# **STL Visualizer for KneeBones3Dify**

# Code Documentation

# Functional requirements

**STL Loading**

The app needs to have everything needed to load an STL model into memory

**VR Visualization**

The app needs to make use of a VR framework that connects with physical hardware, so that the use r may be able to see the STL Model within it

**Model movement control**

The user should be able to move the model around using VR controllers

**Menu visualization**

The user should be able to switch between makinga menu visible or not visible with the touch of a button

**Re-rendering**

If the app is used in conjunction with a specific segmentation software, the user should be able to tell the segmentation software to re-render through the menu, also being able to use It to tweak re-rendering parameters

# The StlVisualizer.py file

This is the main file that everything else is loaded and executed from.

It is worth noting its usage of the OpenVrGlRenderer class, which is a class that comes from the pyOpenVR framework that actually does the rendering of everything we see on the VR visualization.

This program appends three objects onto it, one is the TrackedDevicesActor, which shows us the controllers as we hold them while we are in the VR, and the other two are ThreeDKnee, which handles the rendering of the STL model for a knee and MenuScreen, which does the same for the menu screen.

All objects appened onto the renderer have three methods that the renderer calls:

init\_gl()

display\_gl()

dispose\_gl()

These are necessary in any new object you may wish to append directly to the renderer as well.

There is an Instance of a controlInputModule that is defined here, and shared among the ThreeDKnee and MenuScreen instances. This class handles everything related to control, both for the 3D model and the menu screen.

Lastly, this program can either be executed in stand-alone mode (so that aspects that aren’t related to having the segmentation program re- render the 3D model can be tested on their own) or in non standalone mode, in which case the program assumes that we are using it in conjuction with KneeBones3Dify.

# The ThreeDKnee.py file

Global variables:

loader = STLloader()

model =loader.model

ControlMod

Classes:

class ThreeDKnee

ThreeDKnee class attributes

self.shader

self.vao

self.num\_vert

self.pose\_array

self.stl\_location

self.ssbo

ThreeDKnee class methods

\_\_init\_\_(self,pose\_array,control\_mod,stl\_location= None)

find\_headset\_pose(self)

reload\_model(self,stl\_location)

init\_gl(self)

display\_gl(self, modelview, projection)

dispose\_gl(self)

This is the file that handles the visualization of the knee model.

The model is loaded at the beginning of the file, using an STLloader object which is just a class that uses assimp\_py, a python port of Open-Asset-Importer-Library, to load a 3D model into an easily manageable data structure.

A few class variables are defined and assigned in the \_\_init\_\_ method, including pose\_array, which is just an array of poses for VR devices, and the InputControlModule instance controlMod.

If we aren’t in standalone mode, it loads the model from the path that was passed to it by the segmentation software, otherwise it loads from a default path.

In the initialization routine, there’s a vertex shader and a fragment shader, which will be explained in depth when we go over the display\_gl() routine.

Once the shaders are compiled and put into a shader program, we get the model’s vertices and normals from the assimp\_py data structure and we convert them into numpy arrays.

The vertices are normalized, due to the fact that OpenVR actually counts in meters, so something like (0,1,1) is a meter on the positive Y axis and a meter on the positive Z axis, for example.

So if your vertices are normalized you get an object that is on the scale of a cubic meter in volume.

There is the same exact amount of vertices and normals in the models generated by KneeBones3Dify so we just interpolate both in one array which also changes their coordinate system from a three dimensional one (x,y,z) into a four dimensional one (x,y,z,w).

After we have our array we load it into the vertex shader using a Shader Storage Buffer Object(SSBO).

Now that the initialization is done, let’s explore the most important function in this file, display\_gl().

The very first instruction of this function is an assignment that gets two values from a function called control(). We’ll go over this function in detail when we get to the controlModule.py file, what is important to know here is that it essentially outputs a 3 element vector, TranslationDelta, that represents how much the model must be translated from it’s default position in that instant, and a 4x4 matrix, RotationMatrix, that represents how much it must be rotated.

The last important detail is that those two variables are actually stored within ControlMod, and control() only applies a modification or passes them as stored.

After that there is a check for whether the segmentation program has finished rerendering a model, that checks if the last modified timestamp on a certain file has changed. If it has, the model is reloaded (note, as of this writing, the program assumes that the model’s file path has stayed unchanged since it was passed to it in its’ initialization), and the menu, if visible,is automatically taken down.

After this check, we use the shader program and pass a few variables to the vertex shader,using GLSL uniforms. We send a few of the variables that we have already talked about plus the position of the VR headset, which is used in the fragment shader to compute the light.

There is a call to glDrawArrays() which uses self.num\_vert, which is simply a variable that is a class member of ThreeDKnee, where the number of vertices (taken straight from the assimp\_py data structure that is written after the model is loaded) are written in, both on the first loading of the model and on subsequent rerenders.

Every vertex is elaborated by the shaders that we have already started to talk about, so now is the right time to go more in depth about what the shaders actually do in this code.

**The ThreeDKnee.py Shaders**

The vertex shader does the following operations for every single vertex:

After retrieving the uniform variables’ values, and defining a bunch of variables that will be outputted to the fragment shader to compute lighting, a default translation matrix and a default rotation matrix are defined.

The function of these matrices is to translate and rotate the model from the origin in front of the user’s eyes, making the kneecap visible   
  
Along with these there are UserControlRotationMatrix and a UserControlTranslationMatrix (which is actually computed from the uniform input variable translateDelta), which are essentially a representation of where and how the user has rotated and translated the model so far. We will also talk about these two matrices when we go over the ControlModule.py file.

The 4 matrices we just talked about are then all multiplied together and put into the Modified\_Model matrix

vertex positions and normals are then extracted from the array that we sent to the shader from the SSBO, the vertex positions are multiplied with the Modified\_Model to apply the transpositions that we already talked about, and the normal are multiplied by Modified\_Model’s transpose of its inverse.

All of the aforementioned output variables for the fragment shader are computed now as well, before they get passed to it, and then the actual final vertex position is computed, using the Projection and ModelView that the OpenVrGlRenderer had passed to display\_gl().

The fragment shader handles lighting.

Most of the code was taken from [this tutorial,](https://github.com/opengl-tutorials/ogl/blob/master/tutorial08_basic_shading/StandardShading.fragmentshader) and was then modified by eliminating the calculations relative to the specular element of the color.

At first the normal that is relative to the current fragment is normalized, along with the light direction, then the cosine of the angle between them is calculated through the use of a vector dot product, which is them clamped between 0 and 1.

We then have a few variables, MaterialDiffuseColor, MaterialAmbientColor, LightColor and LightPower, who are computed into color, which is the final output variable of the fragment shader.

# The MenuScreen.py file

Global variables:

controlMod

menu\_width

texControl

texture\_proc\_done

texture\_proc\_generate

ProcessOver

texture\_is\_loading

shared\_queue

texture\_modifying\_process

Global procedures:

tex\_modify\_proc\_routine(shared\_queue, texture\_proc\_done, texture\_proc\_generate,processOver,menu\_width,texture\_is\_loading)

Classes:

class MenuScreen

MenuScreen class attributes:

self.shader

self.vao

self.menu\_texture

self.tex

self.width

self.height

MenuScreen class methods:

init\_gl(self)

display\_gl(self, modelview, projection)

dispose\_gl(self)

This file handles the visualization of the menu.

Before we even get into the MenuScreen class, there are a bunch of global variables, the most important of which are all related to a subprocess that is used to modify textures asynchronously.

Multiple variables of the multiprocessing.Value type are also defined here, this specific type of variable, can have shared scope between processes. In this specific case, we pass these variables to the texture modifying routine that will be put on a subprocess, enabling both the subprocess and the main process to have them in their scopes at the same time, so that the modifications to these variables’ values made by one of them will also be seen by the other.

We also have a multiprocessing.Queue that works similarly, however it’s shared scope is used so that one process can put a value in it that the other process can receive.

The actual texture modifying subroutine will be explained in depth later.

Going into the initialization of the MenuScreen class, we see that the width the menu is initialized from a global variable, and the height is calculated from that. This width will also be passed to the texture generation, which will base the size of everything from it.

Lastly, we have ControlMod, an InputControlModule instance that is actually passed upon the initialization of the the MenuScreen class.

Going into init\_gl(), we see that this time vertex coordinates are defined directly inside of the vertex shader, along with texture coordinates that are then passed to the fragment shader, which draws the texture from a 2D texture sampler.

After a little bit of initialization code, a texture name is defined and an image is generated, using an instance of textureControl, and then glTexImage2D specifies a texture image using the generated image and binding it to the previously generated texture name.

A mipmap for said texture is then generated and a few parameters are then specified.

Lastly, the texture modifying subprocess is started, at this moment it isn’t modifying anything because the flags that make it do that have not been set. However it is initialized and then enters it’s waiting cycle.

After the initialization,and going into display\_gl() we see a call to a function from ControlMod, that has two outputs, status and selected\_param.

This is esentially just a call to a function that interacts with the menu following the controller inputs,just like it happens in ThreeDKnee.py.

The two outputs are a dictionary of values representing the menu parameters value, and a bunch of useful flags.

The menu first checks whether it is enabled ,and if it Is, it renders as a flat rectangle with a texture on it.

The texture is changed each time the menu is interacted with while visible.

Since this is a bit of a costly operation that would make the visualization lag if it was done synchronously, it is done asynchronously, using the subprocess with the texture modifying function in it that was briefly talked about before.

The interaction between the texture modifying subprocess and display\_gl()works like this:

As previously said,the subprocess is started at the end of initialization and some setup is done before it enters its’ permanent cycle.

The setup consists of creating an instance of TextureControl that is used to modify the textures in the subprocess, and generating a base default image with it, so that in its’ memory it already looks correct when it applies the modifications that display\_gl()is going to ask for.

Once display\_gl()starts and checks that the menu is actually enabled, it checks whether one of the flags from status has been set true AND that the subprocess is not already modifying a texture. If both o those conditions are true, it sends both status and selected\_param in to the shared queue and sets a flag that lets the subprocess know it’s time to modify the texture.

Going back into the texture modifying subroutine(tex\_modify\_proc\_routine), the values that were sent are extracted from the queue and a flag that signals the fact that the new texture is being generated is set.

If we know that the “render” button has ben pushed in the menu, a separated modification to the texture is made in order to change the text of that button to “Re-rendering,please wait”, before we call the standard modification routine, modifyTexture().

This last function is called when we change what parameter we are selecting in the menu, or when we change the selected parameter’s value. Once the texture is modified, we put the new texture in the shared queue and signal that the subprocess is done, but setting a specific flag.

When display\_gl() sees this when it checks for this specific flag, the texture image is retrieved from the queue and loaded into the openGL texture. after which texture\_is\_loading and texture\_proc\_done are unset.

Last thing to note, the subprocess detects when the rerendering is finished (through the use of one of the flags included in the status instance that gets passed to it ) and modifies and reloads the texture automatically.

# The MenuStatus.py file

Classes:

class MenuStatus

MenuStatus class attributes:

self.menu\_dict

self.menu\_dict['intensity\_threshold']

self.menu\_dict['convex\_hull\_dilation']

self.menu\_dict['final\_closing']

self.menu\_dict['protrusion\_removal']

self.menu\_dict['final\_dilation']

self.menu\_dict['enabled']

self.menu\_dict['modified']

self.menu\_dict['re-rendering']

self.param\_array= ('intensity\_threshold','convex\_hull\_dilation','final\_closing','protrusion\_removal','final\_dilation','re-render')

self.selected\_param

MenuStatus class methods:

getSelectedParam(self)

selectNextParam(self)

selectPrevParam(self)

augmentSelectedParam(self)

diminishSelectedParam(self)

MenuStatus is an auxiliary class that is used by MenuScreen and controlInputModule.

It offers a dictionary of parameters that are modifiable and selectable through its methods, an array of selectable menu parameters, and three flags that are used by controlInputModule.

self.menu\_dict['enabled'] signals whether the menu is visible right now.

self.menu\_dict['modified'] signals whether the menu’s visualization needs to be changed due to a change in parameter selection or in the selected parameter’s value.

self.menu\_dict['re-rendering'] signals whether a rerendering of the knee 3D model is happening or not, and is also used to change the menu’s “render” parameter text to reflect what is happening, as well as to impede the user from starting a rerendering if another one is already underway

# The TextureControl.py file

Classes:

class TextureControl

TextureControl class attributes:

self.width

self.height=math.ceil(self.width/1.5)

self.fontsize = math.ceil(self.height/15)

self.tex

self.fnt

self.draw

black\_RGB=(0,0,0)

white\_RGB=(255,255,255)

green\_RGB=(0,255,0)

black\_grayscale

white\_grayscale

grey\_grayscale

self.black

self.white

self.green

self.default\_fill\_color

self.highlighted\_fill\_color

self.first\_column\_x

self.second\_column\_x

self.parameterCoordinates={

'intensity\_threshold':(self.first\_column\_x,math.ceil(self.height/5.8)),

'convex\_hull\_dilation':(self.first\_column\_x,math.ceil(self.height/2.3)),

'final\_closing':(self.first\_column\_x,math.ceil(self.height/1.45)),

'protrusion\_removal':(self.second\_column\_x,math.ceil(self.height/5.8)),

'final\_dilation':(self.second\_column\_x,math.ceil(self.height/2.3)),

're-render':(self.second\_column\_x,math.ceil(self.height/1.45))

}

self.parameterTextDescriptions={

'intensity\_threshold':,

'convex\_hull\_dilation':,

'final\_closing':,

'protrusion\_removal':,

'final\_dilation':,

're-render':

}

self.parameterValueTextDistance

self.prevParameter

TextureControl class methods:

generateTexture(self,menu\_dict,selected\_param)

modifyTexture(self,menu\_dict,selected\_param)

flagRerendering(self)

unflagRerendering(self)

TextureControl is another auxiliary class that does all of the image generation for the menu texture.

In the \_\_init\_\_ method, the base image is first generated and saved, along with a bunch of sizes and coordinates that are all calculated from the width that is given to the class when it is instantiated. Also, we generate an object that draws on the image, and a font.

When generateTexture() is called, it adds the default values and text descriptions for every parameter to the base image, along with the logo for KneeBones3Dify and the text “Parameter menu” on top of everything else. It is recommended that this method is only called once per instance of TextureControl.

When modifyTexture() is called, it highlights a selected parameter, de-highlighting the previously selected parameter,if it’s different from the one that is currently selected, or modifies the visualized value of a parameter, by first deleting the old value from the image and then writing the new one.

flagRerendering() and unflagRerendering() modify the text for the “Re-render” option.

# The controlModule.py file

Classes:

class controlInputModule

controlInputModule class attributes:

self.rotationAmount

self.translationDelta

self.rotationAxis

self.left\_controller\_id

self.right\_controller\_id

self.last\_left\_angle

self.last\_right\_angle

self.left\_isdragging

self.right\_isdragging

self.last\_left\_pos

self.last\_right\_pos

self.left\_pause\_pressed

self.right\_pause\_pressed

self.pose\_array

self.lockedRotation

self.rotationMatrix

self.controllerRotationMatrix

self.trackpadRotationMatrix

self.controllerRotationMatrix\_static

self.controllerRotationMatrix\_reset

self.paused

self.re\_rendering

self.temp\_file\_modified

self.dataset\_location

self.temp\_output\_file\_location

self.menuStatus

self.left\_pause\_pressed

self.right\_pause\_pressed

self.right\_menu\_enabled

self.left\_menu\_enabled

self.left\_trackpad\_pressed

self.right\_trackpad\_pressed

self.standalone\_mode

controlInputModule class methods:

getDelta(self,pos1,pos2)

getFinalDelta(self,pos1,pos2)

get\_controller\_ids(self)

get\_controller\_pos(self,controller\_id)

get\_controller\_rotation(self,controller\_id)

get\_rotation\_matrix(self,axis, angle)

packParametersIntoOutputString(self,menu\_dict)

from\_controller\_state\_to\_dict(self,pControllerState)

control(self)

menuControl(self)

automaticUnpause(self)

refresh\_menu\_after\_model\_reloading(self)

controlInputModule is a class that handles inputs from VR controllers and the interpretation of those inputs into outputs for MenuScreen and ThreeDKnee.

The two main points of interest are control()and menuControl(),because every other function in this class, except for automaticUnpause() and refresh\_menu\_after\_model\_reloading(), is used as an helper function for them, and all of the variables initialized and defined in \_\_init\_\_() are used in one way or another here.

control() deals with the control of the 3D model .

Before even getting into the specifics of what goes on in this function, it is better to understand a few things about the general design concept of it:

FIrst off , it is very useful to remember that control()is called every instant that the bone renders.

There are variables that store whether in the instant that a specific call of control() was made the inputs of both controllers separatedly are happening at all, and what the equivalent movement values for every single input are.

Once the checking is done separatedly, the values are then either added together, if they’re scalar values, or multiplied if they’re matrices, there is only one exception to this and it’s the rotation matrix for the controller drag function. That functionality is locked only to the one controller that is doing the drag in that moment.

Once the instant checking is done, there other variables, that are some of the class variables initialized and defined in \_\_init\_\_(), that store the result of every movement from the start of the program.

The local variables are then added or multiplied to these stored class variables, which then store the final result of these operations.

The first few lines of control() define some of these local variables (from localRotationAmount to rightTranslationDelta), then a localRotationAxis is defined. This is used as a parameter, along with localRotationAmount, for the calculation of the trackpadRotationMatrix that stores how much the bone should move if the user has touched (without pressing) the trackpad.

localRotationAxis is a vector representation of the Y axis if the trackpad on the left controller is touched, and gets changed to a representation of the Z axis if the trackpad on the left controller is touched.

We then have modified, which is a flag that if left unset is used to reset two values related to calculation of movement values for trackpad rotation, self.last\_left\_angle and self.last\_right\_angle. (The function that these values have will be explained a bit more in depth once we go into the code that converts controller inputs into movement values)

Once these variables are defined, there is a check for the right and left controller, and then we get into the input checking code.

Most of this is the same for both controllers. Differences will be noted when present.

Once we enter the input checking code for one controller, the controller state at that instant is converted into a Python dictionary using from\_controller\_state\_to\_dict().

This makes input checking very easy and quick.

The first thing that is checked is whether the menu button was just pressed this instant. If it was, we pause or unpause movement for the bone, depending on whether it already was or not.

Also, in order to not have the bone collide with the menu when it is visible, in this same instant it is also moved, or moved back if we have unpaused.

All of this only applies if the user has pressed the pause button in this instant, because then a flag is set so that all of this is not done if the pause button was already pressed and has yet to be released. The flag is then unset when the menu button is released.

The code then checks for a trackpad press, which, if found, translates the bone on the X plane and the Z plane.

If the user hasn’t pressed the trackpad but only touched it we enter the code for trackpad rotation, where self.last\_left\_angle and self.last\_right\_angle play their role.

Those two values essentially do this: if they we set to 0, they get set to the last angle that is calculated from arcosine of the Y position of the user’s finger on the trackpad of the left/right controller, without actually setting any movement value for that instant.

The next instant that control() is called, if the user is still touching the same controller trackpad that they were touching before, a movement value is calculated by subtracting the value of the arcosine of the current Y position of the user’s finger with the one that was just stored, giving us a localRotationAmount. The arcosine of the current Y position of the user’s finger is then saved as the new “last angle” value.

Once we have a localRotationAmount, we get a rotation matrix by inputting it into get\_rotation\_matrix() along with localRotationAngle.

This matrix is then immediately multiplied with the trackpad rotation matrix that was already stored.

The last and most important check is for the pressing of the trigger, which is used to implement the dragging of the bone.

The first thing to note about drag is that it can only be done by one controller at a time, unlike trackpad rotation.  
  
When we enter the code for dragging we first check whether we have been dragging for any amount of time.  
if we haven’t we set a flag that says we are now dragging (self.left\_isdragging or self.right\_isdragging), we save a the current position (self.last\_left\_pos or self.last\_right\_pos) and inverse of the rotation matrix for the controller (self.controllerRotationMatrix\_reset).

If we have already been dragging for at least an instant, the current controller rotation matrix (self.controllerRotationMatrix), is multiplied it for the matrix that we had saved before.

Then, another matrix (self.controllerRotationMatrix\_static , whose function we will explain shortly) is multiplied for the result .

Besides rotation we also obtain a delta for translation, by comparing the current controller position, and the last saved one.

Once we are done dragging, the flag relative to our controller’s dragging state is unset, unlocking the drag function for both controllers, and the current controller rotation is saved into self.controllerRotationMatrix\_static.

So, this whole section essentially functions like this.

By saving an inverse of the rotation of the controller and then multiplying it by the current rotations after that we have the differential in rotation between the two.

self.controllerRotationMatrix\_static stays the same until the current drag action is finished, so that, while we are dragging, we assure that the bone is rotated only by the amount that it was before dragging plus the differential that we multiply this matrix for, since the static matrix stays the same until the drag is done, and the result of the operation that will be used in the final calculations of the general rotation matrix is saved into self.controllerRotationMatrix.

Once the input checking is done for both controllers, the final steps of this code are:

1. Getting the localTranslationDelta for this instant by adding the two deltas from the controllers and then adding it to the self.translationDelta that was already stored
2. Multiplying self.controllerRotationMatrix and self.trackpadRotationMatrix to get the final self.rotationMatrix that is stored in the class.
3. Passing the class members self.rotationMatrix and self.translationDelta to the calling function

Now that the functioning of control() has been thoroughly explained, we can go over menuControl().

Although it has the same basic structure and a lot of similarities with control(), menuControl() also has some differences and specific functions that important to understand.

The first one is that, even if it only checks for two types of inputs (the pressing of the menu button and the pressing of the trackpad), it does so always in the same way, and that is:  
  
1)If this is the first instant that we press this button, a flag is set and the action connected with that input is performed.

2)If the button is still pressed but the flag relative to it has been set, the action connected to it is not performed again.

3)Once the button is released, the flag is unset and we can now perform the action connected to it by pressing it if we so desire.

The second important difference is the output. Instead of outputting movement values, outputs the currently selected parameter and a dictionary(self.menuStatus.menu\_dict) will all the current values of the parameters and a few flags that are used for a few different purposes.

The first and simplest input check is for the menu button, and it’s pretty self explanatory, it just turns on/off the menu itself by setting a flag that the rendering routine of the MenuScreen instance that called menuControl() will check for.

The second input check checks for a track pad press and is both used for changing the selected parameter or its’ value.

When we enter this second input check, we set a flag (status['modified'])that is used to signal to the MenuScreen instance’s rendering routine that it needs to modify the menu texture, since any action that is taken when the track pad is pressed requires that to happen in order to show what change has happened.

Pressing in different areas of the trackpad has different outcomes, the upper and lower corner are for parameter selection and the left and right ones are for parameter modification.

If the press in an area that isn’t any of these ( aka the center of the trackpad), there is a last check that activates rendering if:

1)the selected parameter Is “re-render”

2)we haven’t already started a render

3) we aren’t in standalone mode

If all of those conditions are met, the parameters in the menu dictionary are packed into one string and printed onto a file, whose location is passed by KneeBones3Dify when it starts up an instance of STL Visualizer.

After STL Visualizer is started, KneeBones3Dify keeps watch of this file, and when we modify it when starting up a render from menuControl(), KneeBones3Dify reacts to this modification by reading the contents that were just printed onto the file and rerendering the STL model.

When we have written into the fie that is used to communicate, a timestamp is saved, as KneeBones3Dify will tell the visualizer that it is done re-rendering, and we will check for this using this timestamp (see the paragraphs about MenuScreen.py and ThreeDKnee.py for more).

After all the checks are done for both controllers, we send the menu status dictionary and the selected menu parameter as return values.

The last two small functions left on this file are automaticUnpause() and refresh\_menu\_after\_model\_reloading().

refresh\_menu\_after\_model\_reloading() is called by ThreeDKnee once it is done reloading the model, and simply unsets the flag that are relative to rendering and sets the “modified” flag for the menu.

If the menu is currently visible , automaticUnpause() is called so that it is automatically brought down, unsetting all flags relative to pausing and translating the model back to the positon where it was before pausing.