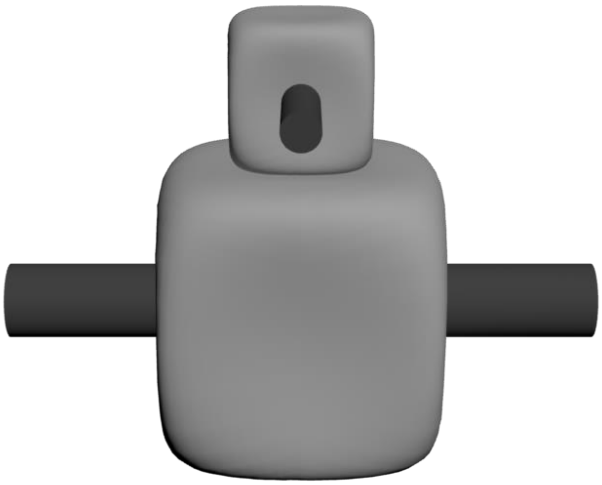
Guide to Quaddle 2.0

Dr. Thilo Womelsdorf

Dr. Marcus Watson

Zachary Simopoulos

Vanderbilt University
Psychology Department



Attention-Circuits-Control Laboratory (www.attention-circuits-control.org)

--Objects are all generated using 3ds Max 2021, and its inbuilt Maxscript--

Website guide: http://accl.psy.vanderbilt.edu/resources/analysis-tools/3d-image-material/

Brief instructions on how to use scripts: http://accl.psy.vanderbilt.edu/quaddlegenerator-brief-instructions/

Scripts, objects and in depth manual available on github: https://github.com/att-circ-contrl/Quaddles-Gen

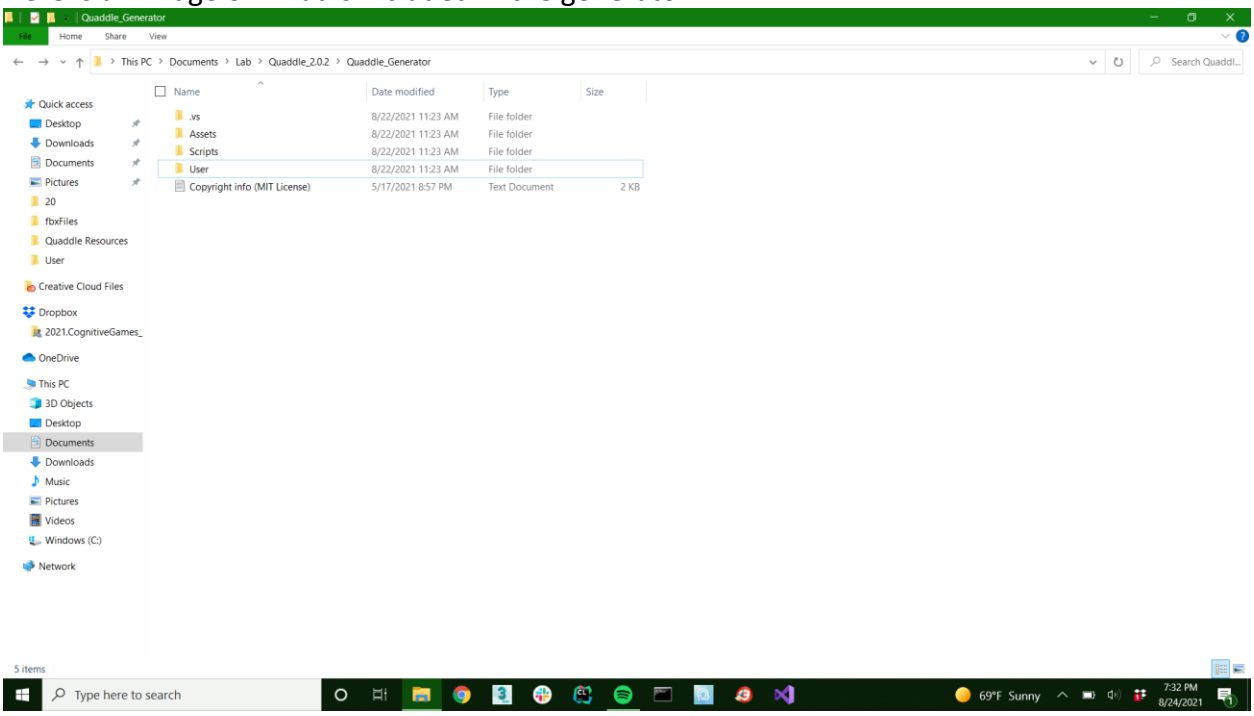
Getting Started

Downloading the Scripts

1. Go to this url.
2. Click on the green button that says “Code” and in the dropdown click “Download Zip”.
3. It will download the main folder as “Quaddle 2.0.2 – main”. Rename it to “Quaddle 2.0.2”

Familiarize Yourself with The Generator

Here is an image of what is included in the generator



User (folder)

This folder contains what a user will use to generate quaddles. It is also the default location where the quaddle “**images**” folder is put.

Preferences.txt

This is where you will enter your preferred paths and settings for generating your quaddles. The settings are discussed in depth later in this guide.

Object Table.txt

This text file specifies the Dimensions and Values you want to use in your next generation. Like Preferences, it will be discussed later.

Assets (folder)

Assets contain all the necessary data to generate quaddles. This includes icons, fractals, colours, and patterns.

Asset Data.txt

This text file holds all information about the Dimensions and Values of the Quaddles. The program uses it to build the data that the user wants.

Scripts (folder)

This folder contains all of the scripts, or code, used to generate the quaddles.

Set Up

1. **Locate where your Quaddle Repo is**
   Most of the time, when a file is downloaded, it can be found in the downloads folder. The default path to it in Windows OS is "*This PC\Downloads*"
   The folder is the one you downloaded be called *"***Quaddle 2.0 – main***"*, unless you renamed it

2. **Move it to wherever you wish to put it**

3. **Copy the Path of that Location**
   Open the generator folder. (You should see folders that say Scripts, Assets, etc.)
   Go to the top bar of File Explorer and click.
   You will see that the text becomes highlighted.
   From there, copy it by pressing ctrl + C.

4. **Open the Scripts Folder in the generator folder**
5. **Open "Main_Script.ms"**
6. **Scroll to where it says "generatorPath"**
7. **Paste yourPath inside the quotations so that all of the text is purple**
8. **Add one more slash to each slash so "\" becomes "\\"**
9. **Save and close the file**

Generating

Now that we have the program set up, we can move on to how to generate the set of quaddles you want. For this, you will interact with two files: “Object Table.txt” and “Preferences.txt”

Preferences

As mentioned earlier, this is where you will enter your preferred paths and settings for generating your quaddles. We will explain the different settings in order.

Here is a picture of what it looks like:

1 – *Specify Your Own Paths*

This setting allows you to set the location of your Scripts, Assets, and Exports if you wish.
If toggled “On” – The program uses the paths specified in Preferences.txt
If toggled “Off” – The program generates its own paths. If toggled Off, make sure that file structure is the same as when you downloaded it

NOTE 1: It may not work if you move files to different locations in the generator.
NOTE 2: Notice that the paths of Object Table.txt and Preferences.txt cannot be set. These must stay in the **User folder** in order to work.
NOTE 3: If it is toggled “Off” then the FBX files and PNGs will be exported into a folder called **Images** located in the **User folder** of the generator

2 – *Export PNGs*

This setting allows you to decide if you want to export PNGs of the Quaddles.
If toggled “On” – The program will export PNGs.
If toggled “Off” – The program will not export PNGs.

3 – *Picture Angle*

This setting allows you to decide what angle on the vertical axis you wish to take the picture of the quaddle in degrees.
If 0 is entered – The program will take a picture directly level with the quaddle.
If 20 is entered – The program will take a picture 20 degrees above the center of the quaddle but focused on the center.

4 – *Export FBXs*

This setting allows you to decide if you want to export FBXs of the Quaddles. These are 3d Models that can be implemented into other programs like Unity.
If toggled “On” – The program will export FBXs.
If toggled “Off” – The program will not export FBXs.

5 – *Grey Colour*

This setting allows you to decide if you want a light grey or dark grey quaddles. It will affect the colours on icons and fractals.
If toggled “Light” – The program will export light grey quaddles.
If toggled “Dark” – The program will export dark grey quaddles.

5 – *Location of Single Arm*

This setting allows you to decide which side of the quaddle you want to generate a single arm on (The Arm_Count Dimension is specified to generate 1 arm).
If toggled “Left” – A single quaddle will be generated with one arm on the left side of the quaddle body.

If toggled “Right” – A single quaddle will be generated with one arm on the right side of the quaddle body.
If toggled “Both” – Two quaddles will be generated. One will have a left arm and the other, a right.

6 – *Keep Icons and Fractals Visible*

This setting allows you to decide that if a quaddle has more than three arms and has an Icon or Fractal, the program will only generate the max number of arms to keep the Icon or Fractal visible.
If toggled “On” – Any quaddle that has more than three arms and an Icon or Fractal will have its Arm Count reduced to 3
If toggled “Off” – Any quaddle that has more than three arms and an Icon or Fractal will have its Arm Count will maintain its designated number of Arms.

7 – *Apply Aspect Ratio to Head*

This setting allows you to decide if you want quaddles that have heads with the same aspect ratio as it’s body.
If toggled “On” – The head of the quaddle will mimic its body’s aspect ratio
If toggled “Off” – the head of the quaddle will be a normal body generation, it will not be stretched or squished.

8 – *Apply Smoothness to Head*

This setting allows you to decide if you want to have the smoothness value of the quaddle applied to the head.
If toggled “On” – The head will adopt the smoothness value.
If toggled “Off” – The head will not adopt the smoothness value, rather it will default to be smooth.

9 – *Apply Smoothness to Body*

This setting allows you to decide if you want to have the smoothness value of the quaddle applied to the body.
If toggled “On” – The body will adopt the smoothness value.
If toggled “Off” – The body will not adopt the smoothness value, rather it will default to be smooth.

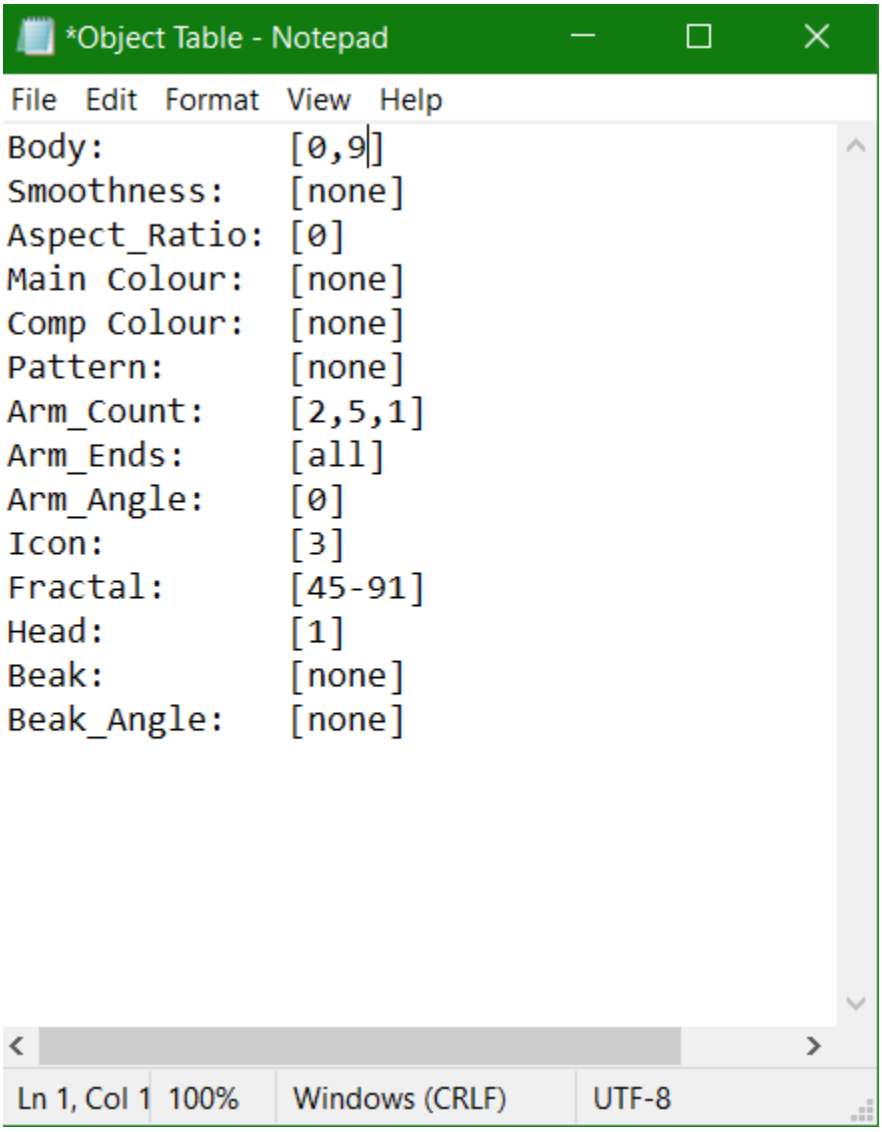
10 – Print Objects that are Not Possible

This setting allows you to see if any objects were not generated. This will happen when the combination of Main Colour, Comp Colour, and Pattern form a texture that does not exist in the asset folder.

If toggled “On” – The quaddles that could not be created will be printed in the maxscript listener
If toggled “Off” – The quaddles that could not be created will not be printed in the maxscript listener

Object Table – Where the next Quaddle's values are defined

Here is what it looks like:



Each line represents a dimension, and each value in the box corresponds with the order found in the Quaddle 2.0 Dictionary and "Asset Data.txt". Asset Data.txt can be found in the Asset folder.

The program supports single values from 0 to 99, lists of items using commas (e.g. 2,7,3), ranges (e.g. 3-7), and the keywords "none" and "all". "None" and "all" should be specified alone, not in a list.

There are three values that hold special meaning: zero (0), none, and all.

Zero
Zero means the null value of any given dimension. The null value means generally, the value which most closely resembles the non-existence of that dimension. For example, if the dimension "Head" was fed the value zero, the generator would generate a quaddle with "No Head." However, not all dimensions have this capability, so the program assigns a default value to those dimensions. This happens with the dimensions that are modifications of quaddle body dimensions. For example, the dimension "Arm Angle" cannot have a 'non-existent' null feature, so the angle chosen is zero, or straight. It is the value most closely resembles a null value. But Arm Count, since it responsible for making the arms themselves, has an option for that dimension's non-existence: zero. You can reference the dictionary to see them all, but this is what zero means to the generator.

None
None means the dimension is completely omitted. If specified, the program will not add that dimension to its data to load in. Obviously, it is similar to zero. The difference is when zero is loaded in as a value of a dimension, the program will execute the code connected to that dimension, but if 'none' is put, it will completely skip over the associated code. So, if you wish to completely omit a dimension, use "None".

All
All means include every value in that dimension. Pretty simple.

Generating the Quaddles

Once you have entered all your desired dimensions and preferences, and **saved the text files**, you are ready to run the program.

1. Open 3DS Max
2. Open the Maxscript Listener
    a. The Listener is where you will see the name of the objects you generate and the ones you don't, if you chose that in your Preferences file. It will also print out the Dimensions and Values it received from the object table.
3. Go to File -> Open
4. Navigate to the Scripts Folder of the generator and open **Main_Script.ms**

NOTE: We will now run the script, but you must know this: whenever the script is run for the first time after opening 3ds Max, it will **always** return an error. So, when you first open it, you will have to run it twice.

5. Run the Script. You can do this by pressing ctr + e or navigating to "Tools" → "Evaluate All"
    a. You will see a Copyright Notice pop up with a button that says "Go" after running the script. Once you press "Go", the quaddles will generate.

6. Remember, whenever the object table or preferences are changed, you must save them before generating.

Understanding the Scripts

Structure
The main scripts that demand attention are

Main_Script.ms
The Parent Script that controls the flow of the program. This includes when assets are loaded in, when to generate quaddles, if to cancel the program when there is faulty entry, etc. It relies on **Helper_Functions.ms** to load in preferences and **Script_Intialization.ms** to load in the user's specifications for quaddle generation.

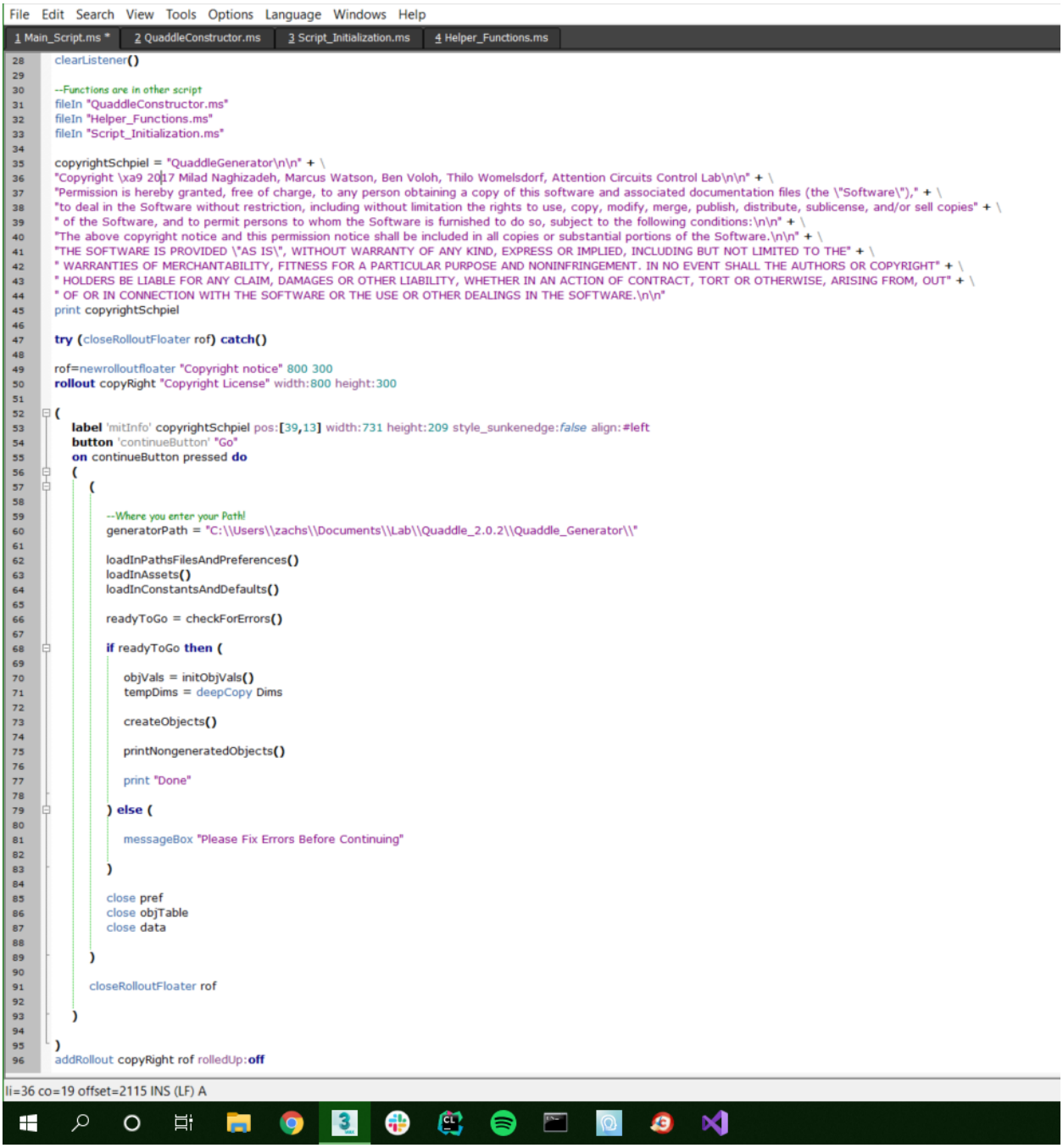
Quaddle_Constructor.ms
The Script that controls the flow of quaddle generation. It calls most of the other scripts in the folder in order to make quaddle parts like arms, heads, and bodies.

Script_Initalization.ms
This Script is the mediator between **Object Table.txt** and **Preferences.txt** with maxscript. It's job is to take what the user enters and construct data structures that the **Quaddle_Constructor.ms** script will understand. It also catches syntax errors and contains a function called by **Main_Script.ms** that catches logic errors. It can be a stand-alone script by switching the Boolean testing in the script to "true". I designed this way so debugging can be fast (you don't have to generate quaddles) and it allows the programmer better control over the flow of the program.

Helper_Functions.ms
This Script helps in preforming miscellaneous tasks for the **Quaddle_Constructor.ms** and is called briefly in the initialization phase by **Main_Script.ms**. It contains zen Boolean and other functions that make the **Quaddle_Constructor.ms** easier to read and write code.

Code Review

Main_Script.ms



```
File  Edit  Search  View  Tools  Options  Language  Windows  Help
1 Main_Script.ms *  2 QuaddleConstructor.ms  3 Script_Initialization.ms  4 Helper_Functions.ms
28  clearListener()
29
30  --Functions are in other script
31  fileIn "QuaddleConstructor.ms"
32  fileIn "Helper_Functions.ms"
33  fileIn "Script_Initialization.ms"
34
35  copyrightSchpiel = "QuaddleGenerator\n\n" + \
36  "Copyright \xa9 2017 Milad Naghizadeh, Marcus Watson, Ben Voloh, Thilo Womelsdorf, Attention Circuits Control Lab\n\n" + \
37  "Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the \"Software\")," + \
38  "to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies" + \
39  " of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:\n\n" + \
40  "The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.\n\n" + \
41  "THE SOFTWARE IS PROVIDED \"AS IS\", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE" + \
42  " WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT" + \
43  " HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT" + \
44  " OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.\n\n"
45  print copyrightSchpiel
46
47  try (closeRolloutFloater rof) catch()
48
49  rof=newrolloutfloater "Copyright notice" 800 300
50  rollout copyRight "Copyright License" width:800 height:300
51
52  (
53      label 'mitInfo' copyrightSchpiel pos:[39,13] width:731 height:209 style_sunkenedge:false align:#left
54      button 'continueButton' "Go"
55      on continueButton pressed do
56      (
57          (
58
59              --Where you enter your Path!
60              generatorPath = "C:\\Users\\zachs\\Documents\\Lab\\Quaddle_2.0.2\\Quaddle_Generator\\"
61
62              loadInPathsFilesAndPreferences()
63              loadInAssets()
64              loadInConstantsAndDefaults()
65
66              readyToGo = checkForErrors()
67
68              if readyToGo then (
69
70                  objVals = initObjVals()
71                  tempDims = deepCopy Dims
72
73                  createObjects()
74
75                  printNongeneratedObjects()
76
77                  print "Done"
78
79              ) else (
80
81                  messageBox "Please Fix Errors Before Continuing"
82
83              )
84
85              close pref
86              close objTable
87              close data
88
89          )
90
91          closeRolloutFloater rof
92
93      )
94
95  )
96  addRollout copyRight rof rolledUp:off
li=36 co=19 offset=2115 INS (LF) A
```

clearListener()

This maxscript command deletes all text that has been displayed by the listener. Remember, the Listener is where the names of objects, Dims, and Vals are printed out. This way, every time the program is run, the listener will contain only information relevant to that generation.

```
fileIn "FunctionsList.ms"
fileIn "Helper_Functions.ms"
fileIn "Init.ms"
```

These lines of code load in the functions from other scripts. The scripts are covered later, however briefly, it contains all the main functions needed get user input, organize Dims and Vals data, check the data for errors, and then generate Quaddles. In the way that maxscript works, functions need to be defined beforehand at the top of the script before they can be called.

Just below the aforementioned code you can see how the copyright pop-up screen is coded. With "try (closeRolloutFloater rof) catch()" we see if there are any other windows open with the name "rof" and close them if they are. A "rollout" is the window that pops up.

```
button 'continueButton' "Continue"
```

^This is defining the button at the bottom of the Window, and attaching the text "Continue" to it. It's given a label of "continueButton"

```
on continueButton pressed do
```

The main body of the main script is executed after the continue button is pressed.

```
closeRolloutFloater
```

After the object generation is completed, this will close the window.
TIP: An easy way to create and edit rollout windows is using the maxscript "rollout editor" GUI. To access it, in MAXscript click **Tools → Edit Rollout.** You will get a window like the one below.

```
generatorPath = ""
```

Next is the location of where the generator is. This tells the function loadInPathsFilesAndPreferences() inside Helper_Functions.ms how to access Object Table.txt and Preferences.txt

The rest of the code is relatively simplified because most of it is defined in other files. Those functions will be discussed in each of their respective files, but I will still explain how to main script flows.

```
loadInPathsFilesAndPreferences()
loadInAssets()
loadInConstantsAndDefaults()
```

These lines of code initialize the program, preparing all the appropriate data. The first function *must* come first. It builds the paths to **Object Table.txt** and **Preferences.txt**, and loads all the preferences from **Preferences.txt**. The second runs all of **Script_Initialization.ms**, accessing the object table, and building the Dims and Vals arrays accordingly. It will also check for syntax errors in this phase. The last function loads in all values that are repeatedly used while generating the quaddles, mostly for mapping and other manipulations that are repeated often.

```
readyToGo = checkForErrors()
```

Here, logic errors are checked, like entering a beak angle but not entering a beak. Having a beak angle is impossible without a beak. Note that if the syntax was incorrect the function will immediately return false and not report any logic errors. Thus, the program makes sure the syntax is correct before it checks for correct logic. If there are no errors, readyToGo will equal true, if not false.

```
if readyToGo then (

	objVals = initObjVals()
	tempDims = deepCopy Dims

	createObjects()

	printNongeneratedObjects()

	print "Done"

) else (

	messageBox "Please Fix Errors Before Continuing"

)
```

Based on readyToGo, it determines what to do next. If there are errors, it will print the message "Please Fix Errors Before Continuing". If not, it goes on with generating.

```
	objVals = initObjVals()
	tempDims = deepCopy Dims
```

Next, objVals and tempDims are initialized. objVals is an array that defines a quaddle. This array is passed to the MakeQuaddle() function in the **Quaddle_Constructor.ms** Script. It takes the values in objVals then builds a quddle with them. Then, when the program moves on to generating a new quaddle, one of the values of objVals will be changed and it will be passed to MakeQuaddle() again. tempDims is used in the **Quaddle_Constructor.ms** as well, however, its purpose is to save the values of mutually exclusive dimensions when there are more than one.

There will be more on this later, however, for now, you should know tempDims is necessary to generate every possible configuration of quaddles given not all dimensions can exist at the same time.

createObjects()

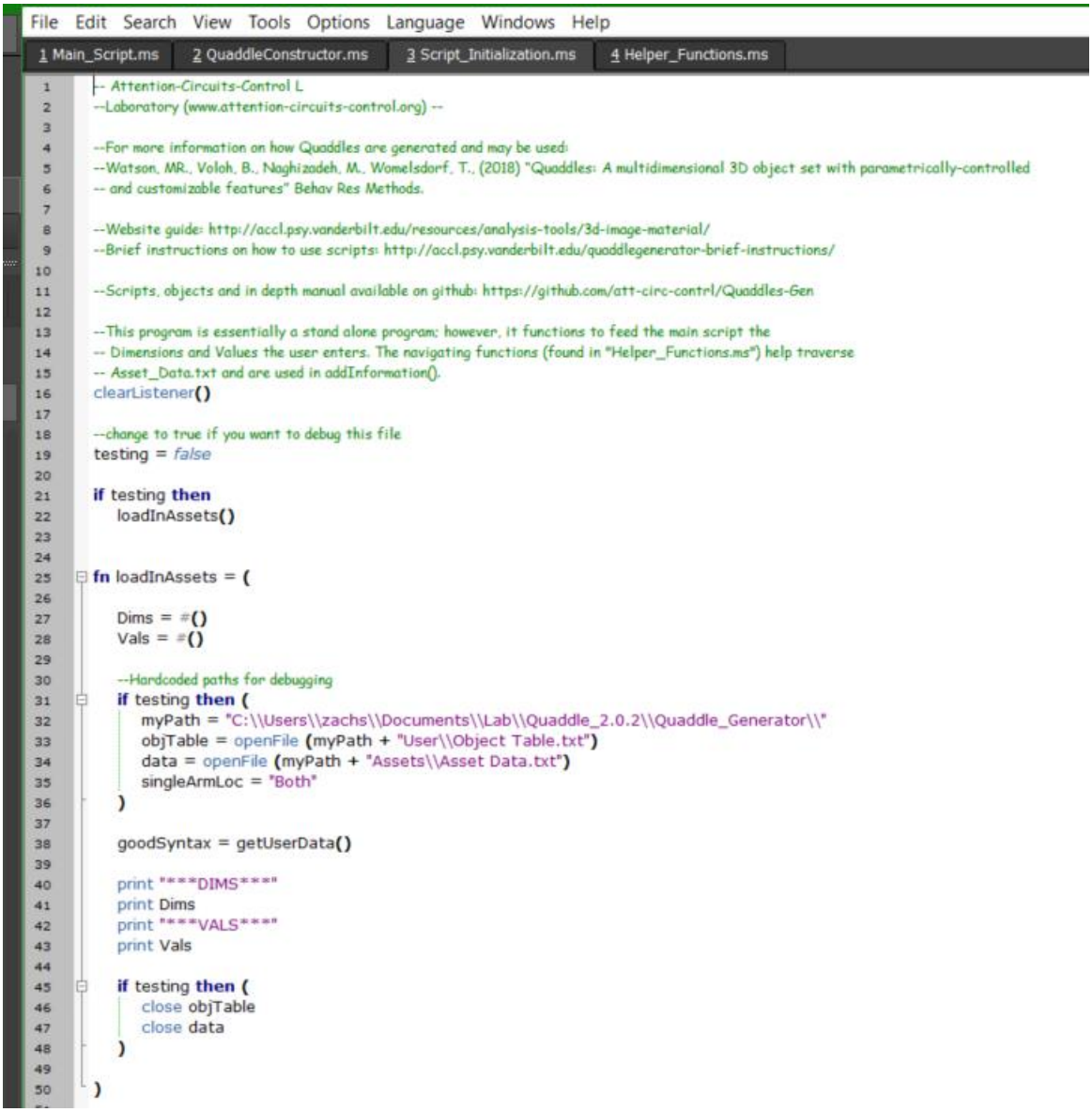
Hopefully this is self-explanatory, but this function generates every single possible combination of quaddles with the values specified by the user. It essentially runs **Quaddle_Constructor.ms**.

printNongeneratedObjects()

If the user specifies that they want to see the names of the objects that could not be made, then this function will print it. At the end of the program, it prints “Done”.

Script_Intialization.ms

Although **Helper_Functions.ms** is called before in **Script_Intialization.ms** in **Main_Script.ms**, I will discuss **Script_Intialization.ms** first. Keep in mind this script can be run on it’s own, it is not dependent on any other scripts in the generator.




```
clearListener()

testing = false

if testing then
        loadInAssets()
```

First, we see clearListener() again. I find this to be a good practice, as it simply lets you debug faster. Next the Boolean testing is initialized. If it is true, you can compile this script on its own, and the program will print out the Dims and Vals it generates. If not in testing mode, it will not call the script's parent function loadInAssets(), and it will just compile the functions.

```
if testing then (
```

```
        myPath = "C:\\Users\\zachs\\Documents\\Lab\\Quaddle_2.0.2\\Quaddle_Generator\\"
        objTable = openFile (myPath + "User\\Object Table.txt")
        data = openFile (myPath + "Assets\\Asset Data.txt")
        singleArmLoc = "Both"
)

goodSyntax = getUserData()

print "***DIMS***"
print Dims
print "***VALS***"
print Vals

if testing then (
        close objTable
        close data
)
```

loadInAssets()'s function is to simply organize the information that getUserData() creates. If in testing mode, it will setup the correct paths to the necessary files **Object Table.txt** and **Asset Data.txt**. If not in testing mode, these paths will already be generated the function loadInPathsFilesAndPreferences() in **Helper_Functions.ms** and be passed to getUserData(). With **Asset Data.txt** now finally mentioned, and as the next code heavily interacts with that file, let me briefly explain what it is.

Asset Data.txt is simply where all possible Dimensions and their respective values are stored. It has a particular structure so the generator can traverse the dimensions and values. The general structure is like so:

Dim1: < “Val1.1”, “Val1.2”, “Val1.3”… >
Dim2: < “Val2.1”, “Val2.2”, “Val2.3”… >

What should be noted: The colon (‘:’) demarcates the end of a dimension name. The less than sign (‘<’) demarcates a list of values. The greater than sign (‘>’) demarcates the end of a list of values. Lastly, the quotation (‘ “ ‘) demarcates the beginnings and ends of value names. With this in place we can continue on to the function called in loadInAssets(): getUserData().

```
--Description: Gets the user data from object table.txt in order to pass it to addInformation()
--Input: none
--Outputs: If the user entered good syntax, or if data needs to be re-entered
--Example Call: getUserData()
fn getUserData
objTable: objTable data: data  Dims: Dims= (

    --keeps track of the number of dimensions
    dimVal = 1
    --keeps track of what line in object table the function is on
    --Allows us to go back in the program if we'd like
    counter = 1

    --puts location of scanner at 0
    seek objTable 0
    seek data 0
    --goto first bracket
    skipToString objTable "["

    while not (eof objTable) do (

        infoAsString = ""

        --Lets us know if a given dimension defined in object table
        --was specified as a quadfile dimension by the user
        dimIncluded = true

        --get data in between brackets
        while (findString infoAsString "]" == undefined) do
            infoAsString = infoAsString + readChar objTable

        --delete bracket in string
        infoAsString = substring infoAsString 1 (infoAsString.count-1)
        --filterstring returns an array of strings parsed by the second argument
        info = filterString infoAsString ","

        --used for syntax error messaging (see if the user entered too high of a number)
        maxVal = findMaxVal()

        for i = 1 to info.count do (

            newVal = info[i]
            newVal = trimLeft newVal
            newVal = trimRight newVal
            newVal = toLower newVal

            --check syntax
            isGoodVal = checkVal(newVal)

            if isGoodVal then
                dimIncluded = addInformation(newVal)
            else  (

                --If there was incorrect syntax, return false
                close objTable
                close data
                return false

            )

        )

        if dimIncluded then (

            resetPosInFile()
            --get the name of the New Dimension
            newDim = readDelimitedString data ":"
            newDim = trimLeft newDim
            append Dims newDim
            dimVal = dimVal + 1

        )

        --goto next dimension value
        skipToString objTable "["
        counter = counter + 1

    )

    modifyArmCount()

    return true

)
```

First thing to note is that getUserData() returns true or false. True is returned when the function did not run into any syntax errors; false is returned when the function ran into syntax errors. Thus, while this function compiles the data from **Object Table.txt**, it makes sure that improper entries are not entered.

Another thing to note is the general flow of the program. For each dimension it has a particular order of events. First, it reads in the info inside the brackets ('[]') from **Object Table.txt**. It then parses the information by the comma. Then for each value entered, it checks for syntax errors, fetches the corresponding value of that info from **Asset Data.txt**, and adds that data to Vals. Lastly it updates the

dimensions if new values were added then moves on to the next dimension. With that said, let's begin with the opening declarations to the function.

```
dimVal = 1
--keeps track of what line in object table the function is on
--Allows us to go back in the program if we'd like
counter = 1
```

The code declares two integers, dimVal and counter. dimVal is used to count the number of dimensions that are included by the user. dimVal increases when values from that dimension have been added to Vals, and the new dimension name will be put at the array index of dimVal. counter is used to keep track of what line of the object table the program is on. It allows the program to return to the beginning of the line during runtime. Since it can do that, the user can enter values out of order because every time it reads in a new entry, the program simply need to reset the position of its scanner to the beginning of the line and iterate to the value specified by the user.

```
--puts location of scanner at 0
seek objTable 0
seek data 0
```

The seek command places the file scanner at the location specified in the second argument of the file specified in the first argument. These commands simply place the scanner at the beginning of the files.

```
--goto first bracket
skipToString objTable "["
```

The skipToString command stops the parser at the character specified by the second argument of the file specified by the first argument. The result of this command is therefore placing scanner at the first opening bracket.

```
while not (eof objTable) do (

    infoAsString = ""

    --Lets us know if a given dimension defined in object table
    --was specified as a quaddle dimension by the user
    dimIncluded = true
```

The while loop goes until it reaches the end of the file. In each cycle of this loop, one dimension is evaluated. We declare the infoAsString, which is the variable that stores all the values that the user entered between the brackets as a string. An example would be "1,34,4-7" or "all". dimIncluded is assumed to be true. This is the variable that will tell the program to add a given dimension to Dims.

```
--get data in between brackets
while (findString infoAsString "]" == undefined) do
	infoAsString = infoAsString + readChar objTable
```

This segment of code builds infoAsString. The loop runs until the character ‘]’ is added to infoAsString. It uses the readChar command which returns the character just after the scanner location. It then increases the scanner location by one. So, the program goes character by character to fill infoAsString.

```
--delete bracket in string
infoAsString = substring infoAsString 1 (infoAsString.count-1)
```

Update infoAsString to get rid of the bracket.

```
--filterstring returns an array of strings parsed by the second arguement
info = filterString infoAsString ","
```

This segment of code declares a new array, info. info is used to organize the values in infoAsString into a traversable data structure. The command filterString builds an array whose values are strings in between the specified parser string in the second argument. In our case, the parser string is “,”. Using the previous example, here is what it does visually.

infoAsString = “1,34,4-7” → filterString → info = [“1”] [“34”] [“4-7”]

Next the max possible value of the given dimension is found. This will be used for syntax messaging, in case the user enters in a value that is too large. The function code will be reviewed in the **Helper_Functions.ms** section.

```
maxVal = findMaxVal()
```

Finally, the program evaluates the user’s input. Since the information is split into an array, it enters a for loop that determines what to do with the information at each index.

```
for i = 1 to info.count do (

	newVal = info[i]
	newVal = trimLeft newVal
	newVal = trimRight newVal
	newVal = toLower newVal
```

It begins simply by standardizing the input. It stores the value into the string newVal then gets rid of whitespace and makes it lowercase (in the event that “none” or “all” is entered, this is relevant).

Next, it checks the syntax of newVal, by calling the function checkVal().

```
	--check syntax
	isGoodVal = checkVal(newVal)
```

checkVal() returns a boolean that tells the program if the entry was good or not. newVal is passed in as userVal. The function begins with:

```
fn checkVal userVal maxVal: maxVal = (

	if doNotUseDim(userVal) or useAllValsinDim(userVal) or isRange(userVal) then
		return true
```

It first determines whether or not the inputted value is any of the special items i.e. “none”, “all”, or a range.

```
	if userVal as integer == undefined then (
		messageBox ("An invalid character was entered in the value: " + userVal)
		return false
	)
```

Next, it checks to see if an integer was entered.

```
	if (userInt >= maxVal) then (
		resetPosInFile()
		messagebox ("The value \"" + userVal + "\" is too large for the dimension \"" +
		(readDelimitedString data ":") + "\"")
		return false
	)
```

Finally, it verifies the user entered a value smaller than the max possible value. If it passes the tests, then it returns true.

Note: The evaluations are simplistic as of now; however, the structure is built so that it can easily be expanded and become more sophisticated.

If the entry was ok then it calls addInformation() to fetch relevant data and build Dims and Vals. If not, then it closes the files and returns false for getUserData(), signaling to the program that the user’s input was erroneous.

```
if isGoodVal then
		dimIncluded = addInformation(newVal)
else	(

		--If there was incorrect syntax, return false
		close objTable
		close data
		return false

	)
```

Let's say that the entry was good, then addInformation() is called. Note that this is the function that traverses **Asset Data.txt**, which means there will be functions like endOfLine() and nextVal() to help addInformation() do what it needs. These, and the other helper functions found in this method, will be reviewed just after addInformation(). This is designed so you can first understand what is happening on a high level, then understand how the parsing works on the lower level. Also note that it returns a boolean. This boolean tells the program if a dimension was added or not.

```
fn addInformation
str dimVal: dimVal Vals: Vals Dims:Dims counter: counter = (

	resetPosInFile()
```

This first function call resets the position of the file scanner to the beginning of the dimension in **Asset Data.txt**. Here is the code for that function.

```
seek data 0
for i = 1 to counter*2 do
	skipToNextLine data
```

Remember counter? This is its most useful function. By storing what line the program is on, it can easily go to the beginning of that line with this code. It simply goes to the beginning of the file, then skips lines until it reaches line counter.

After the position is set, we check to see if "none" what entered.

```
--If user enters "none", there is no new dimension
	if doNotUseDim(str) then
		return false
```

If the input was "none", it returns false, ending addInformation() and signaling that no dimension was added. Obviously, this code is dependent on doNotUseDims(), but I will review this later with the rest of the helper methods. Next, we check if "all" was entered.

```
--If user enters "all", add all values from asset_data.txt
if useAllValsInDim(str) then (

	while not endOfLine() do (

		nextVal()
		addDataToArray()

	)

	return true

)
```

If all was entered, then it will add every value it comes across to the Vals array until it reaches the end of the given line. Then it will return true, signaling those values were added, so it can add that dimension to the Dims array.

If “all” was not entered then it positions itself correctly by going to the first value of the dimension of asset data,

```
--enter correct position in file
nextVal()
```

and checks to see if a range was entered.

```
if isRange(str) then (

	rangeVals = filterString str "-"

	startVal = rangeVals[1] as integer
	endVal = rangeVals[2] as integer

	gotoVal( startVal as string )

	for i = startVal to endVal do (

		addDataToArray()
		nextVal()

	)

	return true

)
```

If it is a range, we use filterString, the same technique that we used earlier with info, to get the two values that mark the beginning and end of the range. This new data is stored in rangedVals and then those values are adopted by startVal and endVal. Then the program traverses to the first value of the range using gotoVal(), and, finally, it adds the data to Vals until the end value. If the value entered was not a range, then this code executes:

```
gotoVal(str)
addDataToArray()

return true
```

Finally, to complete addInformation(), and if all the other if statements are not entered, then the program assumes a single number is to be evaluated. It goes to that value, adds it to Vals, and returns true.

Helper Functions of addInformation()

addInformation() contains the following helper functions:

doNotUseDim()
useAllValsInDim()
isRange()
endOfLine()
nextVal()
gotoVal()
addDataToArray()

doNotUseDim() simply checks if the str value equals “none” using a zen boolean. Recall that earlier in the program, we made it all characters lowercase, so if “NoNE” was entered, this function will still return true.

```
fn doNotUseDim s = (

    return s == "none"

)
```

useAllValsInDim() does the opposite of the former. It checks to see if the str value = equals “all”. The capitalization comment applies here too.

```
fn useAllValsInDim s = (

    return s == "all"

)
```

isRange() checks to see if the value entered is a range. It is not very sophisticated, which is alright since the error chekcers save it from cases in which it could be problematic.
NOTE*** If you want to extend the programs ability to process negative numbers, this functions will HAVE to be modified.

```
fn isRange s = (

    return findString s "-" != undefined

)
```

endOfLine() checks to see if the program has reached the end of the dimension space. This is demarcated by “>”, so the program checks for that character

```
fn endOfLine =(

	return (readChar data) == ">"

)
```

nextVal() moves to the next value of whatever dimension the program is on. It uses the “ “ “ that encapsulate the characters of the values to traverse it. This way, whenever the program moves to the next value, it is ready to read in the value if it calls for it.

```
fn nextVal = (

	skipToString data "

)
```

gotoVal() iterates the program down the list of values by the length enumerated by the user. It also uses determineNum() which determines if the number is one or two digits.

```
fn gotoVal s = (

	--parses double digit numbers
	loc = determineNum(s)

	if (loc != 0) then (

		for i = 1 to (loc*2) do
			nextVal()

	)

)
```

addDataToArray() takes whatever value the program is at, and adds it to Vals.

```
fn addDataToArray dimVal:dimVal Vals: Vals = (

	if Vals[dimVal] == undefined then
		append Vals #()

	append Vals[dimVal] (readDelimitedString data "\"")

)
```

One thing to note is that the if statement in this block of code grows the array as necessary. If there is new dimension being added, it will grow to the appropriate size. It also means that Vals can be passed with any number of elements from 1 to how many dimensions there are (of course the later number changes if you add any).

Remember that addInformation() was returning true or false into dimIncluded to signal the program if a value was added. If a dimension was included, then it runs the following code:

```
if dimIncluded then (

	resetPosInFile()
	--get the name of the New Dimension
	newDim = readDelimitedString data ":"
	newDim = trimLeft newDim
	append Dims newDim
	dimVal = dimVal + 1

)
```

This simply adds the new dimension to Dims. It also increments dimVal to track where the next dimension will be put.

Finally, the program reaches the end of the while loop, and goes to the next dimension and increment counter.

```
--goto next dimension value
skipToString objTable "["
counter = counter + 1
```

After the while loop ends, the program knows that all syntax that was checked was appropriate, so it returns true to be stored in goodSyntax; however, before getUserData() is complete, it calls modifyArmCount(). This function modifies Vals, adding the values “1L” or “1R” based on if the user specified which side a single arm should be generated in Preferences.txt. The code is the following:

```
if objHasArmCount() then (

	if (findItem Vals[ getArmCountLoc() ] "1" != 0) then (

		if singleArmLoc == "Left" or singleArmLoc == "Both" then
			insertItem "1L" (Vals[  getArmCountLoc()  ]) (findItem Vals[  getArmCountLoc()  ] "1")

		if singleArmLoc == "Right" or singleArmLoc == "Both" then
			insertItem "1R"  (Vals[  getArmCountLoc()  ] ) (findItem Vals[ getArmCountLoc() ] "1")
		deleteItem (Vals[ getArmCountLoc() ]) (findItem Vals[ getArmCountLoc() ] "1")
```

The logic is simple. If there is an arm count, check to see if it contains the value “1”. If it does, and the user specified that it should be on the left or they want both left and right, add “1L” to Vals. If the user specified it should be on the right or both, add “1R” to Vals. Lastly, delete the value “1” since that value does not mean anything to the program.

The former discussion is what mainly comprises **Script_Intialization.ms**. After getUserData(), the following code is executed.

```
print "***DIMS***"
print Dims
print "***VALS***"
print Vals

if testing then (
	close objTable
	close data
)
```

This simply prints Dims and Vals to the maxscript listener to report what it generated. If **Script_Intialzation.ms** was in its testing mode, it will close the relevant files. This ends the initialization phase of the program, and we move back to **Main_Sctipt.ms**.

Recall that after the assets are loaded in, the program calls checkForErrors(). This will check for logical errors. It uses the constructed Dims and Vals in order to verify those certain logical requirements that need to be met. Note the opening if statement:

```
if goodSyntax == false then
	return false
```

This means that if the syntax was not good, it will not execute its logical checks; thus, the user must input good syntax before being able to check to see if the logic was correct. If the syntax was good, it will move on.

```
local mybool = true

if not objHasBody() then (

	messageBox "You must have a body"
	mybool = false
	return mybool

)

if objHasCompColour() and not objHasMainColour() then (

	messageBox "You must have Main Colour if you have a Comp Colour"
	mybool = false
	return mybool

)

if objHasPattern() and (not objHasCompColour() or not objHasMainColour() ) then (
```

```
            messageBox "You must have both main and comp colours to have a pattern"
            mybool = false
            return mybool

        )

        if not objHasArmCount() and
          ( objHasArmAngle() or objHasArmEnds() ) then (

            messageBox "You must have an Arm Count dimension to have have an Arm Angle or
            Arm End "
            mybool = false
            return mybool

        )

        if not objHasBeak() and
          ( objHasBeakEnds() or objHasBeakAngle() ) then (

            messageBox "You must have a Head to have a Beak or Beak Angle"
            mybool = false
            return mybool

        )

        if not objHasBeak() and objHasBeakAngle() then (

            messageBox "You must have an Beak to have an Beak Angle"
            mybool = false
            return mybool

        )

        return mybool
```

I put all the checks here since they all have the same structure. Here is the purpose of each in order that they appear in the code:

(1) There must be a body.
(2) There must be a main colour if there is a comp colour.
(3) There must be both main colours and comp colours if there is a pattern
(4) There must be an Arm Count if there is an Arm Angle or Arm End.
(5) If there is a Beak or BeakAngle, there must be a head.
(6) If there is a Beak Angle, there must be a beak.

If none of the if statements are entered, it will return mybool as true, signaling that the generation is ready to go. Thus, with all data initialized and verified, the program moves on to generating the quaddles with createObjects().

Quaddle_Constructor.ms

Once createObjects() is called, the program enters into **Quaddle_Constructor.ms**, for the most part. **Helper_Functions.ms** is called quite a bit, but the main execution happens in **Quaddle_Contructor.ms**.

There are a few things to note about the flow of the program. Its basic order events is as follows:

(1) If there are mutually exclusive dimensions, modify Dims so that the generation cycle can activate with a single and given mutually exclusive dimension. This happens in createObjects().
(2) Generate all possible quaddles with the given Dims and Vals in generateQuaddles()
(3) Reset Dims
(4) Modify Dims to generate with the next mutually dimension and generate all possible quaddles with that dimension (repeat with the next dimension).

The reason that we need to compensate for the mutually exclusive dimensions if primarily because of the way the makeQuaddles() generates the quaddles. The function is implemented to recurse over every single possible permutation of dimension combination, so if Dims contains more than one mutually exclusive dimension in for generation, it will attempt to generate them at the same time. This produces confusion for the program and only one mutually exclusive dimension will ever be generated.

To stop any given mutually exclusive dimension from generating, we simply rename that dimension's name in Dims to a name the program cannot recognize. The default name is "N/A" and is stored in noGen. This works because the program decides whether or not to generate a given dimension if it can locate it in Dims (you will see the code in makeQuaddle()). If it cannot locate it, it will not execute the code to generate it. Therefore, by giving makeQuaddle() sets of Dims that contain only one mutually exclusive dimension that is not blocked out, the program avoids the problems that come with having mutually exclusive dimensions.

One last thing to note is that the dimensions "Main Colour", "Comp Colour", and "Pattern" are all dimensions that compose of one mutually exclusive dimension because they are dependent on each other. Since this is the case, if one of these dimensions is labeled as mutually exclusive, then ALL of them must be labeled as so. In other words, they must be treated as one entity. So, there is code that compensates for this in the generation. I will point it out as we come across it.

Another thing to note is that there are many helper functions in this program, and they are called a lot. For this tutorial, if we come across one in the code, I will explain it right where in encounter in the program; however, I will mention a few of the ones called frequently here. For the most part they are zen booleans that increase the readability of the code. You can find these defined at the end of **Helper_Functions.ms**.

(1) getDimLoc()

    Example Calls: getBodyLoc(), getMainColourLoc(), and getArmEndsLoc()

    This returns the index location of a given dimension in Dims.

    Code Example:

```
fn getBodyLoc Dims: Dims = (

    return (findItem Dims "Body")

)
```

Remember that the array objVals defines the quaddle to be generated; however, what's convenient about objVals is each value at a given index corresponds to the index of the dimension in Dims that is the parent of that value. Therefore, if you want to access a value of a certain dimension of the current object being generated, all you have to call is objVals[getDimLoc()].

(2) dimVal()

Example Calls: smoothnessVal(), armCountVal(), beakVal()

This returns the value of the dimension you specify from objVals.

Code Example:

```
fn bodyVal Dims: Dims = (

    return ( objVals [ getBodyLoc() ] )

)
```

Notice this does what the previous one is useful for; however, you, as the programmer, will find both useful as you edit the scripts.

(3) objHasDim()

Example Calls: objHasBody(), objHasAspectRatio(), objHasPattern()

This returns true if Dims contain the specified dimension. It returns false if not.

Code Example:

```
fn objHasBody Dims: Dims  = (

    return (findItem Dims "Body" != 0)

)
```

These functions are useful for knowing what to generate, and when to generate a given dimension.

With these in place, let's begin with createObjects()

File Edit Search View Tools Options Language Windows Help

1 Main_Script.ms | 2 QuaddleConstructor.ms * | 3 Script_Initialization.ms * | 4 Helper_Functions.ms *

```
28  --NOTE: Since the Dimensions "Colour" and "Pattern" are mutually exclusive but are dependant on each other
29  --          account; however, it only works granted that Colour comes before Pattern in  Dims.
30  fn createObjects Dims: Dims = (
31
32      if numOfME() <= 1 then
33          generateQuaddles(1)
34      else (
35
36          for iter = 1 to Dims.count do (
37
38              if isMutuallyExclusive(iter) then (
39
40                  for y = iter+1 to Dims.count do (
41
42                      if Dims[iter] == "Main_Colour" then (
43
44                          while (Dims[y] == "Pattern" or Dims[y] == "Comp_Colour") do
45                              y = y + 1
46
47                      )
48
49
50                      if isMutuallyExclusive(y) then
51                          Dims[y] = noGen
52
53                  )
54
55                  generateQuaddles(1)
56
57                  resetDims(iter)
58                  if Dims[iter] == "Main_Colour" then (
59
60                      if Dims[iter + 1] == "Comp_Colour" then
61                          Dims[iter + 1] = noGen
62
63                      if Dims[iter + 2] == "Pattern" then
64                          Dims[iter + 2] = noGen
65
66                  )
67
68                  Dims[iter] = noGen
69
70              )
71
72          )
73
74      )
75
76  )
```

It opens with an if else statement that calls numOfME(), which simply returns how many mutually exclusive dimensions there are. Here is the code for that function.

```
fn numOfME Dims: Dims = (

	local num = 0

	if objHasPattern() or objHasMainColour() then
		num = num +1

	if objHasIcon() then
		num = num +1

	if objHasFractal() then
		num = num +1

	return num

)
```

Simply, if there is a mutually exclusive dimension, it will increase num and return that variable.
It helps us with the logic of generation.

```
if numOfME() <= 1 then
		generateQuaddles(1)
	else (
```

If there is only one mutually exclusive dimension, then the program does not need to waste time compensating for multiple mutually exclusive dimensions. The program will only need to run the else block if there are more than one mutually exclusive dimensions.

If there is more than one, it runs through every single dimension in Dims, and it checks to see if the dimension at iter is mutually exclusive using the function isMutuallayExclusive().

```
		for iter = 1 to Dims.count do (

			if isMutuallyExclusive(iter) then (
```

Note the code for isMutuallyExlcusive() is:

```
if Dims [loc] == "Pattern" or Dims[loc] == "Main_Colour" or Dims[loc] == "Comp_Colour" then
	return true

if Dims[loc] == "Icon" then
	return true

if Dims[loc] == "Fractal" then
	return true

return false
```

Simply, it returns true if the dimension is any dimension that the programmer specifies is mutually exclusive. If not, it returns false.

If the dimension is mutually exclusive, then we execute the following:

```
for y = iter+1 to Dims.count do (

    if Dims[iter] == "Main_Colour" then (

        while (Dims[y] == "Pattern" or Dims[y] == "Comp_Colour") do
            y = y + 1

    )


    if isMutuallyExclusive(y) then
        Dims[y] = noGen
```

After the first mutually exclusive dimension is found, the program renames all the other mutually exclusive dimensions following it to a name that will keep it from being read in by the program. More specifically, from iter to the end of Dims, if it finds a mutually exclusive dimension, it will rename it to a string the program can't recognize. Note that the first if statement and the while loop consider the dimensions that are dependent on each other, like talked about earlier. This way if Main Colour was found the other two dimensions that compliment it will be handled accordingly.

After renaming the other mutually exclusive dimensions, the program calls generateQuaddles(); however, before we dive into that we will finish this section on how the program then renames the other mutually exclusive dimensions to do the next set of generation.

```
resetDims(iter)
```

First resetDims() is called, which returns Dims to its original values, the ones that it contains at initialization. Here is the code for that function:

```
for j = iter to tempDims.count do
    Dims[j] = tempDims[j]
```

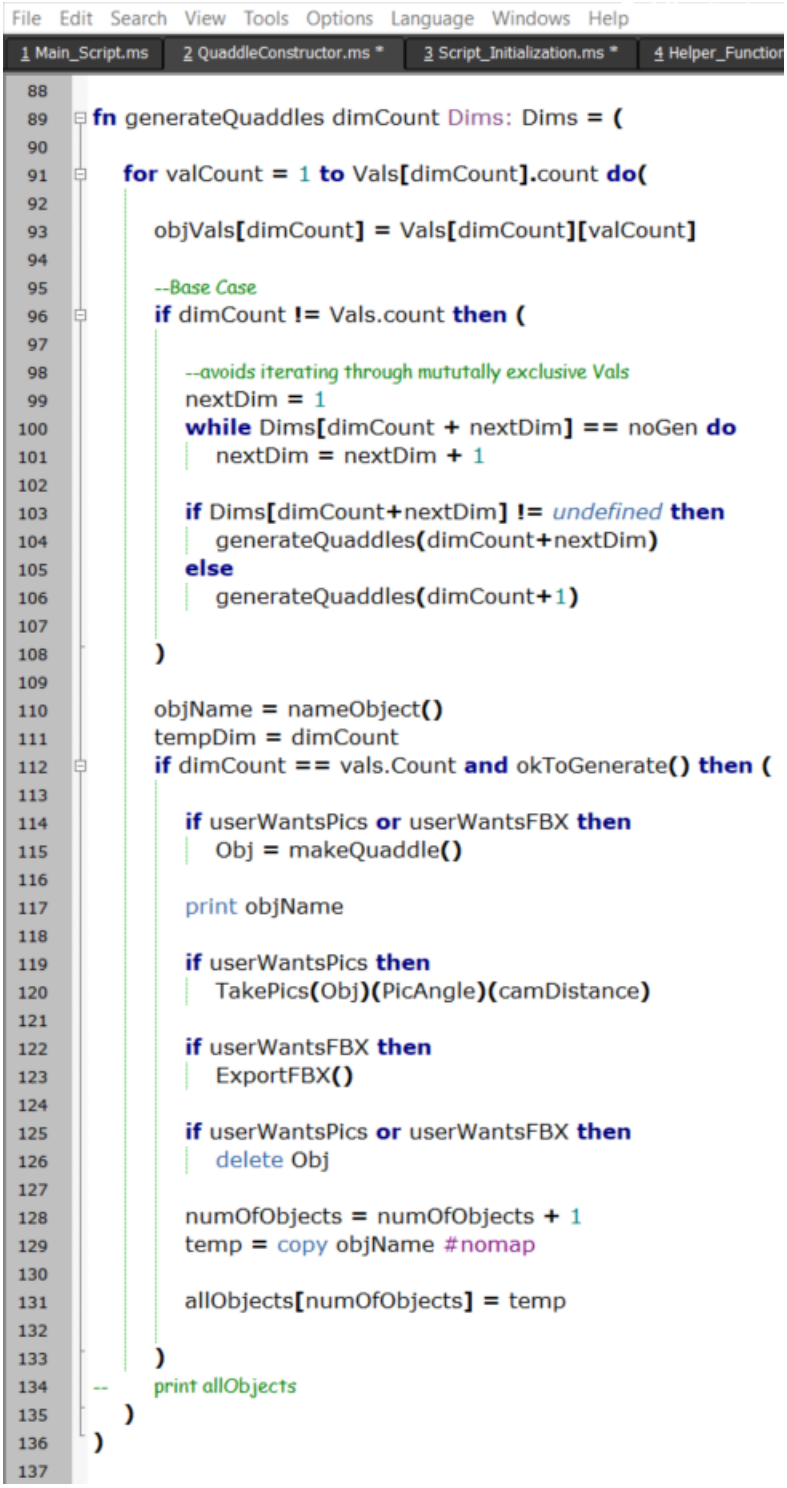
From the current position in createObjects() (iter) to the end of Dims, it puts the index of Dims equal to the index at tempDims, the array initialized in **Main_Script.ms**.

After Dims is reset, the program executes the following:

```
if Dims[iter] == "Main_Colour" then (

        if Dims[iter + 1] == "Comp_Colour" then
                Dims[iter + 1] = noGen

        if Dims[iter + 2] == "Pattern" then
                Dims[iter + 2] = noGen

)

Dims[iter] = noGen
```

The first if statement takes care of the dependency of the three dimensions that are grouped together. After that, it simply renames the current dimension to the noGen string, so that there will not be another generation cycle with that mutually exclusive dimension. After this, the for loop iterates and begins to search for the next mutually exclusive dimension.

With an understanding of how the factor of mutual exclusivity is handled, we can now enter makeQuaddle(). This is where the program defines objectVals so a quaddle can be made.



Although this was mentioned earlier, the first thing to note is that this is recursive function. The base case is when dimcount == vals.count. If the former is not the case, the program will call itself with the parameter parameter as “dimCount + 1/nextDim”. To understand how this function works let me explain the general flow.

(1) The program enters a for loop which iterates through all values of whatever the dimension at index dimCount.
(2) However, if the dimCount does not equal the number of total dimensions that there are, it will go to call itself with dimCount increased by a certain number.
(3) It will do this until it reaches the last dimension. Once it reaches the last dimension, it is as if all for loops are primed to iterate through every single possible value it contains.
(4) Then it will make a quaddle as it recurses back, updating the objVals as it goes.
(5) However, in the process of recursing back, it will continue to call itself again, increasing dimCount. This is because dimCount won't equal the number of total dimensions that there are. This is slightly confusing, but this what permits the program get every single possible combination.
(6) Eventually it will complete recurse back and the function will be complete.

With this in place, let's look at the code.

```
for valCount = 1 to Vals[dimCount].count do(

        objVals[dimCount] = Vals[dimCount][valCount]
```

Here is the for loop that iterates through all the values of a given dimension. Following it is the code that updates and defines the new objVals, so that the new and different quaddle can be made. Then we enter the block of code that defines the recursive properties.

```
    if dimCount != Vals.count then (

        --avoids iterating through mututally exclusive Vals
        nextDim = 1
        while Dims[dimCount + nextDim] == noGen do
                nextDim = nextDim + 1

        --avoids the case where the program iterates outside the index of Dims
        if Dims[dimCount+nextDim] != undefined then
                generateQuaddles(dimCount+nextDim)
        else
            generateQuaddles(dimCount+1)

    )
```

Like mentioned before, if dimCount does not equal the number of total dimensions, this code will be executed, and it will recurse. The while loop iterates over the mutually exclusive dimensions, so that they are not considered into the recursive looping. The next if statement covers the case where a mutually exclusive dimension is the last dimension. If this is not here, there are instances where the program will attempt to access unallocated memory locations. Else, if the next value is not mutually exclusive, it will just go to next dimension, or iterate by 1.

Afterward, the object will be named using nameObject().

```
local oName = objVals[1]
if objvals.count > 1 do (
	for iter = 2 to objVals.count do (
		if Dims[iter] != "N/A" then
			oName = oName + " + " + objVals[iter]
	)
)

return oName
```

This function simply takes the values of all included dimensions and adds them together with a "+" to name the quaddle.

Then we reach the if statement which has two conditions. One is that dimCount == Vals.count, as discussed earlier, but the other is okToGenerate(). This function's primary purpose is to check if certain colour combinations exist. If we do not check this, the program will attempt to access a file that it cannot. Here is the code:

```
if not objHasMainColour() or not objHasCompColour() or not objHasPattern() then
	return true

local pcPath = assetPath + "Patterns and Colours\\"
local texturePath= ( pcPath + "Pattern(" + patternVal() + ")+Colour(" + mainColourVal() + "_" +
compColourVal()+ ").png" )

if (getFiles texturePath).count == 0 then (
	if printNotPossibleObjects then
		append notPossibleObjects objName
	return false
)

return true
```

It begins with an if statement where if any of the dimensions "Main Colour", "Comp Colour", or "Pattern" do not exist, it returns true. This is the case since the only situation in which we need to check if a file exists is if all three of these dimensions are present. All other cases are either handled by checkForErrors() or makeQuaddle(). If it passes through that if statement, it builds the file name, and then searches for it. If it does not exist, then it will add the name to notPossibleObjects, if appropriate, and return false. If the file is there, it will return true.

Next the object is generated and exported.

```
if userWantsPics or userWantsFBX then
	Obj = makeQuaddle()

print objName

if userWantsPics then
	TakePics(Obj)(PicAngle)(camDistance)

if userWantsFBX then
	ExportFBX()

if userWantsPics or userWantsFBX then
	delete Obj

numOfObjects = numOfObjects + 1
temp = copy objName #nomap

allObjects[numOfObjects] = temp
```

The flow of quaddle making is defined here. Simply it makes the quaddle, takes pictures of it and exports it, if necessary, deletes the object, then adds its name to allObjects. We will begin with makeQuddle().

The function opens with some housekeeping.

```
--Ensure Dimensions are labeled correctly
if findItem Dims "Color" != 0 do
	Dims[findItem Dims "Color"] = "Colour"

--qArray is simply an array of all the components which will comprise of the quaddle
qArray = #()
```

We change the name to Colour if it is Color and initialize qArray, which is an array that will hold all the objects that make up the quaddle. Then the program defines important variables.

```
if objHasAspectRatio() then (

	--determine relevant values given an aspect ratio
	if ( aspectRatioVal() == "Null") then (
		ratio = 0
		camDistance = 165
		armScale = 1
	)

	if ( aspectRatioVal() == "Long") then (
		ratio = 0.3
```

```
                camDistance = 215
                armScale = 1.1
        )

        if ( aspectRatioVal() == "Short") then (
                ratio = -0.2
                camDistance = 145
                armScale = 0.9
        )

) else (

        ratio = 0

)
```

The program determines the variables ratio, camDistance, and armScale based on the object's aspect ratio. ratio is used in body generation, multiplying certain values by it to either squish it or elongate it. camDistance is used when taking pictures, making each quaddle appear around the same size no matter the aspect ratio. armScale is used to make the arms of a quaddle the appropriate size for the body.

With these in place we can call execute the next line:

```
bodyResult = MakeBody( bodyVal() )
```

We call MakeBody() here, which actually makes the body of the quaddle. Here is the code for the function.

This function is called to make the main body of the object.

```
if objHasPattern() then
                local pattern = patternVal()
        else
                local pattern = "None"

        if (body == "Oblong") then (

                bodyResult = MakeOblongBody()
                controlPoint = bodyResult[1]
                objbody = bodyResult[2]
                polygonner = bodyResult[3]

        )

        else if (body == "Pyramidal") then (

                bodyResult = MakePyramidBody()
                controlPoint = bodyResult[1]
```

```
            objbody = bodyResult[2]
            polygonner = bodyResult[3]

        )
```

Depending on the body feature value, more specific functions such as MakeOblongBody() are called. Only two are shown here for simplicity, however one exists for each body shape.

```
    fn MakeOblongBody pattern  = (include "Script_OblongBody.ms")
```

When these body functions are called, they make reference to separate scripts. With "include" its as if we're essentially copy and pasting the code in.

```
    append qArray objbody
```

objbody is added to the initially empty qArray

```
    bodyOutput = #(objbody,polygonner,qArra
    y)
    return bodyOutput
```

The end result output of MakeBody().

After the body is made, we save a few variables.

```
    bodyZMax = objBody.max[3]
    bodyZMin = objBody.min[3]
    bodyHeight = objbody.height
```

All of these are used in some specific functions like placing the head on the body, applying the Icon and Fractal textures, and taking pictures. You will see how these are used throughout the rest of the documentation. Next, we deal with smoothness.

```
    if objHasSmoothness() and applySmoothnessToBody then (

        if smoothnessVal() != "Smooth" then (

            smoothOut = ManipulateSurfaceTexture(objbody)(bodyVal())
            objbody = smoothOut[1]

        )

    )
```

The first if statement simply verifies that the smoothness is incorporated into Dims and the user wants to apply the smoothness to the body. If the value is smooth then, the program doesn't have to do anything, and this is the purpose of the second if statement. Then the program reaches

ManipulateSurfaceTexture(), which is used to actually apply the texture. Here is how ManipulateSurfaceTexture() operates.

```
if (smoothness=="Wrinkled") then (

        wrinkler = displace()
        wrinkler.strength = 6.5
        wrinkler.decay = 1
        noisePath = textureMainPath + "noise2d" + ".png"
        wrinkler.bitmap =  openBitMap noisePath


        wrinkler.maptype = 2 --spherical mapping
        wrinkler.length = 100
        wrinkler.width = 100
        wrinkler.height = 100
        addmodifier objbody wrinkler)

)
```

If you want a wrinkly object, then this function will add the “displace” modifier to the surface, which can contort and deform the shape. Here I give it reference to an png file called “noise2d” in order for it to generate random contortions throughout the entire surface.
Similar to the UVW mapping techniques, how this displace function is applied to the object can be tweaked by various paramaters such as maptype, and map length etc.
Note: If you want to generate wrinkled objects than you need to make sure that you have the noise2d.png file in your textures folder.

```
if (smoothness=="Geodesic") or (smoothness=="Blocky") then (
)


if (smoothness=="Inward_Protrusions") or (smoothness=="Outward_Protrusions")  then(
)


--Adding Hair to object if object is specificied to be hairy
if (smoothness=="Hairy") then(
)
```

If you want more information about how some of these other surface topologies are generated, please refer to the end of **Part 1** in the Quaddle 1.0 manual to see how they are generated. In the script, you are essentially inputting the values that you saw earlier in the manual.

After the smoothness is applied, the program moves on to the texture.

```
if not objHasIcon() and not objHasFractal() then (

	textureResult = ApplyColouredTexture(objbody)
	assetPath = textureResult[1]
	map = textureResult[2]
	objbody = textureResult[3]
	outMap = textureResult[4]

)

if objHasIcon() then
	ApplyIFTexture("Icon")(objbody)

if objHasFractal() then
	ApplyIFTexture("Fractal")(objbody)
```

Note there are two functions that apply textures: ApplyColouredTexture() and ApplyIFTexture(). The former handles the textures that involve colours or patterns, while the latter handles Icons and Fractals. They are split because ApplyColouredTexture() has a little less handling required to display them well. ApplyIFTexture() has to focus on clear presentation of the images, so I put it in it's own function. It also saved a little bit of code from being written. With that said, let's look at ApplyColouredTexture().

It opens with building the path to the file with the colours.

```
pcPath = assetPath + "Patterns and Colours\\"
```

Next, you'll see code for pattern inversion that looks like the following:

```
if (findItem Dims "Pattern_Inversion" != 0) then (

	if  (objVals[findItem Dims "Pattern_Inversion"] == "Inverted") then
		invertText = "_inv"
	else
		invertText = ""

) else
	invertText = ""
```

This is not implemented; however, it should be relatively easy. The reason I kept from implementing it is only 5 patterns are capable of inversion and it is very hard to tell when the pattern in inverted. It is not easy for monkeys, nonetheless humans. So, this code will always default inverText to "".

After the inverted pattern bit, the pattern value is determined.

```
if not objHasPattern() then
	local pattern = "Solid"
else
	local pattern = patternVal()
```

If there is no pattern, the program defaults to solid, if not, it gets the pattern value and stores it pattern. Then the colour application is determined.

```
if not objHasMainColour() and not objHasCompColour() and pattern == "Solid" then
	colour = (greyColour + "_" + greyColour)

else if not objHasMainColour() and not objHasCompColour() and pattern != "Solid" then
	colour = "7000000_5000000"

else if objHasMainColour() and not objHasCompColour() and pattern == "Solid" then
	colour = mainColourVal() + "_" + mainColourVal()

else if objHasMainColour() and not objHasCompColour() and pattern != "Solid" then (

	colour = mainColourVal() + "_5000000"
)

else (
	colour = ( mainColourVal() + "_" + compColourVal() )
)
```

Each one of these statements covers a given case. They are:

(1) Where there is no main colour or comp colour and the pattern is solid
(2) Where there is no main colour or comp colour and the pattern is not solid
(3) Where there is a main colour, no comp colour, and the pattern is solid
(4) Where there is a main colour, no comp colour, and the pattern is not solid
(5) Where there is both a main and comp colour

Each time the text values will be stored into colour. Note that pattern secondary grey default color is 5000000 and the default primary colour is 700000. This is why case 2 enumerates those default values for a non-solid pattern.

Once the program has gone through this logic, it build the complete path to the file.

```
texturePath= pcPath + "Pattern(" + pattern + ")+Colour(" + colour + ")" + invertText + ".png"
```

Then it is determined if it is applying the texture to the head or body.

```
if isHead then
        body = headVal()
else
        body = bodyVal()
```

Following, our map is created and how the map is applied to the body is determined. Each if block specifies their own settings, but here are two examples.

```
if  pattern == "HStripes" or
        pattern == "VStripes" or
        pattern == "Squiggly" or
        pattern == "Triangular" then (
        map.length = 30
        map.width = 30
        map.height = 35
        map.utile = 2.5
        map.vtile = 1
        map.cap = false
        addmodifier objbody map
)

else if pattern == "DStripes" then (
        map.length = 20
        map.width = 20
        map.height = 25
        map.utile = 4
        map.vtile = 1
        map.cap = false
        addmodifier objbody map

)
```

Above we see an example of how we might apply different wrapping techniques depending on the texture, or object that it's being applied to. In the above example, a DStripes pattern will have a different setting than the triangular pattern.

The version of the script you may be using may have different specifications for these parameters with different conditionals for different objects, however the key point here is that the following parameters are tweaked/adjusted to make sure that 2D png images are optimally applied to each object;

map.maptype
map.utile
map.vtile
map.height
map.width
map.length
map.cap

You will notice that in the flower if block that it contains something called a gizmo rotation. I will explain what this is later, in ApplyIFTexture().

Now on to actually apply the texture…

```
outMap = Bitmaptexture  fileName: (texturePath)
objbody.material = standardMaterial diffuseMap:(outMap) showInViewport:true;
```

We need to add the texture we want to to a "diffuseMap" which is a Bitmaptexture in our case. We then make the objects material as that diffuse map. Since a proper UVW modifier was previously added, and showInViewport is set to true, than at this point the objects should have a visible texture.

```
textureOutput = #(textureMainPath,map,objbody,outMap)
return textureOutput
```

The object, now with a texture applied is output by the function along with any folder paths it used or textures it applied.

Now lets look at ApplyIFTexture().

It opens up by using the first parameter of the function to determine what kind of image it is projecting on to the body and to set imgVal and pathLoc.

```
if typeOfImage == "Icon" then (
        imgVal = iconVal()
        pathLoc = "Icons\\"
) else (
        imgVal = fractalVal()
        pathLoc = "Fractals\\"
)
```

Next, it enters and if, else block. The conditions are

```
if imgVal != "No Icon" or imgVal != "No Fractal" then (
```

Since "No Icon" and "No Fractal" are values of their respective dimensions, the program must take this into account. If either of these is the case (the else block is executed), it will default to the default grey colour the user has chosen,

```
        iconPath = assetPath + "Patterns and Colours\\" + greyColour + ".png"
        mat = Bitmaptexture filename:(iconPath)
        body.material = standardMaterial diffuseMap:(mat)
```

If both are not true, then the image location is first opened up and then set at the body material.

```
        iconPath = assetPath + pathLoc + greyColour + "\\" + imgVal
```

```
mat = Bitmaptexture filename:(iconPath)
body.material = standardMaterial diffuseMap:(mat)
```

After, the program initializes the map and sets it to the scale size I have predetermined. We also enter modify mode and select the body, which allows us to more precise manipulations to the map. You can find all of the defaults I have defined in loadInConstantsAndDefaults().

```
myMap = uvwmap()
myMap.length = SCALE_SIZE
myMap.width = SCALE_SIZE
myMap.height = SCALE_SIZE

max modify mode
select body
```

Next, we determine more predetermined variables; however, some bodies need different settings so, there is a list of if statements that cover for it. We determine the variables hrApp and angleApp. The former is short for “Height Ratio Application”; it is what determines how to center the image on the z axis of the body. The latter is short for “Angle Application”; it is what determines how much to rotate the object. Again, you can find all these values defined in loadInConstantsAndDefaults().

```
hrApp = DEFAULT_HR; angleApp = DEFAULT_ANGLES
if bodyVal() == "Cubic" then
        angleApp = CUBIC_ANGLES
else if bodyVal() == "Pyrimidal" then
        hrApp = PYRIMIDAL_HR
else if bodyVal() == "Compressed_Oblong" then
        hrApp = CO_HR
```

After, the program simply adds the map to the body, accesses the gizmo of the map to apply the transformations and the body is deselected. Note that **the map must be applied to the body before accessing the gizmo**.

```
addmodifier body myMap
myMap.gizmo.rotation = angleApp
if not isHair then
        myMap.gizmo.position = [0,0,bodyHeight/hrApp]
deselect body
```

After applying the texture in makeQuaddle(), the head is added. It is pretty much the same process as making the body; however, there are some small positioning and sizing transformations done.

```
if objHasHead() do (

        if ( headVal()  != "No Head" ) then (
```

As usual, the program checks to see if there is a head dimensions and, if there is a head dimension, if it’s null. If the value is “No Head”, it will do nothing.

```
if applyAspectRatioToHead == false then
                                ratio = 0

headResult = MakeBody( headVal() )
headBody = headResult[1]

isHead = true
textureResult = ApplyColouredTexture(headBody)
```

Here, we simply see if the user wants to apply the ratio to the head, then make the head and apply the texture; it is just like the body.

```
scale headBody [ 0.5,0.5,0.5]

hBodyZMax = headBody.max[3]

headBody.pos = [0, 0 , (bodyZMax + hBodyZMax) - 0.1]
```

The program then scales the head and positions the head correctly on the top of the body, using the values earlier that it derived from the body (bodyZMax).

```
if objHasSmoothness() and applySmoothnessToHead then (

        if smoothnessVal() != "Smooth" then (

        smoothOut = ManipulateSurfaceTexture(headBody)(headVal())
        headBody = smoothOut[1]

        )
)
```

Finally, it applies the surface texture to the head depending on if is a dimension and if the user specifies in Preferences.txt.

The final part of the object creation is the arms and beaks. These two dimensions share one function since they are the same object, just located in another place on the quaddle. Here are the blocks that determine them in makeQuaddle():

```
if ( objHasBeak() and headVal() != "No Head" and beakVal() != "No Beak") do (
        isBeak = true
        beakResult = makeArmsOrBeak()
        qArray = beakResult[2]
)

--ARM GENERATION
        --NOTE: only Arm Anlge is checked because the program
        if objHasArmCount() and armCountVal() != "0" do (
```

```
            isBeak = false
            armResult = MakeArmsOrBeak()
            listOfArms = armResult[1]
            qArray = armResult[2]

    )
```

I include both here because it is the same structure you have seen throughout the tutorial. The only difference in these if blocks is the boolean isBeak that is passed into makeArmsOrBeak() is set appropriately right before the function is called. With these in place, lets take a look at makeArmsorBeak().

```
    if ( isBeak ) then (

            if objHasBeak() then
                    armEnds = beakVal()
            else
                    armEnds = "Flat"

            if objHasBeakAngle() then
                    arms = beakAngleVal()
            else
                    arms = "Straight"

    ) else (

            if objHasArmAngle() then
                    arms = armAngleVal()
            else
                    arms = "Straight"

            if objHasArmEnds() then
                    armEnds = armEndsVal()
            else
                    armEnds = "Flat"
```

The opening code simply determines if the program is dealing with an arm or beak, and if so, what values to use in generating them. Then inside the else block, where the code for the arms is, it executes:

```
    if objHasArmCount() then (

            if  armCountVal() == "1L" or armCountVal() == "1R" then (

            armString = armCountVal()
            if armString[2] == "L" then
                    genLeft = true
```

```
else
        genLeft = false

numberOfArms = armString[1] as integer

)
else
        numberOfArms = armCountVal() as integer
```

This segment of code handles the numberOfArms. The first section signals to the program where to generate a single arm and the second simply get the arm count as an integer.

```
if (keepIandFVisible and numberOfArms > 3) and ( objHasFractal()  or  objHasIcon() ) then
        numberOfArms = 2
```

Then it handles the preference if the user wants to keep fractals and icons visible.

```
if (arms == "Spherical") or (armEnds == "Spherical") then (listOfArms =
MakeSphericalArms(objName)(numberOfArms)(objbody)(body))
else (…
```

This Script includes the code for spherical arms, **but it has not been implemented yet**. This calls a custom script called "**Script_SphericalArms.ms**" instead of the "**Script_AlternativeArms.ms**" that's used to make all the other conical/cyndrical arms.

NOTE: Putting "*Spherical"* as any one of the objVals, that correspond to any of Arm related feature dimensions (i.e. "Arm_Ends" or "Arm_Angle" in "dims") will negate any other features such as bluntness or angle.

```
--...if the arms are not spherical...
else (
        if (arms == "Bent_Down") then (bendAngle = 45)
        else if (arms == "Bent_Up") then (bendAngle = -45)
        else if (arms == "Straight") then (bendAngle = 0)
        else (bendAngle = 0);
```

If arms are specified as bent downward then they are given +45 degree angle and if they are specified as downward then they are given -45 degree angle. Otherwise angle is set to 0 as default

```
if (armEnds == "Blunt") then (
        endradius = 2 ; startradius = 6;
)
else if (armEnds == "Pointed") then (
        endradius = 0; startradius = 6;
)
else if (armEnds == "Flared") then (
        endradius = 5; startradius = 2;
)
else if (armEnds == "Flat") then (
        endradius = 4; startradius = 4;
)
else (endradius = 4; startradius = 4;)
```

Most arms are made as cones, which have their radius's specified by two different values. Thus if we wanted "Pointed" arms like below then we would need a large initial radius narrowing into a radius of essentially 0 (a point) (see object below as an example). If the arm ends are not specified then the arms are set to be standard cylinders by default.

```
listOfArms = MakeAlternativeArms(bendAngle)(startradius)(endradius)(objName)(numberOfArms)(genLeft);
```

MakAlternativeArms() is the function that contains the script **Script_AlternativeArms.ms** to make the arms. This script is described in detail further below.
In your scripts folder, you may also find a script called **Script_DefaultArms.ms**. Despite the name, none of the arms are made using this script anymore however is included in case anyone wants to use them.

This concludes making the quaddle, then pictures and be taken with TakePics().

As previously mentioned, this script takes in two input floats;

1) angle above the object from which the photo is taken
2) 2) Distance to the object

```
qqArray.pivot = [0,0,0]
```

qqArray is the complete object body (with arms attached to main body). The pivot is set to zero, so that when we want to take photos of the object from different angles, we can rotate the object in the way we want.

```
if headVal() == "No Head" or not objHasHead() then
        targetPos = 0
else
        targetPos = (-bodyZMin + hBodyZMax) / 3.5
```

targetPos is the location in which the camera rotates around. Since the height varies from quaddle to quaddle depending of the head, I use this small formula to calculate. Note, the program uses the values derived from body and head generation.

```
circ = circle()
circ.radius = abs (cos(anglevalue) * camdistance)
```

```
circ.pos = [0,0,(sin(anglevalue) * camdistance)]
```

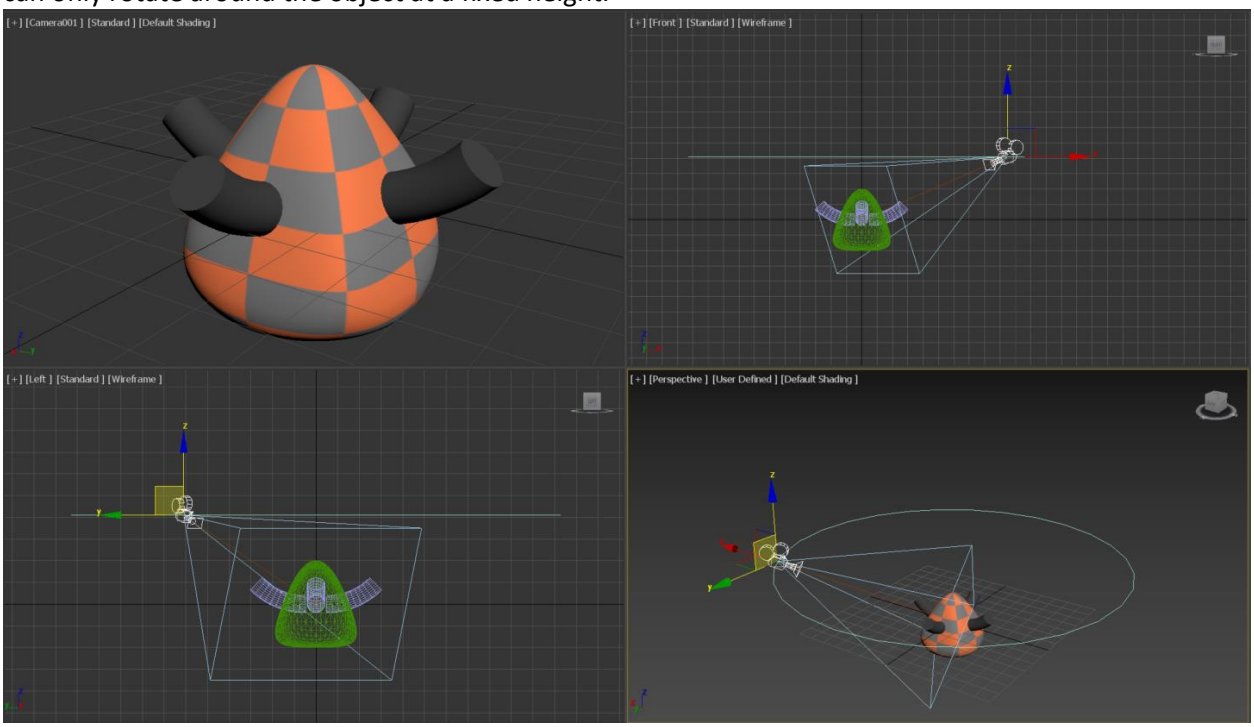
We create a circle that the camera will be bound to and ultimately denotes the camera's position. Using trig we can calculate the camera's x and z coordinates, for the specificied angle and distance we want.

```
cam = freeCamera()
cam.type = #target
cam.target.pos = [0,0,0]
```

We create a camera called cam and have it's eyes centred on the object (cam.target.pos = [0,0,0]).

```
pc = path_constraint()
pc.path = circ
cam.position.controller = pc
```

A path constraint basically means that the camera's position/movement is only limited to the lines defined to specific lines. In this case, the path constraint is set to the circle which means that the camera can only rotate around the object at a fixed height.



^This is what the camera would look like in the viewport

```
- -since camera is locked to path constraint of the circle, rotating circle will move camera
viewport.setType #view_camera
anglestring = anglevalue as string
fileNameForPic =  fileNameForPic + "pictures\\" + anglestring
makeDir fileNameForPic
- -smoothhighlights is a render setting
viewport.SetRenderLevel #smoothhighlights
picturelocation = fileNameForPic + "\\" + anglestring + "_0deg_rotation_" + picname + ".jpg"
render outputFile: (picturelocation) vfb: off outputSize: [1280,720]
picturelocation = fileNameForPic + "\\" + anglestring + "_0deg_rotation_" + picname + ".png"
render outputFile: (picturelocation) vfb: off outputSize: [1280,720]
```

```
- -rotate camera and take photos again
rotate qqArray (eulerangles 0 0 45 )
picturelocation = fileNameForPic + "\\" + anglestring + "_45deg_rotation_" + picname + ".jpg"
render outputFile: (picture location) vfb: off outputSize: [1280,720]
picturelocation = fileNameForPic + "\\" + anglestring + "_45deg_rotation_" + picname + ".png"
render outputFile: (picturelocation) vfb: off outputSize: [1280,720]
rotate qqArray  (eulerangles 0 0 -45)
```

Object is rendered to a jpg and png in both its native rotation and then rotated 45 degrees for another photo. The object is then rotated back to it's original orientation.

```
delete circ
delete cam
```

Circle path, and camera is deleted at the end of the function.

After the picture is taken, the program exports the quaddle in a fbx, or 3d file. It calls ExportFBX() which contains the code.

```
select Obj
fileDir = (ExportPath + "fbxFiles\\")
makeDir fileDir
completeDir = fileDir + objName + ".fbx"
--exporting textures along with the object
FBXExporterSetParam "EmbedTextures" true
exportFile completeDir  #noPrompt selectedOnly:true using:FBXEXP
```

To export the object, the program must select the object, build the file location, and finally call the exportFile command. Pretty simple.

After the program completes this, it deletes the object, then updates objVals and does the whole process again. Hopefully this helped you! Have fun Quaddling!!

Other Helpful Information
Miscellaneous functions from **Helper_Functions.ms**

The rest of the functions that have not been covered are the following:

loadInPathsFilesAndPreferences()
loadPathSetting()
loadOtherPreferences()
loadInConstantsAndDefaults()
getCurrentDirectory()
getAssetPath()

All these functions construct paths based on generatorPath, the path the user enters. Consequently, they all simply build strings and access those strings. They also get information from the preferences file. The general structure is as follows:

```
seek pref 0
skipToString pref "Export fbxs:"
```

```
tempString = readLine pref
tempString = trimLeft tempString
tempString = toLower tempString
if(tempString == "on" or tempString == "yes") then
	userWantsFBX = true
else
	userWantsFBX = false
```

First it resets the position of the scanner in **Preferences.txt.** Then it skips to the string of the setting that it wants to load. It reads in the setting trimming it and making it lower case. It then sets the given setting appropriately.