Assignment 3 Design Document - ArmSim

Robotic Arm Simulator and Movement Programmer

Name: Joshua Riddell
Student Number: 43947241

28 May 2015

1 Summary

Programming the complex movements of a robotic arm using direct angles and low level code is usually out of the scope of most factory operator's expertise. ArmSim allows easy movement of a complex 6DOF robotic arm and also provides a 3D representation of the arm. The 3D representation enables the user to program complex movements with ease, and allow movement programming without the need for a physical arm to be present. ArmSim also includes a sequencer module - this allows the user to quickly prototype complex movement sequences in a matter of minutes.

2 User Interface

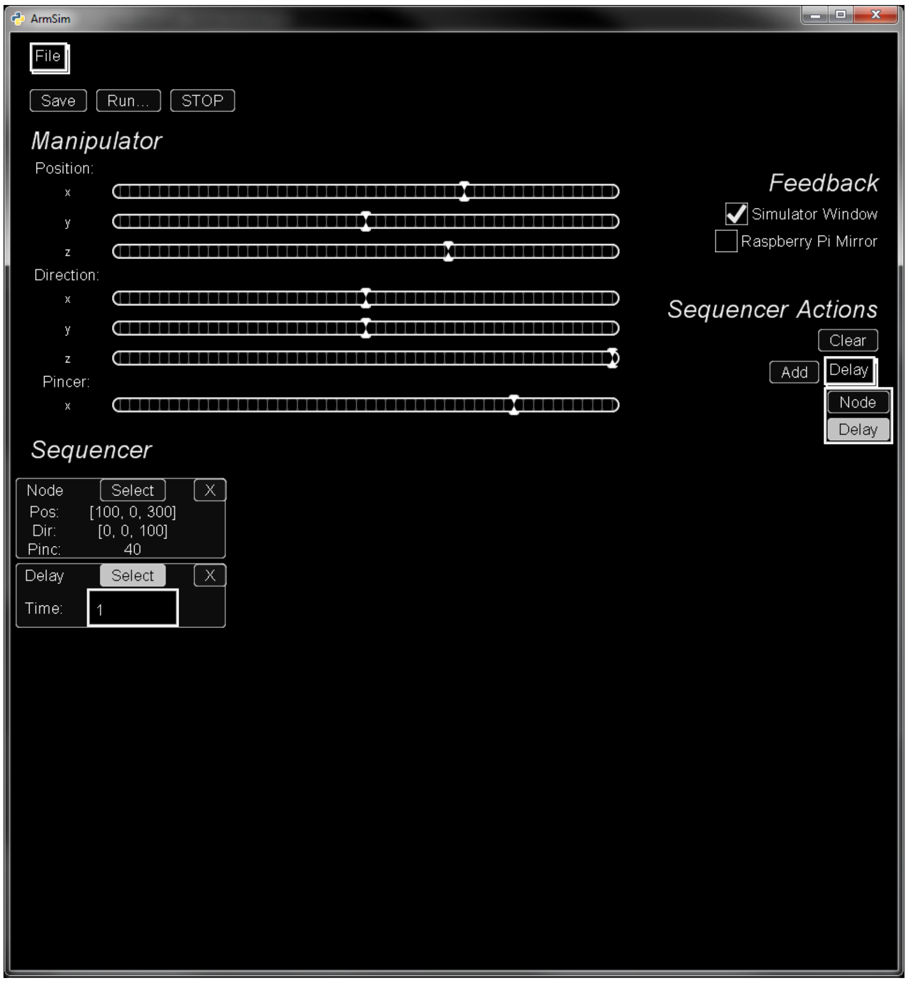
2.1 General Operation

General operation consists of:

1. adding a node using Sequencer Actions → Add Node, or adding a time delay using Sequencer Actions → Add Delay;
2. using the manipulator panel to change position, direction and pincer state of node;
3. repeating steps 1 and 2 until a full sequence is made;
4. saving the sequence to a '.seq' file using the 'Save' and 'Save As' buttons;
5. running the sequence using the 'Run' button (with feedback via a simulator or Raspberry Pi mirror).

2.2 Main Window

Depending on the currently loaded sequence, below is the appearance of the main window.



[✓] Simulator Window

Node
Pos: [100, 0, 300]
Dir: [0, 0, 100]
Pinc: 40

Delay
Time: 1

Figure 1: Main window general appearance

2.2.1 Context Menus

The **File** menu consists of:

- Open – Opens a movement sequence file
- Save – Updates the already made save file
- Save As – Allows changing of the current save file

The **quicksettings** consist of:

- Save – Updates the already made save file (same as File→Save)
- Run... →
  - Simulator – Runs the currently loaded sequence in the simulator window
  - RPi – Runs the currently loaded sequence on the Raspberry Pi
- STOP – Terminates the sequence currently being executed.

2.2.2 Manipulator Panel

The manipulator panel is used to manipulate the position and direction of the arm. Position and direction sliders control their respective coordinates of the end effector in 3D space. Figure 1 shows the coordinate system - what has become the standard for CNC machines and 3D printers is used: z axis is up, y is away from the operator, and x is perpendicular to the operator. The origin is on the bottom of the centre of the base.

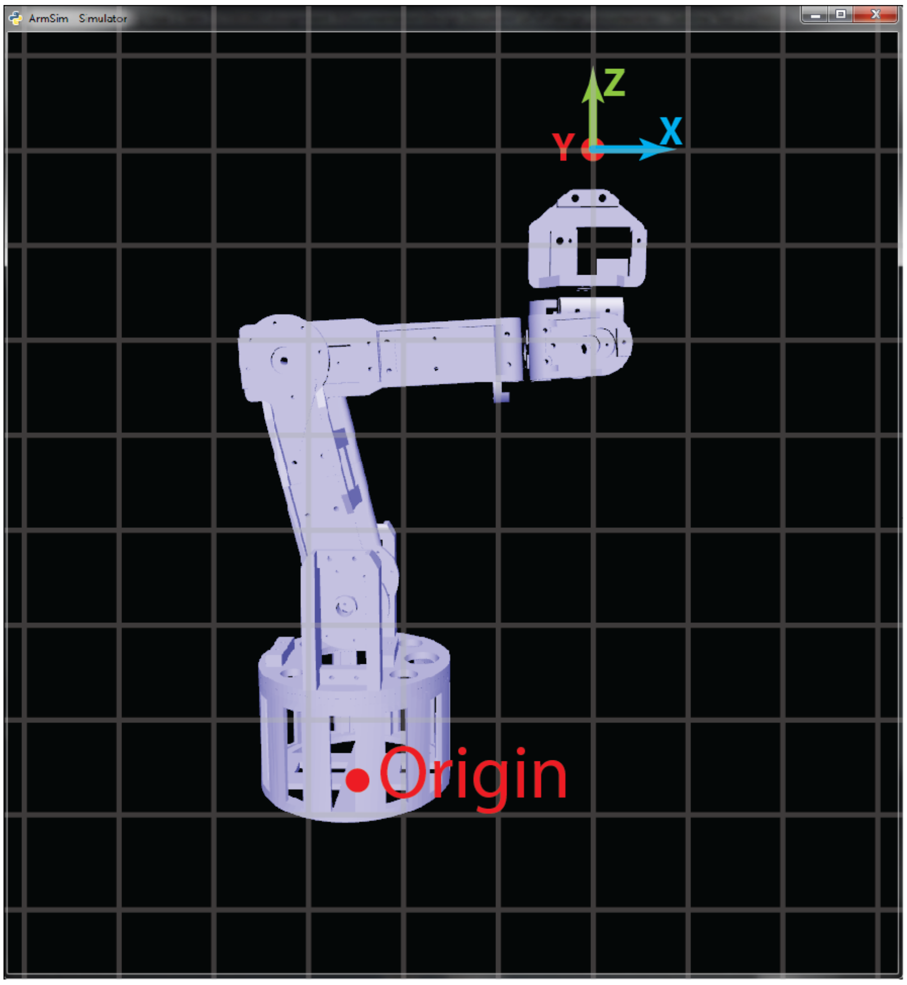


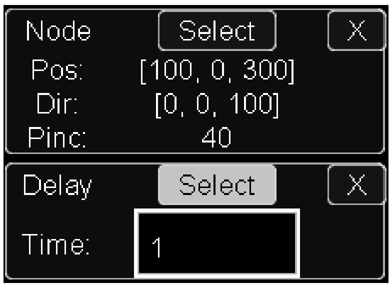
Figure 2: Axis an grid show a general indication of how the coordinate system works

2.2.3 Sequencer Panel

The sequencer panel is used to order movements. There are two types of modules: node and delay. See Figure 2 for examples of each.

Node modules contain a position, direction and pincer state for the arm. They represent a 'node' which the arm can move to. When executed, the position module will animate to their position based on the previous position the arm was in – they currently only support linear animation.

Delay modules contain a time delay. When executed they simply sleep the thread for the specified number of seconds, therefore creating a wait.



Node Select X
Pos: [100, 0, 300]
Dir: [0, 0, 100]
Pinc: 40

Delay Select X
Time: 1

Figure 3: Node module (top) and delay module (bottom)

2.2.4 Feedback Panel

The feedback panel is used to select which methods of feedback will be used. See Figure 3 for the default state of the Feedback Panel.

Simulator Window checkbox will set the VISIBILITY of the simulator window (it does not close the window and as a result there will be no performance improvements unless the window is closed). This was done so that when the visibility is set back, the response time of the window is very quick.

Raspberry Pi Mirror checkbox will initiate a connection with the RaspberryPi, if one is available. Not that the application will not continue until an RPi has been connected after the checkbox has been pressed.

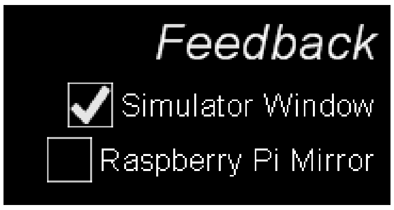


Figure 4: Feedback panel default settings

2.2.5 Sequencer Actions Panel

This contains the main actions for using the sequencer portion of ArmSim. For example of expanded state see Figure 4.

Clear clears all elements in the sequencer panel.

Add [module] adds the relevant module to the end of the current sequence.

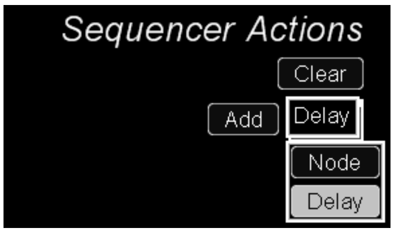


Figure 5: Sequencer actions panel expanded state

2.3 Simulation Window

The simulation window shows the current state of the robotic arm in 3D. It supports camera zoom, pan and orbit in a similar fashion to most CAD programs. The model of the arm is imported from .obj files representing each segment. Figure 6 shows the default setting of the simulator window.

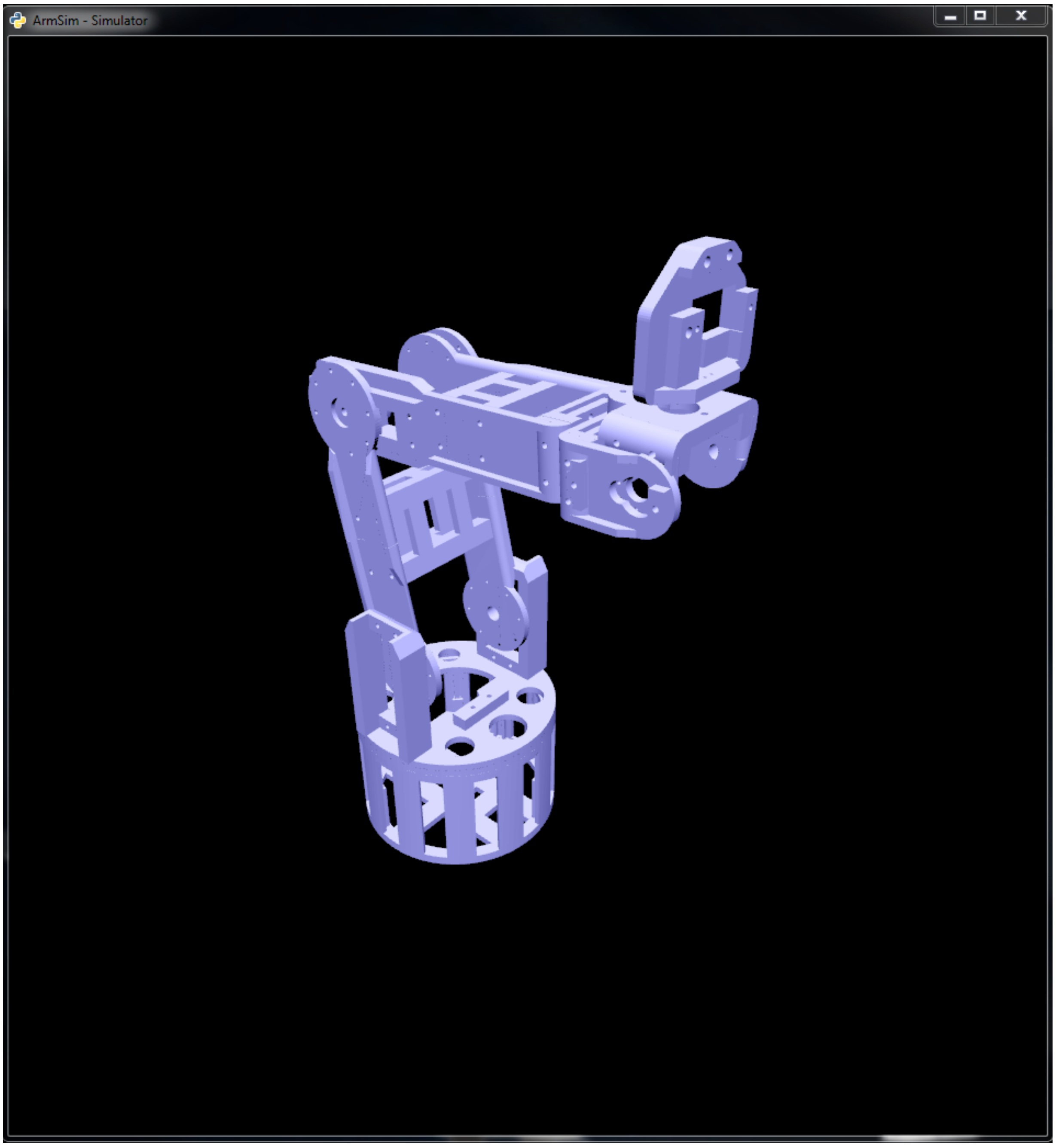


Figure 6: Default view of the simulation window

3 Support Modules

External modules used are:

Pyglet	- http://www.pyglet.org/documentation.html
pyglet_gui	- https://github.com/jorgecarleitao/pyglet-gui
pyglet_obj	- https://github.com/reidrac/pyglet-obj-batch (indcluded in package)
socket	- https://docs.python.org/3/library/socket.html
threading	- https://docs.python.org/3/library/threading.html
time	- https://docs.python.org/3/library/time.html

3.1 pyglet

A low-level OpenGL graphics and window handling module. Direct interaction with pyglet is mainly only done in the simulator window.

3.2 pyglet_gui

A basic GUI library built off pyglet. This was not an ideal choice for the GUI but was used as the simulator was developed first in pyglet, and a compatible GUI library was needed.

3.3 pyglet_obj

An .ojb importer for pyglet. This is used to import standard CADD files exported by Trimble Sketchup.

3.4 socket

Used for socket-based communication between the host computer and RaspberryPi.

3.5 threading

The transmission loop between the host and RaspberryPi, and the execution of sequences are both run on axillary threads. Threading is used to achieve this.

3.6 time

Time delays are achieved by using sleep.

4 Design

This section contains an overview of the modules included in ArmSim.

4.1 main.py

def main() Creates an instance of the sequencer window. Schedules the SequencerWindow update() function every 1/60 seconds and an update_joints() method every 1 second. Then starts the Pyglet even loop.

4.1.1 class SequencerWindow(pyglet.window.Window)

This is the main class, all others stem from here.

def __init__(self) Creates instances of buttons and sliders; the Server class and the SimulationWindow class.

def slider_cb(self, pos_slider, value, axis) Called by the position and direction slider instances. Pos_slider is a boolean which represents whether the slider is position or direction slider. Value is the passed value of the slider and axis is which axis the slider is assigned to.

def update_sliders(self, pos, direct, pincer) Called by the sequencer object, allows for the sliders to change to the new SequenceElement's values when a different one is selected.

def update_joints(self, pos=None, direct=None, update_all=None) Called at any update for the realtime displays. Updates the simulation and raspberry Pi arms if they are activated. Pos and direct represent the updated position and direction vectors, update_all is a boolean which tells the server object to send an update of all angles (if it is activated). update_all is true for example when it is called by the scheduled update function so that a full 'refresh' is performed regularly.

def on_resize(self, width, height) Called when the main window is resized. Expands the sliders and repacks the sequencer based on the new window size.

def clear_sequencer(self) Clears the sequencer of all elements.

def sequencer_add(self, ID) Adds the sequencer with type, 'ID', to the sequence by calling the add_module method of Sequencer.

def rpi_network_toggle(self, activate) Called by the 'Raspberry Pi Mirror' checkbox on the sequencer display. Activate is a boolean which tells whether the connection is to be connected or disconnected.

def simulator_toggle(self, activate) Called by the 'Simulator' checkbox. Sets the visibility of the simulator window based on the value of activate. Setting the visibility of the window rather than closing the window has the disadvantage that the window is never actually closed so memory and processing is reduced but not terminated. The advantage this introduces is that reopening the window is done in under a second.
If the user would like to close the window fully, they can do so using the 'x' in the top corner of the window - this terminates all processing associated with the window.

def on_draw(self) Called at most changes in the window. Clears and redraws all window objects.

def update(self, event=None) Updates the simulation window display if it is activated.

def on_close(self) Exits the whole application when the sequencer window is closed.

4.2 sequencer.py

4.2.1 class Sequencer(object)

This class controls the main sequencer frame.

def __init__(self, parent, ver_offset=-650, **kwargs) Parent is the main window, ver_offset defines the vertical offset of the sequencer frame and kwargs are arguments passed to the modules (defining the parent window, theme etc.)

add_module(self, module_type, args) Adds a module initialised using 'args' specified by the string, 'type'. It also updates the master list of SequenceElements. This is called by the 'add' button in the sequencer controls panel.

load_data(self, data) Iterates through 'add_module' for all of the elements in the list, 'data'. This is called by the 'open' button when an external file is to be loaded.

get_offset(self, height) Gets the pixel offset values for the next sequence element. This is the function which handles wrapping when the SequenceElements would run out of the window. Height is an int representing the height of the sequencer module which is to be added.

radio_select(self, ID, update=True) Selects the SequenceElement specified by the int ‘ID’. If update is true then the parent simulation window is updated. ‘update’ is only false when called by the sequence_executor, which allows the sequence elements to be selected while maintaining a smooth animation.

update_joints(self, pos=None, direct=None, pincer=None) If a ‘node’ SequenceElement is currently selected, this will update its values with pos, direct and pincer. This is called when the sliders are altered.

delete_(self, ID) Deletes the SequenceElement at ‘ID’

repack(self) Repacks all of the loaded SequenceElements.

clear(self) Clears all elements in the sequencer. Called by the ‘Clear’ button.

run(self, update_cb) Runs the current sequence using ExecuteThread. Called by the ‘Run...’ button.

stop(self) Exits the currently running ExecuteThread. Called by the ‘STOP’ button

4.2.2 class ExecuteThread(threading.Thread)

This class executes a sequence on a separate thread.

def __init__(self, sequencer, update_cb) Passes through the original sequencer object and the callback which will update the feedback methods.

def run(self) Runs the execute() method of each SequenceElement. Also selects the current element which is being executed.

4.2.3 class _SequenceElement(pyglet_gui.Manager)

The base class for a SequenceElement.

__init__(self, master=None, ID=None, **kwargs) Initialises a blank sequence element with a type label and delete ‘X’.

def toggle_selected(self, state=None) Toggles the ‘selected’ button by calling Sequencer.radio_select()

def deselect(self) Deselects the ‘selected’ button.

def is_different(self, new_values) A boolean describing whether the new data passed to the SequenceElement is sufficiently different to the old data that it should update the labels (a task which is a respectable cause of latency with pyglet_gui)

def set_offset(self, offset) Sets the offset of the SequenceElement in the window.

def delete_(self) Deletes the instance.

def get_path(self) Returns [‘sequence’] - this is used by pyglet_gui in the formatting definition.

def execute(self, prev_val, update_cb) Placeholder definition - sleeps for 1 second and passes the prev_val through.

4.2.4 class PositionElement(_SequenceElement)

def __init__(self, master, ID,, position, direction, pincer, **kwargs) Initialises ‘node’ SequenceElement with the initial values of ‘position’, ‘direction’ and ‘pincer’.

def update_params(self, pos=None, direct=None, pincer=None) Updates the values of the SequenceElement and, if needed, updates the labels.

def execute(self, prev_val, update_cb) This is the function which handles the animation of the arm. It firstly calculates the difference between the new and old position - driving a resolution and time for the animation. It then iterates through a loop, calling update_cb every iteration with the updated values, and sleeping for a period of time.

4.2.5 class TimeDelay(_SequenceElement)

def __init__(self, master, ID, time, **kwargs) Initialises ‘delay’ SequenceElement with the initial value of ‘time’.

def execute(self, prev_val, update_cb) Waits for the current value of time and passes through ‘prev_val’, unaltered.

4.3 simulator.py

4.3.1 SimulationWindow(pyglet.window.Window)

Class which controls simulator window functions. Inherits camera and graphics object functions from graphics.py.

def __init__(self, settings, joints) Settings is a tuple containing the window initial width, height and position. Joints contains a list of the initial position for the joints.
Creates an instance of the pylget.window.Window OpenGL context as per user settings defined in ‘graphics’. Creates instances of GraphicsPart() for each part of the arm. Creates instance of the graphics.Camera object to describe camera position.

def on_resize(self, width, height) Called when window is resized, changes viewport settings based on new aspect ratio to prevent deformation. Width and height are ints indicating the width and height of the window.

def on_draw(self) Called at almost any change in the system. Updates the viewport based on new settings.

def on_mouse_scroll(self, x, y, scroll_x, scroll_y) Called when the mouse is scrolled over the simulation window. Zooms the view by calling the zoom method of the graphics.Camera object. x, y are the positions of the mouse, scroll_x and scroll_y are the change in the scroll wheel position since last call.

def on_mouse_drag(self, x, y, dx, dy, buttons, modifiers) Called when the mouse is clicked and dragged across the simulation window. Orbits and pans the viewport by calling respective methods in graphics.Camera.

def update_joints(self, joints) Called to update the positions of the arm parts. Joints is a list of tuples describing the position, direction and rotation of each part.

def set_visible(self, visible) Called when the window is to be invisible. This method is redefined to also call self.on_resize(). This was a somewhat simple way of solving a bug where the re-appeared window would be blank.

4.4 graphics.py

def initialise() One-time openGL setup. Clears window and sets up 2 light sources.

4.4.1 Camera(object)

def __init__(self, position, focus, up) Initialises camera position, focus and up vectors. Loads settings from 'camera' configuration file.

def user_pan(self, dx, dy) Pans the camera by altering camera position and focus vectors.

def user_orbit(self, dx, dy) Orbits the camera about the current focus point by altering camera position and up vectors.

def user_zoom(self, scroll_y) Zooms the camera by altering focus and location vectors.

def get_cam(self) Returns a list of values describing the camera in the format required for OpenGL functions.

4.4.2 class GraphicsPart(object)

def __init__(self, filename, trans=[0, 0, 0], direct=[0, 0, 1]) Initialises the variables for the position and direction of the part. Also creates a new instance of the pyglet batch rendering class with the indexed vertices. This uses an external (but included in this package) .obj file importer which was not written by me. filename is the filename of the obj file for the part (without extension), trans and direct are the initial translation and direction vectors respectively.

def update_pos(self, trans=None, direct=None, angle=None) Updates position, direction and/or rotation of the part.

def draw(self, camera) Draws the shape using pyglet batch rendering. Camera is the list of floats describing the location of the camera.

4.5 arm.py

4.5.1 class Effector(object)

Stores a position, direction and pincer state. Also has methods to calculate the angles for the arm. This is the most maths heavy class and contains all of the logic required in solving the inverse kinematics of the arm.

def __init__(self, origin_to_endpoint, direction, pincer) Initialises values for position, direction and pincer state as well as calculating preliminary angles from this data by calling self.calc_angles().

def set_arm(self, pos=None, direct=None) Sets the position and/or direction of the arm and calculates new angles for it.

def calc_angles(self) Calculates the joints and angles both for the simulator and raspberry pi mirror using an analytical method. The analytical method is limited by the fact that there are some obscure positions in which this algorithm does not work but it is favoured as it has an almost constant time complexity resulting in the ability to run the simulator at 60fps - it was also a more interesting vector arithmetic exercise. Future releases will include the option to use the *Jacobian Inverse Kinematics Algorithm* if the current machine's processing capabilities will allow for it.
Results are stored in self. None is returned if the current setting for the arm is not possible.

def get_joints(self) Returns the already calculated joints (for simulation display).

def get_angles(self) Returns the already calculated angles (for raspberry pi mirror).

4.6 vector.py

A unique vector module is used since it is more lightweight when compared with common maths libraries, it also minimises the external dependencies of the applet. Vectors are stored as lists of floats.

def mag(vector) Returns the vector magnitude of the passed vector.

def add(vector1, vector2) Returns the element-wise addition of the two vectors.

def sub(vector1, vector2) Returns the element-wise subtraction of the two vectors.

def scalar_mult(scalar, vector) Returns the passed vector multiplied element-wise by the passed scalar.

def dot_prod(vector1, vector2) Returns the dot product of the two passed vectors.

def cross_prod3(vector1, vector2) Returns the cross product of the two defined vectors using the determinant of a 3x3 method. This was seen as a quicker way of calculating this rather than using recursion - it has the disadvantage that it will only work with vectors in $\mathbb{R}^3$ but since these are the only vectors being used, it represents a very small disadvantage.

def unit(vector) Returns the unit vector of the passed vector.

def angle_between(vector1, vector2) Returns the angle between the two passed vectors using the definition of the dot product.

def matrix_mult(matrix, vector) Returns the vector multiplied by the specified transformation matrix.

def rotate_about_vec(vector, axis, theta) Rotates the passed vector around 'axis', 'theta' degrees using the Euler-Rodrigues formula. Algorithm was adapted from: http://stackoverflow.com/questions/6802577/python-rotation-of-3d-vector.

def plane_projection(vector, plane_normal) Projects 'vector' onto a plane with normal, 'plane_normal'

def change_direction(vector, direction) Imitates the vector translations which occur when a direction change is performed in the graphics module.

4.7 server.py

Handles all RPi networking communications.

4.7.1 NetworkHandler(object)

def __init__(self, host, port) Initialises values without making a connection.

def connect(self) Connects to the raspberry pi and initialises a send loop on a separate thread (see NetworkThread)

def disconnect(self) Closes the connection and as a result terminates the code running on the Raspberry Pi. Tells the NetworkThread to terminate and syncs back with it.

def send(self, message) Passes the string 'message' to the primary queue on NetworkThread if a transmission is not currently being made, or the backup queue of NetworkThread if the primary queue is being used to send data.

def send_angles(self, angles, update_all=None) Given as list of angles, this function checks to see if the previous transmission was significantly different to the proposed angle, if it is then the angle is converted to a string command, encoded, and passed to the send function.

4.7.2 NetworkThread(threading.Thread)

def __init__(self, parent, ID) Initialises values for the queueing system, as well as a boolean which indicates when to exit.

def run(self) Starts an infinite loop of sending commands in the queue to the RaspberryPi, and listening for when it is ready for the next command. This method of multithreading means that the whole application doesn't hang while the program is waiting for a heartbeat from the RaspberryPi.

4.8 file_io.py

def load_config_file(filename) Loads a space separated configuration file stored in the file 'filename'.

def load_servo_poi(filename) Loads the 'points of interest' from a servo_poi file.

4.8.1 class SequencerData(object)

def __init__(self, **kwargs) Initialises the window kwargs (used in the filename popups) and a blank filename vairable.

def write_sequence(self, sequencer_list, force_filename=False, returned_filename=None) Writes a sequence object to a simple space-delimited file.

def write_sequence(self, returned_filename=None, cb=None) Reads a simple space-delimited file and converts to a series lists which are parsed in sequencer.py.

4.9 toolbar_menu.py

4.9.1 MenuDropdown(pyglet_gui.option_selectors.Dropdown)

Redefines the dropdown menu included in pyglet_gui so that it acts like a toolbar menu seen on most applications.