permisos de administrador:

```
sudo apt-get install python-qt4-sql libqt4-sql-psql
```

3. Agregar el módulo *conexion_postgis*

En principio, la *API* de *QGIS* puede cargar capas de *PostGIS* sin que necesitemos hacer consultas a la base de datos. Sin embargo, como queremos facilitar al usuario el cargue de capas, debemos conocer cuáles capas están disponibles en la base de datos, cuál es su esquema y cuál es su campo geometría.

En el módulo *conexion_postgis* definiremos la clase *ConexionPgSQL,* la cual hará una consulta a la base de datos de *PostGIS* para extraer información de la tabla *geometry_columns,* que almacena datos relevantes de las capas disponibles. Los resultados serán retornados en un arreglo con la siguiente información para cada capa:

- Esquema: Útil para determinar la capa a cargar.
- Nombre de las capa: Servirá para darle a elegir al usuario la capa a cargar.
- Nombre de la columna geometría: Útil para determinar la capa a cargar.

El código de la clase *ConexionPgSQL* es el siguiente:

```
1     # -*- coding: utf-8 -*-
2
3     from PyQt4.QtSql import *
4
5     class ConexionPgSQL():
6         """ Provee los métodos para conectarse a PostgreSQL/PostGIS """
7
8         def __init__(self, servidor, puerto, bd, usuario, contrasena):
9
10            if QSqlDatabase.contains("midb"):
11                QSqlDatabase.removeDatabase("midb") # Uso del método estático
12
13            # Paramentros conexion
14            self.db = QSqlDatabase.addDatabase("QPSQL", "midb")
15            self.db.setHostName(servidor)
16            self.db.setPort(int(puerto))
17            self.db.setDatabaseName(bd)
18            self.db.setUserName(usuario)
19            self.db.setPassword(contrasena)
20
21            self.query = QSqlQuery(self.db) # Inicializar objeto query
22
23        def conectar(self):
24            """ Abre la conexión a la base de datos """
25            ok = self.db.open()
26            return ok
27
28        def consultarCapas(self):
29            """ Retorna un arreglo de datos con la información de la tabla
30                geometry_columns de PostGIS """
31            if self.query.exec_("SELECT f_table_schema, f_table_name, " +
32                "f_geometry_column, type FROM public.geometry_columns"):
33                record = self.query.record()
34                posEsquema = record.indexOf("f_table_schema")
35                posTabla = record.indexOf("f_table_name")
36                posGeom = record.indexOf("f_geometry_column")
```

```
37                posType = record.indexOf("type")
38
39                aTabla = []
40
41                while self.query.next(): # Leer los datos de la consulta
42                    esquema = self.query.value(posEsquema).toString()
43                    tabla = self.query.value(posTabla).toString()
44                    geom = self.query.value(posGeom).toString()
45                    tipo = self.query.value(posType).toString()
46
47                    aDatos = [esquema, tabla, geom, tipo]
48                    aTabla.append(aDatos)
49
50                return aTabla
51
52            else:
53                return None
```

La clase tiene tres métodos, el método *__init__* que inicializa el objeto, el método *conectar*, que abre la conexión a la base de datos retornando cualquier eventualidad, y el método *consultarCapas* que consulta la tabla *geometry_columns* y devuelve un arreglo si la consulta tuvo éxito. Este arreglo será utilizado en el módulo *cargue_postgis*.

El código anterior debe guardarse en un archivo *conexion_postgis.py*

4. Agregar el módulo *cargue_postgis*

En el modulo *cargue_postgis* definimos una clase llamada *DialogoCarguePostGIS* que contiene la definición de la interfaz (hereda de la clase *QDialog*), la disposición de los controles en el formulario y la conexión a *PostGIS* para conocer la información de las capas.

Se inicia importando los módulos requeridos: El módulo *re* para construir expresiones regulares que servirán para validar los parámetros de conexión, los módulos básicos de *Qt,* la clase *ConexionPgSQL* del módulo *conexion_postgis* y el módulo de recursos que creamos en el paso 1.

```
1     # -*- coding: utf-8 -*-
2     import re
3
4     from PyQt4.QtCore import *
5     from PyQt4.QtGui import *
6
7     from conexion_postgis import ConexionPgSQL
8     from recursos import *
9
```

Para lograr que los parámetros que ingresa el usuario permanezcan durante la sesión y para que cada vez que se llame el formulario, se obtenga una única instancia del mismo, se emplea el patrón *Singleton*:

```
10    class DialogoCarguePostGIS(QDialog):
11        """ Provee la interfaz de la herramienta Cargar capa PostGIS """
```

```
12
13          instancia = None # Variable para controlar la única instancia
14
15          class SingletonHelper:
16              """ Provee un método factory para el Singleton """
17              def __call__( self, *args, **kw ) :
18
19                  if DialogoCarguePostGIS.instancia is None :
20                      object = DialogoCarguePostGIS(args[0]) # Argumento parent
21                      DialogoCarguePostGIS.instancia = object
22
23                  return DialogoCarguePostGIS.instancia
24
25          obtenerInstancia = SingletonHelper() # Método para obtener la instancia
26
```

El método *__init__* comienza lanzando un error si se intenta instanciar la clase más de una vez. Luego se inicializa la clase *QDialog*, de la cual hereda *DialogoCarguePostGIS*.

```
27          def __init__(self, parent=None):
28
29              if not DialogoCarguePostGIS.instancia == None :
30                  raise RuntimeError, 'Solo se permite una instancia!'
31
32              QDialog.__init__(self, parent) # Inicializar la clase QDialog
33
```

Posteriormente se definen y se ubican los controles de la interfaz. En la línea 54 se utiliza el módulo de recursos para ubicar la imagen *conectar.png* en el botón *btnConectar.*

```
34              # Diagramación de la Interfaz
35              self.setWindowTitle('Cargar capa de PostGIS')
36
37              lbl_servidor = QLabel('Servidor:')
38              lbl_puerto = QLabel('Puerto:')
39              lbl_bd = QLabel('Base de datos:')
40              lbl_usuario = QLabel('Usuario:')
41              lbl_contrasena = QLabel('Contraseña:')
42              lbl_tabla = QLabel('Tabla:')
43
44              self.__servidorEdit = QLineEdit("localhost")
45              self.__puertoEdit = QLineEdit("5432")
46              self.__bdEdit = QLineEdit("")
47              self.__usuarioEdit = QLineEdit("")
48              self.__contrasenaEdit = QLineEdit("")
49              self.__contrasenaEdit.setEchoMode(QLineEdit.Password)
50
51              self.cboTabla = QComboBox() # ComboBox para tablas disponibles
52              self.cboTabla.setObjectName("cboTabla")
53
54              self.btnConectar = QPushButton(QIcon(":/imgs/imagenes/conectar.png"),
55                  "Conectar")
56
57              buttonBox = QDialogButtonBox()
58              self.btnAceptar = buttonBox.addButton('Aceptar',
```

```
59                  QDialogButtonBox.AcceptRole)
60              self.btnAceptar.setEnabled(False)
61              buttonBox.addButton('Cancelar', QDialogButtonBox.RejectRole)
62
63              QObject.connect(self.btnConectar, SIGNAL("clicked()"), self.__conectar)
64              QObject.connect(buttonBox, SIGNAL("accepted()"), self.__aceptar)
65              QObject.connect(buttonBox, SIGNAL("rejected()"), self.cerrar)
66
67              grid = QGridLayout()
68              grid.setSpacing(0)
69
70              grid.addWidget(lbl_servidor, 1, 0)
71              grid.addWidget(self.__servidorEdit, 1, 1)
72
73              grid.addWidget(lbl_puerto, 2, 0)
74              grid.addWidget(self.__puertoEdit, 2, 1)
75
76              grid.addWidget(lbl_bd, 3, 0)
77              grid.addWidget(self.__bdEdit, 3, 1)
78
79              grid.addWidget(lbl_usuario, 4, 0)
80              grid.addWidget(self.__usuarioEdit, 4, 1)
81
82              grid.addWidget(lbl_contrasena, 5, 0)
83              grid.addWidget(self.__contrasenaEdit, 5, 1)
84
85              grid.setRowStretch(6,1)
86
87              grid.addWidget(self.btnConectar, 7, 0,1,2, Qt.AlignCenter)
88
89              grid.setRowStretch(8,1)
90
91              grid.addWidget(lbl_tabla, 9, 0)
92              grid.addWidget(self.cboTabla, 9, 1)
93
94              grid.setRowStretch(10,1)
95
96              grid.addWidget(buttonBox,11,0,1,2, Qt.AlignCenter)
97
98              self.setLayout(grid)
99              self.setFixedSize(260, 275)
100
```

El método *__init__* termina con la inicialización de la variable resultado, que sirve para conocer si el usuario aceptó o canceló el formulario de cargue.

```
101             self.resultado = False # Para saber si se aceptó o canceló el form
102
```

A continuación se define el método *__conectar*, que se encarga de validar los parámetros de conexión y mostrar en un *ComboBox* las capas disponibles. En las líneas 105 y 106 se establecen expresiones regulares para el campo puerto (cuatro dígitos) y para los demás parámetros (longitud de la cadena mayor que cero y sin espacios). Si los parámetros de conexión son correctos, se instancia en la línea 123 la clase *ConexionPgSQL* del módulo

conexion_postgis que agregamos en el paso 3. En la línea 129 se comienza a llenar el combo *cboTabla* agregando el nombre de las capas disponibles y una imagen alusiva a su geometría.

```
103       def __conectar(self):
104           """ Valida parámetros de conexión y carga las capas disponibles """
105           validacion_puerto = re.compile("^(\d{4})$")
106           validacion_cadenas = re.compile("^(\S)*\S$")
107
108           self.cboTabla.clear() # Inicializar cboTabla
109
110           if (validacion_cadenas.match(str(self.__servidorEdit.text())) and
111               validacion_puerto.match(str(self.__puertoEdit.text())) and
112               validacion_cadenas.match(str(self.__bdEdit.text())) and
113               validacion_cadenas.match(str(self.__usuarioEdit.text())) and
114               validacion_cadenas.match(str(self.__contrasenaEdit.text()))):
115
116               self.servidor = self.__servidorEdit.text()
117               self.puerto = self.__puertoEdit.text()
118               self.bd = self.__bdEdit.text()
119               self.usuario = self.__usuarioEdit.text()
120               self.contrasena = self.__contrasenaEdit.text()
121
122               # Cargar capas disponibles en la BD
123               conn = ConexionPgSQL(self.servidor, self.puerto,
124                   self.bd, self.usuario, self.contrasena)
125
126               if conn.conectar():
127                   self.aDatosBD = conn.consultarCapas()
128
129                   if self.aDatosBD:
130                       for i in range(0, len(self.aDatosBD)):
131
132                           if self.aDatosBD[i][3] == "POINT":
133                               icono= QIcon(":/imgs/imagenes/punto.png")
134
135                           elif self.aDatosBD[i][3] == "LINESTRING":
136                               icono= QIcon(":/imgs/imagenes/linea.png")
137
138                           elif self.aDatosBD[i][3] == "POLYGON":
139                               icono= QIcon(":/imgs/imagenes/poligono.png")
140
141                           self.cboTabla.addItem(icono, self.aDatosBD[i][1])
142
143                       self.btnAceptar.setEnabled(True) # Habilitar
144                   else:
145                       self.btnAceptar.setEnabled(False) # Deshabilitar
146                       msg = QMessageBox(QMessageBox.Warning, 'Advertencia',
147                           'No existen tablas espaciales en la base de datos ' +
148                           'que\npuedas visualizar con los parámetros ' +
149                           'ingresados.', QMessageBox.NoButton, self)
150                       msg.addButton('Aceptar', QMessageBox.AcceptRole)
151                       msg.exec_()
152               else:
153                   self.btnAceptar.setEnabled(False) # Deshabilitar
154                   msg = QMessageBox(QMessageBox.Warning, 'Advertencia',
155                       'No se ha podido establecer la conexión con \nla ' +
156                       'base de datos que especificaste.\n\nRevisa los ' +
```

```
157                     'parámetros de conexión.', QMessageBox.NoButton, self)
158                 msg.addButton('Aceptar', QMessageBox.AcceptRole)
159                 msg.exec_()
160         else:
161             self.btnAceptar.setEnabled(False) # Deshabilitar
162             msg = QMessageBox(QMessageBox.Warning, 'Advertencia1',
163                 'Has llenado mal algún campo, por favor revisa.',
164                 QMessageBox.NoButton, self)
165             msg.addButton('Aceptar', QMessageBox.AcceptRole)
166             msg.exec_()
167
```

Se define también el método *__aceptar*, desde el cual se almacenan los parámetros de conexión en variables públicas que son llamadas desde el método *cargarPostGIS* de la clase *VisorShapefiles,* para finalmente cargar la capa *PostGIS* en el mapa. Se cierra la ventana no sin antes almacenar en la variable *resultado* que se ha aceptado el formulario de cargue, útil en la clase *VisorShapefiles.*

```
168     def __aceptar(self):
169         """ SLOT para el click en el botón Aceptar """
170         self.resultado = True # Aceptado
171
172         # Esta asignación permite validar cambios posteriores a la conexion
173         self.servidor = self.__servidorEdit.text()
174         self.puerto = self.__puertoEdit.text()
175         self.bd = self.__bdEdit.text()
176         self.usuario = self.__usuarioEdit.text()
177         self.contrasena = self.__contrasenaEdit.text()
178
179         # Datos de la tabla a cargar
180         self.esquema = self.aDatosBD[self.cboTabla.currentIndex()][0]
181         self.tabla = self.aDatosBD[self.cboTabla.currentIndex()][1]
182         self.geom = self.aDatosBD[self.cboTabla.currentIndex()][2]
183         self.close()
184
```

Finalmente, se definen los métodos *mostrar* y *cerrar*. Nótese que en el método *cerrar* se oculta el formulario de cargue y se establece la variable *resultado* como falsa para tenerlo en cuenta en el método *cargarPostGIS* de la clase *VisorShapefiles*, desde la cual se carga la capa de *PostGIS* al mapa.

```
185     def mostrar(self):
186         self.exec_()
187
188     def cerrar(self):
189         self.resultado = False # Cancelado
190         self.close()
```

Guardamos el código en un archivo con nombre *cargue_postgis.py* en la carpeta de la aplicación.

5. Implementar la nueva funcionalidad en la clase *VisorShapefiles*

Para implementar el cargue de capas de *PostGIS* a la aplicación debemos agregar un nuevo botón que llame el formulario de cargue creado en el paso 4.

Para empezar, en la sección de sentencias para importar módulos debemos importar la clase *DialogoCarguePostGIS* del módulo *cargue_postgis*.

```
from cargue_postgis import DialogoCarguePostGIS
```

En la sección de los comportamientos de los botones, en donde definimos los *SIGNALS/SLOTS,* agregamos un objeto *QAction* con el ícono correspondiente y enlazamos su *SIGNAL activated* al *SLOT cargarPostGIS*, que definiremos más adelante.

```
self.actionCargarPostGIS = QAction(QIcon(qgis_prefix +
    '/share/qgis/themes/classic/mActionAddLayer.png'),
    "Agregar capa de PostGIS", self.frame)
self.connect(self.actionCargarPostGIS, SIGNAL("activated()"),
    self.cargarPostGIS)
```

Luego, en la sección de definición de la *Toolbar* agregamos la *Action* que acabamos de definir:

```
self.toolbar.addAction(self.actionCargarPostGIS)
```

Finalmente, definimos el *SLOT cargarPostGIS* que muestra el formulario de cargue y se encarga de agregar la capa definida por el usuario al mapa.

```
def cargarPostGIS(self):
    """ Agregar capa de PostGIS al canvas """
    dialogoPg = DialogoCarguePostGIS.obtenerInstancia(self)
    dialogoPg.mostrar()

    if dialogoPg.resultado:
        uri = QgsDataSourceURI()
        # Host, puerto, bd, usuario, password
        uri.setConnection(dialogoPg.servidor, dialogoPg.puerto, dialogoPg.bd,
            dialogoPg.usuario, dialogoPg.contrasena)

        # Esquema, tabla, columna geometria y WHERE opcional
        uri.setDataSource(dialogoPg.esquema, dialogoPg.tabla, dialogoPg.geom)

        layer = QgsVectorLayer(uri.uri(), str(dialogoPg.tabla).capitalize(),
            "postgres")

        if layer.isValid():
            # Agregar el layer al registro
            QgsMapLayerRegistry.instance().addMapLayer(layer);

            # Fijar el extent al extent del primer layer cargado
            if self.canvas.layerCount() == 0:
                self.canvas.setExtent(layer.extent())
```

```
            # Fijar el conjunto de capas (LayerSet) para el map canvas
            self.layers.insert(0, QgsMapCanvasLayer(layer))
            self.canvas.setLayerSet(self.layers)
        else:
            msg = QMessageBox(QMessageBox.Warning, 'Advertencia', 'No se ' +
                'ha podido establecer la conexión con \nla base de datos ' +
                'que especificaste.\n\nRevisa los parámetros de conexión.',
                QMessageBox.NoButton, self)
            msg.addButton('Aceptar', QMessageBox.AcceptRole)
            msg.exec_()
```

Con esto termina la implementación del cargue de capas de *PostGIS* en el visor. En la clase *VisorShapefiles* queda algún código redundante (el cargue de capas al *canvas,* que se realiza tanto para cargar *Shapefiles* como para cargar capas de *PostGIS*) y esto puede ser mejorado (agregando una función *cargarCapa*, por ejemplo).

El visor ahora debe lucir así:

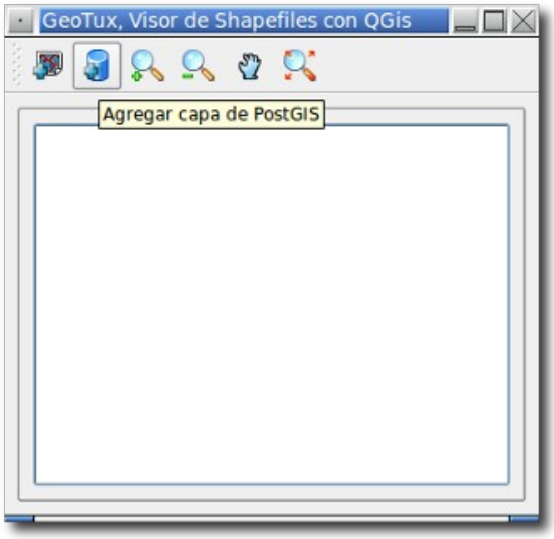

Figura 5. Visor de PyQGIS con la nueva funcionalidad.

Y el formulario de cargue así:

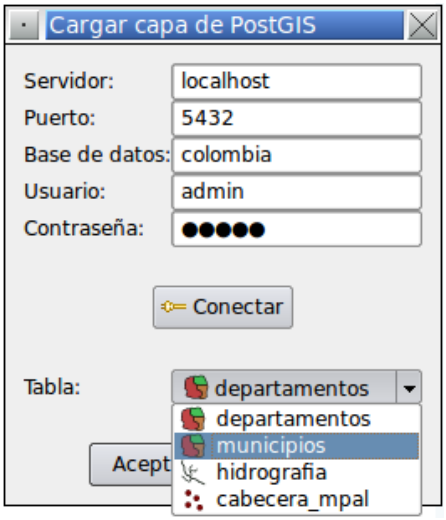

Figura 6. Formulario de cargue.

Figura 7. Capas de PostGIS cargadas.

Archivos para descargar

Los módulos *conexion_postgis* y *cargue_postgis* se encuentran disponibles para su descarga en este enlace.

Conclusiones

La *API* de *Quantum GIS* admite diversos formatos que pueden ayudar a enriquecer el visor de PyQGIS. Por ejemplo, podemos cargar geometrías de bases de datos de *PostgreSQL/PostGIS*, *SpatiaLite*, archivos ráster, servicios *web* de mapas *(WMS)* y archivos de texto delimitados por comas.