UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING

3D Editor Tool Using Vpython

Supervised by: Dr. Richard Conway

STUDENT NAME: DESMOND UMEBEH C.

STUDENT ID: 19014465

COURSE TITLE: COMPUTER AND COMMUNICATION SYSTEM

August 28th 2020

Table of Contents

LIST OF FIGURES

Abstract

Visual python is a library which extends Python used for 3D simulations. The Vpython developers have done a great job simplifying codes syntaxes, to put it in perspective, complex operations which will take naturally from the finest refined codes 10 – 15 lines of statements might just take a single statement on Vpython. However, to carry out a list of 3D simulation activities on Vpython, one will need to punch in codes. These codes might be simplified but are not idle when carry out large or complex projects or a task which may require routine sub activities like recreating a specific object with specified properties by re-typing codes or copying and pasting. This study is aimed at exploring the objects already defined on the Vpython library and re-using it to create an editor tool which will provide to the user a level of interactivity, encapsulating the codes behind the operation behind the scenes. Users could dynamically redefine 3D objects to suit their needs without entering any codes only through the interactions on the tool GUI. Also, it serves as good place for non-coders to manipulate objects not bothering about the logic involved in producing certain results.

However, the study and implementation of the 3D editor tool is a proof of concept to the application of Vpython in designing an editor tool.

The tool design is primarily focused on implementing beneficial static features such as intuitive object rotation using mouse interactions, the functionality of allowing users to get a sense of objects position and scale on the canvas, introduce interactions using widgets, mouse and keyboards in manipulating 3D objects, objects merging, duplication and the snap to grid feature.

Design results show that developing a tool over vpython and glowscript which serves as an engine to allow users easily run and compile vpython applications over the web without installation might come with limitations such as limitations on file operations and restrain on certain Python keywords.

This report will provide a detailed process involved in the design of the tool and proposals to overcome the challenges presented by Vpython.

Chapter 1 – Introduction

Vpython is a visual python programming language written in c++ and python for the primary purpose of simplifying 3D object creation, it provides semi-simplified syntaxes to the user to create objects such as a box, cone, sphere, cylinder and perform more complex functions such as simulating the electromagnetic field on a charged bar. It is an open source application available on Macintosh, Windows and Linux[1]. Vpython has numerous applications; however, my research indicates that its application lies predominately within the academic and research scene, an instance in academic usage is the design of an illustration program in a physics class and on the other hand, researchers use it to model systems and visualize data in 3D[2].

The objective of this project is to illustrate a proof of concept that the available resources/libraries on Vpython could be explored and used in the design of a 3D editor tool. The tool will provide an interactive functionality which allows users to model 3D scenes on a web-based environment which will encapsulate the operation behind the scenes. An application of Glowscript (www.glowscript.org), a free web browser simulation environment will be used in the course of the tool development.

The contents of this report will illustrate the step process taken in creating the 3D editor tool with functionalities that allow the user to create a 3-dimensional object and manipulate these objects properties on the scene.

Chapter 2 – Literature Survey

Evidently, programing skills are becoming increasingly more important in physics and other STEM fields. Existing tools for teaching physics and engineering using computational modeling requires that the students should already have a foundation of programming, thus narrowing students learning opportunities [4]. A 3D editor tool with Vpython will provide a way to encourage developers into exploiting the potential of designing a high performance web-based 3D modelling tool and in the STEM field, allow students and educators with algorithmic thinking without an extensive pre-requisite knowledge of syntaxes to carry out some basic illustrations as the tool will be equipped with an interactive graphical user interface.

Similar projects on creating a 3D editor tool using Vpython was carried out by researchers Cody Blakeney, Michael Dube and Hunter Close at the Texas University, they did create a visual editor for Vpython. This was a prototype for a visual programming environment that allows students to create physics simulations utilizing the open source projects Vpython and Blockly[4].

Chapter 3 – Glowscript

Glowscript is best described as an easy-to-use web-based application for creating and simulating 3D animations. It permits users to perform task on Vpython through the aid of its complier RapydScript. Clearly, one could say that RapydScript is the core to operating Vpython on Glowscript.

GlowScript makes it easy to write programs in JavaScript, RapydScript (a Python look-alike that compiles Python to JavaScript), or VPython (which uses the RapydScript compiler) that generate navigable real-time 3D animations, using the WebGL 3D graphics library available in modern browsers (with modern GPU-based graphics cards). For example, the below syntax is a complete program that creates a 3D canvas in the browser, displays a white 3D cube, creates default lighting, places the camera so that the cube fills the scene, and enables mouse controls to rotate and zoom the camera: [5]

```
box()
```

The key point is that lots of sensible defaults are built into the GlowScript library. You can of course specify the canvas size, the color and other attributes of the objects, the direction of the camera view, etc. [5]

GlowScript was inspired by VPython. The project began in 2011 by David Scherer and Bruce Sherwood. Originally, programs had to be written in JavaScript, but in November 2014 it became possible to use Python, thanks to the RapydScript Python-to-JavaScript compiler created by Alex Tsepkov. GlowScript is now using a later version, RapydScript-ng developed by Kovid Goyal[5].

Now, let’s take a moment to discuss RapydScript.

RapydScript (pronounced 'RapidScript') is a pre-compiler for JavaScript, similar to CoffeeScript, but with cleaner, more readable syntax. The syntax is almost identical to Python, but RapydScript has a focus on performance and interoperability with external JavaScript libraries. This means that the JavaScript that RapydScript generates is performant and quite close to handwritten JavaScript. [6]

RapydScript allows you to write your front-end in Python without the overhead that other similar frameworks introduce (the performance is the same as with pure JavaScript). To those familiar with CoffeeScript, RapydScript is like CoffeeScript, but inspired by Python's readability rather than Ruby's cleverness. To those familiar with Pyjamas, RapydScript brings many of the same features and support for Python syntax without the same overhead. Don't worry if you've never used either of the above-mentioned compilers, if you've ever had to write your code in pure JavaScript you'll appreciate RapydScript. RapydScript combines the best features of Python as well as JavaScript, bringing you features most other Pythonic JavaScript replacements overlook[6].

3.1 Architecture

The diagram on fig1 shows the architecture of vpython used on glowscript.org, this comprises of two core unit which are the Server(Datastore) and the Browser.

The Datastore is made up of a server code, these codes is written in python and serves data to the webpage, the Datastore also provides data storage to the cloud.

The Browser consist of four sub-units namely:

1. Rapydscript library
2. Glowscript graphics library
3. WebGL
4. Web page

The RapydScript library converts Python code to Javascript, this is important as Python code does not have the functionality to run on browsers but Javascript does. The javascript is moved on to the glowscript graphics library which in turn communicates with WebGL which in turn is passed to the webpage and displayed to the user. Essentially, The Javascript code cannot be ran on the browser without the browser built in WebGL 3D library.

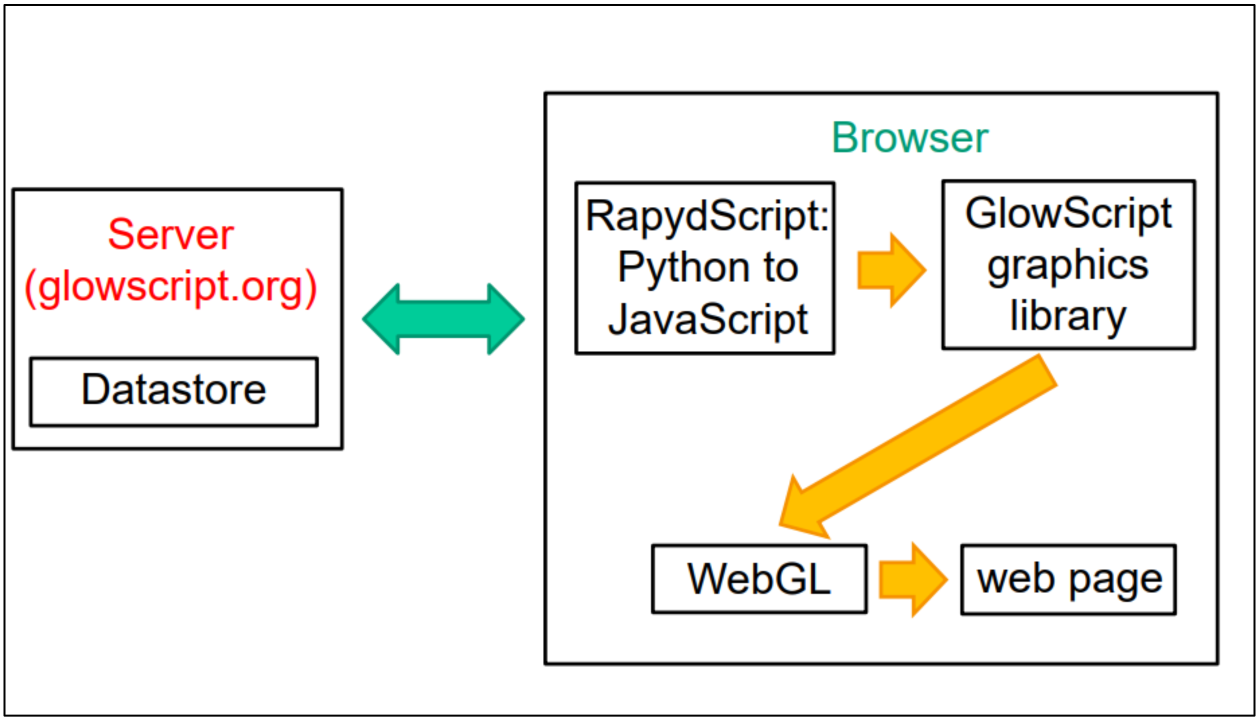

Fig1: Vpython – Glowscript Architecture[11]

3.2 Program Compilation on Glowscript

This section looks at the compilation steps involved in getting the user vpython(python) code into an executable program on the web browser.

The process starts with the user code written in python which goes through a series of pre-processing which is a follow-up step. Instances of activities that go on during pre-processing includes syntax check, identifying function names and where there been called, Python source line number insertion, handling of import statements. This step is necessary to ensure that your program can run at execution time. A validated code will then be passed on to Rapydscript which provides the Javascript equivalent to the code, these codes now Javascript is post-processed. The post-processing actions include insertion of preamble to Javascript code, redefining objects, and various other adjustments where needed. On completion of these set of sub-processes, the output is moved on to operator overloading. During the operator overloading phase, additional data to the code which will serve as references to other libraries that will be used at execution time will be attached. The final process gives you your executable program.

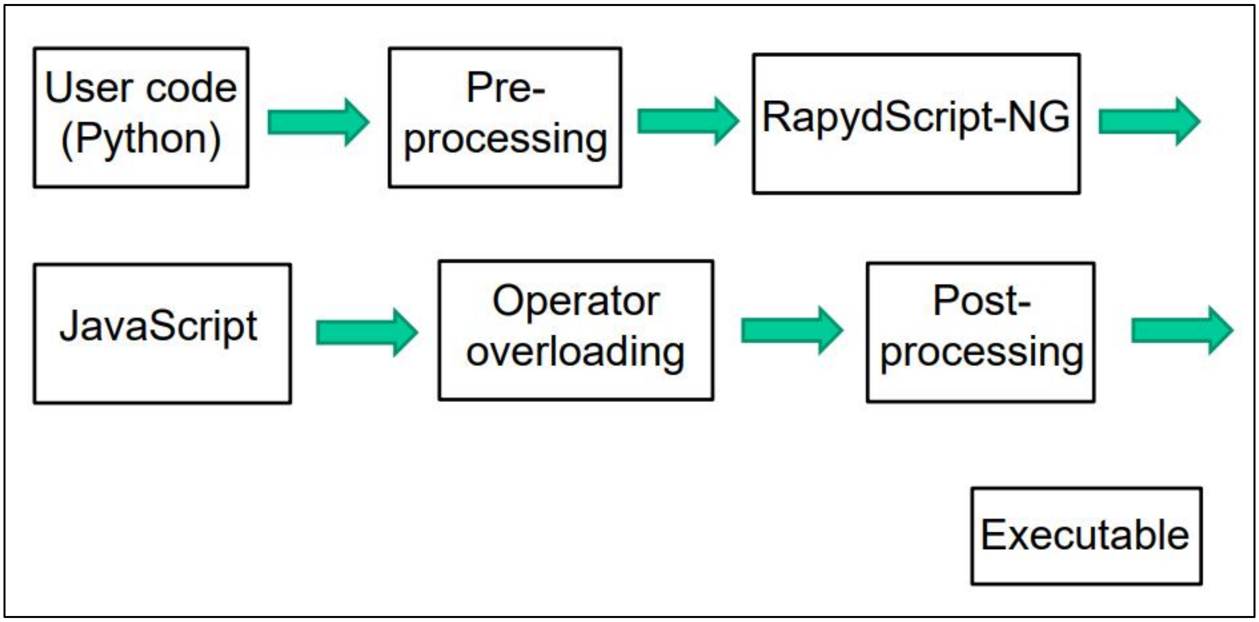

Fig2: Vpython-Glowscript Compilation Process[11]

Chapter 4 – The Concept

The conceptual diagram on fig3 depicts each concept that will be built into the tool to actualize it's 3-dimensional editing function. These set of concepts represents some of the core operations of a professional editor tool and will go a long way in verifying if vpython libraries could be explored in designing a 3D editor. Details relating to the design and implementation of each concepts exploiting the features/objects on vpython will be treated on this report. I will go ahead and briefly explain these units/features.

1. **3D Grid** – The grid provides a work area and a guide to identify the position of an object within the 3-dimensional space.
2. **3D Axis –** This will represent the coordinate axes and will define points along the x,y,z dimensions.
3. **Object Attribute Modification –** A feature that enables user to dynamically modify certain parameters relating to an object.
4. **Read/Write Operation –** This will provide users the functionality of loading file and writing to a file on the local computer.
5. **3D Object Trackline –** Dynamically tracks the object position along the x,y,z axes during position updates.
6. **Widgets –** Introduces an interactive interface, users will have access to some functions of the tool through these widgets. Below is a set of widgets provided on the vpthon library.
    a. Checkbox
    b. Menu
    c. Sliders
    d. Buttons
7. **Rotation –** Allow users to rotate 3D object at an angle in the three-dimensional space.
8. **Grid Snap –** This will allow objects to precisely snap to the grid intersections or points when moved along the x,y,z plane.

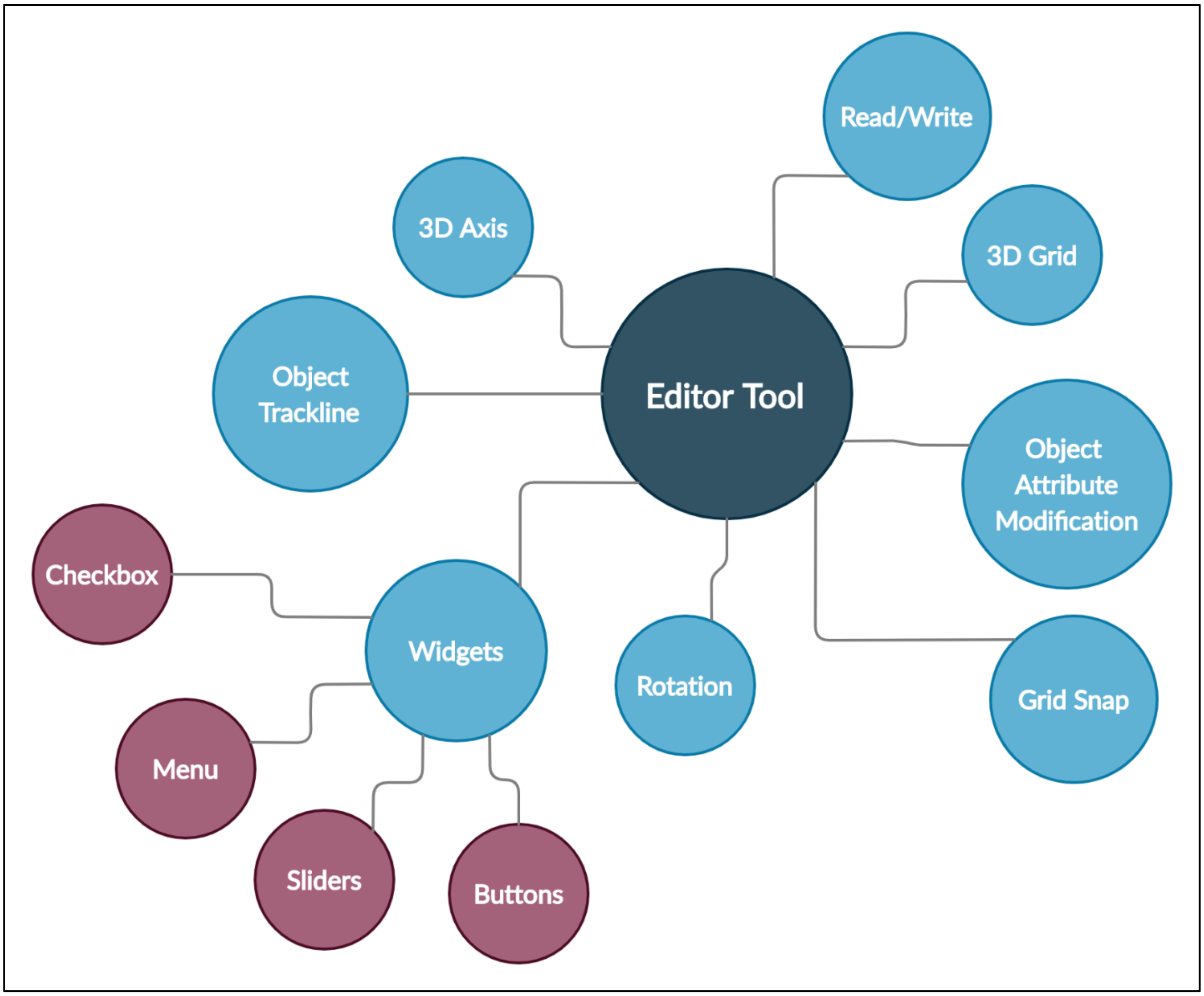

Fig3: Tool Conceptual Diagram

Chapter 5 – Simple 3D Editor: A Demonstration on Mouse/Buttons Interaction

As is applicable to most software applications, creating a 3D editor tool using Vpython is going to take a lot of interactions between the user and the tool. Basic functionalities that link up the user algorithm or intended activity will be established through mouse/button interactions.

Creating a simple program to demonstrate these interactions is a key step to understanding how the available list of functions on the Vpython library can be manipulated to developing a 3D editor tool. This rudimentary demonstration gives the user a perception of creating either a 3D box object, sphere object or some other object and manipulating their position along the x,y,z axis on the canvas without inputting codes.

Figures 6,7,8 shows the flow chart of the program and the corresponding source code entered on Glowscript. The flow chart (fig6) indicates the 5 major processes involved on this simple demonstration, namely:

1. Is Object Button Clicked? – Program will display an empty 3D scene when initiated and will remain in this state until an object button is pressed (see Fig9 for program interface).
2. Display Object – This process will be initiated once an object button is pressed, the respective binding functions shown below will be called and an object with the listed default attribute will be displayed on the scene. Function can be called at any time to create a new object.

```
#Display a box on the scene
def showBox(myBox):
   box(pos=vector(0,0,0), size=vector(2,2,2),visible = True, pickable = True)

 #Display a sphere on the scene
def showSphere(mySphere):
   sphere(pos=vector(0,0,0),radius= 0.5, visible = True, pickable = True)

#Display a cylinder on the scene
def showCylinder(myCylinder):
   cylinder(pos=vector(0,0,0),radius= 0.5, visible = True, pickable = True)
```

Fig4: Syntax to Create 3D Objects with Specific Attributes

3. Interrogate if mouse pointer is on an object: This process will run only if "mousedown" event is TRUE (mouse right button is clicked) and if the pointer position lies over an object, however, an abrupt interruption will occur once "mouseup" event becomes TRUE. The "mousedown" event will enable the drag function and will update an object position if mouse pointer is placed on that certain object.
4. Turn-on object drag switch & obtain the mouse position: This will be enabled once "mousedown" is TRUE and the mouse pointer points to an object. Mouse position along the x,y,z axis will be collected and assigned to a variable.

```
#This function will turn on the mouse drag and assign the mouse position on the scene to displayed #object.
#This will be true if mouse pointer is placed over an object on the scene.
def dragTrue():
    global drag
    global dragObj

    if  (scene.mouse.pick != None):
        drag = True
        dragObj = scene.mouse.pick
```

Fig5: Simple Program Demonstrating Mouse Interaction

5. Update object position: The object position will be modified to the location and stored on variable "dragObj".

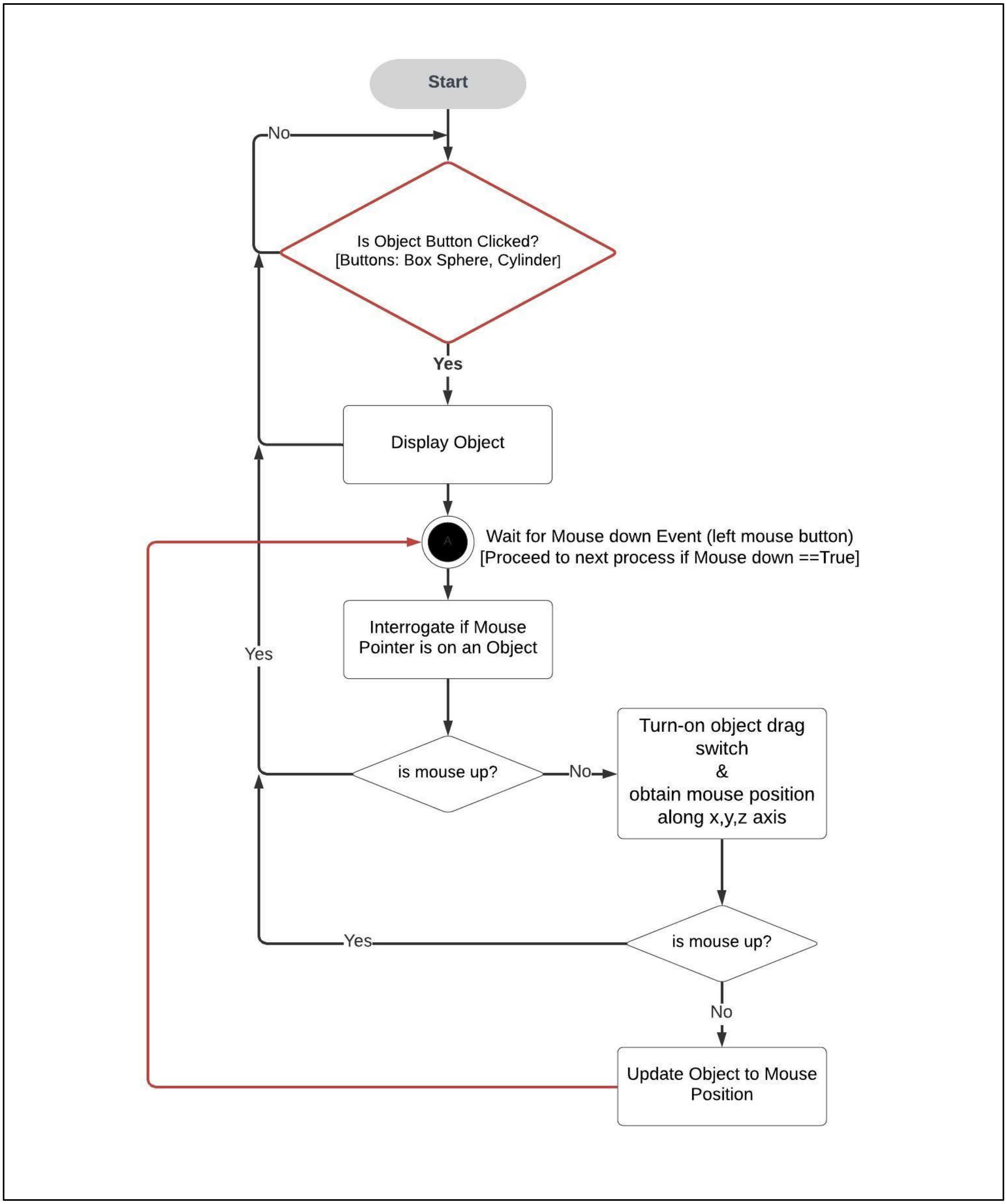

Fig6: Program Flow Chart

```
from vpython import*

scene.caption = "A demonstration on mouse/button interaction"

#A list of called functions if buttons “addBox”, “addSphere” or “addCylinder” are pressed by the user.

#Display a box on the scene
def showBox(myBox):
    box(pos=vector(0,0,0), size=vector(2,2,2),visible = True, pickable = True)

 #Display a sphere on the scene
def showSphere(mySphere):
    sphere(pos=vector(0,0,0),radius= 0.5, visible = True, pickable = True)

#Display a cylinder on the scene
def showCylinder(myCylinder):
    cylinder(pos=vector(0,0,0),radius= 0.5, visible = True, pickable = True)

#Create buttons and display on scene.title.
addBox = button(text="Box", pos=scene.title_anchor, bind = showBox)
addSphere = button(text="Sphere", pos=scene.title_anchor, bind = showSphere)
addCylinder = button(text="Cylinder", pos=scene.title_anchor, bind = showCylinder)


drag = False
dragObj = None
X=vector(0,0,0)
```

Fig7: Extended Code on Mouse Interaction

```
#This function will turn on the mouse drag and assign the mouse position on the scene to displayed
#object.
#This will be true if mouse pointer is placed over an object on the scene.
def dragTrue():
    global drag
    global dragObj

    if  (scene.mouse.pick != None):
        drag = True
        dragObj = scene.mouse.pick

#Turn off  mouse drag.
def dragFalse():
    global drag
    drag= False

#Binding function: Call dragTrue function on the event the mouse left button is pressed down.
scene.bind("mousedown", dragTrue)
#Binding function: Call dragFalse function on the event the mouse left button is pressed down.
scene.bind("mouseup", dragFalse)


 #Update object position on the scene to the mouse position.
def changeObjPos(displayObj):
    if drag :
        X=scene.mouse.pos
        displayObj.pos=X

#Loop statement to interrogate if drag function is true and update object location
while True:
    rate(100)
    changeObjPos(dragObj)
```

Fig8: Extended Code on Mouse Interaction

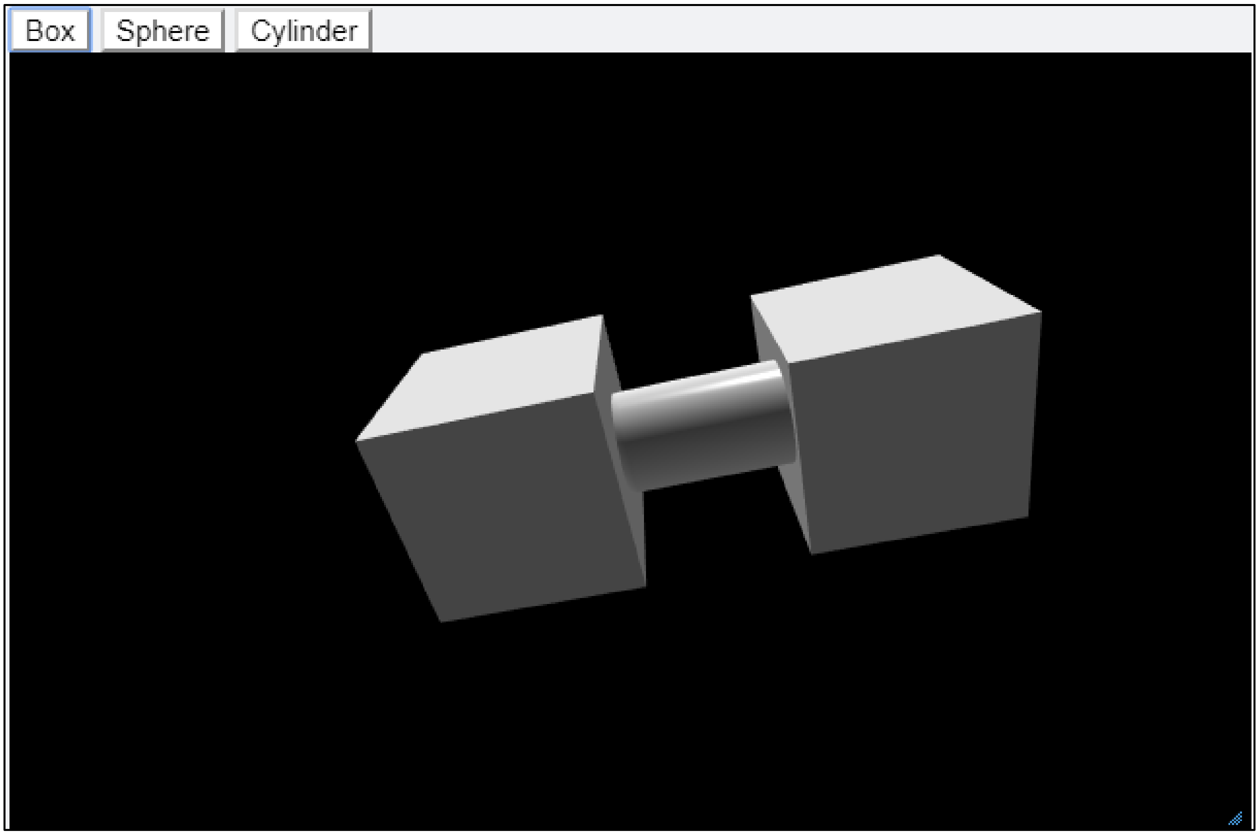

Fig9: Scene showing an object created using the buttons and mouse interaction.

Chapter 6 – 3D Tool Design and Implementation

To improve user visualization of the object on the scene, we will implement a 3-dimensional grid, the grid will help align objects and aid in visualizing the distance between them. Objects on the scene will only exist within the grid scope, meaning that all objects on the scene will be bounded to the 3-dimensional grid.

In this chapter we will discuss the grid design, object track line design, the snap to grid, object rotation and other tool features implementation, but before we do, let's briefly discuss the box object. Designing the grid was achieved on understanding how to tweak the properties of the box object.

6.1 Box Object

Vpython provides to the user a variety of geometric objects, the below simple program will display a box object on the scene.

```
from visual import*

box()
```

One could provide other specific properties pertaining the box such as position, size, axis, color and visibility into the object constructor syntax. Here is a sample syntax format specifying these properties.

```
newBox = box(pos=vec(x0,y0, z0), size=vec(L,H,W), axis=vec(a,b,c),
color=color.red)
```

As shown from the above syntax, attributes such as position (pos), size and axis have vector components and these attributes, and their respective component can be accessed separately by using the syntax (in this case)

```
newBox.pos.x = x0

newBox.size.z = W

newBox.color = color.red
```

The position attribute is at the center of the box, the length is along the x-axis, height along the y-axis and width on the z-axis (this is the default box orientation on visual python). However, the axis attribute defines the direction for the length or the orientation of the length of the box (see an illustration diagram below)[7].

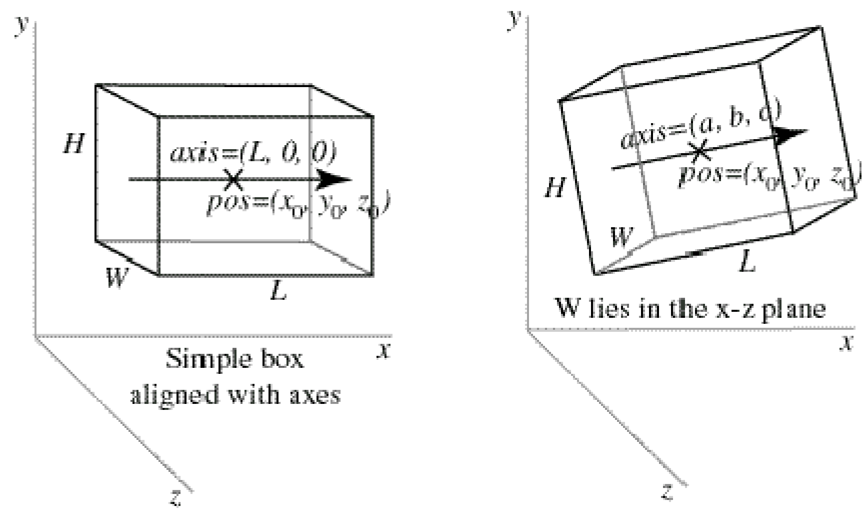

Fig10: Illustration showing effect of the axis attribute on the box length.

6.2 Grid Design

The design is focused on creating a series of intersecting boxes viewed as lines, having the pickable attribute on each box set to False to prevent the box object from been picked by any interactive pointer such as the mouse(ie, box = None) . Each intersection will be 0.5unit apart on the canvas producing squares with an area of 0.25unit^2. The grid will be plotted on the x,y,z plan, this was achieved by modifying the axis attribute following principles on the box object, see code on fig11.

Grid Total Area = 10unit * 10unit

Grid step size = 0.5unit

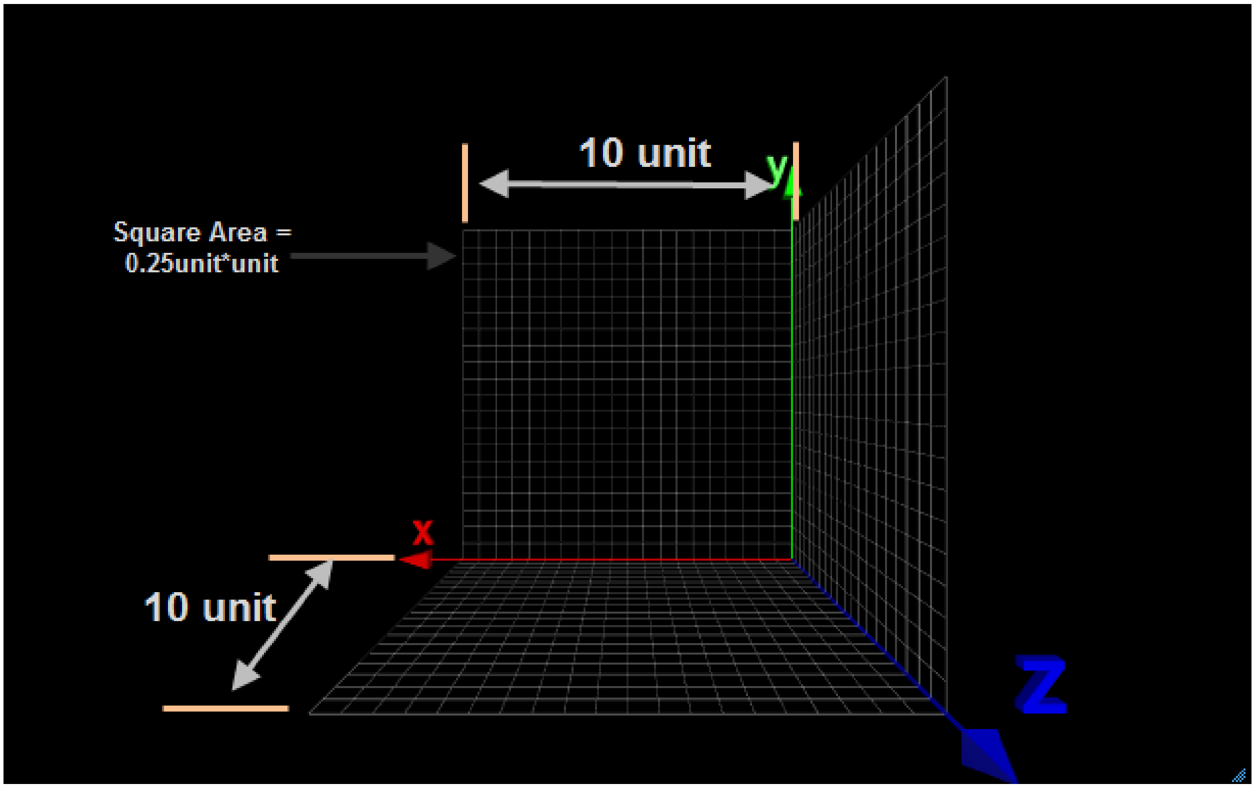

Fig11: Grid display on scene.

```
#****************************************SHOW GRID LINES***********
GRID_TOTAL_LENGTH = 10
GRID_STEP_SIZE = 0.5
GRID_HALF_LENGTH = GRID_TOTAL_LENGTH/2
gridLimit = (GRID_HALF_LENGTH+GRID_STEP_SIZE);
gridlines = []; #hold grid objects
H = 0.02*GRID_STEP_SIZE;
W = 0.02*GRID_STEP_SIZE;

#**********************back grid****************************
for eachStraightLine_Back in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(eachStraightLine_Back,0,-GRID_HALF_LENGTH), size=vector(GRID_TOTAL_LENGTH,H,W),
                            axis=vector(0,1,0), color = color.gray(.9),pickable = False))

for eachCrossedLine_Back in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(0,eachCrossedLine_Back,-GRID_HALF_LENGTH), size=vector(GRID_TOTAL_LENGTH,H,W),
                            axis=vector(1,0,0), color = color.gray(.9),pickable = False))

```

```


#********************bottom grid****************************
for eachStraightLine_Bottom in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(eachStraightLine_Bottom,-GRID_HALF_LENGTH,0), size=vector(GRID_TOTAL_LENGTH,H,W),
                            axis=vector(0,0,1), color = color.gray(.9),pickable = False))

for eachCrossedLine_Bottom in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(0,-GRID_HALF_LENGTH,eachCrossedLine_Bottom), size=vector(GRID_TOTAL_LENGTH,H,W),
                            color = color.gray(.9),pickable = False))

#*********************right grid****************************
for eachStraightLine_Right in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(GRID_HALF_LENGTH,0,eachStraightLine_Right), size=vector(GRID_TOTAL_LENGTH,H,W),
                            axis=vector(0,1,0), color = color.gray(.9),pickable = False))

for eachCrossedLine_Bottom in range(-GRID_HALF_LENGTH, gridLimit, GRID_STEP_SIZE):
    gridlines.append(box( pos=vector(GRID_HALF_LENGTH,eachCrossedLine_Bottom,0), size=vector(GRID_TOTAL_LENGTH,H,W),
                            axis=vector(0,0,1), color = color.gray(.9),pickable = False))


#***************************************************************
```

Fig12: Grid object build syntax.

6.3 3-Dimensional Track Lines

As a default, they are no extended features on visual python that gives the user further perspective of the object position and movement on the scene, hence the grid implementation. The user experience with the grid will be further improved when a track line which extends from the three-dimensional object to the grid. The implementation of a three-dimensional track line method to allow the user to visually track the three-dimensional object as it moves on the grid.

Considering the box attributes explained on section 6.1, the following points should be noted:

1. The position of the line(box) is at the center.
2. The position will always move to the center as either the length, width or height attribute of the line(box) varies.
3. The axis defines the orientation of the length

The track lines will at all time extend from its center to the object position and the grid wall, depending on the axis attribute setting, this will be either to the wall on the x-axis, y-axis or z-axis. Three track lines have been defined to track the three-dimensional object along all three axis. Implementation was done by applying principles surrounding vectors in three-dimensional space.

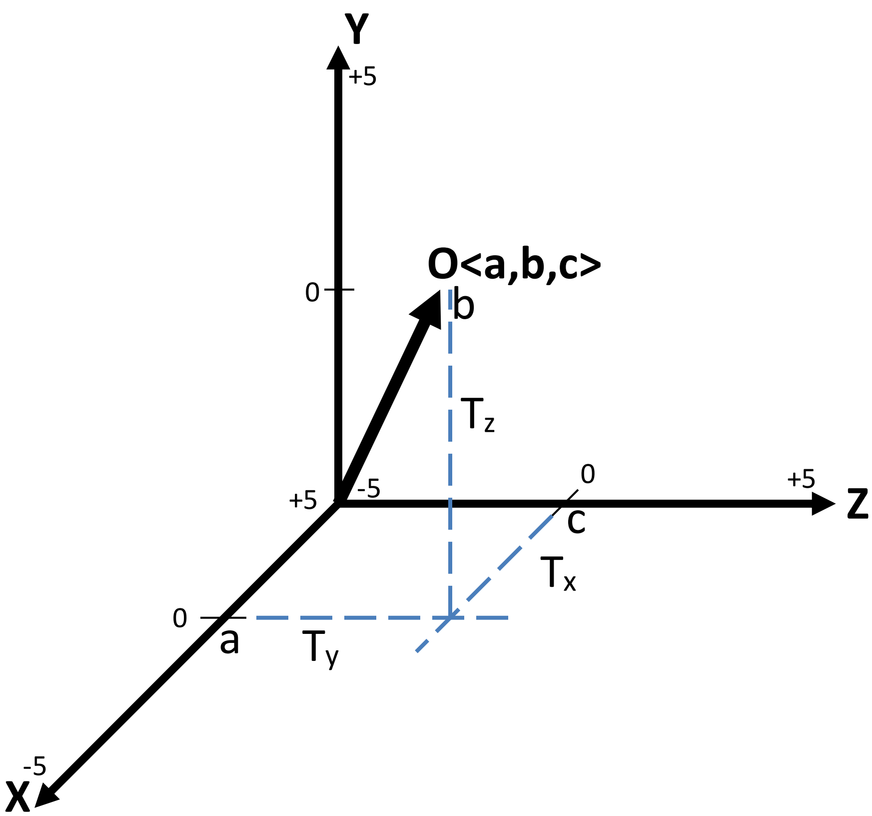

Fig13: Vector projection on scene.

For the sake of an accurate interpretation, we will represent the axis as vectors to reflect what was implemented on the tool.

X represents a two-dimensional vector that goes from point +5 to -5.

Y represents a two-dimensional vector that goes from point -5 to +5.

Z represents a two-dimensional vector that goes from point -5 to +5.

The vector projection **O <a,b,c>** represents the position of the three-dimensional object and the grid half length G_{hl} = 5unit as shown on fig13.

The dynamic length and position of the track line along the x-axis is given by:

$T_x = G_{hl} - a$

$T_{x.posX} = (G_{hl} + a)/2$

$T_{x.posY} = b$

$T_{x.posZ} = c$

The dynamic length and position of the track line along the y-axis is given by:

$T_y = -G_{hl} - b$

$T_{y.posX} = a$

$T_{y.posY} = (-G_{hl} + b)/2$

$T_{x.posZ} = c$

The dynamic length and position of the track line along the z-axis is given by:

$T_z = -G_{hl} - c$

$T_{y.posX} = a$

$T_{y.posY} = b$

$T_{x.posZ} = (-G_{hl} + c)/2$

Where:

T_x – Dynamic length of the track line along the x-axis.

T_y – Dynamic length of the track line along the y-axis.

T_z – Dynamic length of the track line along the z-axis.

G_{hl} – Grid half length (this also defines the axis origin).

a – Vector position on the x-axis.

b - Vector position on the y-axis.

c - Vector position on the z-axis.

$T_{x.posX}$ – X-axis trackline position on the x-axis.

$T_{x.posY}$ – X-axis trackline position on the y-axis.

$T_{x.posZ}$ – X-axis trackline position on the z-axis.

$T_{y.posX}$ – Y-axis trackline position on the x-axis.

$T_{y.posY}$ – Y-axis trackline position on the y-axis.

$T_{y.posZ}$ – Y-axis trackline position on the z-axis.

$T_{z.posX}$ – Z-axis trackline position on the x-axis.

$T_{z.posY}$ – Z-axis trackline position on the y-axis.

$T_{z.posZ}$ – Z-axis trackline position on the z-axis.

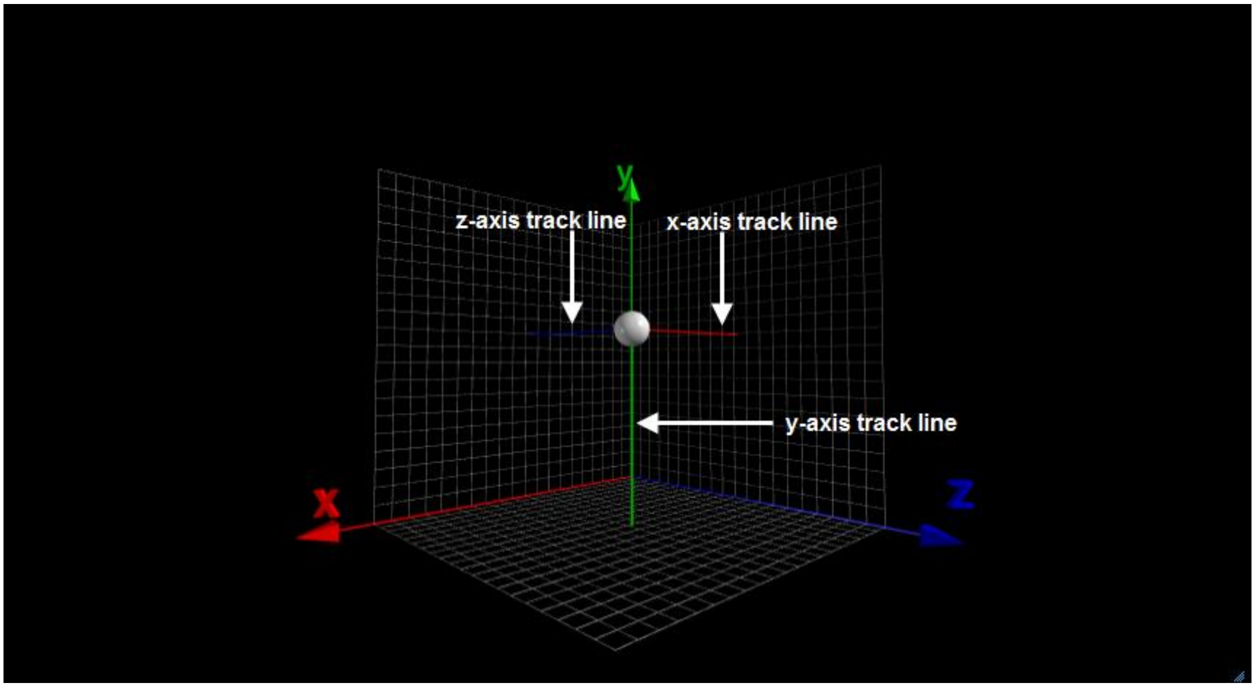

Fig14: x,y,z-axis Track line displayed on grid

```
#*********************OBJECT TRACK LINE**********************
trackLineSwitch = False;

#Define track lines on all 3 dimensional axis, not visible by default
trackLineX = box(pos=vector(0,0,0), size=vector(GRID_HALF_LENGTH,5*H,5*W), color = xAxis.color, visible = trackLineSwitch, axis = vector(1,0,0))
trackLineY = box(pos=vector(0,0,0), size=vector(GRID_HALF_LENGTH,5*H,5*W), color = yAxis.color, visible = trackLineSwitch, axis = vector(0,1,0))
trackLineZ = box(pos=vector(0,0,0), size=vector(GRID_HALF_LENGTH,5*H,5*W), color = zAxis.color, visible = trackLineSwitch, axis = vector(0,0,1))

#Method to calculate object position and display 3 dimensional track lines
def showTrackLine(obj):
    global trackLineSwitch

    trackLineX.visible = trackLineSwitch #Turn on x-axis track line visibility
    trackLineY.visible = trackLineSwitch #Turn on y-axis track line visibility
    trackLineZ.visible = trackLineSwitch #Turn on z-axis track line visibility

    #Calculate and track object position along the x-axis
    trackLineX.pos.x = ((GRID_HALF_LENGTH + obj.pos.x)/2)
    trackLineX.pos.y = (obj.pos.y)
    trackLineX.pos.z = (obj.pos.z)
    trackLineX.size.x = GRID_HALF_LENGTH - obj.pos.x

    #Calculate and track object position along the y-axis
    trackLineY.pos.x = (obj.pos.x)
    trackLineY.pos.y = (-GRID_HALF_LENGTH + obj.pos.y)/2
    trackLineY.pos.z = (obj.pos.z)
    trackLineY.size.x = GRID_HALF_LENGTH + obj.pos.y

    #Calculate and track object position along the z-axis
    trackLineZ.pos.x = (obj.pos.x)
    trackLineZ.pos.y = (obj.pos.y)
    trackLineZ.pos.z = (-GRID_HALF_LENGTH + obj.pos.z)/2
    trackLineZ.size.x = GRID_HALF_LENGTH + obj.pos.z
```

Fig15: Program code: Object 3D track line.

6.4 Snap to Grid

Three-dimensional objects on the grid on interaction with the mouse pointer will move at an imprecise manner within the grid, hence, in some design cases might not provide object position control. Implementing a snap to grid will enable objects position along all axis to snap to the grid intersections, this will allow objects to align precisely in this case (based on implementation) at steps of 0.5units. The user has an option to disable this if not needed.

```
#****************************Snap to Grid*******************
snapToGrid = False

def snap(grid, x):

    return grid*round(x/grid)

#**********************************************************
```

Fig16: Program code: enable snap to grid feature.

```
#Enable snap to grid
snapCheckBox = checkbox(bind=checkSnap, text='Snap to grid')#Enable snap to grid feature
scene.append_to_caption('    ')

#
def checkSnap(checkBox):
    global snapToGrid
    if checkBox.checked:
        return snapToGrid = True
    else:
        return snapToGrid = False
```

Fig17: Program code: check/modify snap to grid feature state.

6.5 Rotating 3D Objects

In general, I think rotating 3D objects with precision in an intuitive manner is a big problem. I did go through multiple documentations to grasp an understanding of how to implement this on the tool. At the time of this documentation, I would like to point out that I do not have a full understanding of 3D rotation but through my understanding on geometrical principles as regards to vectors with a series of trial and errors, I was able to pull it through. However, further studies are been carried out on it to further optimize the rotation. First, I had to under study the 2D rotations.

6.5.1 Rotation in 2D

Direct techniques for rotation in two dimensions require three points as shown in figure 18.

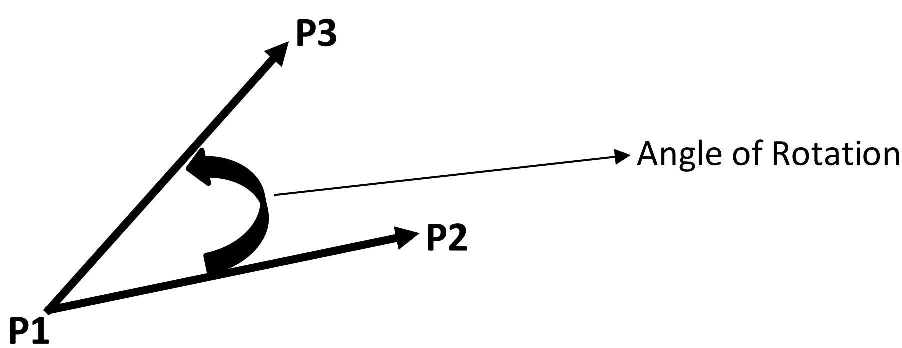

Fig18: Three-point 2D rotation[8]

As with translations we use object information to perform the manipulation. This helps to consider a 3D rotation as occurring in a plane rather than about an axis. Figure 19 shows the rotation about an edge[8].

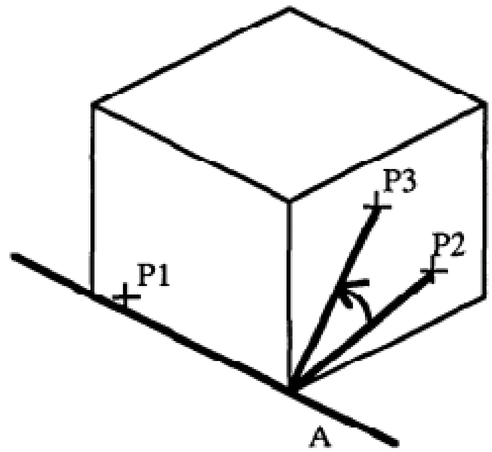

Fig19: Edge rotation[8]

P1 selects an edge A which becomes the axis of rotation. In addition to the axis of rotation, the angle of is required. The 2D point P2 is projected to the surface of the object to obtain a 3D point P2'. The point P3 is projected to a 3D point P3' on the plane P which contains P2' and has A as it's normal. The angle of rotation is the angle between P2' and P3'[8].

Figure 20 shows a similar technique based on a face of an object. P1 selects a face and is projected onto it to form P1'. The normal vector to the face and P1' are used to compute the axis of rotation A and the face itself defines the plane of rotation P. Points P2 and P3 are also projected onto the plane of rotation to form P2' and P3'. The actual rotation is computed as in the edge technique. Continuous rotation techniques are also possible by using the plane and axis initially selected as the basis for each incremental rotation. [8]

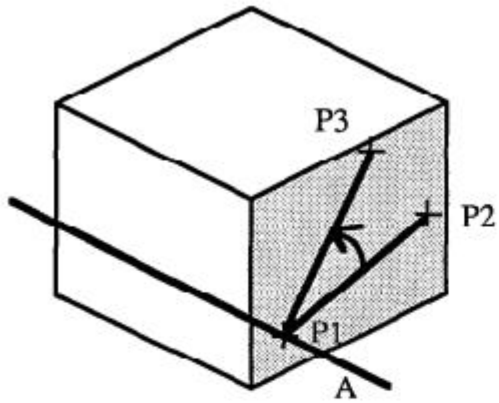

Fig20: Face rotation [8]

6.5.2 Mapping 3D Coordinates to 2D Screen Coordinates

A key operation in accurately presenting the 3D images on the canvas on the 2D screen is converting from a three dimension to two dimensions. As an instance, a viewer looking out through a window to a building within line of site as shown on figure21. Been still, the viewer can trace out the image or outline of the building onto the window screen. The traced outline of the building will be considerably smaller in size as compared to the actual building, how small will depend on the viewer distance from the building. Each point a, b on the actual building will intersect on the viewer eye at a point a’, b’ on the screen as shown on figure 21. These lines are called sight lines and they meet in one point in the eye of the viewer [9].

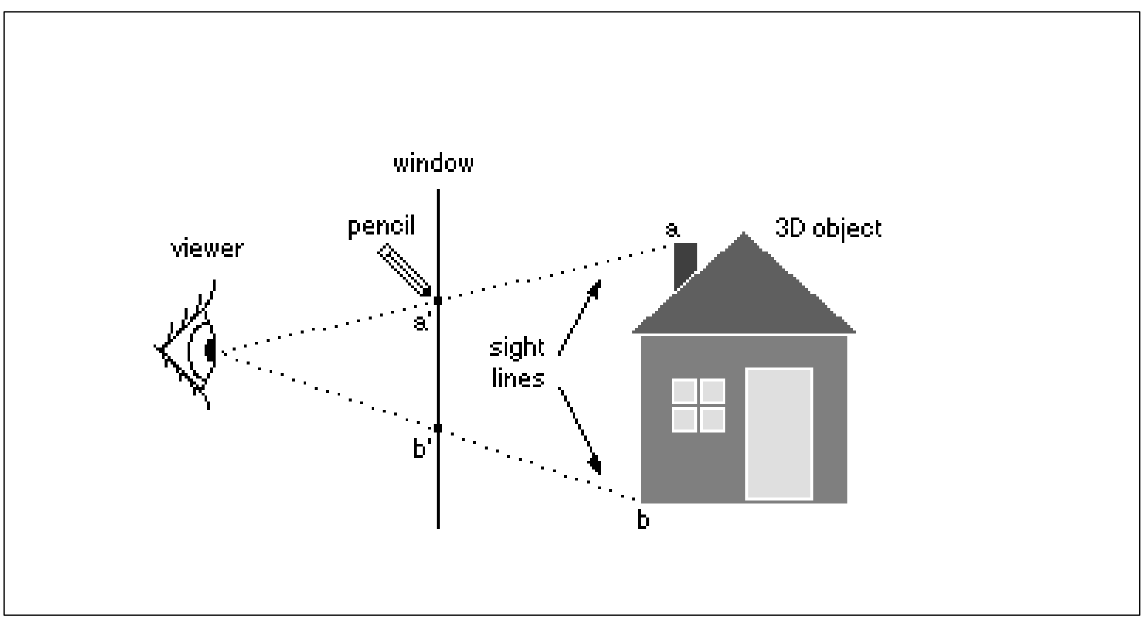

Fig21: Side view of window and 3D object[9].

Replacing the window screen to a video screen and the house to a virtual 3D object viewed by the user through the video screen shown on figure 22. At this point, I will like to point out that the orientation of the 3D axes used for this study does vary with the 3D orientation on the Vpython canvas as considered from the user's perspective, but this is no problem. For the purpose of this study, when looking from the video screen, the x-axis is horizontal and increases to the right, the y-axis is the depth axis and increase as it moves away from the viewer and the z-axis is vertical and increases upwards. The axis orientation to the two-dimensional object seen by the viewer on the video screen is the horizontal x-axis which increase to the right and the vertical y-axis which increases downwards and the origin (0,0) is the top left corner of the screen. Referring to figure 22, the diagram presents a side view of the video screen, hence the real world is to its left and the virtual world to its right. The y-axis (the horizontal line) represents the depth, z-axis (the vertical line) represents the height and from the current view perspective, the x-axis extends from the screen (i.e, towards the reader), however, the x-axis will be ignored at the moment. Point v represents the viewer eye, p is a point on the 3D image and p' is the point on the screen where the computer will draw p. Points behind the screen have positive y-coordinates while points in front of the screen have negative y-coordinates.. Hence, v_y is negative, p_y is positive and p'_y is zero[9].

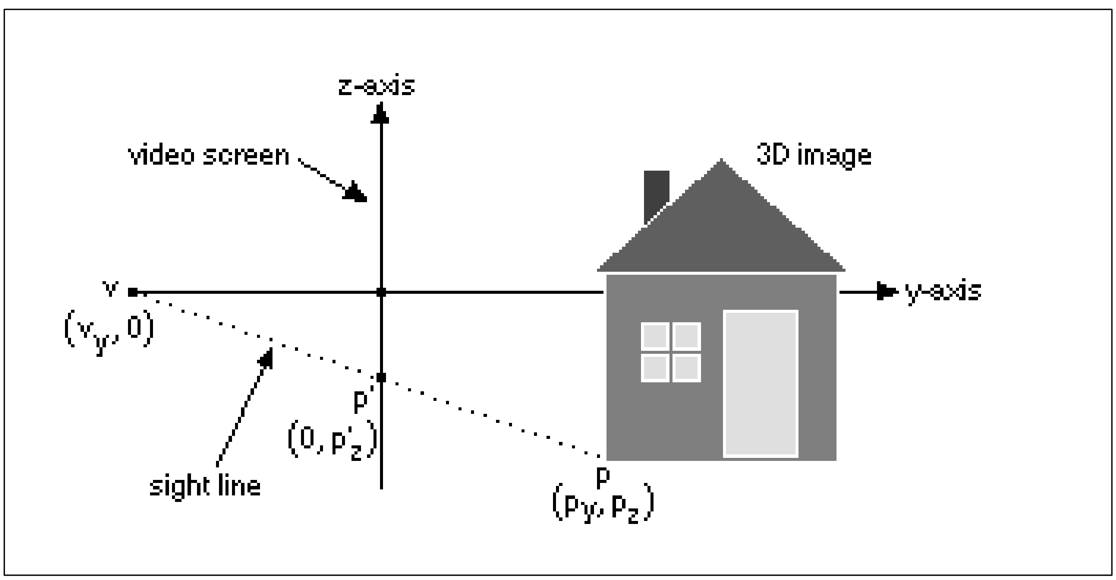

Fig22: Side view of video screen and a virtual 3D object[9].

Figure 23 is termed the aerial view of figure 22, allowing us to see the x-axis. To map a 3D point p on the 3D object to a 2D point p' on the screen, we imagine a straight line referred as the sight line from point p to the user’s eye at point v. p’ is located on the screen at the intersection of the sight line and the screen. All 3D points located on this sight line are mapped to the one 2D point where the sight line intersects the screen[9].

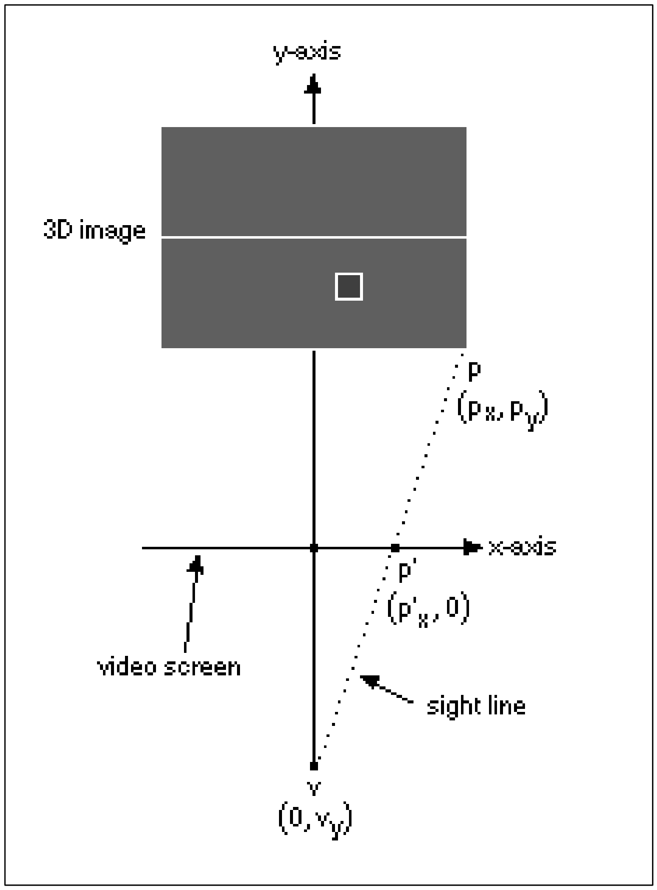

Fig23: Aerial view of screen and virtual 3D object[9].

This information changes the whole idea on how the mouse interacts with the object on the 3D canvas, the idea that the mouse pointer makes actual contact with the 3D image when it overlays the object on screen is false. However, in real sense, the mouse pointer does not exist in the 3D scene but within the 2D screen. Hence, the mouse position does not correlate in the 3D space as is been presumed by the user. The Vpython ‘scene.mouse.project()’ function will convert the 3D position of the mouse into 2D, providing a mapping of the object on the two-dimensional plane.

6.5.3 Projecting Mouse Position

The code line '**scene.mouse.project(normal=vec(0,0,1), point =vec(0,0,3)**' will project the mouse cursor onto a xy plane that is perpendicular to the specified normal vector as depicted on fig 24(the x-axis points out of your screen), the point vector specifies the height above the plane where the mouse cursor correlates with the plane. The below set of codes will illustrate and provide further clarity on the vpython mouse projection object.

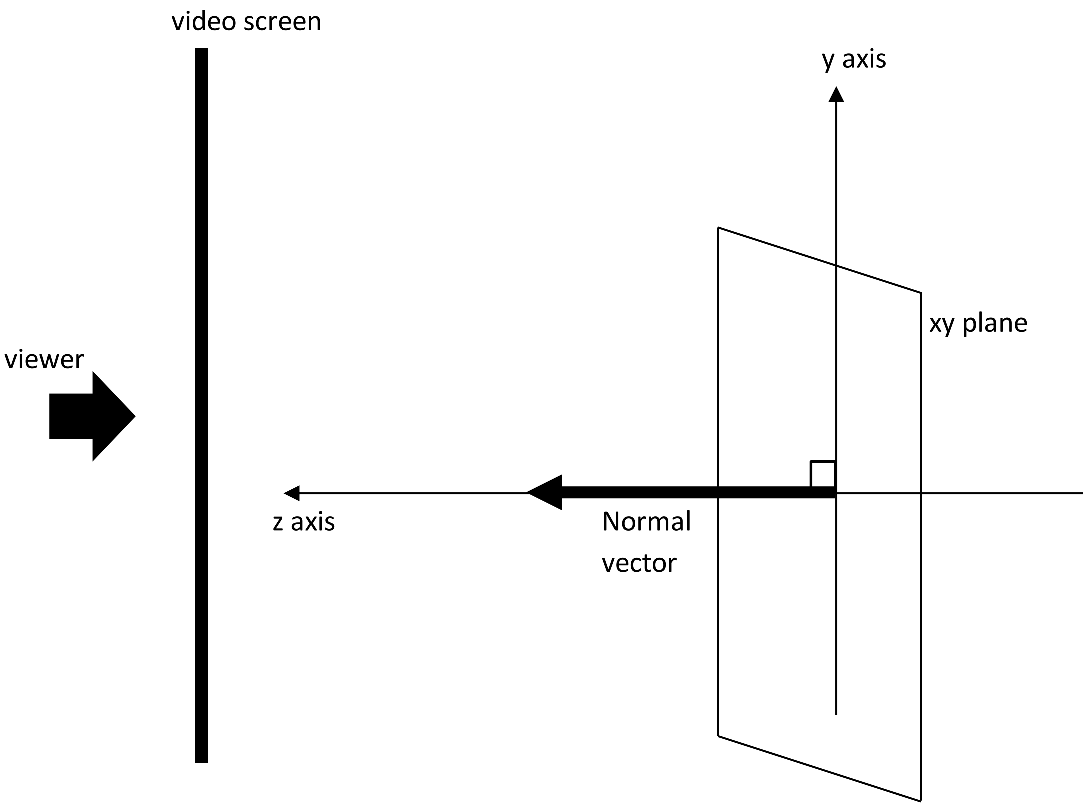

Fig24: Side view diagram showing the screen and the xy plane

The program shown fig 25 will display a box, then wait for the mouse down event and will display a red sphere at the mouse location whenever the right mouse button is pressed. However, it is noticed that the red sphere is not displayed whenever the mouse cursor is pressed inside the box as shown on fig 26. This is because the mouse click is in the xy plane and the sphere buried inside the box[10].

If you change the program as shown on fig 27, you will have all the spheres go into the xy plane, perpendicular the z axis, hence, with the cursor clicked inside the box you will have the spheres displayed at a unit height above the box as shown on fig 28[10], clearly the spheres now showup Infront of the box..

```
scene.range = 1 #disable auto-scaling
Box = box(pos=vec(0,0,0))#draw box

#binding function calls drawRedSphere on the event the mouse right button is pressed
scene.bind("mousedown", drawRedSphere)

def drawRedSphere():
    #assign the current mouse 3D position to the 'temp' variable.
    temp = scene.mouse.pos
    # None if no intersection with plane:
    if (temp != None):
        #draw red sphere on the assigned position
        sphere(pos = temp, radius = 0.1,color = color.red)
```

Fig25: Program code: mouse projection not used

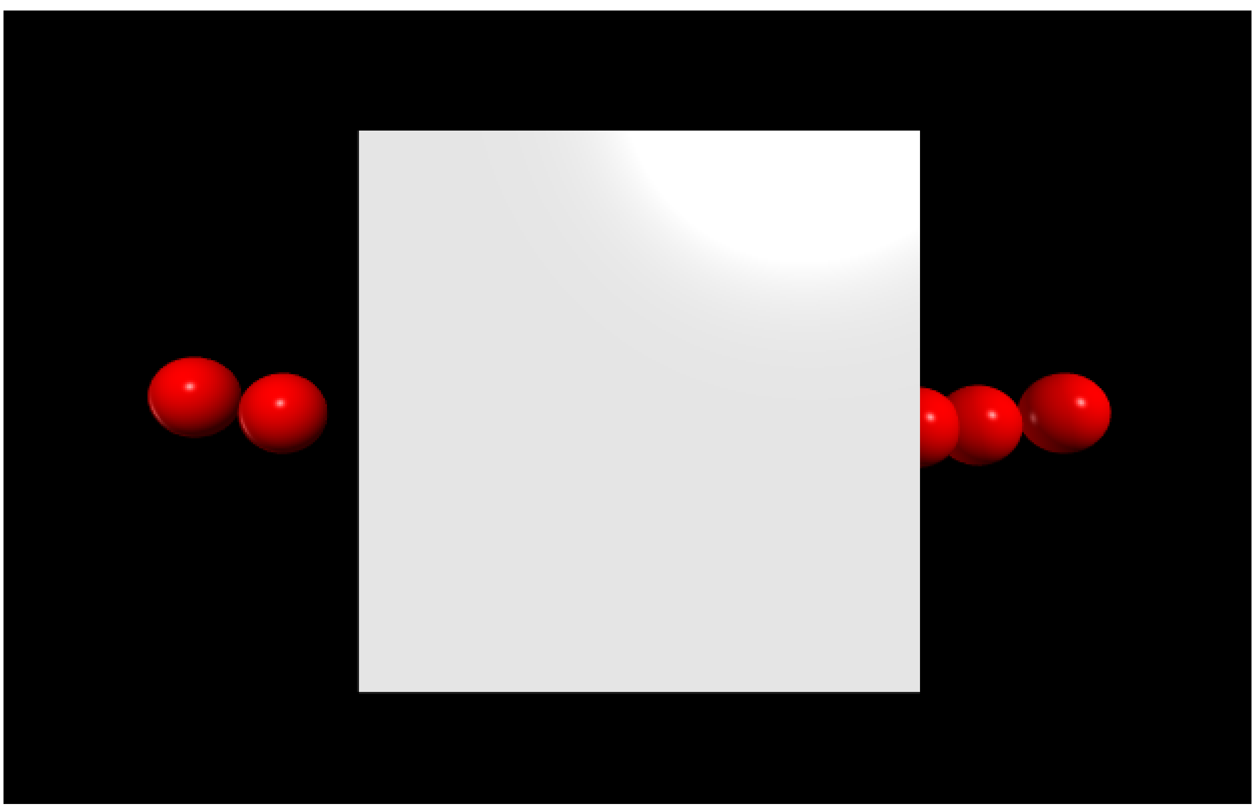

Fig26: Scene display: mouse projection not used

```
scene.range = 1 #disable auto-scaling
Box = box(pos=vec(0,0,0))#draw box

#binding function calls drawRedSphere on the event the mouse right button is pressed
scene.bind("mousedown", drawRedSphere)

def drawRedSphere():
    #projects the current mouse 3D position to the xy plane and is assigned to 'temp' variable.
    temp = scene.mouse.project(normal=vec(0,0,1),point=vec(0,0,1))
    # None if no intersection with plane:
    if (temp != None):
        #draw red sphere on the assigned position
        sphere(pos = temp, radius = 0.1,color = color.red)
```

Fig27: Program code: mouse projection implementation

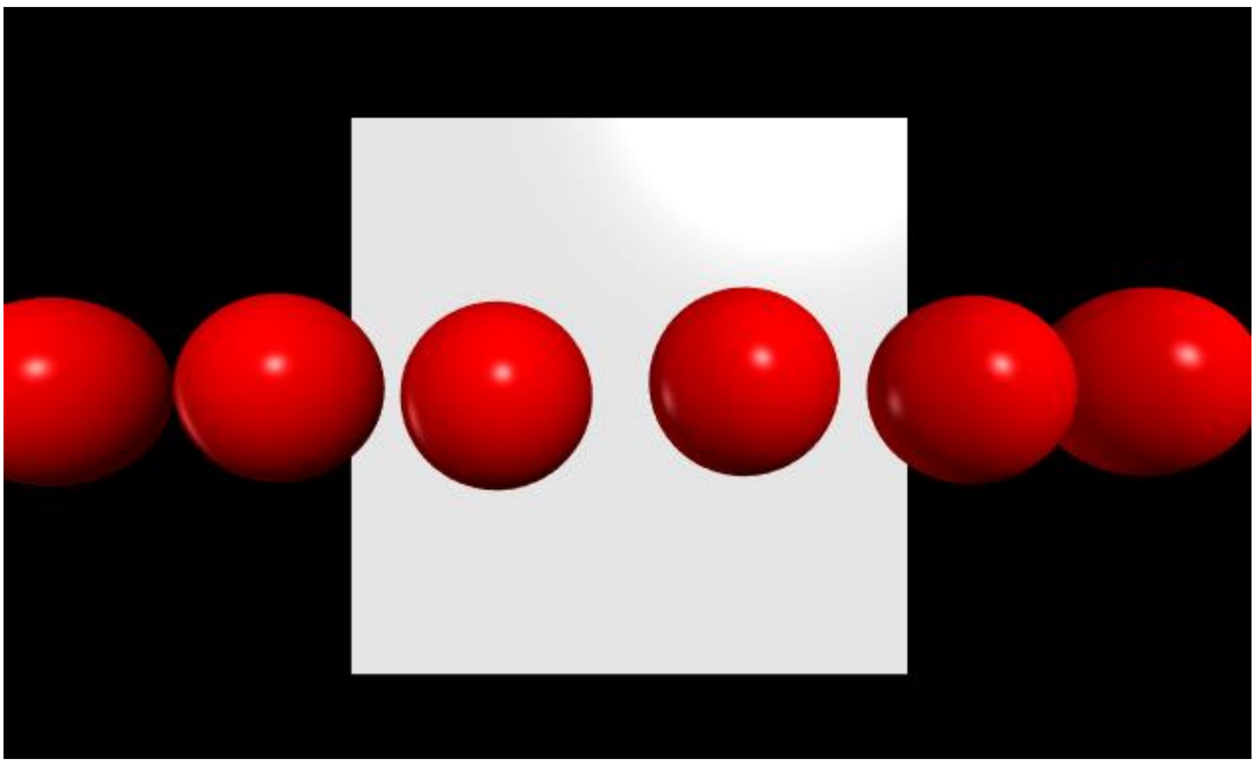

Fig28: Scene display: mouse projection implementation

6.5.4 Rotation Implementation

The statement below will rotate the object obj through an angle about an axis relative to an origin:

obj.rotate(angle=a,axis=vec(x,y,z),origin=vec(xo,yo,zo))

If the origin is not specified, rotation will be relative to obj.pos[10].

To implement the three-dimensional rotation on the tool, I had to hold on to the below mathematical/geometrical principles, namely:

1. The difference of two vectors A and B results in the formation of a third vector Y. if the vector Y= B - A, then the Y vector having both magnitude and direction goes from the head of A to B as shown on fig 29. This will determine the direction/angle of rotation of the 3D object. Relating the fig 29 to the program code on fig 31, the 'lastMousePosition' is represented with the vector A, the 'currentMousePosition' with the vector B and 'move' with the difference vector Y.
2. The cross product of two different 3D vectors will generate a new 3D vector (resultant vector) perpendicular to the parallelogram produced by the two vectors. Shown on fig30 is the cross product between the difference vector and the default rotation axis vector <0,0,1> in the xy plane which produces a resultant vector which will be assigned to the axis of rotation. Using the right-hand rule, you can confirm the direction of the resultant vector.

Implementation code shown on fig 31.

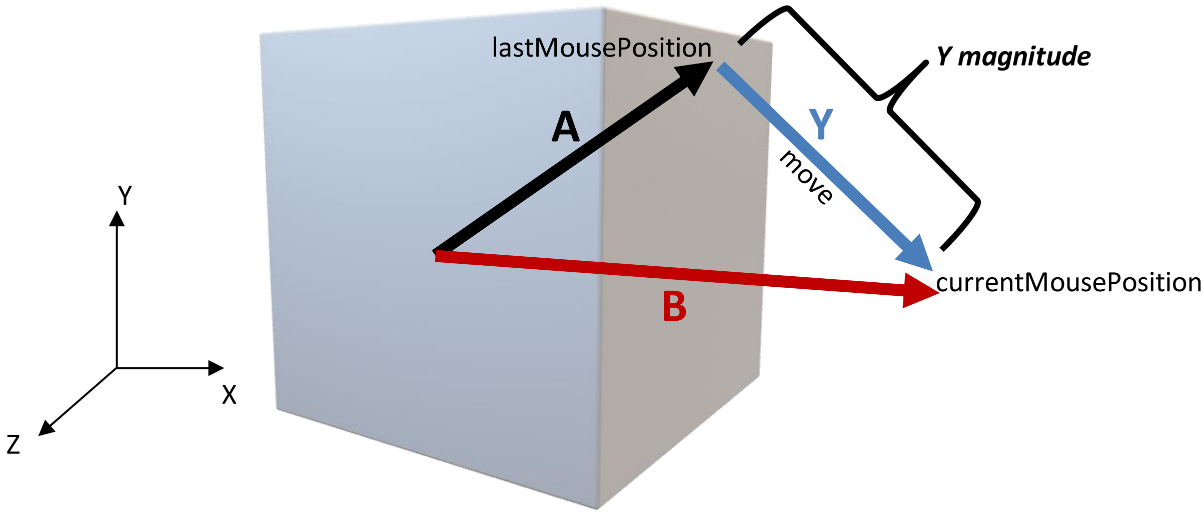

Fig29: 3D Object showing vector relation

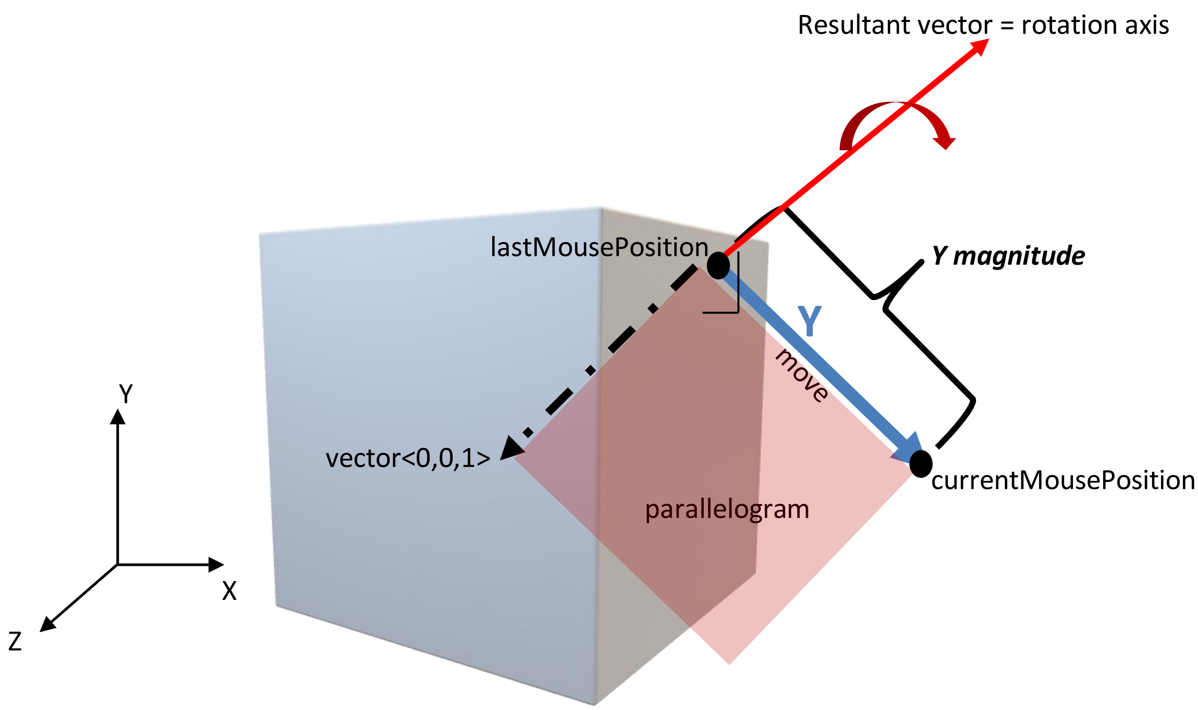

Fig30: 3D Object showing vector relation: cross product effect

```
# Function to implement object rotation
def rotateObj():
    global dragObj,enableRotate,lastMousePosition
    if (enableRotate) :
        currentMousePosition = scene.mouse.project(normal=vec(0,0,1))
#identifies the current position of the mouse on the 2D plane
        move = currentMousePosition - lastMousePosition
        dragObj.rotate(angle=(-move.mag)*0.1,
axis=move.cross(vec(0,0,1)))
        lastMousePosition = currentMousePosition
```

Fig31: Program code: Rotation implementation

6.6 Widgets

Vpython has a list of widgets on its libraries such as buttons, sliders, radio buttons, check box e.t.c which will be used to introduce interactive experience on the tool. Implementation requires binding these widgets to specific methods which will execute the required action. The diagram on fig32 shows the layout of some widgets implemented on the tool while fig33 shows the implementation code.

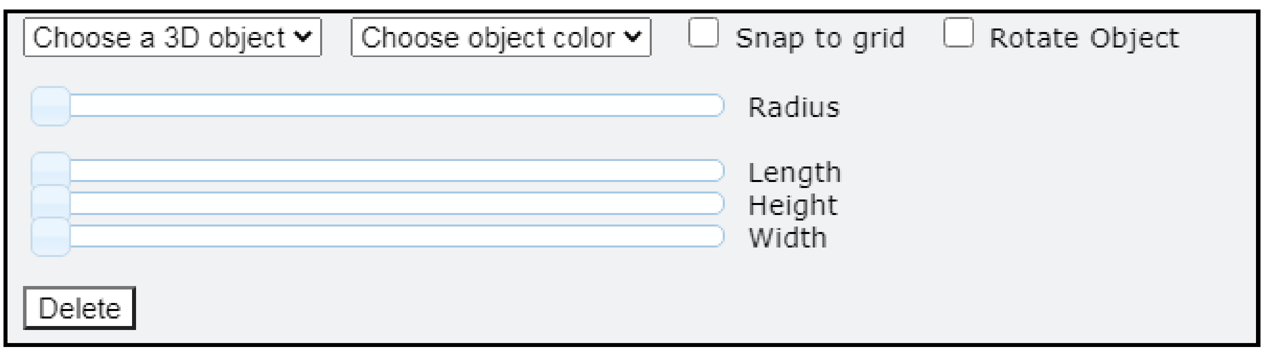

Fig32: Implemented widgets display

```
self.sceneColorButton = button(text="Day", pos=self.scene.title_anchor,
bind=editor_object.clickThrough, editor_object=editor_object,
method="thisWidget.editor_object.toggleScene()") #Switch scene
background color
self.objMenu = menu( choices=['Choose a 3D object', 'Sphere', 'Box',
'Cylinder','Cone'], bind=editor_object.clickThrough,
editor_object=editor_object,
method="thisWidget.editor_object.selectObj()") #Creates object menu,
allows user select oject to display
self.scene.append_to_caption('   ')
self.colorMenu = menu( choices=['Choose object  color','Red', 'Green',
'Blue'],bind=editor_object.clickThrough,
editor_object=editor_object,
method="thisWidget.editor_object.changeColor()" )#Modify object color
self.scene.append_to_caption('   ')
self.snapCheckBox = checkbox(bind=editor_object.clickThrough,
text='Snap to grid',editor_object=editor_object,
method="thisWidget.editor_object.checkSnap()")#Enable snap to grid
feature
self.scene.append_to_caption('   ')
self.rotateCheckBox = checkbox(bind=editor_object.clickThrough,
text='Rotate Object',editor_object=editor_object,
method="thisWidget.editor_object.checkRotate()")#Enable rotation
self.scene.append_to_caption('\n\n')
self.radius_slider = slider( bind=editor_object.clickThrough, min=0.1,
max=5,editor_object=editor_object,
method="thisWidget.editor_object.radiusSlide()")#Radius slider bar
self.scene.append_to_caption('Radius\n\n')
self.length_slider = slider( bind=editor_object.clickThrough, min=0.1,
max=10,editor_object=editor_object,
method="thisWidget.editor_object.lengthSlide()")#Length slider bar
self.scene.append_to_caption('Length\n')
self.height_slider = slider( bind=editor_object.clickThrough , min=0.1,
max=10,editor_object=editor_object,
method="thisWidget.editor_object.heightSlide()")#Height slider bar
self.scene.append_to_caption('Height\n')
self.width_slider = slider( bind=editor_object.clickThrough, min=0.1,
max=10,editor_object=editor_object,
method="thisWidget.editor_object.widthSlide()")#Width slider bar
self.scene.append_to_caption('Width\n\n')
self.deleButton = button(text="Delete",bind=editor_object.clickThrough,
editor_object=editor_object,
method="thisWidget.editor_object.clearObj()") #Delete currrent object
```

Fig33: Program code: Widget implementation

6.7 File Operation

6.7.1 Import a Library

Vpython Glowscript allows users to import JavaScript library, this could be done using the syntax:

self.get_library("file directory") #import javascript file

This triggers a pop up box which prompts the user to input the directory to the JavaScript file, an error message will be displayed if the file is not found after a specific amount of time[12].

However, I would have to note that although I have this functionality implemented on the tool, there isn't a way to import from another Vpython Glowscript program currently.

6.7.2 Read Write File

In contrast to other applications, reading and writing files is different on the web browser due to security issues, these issues has to do with the protections built into the browsers to prevent it from overwriting files on the hard drive. Writing a file on the vpython glowscript application is not supported for this reason. The read_local_file function appends a button that says "Choose File" on the screen at the location specified. A file box appears which allows the user to choose a file when the button is clicked. I have allowed file viewing using the print() function, this is not ideal but is the only to view local files. Files will be shown as text in a scrolling text region at the bottom of the application window.

Clearly, there is a couple bottlenecks implementing file operations posed by the limitations on Vpython Glowscript. These challenges are:

1 Supports only importation of JavaScript file.
2 Imported file cannot be written in Vpython because the library does not go through preprocessing which occurs when you run a program on Glowscript.
3 The file cannot include options that require using "rate" or "waitfor".
4 Vector operations must be written in specific formats e.g:

Regular Format	Supported Format
A + B	A.add(B)
A - B	A.sub(B)
k * A	A.multiply(k)
A/k	A.divide(k)

k is an ordinary number

5 Importing from your own glowscript.org files is not possible, can only work if the library is placed on a different website.

6 Read files on Vpython Glowscript are displayed on the computer if printed, meaning they would have to be displayed in a text format on the scrolling text region.

7 Writing directly to the user's device storage is not allowed on Vpython Glowscript, posed by built-in securities on web browsers.

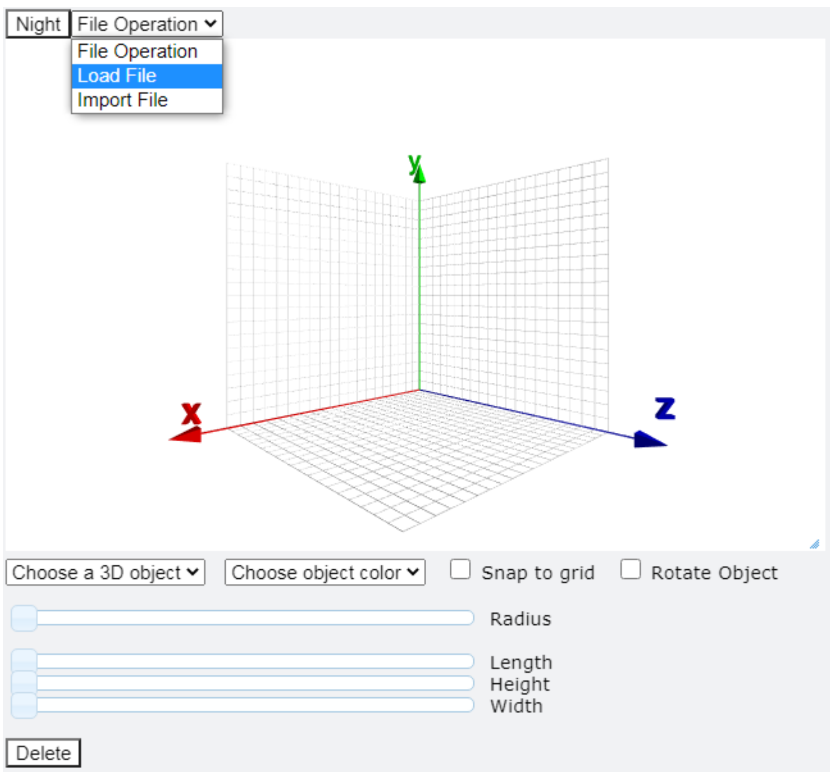

Fig34: File operation

6.8 Other Features

6.8.1 Object Cloning

This will allow users to generate an exact copy of a selected object on the scene, modifications on the clone does not affect the original object. Limitations on Vpython does not allow objects such as triangle and quads to be cloned but permits cloning of compound objects as shown on figure 36. The program code shown below which is a method of class editor (refer to class diagram) implements cloning on the current object selected by the user with just a difference in position.

```
def cloneObj(self):
        if self.getCurrentObj() != None:
            self.getCurrentObj().clone(pos =vec(-2,0,0)) #clones
current object.
```

Fig35 Program code: Object cloning

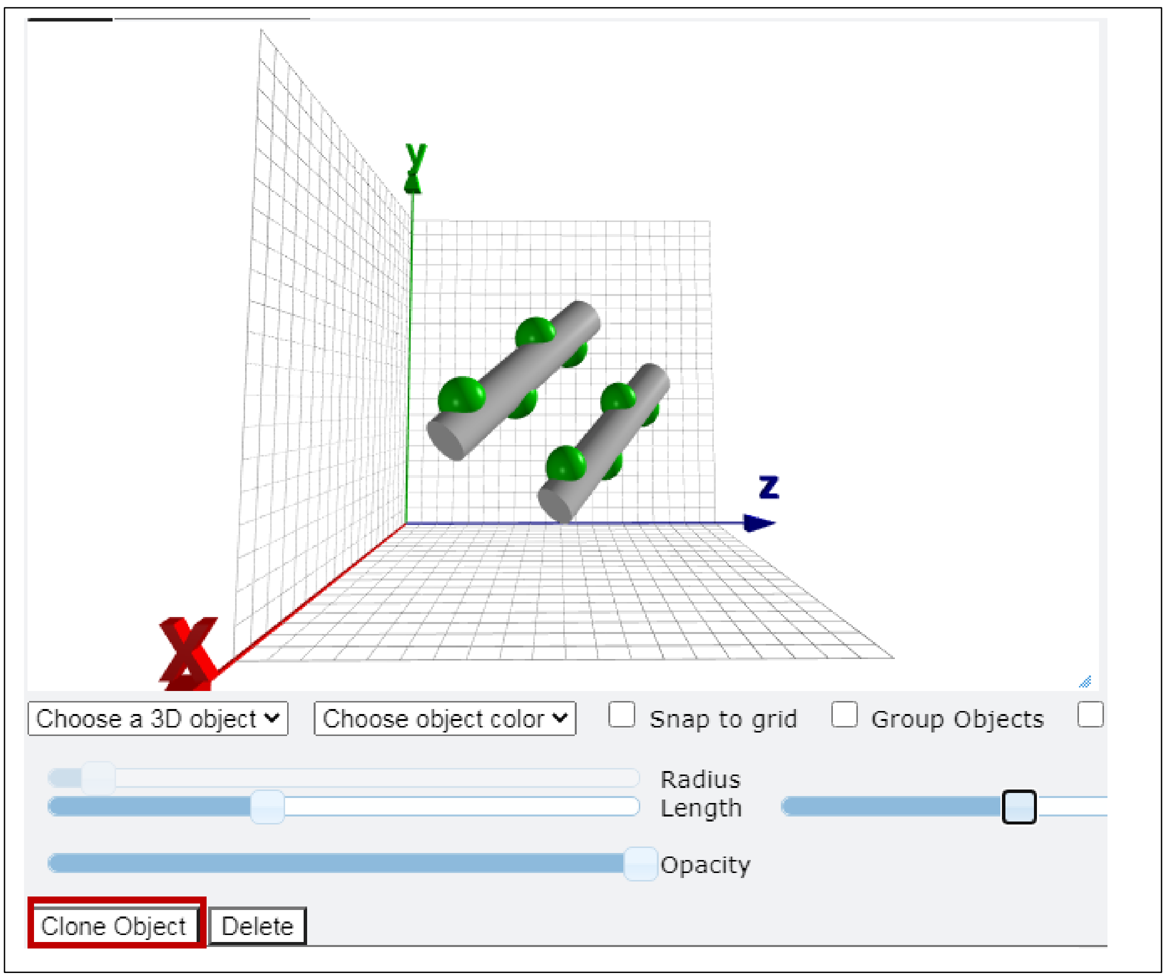

Fig36 Cloned objects on application scene

6.8.2 Object Opacity

The transparency of a selected object on the tool can be modified within a range of 0.2 to 1 (both inclusive), however, Vpython allows you to specify this from 0 to 1. Currently, there's an unexplained glitch on the program which impacts the visbility of the gridlines when object opacity is lowered to values less than 1, see effect on figure 38. Currently, I due suspect that this could be due to the webGL machine, however, this is not verified.

```
self.opacity_slider = slider( bind=editor_object.clickThrough, min=0.2,
max=1,editor_object=editor_object,
method="thisWidget.editor_object.objOpacity()")#Adjust object opacity
```

Fig37 Program code: Opacity tuning

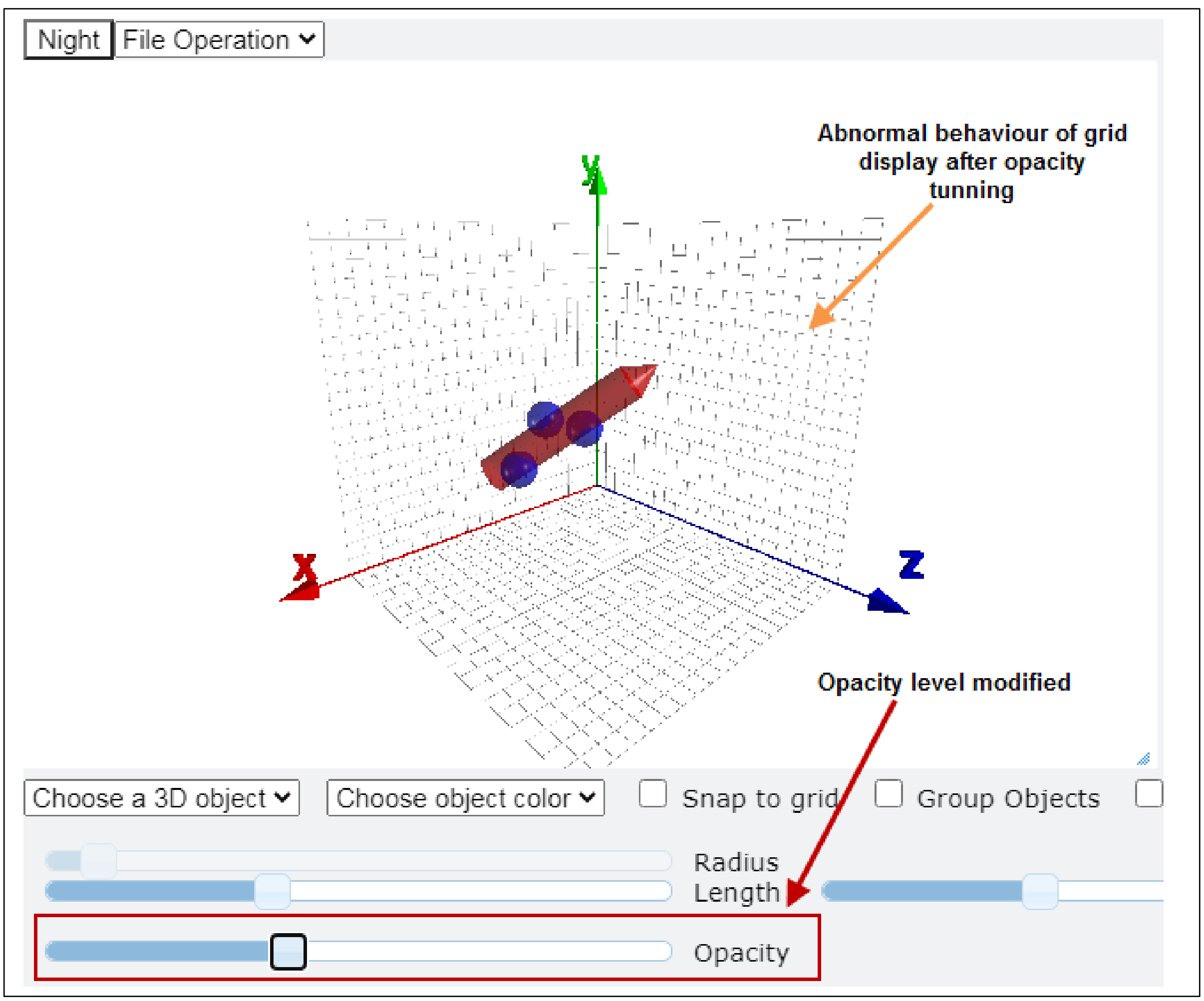

Fig38 Glitch on grid display after opacity tuning

6.8.3 Object Grouping

The Vpython compound object allows different objects to be grouped together and managed as a single object. Vpython allows you to specify attributes such as pos, color, size (or length, width, height), axis, up, opacity, shininess, emissive, and texture just like you will on a regular default 3D object. This has been explored in the tool, allowing users to group multiple objects on the scene and operating them as a single unit, however, once objects have been grouped on the tool, attributes related to dimensions cannot be modified, this limitation can be corrected by refining the implementation codes. The class editor (refer to the class diagram) manages the grouping function switch, however the algorithm implementing this is defined outside the class editor. To group a set of objects on the tool, follow the below steps:

1. Check the “Group Objects” checkbox.
2. Select the set of objects to be grouped using the mouse.
3. Press the “Ctrl” key on the keyboard when done to group objects
4. Uncheck the “Group Objects” checkbox.

Once objects are grouped, you could carry out other operations like cloning, color modification, positioning, e.t.c. However, grouped objects cannot be ungrouped currently. The code below shows implementation.

```
if thisEditor.isCompoundObj(): #checks if object grouping is enabled
        if thisEditor.getCurrentObj() != None:  #check if an object is selected
            objList.append(thisEditor.getCurrentObj()) #add selected object to object list

#######################################################################

# group objects in objList
        if 'ctrl' in theKey:
                compound(objList) #group objects contained in list
                objList = [] # clear object list
```

Fig39 Program code: Object grouping

Chapter 7 – Program Structure and Operation

7.1 Class Structure

This chapter will discuss the restructuring of the set of tool functions and operations described in the previous chapters, I have taken an object-oriented approach and categorized the tool functionality into three custom classes as shown on the class diagram on fig40. The naming convention of the custom classes does not follow what will be considered as best practice as I have named the first characters using lower cases, this is unavoidable as vpythons throws an execution failure if classes are named with the first characters as upper cases. A list of other limitations presented by vpython will be discussed later in this report.

The objective of this project has always been how we can exploit the features on the Vpython library to create a three-dimensional editor, the program have been structured around the tools available on the library.

The list of custom classes are:

- class editor
- class object3D
- class trackline

These custom classes have been realized using the functionality defined in the vpython library. The class diagram (fig40) shows the relationship in UML, the interaction between the custom classes and the vpython default library classes. At the time of this documentation, my search for a class diagram for the vpython program was fruitless, so I had to come up with the default library diagram which describes the library conceptually based on my perception, the library classes and how it relates to the custom classes. Hence, The Vpython Default Library shown on the class diagram is not accurate but will be used for the purpose of this report to best define the program structure and class relationships. Also, not all classes, class fields and class methods are presented on the default library on figure 40.

The editor class inherits functionality from the default class 3-Dimensional object, class Widgets, class Vector and class Math. The 3-Dimensional object has a list of 3D objects such as cone, box, sphere, ring, etc as child classes. The editor class also inherits functionality from the tools' custom class object3D class. The editor class primary function includes setting up the tool interface, basically the GUI such as scene, the grid structure, widgets and the 3D axis. The editor class requires the assistance of object3D class, hence, an instance of the object3D class is passed as an argument during editor initialization, this allows the class to access methods within object3D, providing functions like object creation, type, color modification and object deletion. The editor class also provides method allowing object rotation, mouse position tracking, widget handling/interaction, snap to grid, object reshaping e.t.c. Functionalities inherited from the default classes Math and Vector where implemented into editor functions that implements snap to grid and object rotation.

The object3D class basically has four functions, these functions are object creation, color modification, deleting objects and providing an instance type (e.g: sphere, ring, box, e.t.c). These methods are called from the editor class. The object3D class also inherits from the default library classes 3-Dimensional Object.

The custom trackLine class handles the three-dimensional object tracking line and object position labeling on the scene, it inherits functions from both the default class 3-Dimensional Object and the custom class editor to carry out its operation.

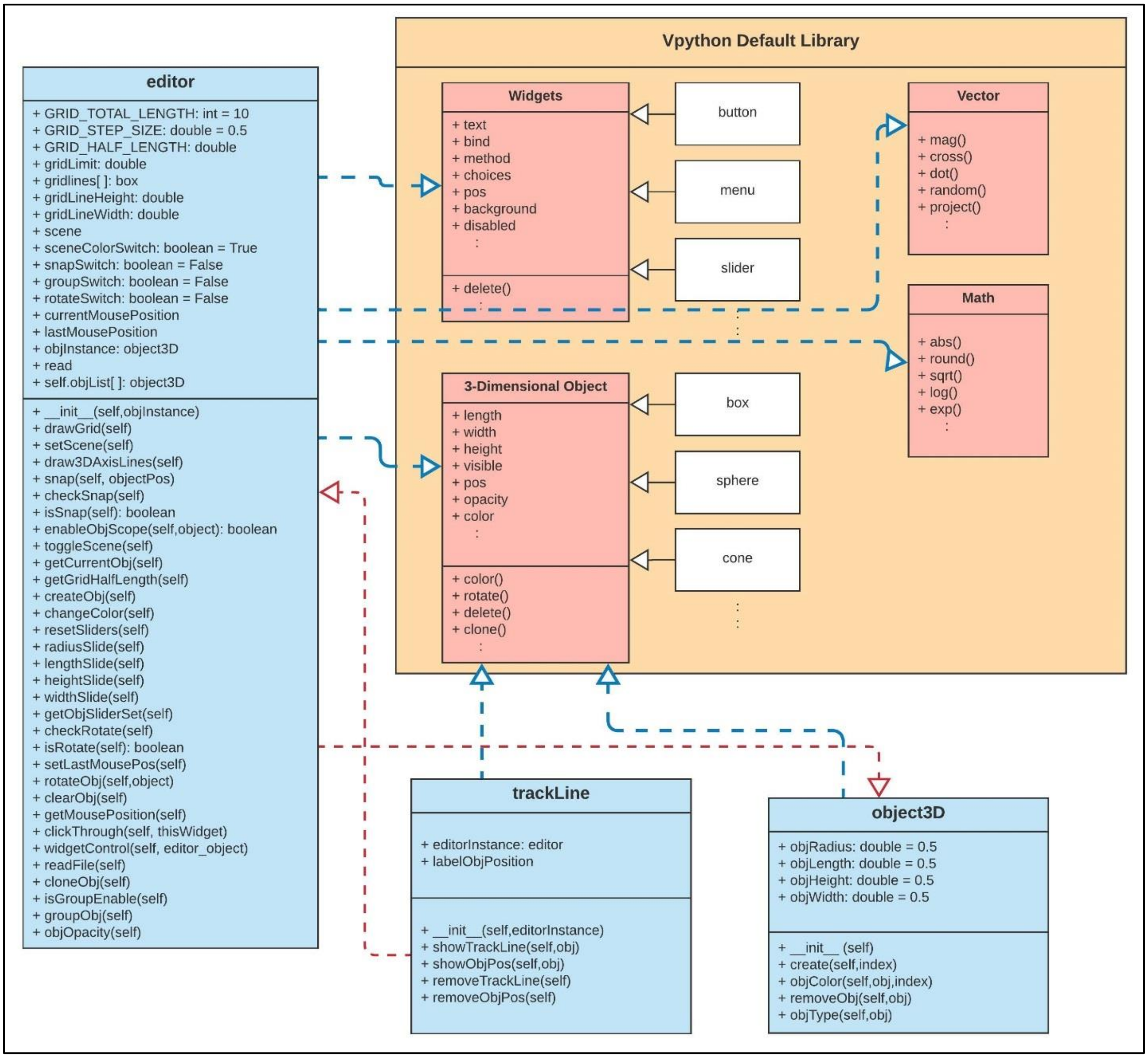

Fig40: Class diagram

7.2 Tool Operation

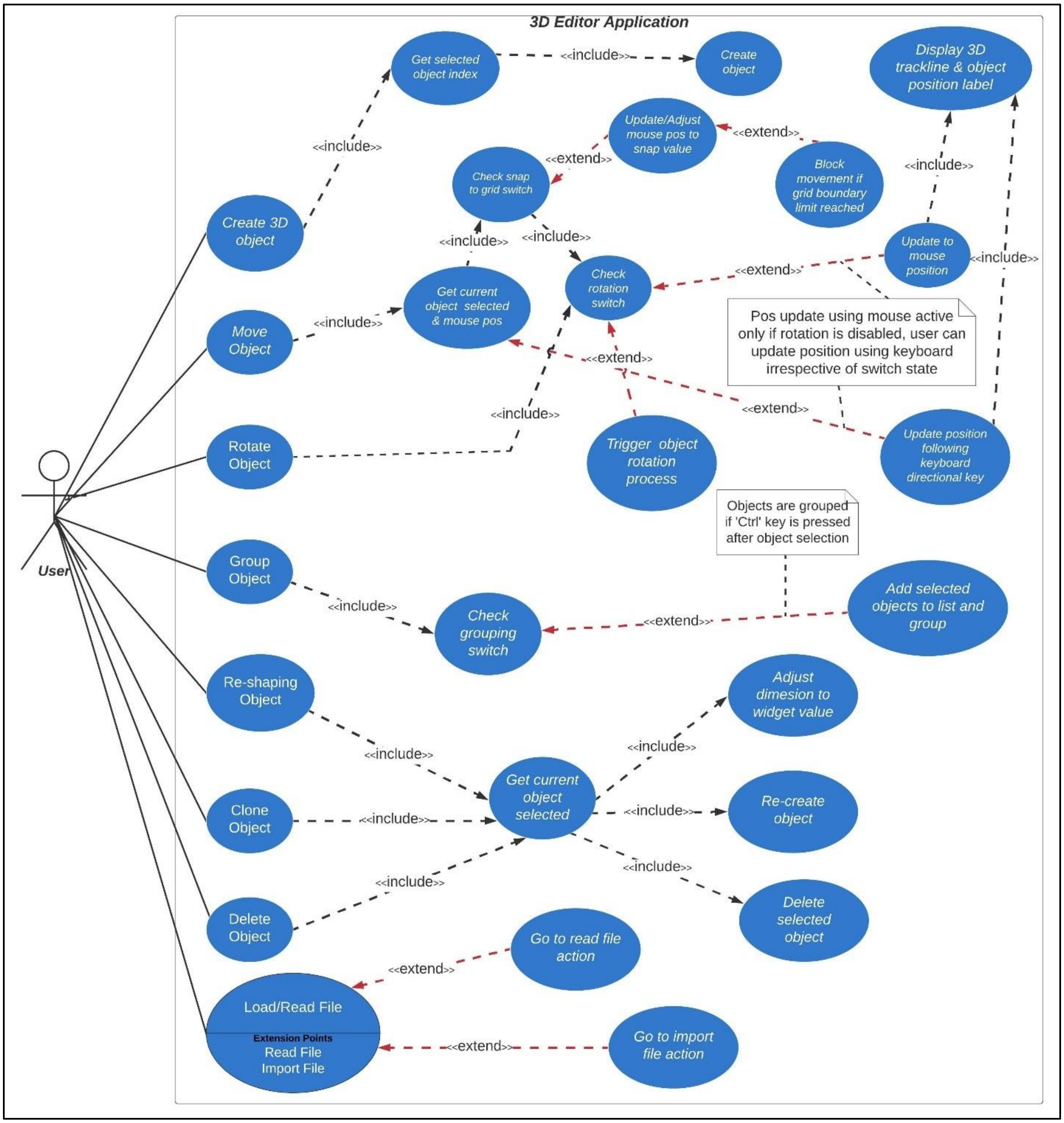

Fig41: 3D Editor use case diagram

The use case diagram (figure 41) describes all use cases associated with the tool, the user can choose to carry out varying activities represented as use cases. Let’s have a concise look at each case

7.2.1 Creating a 3D Object

Here are the operations involved in creating a 3D object:

- Each object defined on the menu widget has a tagged index.
- If a user selects a specific object, the tagged index is provided to the editor class.
- The editor class will pass on this index to object3D class method create() for object creation and display on the scene.

7.2.2 Moving Objects

This could be carried out either by the mouse on the keyboard direction keys. Here are the operations involved in modifying object position on scene using the mouse:

- Get selected objected on scene and its position once mouse left button is pressed.
- Run a check on the status of the snap to grid switch.
- If snap to grid is enabled, trigger snap feature to modify mouse position to nearest grid intersection.
- Check if rotation switch is set, if True, block object move using mouse.
- Check if mouse position exceeds grid limits, if True, block further move.
- If grid limits is not exceeded and rotation disabled, move object to mouse position.

Modifying object on scene using keyboard direction keys:

- Get selected objected on scene and its position
- Check if mouse position exceeds grid limits, if True, block further move.
- If grid limits is not exceeded, move object in an order responding to the direction keys.

Modifying object position using the keyboard function does not care if rotation or snap to grid feature is enabled.

7.2.3 Rotating Objects

Here are the operations involved in rotating an object on the scene using the mouse:

- Check if rotate switch is enabled.
- If enabled, call the rotate method of class editor to trigger intuitive object rotation using mouse interaction.

7.2.4 Grouping Objects

Operations involved in multiple object grouping:

- Check if object grouping switch is enabled.
- If enabled, add user selected objects on scene to an object list.
- Group objects in the list once ‘Ctrl’ button is pressed.

7.2.5 Reshape Objects

Simple object reshaping is achieved through a series of dimension modifications, here are the operations involved:

- Get the current object selected by the user.
- Adjust specified dimensions using the interactive widgets (sliders have been limited to modify dimensions within the value range of 0.1 – 10 units).

7.2.6 Object Cloning

Allow user to recreate selected object, here are the operations involved:

- Get the current object selected by the user.
- Call the editor method cloneObj() to recreate object on the occasion the ‘Clone Object’ interactive button is pressed.

7.2.7 Deleting Object

Operations involved in deleting an object from the editor scene:

- Get the current object selected by the user.
- The selected object will be passed on to object3D method removeObj() by the editor once the delete button is pressed.

7.2.8 Load/Read File

This is extended to two sub options which could be selected by the user.

- Option 'Read File' allows the user to read file displayed as text on the printing region below the scene.
- Option 'Import File' allows the user to import a JavaScript file.

Chapter 8 – Discussion and Result

This section will discuss the progress made at the time of this documentation, challenges experienced, results and proposals for future work.

Developing the 3D tool has been a learning curve, improving areas like my logical approach to coding, improved my skill set on the python language, simplified my complex understanding of object-oriented programing and better usage of Vpython and 3D simulations.

Developing a high-end 3D editor will require the ability to handle complex modelling operation and a bank of static and dynamic features accessible to the user to carry out a much more intrinsic detailed actions on a model. I have been able to show that the list of objects within the Vpython library could be exploited into developing a 3D editor tool with a lot more interactivity as compared to Vpython application usage in its default state. Relevant static features to 3D modeling implanted on the tool include:

- User ability to intuitively rotate a three-dimensional object using mouse interaction.
- Duplicate simple and compound three-dimensional objects
- Provisioned grid to aid users keep track of specific object position and scale.
- Object position labeling, giving the user the ability to tell at an instance the position of an object when navigating through the scene.
- Object three-dimensional track lines along the xyz axes.
- Dynamically re-dimension basic 3D objects with the aid of interactive widgets
- Delete and re-create objects at will.
- The ability to mesh simple objects to form a compound object.
- e.t.c

Unarguably, some of these features provisioned by the tool are not within high-end zone of a professional 3D editor tools out there but it goes a long way verifying the fact that a 3D editor could be designed exploiting the Vpython library and showcases an application of Vpython.

Most challenges experienced while developing the 3D editor tool are posed by the limitations on Vpython – Glowscript. These limitations include:

- Restrain on some built-in functions – This introduces some challenge during program coding and custom function development as certain python built-in function have been reserved for Vpython and cannot be used on a program overlaying it. Instances such as methods of a base class cannot be accessed using the super() built-in function and the switch statement cannot be used as these are reserved for Vpython.
- A much more suitable and functional file operation(read/write) could not be implemented on the tool due to limitations associated with the built-in protections designed on web browsers. This impacts the tool development as Vpython-Glowscript is web based.
- They are no surplus research work or projects done on Vpython out there for reference, most documented content comes within the cycle of developers involved in Vpython development, Vpython.org and Glowscript.org

Currently, users can use the developed tool to manipulate basic 3D objects interactively which has been the intent of the project as it serves more as a proof of concept. Although this has been cited repeatedly on this report, there is a drawback on the tool file operations due to the limitation on Vpython - Glowscript. Figure 42 shows developed tool interface.

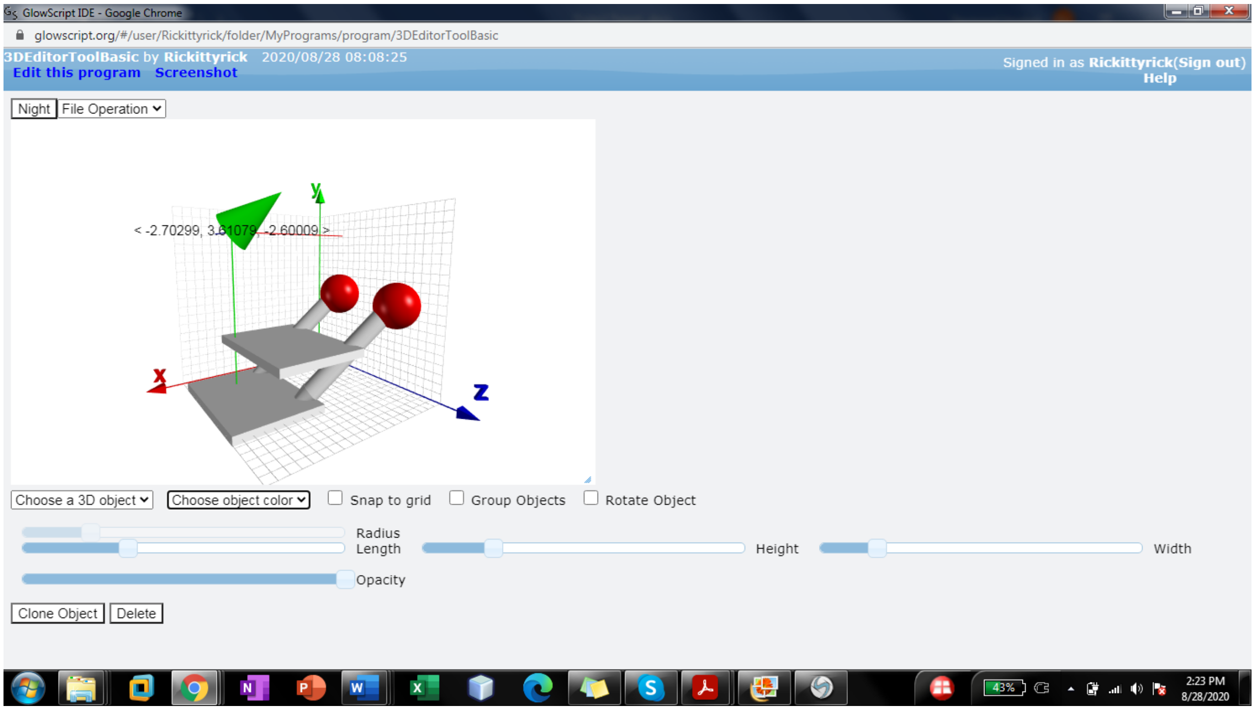

Fig42: 3D Editor Tool Interface

They are whole lot of objects within the Vpython library which could serve as great features to the editor if implemented, these include lighting, textures, object extrusions, sculpturing using curves. I do propose for future work, we have these additional library objects explored and implemented. Also, due to limitations posed by Vpython-Glowscript, it is advised re-implementation of the tool be done on Vpython 7 which is not web based. Vpython 7 will allow developers have more access to the standard Python methods and allow operations like file read/write to be deployed more effectively.

Application link:
https://www.glowscript.org/#/user/Rickittyrick/folder/MyPrograms/program/3DEditorToolBasic

Appendix

```
GlowScript 2.9 VPython


######################################## Program Main ############################################################

myObj3D = object3D() #create an instance of object3D class
thisEditor = editor(myObj3D) #instantiate an editor
thisEditor.setScene() #create and setup scene
thisEditor.drawGrid() #draw 3D grid on scene
thisEditor.draw3DAxisLines() #draw 3D lines on scene
thisEditor.widgetControl(thisEditor) #setup associated widgets
obj = None # object pointer
thisTrackLine = trackLine(thisEditor) #instantiate trackline


#Binding functions
thisEditor.scene.bind("mousemove",movemoveActions) # Call the binding function
thisEditor.scene.bind("mouseup",mouseupActions) # Call the binding function
thisEditor.scene.bind("mousedown", mousedownActions) # Call the binding function
thisEditor.scene.bind('keydown',keydownActions) # Call the binding function
thisEditor.scene.bind('keyup',mouseupActions) # Call the binding function
```

```
def mousedownActions():
    global obj #allow modification to be made to object pointer
    obj = thisEditor.getCurrentObj() #assign current object to object pointer
    thisEditor.resetSliders() #adjust sliders to match current object properties
    thisEditor.getObjSliderSet()   #Adjust sliders to inhereit current object dimension values
    thisEditor.setLastMousePos() #get and project mouse position to xy plane when user press right mouse button

    if thisEditor.isRotate(): #check if user enabled rotation
        thisTrackLine.removeObjPos() #disable trackline on scene
    else:
        thisTrackLine.showObjPos(obj) #show current object position label

    if thisEditor.isGroupEnable(): #checks if user enabled object grouping
        if thisEditor.getCurrentObj() != None:  #check if an object is selected
            thisEditor.objList.append(thisEditor.getCurrentObj()) #add user selected object to object list for grouping
```

```
def movemoveActions():
    global obj #allow modification to be made to object pointer
    temp=thisEditor.getMousePosition() #temproary hold mouse position

    if thisEditor.isSnap(): #check if snap to grid is enabled by user
        temp = thisEditor.snap(temp) #if true, round mouse position to snap on grid intersections

    if thisEditor.enableObjScope(temp): #check if mouse position falls within grid defined boundaries
        if thisEditor.isRotate(): #check if user enabled rotation
            thisTrackLine.removeObjPos() #disable trackline on scene if rotation is enabled
            thisEditor.rotateObj(obj) #rotate the object
        else:
            obj.pos=temp #update object position to mouse position on the grid if rotation is disabled
            thisTrackLine.showTrackLine(obj) #display object trackline
            thisTrackLine.showObjPos(obj) #display object position label


def mouseupActions():
    global obj #allow modification to be made to object pointer
    thisTrackLine.removeTrackLine() #remove object track lines
    thisTrackLine.removeObjPos(obj) #remove object position label
```

```
def keydownActions():
    global obj #allow modification to be made to object pointer
    if obj != None: #check if object is selected
        temp = obj.pos #assign object position to a temproary variable
        dv = 0.05 #object move step value using keyboard
        theKey = keysdown() #get the pressed key
        if 'left' in theKey: #check if left directional key is pressed
            temp.x-=dv #move object towards the left on thex-axis at step value of 0.05
            if (temp.x < -thisEditor.getGridHalfLength()): #check grid boundaries
                temp.x+=dv #if boundary value is surpassed modify to last value within grid boundary
        if 'right' in theKey: #check if right directional key is pressed
            temp.x+=dv #move object towards the right on thex-axis at step value of 0.05
            if (temp.x > thisEditor.getGridHalfLength()): #check grid boundaries
                temp.x-=dv #if boundary value is surpassed modify to last value within grid boundary
        if 'alt' in theKey: #check if 'alt' key is pressed
            if 'up' in theKey: #check if up directional key is pressed with the alt key
                temp.z-=dv #move object away from the user along z-axis at step value of 0.05
                if (temp.z < -thisEditor.getGridHalfLength()): #check grid boundaries
                    temp.z+=dv #if boundary value is surpassed modify to last value within grid boundary
            if 'down' in theKey: #check if down directional key is pressed with the alt key
                temp.z+=dv #move object towards the user along z-axis at step value of 0.05
                if (temp.z > thisEditor.getGridHalfLength()): #check grid boundaries
                    temp.z-=dv #if boundary value is surpassed modify to last value within grid boundary
        elif 'up' in theKey: #check if up directional key is pressed
            temp.y+=dv #move object upwards  on the y-axis at step value of 0.05
            if (temp.y > thisEditor.getGridHalfLength()): #check grid boundaries
                temp.y-=dv #if boundary value is surpassed modify to last value within grid boundary
        elif 'down' in theKey: #check if down directional key is pressed
            temp.y-=dv #move object downwards  on the y-axis at step value of 0.05
            if (temp.y < -thisEditor.getGridHalfLength()): #check grid boundaries
                temp.y+=dv #if boundary value is surpassed modify to last value within grid boundary

        # group objects in objList
        if 'ctrl' in theKey: #check if ctrl key is pressed by user
                compound(thisEditor.objList) #group objects contained in list if ctrl key is pressed
                thisEditor.objList = [] # clear object list


        #delete object if delete key is pressed
        if 'delete' in theKey:
            myObj3D.removeObj(obj) #delete object
            obj = None
            thisTrackLine.removeTrackLine() #disable trackline
            thisTrackLine.removeObjPos(obj) #disable object position


        obj.pos=temp #update object position
        thisTrackLine.showTrackLine(obj)#show 3D trackline
        thisTrackLine.showObjPos(obj) #print object position label on scene

```

```
############################################## Program Classes #########################################################

# Editor class
class editor:

    def __init__(self,objInstance):

        self.GRID_TOTAL_LENGTH = 10
        self.GRID_STEP_SIZE = 0.5
        self.GRID_HALF_LENGTH = self.GRID_TOTAL_LENGTH/2
        self.gridLimit = self.GRID_HALF_LENGTH+self.GRID_STEP_SIZE
        self.gridlines = [] # hold objects which makeup the grid
        self.gridLineHeight = 0.02*self.GRID_STEP_SIZE; #height
        self.gridLineWidth = 0.02*self.GRID_STEP_SIZE; #width
        self.scene = canvas(background=color.black) # Setup scene default background color
        self.sceneColorSwitch = True # switch to enable scene background color toggle
        self.snapSwitch = False # snap to grid switch
        self.rotateSwitch = False # rotate switch
        self.groupSwitch = False # object grouping switch
        self.currentMousePosition = None # track current mouse position on the scene
        self.lastMousePosition = None # track last mouse position on the scene
        self.objInstance = objInstance # hold an instance of class object3D
        self.read = None #Hold content of loaded file
        self.objList = [] #Holds list of objects on the scene
```

```
    def drawGrid(self):
        for eachStraightLine_Back in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(eachStraightLine_Back,0,-self.GRID_HALF_LENGTH),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        axis=vector(0,1,0),
                                        color = color.gray(.9),pickable = False)) #draw boxes and add to grindline list

        for eachCrossedLine_Back in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(0,eachCrossedLine_Back,-self.GRID_HALF_LENGTH),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        axis=vector(1,0,0),
                                        color = color.gray(.9),pickable = False)) #draw boxes and add to grindline list

        ################################################## show bottom grid ##############################################
        for eachStraightLine_Bottom in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(eachStraightLine_Bottom,-self.GRID_HALF_LENGTH,0),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        axis=vector(0,0,1),
                                        color = color.gray(.9),pickable = False)) #draw boxes and add to grindline list

        for eachCrossedLine_Bottom in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(0,-self.GRID_HALF_LENGTH,eachCrossedLine_Bottom),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        color = color.gray(.9),pickable = False))  #draw boxes and add to grindline list
```

```
        ####################################################show right grid#############################################
        for eachStraightLine_Right in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(self.GRID_HALF_LENGTH,0,eachStraightLine_Right),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        axis=vector(0,1,0),
                                        color = color.gray(.9),pickable = False)) #draw boxes and add to grindline list

        for eachCrossedLine_Bottom in range(-self.GRID_HALF_LENGTH, self.gridLimit, self.GRID_STEP_SIZE):
            self.gridlines.append(box( pos=vector(self.GRID_HALF_LENGTH,eachCrossedLine_Bottom,0),
                                        size=vector(self.GRID_TOTAL_LENGTH,self.gridLineHeight,self.gridLineWidth),
                                        axis=vector(0,0,1),
                                        color = color.gray(.9),pickable = False)) #draw boxes and add to grindline list

    #editor class method: setup editor initial scene
    def setScene(self):
        self.scene.range= 9.5; # Prevent scene autoscaling
        self.scene.camera.pos = vec(-11.216, -1.99808e-16, 12.0396) #set camera position
        self.scene.camera.axis = vec(11.216, 1.99808e-16, -12.0396) #set camera axis

    #editor class method: draw 3D axes
    def draw3DAxisLines(self):
        xAxis = arrow(pos=vec(5,-5,-5), axis=vec(-12,0,0), shaftwidth = 0.05,
                headwidth = .5, headlength =1, color=color.red,pickable = False) #draw x-axis idicator
        xAxisText = text(text='x', color=xAxis.color, pos=vec(-6.5,-4.5,-5), billboard = True) #x-axis label

        yAxis = arrow(pos=vec(5,-5,-5), axis=vec(0,12,0), shaftwidth = 0.05,
```

```
                        headwidth = .5, headlength =1, color=color.green,pickable = False) #draw y-axis indicator
        yAxisText = text(text='y', color=yAxis.color, pos=vec(4.2,6.5,-5), billboard = True) #y-axis label

        zAxis = arrow(pos=vec(5,-5,-5), axis=vec(0,0,12), shaftwidth = 0.05,
                        headwidth = .5, headlength =1, color=color.blue,pickable = False) #draw z-axis indicator
        zAxisText = text(text='z', color=zAxis.color, pos=vec(5.2,-4.2,6.5), billboard = True) #z-axis label

    #editor class method: implement snap to grid
    def snap(self, objectPos):
        objectPos.x = self.GRID_STEP_SIZE*round(objectPos.x/self.GRID_STEP_SIZE) #modify x pos in steps of 0.5units
        objectPos.y = self.GRID_STEP_SIZE*round(objectPos.y/self.GRID_STEP_SIZE) #modify y pos in steps of 0.5units
        objectPos.z = self.GRID_STEP_SIZE*round(objectPos.z/self.GRID_STEP_SIZE) #modify z pos in steps of 0.5units
        return objectPos # return snapped position


    #editor class method: allow toggle of snap to grid switch
    def checkSnap(self):
        if !self.isSnap():
            self.snapSwitch = True
        else:
            self.snapSwitch = False


    #editor class method: get snap switch state
    def isSnap(self):
        return self.snapSwitch
```

```
    #editor class method: get object grouping switch state
    def isGroupEnable(self):
        return self.groupSwitch

    #editor class method: allow toggle of object grouping switch
    def groupObj(self):
        if !self.isGroupEnable():
           self.groupSwitch = True
        else:
            self.groupSwitch = False

    #editor class method: check object boundaries within grid
    def enableObjScope(self,object):
        if ((object.x <= (self.GRID_HALF_LENGTH)) & (object.x >= -(self.GRID_HALF_LENGTH))
            & (object.y <= (self.GRID_HALF_LENGTH)) & (object.y >= -(self.GRID_HALF_LENGTH))
            & (object.z <= (self.GRID_HALF_LENGTH)) & (object.z >= -(self.GRID_HALF_LENGTH))): #limit object scope to grid
            return True

```

```
    #editor class method: toggle editor scene color between day & night
    def toggleScene(self):
        if self.sceneColorSwitch: #check if switch is set to true
            self.scene.background= color.white #change editor scene color to white
            self.sceneColorSwitch = False #change switch to false
            self.sceneColorButton.text ="Night" #change displayed text on widget to "Night"
        else:                                   #otherwise
            self.scene.background= color.black #change editor scene color to black
            self.sceneColorSwitch = True #change switch to true
            self.sceneColorButton.text="Day" #change displayed text on widget to "Day"

    #editor class method: get current object picked by mouse pointer
    def getCurrentObj(self):
        theObj = self.scene.mouse.pick
        return theObj

    #editor class method: get editor grid half length
    def getGridHalfLength(self):
        return self.GRID_HALF_LENGTH

    #editor class method: create user selected object from menu widget
    def createObj(self):
        self.objInstance.create(self.objMenu.index) #call create method of class object3D to draw object
        self.objMenu.index = 0 #reset widget menu
```

```
    #editor class method: change selected object color
    def changeColor(self):
        if self.getCurrentObj() != None: #check selected object, ignore if none
            self.objInstance.objColor(self.getCurrentObj(),
                                      self.colorMenu.index)#call objColor method (class object3D) to change color
        self.colorMenu.index = 0 #reset widget menu

    #editor class method: adjust sliders to match current object properties
    def resetSliders(self):
        if isinstance(self.getCurrentObj(),sphere): #check if current object is sphere
            self.length_slider.value = 0 #reset length slider value to zero
            self.height_slider.value = 0 #reset height slider value to zero
            self.width_slider.value = 0 #reset width slider value to zero
            self.length_slider.disabled = True #gray out length slider
            self.height_slider.disabled = True #gray out height slider
            self.width_slider.disabled = True #gray out width slider
            self.radius_slider.disabled = False #enable radius slider
        else:
            self.radius_slider.value = 0 #reset radius slider value to zero
            self.length_slider.disabled = False #enable length slider
            self.height_slider.disabled = False #enable height slider
            self.width_slider.disabled = False #enable width slider
            self.radius_slider.disabled = True #gray out radius slider

```

```
    #editor class method: adjust object dimesions to slider values
    def radiusSlide(self):
        self.getCurrentObj().radius = self.radius_slider.value #set object radius to radius_slider value

    def lengthSlide(self):
        self.getCurrentObj().length = self.length_slider.value #set object length to length_slider value

    def heightSlide(self):
        self.getCurrentObj().height = self.height_slider.value #set object height to height_slider value

    def widthSlide(self):
        self.getCurrentObj().width = self.width_slider.value #set object width to width_slider value


    #editor class method: Adjust sliders to inhereit current object dimension values
    def getObjSliderSet(self):
        self.radius_slider.value = self.getCurrentObj().radius
        self.length_slider.value = self.getCurrentObj().length
        self.width_slider.value = self.getCurrentObj().width
        self.height_slider.value = self.getCurrentObj().height
        self.opacity_slider.value = self.getCurrentObj().opacity
```

```
    #editor class method: toggle rotate switch
    def checkRotate(self):
        if !self.rotateSwitch:
            self.rotateSwitch = True
        else:
            self.rotateSwitch = False

    #editor class method: get rotate switch state
    def isRotate(self):
        return self.rotateSwitch

    #editor class method: set last mouse position
    def setLastMousePos(self):
        #get and project mouse position to xy plane when right click button pressed
        self.lastMousePosition = self.scene.mouse.project(normal=vec(0,0,1))

    #editor class method: object rotation
    def rotateObj(self,object):
        #project current mouse position onto a 2D plane
        self.currentMousePosition = self.scene.mouse.project(normal=vec(0,0,1))
        self.move = self.currentMousePosition - self.lastMousePosition # defines the resultant vector

#     if (scene.mouse.pick!= None):
        object.rotate(angle=(-self.move.mag)*0.1, axis=self.move.cross(vec(0,0,1))) #rotate object
        self.lastMousePosition = self.currentMousePosition #update last mouse position
```

```python
    #editor class method: Clear selected object
    def clearObj(self):
#        self.objInstance.remove(self.getCurrentObj())
        if self.getCurrentObj() != None:
            self.objInstance.removeObj(self.getCurrentObj()) #delete current object

    #editor class method: return mouse position
    def getMousePosition(self):
        return self.scene.mouse.pos

    #editor class method: object cloning
    def cloneObj(self):
        if self.getCurrentObj() != None:
            self.getCurrentObj().clone(pos =vec(-2,0,0)) #clones current object.

    #editor class method: modify current object opacity
    def objOpacity(self):
        if self.getCurrentObj() != None:
            self.getCurrentObj().opacity = self.opacity_slider.value #set object transparency to opacity slider value

```

```python
    #editor class method: File Operations
    def readFile(self):
            if self.fileMenu.index == 1:
                self.fileMenu.index = 0 #reset menu
                print_options(delete=True) #clear printing region
                self.read = read_local_file(self.scene.title_anchor) #trigger read operation
                print("-------------------------------------------------------------------")
                print("File Name: "+self.read.name) # The file name
                print("File Size: "+self.read.size+"kb") # File size in bytes
                print("File Type: "+self.read.type) # What kind of file
                print("Creation Date: " + self.read.date) # Creation date if available
                print("-------------------------------------------------------------------")
                print(self.read.text) # The file contents
                print("################ End #################")

            if self.fileMenu.index == 2:
                self.fileMenu.index = 0 #reset menu
                #create a widget input allowing user to enter file directory
                xfile = winput(prompt ="Import JavaScript File",
                                type = "string", text = "Enter directory")
                self.get_library(xfile) #import javascript file

```

```python
    #editor class method: resused method, points to actual called method for each widget
    def clickThrough(self, thisWidget):
        eval(thisWidget.method)

    #editor class method: class widgets
    def widgetControl(self, editor_object):

        self.sceneColorButton = button(text="Day", pos=self.scene.title_anchor, bind=editor_object.clickThrough,
                                       editor_object=editor_object,
                                        method="thisWidget.editor_object.toggleScene()") #Switch scene background color


        self.objMenu = menu( choices=['Choose a 3D object','Sphere','Box','Cylinder','Cone'],
                            bind=editor_object.clickThrough,
                            editor_object=editor_object,
                            method="thisWidget.editor_object.createObj()") #Creates object menu
        self.scene.append_to_caption('    ')

        self.colorMenu = menu( choices=['Choose object  color','Red', 'Green', 'Blue','default'],
                               bind=editor_object.clickThrough,
                               editor_object=editor_object,
                               method="thisWidget.editor_object.changeColor()" )#Modify object color
        self.scene.append_to_caption('    ')

```

```
        self.snapCheckBox = checkbox(bind=editor_object.clickThrough,
                                     text='Snap to grid',editor_object=editor_object,
                                     method="thisWidget.editor_object.checkSnap()")#Enable snap to grid feature
        self.scene.append_to_caption('    ')

        self.compoundCheckBox = checkbox(bind=editor_object.clickThrough,
                                         text='Group Objects',editor_object=editor_object,
                                         method="thisWidget.editor_object.groupObj()")#Trigger the compound object feature
        self.scene.append_to_caption('    ')

        self.rotateCheckBox = checkbox(bind=editor_object.clickThrough,
                                       text='Rotate Object',editor_object=editor_object,
                                       method="thisWidget.editor_object.checkRotate()")#Enable rotation
        self.scene.append_to_caption('\n\n')

        self.radius_slider = slider( bind=editor_object.clickThrough, min=0.1, max=5,
                                     editor_object=editor_object,
                                     method="thisWidget.editor_object.radiusSlide()")#Radius slider bar
        self.scene.append_to_caption('Radius\n')

        self.length_slider = slider( bind=editor_object.clickThrough, min=0.1, max=10,
                                     editor_object=editor_object,
                                     method="thisWidget.editor_object.lengthSlide()")#Length slider bar
        self.scene.append_to_caption('Length  ')

```

```
        self.height_slider = slider( bind=editor_object.clickThrough , min=0.1, max=10,
                                     editor_object=editor_object,
                                     method="thisWidget.editor_object.heightSlide()")#Height slider bar
        self.scene.append_to_caption('Height  ')

        self.width_slider = slider( bind=editor_object.clickThrough, min=0.1, max=10,
                                    editor_object=editor_object,
                                    method="thisWidget.editor_object.widthSlide()")#Width slider bar
        self.scene.append_to_caption('Width\n\n')

        self.opacity_slider = slider( bind=editor_object.clickThrough, min=0.2, max=1,
                                      editor_object=editor_object,
                                      method="thisWidget.editor_object.objOpacity()")#call method to adjust object opacity
        self.scene.append_to_caption('Opacity\n\n')

        self.fileMenu = menu( pos =self.scene.title_anchor, choices=['File Operation','Load File', 'Import File'],
                              bind=editor_object.clickThrough,
                              editor_object=editor_object, method="thisWidget.editor_object.readFile()" )#File operation

        self.cloneButton = button(text="Clone Object",  bind=editor_object.clickThrough,
                                  editor_object=editor_object,
                                  method="thisWidget.editor_object.cloneObj()") #Trigger object cloning
        self.scene.append_to_caption(' ')

```

```
        self.deleButton = button(text="Delete",  bind=editor_object.clickThrough,
                                 editor_object=editor_object,
                                 method="thisWidget.editor_object.clearObj()") #call method to delete currrent object


######################################################## class trackLine ##################################################
class trackLine:
    def __init__(self,editorInstance):
        self.editorInstance = editorInstance
        self.labelObjPosition = label( pos=scene.mouse.pos, text=scene.mouse.pos, box = False,visible = False, opacity = 0)

        #Define track lines on all 3 dimensional axis, not visible by default
        self.trackLineX = box(pos=vector(0,0,0), size=vector(self.editorInstance.GRID_HALF_LENGTH,
                                        5*self.editorInstance.gridLineHeight,
                                        5*self.editorInstance.gridLineWidth),
                                        color = color.red, visible = False, axis = vector(1,0,0))
        self.trackLineY = box(pos=vector(0,0,0), size=vector(self.editorInstance.GRID_HALF_LENGTH,
                                        5*self.editorInstance.gridLineHeight,
                                        5*self.editorInstance.gridLineWidth),
                                        color = color.green, visible = False, axis = vector(0,1,0))
        self.trackLineZ = box(pos=vector(0,0,0), size=vector(self.editorInstance.GRID_HALF_LENGTH,
                                        5*self.editorInstance.gridLineHeight,
                                        5*self.editorInstance.gridLineWidth),
                                        color = color.blue, visible = False, axis = vector(0,0,1))

```

```
    #trackLine class method: show track lines
    def showTrackLine(self,obj):
        self.trackLineX.visible = True #Turn on x-axis track line visibility
        self.trackLineY.visible = True #Turn on y-axis track line visibility
        self.trackLineZ.visible = True #Turn on z-axis track line visibility

        #Calculate and track object position along the x-axis
        self.trackLineX.pos.x = ((self.editorInstance.GRID_HALF_LENGTH + obj.pos.x)/2)
        self.trackLineX.pos.y = (obj.pos.y)
        self.trackLineX.pos.z = (obj.pos.z)
        self.trackLineX.size.x = self.editorInstance.GRID_HALF_LENGTH - obj.pos.x

        #Calculate and track object position along the y-axis
        self.trackLineY.pos.x = (obj.pos.x)
        self.trackLineY.pos.y = (-self.editorInstance.GRID_HALF_LENGTH + obj.pos.y)/2
        self.trackLineY.pos.z = (obj.pos.z)
        self.trackLineY.size.x = self.editorInstance.GRID_HALF_LENGTH + obj.pos.y

        #Calculate and track object position along the z-axis
        self.trackLineZ.pos.x = (obj.pos.x)
        self.trackLineZ.pos.y = (obj.pos.y)
        self.trackLineZ.pos.z = (-self.editorInstance.GRID_HALF_LENGTH + obj.pos.z)/2
        self.trackLineZ.size.x = self.editorInstance.GRID_HALF_LENGTH + obj.pos.z

```

```
    #trackLine class method: display object position label on scene
    def showObjPos(self,obj):
        self.labelObjPosition.visible = True
        self.labelObjPosition.pos = obj.pos
        self.labelObjPosition.text.x = obj.pos.x
        self.labelObjPosition.text.y = obj.pos.y
        self.labelObjPosition.text.z = obj.pos.z

    #trackLine class method: disable object track line
    def removeTrackLine(self):
        self.trackLineX.visible = False #Turn on x-axis track line visibility
        self.trackLineY.visible = False #Turn on y-axis track line visibility
        self.trackLineZ.visible = False #Turn on z-axis track line visibility

    #trackLine class method: disable object position label
    def removeObjPos(self):
        self.labelObjPosition.visible = False

```

```python
############################################### Class object3D ##############################################

class object3D:

    def __init__ (self):
        self.objRadius = 0.5 #default object radius
        self.objLength = 0.5 #default object length
        self.objHeight = 0.5 #default object height
        self.objWidth = 0.5  #default object width

    #object3D class method: draw selected object
    def create(self,index):

        if index==1:
            return sphere(pos=vec(0,0,0), radius = self.objRadius, visible = True, pickable = True)
        if index==2:
            return box(pos=vec(0,0,0),
                       length=self.objLength,height=self.objHeight,
                       width=self.objWidth, visible = True, pickable = True) #display sphere on grid
        if index==3:
            return cylinder(pos=vec(0,0,0),
                            radius = self.objRadius,
                            axis= vec(self.objLength,self.objHeight,self.objWidth),
                            visible = True, pickable = True) #display
```

```python
        if index==4:
            return cone(pos=vector(0,0,0),
                        radius = self.objRadius,
                        axis= vec(self.objLength,self.objHeight,self.objWidth),
                        visible = True, pickable = True) #display sphere on grid

     #object3D class method: modify object color
    def objColor(self,obj,index):
        if index ==1:
            obj.color = color.red #set red
        if index ==2:
            obj.color = color.green #set green
        if index ==3:
            obj.color = color.blue #set blue
        if index == 4:
            obj.color = vec(1,1,1) #set to default gray color

    #object3D class method: delete object
    def removeObj(self,obj):
        obj.pickable = False
        obj.visible = False
        del obj

    #object3D class method: get object  type
    def objType(self,obj):
        return type(obj)
```

References

[1] http://vpython.org. Retrieved 2019-11-10

[2] B. Sherwood, R. Chabay, "VPYTHON: 3D PROGRAMMING FOR ORDINARY MORTALS", MPTL14 2009, Udine 23-27 September 2009, p. 1. Accessed on Nov. 2, 2019. [Online]. Available: http://www.fisica.uniud.it/URDF/mptl14/ftp/full_text/WS3%20Full%20Paper.pdf

[3] https://www.glowscript.org/docs/VPythonDocs/index.html. Retrieved 2019-11-09

[4] C. Blakeney, M. Dube, H. Close, "Abstract: J6.00003 : Developing a visual programming editor for VPython", Accessed on Nov.3, 2019. [Online]. Available: http://meetings.aps.org/link/BAPS.2016.TSF.J6.3

[5] GitHub, Inc. (2019). RapydScript, Accessed on Nov.3, 2019. [Online]. Available: https://github.com/vpython/glowscript

[6] GitHub, Inc. (2019). RapydScript, Accessed on Nov.13, 2019. [Online]. Available: https://github.com/kovidgoyal/rapydscript-ng

[7] https://guigui.developpez.com/cours/python/vpython/en/?page=object#Lbox Retrieved 2020-03-19

[8] G.M. Nielson, D.R. Olsen, "Direct manipulation techniques for 3D objects using 2D locator devices", I3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics, January 1987, p.179 -180. Accessed on July 27, 2020. [Online]. Available:

https://dl.acm.org/doi/pdf/10.1145/319120.319134?casa_token=NbooSlu2HdUAAAAA:-XB0L-LvYG2fXVxSAC5Nc7aesUvm7DKr2eozEQ5_CGh9645gksuI2IXRTBlNk_LoztmROLCRNQ0s-Q

[9] R. Kasparian, "True to life 3D object rotations using a mouse – A new perspective on interpreting the mouse's movement". Article accessed on the July, 3rd, 2020. [Online]. Available:

http://www.quantimegroup.com/solutions/pages/Article/Article.html

[10] https://devbasherwo.org/docs/VPythonDocs/mouse_click.html Retrieved 2020-07-19

[11] B. Sherwood, R. Chabay, “ Vpython Architecture”. Article accessed on Decemeber, 13th, 2019.[Online].
Available:

https://vpython.org/contents/VPythonArchitecture.pdf

[12] https://www.glowscript.org/docs/VPythonDocs/files.html. Retrieved 2020-01-09

ACKNOWLEDGEMENTS

Firstly, I would like to wholeheartedly thank my project supervisor Dr. Richard Conway, my course director Prof. Hussain Mahdi and the department of ECE, University of Limerick for sharing their colossal knowledge and resolving all my doubts during the entire course of this project. I believe without their valuable guidance and feedback; this project could not have been achieved.

I would also like to express my gratitude to all the staff members of ECE department at University of Limerick for their excellent co-operation.

Heartfelt thanks to my supporting friends Waqas Latif, a software engineer at JLR, Shannon and Mohammed Faud for their valuable support. Last but not the least, I would like to thank my parents, family and friends for their constant motivation and words of wisdom.