University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Actuating a Motor in Response to Sensory Input from a Tracked Body

Self Study

Prepared by
Robert James Aleksandr Baker
ID 20382762
rjabaker
Electrical Engineering
12 January 2013

473 Devanjan Circle
Newmarket, Ontario, Canada
L3Y 8H6

12 January 2013

Manoj Sachdev, chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir:

This report, entitled "Actuating a Motor in Response to Sensory Input from a Tracked Body," was prepared as my 2B Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 301. The purpose of this report is to discuss the design of a system which can actuate a motor based on sensory input, such as a target waving his right arm. This system should allow a motor to imitate a joint in the tracked body. It discusses two different design solutions to achieve this. It is a self-study report.

Although this project was a self-study, I did learn some useful engineering practices from my employer, Ontario Die International (ODI). ODI manufactures cutting dies. It employs an engineering division that works to optimize this process. I worked on a .NET application at my work. The programming skills I learned while working on ODIs application contributed to the design discussed in this report.

This report was written out of personal curiosity. The intention of this project is to aid in the development of a robotic hand or appendage that can imitate a tracked human body. For example, a robotic hand using the design discussed in this report would imitate the hand of a tracked body in real-time.

At the start of this report, I discussed Microsoft Kinect with two of my peers, Andrew Hassan and Garrett Everding. However, both Mr. Hassan and Mr. Everding withdrew from the project before development began. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Robert James Aleksandr Baker
ID 20382762

Contributions

I was the only member of my team developing the project discussed in this report. At the beginning of the term, I began discussing the project with two of my peers, but as the term progressed, both of them withdrew from the project.

My main goal was to develop a system which could actuate a motor in response to body or hand gestures. This goal consisted of three smaller yet unique goals. The first of these goals was to capture and classify body or hand gestures. The second goal was to communicate with some external motor controller and send complex commands. The final goal was to actuate a motor based on incoming signals. This system involved both low and high level programming and circuit design. At the start of the project, I decided to use a Microsoft Kinect camera for motion capture and an Arduino as my motor controller. Consequently, my prerequisite goals were to communicate with the Arduino from a .NET application and listen to image capture information from Kinect.

Since I was the sole member of my development team, my tasks included all of the aforementioned goals, namely .NET development, programming an Arduino, and circuit design. At the start of development, I spent the bulk of my time learning how to read and write to the serial port from both an Arduino and .NET application. After this was accomplished, I spent time developing a .NET framework in which to develop the bulk of my system. While I developed this framework, I also designed the motor control circuit, which was attached to the Arduino; as my framework progressed, I was able to test this circuit and validate its functionality. Finally, after coming into possession of a Kinect camera, I wrote the .NET routines required to listen to the device. Once this was complete, I spent the remainder of my time developing algorithms to classify captured images. Actuating a motor based on these classifications was already completed in the framework I had laid out.

The project discussed in this report is a self-study and is not related to my co-op job, where I worked as an Automation Programmer at Ontario Die International (ODI). However, many of the skills I learned and honed at my work were applied to the development of this project. My tasks at work included .NET development, and near the end of my placement I spent the bulk of my time laying down a framework for one of our products, the Die Tracker System. Though I had to teach myself most of the skills required to design the motor control circuit and program the Arduino, I was already well equipped to write the .NET components of my project. Throughout my co-op placement, I was exposed to many technologies that I have rarely seen during my study terms; ODI employs several Mechanical engineers, whereas I'm usually exposed to Electrical and Computer Engineers during my study terms. Whenever I was introduced to a new technology at work, I asked my colleagues to explain the technology. As this project progressed, the information I learned at work began to come in useful, especially in the design of the motor control circuit.

This project is only a small piece in a much larger, more complicated project. Previously, I mentioned that I began this project with two of my peers, both of whom withdrew from the project before development began. However, our original discussions were of a robotic hand that could mimic human hand gestures or motion. Since our technical expertise, especially in machine design, was lacking, this was quite an ambitious project. But developing a system than can at least actuate one motor in response to body movement or gestures would lay a definite framework for our final objective. Consider a system that can map a single body joint to a motor. If that motor can accurately mimic the joint's rotation, I could build a much more complicated set of motors, each mapped to a different joint, that could mimic an entire body. The most important objective of my project is developing a modular system; that is, the component controlling the motor has no understanding of the component classifying body gestures. As a result, multiple motor control components can easily be added to the system without much modification to the existing framework, since each component is independent of the others. Despite my modular design, my final objective of creating a robotic hand is still quite ambitious and may require much more planning and development before it's anything tangible. But in the broader scheme of things, the project discussed in this report is a small milestone leading up to a much more complicated and intelligent system that can mimic body gestures and motion.

Summary

The main purpose of this report is the design of a system which can actuate a motor based on sensory input. This sensory input is received from a tracked body. This system is a milestone in the design of a robot that imitates a tracked human body, such as a robotic hand or arm.

The major points discussed in this report refer to a comparison between two possible body tracking algorithms: gesture trees and joint tracking. These algorithms have different motor control techniques. The gesture tree algorithm controls a DC motor, while the joint tracking algorithm controls a stepper motor. This report outlines a set of objectives that the system must satisfy in its design. The system must actuate a motor based on some change in sensory input, such as a hand moving. Therefore, it must also track some target body. The system must be accurate in its response to input. It must also be expandable.

The major conclusion in this report is that the gesture tree algorithm fails to meet all the system’s design objectives because it lacks the required accuracy. The joint tracking algorithm meets all of the system’s design objectives.

The major recommendations in this report are that the joint tracking algorithm should be fully implemented and that research into a new method of body tracking should be completed. The current method of body tracking does not support accurate hand joint tracking.

Table of Contents

List of Figures

List of Tables

1 Introduction

Robotics is a wide field encompassing many different applications. The design of a robotic appendage that could imitate human motions could be used in both medical and manufacturing applications, to name a few. The project discussed in this report is the design of a system that can actuate a motor based on human motions, received as sensory input. Such a system could be eventually expanded to include an entire appendage. Since the scope of such a project is so large, the system discussed in this report has been limited to the control of a single motor, imitating a human joint. However, this system was designed with expansion in mind.

2 Design

2.1 Design Objectives

The objective of this project is to design a system which can actuate a motor based on sensory input from a moving body. Qualitatively, success is judged by the weighted objectives listen in Table 2-1, where an overall passing grade is a score of 7.0.

Table 2-1 – Design Objectives

Objective Code	Objective Description	Weight
A	The system should track a body or hand in real time	2.0
B	An attached motor should turn on, rotate or change direction in response to some change in sensory input from the tracked body	2.0
C	The system should support accurate motor control	4.0
D	The system should support multiple motors, though only one motor needs to be implemented	2.0

Table 2-1 identifies objectives A and B as the most important aspects of this project. Obviously, objective B relies heavily on the success of objective A. Even so, these two objectives were separated because of the complexity in classifying body gestures, a problem which umbrellas some advanced fields of study (namely, machine learning) and will be discussed further in Section 4. Objectives C and D were added because this project is actually part of the design for a robotic hand; therefore, it must support precise motor movements and be expandable for multiple motors.

2.2 Design Outline

This project involves three major phases: the capture and classification of body image frames, the construction and execution of appropriate motor commands, and the actuation of a motor based on these commands. Figure 1-1 describes the general procedure as it occurs in the project.

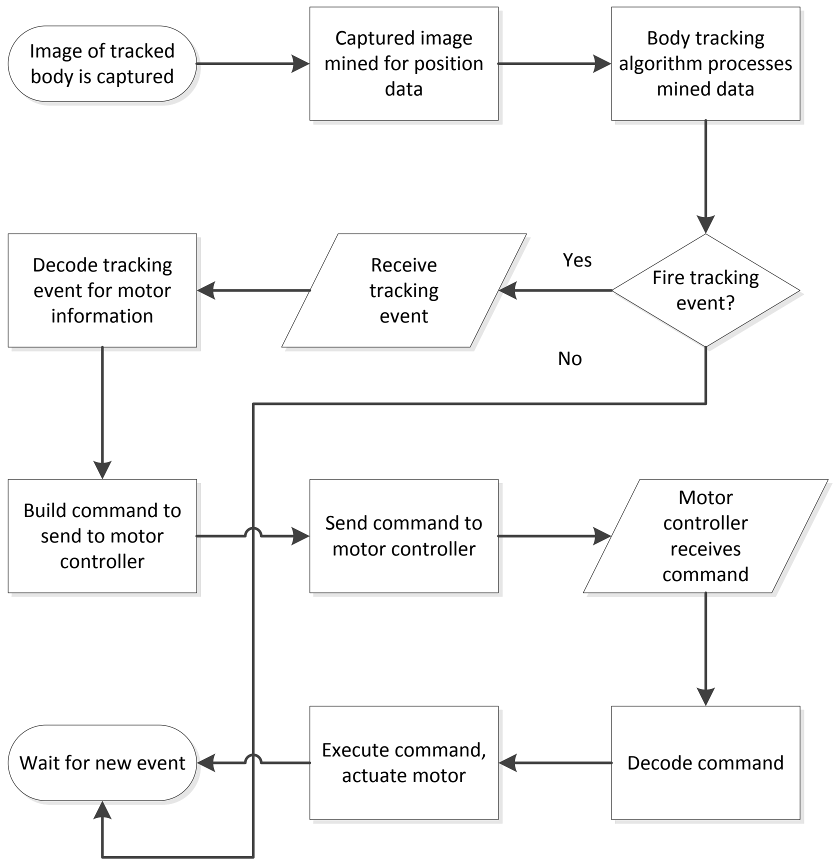

Image of tracked body is captured
Captured image mined for position data
Body tracking algorithm processes mined data
Fire tracking event?
Yes
No
Receive tracking event
Decode tracking event for motor information
Build command to send to motor controller
Send command to motor controller
Motor controller receives command
Decode command
Execute command, actuate motor
Wait for new event

Figure 1-1 – General Procedure For Motor Actuation Based on Tracked Body Motion

Microsoft Kinect was selected to capture body motions. The camera has its own API which provides some image frame intelligence (Section 3.1). Since Kinect’s API supports .NET development [1], development of the main project was allocated to a .NET background. Motor control requires a device with basic I/O capabilities, one that can communicate with a .NET application. A programmable board called an Arduino was selected for this job (Section 3.2).

The .NET application contains three projects: ArduinoUtilities, KinectUtilities and Skynet. ArduinoUtilities handles communication with an Arduino. KinectUtilities receives and classifies image frames from Microsoft Kinect. This project contains dependencies on the Microsoft Kinect

Software Development Kit (SDK) discussed in Section 4. Skynet references both ArduinoUtilities and KinectUtilities, using them as class libraries to host the main project.

2.3 Proposed Solutions

This document discusses two major solutions to the design objectives listed in Table 2-1. Section 4.2 describes the distinct joint tracking and gesture tree algorithms, both of which are used as body tracking algorithms in the .NET application. Joint tracking focuses on the real-time three-dimensional position of tracked joints, while gesture trees searches for complicated patters (gestures) in the tracked body.

Originally, the gesture tree algorithm (Section 4.1) was implemented to control a DC motor. As an alternate solution, the joint tracking algorithm was introduced, running independently of the gesture tree algorithm. Due to the nature of joint tracking (Section 4.2), support for a stepper motor was added to the application. Even though the gesture tree algorithm could support stepper motor control, for the purposes of this document, it has been grouped with DC motor control. Table 2-2 summarizes the two major solutions discussed in this document.

Table 2-2 – Major Design Solutions Discussed in this Document

Solution Code	**Body Tracking Algorithm**	**Motor Type**
A	Gesture Trees	DC Motor
B	Joint Tracking	Stepper Motor

Both solutions listed in Table 2-2 adhere to the design outline laid out in Table 2-1. As the remainder of this document will discuss, the applications used in this project were designed to run independently of one another and make use of the abstraction methods provided by .NET (interfaces and inheritance).

3 Required Technologies

This project requires the use of two external devices: Microsoft Kinect and Arduino. This section discusses their basic architecture, the out-of-the-box utilities they provide and their usage protocols.

3.1 Microsoft Kinect

Microsoft Kinect is a motion sensing camera that is in a wide variety applications. Equipped with an RGB camera, depth sensor and multi-array camera, it can be used for gesture tracking, facial recognition and voice recognition.

The most recent Kinect SDK, version 1.6, can be used to develop Kinect applications in a .NET environment. The SDK provides a KinectSensor object that can be used to control the camera and read the captured image at a rate of up to 30 frames per second [2]. The captured image frames can be mined for an array of Microsoft Skeleton objects. Each Skeleton contains a collection of Microsoft Kinect Joint objects that contain their respective coordinates in three-dimensional space. Figure 3-1 illustrates the position of these Kinect Joint objects relative to an actual body.

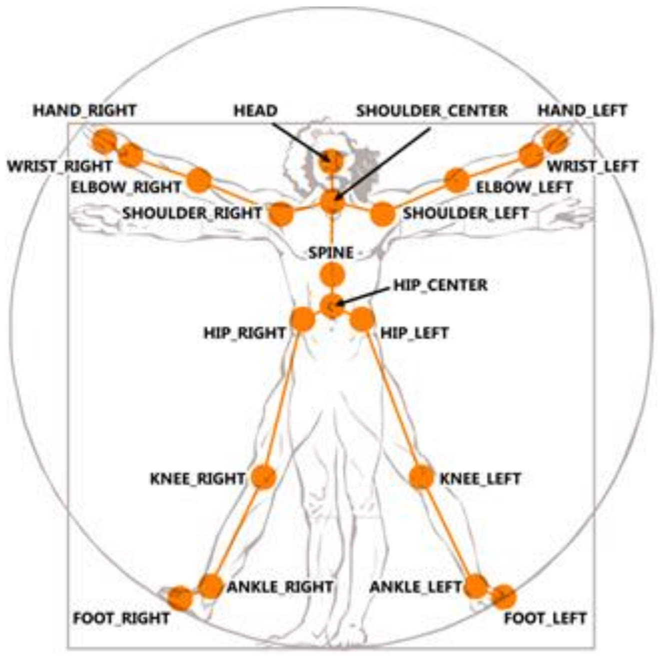

Figure 3-1 – Position of Skelton Joints on Body [3]

The Kinect SDK does not provide any out-of-the-box hand tracking or gesture recognition algorithms.

3.2 Arduino

The Arduino is an "open-source electronics prototyping platform" [4]. The board contains a microcontroller that can be programmed using a proprietary Arduino programming language. The board contains a number of digital and analog pins that can be used to attach sensors and actuators to the board. Through the USB serial port, applications written with an Arduino IDE can be loaded onto the board, and this serial port can also be used during application runtime to send and receive information. An Arduino Uno was purchased for this project's development. Appendix A summarizes the board's characteristics.

A generic Arduino application contains two functions, setup and loop. The setup function is executed once at the start of the application. It can be used to initialize variables or set pin modes. The loop function runs exactly as the name implies, executing consecutively in an infinite loop.

The board's documentation claims that its output pins can provide up to 5V. Table 3-1 describes the measured characteristics of a digital output pin. The voltages were measured by attaching a resistor's leads to the output pin and ground pin on the board.

Table 3-1 – Digital Output Pin Voltage Characteristics

Resistor	Pin High Voltage	Pin Low Voltage
1K	4.98 V	0 V
100k	5.09 V	0 V
100k	5.11 V	0 V

Output pins can also be set to an analog intensity ranging from 0 to 255, corresponding to an output voltage of 0 to 5V. Figure 3-2 describes a pin's analog output characteristics, plotting the output voltage against the corresponding intensity. The figure's data was measured via the same method in Table 3-1.

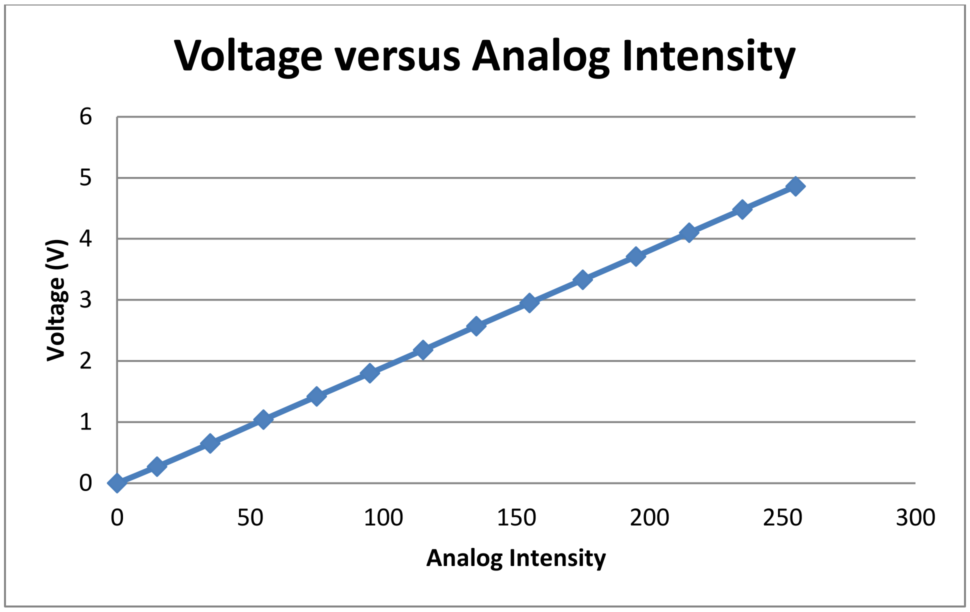

Figure 3-2 – Analog Output Pin Voltage Characteristics, Voltage versus Analog Intensity

The data in Figure 3-2 suggests that pin output voltage and intensity share a linear relationship.

4 Body Tracking and Classifying

This section discusses "classifying" a tracked body, where classifying is the process of identifying important characteristics from captured frames, such as gestures or joint positions. In designing an algorithm to classify a tracked body's position, it was important to consider what information Kinect provides. The Skeleton object has very little intelligence, only providing information on the body's position at the time of frame capture. Therefore, algorithms need to be intelligent enough to classify a gesture based on simple coordinate data.

This led to the design of two different algorithms: gesture trees and joint tracking. As the name implies, the gesture tree algorithm focuses on classifying potentially multiple Skeleton frames into complicated gestures. Conversely, the joint tracking algorithm deals with single frames, mining information from a single Skeleton and forgetting that information as soon as the next frame is captured.

Both algorithms share a common ICapturingFunction interface. In fact, other modules implement this interface, such as a Skeleton image renderer used for debugging purposes. A SkeletonController object receives Skeleton image frames from the KinectSensor, an event that occurs up to 30 times per second. Each ICapturingFunction receives the collection of Skeleton objects contained in the captured frame and also the corresponding timestamp.

4.1 Gesture Trees

A GestureTree object represents the relative configuration of a collection of joints at a given time. It is used for comparing two different gesture frames. For example, a subject flexing his right arm will create a GestureTree of his shoulder, elbow, wrist and hand joints. The relative angle between each of these joints is recorded in the GestureTree. Comparing this GestureTree with another flexing gesture frame, the tree will verify if the recorded angles within some given tolerance of one another. If the angles are the same, the GestureTree is satisfied.

Classifying complicated gestures, such as a subject waving his right arm, requires multiple GestureTrees. These objects are stored in a MovingGestureTree and are associated with a minimum and maximum timespan. Figure 4-1 illustrates the process by which a MovingGestureTree is satisfied by a real-time gesture.

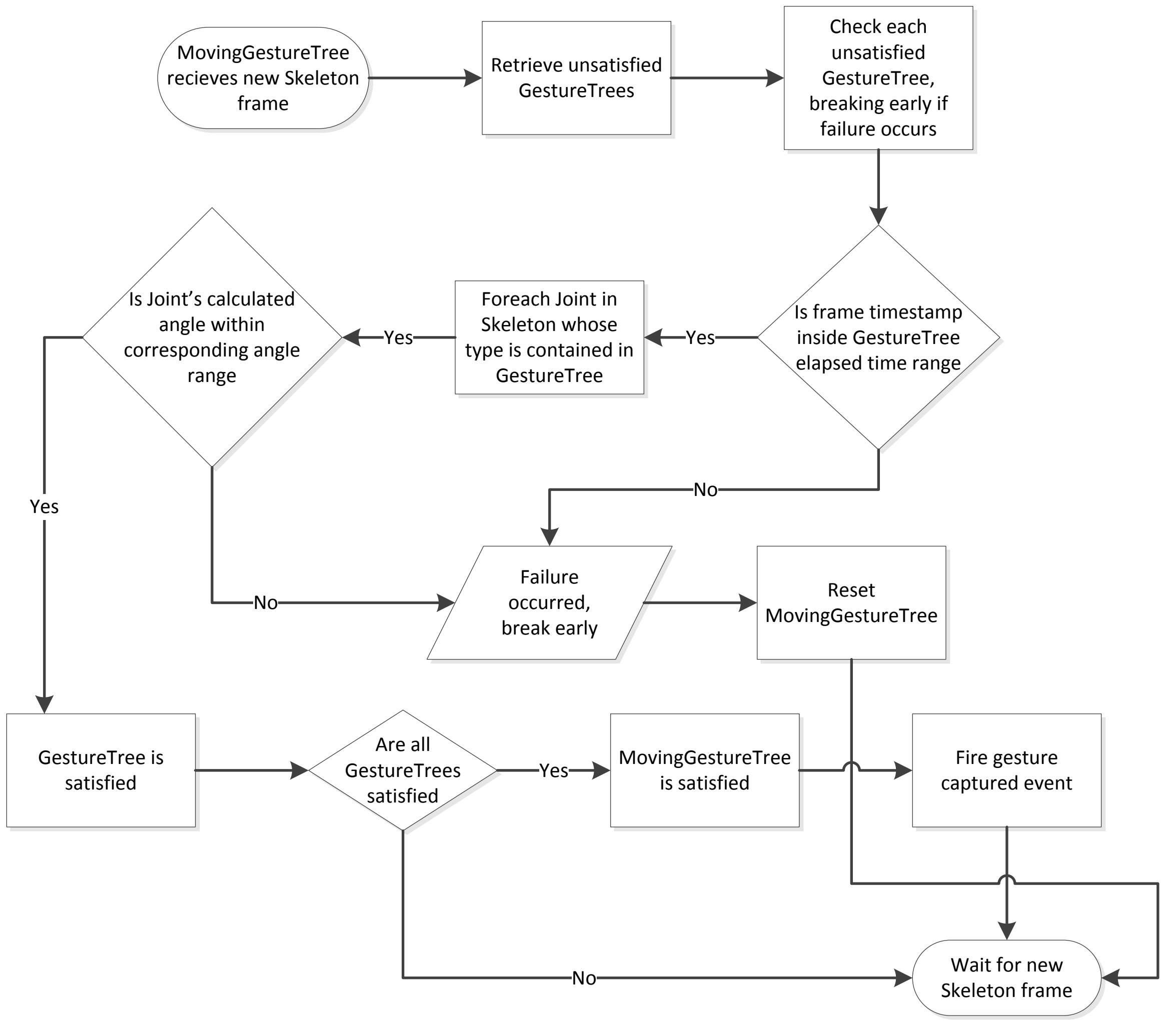
MovingGestureTree recieves new Skeleton frame
Retrieve unsatisfied GestureTrees
Check each unsatisfied GestureTree, breaking early if failure occurs
Is frame timestamp inside GestureTree elapsed time range
Yes
Foreach Joint in Skeleton whose type is contained in GestureTree
Yes
Is Joint's calculated angle within corresponding angle range
Yes
No
No
Failure occurred, break early
Reset MovingGestureTree
GestureTree is satisfied
Are all GestureTrees satisfied
Yes
MovingGestureTree is satisfied
Fire gesture captured event
No
Wait for new Skeleton frame

Figure 4-1 – Gesture Tree Algorithm, Satisfying a MovingGestureTree

MovingGestureTrees are loaded from XML files. This provides the ability to record and save various types of gestures and load them at the start of the application. A GestureTreeBuilder class is used to build the MovingGestureTree and serialize it into XML.

4.2 Joint Tracking

The joint tracking algorithm looks at the angle created by the specified joint. It only supports single Skeleton processing. Each time it receives a new Skeleton object collection, it extracts the first Skeleton and iterates through each Kinect Joint whose type is supported by joint tracking (see Appendix C). Figure 4-2 illustrates the joint's measured angle, which exists in three-dimensional space.

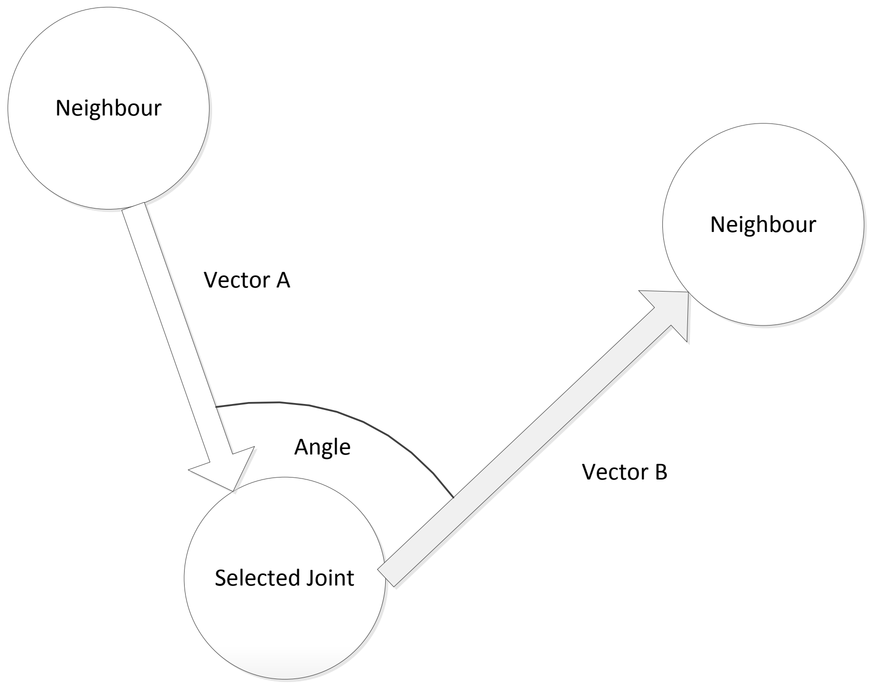

Figure 4-2 – Selected Joint's Measured Angle

The angle is calculated by building vectors A and B, described in Figure 4-2, and inserting them into the following dot product equation:

$$\text{joint\_angle} = \cos^{-1}\left(\frac{A \cdot B}{|A||B|}\right) \tag{1}$$

The calculated joint_angle is then packaged into a MovingJoint object that includes the original joint type. The algorithm fires an event that includes the calculated MovingJoint and the original timestamp. A distinct event is fired for each iteration (one for each calculated MovingJoint).

5 Arduino Pin Control

Once the tracked joint or gesture has been classified, a series of component mappings are alerted within the .NET application. Each component mapping inherits the IComponentMapping interface, which removes any unnecessary inter-project dependencies.

Depending on which body tracking algorithm was used, these alerts may contain different information. However, each alert is packaged in the same IJointFrameParameter interface, which is sent to the corresponding Skynet Joint, a class representing a motor mapped to a joint, immediately after the alert is received. As described in Section 6, different motors require different control techniques. This intelligence is handled at a later phase in the control process, during construction of the command package (Section 5.2).

5.1 Kinect Body Tracker

As previously mentioned, after the KinectUtilities project classifies a gesture or tracked joint, it fires an alert. The KinectBodyTracker contained in the Skynet project listens for this alert. To maintain generic functionality, this class inherits an IBodyTracker interface; however, at this early stage in the project's development this interface has not been explicitly used. Since this project can involve various body tracking algorithms, the KinectBodyTracker is equipped to listen to various classes in the KinectUtilities project that fire tracking alerts.

Immediately after an alert is captured, the KinectBodyTracker determines which Skynet Joints need to be made aware of the event. In the case of joint tracking, each alert targets a single joint type; therefore, the KinectBodyTracker simply determines which Skynet Joint in its collection is of the specified type and proceeds with alerting that Skynet Joint of the event. It is possible that the alert may correspond to a joint type not contained in the KinectBodyTracker's collection, but this event is then ignored by the class.

The JointController's alerts consist of a MovingJoint object and a timestamp. The KinectBodyTracker builds an IJointFrameParameter from this MovingJoint object, including the timestamp. At this point, the KinectBodyTracker has identified which of its Skynet Joint objects was specified in the alert. The Skynet Joint is passed the IJointFrameParameter and left to handle the event as it sees fit. The reason for giving the Skynet Joint so much control over the IJointFrameParameter is that different tracking algorithms and motor control techniques require different alert handling.

5.2 Arduino Command Package Building

Once a Skynet Joint has possession of an IJointFrameParameter, it executes the appropriate command, saves any pertinent information, which depends on the tracking algorithm currently in use, and disposes of the object. This is the entire lifecycle of an IJointFrameParameter. The most important event in this lifecycle is the execution of an appropriate command, which must be constructed by the Skynet Joint.

These commands are called command packages. In the current project's state, they contain three bytes of information. Table 5-1 describes a generic command package syntax, which accommodates both high and low level commands.

Table 5-1 – Generic Command Package Syntax

Byte 1	**Byte 2**	**Byte 3**
Command ID	Argument 1	Argument 2

Appendix XX contains a complete list of the currently existing command packages.

Before a Skynet Joint can execute a command, it needs to build the appropriate command package based on the IJointFrameParameter. In the project's current state, the Skynet Joint can build one of two command types: a stepper motor command and an analog set pin command.

5.2.1 Stepper Motor Command Packages

The Arduino application supports a stepper motor instance this is provided by a generic Stepper library. As a result, it accepts stepper control command packages with the following syntax:

Table 5-2 – Stepper Control Command Package Syntax

Byte 1	**Byte 2**	**Byte 3**
Set Stepper Motor X Command ID	Speed (rpm)	Steps

The first byte in Table 5-2 contains the command ID, which is tailored to a specific stepper motor. That is, stepper motor X and stepper motor Y will have different command IDs. As a result, the application can only support a finite number of stepper motors.

Table 5-2 implies another restriction: since the stepper motors are hardcoded into the application, the motor objects need to be initialized via command packages or to have a set of reserved pins in both the .NET and Arduino applications. The latter solution is implemented. Each stepper motor requires four reserved pins, whose numbers are stored in both the .NET and Arduino applications. If the motors are not attached to the system, the applications ignore the reserved pins; otherwise, it is assumed that the correct stepper motor is attached to its reserved pins.

Building the stepper motor command package requires the calculation of its speed and the number of steps required. The IJointFrameParameter's MovingJoint property contains the new joint angle. Its attached timestamp contains the elapsed time since the last joint frame. The number of steps per revolution for each stepper motor is saved in the .NET application. Therefore, the command's steps argument (byte 3) can be calculated from the following equation:

$$\text{steps} = \frac{\text{joint\_angle} \times \text{STEPS\_PER\_REV}}{2\pi} \tag{2}$$

The calculated steps value is rounded to an integer. It is restricted to a signed value between 0 and 127, where the sign indicates the step direction.

The speed argument (byte 2) can be calculated from the following equation, which is dependent on the previously calculated steps argument. It is in rotations per minute.

$$\text{speed} = \frac{\text{steps} \times \text{STEPS\_PER\_REV}}{\text{timespan\_in\_minutes}} \tag{3}$$

The calculated speed is also rounded to an integer, and is an unsigned value between 0 and 255.

Once the steps and speed values are calculated, the Skynet Joint constructs the command package into an array of three bytes and executes the command (Section 5.3).

5.2.2 Analog Set Pin Command Packages

The analog set pin command package maps to a single analog write command in the Arduino application. Table 5-3 described the package's syntax:

Table 5-3 – Analog Set Pin Command Package Syntax

Byte 1	**Byte 2**	**Byte 3**
Analog Set Pin Command ID	Pin Number	Intensity

This command supports up to 256 different pins (byte 2). The intensity argument (byte 3) sets a pin's analog value from 0 to 255. In terms of a DC motor, this argument controls the angular velocity of the motor's rotation. The motor's clockwise and anti-clockwise rotation directions are controlled by separate pins. This command package supports very basic motor control and lacks the precision of the stepper control command package.

5.3 Command Package Execution

Once the command package is built, the Skynet Joint needs to execute it. Since the analog set pin command lacks direction control, the motor requires two pin mappings to rotate in both the clockwise and anti-clockwise directions. The stepper motor control command handles the rotation direction, since its step argument is a signed integer. Regardless, both execution methods implement the same IComponentMapping interface, which is used to communicate with the ArduinoUtilities project from the Skynet project.

5.3.1 Component Mappings

An IComponentMapping represents a pin mapping or a stepper motor, although it can be extended to accommodate many different sensors and devices. The ArduinoSerialPort class in the ArduinoUtilities project contains a collection of IComponentMappings; each of the items in this collection is connected to the ArduinoSerialPort, and can request a write or read command to the Arduino via the serial port (Section 5.3.2).

There are two main implementations of the IComponentMapping interface: the PinMapping and StepperMotor classes, both contained in the Skynet project. Since a standard DC brush motor requires two pins to control, one for each direction, a SmartPinMapping wrapper class contains two PinMapping instances. These instances are named in the wrapper class based on their rotation direction. The StepperMotor class is more high level; it is aware of the pins attached to the stepper motor (a total of four pins), but it does not communicate directly with these pins.

When the Skynet Joint constructs analog command packages, it builds one for the clockwise direction and another for the anti-clockwise direction. In all cases, one of the pin’s intensity is set to 0, while the other is set to a value greater than or equal to 0, depending on the calculated angular velocity. For stepper motors, control of the individual pins is handled in the Stepper library on the Arduino.

Executing a command package via the PinMapping or StepperMotor classes requires calling the ExecuteCommand delegate guaranteed by the IComponentMapping interface and passing it the corresponding command package. The SmartPinMapping wrapper class calls this delegate twice, once for both of its PinMappings, while the StepperMotor only calls it once.

5.3.2 Arduino Serial Port

The ArduinoSerialPort class contained in the ArduinoUtilities project is a wrapper for the standard .NET SerialPort class. It is the only application that communicates with the Arduino, a process which occurs via the serial port. The ArduinoSerialPort writes command packages, an array of three bytes, into the serial port. If the Arduino writes a response package into the serial port, the SerialPort class throws an event that the wrapper class responds to accordingly.

The ArduinoSerialPort contains a collection of IComponentMapping objects. As described in Section 5.3.1, when an IComponentMapping attempts to write a command package, it calls its ExecuteCommand delegate function. This function is mapped to the WriteCommandPackage method in the ArduinoSerialPort class, which writes the specified byte array into the serial port. This concludes the .NET application’s role in executing a command package in response to a tracking event.

5.3.3 Arduino Command Package Parsing

The Arduino is programmed to read an array of three bytes from the serial port. It expects this byte array to abide by the generic command package syntax described in Table 5-1.

A typical Arduino application executes an infinite main loop (Section 3.2). The Arduino application used in this project searches the serial port for any incoming byte data. During development, it was found that there is a slight delay between writing a byte to the serial port and that byte becoming available to read. For example, an external .NET application wrote three bytes into the serial port. When the Arduino application noticed there was data in the serial port, it immediately read three bytes. The first byte was correct, but the other two bytes were gibberish because the .NET application had not finished writing the correct bytes into the serial port. Two possible solutions to this problem were identified (Table 5-4).

Table 5-4 – Possible Methods for Reading Correct Byte Arrays from the Serial Port

Solution Code	Solution	Strengths	Weaknesses
A	When the application first recognizes a byte to read in the serial port, delay a few milliseconds to wait for the entire array to be written into the port. After the delay, read the expected byte array length from the port and execute the command package.	- Simple solution, requiring very little modification to the existing application. - Will be a quick implementation.	- The correct delay time may vary between different computers/serial ports. This means that the application could fail to wait long enough and still read incorrect data. The application could also wait too long and unnecessarily extend execution time.
B	Each time a byte is recognized in the serial port, add it to a buffer array. When the buffer is full, execute it as the command package.	- Guarantees that the correct byte array has been read from the serial port. - Robust solution	- Requires severe modifications to the existing application.

Solution A from Table 5-4 was implemented to solve the issue because it required less time to implement. It was noted as a temporary solution.

Once the byte array has been read from the serial port, the application parses it. Figure 5-1 illustrates this process.

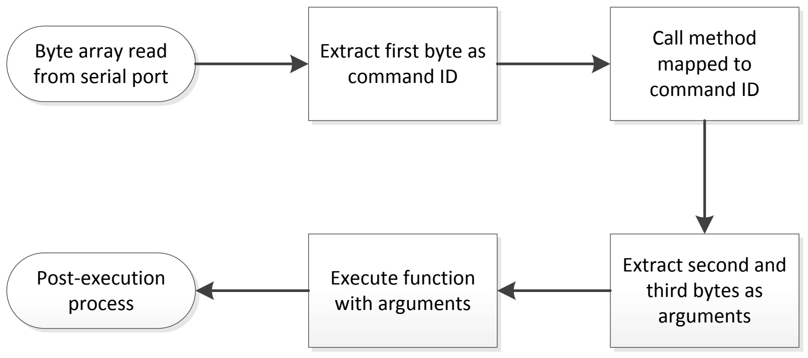
Byte array read from serial port
Extract first byte as command ID
Call method mapped to command ID
Extract second and third bytes as arguments
Execute function with arguments
Post-execution process

Figure 5-1 – Command Package Execution

The post-execution process in Figure 5-1 could include error checking or sending a response byte package.

6 Motor Control and Actuation

This section will discuss both motor types introduced in Table 2-2 and the circuits required to control them through the Arduino. The circuit configurations shown in this section were designed with “Fritzing” software, a popular tool used in the design of Arduino-controlled systems.

6.1 Stepper Motor

The major benefit of a stepper motor is that it provides precise control without the use of a feedback sensor. The motor’s rotation is restricted to a constant number of step positions. Commanding the motor to move five steps will always result in the motor rotating a constant number of degrees, which is defined by the number of steps in one revolution [5]:

$$\mathrm{roation\_in\_degrees} = \frac{360 \times \mathrm{steps}}{\mathrm{STEPS\_PER\_REVOLUTION}} \tag{4}$$

The above equation is directly related to the one in Equation 1, which is used to build the stepper motor control command package in the .NET application.

The Stepper library used in programming the Arduino builds an output signal called pulse train. This signal is based on a specified number of steps and rotation speed (rpm) and is used to control the motor. Specifying a positive number of steps rotates the motor in the clockwise direction, while specifying a negative number of steps rotates it in the anti-clockwise direction. Both pieces of information are extracted from the stepper motor control command package.

A downside in using a stepper motor is that it requires four pins to operate. However, unlike the DC Motor in Section 6.2, the stepper motor does not require a motor control circuit to control its rotational direction. Controlling a stepper motor via an Arduino requires very little extra intelligence. The Stepper library handles both the pulse train construction and controlling the motor’s rotational direction, a major obstacle in using a DC motor.

6.2 DC Motor

A typical DC motor has two input leads that are used to control the motor’s rotation in the clockwise and anti-clockwise directions by reversing the input signal’s polarity. Both leads form a closed loop with the input signal, and the signal’s magnitude controls the motor’s rotational speed. The motor requires some intelligence to control its rotational direction by varying the input signal's polarity (the signal's magnitude can be controlled from the Arduino output pins). Table 6-1 describes the operational characteristics of the motor control circuit, including an emergency stop switch (or enable switch) to provide a manual level of control over the motor. Since the motor has two input leads, it requires two output pins on the Arduino, one for both directions of rotation.

Table 6-1 – Motor Control Circuit Operational Characteristics

Enable Switch	**Pin A**	**Pin B**	**Motor Rotation Direction**
High	High	High	Motor stopped, no rotation
High	High	Low	Clockwise rotation
High	Low	High	Anti-clockwise rotation
High	Low	Low	Motor stopped, no rotation
Low	--	--	Motor disabled, no rotation

The functionality described in Table 6-1 allows the Arduino to control the motor via two output pins. Consequently, controlling the motor requires two separate analog set pin command packages.

To provide one more level of control over the motor, the enable switch can be attached to a digital output pin on the Arduino. Toggling the pin's state will enable or disable the motor.

Achieving the operational characteristics in Table 6-1 requires the use of an integrated circuit which can be used to provide bidirectional current. This is called an H bridge. L293D was selected, and it can drive up to two motors [6].

Figure 6-1 illustrates the motor control circuit for the DC brush motor, also called the "Fritzing" configuration.

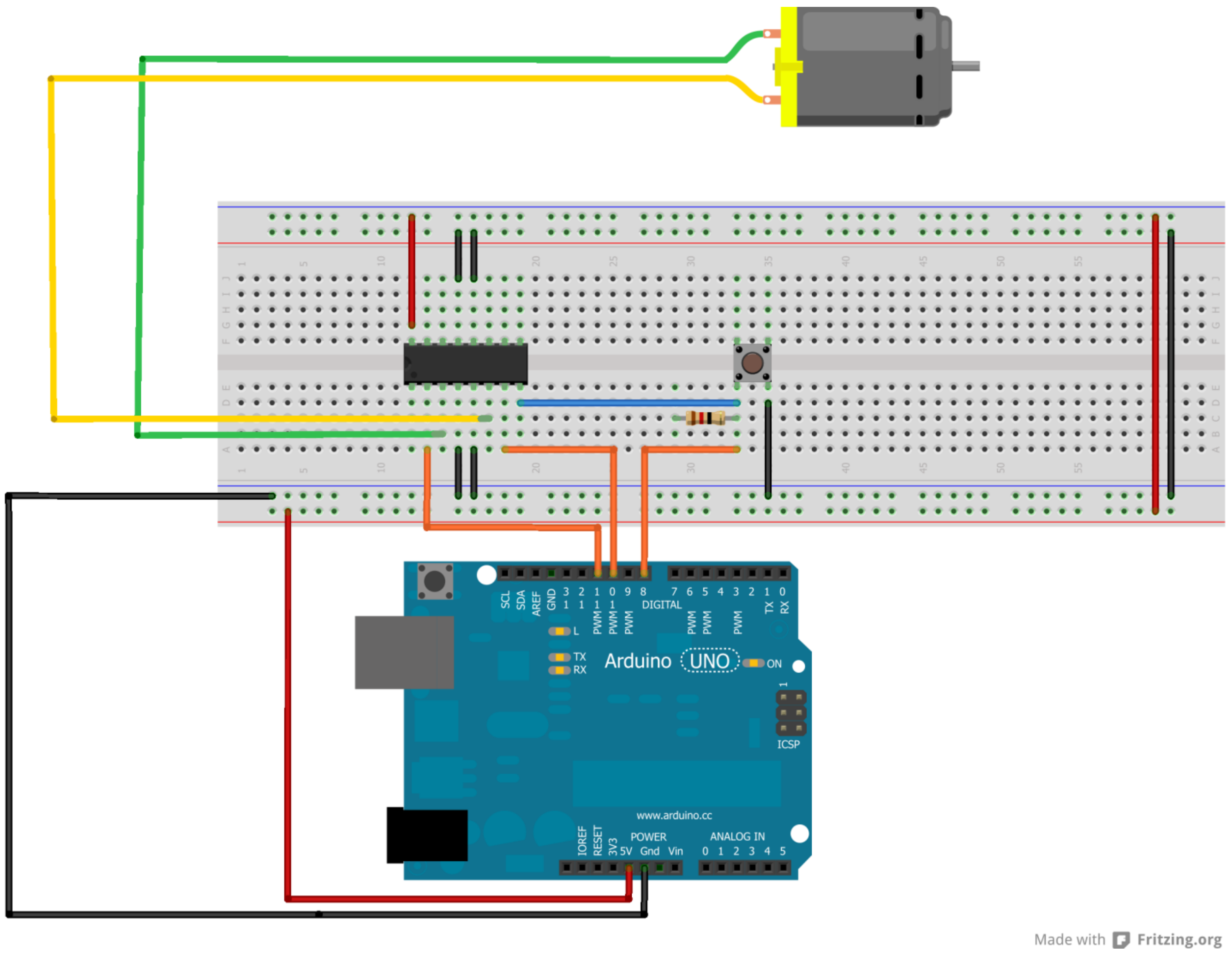

Figure 6-1 – DC Brush Motor Control Circuit, "Fritzing" Configuration

The emergency stop switch was replaced with the push button in Figure 6-1 because no adequate switches were available when the components were purchased. Pressing the push button disables the motors, as does setting the enable pin to its low voltage.

7 Engineering Analysis

Table 7-1 introduces two distinct body tracking algorithms, which also include separate motor control techniques. The evaluation scheme used in this section is based off Table 2-1. Table 7-1 summarizes the evaluation of the two body tracking algorithms.

Table 7-1 – Satisfied Design Objectives Summary

Objective Code	**Objective Description**	**Weight**	**Solution A**	**Solution B**
A	The system should track a body or hand in real time	2.0	2.0	2.0
B	An attached motor should turn on, rotate or change direction in response to some change in sensory input from the tracked body	2.0	2.0	2.0
C	The system should support accurate motor control	4.0	0	4.0
D	The system should support multiple motors, although only one motor needs to be implemented	2.0	2.0	2.0
Total		10.0	6.0	10.0

The gesture tree algorithm (solution code A) satisfies objectives A, B and D in Table 2-1. It is able to track a body in real-time by classifying image frames into GestureTrees. When a tree is satisfied, it is able to actuate a motor. Using the project's .NET framework and BCPs, it can support multiple motors. However, the algorithm fails to satisfy objective C. The system must provide support for precise motor control, which both the gesture tree algorithm and its motor control technique are unable to provide. Since the gesture tree algorithm only fires an alert when a certain pattern has been recognized, it is unequipped to track minute body movements or provide an accurate description of a tracked body's position. The DC motor that the algorithm controls does cannot imitate a joint's position with enough accuracy to resemble a real hand or body part. Therefore, solution A receives a failing grade of 6.0.

The joint tracking algorithm (solution code B) satisfies all of the objectives listed in Table 2-1. The algorithm classifies a tracked body's joint positions by mining joint angles from the captured image frames. It actuates a stepper motor based on these angles, rotating it to a precise angle by incrementing a number of steps. As previously mentioned for solution A, the system is still expandable with this algorithm because it uses the specialized .NET framework and BCPs. Therefore, solution B receives a passing grade of 10.0.

Conclusions

From the analysis in the report body, it was determined that a system meeting the objectives in Table 2-1 can be implemented by using solution B in Table 2-2. Table 7-1 summarizes each objective's weighting and the achieved score based on the solutions proposed in Table 2-2 and the design in this report.

Table 7-1 – Satisfied Design Objectives Summary

Objective Code	**Objective Description**	**Weight**	**Solution A**	**Solution B**
A	The system should track a body or hand in real time	2.0	2.0	2.0
B	An attached motor should turn on, rotate or change direction in response to some change in sensory input from the tracked body	2.0	2.0	2.0
C	The system should support accurate motor control	4.0	0	4.0
D	The system should support multiple motors, although only one motor needs to be implemented	2.0	2.0	2.0
Total		10.0	6.0	10.0

Solution B, which implements the joint tracking algorithm and uses a stepper motor control technique, receives a passing grade of 10.0. Solution A, which implements the gesture tree algorithm and uses a DC motor control technique, receives a failing grade of 6.0. Solution A lacks the precision required by objective C and is therefore unable to meet the design requirements necessary to design an accurate system. Therefore, it can be concluded that by using solution B (Table 2-2), the system proposed in this report meets the project’s design requirements (Table 2-1).

Recommendations

Based on the analysis and conclusions in this report, it is recommended that this system be fully integrated with solution B in Table 2-2 because it meets all of the design requirements specified in Table 1-1. To fully implement solution B, it is recommended that the following steps be taken:

- Support for at least three stepper motors should be added to the system. This requires the addition of command IDs for the BCPs, Stepper class objects and reserved pins for the actual motors.
- A feedback network should be implemented. This requires expanding the existing byte response package infrastructure.
- Research into stepper motor response time (the amount of time required to make one full rotation) should be completed and the Arduino application should be adjusted accordingly.

It is recommended that the Arduino application no longer reads the serial port based on a timer (solution A in Table 5-4). Instead, the application should be modified to read the recieved byte packages via a buffer (solution B in Table 5-4).

Finally, it is recommended that research into a supplementary method of body tracking be completed, with a special emphasis on hand tracking. Since Microsoft Kinect does not support accurate hand joint tracking, a supplementary device may be required to capture this information.

Glossary

Arduino: A programmable board used for I/O.

Analog Intensity: An integer from 0 to 255 that can be used to set the pin output voltage on an Arduino.

Body Tracking Algorithm: The process that tracks a person's movements through some camera or sensor. Two examples of this process are the joint tracking and gesture tree algorithms.

Byte Command Package (BCP): A collection of three bytes representing a command sent from some external application to the Arduino via the serial port. The first byte contains the command ID, while the second and third bytes are command arguments.

Captured Image Frame: An image frame received from the Kinect sensor, containing an array of Microsoft Skeleton objects. This term usually refers to the camera's most recent image frame.

Classify: Used in the context of a body tracking algorithm, it refers to the process of mining information from an image frame and determining what time a movement has occurred. This process refers to gesture recognition and joint tracking.

DC Motor: An electric motor powered by direct current.

Frame Capture: The event that occurs when the Kinect sensor captures a new image frame.

Fritzing: The design of an Arduino-controlled application with an external circuit. Fritzing software is available to aid in this design.

Gesture Frame: Represents a collection of pertinent Joints (and their three-dimensional coordinates) associated with a certain configuration. For example, a right arm flexing would have an associated gesture frame containing right shoulder, right elbow, right wrist and right hand Joints.

Gesture Tree: A type of body tracking algorithm, collects joint angles from multiple image frames and searches for relevant patterns.

H Bridge: An electronic circuit which can be used to provide bidirectional current.

Joint Tracking: A type of body tracking algorithm, looks at the angle made by a specified joint in the current image frame.

Microsoft Kinect: A camera used to classify a number of tracked bodies, noting their three-dimensional position in space. Equipped with an RGB camera, depth sensor and multi-array camera, it can be used for gesture tracking, facial recognition and voice recognition.

Pulse Train: A signal composed square-wave pulses. In the context of a stepper motor, each pulse is used to move the motor forward a step.

Real-time Gesture: A subject's gesture captured in real-time by the Kinect sensor, represented in the application by a collection of Skeleton objects.

Step: In the context of a stepper motor, represents the smallest increment in a stepper motor's rotation (see Stepper Motor).

Stepper: A generic library used for stepper motor control by Arduino applications.

Stepper Motor: A DC motor that divides one full rotation into a number of equally spaced steps.

References

[1] Thomas E. Kissell, “Stepper Motor Theory of Operation.” Internet: http://zone.ni.com/devzone/cda/ph/p/id/248, Sep. 6, 2006 [Nov. 25, 2012].

[2] “Arduino Uno.” Internet: http://arduino.cc/en/Main/ArduinoBoardUno, [Nov. 12, 2012].

[3] “Arduino.” Internet: http://arduino.cc, [Nov. 12, 2012].

[4] “Kinect for Windows Sensor Components and Specifications.” Internet: http://msdn.microsoft.com/en-us/library/jj131033.aspx, [Nov. 5, 2012].

[5] “JointType Enumeration.” Internet: http://msdn.microsoft.com/en-us/library/microsoft.kinect.jointtype, [Nov. 27, 2012].

[6] SGS-THOMSON Microelectronics. "Push-Pull Four Channel Driver With Diodes." L293D datasheet, [Revised 1996].

Appendix A Arduino Uno Characteristics Summary

Table A1 – Arduino Uno Characteristics Summary

Characteristic	**Value**
Microcontroller	ATmega328
Operating Voltage	5 V
Input Voltage (recommended)	7 – 12 V
Input Voltage (limits)	6 – 20 V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3 V Pin	50 mA
Flash Memory	32 KB with 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

Appendix B Arduino Output Pins Analog Voltage versus Intensity Data

The data in Table A2 was measured by attaching pin 11 on an Arduino Uno to a 10 kΩ resistor and varying the pin's analog intensity.

Table A2 – Arduino Output Pins Analog Voltage versus Intensity Data

Analog Write Intensity	**Output Voltage (V)**
255	4.86
235	4.48
215	4.1
195	3.71
175	3.33
155	2.95
135	2.57
115	2.18
95	1.8
75	1.42
55	1.04
35	0.65
15	0.27
0	0

Appendix C Microsoft Kinect Joint Types Support for Tracking

The following Microsoft Kinect Joint types are supported for tracking via the Kinect MovingJoint class:

- Left Wrist
- Right Wrist
- Left Elbow
- Right Elbow
- Left Shoulder
- Right Shoulder
- Left Knee
- Right Knee

It is possible to support other Kinect Joint types, although their implementation will require additional routines.

Appendix D .NET Project, Class and Interfaces Summary

ArduinoUtilities

- **ArduinoSerialPort**: Used to send and received packages of bytes (byte command packages) to an Arduino.
- **IComponentMapping**: An interface which represents a device attached to the Arduino.
- **PinMapping**: Represents a device attached to a single Arduino pin. Implements the IComponentMapping interface.
- **SmartPinMapping**: Contains to PinMapping objects. Represents a two-pin device (like a DC motor) attached to an Arduino.
- **StepperMotor**: Represents a stepper motor attached to an Arduino. Implements the IComponentMapping interface.

KinectUtilities

- **GestureTree**: A class which contains the relative configuration of a collection of Skynet Joints. It is used to compare a Skeleton image frame to a previous configuration, within a certain tolerance.
- **GestureTreeBuilder**: Used to build MovingGestureTrees and serialize them into XML.
- **ICapturingFunction**: An interface inherited by classes who run post-Skeleton image capture algorithms, such as a joint tracking or Skeleton rendering function.
- **IJointFrameParameter**: A package of joint data information, representing a tracking alert. Contains the joint's new angle in three-dimensional space.
- **JointController**: A class which implements the joint tracking algorithm, used for body tracking. Fires MovingJoint tracking alerts.
- **MovingGestureTree**: A class which contains a collection of GestureTrees. It is used to classify multiple Skeleton image frames into a gesture by the gesture tree body tracking algorithm.
- **MovingJoint**: A class containing a mapped joint's angle. Used for tracking alerts.
- **SkeletonController**: A class which manages all ICapturingFunctions. It receives image capture information from the SmartKinectSensor class.

Microsoft Kinect

- **Joint**: Represents a real joint in a tracked body. Contains a Kinect Joint type and the three-dimensional position of the mapped joint.
- **KinectSensor**: Used to interface with the Microsoft Kinect camera. Recieves Skeleton image frame information from the camera.
- **Skeleton**: A wrapper class containing a collection of Kinect Joints. This object represents a tracked body.

Skynet

- **Joint**: Represents a real joint in a robotic device which is mapped to an actual joint in a tracked body. This joint is represented by a single motor in the robotic device.
- **IBodyTracker**: Used to interface with the KinectUtilities class through the reception of tracking alerts. Classes inheriting this interface pass tracking alerts along into the Skynet project.
- **KinectBodyTracker**: A class inheriting the IBodyTracker interface. Functions through the reception of tracking alerts containing joint information (MovingJoints).