VBASE: A GIS Visual Database

Junior Independent Study CS-200

Brianna Brown Richardson

April 28, 2018

Abstract

Geographic Information Systems, GIS, is a system that interprets and presents geographic related data visually. The data GIS uses is spatial data which is information that identifies physical and non-physical features of a particular space [1]. For the scope of this project, the development of a 2D and 3D visual GIS database with a User Interface is a realistic goal. This software, referred to as VBASE, allows users to upload .shp, and .dbf files to a SQLite database while also being able to visualize such data both two and three-dimensionally. The premise of this paper is to provide in-depth explanation of the structure and software development process of VBASE. VBASE is a tool that allows users both with or without extensive knowledge of GIS to visualize such data.

List of Figures

List of Tables

Contents

1 Introduction

This paper looks at the design process and feasibility of developing ones own geographic information system, GIS, Visual Database. The idea is not to reinvent the foundation of GIS, but to instead evolve it. GIS Powerhouse Environmental Systems Research Institute (ESRI), has developed that include tools and functions to perform a number of high level tasks [2], but are catered towards specific groups. VBASE is a GIS visual database developed in Python. VBASE uses a simple database to which users are able to upload shapefiles and then visualize that data in either 2D or 3D without having expertise in disciplines involving geography.

2 Background

In order to understand the development of VBASE, it is vital to have a grasp and understanding of certain frameworks and individual Application Programming Interfaces (APIs).

2.1 What is GIS?

To develop a geographic information system, it is necessary to understand GIS. GIS is simply an information system that deals with the interpretation of spatial and geographic data. These systems make it possible to visualize, manipulate, and manage these data. GIS also has many other functionalities that are used to analyze and connect data with geography [1].

2.1.1 History

The ideology of GIS dates all the way back to the 1800s. Cartography and science used mundane practices involving pen and paper as a tool to map geographic data and perform geospatial analysis. For example, Charles Picquet, a famous French cartographer, used geospatial analysis to track disease-related deaths based on location using a map containing detailed census tracts of these data. [4].

The Canadian Department of Forestry and Rural Development was the first to develop GIS. They coined the term by referring to their system as Canada GIS, CGIS. The invention of CGIS played a significant role in the future development of Graphical User Interfaces, GUIs, implemented for GIS [4]. CGIS paved the way for systems such as Aeronautical Reconnaissance Coverage Geographic Information System, ArcGIS, and Quantum Geographic Information Systems, QGIS. These two systems have implemented highly touted GUIs alongside high level GIS functionalities. Overall, CGIS's purpose was to implement tools to help manage and assess spatial data, which played a crucial role in the evolution of GIS.

2.1.2 Spatial Data

The data GIS uses is spatial data, which is information that identifies physical and non-physical features of a particular space. Spatial data is data relating to a specific geometric location, whether it is the geometry of that location or non-physical entities describing a trend occurring in the given location. The physical side of spatial data represents the shape and location of physical objects on the earth such as terrain, buildings, bodies of water, townships, and much more. The non-physical spatial data is information that relates to physical objects, while lacking the physical entity of being an object as seen in data involving trends correlated with physical locations.

2.1.3 Shapefiles

A popular data structure that is used to store spatial data for GIS is a shapefile. Shapefiles are files that contain spatial data relating polygons, lines, and points. The term shapefiles was coined and created by ESRI, which is the company behind the creation of ArcGIS. The term shapefile can be confusing, as it implies that it is a singular file, when in fact, a shapefile is a collection of different file types that point to multiple file extensions, and all files must fall under the same filename prefix. Each file type has a specific purpose. The adaptation of having a file format such as shapefiles allows flexibility and compromise for the sake of complexity. Most systems require three files to be contained in a shapefile. The first is the main file, a file with the .shp extension. The main file contains all geometric data: polygons, lines, and points, which ESRI refers to as shapes. In the main file, all shape types are stored. There are fourteen shape types a shapefile can contain and each is represented by an integer. The shape types and their integer keys are listed in table 1.

Table 1: All possible shape types for a shapefile

Key	ShapeType	Description
0	NULL SHAPE	a none shape value
1	POINT	an x,y coordinate
3	POLYLINE	a set of ordered vertices
5	POLYGON	a sequence of ordered vertices that created a closed loop
8	MULTIPOINT	a collection of POINT entities
11	POINTZ	an x,y,z coordinate
13	POLYLINEZ	a set of ordered vertices that are in the z-axis
15	POLYGONZ	a sequence of ordered vertices that created a closed loop in the z-axis
18	MULTIPOINTZ	a collection of POINTZ entities
21	POINTM	an x,y coordinate with an m measurement
23	POLYLINEM	a set of ordered vertices with an m measurement
25	POLYGONM	a sequence of ordered vertices that create a closed loop with an m measurement
28	MULTIPOINTM	a collection of POINTM entities
31	MULTIPATCH	a collection of surfaces that create a 3D object

Overall the main file is contains the data to visualize a shapefile and is considered the reference file to the dBASE file. The second required file is a dBASE file. It contains all the attributes and records of all the geometric data present in the main file. The final required file is the index file with a .shx extension. As the name suggests, the index file performs indexing into the shapefile records and shapes [2]. Note that even though it is stated that most systems require all three files, for this project, only the main file, .shp, and the dBASE file, .dbf, are needed, as index files are geared towards larger scaled systems similar to ArcGIS.

2.2 Kivy

In order to visualize shapefiles a User Interface (UI) is needed and Kivy provides the UI for applications. In the section 2.1, there is a brief discussion on the evolution of GIS. Recall GIS started as a system that was implemented via pen and paper and later evolved to system containing a GUI. The GUI of an application is a subset of the user interface, UI. The UI makes up the side of the application with which the user interacts, also referred to as the frontend. For VBASE, the UI is developed using the Kivy framework to handle all frontend interactions and events. Kivy is a Python framework created for user interface and application development.

A popular feature of Kivy is the KV language, also known as the Kivy Language. The KV language is an intuitive way to create Kivy applications. This language is simply a declarative and cascading style sheet (CSS) structured version of the pure python approach and developed in .kv files. The benefit of using a .kv file to build an application is its structure. These files are space oriented making it easy to view the relationship between all entities used in an application, which is one of the benefits to using Kivy in general. This framework allows for the application to be cross-platform meaning the application works within different platforms such as Windows, Mac OS, Android, and IOS. The foundation of Kivy is an interface of widgets. From the layout to the buttons and labels presented on a screen, every entity in a Kivy based application is a widget [6]. Such an architecture allows for components to essentially snap together or build upon one another fairly easily. An ideal way of thinking of the relationships between widgets is a tree. The root of the tree, also known as the absolute parent widget, is the base that supports a series of child and grandchild widgets. The absolute parent widget is the application class that holds all the UI widgets.

2.3 Matplotlib

Kivy is used to develop the UI with widgets, and an important widget used in VBASE is Matplotlib. In general Matplotlib is an open source data analysis tool created by John Hunter and his associates. Its purpose is to serve as a Gimp Tool Kit (GTK+) for visualizing data. Matplotlib's structure is based on a figure, which is an object that contains plotted data. Figures can have multiple functions performed on them. These functions create subplots that allow different figure types and styles to be presented on a single figure [5]. A simple of way of thinking of Matplotlib is a figure that contains a graph users can customize. Matplotlib supports the typical two dimensional data alongside data of three or more dimensions. Besides plotting points in different dimensions, one can also draw surfaces of different styles. Matplotlib creates figures that visualize the data from the shapefiles in VBASE.

2.4 SQLite

A key component of a geographic information system is the database management. To store GIS data, a database is necessary, and in order to fulfill that requirement, SQLite is implemented. SQLite is a light embedded version of Structure Query Language (SQL), which is a relational database management system (RDBMS) [3]. An easy way of thinking of SQL's structure is as a collection of simple tables that have rows and columns. Each column represents a field of the given table and each row is an entry in those fields. This architecture is the foundation that makes up SQL databases. The biggest difference between SQLite and other SQL frameworks is that SQLite is an embedded database system, meaning that all of its content and functionalities remain on a local system instead of using a server. What makes an SQL database useful is the simple yet powerful structure. The framework is based on the usage of tables and relational keys that link said tables together. SQL may appear to be simplistic instead of a tool for complex, multilayered databases, but can be an intricate database system via the ability to link tables. The key to creating a multilayered and complex database is to take advantage of SQL being a relational system. Using foreign keys and references to identification numbers of other tables is a key concept of RDBMS in SQLite. SQLite enables VBASE to be a database that holds unique entries.

3 Software Development

VBASE was created using Python alongside the frameworks and APIs discussed in the background section 2. In order to achieve the goal of implementing a visual database for shapefiles, frontend and backend development for this application is necessary. Visualization is not possible without frontend implementations. The functions that allow the frontend to perform the necessary actions invoked by the user are what make the backend necessary as well. For this section, the frontend, backend, and the interface that brings the two facets together are explained in great detail. A Unified Modeling Language (UML) diagram of VBASE is shown in Figure 1.

3.1 Interface

All interaction between the frontend and backend occurred in an interface module, interface.py. As shown in Figure 1, the interface module is placed in the center of diagram. The reason for this is the interface is essentially the middle man between the frontend and backend.

Because of the interface, the frontend and backend do not directly interact with one

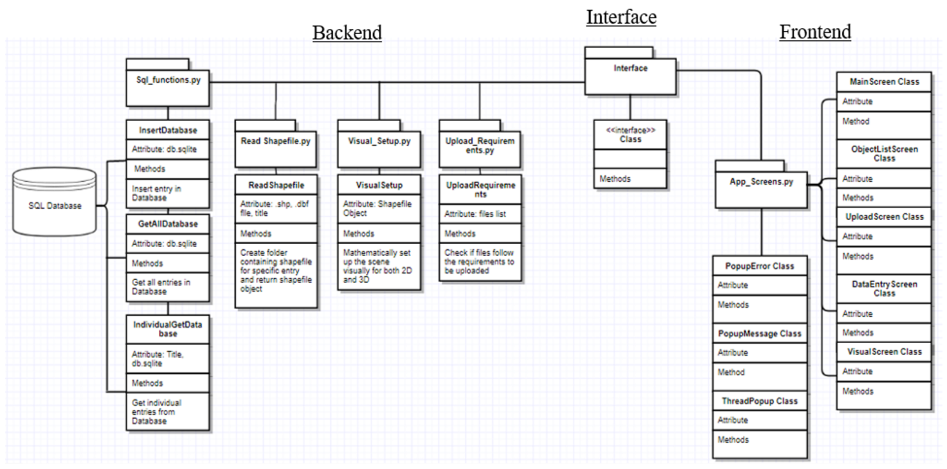


Figure 1: UML Diagram: Outlining Structure of Software

another as the only interaction either side of the application has is with an interface class. The design of this class is structured this way to avoid high dependency and coupling. High dependency and coupling make it difficult to isolate bugs in one's software, as functions are entangled with one another. This entanglement causes finding the source of an issues to be quite difficult. Developing the interface module this way, it allows for a smooth integration of the frontend and backend.

The interface class is compromised of a number of methods that perform multiple backend functionalities in order to collectively perform a certain task. These methods in the interface class are called by the frontend to handle a particular event occurring on the frontend's side. For example, when a button to visualize a particular profile entry in 3D is pressed, the method, `visual_screen_instance_3d(self, instance)`, is called to perform the backend runs the necessary functions to successfully performed the task, Figure 11.

3.2 Frontend

Kivy's framework is the driver of VBASE's frontend. It consists of one module, app_screens.py, and multiple classes that make up the side of the application the user interacts with.

3.2.1 App_screens.py

The App_screens.py module contains all Kivy related code. Classes that make up the screens for this application all inherit Kivy's screen class. The level of complexity and desired interaction of a screen depends on whether or not the screen's constructor is developed in either the classs constructor or in the .kv file. The app_screens module is designed this way for the sake of organization and maximizing different class's complexity in the constructor. If a screen is simple and its structure does not rely dynamic attributes, then the screen is developed in the .kv file. If the screen requires a set of dynamic data, the screen is constructed in the class's constructor.

3.2.1.1 IntroScreen Class

The IntroScreen Class is the first screen the user is introduced to at the start of the application. It contains a simple layout that is constructed solely in the .kv file. There is a button labeled "Continue", and when pressed, it prompts a transition to ObjectlistScreen. Figure 2.

3.2.1.2 ObjectListScreen Class

ObjectlistScreen Class is developed in the super class constructor of the ObjectListScreen class. This screen contains all profile entries present in the database. Each profile extracted from the database has a separate layout that contains a series of buttons that perform specified functions for that particular profile. This screen also contains a transition button to the upload screen. The entire structure of the screen can be seen in Figure 3.

3.2.1.3 UploadScreen Class

The UploadScreen Class is constructed in the .kv file. It contains an embedded file explorer for users to select files to upload to the database as shown in Figure 4. In order to view

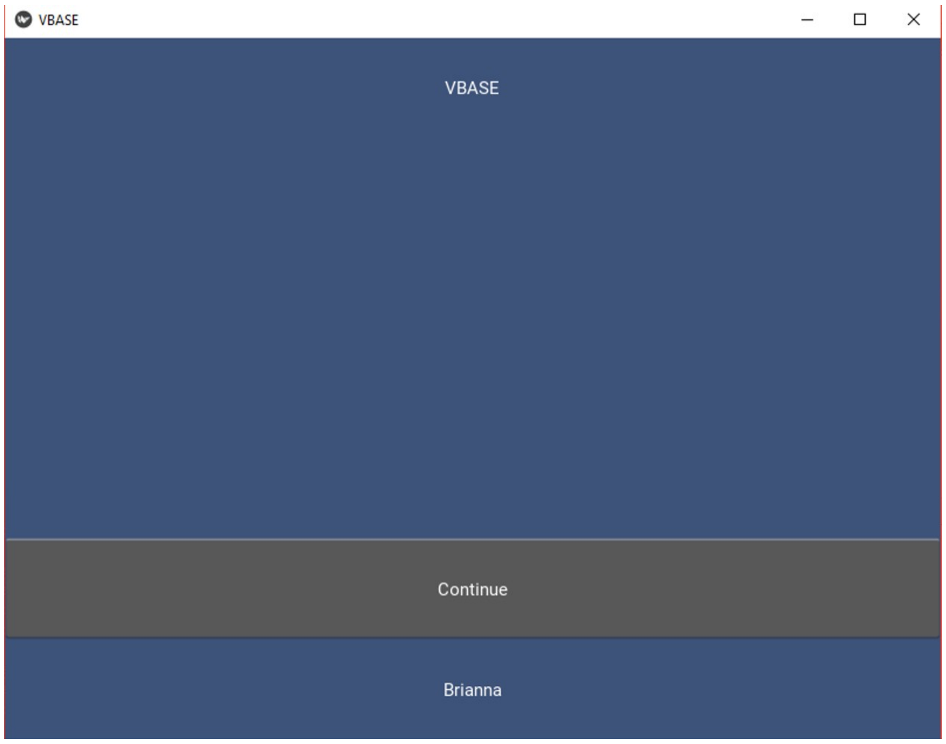
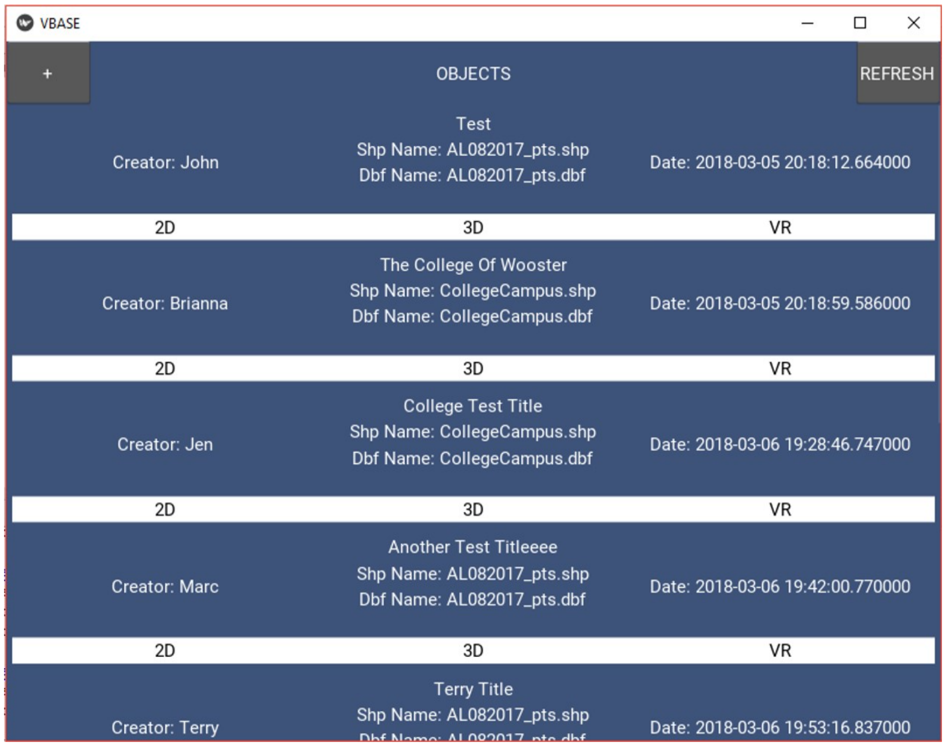


Figure 2: Screenshot of IntroScreen



Creator: John	Test Shp Name: AL082017_pts.shp Dbf Name: AL082017_pts.dbf	Date: 2018-03-05 20:18:12.664000
2D	3D	VR
Creator: Brianna	The College Of Wooster Shp Name: CollegeCampus.shp Dbf Name: CollegeCampus.dbf	Date: 2018-03-05 20:18:59.586000
2D	3D	VR
Creator: Jen	College Test Title Shp Name: CollegeCampus.shp Dbf Name: CollegeCampus.dbf	Date: 2018-03-06 19:28:46.747000
2D	3D	VR
Creator: Marc	Another Test Titleeee Shp Name: AL082017_pts.shp Dbf Name: AL082017_pts.dbf	Date: 2018-03-06 19:42:00.770000
2D	3D	VR
Creator: Terry	Terry Title Shp Name: AL082017_pts.shp	Date: 2018-03-06 19:53:16.837000

Figure 3: Screenshot ObjectScreen

current selected files, one can select the button labeled "See Selected Files", and a popup showing the current files will show up Figure 5.

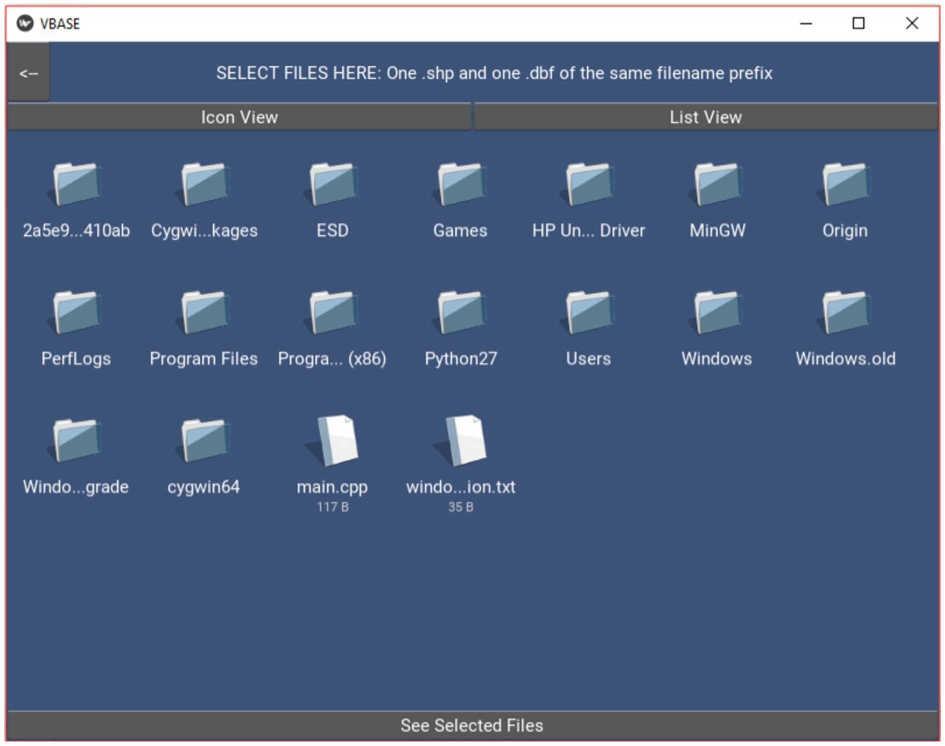
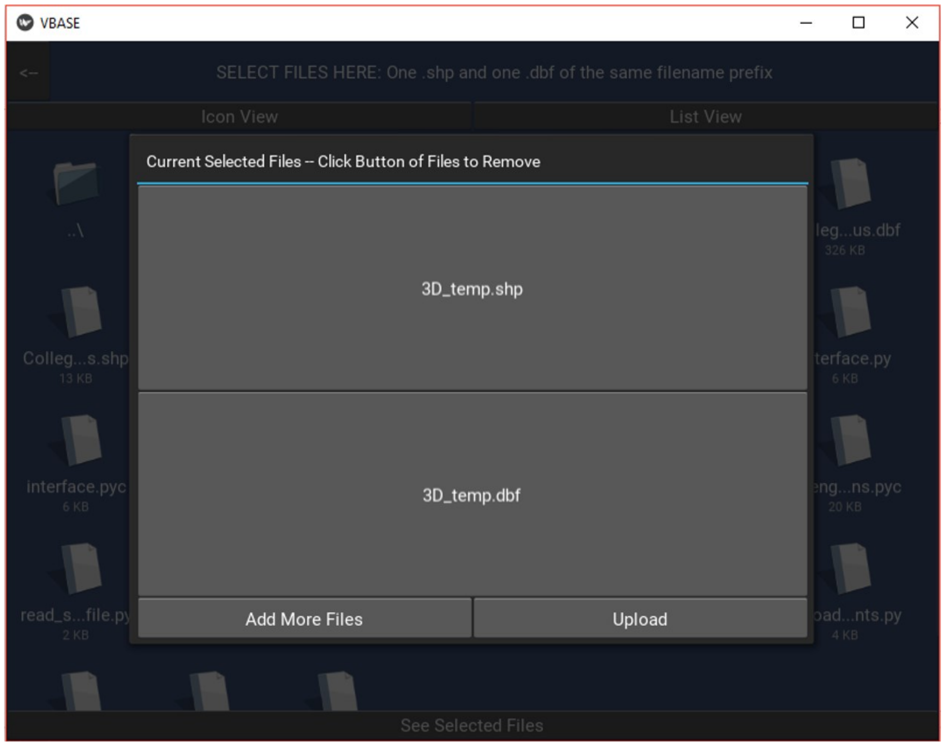


Figure 4: Screenshot of UploadScreen

3.2.1.4 DataEntryScreen Class

After the user has successfully selected files in the UploadScreen Class, a transition to the DataEntryScreen occurs. In this screen, the user enters the data that will be stored in the database. The screen contains a field for each of the following: creator, title, address, city, state, country, and zipcode Figure 6. There are two conditions that must be followed to successfully have data entered in VBASE. The creator, title, and country field must have an entry, and if the zipcode field has an entry, it must be an integer. Any failures to meet these conditions are met with a popup message informing the user to make changes in order to proceed to the next screen.

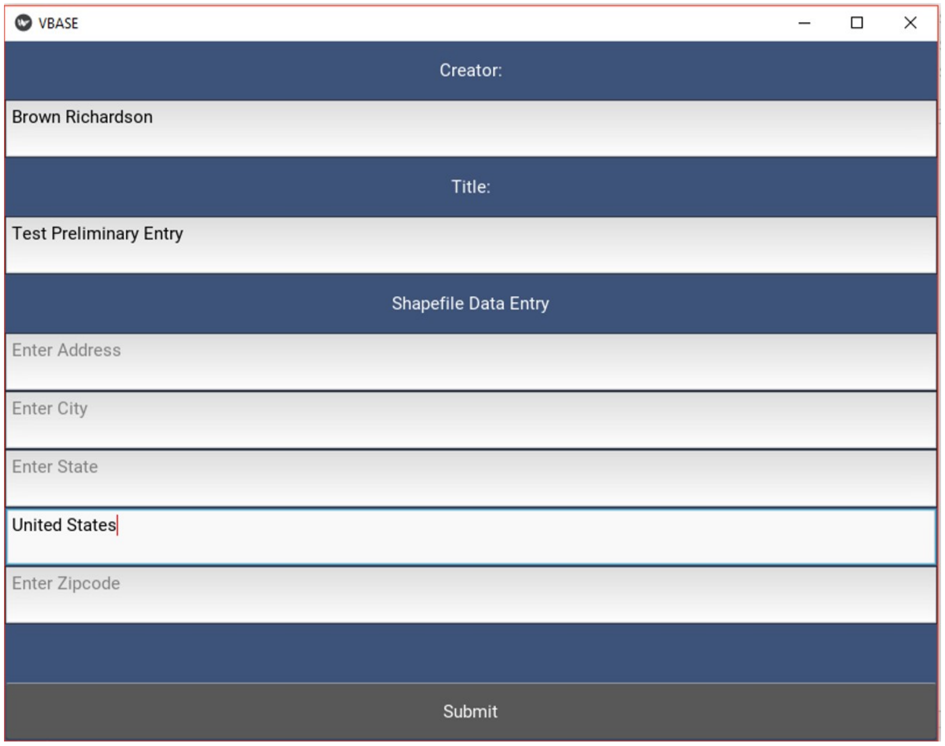


Current Selected Files -- Click Button of Files to Remove

3D_temp.shp

3D_temp.dbf

Figure 5: UploadScreen Screenshot - Popup to view selected files



Creator:

Brown Richardson

Title:

Test Preliminary Entry

Shapefile Data Entry

Enter Address

Enter City

Enter State

United States

Enter Zipcode

Figure 6: Screenshot of DataEntryScreen

3.2.1.5 VisualScreen Class

The VisualScreen Class is only accessible from the ObjectListScreen Class. As stated in section 3.2.1.2, each profile has a separate layout and in this layout there are buttons for both 2D and 3D that transition to the VisualScreen, which contains a Matplotlib widget that takes in a Matplotlib figure of the profile's shapefile in the specified dimension. An example of 2D and 3D implementation of a shapefile can be seen in Figures 7 and 8.

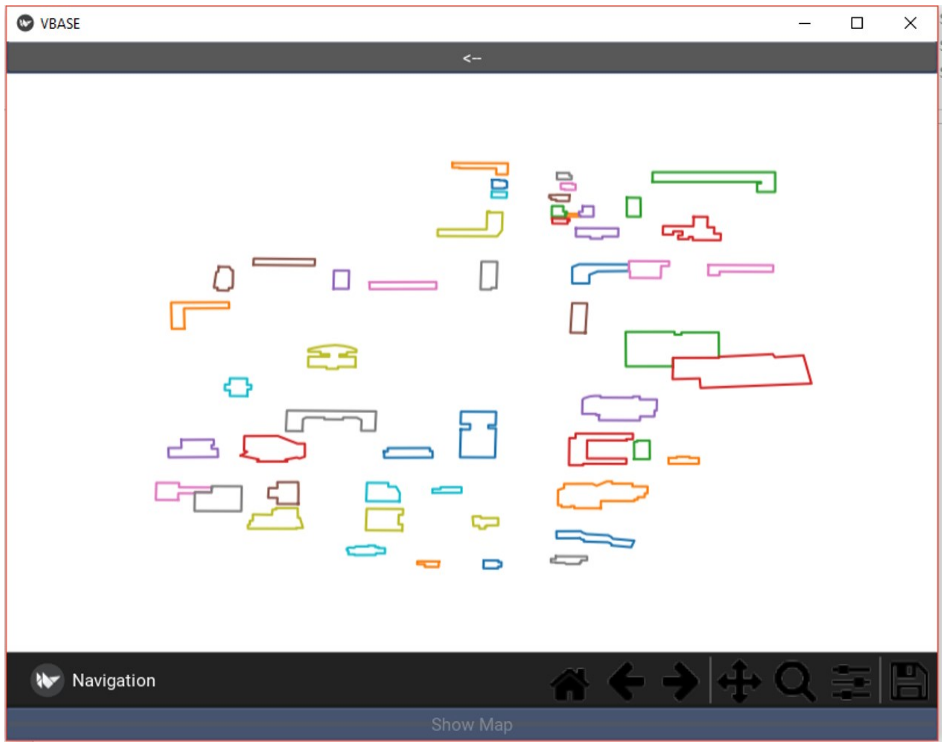


Figure 7: VisualScreen Screenshot - Visualizing data in 2D

3.3 Backend

In order for the app_screen module to perform all actions related to any functionality present in section 3.2, there must be an implementation present in the backend. This section explains each modules purpose and role in performing certain functionalities for VBASE.

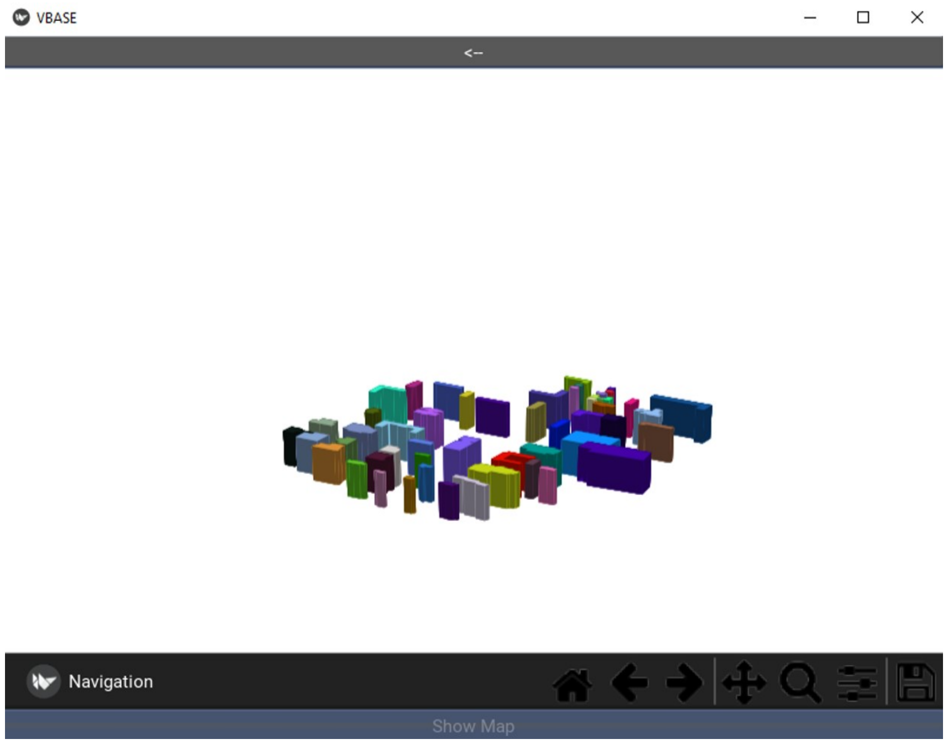


Figure 8: VisualScreen Screenshot - Visualizing data in 3D

3.3.1 Main.py

The Main.py module contains the driver method that builds the entire application. A screen manager is present in the module. The screens developed in the app_screens.py are added to this screen manager. The screen manager is a widget in itself that manages all the screens for the application. If a Kivy application is to have more than one screen, a screen manager must be used to handle the transitions between each screen. After all desired screens are added to the screen manager, the Python interpreter looks builds the application by returning the screen manager in the build method.

3.3.2 Upload_requirements.py

Upload_requirements.py contains functions that check if the selected files follow the requirements to be uploaded and visualized in the database. Selected files must follow the format of a shapefile as explained in section 2.1.3, that is, files must have the same filename prefix and be .shp and .dbf files. Overall, the main purpose is to have a boolean value returned to reflect whether one's file selections fulfill the upload requirements or not.

3.3.3 Sql_functions.py

The schema of VBASE contains four tables: Profile, Object, Location, and Files. The relationship between the four tables can be viewed in Figure 9. Even though the schema of VBASE can be implemented in Python, it is important to note the schema, an .sql file and the database file, db.sqlite, were both created outside the Python environment. Sql_functions.py handles all SQLite related functions. Any interactions with the database are implemented in this module, as it interacts with the database .sqlite file, db.sqlite, containing the database schema. This module consists of three classes: a class to handle inserting data into the database file, a class to extract individual entries, and a class that returns all data entries. Each class contains functions to insert or select data involving all tables.

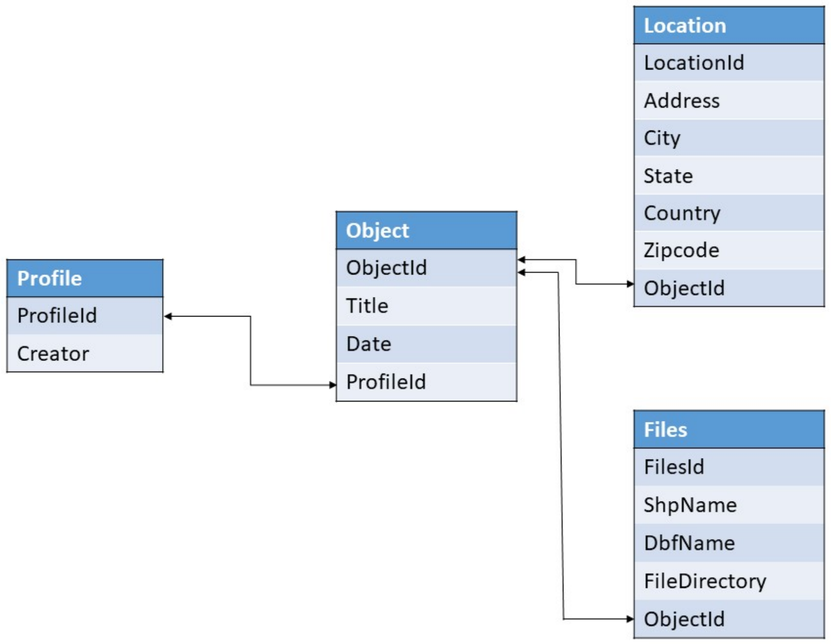


Figure 9: VBASE database schema

3.3.4 Read_shapefile.py

Read_shapefile.py's purpose is to read shapefiles and create directories. A folder that will contain the shapefiles is created in the main upload directory, which is a folder titled Uploaded_Shapefiles in the current working directory. Once shapefiles are processed, a subfolder is created and designated the same title the user selected for the given profile on the DataEntryScreen. This data is extracted from the object table from the SQLite database. This module also contains a method, `read_files(shp_file, dbf_file)`, that reads the reads the shapefiles and returns a shapefile reader object specified to the shapefile being passed to the method Figure 12.

3.3.5 Visual_setup.py

The module, visual_setup.py, contains functions that handle the mathematics required to visualize the shapefile in 2D and 3D. In visual_setup.py, Python Shapefile (Pyshp), a shapefile reading and writing library for Python, is used to extract the information from the desired shapefile. It is important to know that the information that can be extracted from the shapefile also includes all shape types. A number of methods in the visual_setup.py module use Pyshp to perform certain functionalities such as converting the shapefiles from 2D to 3D and vice versa and drawing these files onto a Matplotlib figure. After all necessary methods are excuted and the visual setup is complete, the figure containing either the 2D or 3D data is passed to the interface to be displayed on the visual screen.

4 Results and Analysis

By implementing the modules, classes, methods, and attributes discussed in section 3, the creation of VBASE was a result. The application allows users to upload shapefiles, visualize them both in 2D and 3D, and see all entries in the database. Figures 2 - 8 contain screenshots of the final product of VBASE.

A key reason for writing this paper is to inform readers of the process in developing VBASE, which includes an analysis of the results. The following sections inform readers of precautions, mathematical implications, and possible directions and improvements for future development.

4.1 Kivy: Strengths and Weaknesses

Kivy is a great tool to develop natural user interfaces (NUIs). It is an effective way to get one's application up and running fairly quickly, but as with most frameworks, Kivy does have some weaknesses one should consider when using this framework. The community and support for Kivy dwindles as the complexity of an application increases. Documentation and examples for basic and simple implementations of Kivy are fairly easy to find and understand. Developers and user support groups are available to answer most questions, but as stated previously, the support is limited. Using Kivy's graphics API proved to be quite difficult, as documentation appeared to be fairly vague and required some chicanery to figure out. That being said, Kivy is a continually developing framework that recently implemented a 3D engine, called Kivent, and is an easy way to quickly develop an NUI.

4.2 Visual Implementations

After developing the visual implementations for VBASE and reflecting on the process, a few issues were encountered. As result, improvements could be made.

4.2.1 3D Visualization Improvements

When visualizing a shapefile's data in 3D, a temporary MULTIPATCH shapefile is created in order to create 3D objects. This temporary shapefile takes the shapes of shapefile being visualized and adds z-axis components. The temporary shapefile is a copy of the original shapefile with z-axis features. In Figure 10, it shows that all the original shapefiles shapes except MULTIPATCH shapes are copied to a temporary MULTIPATCH shapefile. The reason for this is that copying MULTIPATCH shapes proved to be quite an issue. When copying all parts of a MULTIPATCH shape to another shapefile, the shapes were becoming distorted and not accurately displaying the shapefile due to some unknown underlying issue.

As a solution to this issue, it was decided to simply draw the MULTIPATCH shape directly to the matplotlib figure versus copying it to a temporary file. It was realized thhat the purpose of creating the temporary shapefile was to convert the 2D shapes to MULTIPATCH, and considering the original shape is already a MULTIPATCH shape, it was unnecessary to copy this shape to the temporary shapefile.

After developing a solution to this issue, it was realized that the creation of a temporary shapefile was not the most optimal approach and could be improved. When visualizing shapefiles in 3D in one's own visual GIS database, instead of having to create a temporary shapefile, it is highly suggested to draw the shapes directly to the matplotlib figure using the mathematical implementations used to create the temporary shapefile. This will cut out the middle man of a temporary file being needed and will improve the overall runtime in visualizing shapefiles in 3D.

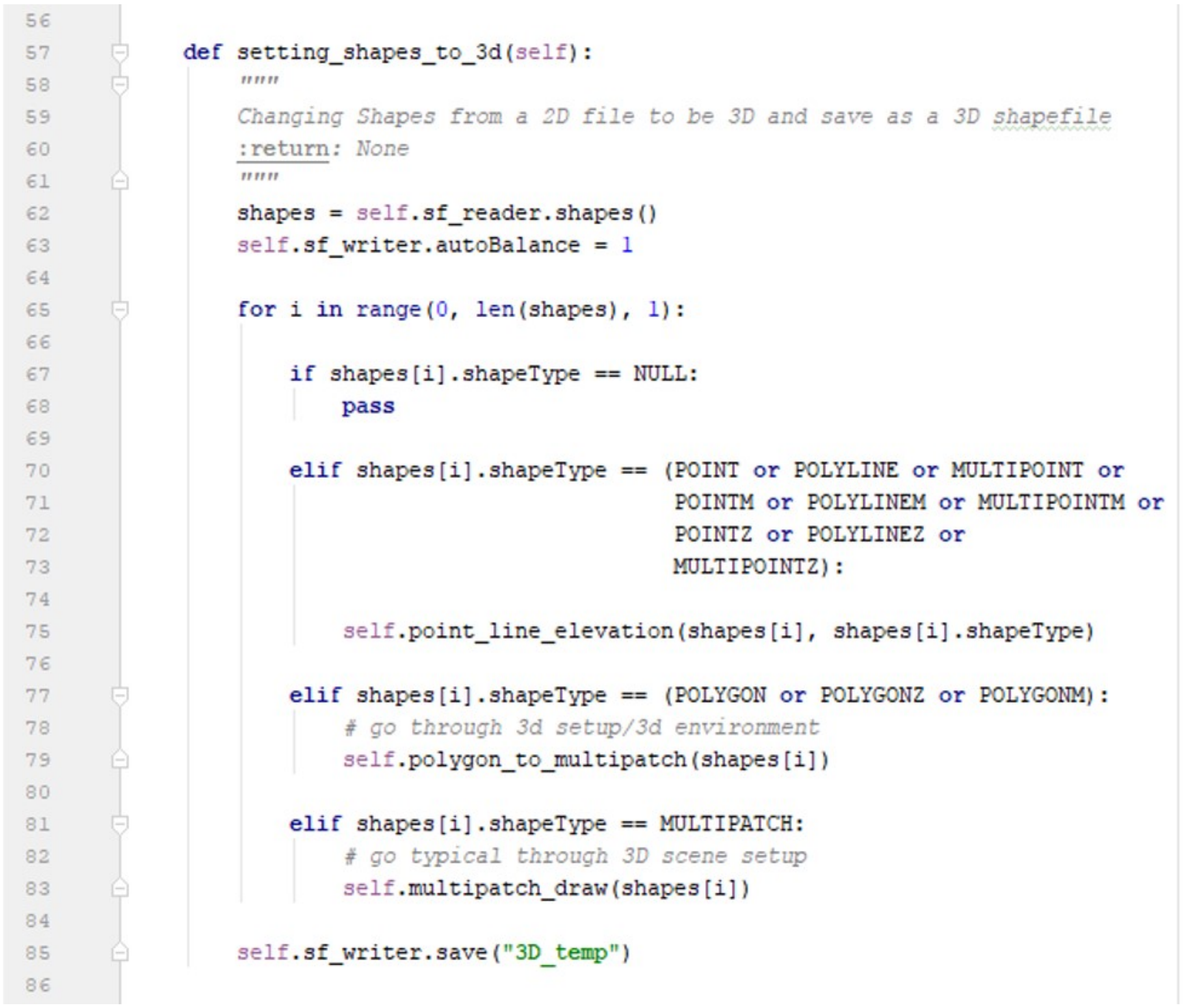
4.2.2 Matplotlib: Planar and Non-Planar Surfaces on the Same Figure

Another issue encountered in the visual implementations is the drawing of planar surfaces on matplotlib figures. When reading shapefiles with MULTIPATCH shapes, the `setting_shapes_to_3d()` method (Figure 10) in visual_setup.py analyzes whether or not the shape is a planar surface on the z-axis. In early stages of developing the visual implementation, it was noted that matplotlib will only successfully draw planar, or flat, surfaces (meaning all z-coordinates are of the same value) alongside a non-planar surface only after the non-planar surfaces are drawn first. If drawn in the opposite order, the planar surface would not be drawn to the figure. This is an issue because if a shapefile stores planar surfaces first and non-planar surfaces afterwards, then the planar surfaces would not be drawn.

5 Conclusion

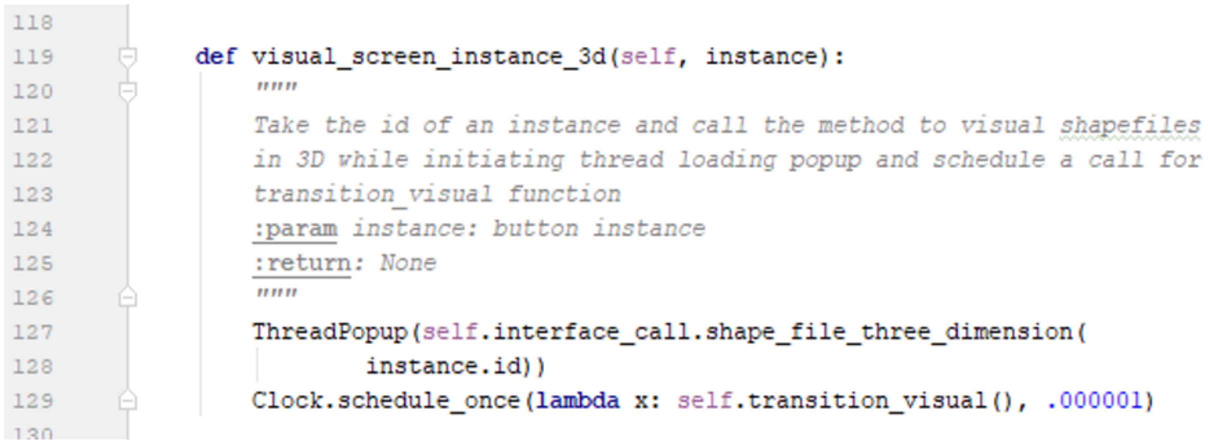
When GIS first came out, it was seen as a tool only for geographers or groups of individuals looking to interpret spatial data. It was an exclusive system geared towards a specific crowd. Today, GIS has evolved and reached disciplines all across the spectrum. The development of VBASE allows users to visualize spatial data without having an intricate or professional understanding of GIS data by simply uploading a shapefile. VBASE's current state includes visualizing data in 2D and 3D; the next step is to potentially implement virtual reality (VR) visuals of shapefiles. The main premise of this paper is to provide readers an in-depth explanation of the process in developing one's own GIS, as well as to contribute to the community of computer science and GIS interdisciplinary development. The evolution of such development continues to create more accessible and straightforward tools such as VBASE.

6 Appendix



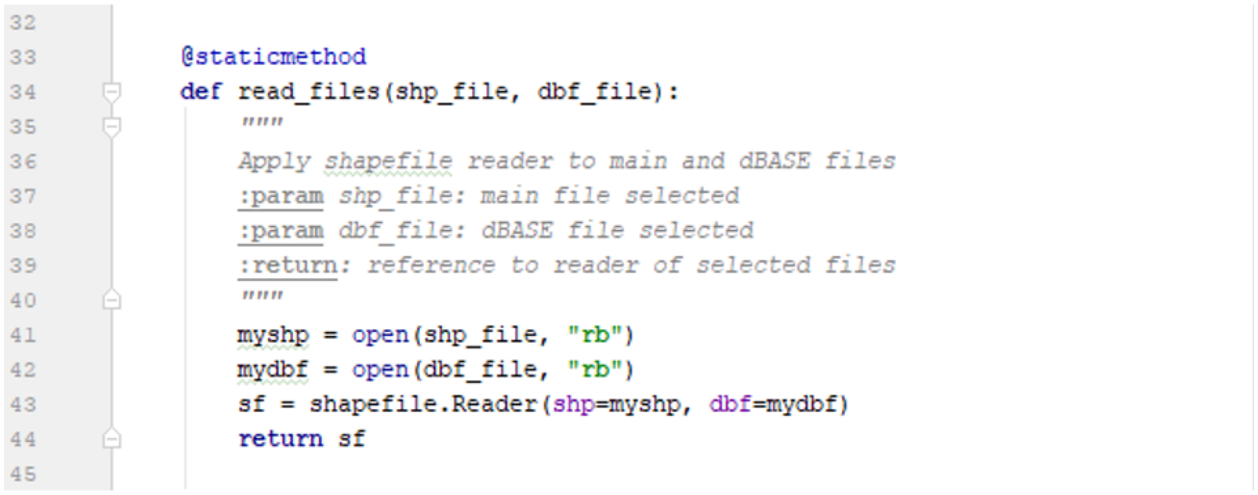
```
def setting_shapes_to_3d(self):
    """
    Changing Shapes from a 2D file to be 3D and save as a 3D shapefile
    :return: None
    """
    shapes = self.sf_reader.shapes()
    self.sf_writer.autoBalance = 1

    for i in range(0, len(shapes), 1):

        if shapes[i].shapeType == NULL:
            pass

        elif shapes[i].shapeType == (POINT or POLYLINE or MULTIPOINT or
                                      POINTM or POLYLINEM or MULTIPOINTM or
                                      POINTZ or POLYLINEZ or
                                      MULTIPOINTZ):

            self.point_line_elevation(shapes[i], shapes[i].shapeType)

        elif shapes[i].shapeType == (POLYGON or POLYGONZ or POLYGONM):
            # go through 3d setup/3d environment
            self.polygon_to_multipatch(shapes[i])

        elif shapes[i].shapeType == MULTIPATCH:
            # go typical through 3D scene setup
            self.multipatch_draw(shapes[i])

    self.sf_writer.save("3D_temp")
```

Figure 10: `setting_shapes_to_3d()` method to setup 3D scene



```python
    def visual_screen_instance_3d(self, instance):
        """
        Take the id of an instance and call the method to visual shapefiles
        in 3D while initiating thread loading popup and schedule a call for
        transition_visual function
        :param instance: button instance
        :return: None
        """
        ThreadPopup(self.interface_call.shape_file_three_dimension(
                instance.id))
        Clock.schedule_once(lambda x: self.transition_visual(), .000001)
```

Figure 11: `visual_screen_instance_3d()` method to initializes 3D visuals



```python
    @staticmethod
    def read_files(shp_file, dbf_file):
        """
        Apply shapefile reader to main and dBASE files
        :param shp_file: main file selected
        :param dbf_file: dBASE file selected
        :return: reference to reader of selected files
        """
        myshp = open(shp_file, "rb")
        mydbf = open(dbf_file, "rb")
        sf = shapefile.Reader(shp=myshp, dbf=mydbf)
        return sf
```

Figure 12: `read_files()` method to return shapefile reader object

7 Bibliography

References

[1] Research guides: Mapping and geographic information systems (gis): What is gis?

[2] Esri shapefile technical description. `http://downloads.esri.com/support/whitepapers/mo_/shapefile.pdf`, 1998.

[3] Grant Allen, Michael Owens, Ohio Library, and Information Network. *The definitive guide to SQLite*. Apress, New York, 2nd;2;second; edition, 2010;2011;.

[4] Joel Lawhead. *Learning Geospatial Analysis with Python - Second Edition*. Packt Publishing, GB, 2 edition, 2015.

[5] Duncan M. McGreggor, Ohio Library, and Information Network. *Mastering matplotlib: a practical guide that takes you beyond the basics of matplotlib and gives solutions to plot complex data*. Packt Publishing, Birmingham, UK, 2015.

[6] Mark Vasilkov, Ohio Library, and Information Network. *Kivy blueprints: build your very own app-store-ready, multi-touch games and applications with Kivy!* Packt Publishing, Birmingham, UK, 2015.