ANN MCNAMARA

USING PYTHON SCRIPTING TO ENHANCE WORKFLOW IN AUTODESK MAYA: COURES NOTES

Contents

Listings

1
Introduction

1.1 Motivation

Coding empowers automation. Scripts can handle mundane and repetitive tasks in an efficient and precise manner. This course will offer will use an hands-on interactive format to walk attendees through representative scripting projects, selected to be useful for everyday workflows. It is intended to be an intermediate course. The goal is to cover provide enough information for attendees to build on later. Python scripting can automate many tasks in Maya, from running simple commands to developing plug-ins. Attendees will learn how to automate simple tasks using the magic of scripting, through four distinct projects. The course will placing objects randomly in a scene, designing custom User Interfaces (GUIs) in Maya, scripting MASH (motion graphic) networks, and scripting a leg rig, with foot-roll. By the end of the course, attendees should walk away with a solid understanding of the power Python scripting and Maya commands provide, and the the ability to build their own advance projects for Maya. This course will equip attendees with the tools, confidence, and initiative to explore more advanced scripts independently. Attendees should have programming experience, preferably in Python, but a solid grasp of the foundational programming constructs should suffice. Attendees should have Autodesk Maya, Python, and Visual Studio Code pre-loaded on their devices if they intend to follow along.

1.2 Focus Areas

This course will focus on a handful of small, yet meaningful, projects that use the of the Python programming language to create effective scripts that seamlessly execute in Autodesk Maya. Each project is useful in it's own right, but also serves as a foundation to build larger projects. The hope is that introducing projects that demonstrate the power of scripting will motivate programmers and artists to experiment and explore the possibilities. We will spend about 20 to 25 minutes on each project to allow for questions.

1. Welcome and Introduction - 5 minutes
2. Getting Set Up (Python/Maya/VisualStudioCode) - 5 minutes
3. Project 1: Randomly placing objects in a scene - 20 minutes
4. Project 2: Creating a Custom GUI - 20 minutes
5. Project 3: Programming MASH (motion graphics) Networks - 25 minutes

6. Project 4: Creating a Leg Rig with Foot Roll - 25 minutes

7. Wrap Up, Resources, Next Steps, Questions - 5 minutes

1.3 *A note on code listings*

Each code listing, as shown in Listing 1.1, represents a python file that is also included with these Course Notes. All the files can be downloaded from the authors web page, linked here. Comments appear in green and are included to provide documentation for the human reader. Python ignores any text after a #.

```python
# A first example in python
# print statements will display the
# result of evaluating the expressions
# everything in green
# after a # is a comment and is for the
# human reader and ignored in the program

print 10            # output 10
print 10 + 10       # output 20
print 10 * 10       # output 100
print 5  * 10       # output 50
print 40/10         # output 4
print 42/10         # output 4 (only whole number portion)

print 'Hello'       # output Hello
```

Listing 1.1: python code example (1.1_firstExample.py)

2
Setting Up Python & Autodesk Maya

There are some steps that are necessary to set up communication between Python and Autodesk Maya.

1. Download and install Autodesk Maya 2022 - you can select the educational version if you are affiliated with a university by providing your user id. (or you can download a 30-day free trial while you wait for your id to be approved. For this course we just need Maya 2022 (there is no need for additional installations such as Arnold, Bitfrost etc)
   - Student Version
   - 30 Day Trial Version
2. Download and install Visual Studio Code. Visual Studio Code is a text editor that allows extensive and specialized extensions/plugins that help speed up your workflow. Youcan think of VSC as Notepad/-TextEdit but with extendable functionality.
   - Visual Studio Code
3. Install Maya Code (click the green install button and open in Visual Studio Code). This extension allows you to see the Maya hierarchy within Visual Studio code, colors the words of your script for readability (syntax highlighting), and provides intellisense (smart autocomplete) for MEL scripting.
   - Maya Code
4. Put the following two lines of code in a file called **userSetup.mel**. Maya will execute this file every time it opens - the code basically tells VSCode how to "talk" to Autodesk Maya. You can create the file in notepad or any text editor you like (even VSCode)

   ```
   commandPort -name "localhost:5678" -sourceType "python";
   commandPort -name "localhost:7001" -sourceType "mel";
   ```
5. Put the following line of code in a file called **userSetup.py**. Maya will automatically execute this file on startup when Maya opens. We are just going to save ourselves from typing this line of code over and over.

   ```
   import maya.cmds as cmds
   ```
6. Be careful with the file names. They need to be *exactly* as listed in bold
7. Place both **userSetup.mel** and **userSetup.py** in the following folder:

- MAC:
  `/Users/username/Library/Preferences/Autodesk/maya/2020/scripts`
  For example on my mac my folder is called
  `/Users/annmcnamara/Library/Preferences/Autodesk/maya/2020/scripts`
- PC:
  `..\MyDocuments\maya\<Version>\scripts`
  The version is 2022 as that is what you just downloaded so the path should be
  `..\MyDocuments\maya\2022\scripts`

8. Navigate to VSCode extension directory. Because the VSCode extensions folder is hidden, you normally won't be able to find it with your file browser. Instead, make sure MayaCode is already installed and try these:

   On Windows machines, click Start > Run and paste the following in, then click Run:

   ```
    %USERPROFILE%\.vscode\extensions\saviof.mayacode-1.4.0\out
   ```

   On Mac machines, open a Finder window, then go to the menu bar and click Go > Go to Folder... and paste the following in, then click Go:

   ```
   ~/.vscode/extensions/saviof.mayacode-1.4.0/out
   ```

9. Open **extension.js** in a text editor Find the following line using Ctrl-F; it is around line 236:

   ```
   cmd = `python("execfile('${posixPath}')")`;
   ```

   Replace it with this:

   ```
   cmd = `python("exec(open('${posixPath}').read())")`;
   ```

   Be sure to keep the indentation the same!

10. Save the file extension.js and **relaunch both VSCode and Maya 2022**. You should now be able to send your code to Maya 2022 from within VSCode.

11. Once you have the files correctly in the directory open Maya. Open Visual Studio Code if you don't have it open and create a new python script (File->New). Type in the following two lines of code and save the file as **test.py** (File->Save As) - make sure you save it as test.py, the .py extension at the end tells VSCode that it is a python file.

    ```
    cmds.polyCube(name='myCube')
    cmds.polySphere(name="mySphere")
    ```

12. Now the moment of truth, right-click anywhere in your script test.py window in Visual Studio Code and you should see an option called Maya: Send Python Code to Maya, or you can use use the hotkey combo: shift plus ctrl plus M, since you will be doing this all throughout it might be faster. The very first time you do this you may need to scroll down to the bottom and open the command palette. if you start typing Maya at the > then select Maya: Send Python Code to Maya, it should work thereafter.

13. If you have followed all the instructions carefully you should see a sphere and a cube in your Maya window with the names mySphere and myCube, as shown in Figure 2.1. When first drawn the sphere will be on top of the cube so you wont be able to see the cube unless you move the sphere (or move the cube).

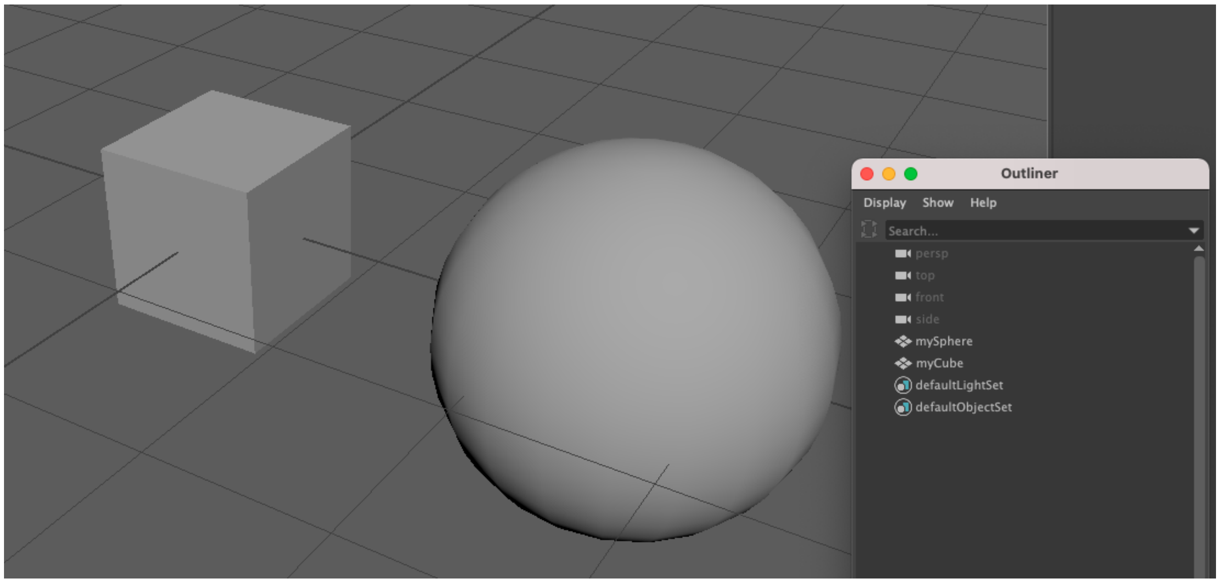


Figure 2.1: A sphere and a cube using python for Maya.

2.1 *A Note on Maya.cmds*

Maya provides the **cmds** API to allow us to program Maya functions. At the beginning of each Python program that uses Maya cmds, we must first **import the maya.cmds API**.

```
import maya.cmds as cmds
```

Here we could import maya.cmds as any name we would like, but a popular convention is to use cmds. I have seen it written as

```
import maya.cmds as mc
```

where mc is short for maya.cmds, again you can call it anything you like. For this course we will use the first way and then precede our calls to the API with cmds. An example to to create a cube would be:

```
cmds.polyCube()
```

This line of code will create a polygon cube in Maya with all the default values. If we want to set specific values at creation time we can look to the Maya command reference, Figure **??** If we search for **polyCube** the documentation will list all the *flags*, or attributes, available to set. Each flag has a long name and an abbreviation. The examples in this course use the long names for completeness but once you get used to the flags it's very handy to be able to use the abbreviations.

The polyCube() command has the following flags, axis, caching, constructionHistory, createUVs, depth, height, name, nodeState, object, subdivisionsDepth, subdivisionsHeight, subdivisionsWidth, subdivisionsX, subdivisionsY, subdivisionsZ, texture, width. It is not necessary to remember all of these, or to use them every time you create a cube. It is good to know what is available. An example to create a cube called "box" with a width of three and height of two would be as follows. We are going to store this in the variable called aBox. This is separated from the name we set to "box" using the **-name** flag. that name, "box" will be used inside Maya to name the object.

```
cmds.polyCube(name='box', width=3, height=2))
```

Once you are familiar with the long names you can use the short names. The following line of code is exactly the same as the line above, but the abbreviated versions of the flag names are used.

```
cmds.polyCube(n='box', w=3, h=2))
```

The documentation typically includes an example of how to use the Maya cmd. This example is taken directly from the documentation and illustrates how to create a cube and query the width attribute. The documentation does use the abbreviated version of the flags.

```
import maya.cmds as cmds

cmds.polyCube( sx=10, sy=15, sz=5, h=20)

#result is a 20 units height rectangular box

#with 10 subdivisions along X, 15 along Y and 20 along Z.

cmds.polyCube( sx=5, sy=5, sz=5 )

#result has 5 subdivisions along all directions, default size

# query the width of a cube

w = cmds.polyCube( 'polyCube1', q=True, w=True )
```

For completeness, the above code is repeated but with the full names for each flag.

```
import maya.cmds as cmds

cmds.polyCube( subdivisionsX=10, subdivisionsY=15, subdivisionsZ=5, height=20)

#result is a 20 units height rectangular box

#with 10 subdivisions along X, 15 along Y and 20 along Z.

cmds.polyCube(subdivisionsX=5,subdivisionsY=5, subdivisionsZ=5 )

#result has 5 subdivisions along all directions, default size

# query the width of a cube
```

```
cubeWidth = cmds.polyCube( 'polyCube1', query=True, width=True )
```

Both code segments produce the exact same results. The second one where the flags are written in full is a little easier to read and understand when you are starting out. Once the flags become familiar it is easier to use the abbreviated versions. Note while this example stored the width in the variable called **cubeWidth** (or w) we do no actually do anything with this value, if we wanted to examine it we could print it out.

```
print cubeWidth
```

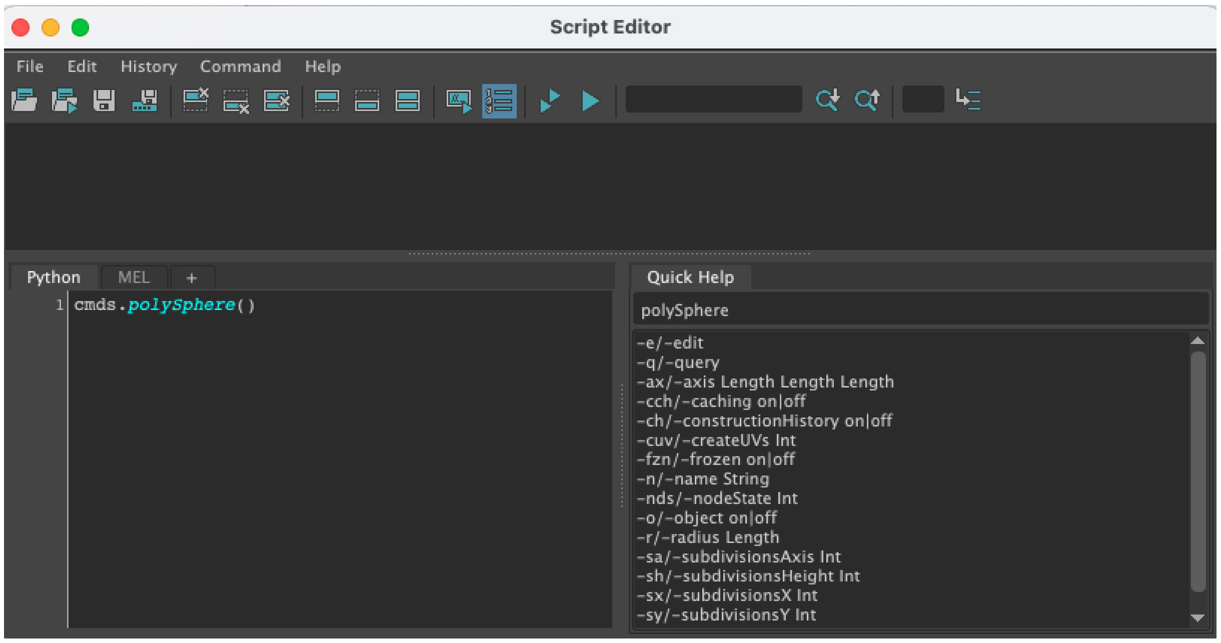
Using maya.cmds in Python allows us to create many objects including spheres, torii, cylinders, planes and disc. To create a sphere, we can use polySphere()

```
cmds.polySphere() # Create a default sphere

cmds.polySphere(name='globe', radius=2) # Create a sphere with the name globe and a radius of 2

cmds.polySphere(n='globe', r=2) # Same as preceding line with flag abbreviations
```

2.1.1 *Getting Quick Help on flags*



```
cmds.polySphere()
```

```
polySphere
-e/-edit
-q/-query
-ax/-axis Length Length Length
-cch/-caching on|off
-ch/-constructionHistory on|off
-cuv/-createUVs Int
-fzn/-frozen on|off
-n/-name String
-nds/-nodeState Int
-o/-object on|off
-r/-radius Length
-sa/-subdivisionsAxis Int
-sh/-subdivisionsHeight Int
-sx/-subdivisionsX Int
-sy/-subdivisionsY Int
```

Figure 2.2: Getting Quick Help

The script editor in Maya has a nice feature to enable a listing of all the flags called *Quick Help*. To activate this, simply type the maya cmd into the Python tab of the script editor, right click and choose *Quick help*. This reveals a list of all possible flags along with the data type of the expected argument. Figure 2.2 shows the Quick Help for cmds.polySphere(). We can see, for example, that **-r** is the abbreviation for **radius**, and that we need to provide a length for the radius. We can also see that -sx is short for subdivisionsX and we are expected to provide a **int** data type.

2.2 *Summary*

There are several steps necessary to complete before communication between Visual Studio Code and Maya 2022 is established. This section outlined those steps. It also described the maya.cmds library, and how to access the documentation that accompanies the functions available in that library.

3
Project 1: Randomly Placing Objects in a Scene

NOTE: SAMPLE NOTES ONLY, THIS SECTION WILL INCLUDE ADDITIONAL RESOURCES UPON ACCEPTANCE

3.1 Introduction

In this project we will learn how to place objects randomly in a scene. We will begin by placing objects randomly on the x, y, and z planes individually, then show how we can combine them.

3.2 The Random Library

The random library in Python allows us to generate (pseudo) random numbers. To use the library, as with maya.cmds, we must first import the library using the following line of code.

```
import random
```

If you prefer you can import random *as* an alias, like we did with cmds, we could use:

```
import random as rdm
```

The method random will return a float between 0 and 1. While the method randint(start, stop) returns a random number between the given range. These are the two methods we will use most in this course. The documentation for random will provide more insight into the methods available through the random library.

NOTE: SAMPLE NOTES ONLY, THIS SECTION WILL INCLUDE SIGNIFICANT ADDITIONAL RESOURCES UPON ACCEPTANCE

3.3 Code

For this project we will begin with a sphere as our object. We will just use the default sphere, then scale it later as we wish. To create a sphere we use the maya command **polySphere**. There are several flags (parameters) available for polySphere(), but for now we will just use the defaults to create a unit sphere.

```python
#import the maya commands library
import maya.cmds as cmds

cmds.polySphere() # This will create a default (unit) sphere
```

3.4 *Creating Multiple Objects*

To create multiple objects we will use a for loop. We first create a variable to hold the number of objects we want to create. Just so we can see everything working we will start out with five objects then increase than number later. Our for loop will then iterate over that number of objects and create the objects. This is shown in Listing 3.1. However, we have not moved the objects so they will appear on top of each other at the default position, the origin. This is shown in Figure 3.1. Listing **??** adds a single line of code to move the objects to random locations along X. Again shown in the center of Figure 3.1 with fifty spheres, and on the right with 250 spheres. This demonstrates the ease at which we can build scenes with Python scripts to place objects randomly in a scene.

```python
import maya.cmds as cmds    #import the maya commands library
import random               #import the random library

# Create a variable to hold the number of objects
numObjects = 5

# Loop over number of objects
# creating and placing each object as it is created

for object in range(numObjects):
    cmds.polySphere()   # Create a polySphere
```

Listing 3.1: Using Maya functions to create mutliple objects in a scene (3.1_justLoop.py)

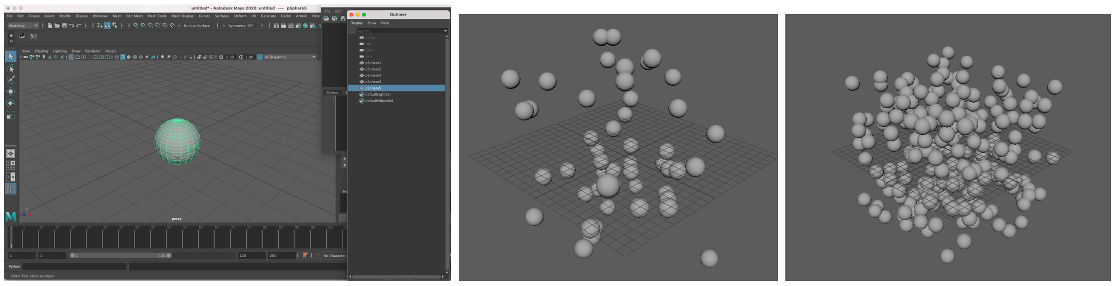


Figure 3.1: The result of using functions to create, size and randomly place objects in a scene and place them randomly in the x-direction

Five objects on a single dimension does not yield very complex results. Listing **??** shows how we can place objects randomly in each axis.

```python
import maya.cmds as cmds    #import the maya commands library
import random               #import the random library

cmds.file(new = True, force = True)

# Create a variable to hold the number of objects
numObjects = 5

# Loop over number of objects
```

```
10 # creating and placing each object as it is created
11
12 for object in range(numObjects):
13     cmds.polySphere()   # Create a polySphere
14     # now we will move the spheres as they are created
15     cmds.move(random.randint(-10, 10), 0, 0)
```

Listing 3.2: Using Maya functions to create multiple objects in a scene (3.2_loopRandomX.py)

This code can easily be adapted to create different objects for example changing line 11 in Listing 3.2 would yield an scenes as shown in Figure 3.2. We can also move in all directions as shown in Listing 3.3. Here we have created variables to hold the upper and lower values to provide to our randomint() function. The values for each dimension do not have to be the same. For example, if we wanted the objects placed closer together in Y we could reduce the bounds for Y. We also do not have to have a negative number as the lower bound. As shown in Figure 3.3, if we change line 25 in Listing 3.3 from cmds.polySphere() to cmds.polyCube() we will change the result. This also applies to all the objects available to create in Maya.

```
1  import maya.cmds as cmds    #import the maya commands library
2  import random               #import the random library
3
4  cmds.file(new = True, force = True)
5
6  # Create a variable to hold the number of objects
7  numObjects = 250
8  # Create variables for the random bounds so we can
9  # easily update them
10 # we can have more control if we have separate
11 # bounds for X Y and X
12 upperX = 10
13 lowerX = -10
14
15 upperY = 3
16 lowerY = -3
17
18 upperZ = 10
19 lowerZ = -10
20
21 # Loop over number of objects
22 # creating and placing each object as it is created
23
24 for object in range(numObjects):
25     cmds.polySphere()   # Create a polySphere
26     # now we will move the spheres as they are created
27     cmds.move(random.randint(lowerX, upperX), random.randint(lowerY, upperY), random.randint(lowerZ, upperZ))
28     cmds.move(random.randint(lowerX, upperX), random.randint(lowerY, upperY), random.randint(lowerZ, upperZ))
29     cmds.move(random.randint(lowerX, upperX), random.randint(lowerY, upperY), random.randint(lowerZ, upperZ))
30
31 # clear the active list
32 cmds.select( clear=True )
```

Listing 3.3: Using Maya functions to create multiple objects in a scene (3.3_loopRandomXYZ.py)

3.5 *Bundling Code into Functions*

Functions are a convenient way to package up a group of statements that accomplish a specific task. Once we package our code in a function it can be reused over and over again. We *call* a function by using its name,

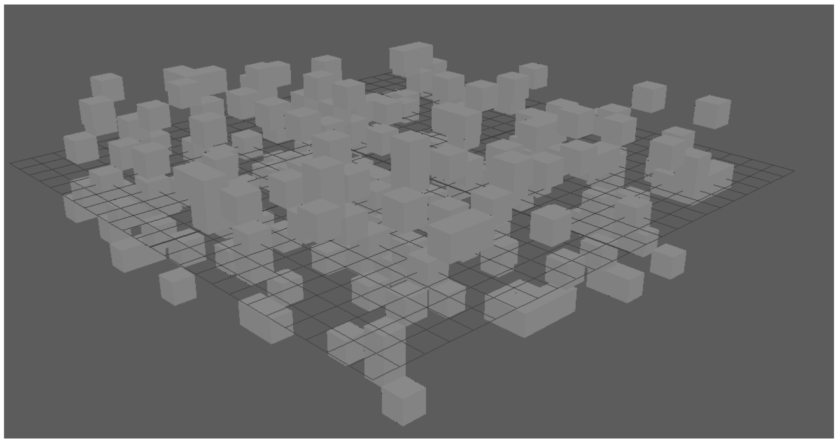


Figure 3.2: The result of changing the object created in Listing 3.3

and we provide any input the function might need, and store any results the function might return back to us. We have already encountered many functions but we just used them (ex. scale(), move(), select(), print()). These are **built in functions**, or functions that come with the programming language. As programmers we want to write our own functions to solve our own particular problems.

You use functions in programming to bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. *That means that a function is a piece of code written to carry out a specified task.*

To carry out that specific task, the function may or may not need multiple pieces of information provided to it, these are called **inputs**. (Functions can have optional inputs.). When the task is carried out, the function might have a value to return, so a function may or may not return values. (Functions can have optional outputs/return-values.)

Generally, a good rule of thumb is if you've written the same code twice, then you should think about putting it into a function. One of the objectives of good programming is to keep things DRY (Don't Repeat Yourself), that way if you need to make a change it will only need to happen in one place and it also makes debugging far quicker and simplified.
There are a few steps to define a function

1. Use the keyword **def** to declare a function and give it a name NOTE the name should reflect what the function does - believe me you want to name your functions well

2. If you have parameters they go inside () after the function name and then end that line with a :

3. Add your code indented

4. End your function with the keyword "return" and a value if you want the function to return anything (if not then it will simply continue to the next line of code)

```
def functionName(parameter1, parameter2...):
    code to execute
    return value # optional
```

```
# the function name should describe the action
# to INVOKE a function we have to CALL it

functionName(argument1, argument2). # this is calling the function
# You can pass any number of arguments, each argument is mapped to each parameter in the function
# If you have more than a few parameters you might # want to rethink if the function could be
# separated into multiple function, maybe you are # not trying to just do one thing.
```

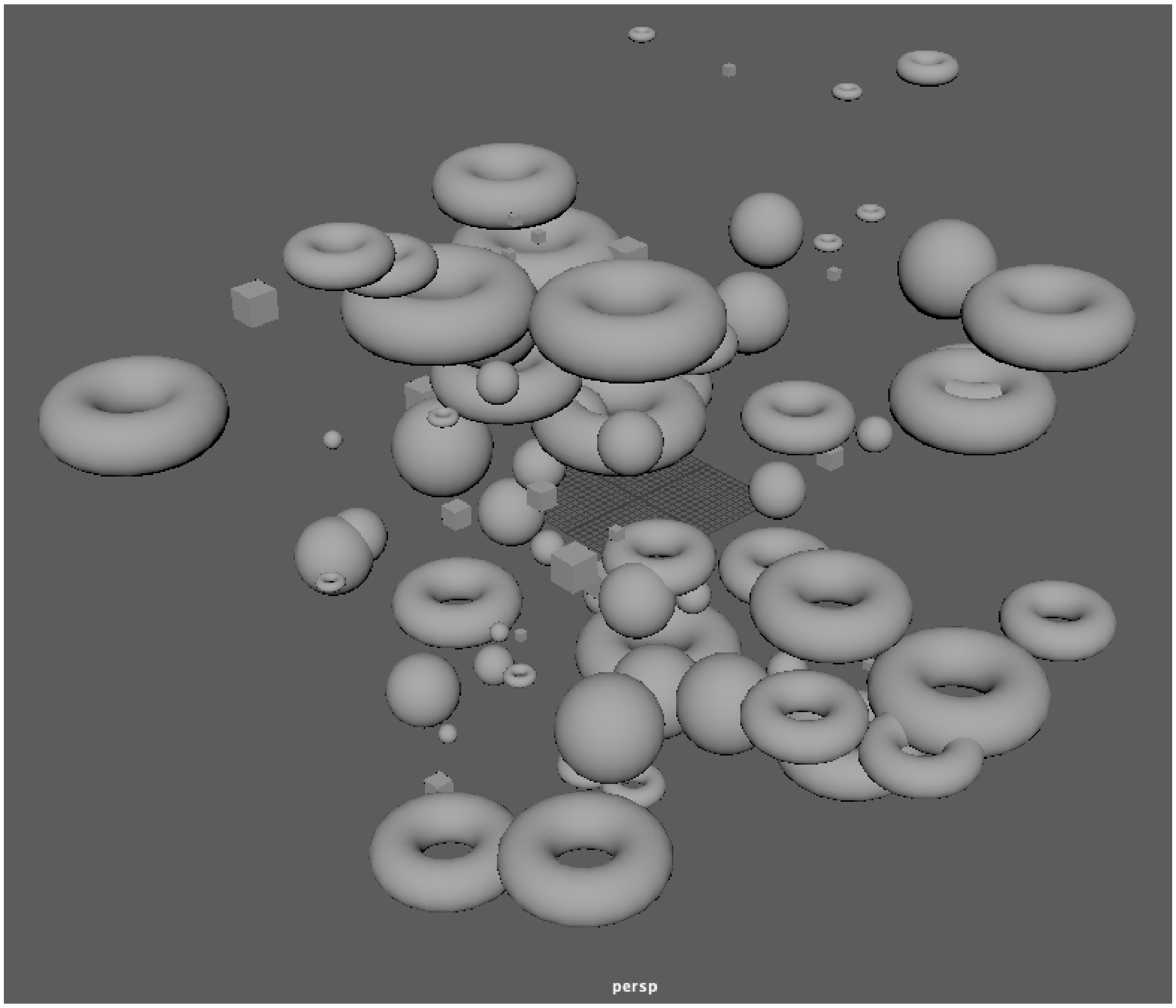


Figure 3.3: The result of using functions to create, size and randomly place objects in a scene

```
# MAYA
import maya.cmds as cmds
import random
#This Creates a new file
cmds.file( force=True, new=True )

# create a sphere
```

```
def createSphere(theRadius):
    print 'Creating A Sphere'
    sphere = cmds.polySphere(radius = theRadius)
    return sphere

# scale an object - this will work for any object
def scaleObject(theObject, scaleX, scaleY, scaleZ):
    cmds.select(clear = True)
    cmds.select(theObject)
    cmds.scale(scaleX, scaleY, scaleZ, theObject)

# uniformScale
#   2 parameters
#   theObject
#   scaleFactor - this will work on any object
def uniformScale(theObject, scaleFactor):
        cmds.scale(scaleFactor, scaleFactor, scaleFactor, theObject)

def randomPlace(theObject, gridSize):
    xTranslate = random.randint(0, gridSize)
    yTranslate = random.randint(0, gridSize)
    zTranslate = random.randint(0, gridSize)

    cmds.move(xTranslate, yTranslate, zTranslate, theObject)


mySphere   = createSphere(5)
scaleObject(mySphere[0], 1, 3, 4)

myCube = cmds.polyCube()
scaleObject(myCube[0], 1, 8, 1)

myTorus = cmds.polyTorus()

uniformScale(mySphere[0], 10)
uniformScale(myCube[0], 10)

#This Creates a new file
cmds.file( force=True, new=True )

numberOfSpheres = 30
for i in range(numberOfSpheres):
    sphereRadius = random.randint(1, 5)
    sphere = createSphere(sphereRadius)
    randomPlace(sphere[0], 55)

numberOfCubes   = 20
for i in range(numberOfCubes):
    cubeSize = random.randint(1, 3)
    cube     = cmds.polyCube()
    uniformScale(cube[0], cubeSize)
    randomPlace(cube[0], 60)

numberOfTorii   = 40
for i in range(numberOfTorii):
    torusSize = random.randint(1, 6)
    torus     = cmds.polyTorus()
    uniformScale(torus[0], torusSize)
    randomPlace(torus[0], 70)

```

```python
cmds.select(clear=True)
```

Listing 3.4: Using Maya functions to create size and randomly place objects in a scene (3.4_mayaFunctions.py)

In Listing 3.5 the code first uses a loop to call the functions, then packages that loop into its own function so it can be called with a single line of code. Here we introduce *default parameter values*. If no argument values are provided when the function is called then the function will use the default values. Figure 3.4 shows the result of calling this function with fifty spheres, and an a thirty by thirty by thirty grid. Of course we can repeatedly call the function, we could even put the function call inside a loop, and this would result in a denser grid as shown in Figure 3.5.

```python
# Remember a function should really just do one thing
# We have 2 functions one creating the sphere and shading it
import maya.cmds as cmds
import random

#This Creates a new file
cmds.file( force=True, new=True )

# Creates a sphere, prints out its name, and returns the sphere.
# we can assign default values to the parameters
# here if no values are passed in the the default values are used
def createSphere(theRadius = 3, xScale = 1, yScale = 1, zScale = 1):
    sphere = cmds.polySphere(radius = theRadius)
    print 'Creating: ' + sphere[0]
    return sphere

# Shades the object based off rgb values.
def shadeObject(theObject, red = 1.0, green = 1.0, blue = 1.0):
    shadingNode = cmds.shadingNode( 'blinn', asShader=True )
    cmds.setAttr( shadingNode+".color", red, green, blue, type='double3' )
    shadingGroup = cmds.sets(name=theObject+'SG', empty=True, renderable=True, noSurfaceShader = True)
    print shadingGroup
    cmds.connectAttr(shadingNode+'.outColor', shadingGroup+'.surfaceShader')
    cmds.select(theObject)
    cmds.sets(e=True, forceElement=shadingGroup)

# Scales the object, but uses default values
def scaleObject(theObject, xScale = 1, yScale = 1, zScale = 1):
    cmds.scale(xScale, yScale, zScale, theObject)

# Scales the object uniformly
def uniformScale(theObject, scaleFactor = 1):
    cmds.scale(scaleFactor, scaleFactor, scaleFactor, theObject)

# places the object randomly in a 3D grid
def randomPlace(theObject, gridSize):
    xTranslate = random.randint(0, gridSize)
    yTranslate = random.randint(0, gridSize)
    zTranslate = random.randint(0, gridSize)

    cmds.move(xTranslate, yTranslate, zTranslate, theObject)


sphere = createSphere(2)
shadeObject(sphere[0], 1, 0, 0)

```

```python
cube = cmds.polyCube()
cmds.move(-2, 0, 0, cube)
shadeObject(cube[0], 1, 0, 1)  # Change this to see different colors
scaleObject(cube[0],1, 8, 5)

# Lets make something colorful
numSpheres  = 30
gridSize    = 40
for i in range(numSpheres):
    xTranslate = random.randint(0, numSpheres)
    yTranslate = random.randint(0, numSpheres/10)
    zTranslate = random.randint(0, numSpheres)

    sphereRadius = random.randint(1, 3)
    sphere  = createSphere(sphereRadius)
    shadeObject(sphere[0], random.random(), random.random(), random.random())
    randomPlace(sphere, gridSize)

# Completely reset the Maya scene (similar to using "cmds.select(all=True) cmds.delete()")
cmds.file( force=True, new=True )

####################################################
# Now lets actually put that loop into a function so
# we can call it easily.

# Returns a random integer based off the number of objects
def myRandom(start = 0, numObjects = 100, divideBy = 1):
    return random.randint(start, numObjects/divideBy )

# Creates a specified number of spheres, randomly moves them, and adds a random color
def lotsOfSpheres(numObjects = 100, gridSize=100):
    for i in range(numObjects):
        sphereRadius = random.randint(1, 3)
        sphere  = createSphere(sphereRadius)
        shadeObject(sphere[0], random.random(), random.random(), random.random())
        randomPlace(sphere, gridSize)
        cmds.select(all=True, clear=True)

# lotsOfSpheres()  # Calling like this will use the default values
lotsOfSpheres(50, 30)
```

Listing 3.5: Using Maya functions to create shade and randomly place objects in a scene (3.5_mayaFunctions.py)

3.6 *Summary*

In this section, we illustrated how we can use Python scripting to quickly populate scenes. The examples used simple primitives but are trivial to generalize to more complex objects, or groups of objects. These examples begin to show the power of Python scripting when compared to manually adding a large number of diverse objects to a complex scene. It is effective and efficient to use python scripting to create, place, and even shade multiple objects in Maya scenes. The code presented in this chapter is intended as a foundation to encourage the reader to imagine and innovate new examples that will satisfy their own project requirements.

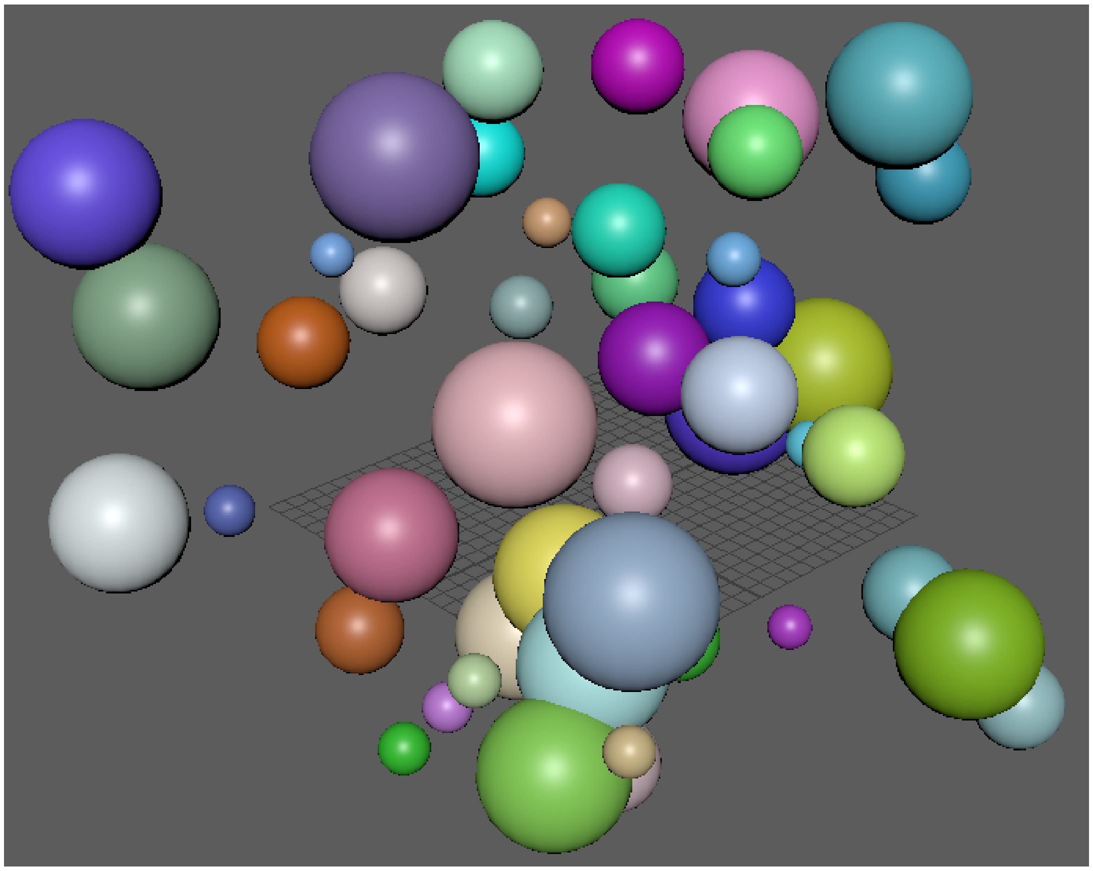


Figure 3.4: The result of using functions to draw multiple spheres in random colors on a 3D grid in Maya

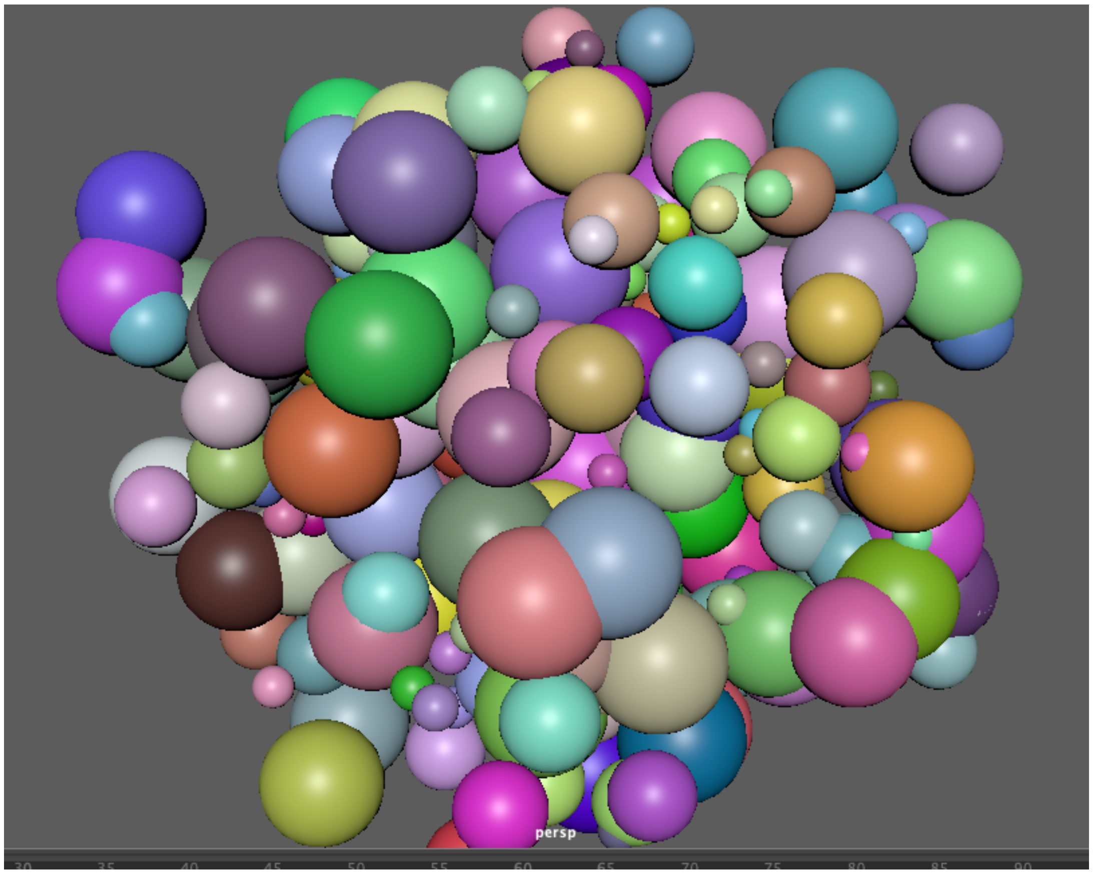


Figure 3.5: The result of repeatedly calling a function to draw multiple spheres in random colors on a 3D grid

4
Project 2: Creating a Custom GUI

4.1 Graphical User Interfaces in Maya

Creating Graphical User Interfaces (GUIs) is a straightforward process using Python and Maya 2022. We create a window, populate it with buttons, or other UI elements. Each button (UI element) in the GUI can then be connected to a Python Function. In this chapter we will cover

- Creating a simple window
- Populating a window with buttons
- linking buttons to commands (actions)
- Organizing a UI through classes
- Using tabs and scrolling

4.2 Creating a Window

To create a window we use the `window` command. Before we create the window we will want to check if another copy of this window exists (otherwise we will simply create multiple copies of the same window - this is typically not the desired result. This is shown in Listing 4.1. Prior to creating the window we first check if it exists. Then we create the window. There are some optional arguments. The first is the name of the window, followed by the title of the window, in this case *My First Window*, the icon title (which by default is the same as the title) and the `widthHeight` sets the size of the window (500, 150). We assign the new window to the variable name `myWindow` so we can reference it later. Before we can add any elements to the window we must specify a **layout**. In listing 4.1 we set up column layout to add UI elements vertically to the window. The code checks for the existence of the window and if it already exists, deletes it before we create a window. Otherwise, we will end up with multiple windows, which is rarely the intended outcome.

```python
import maya.cmds as cmds

# define a function to create the window
def createUI():
    # As we dont want multiple copies of the window
    # we first check if the window exists and if it does
    # we need to delete it
    if cmds.window('myWindow', exists=True):
        cmds.deleteUI('myWindow')

    # create a new window with the title My First Window,
    # with a width of 200 and a height of 150
    # Assign the window to a variable myWindow
    window = cmds.window("myWindow", title="My First Window", iconName='Show me on Icon',widthHeight=(500, 150) )


    # Before adding elements we must provide a layout
    # This is a single column layout so elements are
    # added vertically
    cmds.columnLayout()

    # now we can add some text in the form of a label
    # Whatever appears between the " " will appear in the window
    cmds.text(label="My Window, Hello...")

    # finally we must show the window
    cmds.showWindow(window)

# Call the function to create the UI
createUI()
```

Listing 4.1: Using Python functions to create a simple Window in Maya (4.1_Window.py)

4.3 *Creating Buttons*

Once we have our window established we can start to add UI elements. One of the most common UI elements is a button. We add a button using `cmds.button` as follows:

```
cmds.button(label = "Create a Cube", command = ('cmds.polyCube()'))
```

Listing 4.2: Creating a cube by calling polyCube() directly

4.4 *Linking Buttons to Actions*

In the previous section we added buttons, but they were just placeholder buttons. No action resulted from a button press. To associate a button press with an action we can provide a couple of different options. The first way is to simply connect a `command` to the button as shown here:

```
cmds.button(label = "Create a Cube", command = ('cmds.polyCube()'))
```

Listing 4.3: Using command to trigger an action from a button press

Another way to connect functionality is to call a function. There is a small idiosyncrasy here in that the function will accept arguments, `*args`, even though we don't provide, or use any arguments. This is because when Maya triggers a function it passes some data to the function. If we write the function without the `*args` it will result in an error saying the function takes no arguments (1 given). Even if we don't want to use these arguments we still need to write the UI functions to accept arguments.

```
# Simple function to create a cube
# We need to include *args as MAYA will
# pass information regardless of whether
# that information is used in the function
def createCube(*args):
    cmds.polyCube()

# body of code

# create the button and add the function name as the trigger
cmds.button(label = "Create a Cube", command = createCube)
```

Listing 4.4: Using a function call to trigger an action from a button press

4.5 *Capturing Input from GUIs*

Often we want to allow the user to provide data as input to functions. Let's say we want to allow the user to input the number of objects and the vertical distance between them. We can create two fields to capture this data. The number of objects would be a whole number, *integer*, while the distance between them can be a *floating point* number. We can create these fields in our UI using `intField` and `floatField` respectively. Each creates a field that can only accept data of the type specified. The `intField` can only accept integers, and the `floatField` only captures floats. The values in each case are bound by a minimum and maximum value. There is an **invisible slider** built in to the field. To access the slider hold down CTRL and press the mouse button. To increase the value, slide the mouse to the right, and to decrease the value slide it to the left. The step is specified with the `-s/step` parameter.

Listing 4.5 illustrates the simplicity of a basic GUI.

```
# Import the Maya commands library
import maya.cmds as cmds

# Use the cmds.window command to create a window.
# here we are providing a title, an icon name and
# the dimensions of the window
myWindow = cmds.window( title="Long Name", iconName='Short Name', widthHeight=(150, 100) )

# columnLayot allows us to align window contents
cmds.columnLayout( adjustableColumn=True )

# Create a button
# Display "Click me"
cmds.button( label='Click Me' )

# Create an EXIT button with a command to delete the UI
cmds.button( label='EXIT', command=('cmds.deleteUI(\"' + myWindow + '\", window=True)') )

# Set its parent to the Maya window (denoted by '..')
cmds.setParent( '..' )

# Display the window myWindow)
cmds.showWindow(myWindow)
```

Listing 4.5: Using Python functions to create a simple GUI (4.6_simpleGUI.py)

Adding buttons to a UI is just a matter of using the `cmds.button()` function.

```
cmds.button(label = "Create a Cube", command = ('cmds.polyCube()'))
```

Listing 4.6: Capturing input from the user

Listing 4.8 populates the UI with buttons to create shapes.

```python
import maya.cmds as cmds

def createSphere(*args):
    cmds.polySphere()

def createCube(*args):
    cmds.polyCube()

def deleteCubes(*args):
    allCubes = cmds.ls("pCube*")
    if(len(allCubes)>0):
        cmds.delete(allCubes)

# define a function to create the window
def createUIwithButtons():
    # As we dont want multiple copies of the window
    # we first check if the window exists and if it does
    # we need to delete it
    if cmds.window('myWindow', exists=True):
        cmds.deleteUI('myWindow')

    # create a new window with the title My First Window,
    # with a width of 200 and a height of 150
    # Assign the window to a variable myWindow
    window = cmds.window("myWindow", title="My First Window", iconName='Show me on Icon',widthHeight=(200, 150) )


    # Before adding elements we must provide a layout
    # This is a single column layout so elements are
    # added vertically
    cmds.columnLayout()

    # now we can add some text in the form of a label
    # Whatever appears between the " " will appear in the window
    cmds.text(label="My Window, Hello...")

    #now we can add buttons
    #cmds.button(label = "Create a Cube", command = ('cmds.polyCube()'))

    # it might be better to wrap the command in a function
    cmds.button(label = "Create a Sphere", command = createSphere)
    cmds.button(label = "Create a Cube", command = createCube)
    # we can also add an option to delete the cubes
    cmds.button(label = "Delete Cubes", command = deleteCubes)

    # deleting the spheres is left as an exercise.

    # finally we must show the window
    cmds.showWindow(window)

# Call the function to create the UI
createUIwithButtons()
```

Listing 4.7: Using UI to create shapes at the touch of a button (4.7_addButtonsFunctions.py)

Listing 4.8 creates a stack of spheres or cubes. The number and size of the objects is dictacted by the values entered in the fields.

```python
import maya.cmds as cmds

global shapeCountField
global shapeSizeField
```

```python

def showUI():
    global shapeCountField
    global shapeSizeField
    myWin = cmds.window(title="Make Shapes", widthHeight=(300,200))
    cmds.columnLayout()
    shapeCountField  = cmds.intField(minValue=1)
    shapeSizeField   = cmds.floatField(minValue=0.5)
    cmds.button(label="Stack Spheres", command=makeSpheres)
    cmds.button(label="Stack Cubes", command=makeCubes)

    cmds.showWindow(myWin)

def makeSpheres(*args):
    global shapeCountField
    global shapeSizeField
    count = cmds.intField(shapeCountField, query=True, value=True)
    rad   = cmds.floatField(shapeSizeField, query=True, value=True)

    for i in range(count):
        cmds.polySphere(radius=rad)
        cmds.move(0,(i * rad * 1.5), 0)

def makeCubes(*args):
    global shapeCountField
    global shapeSizeField
    count = cmds.intField(shapeCountField, query=True, value=True)
    size   = cmds.floatField(shapeSizeField, query=True, value=True)

    for i in range(count):
        cmds.polyCube(depth=size, height=size, width=size)
        cmds.move(0,(i * size * 1.5), 0)


showUI()
```

Listing 4.8: Using UI to create stacks of spheres or cubes (4.8_getinput.py)

Instead of a `field` we can use a slider using `intSlider` as shown in Listing 4.10

```python
import maya.cmds as cmds

global numberOfCubesInput
global sizeSlider

def CreateUIwithInput():
    # use the key word global here or
    # we would create a local variable with
    # the same name
    global numberOfCubesInput
    global sizeSlider
    # As we dont want multiple copies of the window
    # we first check if the window exists and if it does
    # we need to delete it
    if cmds.window('myWindow', exists=True):
        cmds.deleteUI('myWindow')

    # create a new window with the title My First Window,
    # with a width of 200 and a height of 150
    # Assign the window to a variable myWindow
    window = cmds.window("myWindow", title="Create A Stack of Cubes Window", iconName='Show me on Icon',widthHeight
     =(200, 150) )

    # Before adding elements we must provide a layout
    # This is a single column layout so elements are
    # added vertically
    cmds.columnLayout()

    # now we can add some text in the form of a label
    # Whatever appears between the " " will appear in the window
    cmds.text(label="Create Cubes")

    #now we can add some input and some buttons
    cmds.text(label = "Number of Cubes")
    numberOfCubesInput = cmds.intField("noCubes", minValue = 0, maxValue = 10, value= 1, step = 1)
    cmds.text(label = "Cube Size")
    sizeSlider = cmds.intSlider(min = 1, max = 100, value = 1, step = 1)

    cmds.button(label = "Create a Stack of Cubes", command = createCubes)

    # finally we must show the window
    cmds.showWindow(window)

def createCubes(*args):
    # use the key word global here or
    # we would create a local variable with
    # the same name
    global numberOfCubesInput
    global sizeSlider

    # retrieve the values in the UI fields
    numberOfCubes = cmds.intField(numberOfCubesInput, query = True, value = True)
    cubeSize      = cmds.intSlider(sizeSlider, query = True, value = True)


    for cube in range(numberOfCubes):
        cmds.polyCube(width = cubeSize, height = cubeSize, depth = cubeSize)
        cmds.move(0, cube * cubeSize, 0)


    cmds.select(clear = True)

# Call the function to create the UI

cmds.file( f=True, new=True )
CreateUIwithInput()
```

Listing 4.9: Using UI to create stacks of spheres or cubes (4.9_slider.py)

4.6 *Using Classes*

Rather than use global variables, we can use classes to maintain state. This is illustrated in Listing **??** The `TorusClass` encapsulates the attributes of the torus so we can initialize and create the shape using classes. The output is shown in Figure 4.1

```python
import maya.cmds as cmds

class TorusClass:
    def __init__(self):
    # Define an id string for the window first
        winID = 'windowID'

    # Test to make sure that the UI isn't already active
        if cmds.window(winID, exists=True):
            cmds.deleteUI(winID)
        self.win = cmds.window(winID,title="Stack of Donuts", widthHeight=(300,200))
        cmds.columnLayout()
        self.numDonuts= cmds.intField(minValue=1)
        cmds.button(label="Make Donuts", command=self.makeDonuts)
        cmds.showWindow(self.win)

    def makeDonuts(self, *args):
        number = cmds.intField(self.numDonuts, query=True, value=True)
        for donut in range(0,number):
            cmds.polyTorus(name = 'DoNut#')
            cmds.move(0, donut*2, 0)

stack = TorusClass()
```

Listing 4.10: Using UI to create stacks of spheres or cubes (4.10_Classes.py)

.

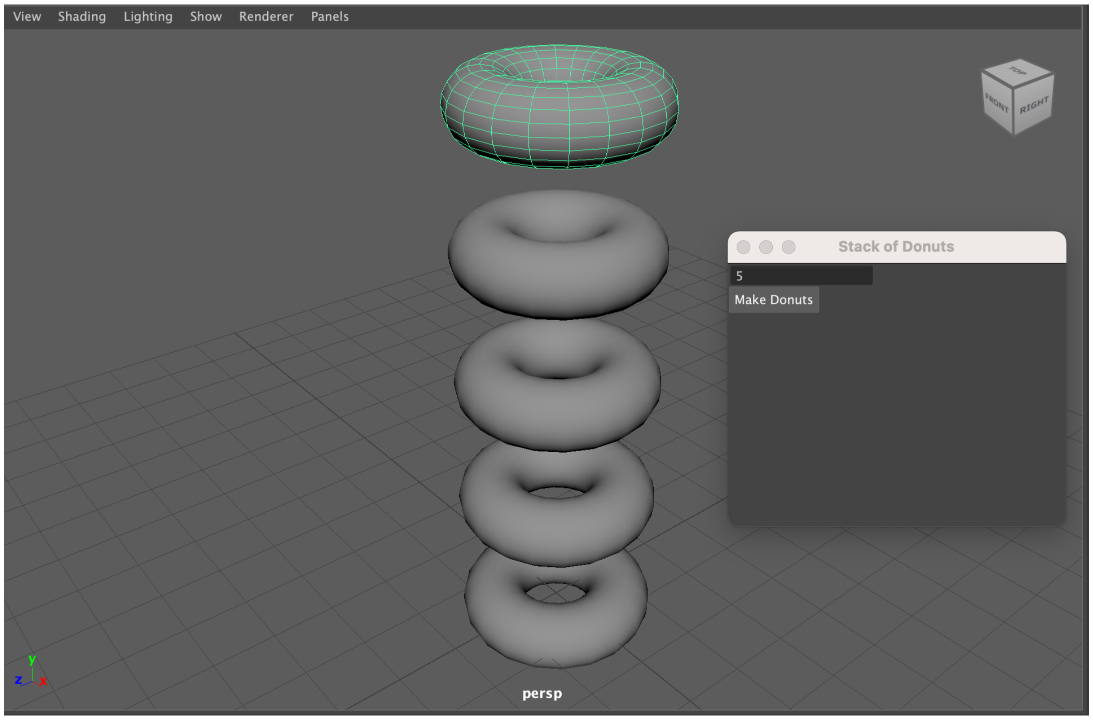


Figure 4.1: A stack of Torii generated using a TorusClass class to maintain state attributes

4.7 *Summary*

MAYA GUIs are straightforward to set up in Python. This can make automating repetitive tasks as simple as pressing a button. In this course we just looked at some of the elements available to populate UIs.

5
Project 3: Programming MASH Networks

5.1 Introduction to MASH Networks

MASH can be used to create “versatile motion design animations with procedural node networks". Effects can be quickly assembled in a custom manner, and can be chained to realize unique variations.

5.2 Creating Dynamics with MASH

Listing 5.1 is a script that creates a MASH network to generate some rectangular shapes of random length and adds effects to them so when the user presses play dynamic simulation is invoked. Figures **??** and **??** illustrate the progression of the dynamic animation. The real benefit of such a network is experienced through viewing the actual animation.

```python
import MASH.api as mapi
import maya.cmds as cmds

#new file
cmds.file(force=True, new=True)

#sphereToDistributeOn = cmds.polySphere(r=12)
originalCube = cmds.polyCube()

# create a new MASH network
mashNetwork = mapi.Network()
mashNetwork.createNetwork(name="Cubes")

# When you create a MASH network it creates
# some default nodes
# print out the default node names
print (mashNetwork.waiter)
print (mashNetwork.distribute)
print (mashNetwork.instancer)

cmds.setAttr(mashNetwork.distribute+'.arrangement', 3)
cmds.setAttr(mashNetwork.distribute+'.pointCount', 75)

# We can change the distribution
# each is represeneted by a number
# 1 = linear, 2 = radial, 3 = spherical, 4 = mesh, 5 = inPositionPP, 6 = Grid,
# 7 = Initial State 8 = Paint Effects 9 = volume
print (mashNetwork.distribute+'arrangement')
cmds.setAttr(mashNetwork.distribute+'.arrangement', 6)
cmds.setAttr(mashNetwork.distribute+'.gridx', 5)
cmds.setAttr(mashNetwork.distribute+'.gridy', 5)
cmds.setAttr(mashNetwork.distribute+'.gridz', 5)

node = mashNetwork.addNode("MASH_Dynamics")

cmds.setAttr(mashNetwork.waiter+'_BulletSolverShape.groundPlanePositionY', -10)

cmds.setAttr(node.name+'.bounce', 1)

randomNode = mashNetwork.addNode("MASH_Random")
print (randomNode)
cmds.setAttr(mashNetwork.waiter+'_Random.scaleX', 1)
cmds.setAttr(mashNetwork.waiter+'_Random.scaleY', 10)
cmds.setAttr(mashNetwork.waiter+'_Random.scaleZ', .5)

# There are bunch of nodes we can add.
# another is  SIGNAL NODE
# it basically adds random noise - lets add one
# add a Signal node
node = mashNetwork.addNode("MASH_Signal")
# set the signal node to have some scale noise
cmds.setAttr(node.name+".scaleX", 10)
# There are different types of noise
print (node.name)
```

Listing 5.1: Using Using MASH to create some dynamics (5.1_MASH.py)

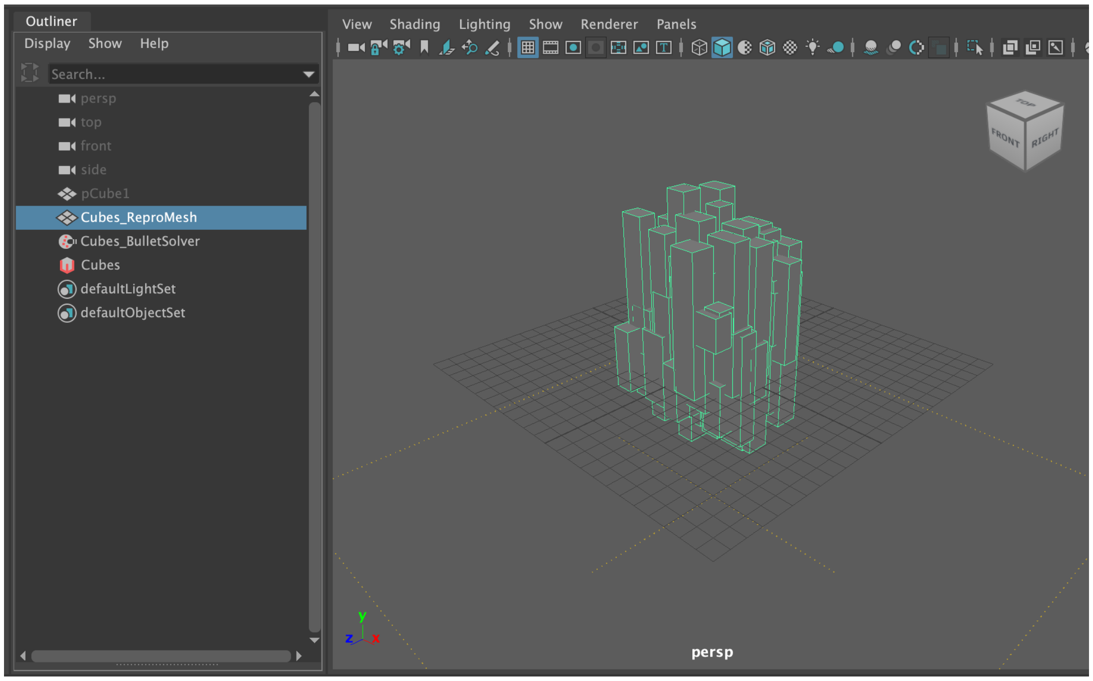


Figure 5.1: A random collection of rectangular shapes generated using a MASH network

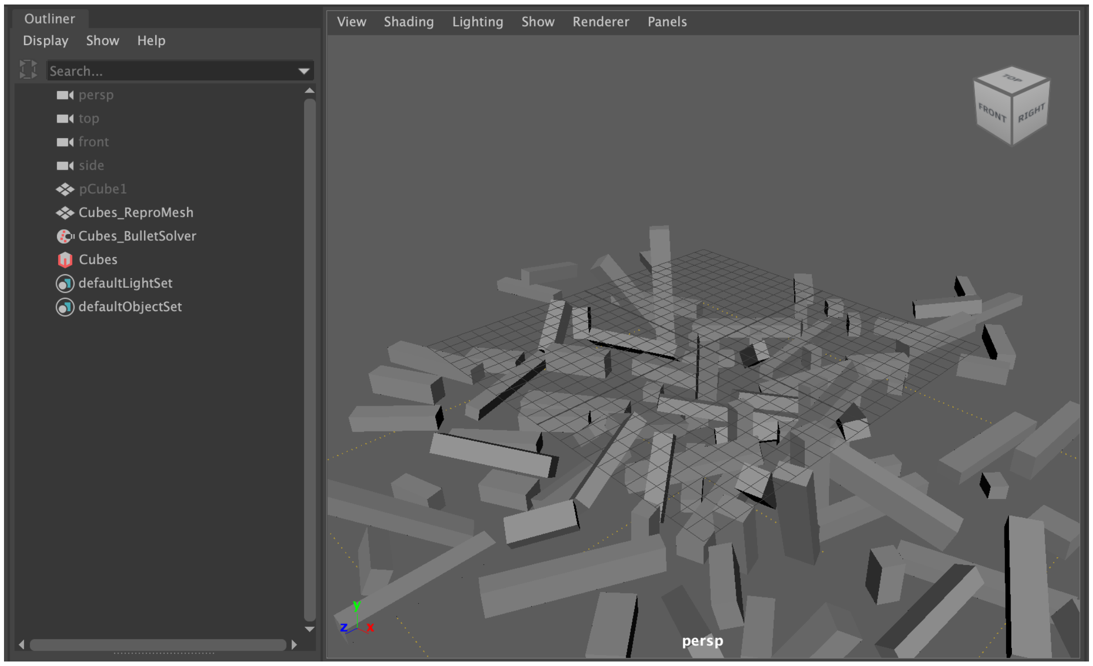


Figure 5.2: The dynamic simulation applied to random collection of rectangular shapes generated using a MASH network

5.3 *MASH with FallOff Element*

Figure 5.3 shows the result of introducing a falloff element in our MASH network. The falloff element influences the attributes of the network elements it encloses. The code to generate this is listed in Listing 5.2. The real benefit of such a network is experienced through viewing the animation.

```

###########
import MASH.api as mapi
import maya.cmds as cmds
#new file
cmds.file(force=True, new=True)
sphereToDistributeOn = cmds.polySphere(r=15)
cmds.polyCube()

# create a new MASH network
mashNetwork = mapi.Network()
mashNetwork.createNetwork(name="Cubes")
# print out the default node names
print (mashNetwork.waiter)
print (mashNetwork.distribute)
print (mashNetwork.instancer)
# add a Signal node
node = mashNetwork.addNode("MASH_Signal")
# set the signal node to have some scale noise
cmds.setAttr(node.name+".scaleX", 10)
# print out the name of the signal node
print (node.name)
# add a Falloff to the Signal node
falloff = node.addFalloff()
# move the falloff
falloffParent = cmds.listRelatives(falloff, p=True)[0]
cmds.setAttr(falloffParent+".translateY", 8)
```

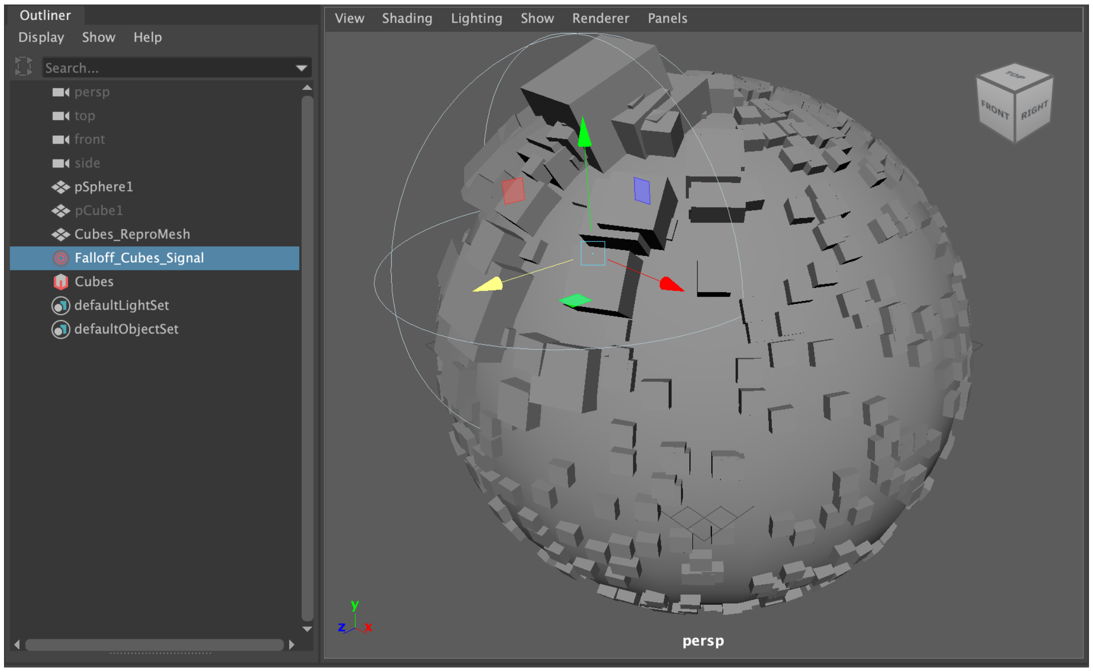


Figure 5.3: We can move (animate) the offset through the MASH network to quickly generate interesting effects

```python
# make it so the network distributes onto the surface of a mesh
mashNetwork.meshDistribute(sphereToDistributeOn[0])
# set the point count of the network
mashNetwork.setPointCount(1000)
# print all the nodes in the network
nodes = mashNetwork.getAllNodesInNetwork()
print ("All nodes in network: ")
print (nodes)
# find all the falloffs in the network
for node in nodes:
    mashNode = mapi.Node(node)
    falloffs = mashNode.getFalloffs()
    if falloffs:
        print (node+" has the following falloffs: " + str(falloffs))
```

Listing 5.2: Using Using MASH to create some dynamics (5.2_MASH_FallOff.py)

5.4 *Summary*

MASH networks are a powerful tool to quickly generate repeating elements, dynamic processes and stochastic simulations. You can even embed Python scripts to inject more autonomy into your MASH networks.

6

Project 4: Creating basic Joint Chains

6.1 Introduction

In this chapter we will introduce two simple scripts one to create a simple joint chain and a second to generate a generic hand skeleton.

6.2 Creating a simple joint chain

This simple chains is shown in Figure **??**. We provide the number of joints we want and the script will generate them. As can be seen in Listing 6.1 Its important to **orient the joints**. Here we orient the bones as travelling *down* the x-axis, and the y-axis is up. You can use any orientation you desire.

```python
import maya.cmds as cmds

# create a skeleton based on the joints paramter
def createSkeleton(joints):

    # clear the selection
    cmds.select(clear=True)

# Create an empty list to hold the bones
    bones = []
    # set the position to the origin
    pos = [0, 0, 0]


    # for each joint we need to create a joint
    # and append it to our bones list
    for i in range(0, joints):
        pos[1]= i * 5
        bones.append(cmds.joint(p = pos))

    # set the selection back to the first bone
    cmds.select(bones[0], replace = True)

# call our functions with different numbers of joints
createSkeleton(5)
createSkeleton(3)
```

Listing 6.1: Using Python functions to create a simple GUI (6.1_singleChain.py)

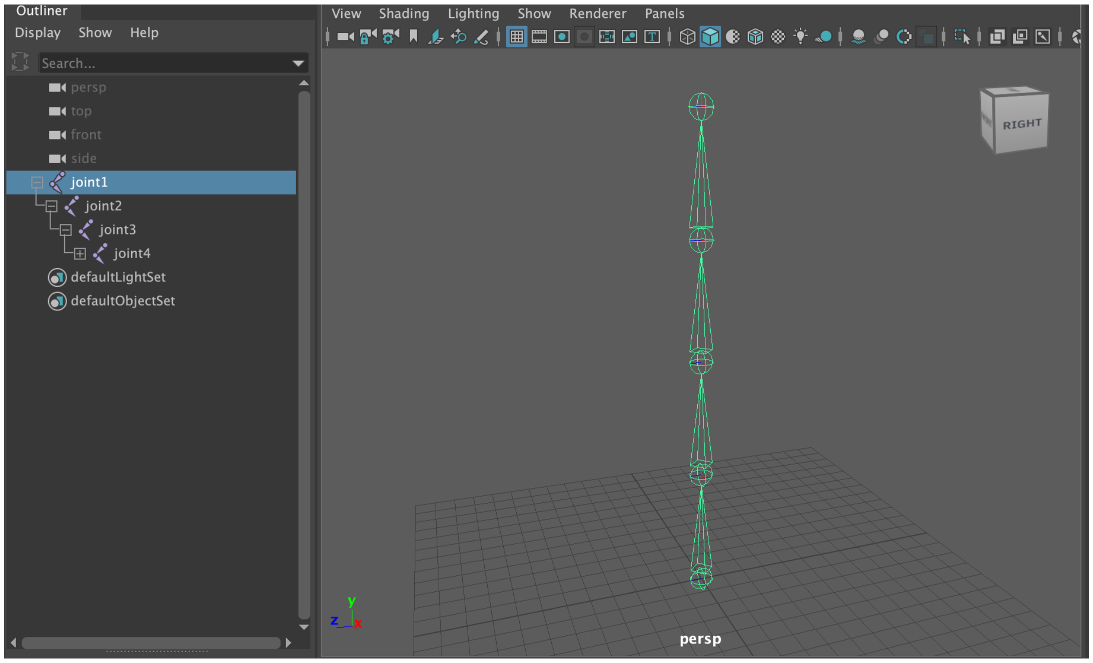


Figure 6.1: A simple joint chain with five joints

6.3 *Creating a jointed hand*

Listing 6.2 shows the python code to create a simple jointed hand.

```python
import maya.cmds as cmds
# Now create the hand
def createHand(fingers, joints):
    # Clear the selection
    cmds.select(clear=True)

    # create the base joint, name it wrist
    # set the postion to the origin
    baseJoint = cmds.joint(name = 'wrist', p = (0, 0, 0))

    # set reasonable values for the space between fingers,
    # palm length and joint length
    fingerSpacing = 2
    palmLength = 3
    jointLength = 1.5

    # now set up each finger
    for i in range(0, fingers):
        cmds.select(baseJoint, replace = True)
        pos = [0, palmLength, 0]
        pos[0] = (i * fingerSpacing) - (( fingers -1 ) * fingerSpacing)/2

        # create a base joint
        cmds.joint(name='finger{0}base'.format(i+1), p=pos)

        # create a joint for the number of
        # joints requested
        for j in range(0, joints):
            cmds.joint(name='finger{0}joint{1}'.format((i+1), (j+1)), relative=True, p=(0, jointLength, 0))

        cmds.select(baseJoint, replace=True)

# Call our function
createHand(5, 3)
```

Listing 6.2: Using Python functions to create a simple GUI (6.2_createHand.py)

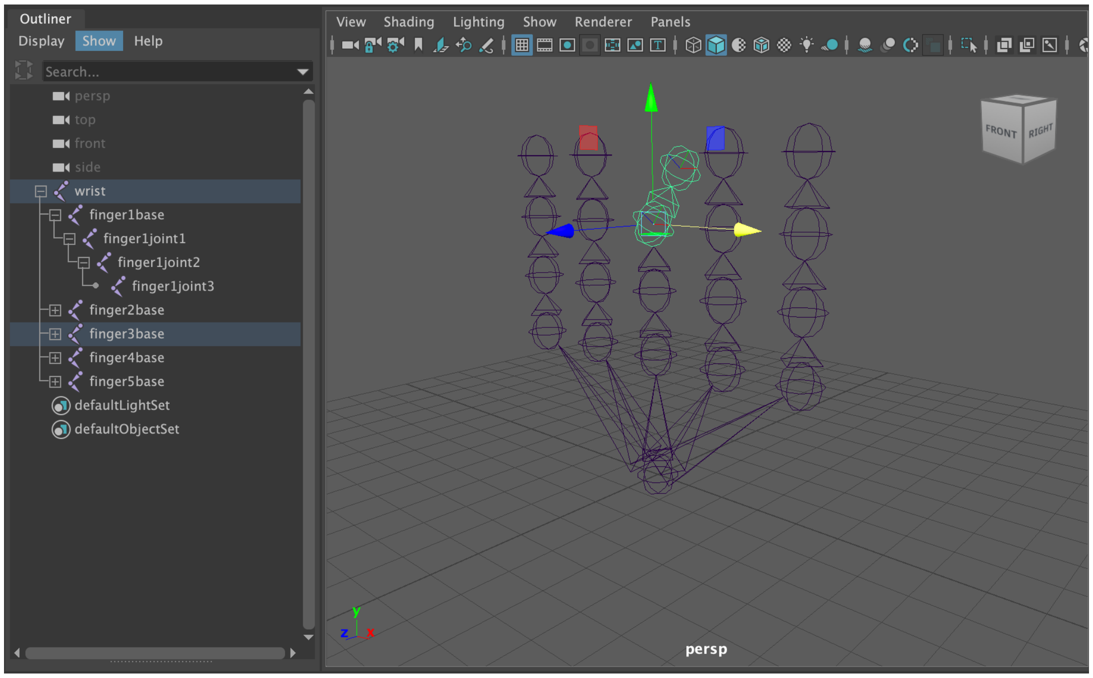


Figure 6.2: A simple jointed hand

6.4 *Summary*

This chapter gave a quick overview of how to create joint chains using python in Maya. We can also use scripts for more advance rigging processes, including setting up IK chains and generating entire bipedal rigs. The examples provided here are just a very basic representation of what is possible with python scripting for generating rigs.

7
Additional Resources Resources

7.1 Resource Links

1. Python in Maya, Autodesk
2. Using Python, Autodesk
3. Maya Python Commands Documentation, Autodesk
4. Maya Programming with Python Cookbook, Adrian Herbez