A Haptic Interface:
Design and Construction

by

Daniel G. McIlvaney
Bachelor of Science (Honours) with Distinction, University of Victoria, 2014

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

A Haptic Interface:
Design and Construction

by

Daniel G. McIlvaney
Bachelor of Science (Honours) with Distinction, University of Victoria, 2014

Supervisory Committee

Dr. M. Cheng, Supervisor
(Department of Computer Science)

Dr. S. Ganti, Departmental Member
(Department of Computer Science)

ABSTRACT

A haptic interface was built on top of a general purpose particle physics engine running on an Arduino Nano (ATmega328 CPU). Various tests were conducted to determine if emulated floating-point, or fixed-point arithmetic, was more performant on an 8-bit CPU with no FPU. Testing showed that 32-bit fixed-point arithmetic is required to meet the precision and range requirements of the physics engine. However, emulated floating-point calculations we also found to have very similar performance to the 32-bit fixed-point implementation. As such, the physics engine powering the haptic interface was built to work with both types. A one degree-of-freedom haptic interface was custom designed, 3D printed, and built.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Mantis Cheng for accepting me as his graduate student, always having time to suggest new topics to learn about, and letting me have fun in a lab full of neat projects.

My Parents for supporting me during both my under graduate and graduate studies.

Chapter 1

Introduction

The objectives of this project were: (1) to experiment with tangible devices (the haptic interface), and (2) to understand some of the challenges of working under heavily restrictive conditions. Developing for resource constrained systems requires many compromises and careful design decisions. This report describes the work required to implement a haptic interface based on an Arduino Nano using a 3D physics engine with collision detection, constraints, and general forces. This engine was designed to run at 20 frames per second on the Arduino Nano despite the lack of a floating point unit.

1.1 System Constraints

Developing for resource constrained systems revolves around the constraints imposed by a lack of hardware resources. The two most limiting factors for the ATmega328 are (1) the 2KB of RAM and (2) a lack of floating-point unit (FPU) [1]. The clock speed, while fairly slow compared to modern desktop systems at $16MHz$, should not be a critical limiting factor. The output of a basic renderer running over UART is expected to be actually a much more impactful issue, as discussed in Section 3.3.

1.2 Report Layout

This report covers the design and implementation of both the haptic interface and a physics engine. Chapter 2 covers the hardware and software design of the haptic interface. The interface was built on top of a custom-designed, general-purpose physics engine, which is covered in Chapter 3. Chapter 3 also discusses fixed-point versus floating-point arithmetic. Finally, Chapter 4 discusses the performance of the various components and other issues which arose during testing, as well as a discussion of the overall results.

Chapter 2

Haptic Interface

A haptic interface uses force feedback to allow a user to feel the virtual world with which they are interacting [2]. The end result of this project, a haptic interface, is shown in Figure 2.1.

2.1 Background

Haptic interfaces have been under development since the mid 50's for use in tele-operated robots [3, 4]. These robots were very inaccurate, slow and cumbersome. By the late 60's performance had improved; additional work including secondary affects such as air jets, vibrations, and moving buttons was also starting[3, 4, 5]. Haptic interfaces range from simple vibrations in phones, to full 6 degree-of-freedom interfaces. They also include surfaces which change shape and texture [2].

Modern haptic interfaces are often used to interact with virtual environments. These virtual environments require accurate simulation of collisions, momentum and forces to be believable. Modern interfaces often attempt to minimize friction to allow completely free movement of the input device by the user. A representation of the input device is then collided with the virtual environment, and the resulting forces are fed back to the user input device through motors. One way to accomplish this is by using layered objects inside the physics simulation [6]. One object, representing the input device, is allowed to move through the virtual world and collide and interact as

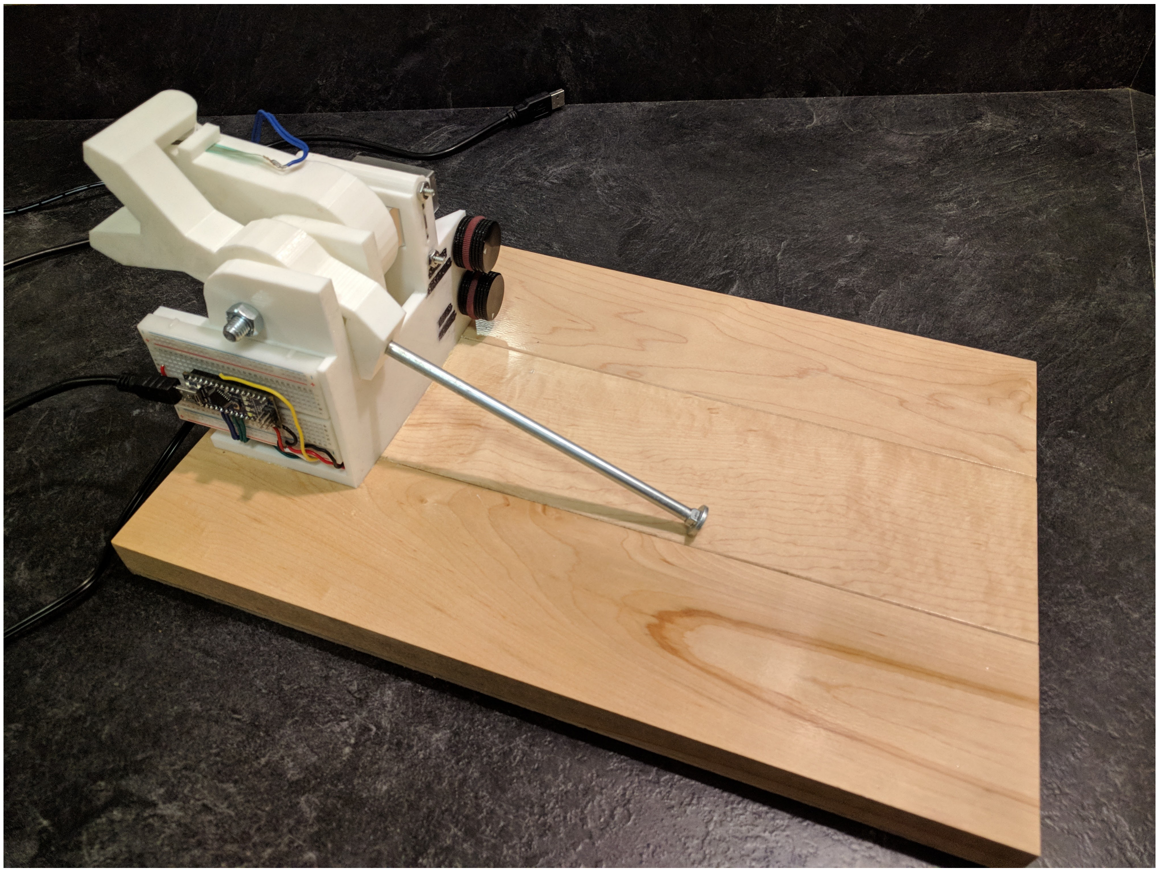

Figure 2.1: Completed haptic interface

normal. A second version of the object is coupled to the first, but cannot collide or interact with other objects. The motion of the second object is tied directly to the measured motion of the interface device. As the interface is moved, the second object is pulled away from the first. The coupling between them applies a force to the first object, moving it. However, if the first object collides with the virtual environment, it will be unable to move and the coupling-forces will increase with distance. These coupling forces are monitored, passed back, and applied to the interface device via motors.

Even though the interface device is designed to be able to move as freely as possible, when the force feedback from the coupling-force is applied the interface will experience resistance. If the motors are strong enough the interface can be completely locked in place against the force applied by the user. Usually DC motors are used; accurate current control results in accurate torque output [3].

2.2 Design Overview

The haptic interface which was built has a single degree of freedom, in the form of a small lever. One end of the lever is physical, and the other is virtual. The virtual end of the lever is connected to a weight which can be lifted using the physical end. Depending on the length of the virtual end (adjustable in real time), the mechanical advantage changes and the user will need to push harder or softer to move the lever. If the lever is released, it should flip up as the virtual weighed end falls, bouncing as it hits the ground. The virtual end of the lever is visualized in real time on a connected computer using a renderer running on the Arduino.

Unlike the more common type of haptic interface discussed above (impedance type), which focuses on minimizing the resistance of the interface handle and simply measuring its displacement[3], an alternative approach was taken. The position of the interface handle is locked in place as best as possible (dependent on the strength of the motors used), and only force measurements are taken (admittance type) [4]. These force measurements are passed to a physical device or a physics simulation which attempts to accurately predict the correct motion of the interface handle. That motion is then applied back to the handle using the motors and some sort of closed-loop positional controller. Admittance interfaces are often less accurate than impedance interfaces, but are less affected by the dynamics of the physical interface (important given the relatively cheap components and basic construction used).

The entire system, both hardware and software, was designed from scratch for this project. A haptic interface has two core components: the sensors which detect the users input (in this case the torque being applied to the lever by a user), and a motor system which provides physical feedback to the user (a servo motor). Also critical is the software which determines how the interface should behave. The software must be both physically realistic and able to run in real time.

2.3 Hardware Design

2.3.1 Torque Sensing

The first critical component of a haptic interface is user input. For a lever, the input which must be measured is torque (rotational or twisting force). When a user pushes the lever up or down they are applying torque to the lever around its pivot. While torque sensors and load cells (able to measure bending forces) are available off the shelf, they are prohibitively expensive, often costing hundreds of dollars; a more reasonable alternative was needed. A pair of force-sensitive resistors (FSRs) was used to replicate the functionality of a torque sensor. FSRs are flat resistors which decrease their resistance in proportion to how hard they are squeezed.

Two FSRs placed in opposition to each other, shown as two green ovals in Figure 2.2, can be used to calculate the net force. If a torque is applied to the outer structure (blue) relative to the inner structure (red), one of the FSRs will be squeezed harder. By comparing the respective resistances of the two FSRs, it is possible to calculate the torque being applied. The specific sensors used were Interlink FSR 402s[7]. Interlink offers a rough guide to the expected resistance curve as the force changes, but each specific implementation is different. While the sensors are reliable and have little variation, the area over which the force is applied will greatly change the resistance curve of the sensor.

To minimize backlash in the haptic interface, it is useful to have both FSRs under force at all times. To this end, double-sided foam tape was inserted between the sensors and the outer structure, as seen between the FSR and outer structure as shown in Figure 2.3. Even though both sensors will report a force, the net force can be found by subtracting one sensor's reading from the other. The FSRs and structure are also highlighted in Figure 2.3 using the same colours as Figure 2.2 so the layout of the FSRs can been more easily understood.

Because the haptic interface is connecting the physical world to the virtual world, it is important that force measurements be accurate. A mock-up (Figure 2.4) was built, consisting of just one side of the structure and the mounting bracket for the FSR. This mock-up was used to measure the voltage of the sensor, when assembled into a voltage divider with a $10K\Omega$ resistor and connected to an analogue input

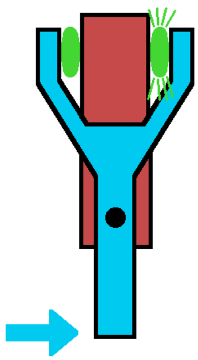

Figure 2.2: FSR Layout

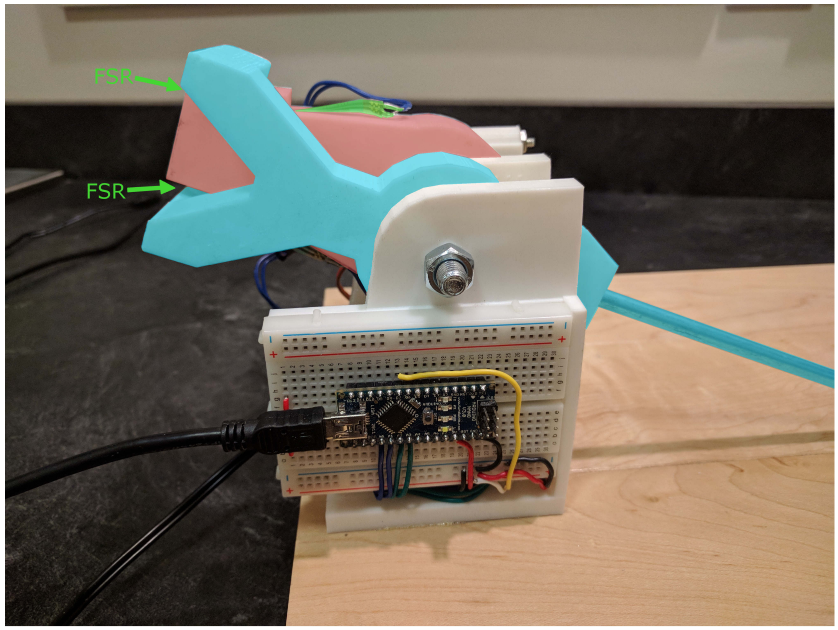

Figure 2.3: Actual FSR positioning

port on the Arduino. Different forces were applied to the assembly and measured using a weighing scale; the resulting analogue voltage readings were recorded. A wiring diagram is included Appendix C, showing how the FSRs were connected to the Arduino in the final design.

These analogue values were entered into an online curve fitting program[8] to generate a function to convert any analog reading (0-1023) into a force in grams, as read from the scale. Function 1: $f(x) = x/(-1.30315751 \times 10^{-5}x^2 + 1.063940729 \times 10^{-2}x + 2.30339406)$ was selected, as it gave fairly good results (as determined by least squares). Other families of curves had better fits, but required operations which would be costly to calculate (logarithms, exponentiation etc.). Table D.1 in the Appendix shows the initial measurements, and summarizes the results of Function 1. The formula used was fit to the data points from 850 and up. A second curve (Function 2: $f(x) = 8.536426146 \times 10^{-4}x^2 - 0.2957452506x + 55.22287182$) was calculated for readings below 850. However only Function 1 was used. Its accuracy is not excellent

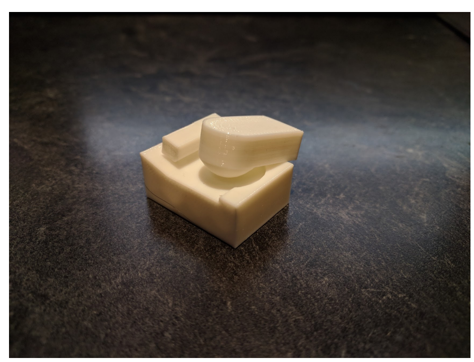

Figure 2.4: FSR mock-up

for lower input values, but this is not an issue since the two FSRs are always under force from the foam tape and will rarely, if ever, see those forces.

2.3.2 Force Feedback

The second critical component of a haptic interface is force feedback, which allows the user to feel the virtual world as it responds to their physical input. In this case, the lever must act as if it had a weight on the its far end; i.e. flipping upwards as the virtual weight falls and resisting a user trying to push it back down. Because of the foam inserts placed between the lever and the FSRs, there is a direct relationship between the angle of the lever relative to the sensor structure and the force applied to the lever. A servo motor can be used to create various forces by changing that angle. At 0° difference, the net force being applied to the lever by the two foam squares is 0 N. As the angle increases, one of the foam squares is crushed and the net force increases in the opposite direction (Section 4.2.1 has more discussion about the trade-offs of this approach).

Originally a basic 5V servo motor was used (TowerPro SG-5010[9]); however it proved to be far too weak to give a good result. Due to the small physical size of the haptic interface, the virtual world is correspondingly also small; this includes the weight at the end of the lever. At this scale, objects move relatively quickly; most importantly the angle of the lever changes very quickly. Consider a very long lever and a very short lever. If both levers are raised so that their weighted ends are angled upwards 45°, the short lever's weighted end will be much closer to the ground (As shown in Figure 2.5). Due to the nature of gravity, both weights will accelerate towards the ground at nearly the same rate (slightly effected by the current angle of the lever), but the longer lever's weight has much farther to fall. This results in the change in angle taking much longer for the long lever, and thus a much lower speed of rotation.

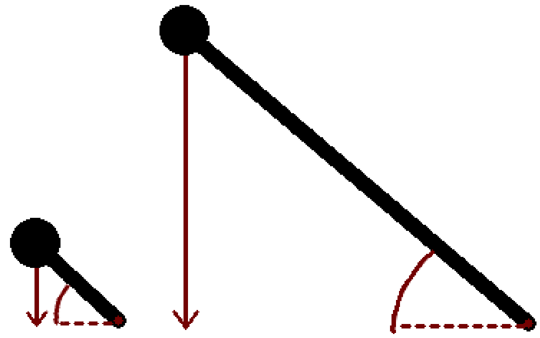

Figure 2.5: Lever length considerations

The SG-5010 was completely unable to keep up with the rotation rates needed to model a short (20 cm) lever. In theory it should have been able to rotate the 65° needed to move the lever its full range in slightly over 0.2 s (0.2 s/60° at 4.8 V), which closely matches the time a 20 cm lever should take to fall (0.18 s). Unfortunately the SG-5010's meagre torque of 5.5 kg cm meant that in the real world it was much too slow. The motor was not able to meet the design speed even with no load, and especially not with a large load such as a human hand pushing it the other way.

A significantly more powerful servo motor was required. The RJXHobby FS0521HV [10] was selected as a suitable replacement. It was designed to cover the same distance as the SG-5010 in one quarter the time (0.05 s), with a torque of 21.3 kg cm. A more powerful power source was required to drive this motor so a 7.5 V, 3.2 A wall adapter was used instead of the 5 V breadboard power supply used with the SG-5010.

The new motor was able to swing the arm extremely violently, exactly what was needed to model small, fast moving objects. The mounting structure is shown in Figures 2.6 and 2.7. Figure 2.7 also shows the bearings used to hold the lever (green) and sensor mounting (grey). The main bearing rod is a 8mm bolt, with washers and 8mm skateboard bearings (yellow) holding the components in place.

Several alternative methods of providing force feedback were considered. The first option would have involved placing a 2:1 or 3:1 step-down gearbox between the servo motor and the sensor structure. The sensor structure only rotates 60°, while the servo motor is able to rotate 180°. This means that the servo motor could be made two times more accurate and stronger. The gearbox would reduce the maximum speed of the arm but the FS0521HV is fast enough to compensate. The gearbox would have to be built to very tight tolerances since backlash is quite undesirable in this application.

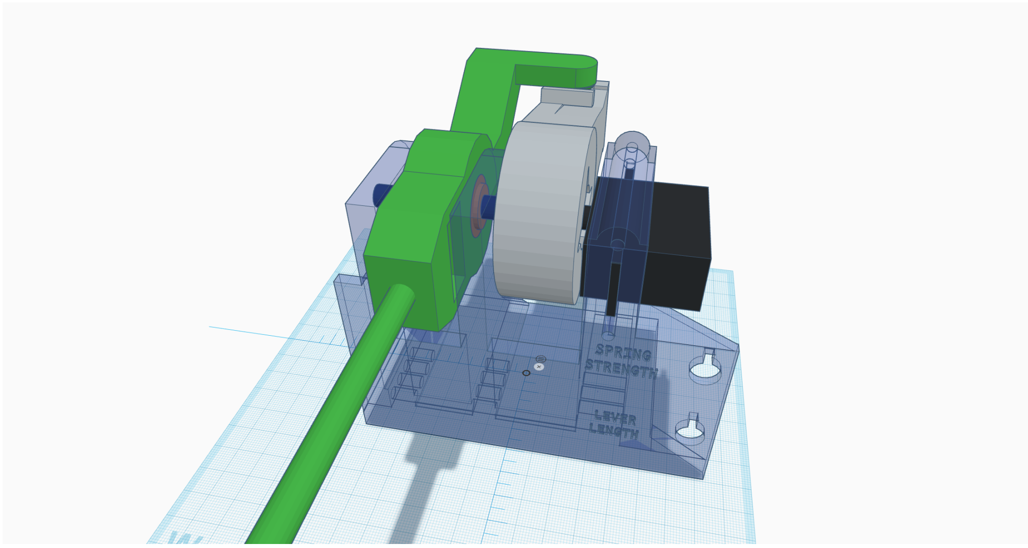

Figure 2.6: Servo motor mounting structure

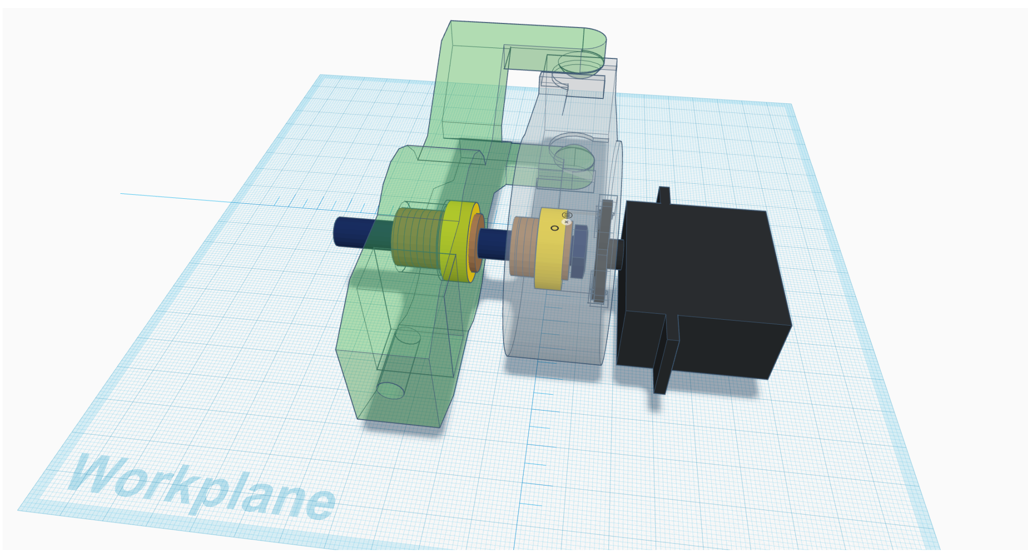

Figure 2.7: Bearings and spacers for the arm and sensor structure

The second option which was considered was a simple DC motor and separate encoder, instead of a servo motor. The encoder would be used to determine the position of the arm, while a proportional-integral-derivative (PID) controller would be responsible for regulating the voltage provided to the motor to keep it there. There is a relationship between the voltage provided to a motor and the torque it produces [3]. Measuring the voltage could provide an alternative to the FSRs, or alternatively the voltage could be carefully controlled to provide very accurate force feedback.

Both options significantly increase the complexity of the design, structural for the first option, and electrical and programming for the second. Because the basic implementation worked well enough it was left in place.

2.3.3 Structure

The entire structure of the haptic interface was custom designed and 3D-printed in PLA plastic, specifically for this project (Figure 2.8). All components were printed on a Makerbot Replicator 2 in PLA with 30% infill, but otherwise normal settings.

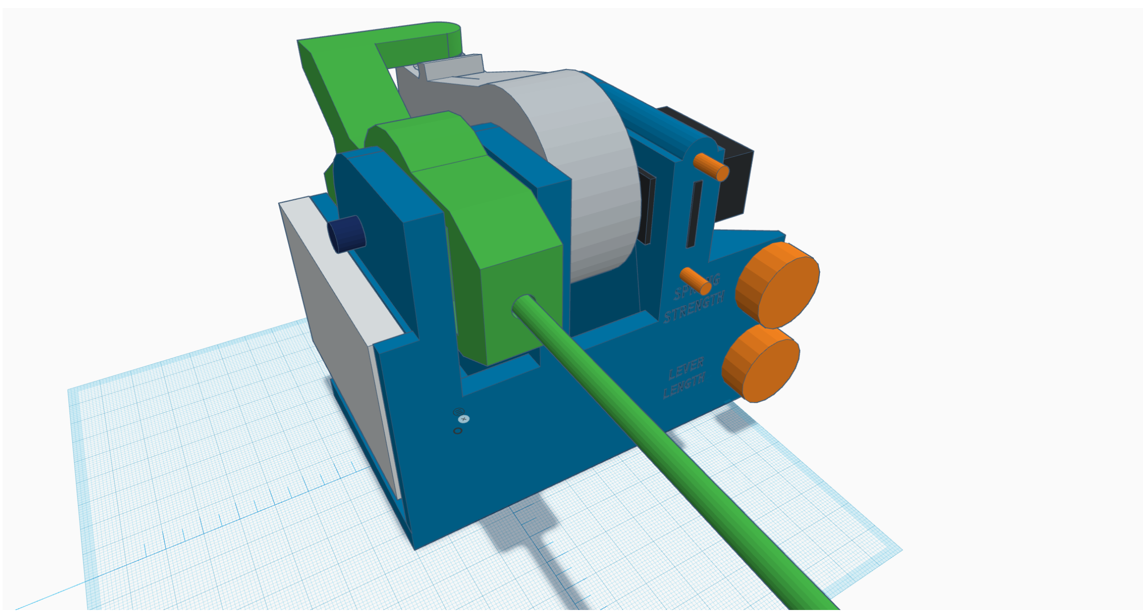

Figure 2.8: Tinkercad model with all components included

A basic online 3D modelling package (Tinkercad from Autodesk) was used for this purpose. While Tinkercad lacks advanced features, it it very quick and easy to use, and includes the option to directly enter measurements for dimensions, which is critical for modelling real-world components. Surfaces to mount two breadboards were included in the design, along with conduits for running wires. A control panel with labels was also added to the front panel.

2.4 Software Implementation

Writing code to accurately simulate a single lever with a weight is not particularly challenging for anyone with a basic understanding of physics; however, bespoke code makes changing the design of the system difficult. While a single lever is relatively easy, adding a second pendulum to the end of the lever would require a complete re-write; adding a spring or air resistance would again require modifications. A general-purpose physics engine, as will be described in Chapter 3, was developed to run the virtual side of the haptic interface. Because a general-purpose physics engine was used, the haptic interface is malleable and can be changed relatively easily to simulate different environments.

Because the physics engine handles most of the math, the code which drives the haptic interface is extremely short, as will be seen in Code Blocks 2.1 and 2.2. Because the complex simulation code is already included in the physics engine, all that is required is to configure the engine to represent the desired setup, and update the sensor data and motors each frame.

2.4.1 Physics Engine Configuration

The most basic and reliable setup for the haptic interface includes a simple lever and weight (Shown in Code Block 2.1) - other options which were tested include the springs and pendulums described above. Configuring the physics engine involves adding two objects to the world to form the lever: a pivot and the rod end. A rod constraint is added to force the ends to remain at a fixed distance, while the pivot is given infinite mass (so it can't be moved). A gravity force is added to the rod end, and the lever is ready. (Refer to Chapter 3 for details about the physics engine implementation)

To connect the user to the lever, a `CustomForceGenerator` (force generators are described in Section 3.2.4) is added, which reads a user-generated force vector into the physics engine via a pointer and applies it to the object each frame update. Each frame the user-generated force vector is updated from the FSRs before the physics engine runs.

```
void setup() {
  Serial.begin(250000);
  leverServo.attach(7);
  initSimulationEngine();

  pivotObj = simulationGetFreeObject();
  pivotObj->m_inUse = true;
  pivotObj->m_objectType = PARTICLE;
  //Location: .38m, 0.15m, 0m
  pivotObj->m_position = Vector3D(FROM_INT_SHIFT(38,2),
        FROM_INT_SHIFT(15,2), FROM_INT(0));
  pivotObj->m_velocity = Vector3D(0,0,0);
  pivotObj->m_invMass = 0;
  pivotObj->m_particleData.m_radius = ZERO;

  rodEnd = simulationGetFreeObject();
  rodEnd->m_inUse = true;
  rodEnd->m_objectType = PARTICLE;
  rodEnd->m_position = Vector3D(FROM_INT_SHIFT(1,1),
        FROM_INT_SHIFT(0,2), FROM_INT(0));
  rodEnd->m_velocity = Vector3D(0,0,0);
  //0.1 kg
  rodEnd->m_invMass = DIV(FROM_INT(1),FROM_INT_SHIFT(1,1));
  rodEnd->m_particleData.m_radius = FROM_INT_SHIFT(1,2);

  buildGravityForce(simulationGetFreeForce(), rodEnd);
  leverRodConstraint = simulationGetFreeConstraint();
  buildRodConstraint(leverRodConstraint, pivotObj,
        rodEnd, FROM_INT_SHIFT(15,2));
  buildCustomForce(simulationGetFreeForce(), rodEnd, &analogueForce);
}
```

Code Block 2.1: Haptic setup

The main loop seen in Code Block 2.2 is also fairly straight forward. A vector representing the lever is calculated, based on the current state of the physics simulation. The FSRs are then checked to determine the current torque being applied to the lever. The readings are averaged with previous readings to help smooth out any noise. The current length of the virtual lever is then checked. The torque is then converted to a force which will be applied to the end of the rod .

```
#define FSR_MEASUREMENTS 2
FixedPoint fsrReadings[FSR_MEASUREMENTS];
int currentFsrReading = FSR_MEASUREMENTS + 1;
void loop() {
  Vector3D lever = calculateLeverVector();
  fsrReadings[(currentFsrReading++) % FSR_MEASUREMENTS] = FSRCalc(analogRead(
      A_DOWN)) - FSRCalc(analogRead(A_UP));
  FixedPoint averageFsrReading = 0;
  for(int i = 0; i < FSR_MEASUREMENTS; i++) {
    averageFsrReading += fsrReadings[i];
  }

  FixedPoint torque = DIV(averageFsrReading,FROM_INT(FSR_MEASUREMENTS));
  analogueForce = calculateLeverForceVector(lever) * torque;
  //Sensors are 4cm from the pivot
  FixedPoint mechAdvantage = DIV(FROM_INT_SHIFT(4,2),lever.magnitude());
  analogueForce *= mechAdvantage;

  FixedPoint leverAngle = calculateLeverAngle(lever);
  if((leverAngle > FROM_INT(60) && rodEnd->m_velocity.m_y > 0) || (leverAngle <
      FROM_INT(0) && rodEnd->m_velocity.m_y < 0)) {
    rodEnd->m_velocity*= FROM_INT_SHIFT(-5,1);
  }
  moveServo(leverAngle);

  leverRodConstraint->m_rodData.m_length = calculateLeverLength();

  //Damp oscillations
  Vector3D accelDirection = rodEnd->m_velocity;
  accelDirection.normalize();
  FixedPoint coincidentAcceleration = analogueForce * accelDirection;
  FixedPoint threshold = FROM_INT_SHIFT(5,1);
  if(coincidentAcceleration < -threshold) {
    coincidentAcceleration *= -1;
    if(coincidentAcceleration < threshold) {
      coincidentAcceleration = threshold;
    }
    FixedPoint dampingMultiplier = DIV(threshold,coincidentAcceleration);
    rodEnd->m_velocity *= dampingMultiplier;
  }

  stepSim();
}
```

Code Block 2.2: Haptic main loop

Torque, aka moment of force, is defined as $\vec{\tau} = \vec{r} \times \vec{F}$. Increasing the lever vector $\vec{r}$, while leaving the force vector $\vec{F}$ unchanged, increases the torque. It is worth noting that only the perpendicular component of the force vector $\vec{F}$ affects the torque. Due to the nature of how the FSRs are used, only the perpendicular force is detected, so simple scalar math can be used.

It is necessary to convert the torque back into a force which can be applied to the lever end object. To do this, the mechanical advantage of the lever needs to be calculated. The length of the physical lever is fixed, as is the distance of the FSRs from the pivot (4cm). Based on the ratio of distance between the sensors and the pivot, and the current length of the virtual rod, the mechanical advantage can be determined ($mechanical\ advantage = sensor\ distance \div virtual\ lever\ length$).

The force vector used by the physics engine is then updated so it is perpendicular to the lever and ready for use. The $atan2()$ code (discussed in Section 3.2.1) is used to determine what angle the lever is at relative to the x-axis. The servo motor is then commanded to move the physical lever to match the virtual lever.

Several simple functions (1) calculate the lever vector (`calculateLeverVector()`), (2) determine the vector perpendicular to the lever along with the forces from the user should be applied (`calculateLeverForceVector()`), and (3) convert the lever vector into an angle in degrees using $atan2()$ for use with the servo motor (`calculateLeverAngle()`). The range of motion of the lever is also constrained to that of the hardware device (65 deg), with a bounce of 50% applied to the lever should it hit the end of its range.

A reading is also taken from one of the front dials to set the length of the virtual lever between 23cm and 48cm. This range was selected to minimize undesirable behaviours: too short and the lever begins to oscillate uncontrollably, too long and the lever becomes too difficult for the servo motor to simulate without stalling. The servo motor must have enough torque to both match that generated by gravity and overcome the force of the user. The motor is designed to output 21.3kg-cm, which translates to 0.5kg at 40 cm. The simulation only has a 0.1kg mass at the end of the lever, but if the lever is moving and the user tries to stop it quickly, the torque can quickly exceed what the motor can produce. The motor is also likely not performing at peak capacity due to limited power - power supplies in the 7.5V to 8.5V range are not common, and 3.2A was the highest current level available.

Finally, just before the physics engine performs its frame calculations, a section of code which is discussed in detail in Section 4.2.2 deals with some special damping to reduce unwanted oscillations.

Chapter 3

General Purpose Physics Engine

3.1 Rationale

As discussed in Chapter 2, bespoke code to run the haptic interface would have been much simpler to implement. Instead, a full blown 3D particle physics engine was developed. The benefits of this approach are two fold: the implementation of different projects becomes much easier, and improvements made to the physics engine can be easily applied across everything which relies on it. The physics engine which was developed has a basic set of capabilities, suitable for simple projects like a simple flight simulator, or for small games. The interface is fairly simple; the *game logic* (the code written on top of the physics engine) simply needs to call `stepSim()` each time it wants to step the simulation forward.

The physics engine maintains a statically-allocated pool of various structures to represent game objects, forces and constraints. The game logic can create fire and forget forces by simply calling something like `buildGravityForce( simulationGetFreeForce(), rodEnd );`. A pointer to an object can be kept if that object needs to be modified or monitored such as: `leverRodConstraint = simulationGetFreeConstraint(); buildRodConstraint( leverRodConstraint, pivotObj, rodEnd, FROM_INT_SHIFT(15,2) );`. More interesting game logic can be built on top of these structures. The objects stored by the physics engine can be updated via the pointer; or, information like position or velocity can be read back out.

3.2 Physics Engine Implementation Details

While all Arduino projects in the Arduino IDE are compiled into C++, large parts of the physics engine code was written to be C compatible. The project was originally written for compilation in C due to performance concerns which did not materialize. Thus, the decision was made to switch to C++ due to ease of implementing a new Vector class. The physics engine as currently implemented only handles particles of various sizes, but was designed to be modifiable to handle other shapes. These particles are able to bounce off of each other, react to various forces, and be constrained to each other or the environment.

3.2.1 Fixed-Point Arithmetic

Because the ATmega328 does not have a FPU, all floating-point arithmetic must be emulated in code. While highly optimized, this code is much slower than integer math. Fixed-point arithmetic is an alternative which uses integers to represent decimal values. A simple way to explain fixed-point arithmetic is to consider the SI unit system. The physics engine uses metres to measure distances. Instead of rounding to the nearest metre, it is a simple matter to multiply all distances by 1000 and round to the nearest millimetre instead. Fixed-point arithmetic takes an integer and splits off some number of bits to store the integer component; the remaining bits store the decimal component.

Details

A custom fixed-point implementation was created from scratch for use in the physics engine. The ATmega328 is an 8-bit CPU, but the compiler is able to generate code to emulate up to 64-bit integers with reasonable performance. Four sizes of fixed-point numbers are available in the current implementation: 4:4 (4 bits for the integer, 4 bits for the decimal), 8:8, 16:16, and 32:32. Anything less than 32 bits (16:16) proved to be unsuitable for the physics engine, which created issues for both performance and memory footprint.

Converting values to `FixedPoint` is fairly straightforward. The `FixedPoint` data type used is an `int32_t`. To convert an integer to a `FixedPoint`, the integer is simply left shifted by 16 bits. A special version is provided to allow easy input of decimal values by repeatedly dividing an integer by 10, to position the decimal point as seen in Code Block 3.1.

```
#define FixedPoint int32_t
#define FixedPointLarge int64_t
#define SHIFT_VAL 16
#define SQRT_SHIFT_VAL 8
#define FP_MAX 0x7FFFFFFF
#define FP_2PI 411774
#define NUM_DECIMAL_PLACES 4

#define FROM_INT(a) (((FixedPoint)a) << SHIFT_VAL)
#define FROM_INT_SHIFT(a,decimalShift) (fp_fromIntShift(a,decimalShift))

inline FixedPoint fp_fromIntShift(FixedPoint a, FixedPoint decimalShift) {
  int shift = 1;
  for(int i = 0; i < decimalShift; i++) {
    shift *= 10;
  }
  return DIV(FROM_INT(a),FROM_INT(shift));
}
```

Code Block 3.1: FixedPoint basics

Addition and subtraction are also very straight forward, and simply rely on the existing arithmetic operators for `int32_t`s.

Unfortunately, multiplication and division present a more substantial challenge. Consider $1 \times 1 = 1$ and $1 \div 1 = 1$: Say we change this to a fixed-point implementation where all values are multiplied by 1000. We now have $1000 \times 1000 = 1000000$ and $1000 \div 1000 = 1$. The expected result for both is 1000 (our representation of 1), but $1000000 \neq 1000 \neq 1$! All multiplications must be shifted back to the right, and all divisions must be shifted to the left.

This presents an issue. Since it is possible that two medium sized numbers will be multiplied together, and even though the end result will not overflow a 32-bit integer, the intermediate value, before it is shifted back, will overflow. There are two possible solutions to this: either (1) accept the loss of data and shift the parameters to the right before the multiplication such that the final output will be correct, or (2) store the intermediate result as a 64-bit value until it is shifted back.

Since physics engines are susceptible to numerical instabilities at the best of times, because they are built around the idea of repeated integration, the loss of precision was not acceptable. As seen in Code Block 3.2, a 64-bit integer is used to store the intermediate values of both multiplications and divisions.

```
#define MULT(a,b) fp_multiply(a, b)
#define DIV(a,b) fp_divide(a, b)

inline FixedPoint fp_multiply(FixedPoint m1, FixedPoint m2) {
  int signFlip = 1;
  if (m1 < 0) {
    m1 *= -1;
    signFlip = -1;
  }
  if (m2 < 0) {
    m2 *= -1;
    signFlip *= -1;
  }
  return (FixedPoint)((((FixedPointLarge)m1) * ((FixedPointLarge)m2)) >>
      SHIFT_VAL) * signFlip;
}

inline FixedPoint fp_divide( FixedPoint num, FixedPoint denom) {
  int signFlip = 1;
  if (num < 0) {
    num *= -1;
    signFlip = -1;
  }
  if (denom < 0) {
    denom *= -1;
    signFlip *= -1;
  }
  return (FixedPoint)(((FixedPointLarge)(num) << SHIFT_VAL) / denom) * signFlip;
}
```

Code Block 3.2: FixedPoint multiplication and division

The program behaviour when shifting negative integers is not defined as part of the C or C++ language and is open to interpretation. In the interest of compatibility, all fixed-point numbers are changed to positive during multiplication or division. The overhead from this is insignificant relative to the cost of the multiplication or division.

Functions for both calculating square roots and arc-tangents were necessary for the physics engine and final haptic interface project, as shown in Code Block 3.3. Square roots were calculated based on an algorithm described by Turkowski[11]. The square root function is used in a number of places inside the physics engine, when calculating the magnitude of vectors, or normalizing them. Unfortunately, the algorithm cannot come close to matching the performance of floating-point implementations. When working with 32-bit `FixedPoint` the algorithm takes approximately 10 times as long at 12571ms for 30000 operations (0.4ms per operation) versus 1213ms (0.04ms) for the standard floating-point implementation.

```
FixedPoint fp_sRoot(FixedPoint num) {
  uint32_t root, remHi, remLo, testDiv;
  root = 0;
  remHi = 0;
  remLo = num;
  int count = 15 + SQRT_SHIFT_VAL;
  do {
    remHi = (remHi << 2) | (remLo >> 30); remLo <<=2;
    root <<= 1;
    testDiv = (root <<1) + 1;
    if(remHi >= testDiv) {
      remHi -= testDiv;
      root += 1;
    }
  } while (count-- != 0);
  return root;
}

const int64_t arctan_lookup_table[] PROGMEM = {0,21474657, ... ,3373259426,};

//Only works from 0 to pi/4.
FixedPoint fp_arctan_lookup(FixedPoint x) {
  if(x > ONE || x < 0) {
    Serial.println("Arctan out of bounds!");
    for(;;);
  }
  const FixedPoint lookupX = MULT(FROM_INT(200), x);
  int i = TO_INT(lookupX);
  int64_t n1;
  memcpy_P (&n1, &arctan_lookup_table[i], sizeof(int64_t));
  int i2 = i + 1;
  if (i2 > 200) {
    return (FixedPoint)(n1 >> (32 - SHIFT_VAL));
  }
  int64_t n2;
  memcpy_P (&n2, &arctan_lookup_table[i+1], sizeof(int64_t));
  FixedPoint n1Fp = (FixedPoint)(n1 >> (32 - SHIFT_VAL));
  FixedPoint n2Fp = (FixedPoint)(n2 >> (32 - SHIFT_VAL));
  FixedPoint diffFromTableEntry = lookupX - FROM_INT(i);
  return n1Fp + MULT(n2Fp - n1Fp, diffFromTableEntry);
}

//Returns number of radians counter clockwise from x positive axis.
FixedPoint fp_arctangent2(FixedPoint x, FixedPoint y) {
  FixedPoint temp;
  FixedPoint offset = 0;
```

```
  if(y < 0) {
    //Flipping above y=0
    x *= -1;
    y *= -1;
    offset += 4;
  }
  if(x<=0) {
    //Flipping to the right
    temp = x;
    x = y;
    y = -temp;
    offset += 2;
  }
  if(x <= y) {
    //Move to lower half
    temp = y - x;
    x = x+y;
    y = temp;
    offset += 1;

  }
  FixedPoint at = fp_arctan_lookup(DIV(y,x));
  return at + (FP_PI>>2)*offset;
}
```

Code Block 3.3: Advanced FixedPoint functions

The arc-tangent and arc-tangent2 functions were implemented using a lookup table. The `fp_arctangent2` function converts x and y values to an angle for use in the single argument `fp_arctan_lookup` function, using an algorithm described by Jasper Vijn on his website[12]. The result can then be read from a table of 64-bit `FixedPoint` values stored in program memory. The lookup table outperforms the floating point implementation, taking only 75% as long at 3626ms for 30000 operations (0.12ms per operation). (Table 3.2 in Section 3.2.1 shows timing information for basic fixed-point operations)

Accuracy

The largest positive value a (16:16) fixed-point can hold is 32768.0, and the smallest is 0.000015. The smallest value is misleading because fixed-point does not work using powers of 10 but instead is binary - the second smallest positive value is 0.000031, then 0.000046 and so on. Table 3.1 shows the accuracy of the various sizes of data type.

$$01111111111111111111111111111111$$

$$2147483647 \div (1 << 16) = 32768.0$$

$$00000000000000000000000000000001$$

$$1 \div (1 << 16) = 0.0000152588$$

FixedPoint Size	Max Positive Value	Min Positive Value	Largest Squarable Number
4:4	7	0.0625	2.6
8:8	127	0.00390625	11.3
16:16	32767	0.0000152588	181
32:32	2147483647	0.000000000232831	46341

Table 3.1: Accuracy of various fixed-point sizes

Of some concern is the maximum positive size. There are a number of points in the physics engine where integers must be squared (for example, finding the magnitude of a vector by using the Pythagorean theorem), and it is not difficult to overflow when squaring numbers.

32 bits ended up being sufficient for the purposes of the physics engine. While the 16:16 split gives a good mix of accuracy and range for this use case, the bits do not need to be partitioned evenly; the same `FixedPoint` implementation could be used with 20:12, or 29:3, etc., just by changing the shift values.

Performance

Testing was done before development of the physics engine was undertaken to find the costs of various operations for integer, floating-point, and fixed-point numbers. As shown in Table 3.2, initial testing showed integer arithmetic with a strong lead over floating-point at both 16- and 32-bits, with the exception of division. The reduced performance of the division operation was considered the be acceptable given the performance gains in other areas. Division operations are not exceptionally common in the physics engine so this would not result in a significant impact on performance. There were no memory savings for 32-bit fixed-point however since all floating point numbers on Arduino are 32 bits [13].

Type	Duration of test (ms)			
	Add	Subtract	Multiply	Divide
`int16_t`	44	34	63	461
`int32_t`	84	60	226	1230
`int64_t`	214	202	831	2125
`FixedPoint(8:8)`	99	67	322	1417
`FixedPoint(16:16)`	137	138	841	3605
`Float`	272	232	448	433

Table 3.2: Milliseconds per 30000 arithmetic operations on different data types

Unfortunately, as discussed above, `FixedPoint` values suffer loss of data during division and multiplication unless they are cast into larger data types. Once this cast was included in the `FixedPoint` library, the performance suffered significantly, being even worse than the corresponding integer operation would suggest. While shifting a 64-bit integer 30000 times takes 356ms, 17% of the time for a full division, this should only give a time of approximately 2480ms for a 64-bit division, much less than the 3605ms actual result. The additional overhead of the division function is shared with the multiplication function, which only showed a very modest decrease in performance compared to the integer operation. Since multiplication performance did not change significantly, the cast from `int32_t` to `int64_t` may not be the issue. This value could not be decreased even with very aggressive compiler optimization. Either there is some other source of performance loss, or the multiplication code can be more easily optimized by the compiler. The real world performance of he `FixedPoint` multiplication and division implementations is measured in Section 4.1. The timing values for multiplication are validated, but the cost of division operations is half what was measured here. Both implementations used the same `FixedPoint` library and optimization options, and the cause of the discrepancy could not be found.

Regardless of the source of the slowdown, 32-bit division is much more palatable than 64-bit division. It is possible to calculate how much a number can be safely shifted without loss of data if a Most Significant Bit (MSB) operator is available. Doing this, it should be possible to create a division function which only casts to 64-bit if data will be lost. The left shift is currently applied to only the divisor, but it could just as easily be spread among the divisor, dividend (right shift) and quotient to avoid switching to 64-bit. This could be made even more flexible if a small loss of data was permissible.

In the interests of experimentation and further learning the decision was made to continue using the 16:16 `FixedPoint` despite the reduced performance. Because the `FixedPoint` logic is kept in a separate file it is very easy to swap the physics engine to using `Floats`.

3.2.2 Vector Class

A vector struct was created early on which was compatible with C; and, a set of functions were written to calculate addition, subtraction, the dot product, and other values based on these structs. This proved to be very difficult to work with, so a `Vector3D` class with methods was created for use with C++ to test performance as shown in Code Block 3.4. The `Vector3D` class with methods ended up being just as fast as the basic struct with global functions, so the decision was made to switch to C++ as there seemed to be no significant benefit to avoiding simple classes.

```
struct Vector3D
{
  FixedPoint m_x = 0;
  FixedPoint m_y = 0;
  FixedPoint m_z = 0;

  Vector3D()
    : m_x(0),
      m_y(0),
      m_z(0) {
  }

  Vector3D(FixedPoint x, FixedPoint y, FixedPoint z)
    : m_x(x),
      m_y(y),
      m_z(z) {
  }
  ...
```

Code Block 3.4: Vector3D implementation

Game Physics Engine Development by Ian Millington[14] describes a Vector class which is very similar. Later in development, some of the differences, such as more user-friendly operator overloading, were included in the `Vector3D` class to make it more useful.

The `Vector3D` class includes methods to calculate the magnitude, the magnitude squared (useful for comparisons since it is much faster to square the other value than to find the square root), and to normalize the vector. These are shown in Code Block 3.5.

```
...
FixedPoint magnitude() const {
  return fp_sRoot(MULT(m_x, m_x) + MULT(m_y, m_y)  + MULT(m_z, m_z) );
}
FixedPoint magnitude2() const {
  return MULT(m_x, m_x) + MULT(m_y, m_y)  + MULT(m_z, m_z);
}
void normalize() {
  FixedPoint mag = magnitude();
  if (mag != 0) {
    m_x = DIV(m_x, mag);
    m_y = DIV(m_y, mag);
    m_z = DIV(m_z, mag);
  }
}
...
```

Code Block 3.5: Vector3D methods

The `Vector3D` class also includes many simple overloads of the normal arithmetic operators to make it easier to work with as shown in Code Blocks 3.6 and 3.7.

```
...
  //  ***(scalar)
  void operator*=(const FixedPoint scal) {
      m_x = MULT(m_x, scal);
      m_y = MULT(m_y, scal);
      m_z = MULT(m_z, scal);
  }
  Vector3D operator*(const FixedPoint scal) const {
    return Vector3D(MULT(m_x, scal), MULT(m_y, scal), MULT(m_z, scal));
  }
  //  ***(vector) - Dot product
  Vector3D componentProduct(const Vector3D& vector) const {
    return Vector3D(MULT(m_x, vector.m_x), MULT(m_y, vector.m_y), MULT(m_z,
        vector.m_z));
  }
  FixedPoint operator*(const Vector3D &vector) const {
    return MULT(m_x, vector.m_x) + MULT(m_y, vector.m_y) + MULT(m_z, vector.m_z)
        ;
  }
...
```

Code Block 3.6: Vector3D operator overloading with FixedPoint

```
...
  //  %%% - Cross product
  Vector3D operator%(const Vector3D& vector) const {
    return Vector3D( MULT(m_y, vector.m_z) - MULT(m_z, vector.m_y), MULT(m_z,
        vector.m_x) - MULT(m_x, vector.m_z), MULT(m_x, vector.m_y) - MULT(m_y,
        vector.m_x));
  }
  void operator%=(const Vector3D& vector) {
    *this = (*this)%vector;
  }

  //  +++
  void operator+=(const Vector3D& vector) {
    m_x += vector.m_x;
    m_y += vector.m_y;
    m_z += vector.m_z;
  }
  Vector3D operator+(const Vector3D& vector) const {
    return Vector3D(m_x + vector.m_x, m_y + vector.m_y, m_z + vector.m_z);
  }
  void addScaledVector(const Vector3D& vector, FixedPoint scal) {
    m_x += MULT(vector.m_x, scal);
    m_y += MULT(vector.m_y, scal);
    m_z += MULT(vector.m_z, scal);
  }

  //  ---
  void operator-=(const Vector3D& vector) {
    m_x -= vector.m_x;
    m_y -= vector.m_y;
    m_z -= vector.m_z;
  }
  Vector3D operator-(const Vector3D& vector) const {
    return Vector3D(m_x - vector.m_x, m_y - vector.m_y, m_z - vector.m_z);
  }
```

Code Block 3.7: Vector3D operator overloading with Vector3D

3.2.3 The Core (Integrator)

At the core of the physics engine is the integrator. Each entity in the simulated world experiences various forces. These forces then act on the entities to create an acceleration inversely proportional to the mass of the entity ($f = ma$, from high school physics). This acceleration then causes a change in velocity proportional to time Δt. Finally, the velocity changes the location of the entity, also proportional to Δt. These changes are described by the basic kinematic equations familiar to anyone who has studied basic physics.

The physics engine runs in a series of steps, ideally quite small. The time since the last simulation update is the Δt. Since velocity is the rate of change in displacement (derivative), and acceleration is the derivative of velocity, it is possible to calculate the next location for each particle by working backwards using integration (the physics engine assumes a jerk of 0, changes to acceleration are achieved via changing forces between steps).

This core integration loop, while re-written several times, has remained functionally unchanged from the very early versions of the engine which were built from first principals and without external references.

Engine Versions 1 and 2

The initial engine implementation included rotational velocity. Rotational physics can be processed the same way as the kinematic motions, using integration at each step. The only difference is the idea of a moment of inertia: how easy a mass is to spin given a torque. Fairly early on, this first version of the physics engine became unwieldy and was taking too much memory. Each object in the simulation was requiring seven `FixedPoint`s and three `Vector3D`s to keep track of basic motion, for a combined total of 64 bytes. The final version of the object structure ended up taking only 52 bytes (there are an additional 16 bytes of information describing the object's attributes which would have been common to both).

Version 1 was also designed with support for various basic geometric shapes. However the collision detection and resolution code became overly complex for the resources available, while also adding unneeded code complexity. Version 2 was simplified to use only points (referred to as `Particle`s) with the ability to set a radius for collision purposes. These particles act like spheres with no surface friction or rotation. While undeniably more limited in scope, the collision resolution was made much more manageable.

A basic physics engine was fleshed out from this point. However collision detection and resolution were still not reliable. A rewrite was undertaken using ideas from Millington [14] and with reference to *Real-Time Collision Detection* by Christer Ericson [15]. In his book Millington describes a C++ physics engine with heavy use of classes and inheritance, which impose too much overhead given the extreme lack of memory available. Regardless, the ideas presented were applicable, with some modification, to the Arduino physics engine, and the collision detection system was rebuilt based on the concepts from the book.

The final version of the integrator shown in Code Block 3.8 is quite straightforward and is called for all active objects. Since the mass of an object is most often used as a denominator in divisions, it is quicker to store the inverse mass and multiply by that number instead.

```
void integrateObject(Object &obj, const FixedPoint& timeDelta) {
  if(obj.m_invMass <= 0) {
    return;
  }
  switch (obj.m_objectType) {
    case PARTICLE:
      particleIntegrate(obj, timeDelta);
      break;
    default:
      break;
  }
  //Clear all forces.
  obj.m_force = Vector3D(0,0,0);
}

...

void particleIntegrate(Object& obj, const FixedPoint& timeDelta) {
  obj.m_position.addScaledVector(obj.m_velocity, timeDelta);

  Vector3D forceAcceleration = obj.m_acceleration;
  forceAcceleration.addScaledVector(obj.m_force, obj.m_invMass);

  obj.m_velocity.addScaledVector(forceAcceleration, timeDelta);
  obj.m_velocity *= obj.m_damping;
}
```

Code Block 3.8: Physics integrator

Some future proofing of the physics engine is visible here in the form of the object types. While not trivial, it would nonetheless be possible to re-add other types of objects, such as boxes, rods, etc. to the physics engine without changing any of the core logic.

A damping factor is used to apply a global friction to all objects. If there is no damping, numerical instability will eventually cause unwanted and unpredictable behaviour[14]. This is especially important for the Arduino physics engine due to the larger than normal numerical instability caused by using fixed-point arithmetic.

3.2.4 Forces

From the very beginning, each object has contained a force accumulator `Vector3D`, which records the net force experienced by each object. The accumulator initially starts each frame with magnitude zero. Any forces, such as gravity or a spring which are acting on the object, are calculated and added to the accumulator. In Version 2 of the engine forces can either be applied directly to an object, or a `ForceGenerator`[14] can be created. The `ForceGenerator` consists of a function pointer and some parameters. Each `ForceGenerator` is associated with an object. Each frame the force generator function is called and the object is passed in as a parameter. The function then calculates and adds the desired force to the object. If needed, each `ForceGenerator` delegates to object type specific implementations. The implementation of gravity in the physics engine is shown in Code Block 3.9 as an example.

```
struct GravityForceData {
  FixedPoint m_gravMult;
};

struct ForceObject {
  void (*m_generator)(ForceObject&, const FixedPoint&);
  Object* m_obj = NULL;
  union {
    GravityForceData m_gravData;
    SpringForceData m_springData;
    CustomForceData m_analogForceData;
  };
};

void gravityForce(ForceObject& fo, const FixedPoint& timeDelta) {
  if(fo.m_obj->m_invMass > 0) {
    switch (fo.m_obj->m_objectType) {
      case PARTICLE:
        particleGravityForce(fo, timeDelta, fo.m_gravData.m_gravMult);
        break;
      default:
        break;
    }
  }
}

void buildGravityForce(ForceObject *fo, Object* obj, FixedPoint gravMult = ONE)
    {
  fo->m_obj = obj;
  fo->m_generator = gravityForce;
  fo->m_gravData.m_gravMult = gravMult;
}
```

Code Block 3.9: Gravity force generator

3.2.5 Collision Detection

The first attempt at a collision detection and resolution system was a tangle of special cases. Each type of object needed to know how to collide with all other types of objects. A basic bounding box check was done for each pair of objects. More advanced collision detection code would only run if the two objects had intersecting bounding boxes. If the first object was a circle and the second a box, then the circle needed to know how to collide with a box (only one ordering of each pair of types was required, the objects could just be swapped to handle the other order). If they were both circles, a different function would be required. Clearly this was not sustainable or expandable for the future.

While Ericson describes many excellent optimizations for collision detection[15], the ATmega328's limited memory meant that the object count would be very low (currently limited to four). With such a small number of objects, these optimizations would actually hurt performance by increasing overhead, while consuming even more memory.

Instead, a brute force collision detection system is used as described by Millington[14], where each pair of objects is checked for collision and penetration. Since all objects are particles, this requires only a single function to check collisions. Unfortunately, adding in more shapes would again require custom functions for each type, unless the physics engine was changed to deal with generic shapes of any geometry.

Each contact event stored in a `ContactObject` struct, which records the objects involved, the bounciness of the collision (restitution), the direction of the collision (normal), the penetration, and the speed of the collision. The collision detection code for particles is shown in Code Block 3.10.

```
struct ContactObject {
  Object* m_c1;
  Object* m_c2;

  FixedPoint m_restitution;
  Vector3D m_contactNormal;
  FixedPoint m_penetration;
  FixedPoint m_seperatingVelocity;
};

...

bool particleCheckIfCollision(ContactObject* newContact, Object* o1, Object* o2)
    {
  bool isCollision = false;
  switch (o2->m_objectType) {
    case PARTICLE: {
      Vector3D seperation = (o1->m_position - o2->m_position);
      FixedPoint sumRadius = o1->m_particleData.m_radius + o2->m_particleData.
          m_radius;

      if (seperation.magnitude2() < MULT(sumRadius,sumRadius)) {
        isCollision = true;
        newContact->m_penetration = sumRadius - seperation.magnitude();
        seperation.normalize();
        newContact->m_contactNormal = seperation;
      }
      break;
    }
    default:
      break;
  }
  if(isCollision) {
    newContact->m_c1 = o1;
    newContact->m_c2 = o2;
    newContact->m_restitution = POINT_FIVE;
  }
  return isCollision;
}
```

Code Block 3.10: Collision detection

Collision resolution is an iterative process. First, all the collisions for the current frame are generated as shown in Code Block 3.10. The resolution system shown in Code Block 3.11 processes collisions in order of severity of closing velocity (actually stored as separating velocity in the engine). Only those collisions which have a positive closing velocity, and are currently inter-penetrating, are processed.

```
for(int i = 0; i < sim.m_numObjects; i++) {
  sim.m_worldObjects[i].m_penetrationAdjustment = Vector3D(0,0,0);
  for(int j = i+1; j < sim.m_numObjects; j++) {
    if(sim.m_worldObjects[i].m_inUse && sim.m_worldObjects[j].m_inUse) {
      ContactObject tentativeContact;
      if( checkIfCollision(&tentativeContact, &(sim.m_worldObjects[i]), &(sim.
          m_worldObjects[j]))) {
        *simulationGetFreeContact() = tentativeContact;
      }
    }
  }
}
while(totalCollisions < 10) {
  int worstCollision = -1;
  FixedPoint worstCollisionVelocity = FP_MAX;
  for(int i = 0; i < sim.m_activeContacts; i++) {
    calcSeperatingVelocity(sim.m_worldContacts[i]);
    if((sim.m_worldContacts[i].m_seperatingVelocity < -EPSILON ||
        calculateCurrentPenetration(sim.m_worldContacts[i]) > EPSILON) && sim.
        m_worldContacts[i].m_seperatingVelocity < worstCollisionVelocity) {
      //Only bother adding contacts which are worse than the current worst, we
          will only calculate the worst in
      // each iteration.
      worstCollisionVelocity = sim.m_worldContacts[i].m_seperatingVelocity;
      worstCollision = i;
    }
  }
  if(sim.m_activeContacts == 0 || worstCollision == -1) {
    break;
  }
  resolveContact(sim.m_worldContacts[worstCollision], sec);
  totalCollisions++;
}
```

Code Block 3.11: Collision resolution

As seen in Code Block 3.12 collisions are resolved in two parts, inter-penetration and velocity. First, the two objects are moved so they no longer are in contact with each other. The two objects then have their velocities adjusted so they are moving away from each other. The relative magnitudes of the changes for each object is determined by their relative masses.

```
void resolveContact(ContactObject& contact, const FixedPoint& timeDelta) {
  //Fix velocity
  const FixedPoint& seperatingVelocity = contact.m_seperatingVelocity;
  FixedPoint totalMass = contact.m_c1->m_invMass + (contact.m_c2 != NULL ?
      contact.m_c2->m_invMass : 0);
  if (totalMass > 0) {
    if(seperatingVelocity < 0) {
      FixedPoint newVelocity = MULT(-seperatingVelocity, contact.m_restitution);
      FixedPoint delta = newVelocity - seperatingVelocity;
      FixedPoint impulse = DIV(delta, totalMass);
      Vector3D impulseVector = contact.m_contactNormal * impulse;
      contact.m_c1->m_velocity += impulseVector * contact.m_c1->m_invMass;
      if( contact.m_c2 != NULL ) {
        contact.m_c2->m_velocity += impulseVector * -(contact.m_c2->m_invMass);
      }
    }
    //Fix Penetration
    FixedPoint currentPenetration = calculateCurrentPenetration(contact);
    if(currentPenetration > 0) {
      Vector3D movementVectorPerMass = contact.m_contactNormal * DIV(
          currentPenetration, totalMass);
      Vector3D c1Movement = (movementVectorPerMass * contact.m_c1->m_invMass);
      contact.m_c1->m_position += c1Movement;
      contact.m_c1->m_penetrationAdjustment += c1Movement;
      if( contact.m_c2 != NULL ) {
        Vector3D c2Movement = (movementVectorPerMass * -contact.m_c2->m_invMass)
            ;
        contact.m_c2->m_position += c2Movement;
        contact.m_c2->m_penetrationAdjustment += c2Movement;
      }
    }
  }
}
```

Code Block 3.12: Contact resolution

After a collision between two objects is resolved, they will be moving away from each other and will have been displaced so as to not inter-penetrate. This means that the previously calculated collision data for other objects may no longer be valid. To combat this, each object has a `Vector3D` which records all changes made to position during the collision resolution phase. When inter-penetration is checked, it is offset by this amount. Closing velocity is re-calculated for each pair as needed.

Each time a collision is resolved, it may cause a previously resolved collision to re-occur. In theory the system should eventually come to a solution with no outstanding collisions[14] however this may take too long. To maintain a reasonable frame-rate a hard limit is put on the number of collisions which will be resolved in a given frame.

3.2.6 Constraints

Constraints behave very much like conditional collisions which are configured ahead of time. For example, a rope would cause the two objects it was connecting to collide when taut, bouncing back towards each other. Constraints can also be used to simulate floors, rods, or other rigid connections.

Constraints are implemented in a similar way to `ForceGenerator`s. Each constraint contains pointers to some number of objects, and a function to call which processes the constraint. If the function determines the constraint has been violated, it will generate collisions to restore the constraints. For example, a rod constraint will check the current distance between its two ends. If the ends are either too close or too far, a pair of collisions are generated. The collision normals are along the rod, and the interpenetration is the error in the rod length. Two collisions are needed because, during the course of resolving other collisions, the rod may have changed from being too long to too short. If only one collision was included, the rod might allow the two ends to collapse towards each other until the next frame. These collisions are passed to the collision resolution system to deal with.

3.3 Rendering

The Arduino Nano has no traditional video output and lacks the memory required to drive almost all pixel displays - most require the entire screen buffer to be available at once in memory. Instead, a basic renderer was built which runs over UART. The current implementation simply renders a 2D-slice along the Z-axis of the world and outputs it as characters to a console running on a computer, refreshing the screen by writing multiple new-line characters. Even running at 250000 BAUD, the output over UART consumes a significant portion of the CPU time (see Section 4.1 for details).

A frame buffer is used to store the state of the screen before it is transmitted. The screen resolution can be adjusted, but 20×20 gave a good trade-off of quality and size. Due the the lack of available memory, the frame buffer is currently configured to allow only a single ”colour”, but can support multiple ”colours”. Each colour is represented by a different character on screen.

Each frame the simulation state is examined, and each active object is drawn into the screen buffer. Because all objects are spheres, the midpoint circle algorithm can be used to quickly find their edges[16]. A very simple line drawing algorithm is also used to draw a line between objects which are under the effect of a constraint. Each pixel is marked in the buffer for later use. Once all objects are drawn into the buffer, it is output over UART one line at a time. A size multiplier can be used to scale the rendering so that different size worlds can be visualized (default is 1m per pixel, while the haptic interface uses 2cm per pixel).

The engine attempts to maintain a constant frame rate (currently set at one frame every 50ms, for a frame rate of 20 fps). There is an upper limit to the desirable frame rate - too fast and the image appears to jump around in the console on the connected computer. While many physics engines match the physics calculations to the visible frame rate, that would not work well here. large time steps can give unrealistic behaviour, especially for small objects like those being simulated by the haptic interface described in Chapter 2. As an example: small objects could pass completely through each other from one step to the next if the time step is too large. To avoid this, the physics simulation timing was decoupled from the renderer and allowed to run faster.

Because the physics simulation is being used to interface with the real world, it cannot be allowed to slow down relative to real time. After each frame is rendered, the simulation falls behind real time due to the slow UART. The physics engine could simply run one large step to catch back up, but the large jump would cause issues as discussed above. Instead, the physics engine runs several smaller time steps in quick succession until it has reached real time again. The time-step for each of these steps is based on the actual time it takes to calculate the physics. The time taken for the last frame is used, with a small extra amount added (5ms). This allows the physics simulation to catch up with real time, 5ms at a time. Once the physics engine has reached real time, it will try to render a new frame. The renderer will skip rendering a frame to avoid exceeding the maximum desired frame rate, if needed. If the engine is running too fast, it will busy loop until it is 10ms behind real time. If the time-steps taken are too small, the integration can start to lose precision - multiplying each acceleration, and velocity by a very small fraction of a second will cause this, so it is better to wait.

Chapter 4

Results

4.1 Physics Engine Performance

4.1.1 Speed

The performance of the underlying physics engine was measured using a logic analyzer as described in Appendix B. Figure 4.1 shows that the physics engine is easily able to maintain a good frame rate. Even assuming 5ms (as seen in Figure 4.3) to calculate a frame gives a theoretical physics maximum frame rate of 200 frames per second. Figure 4.2 shows switching to floating-point can boost that to over 500 frames per second. The time in-between the frames is where all the extra logic for the haptic interface is run and slows the frame rate considerably.

Human fingers are able to perceive vibrations and changes in position at 25 Hz, and the finger tips at 300 Hz, and sometimes up to 1000 Hz. Because the haptic interface works mostly on gross movements applied to the whole hand it is not necessary to achieve these numbers, but the higher the performance the better [3].

The renderer is clearly the most significant single time cost, taking 20ms to output the frame over UART. When running with `FixedPoint`, the actual frame rate is limited by the ability of the physics engine to catch up to real time, resulting in an actual rendering frame rate of 16 frames per second ($1000ms \div 62ms$) versus the requested frame rate of 20. The floating-point implementation is easily able to catch

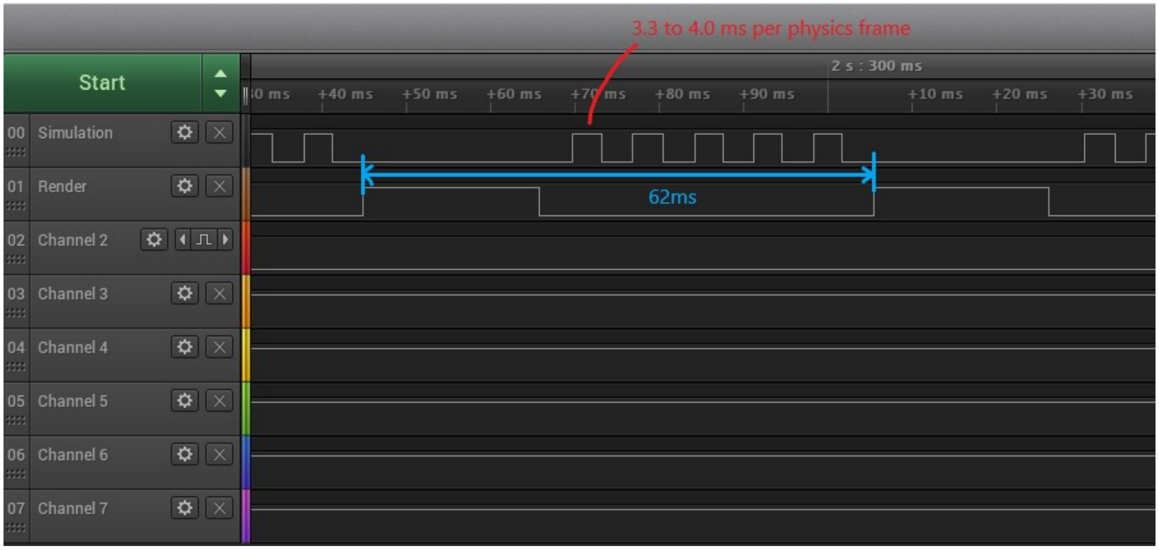

Figure 4.1: FixedPoint frame rate

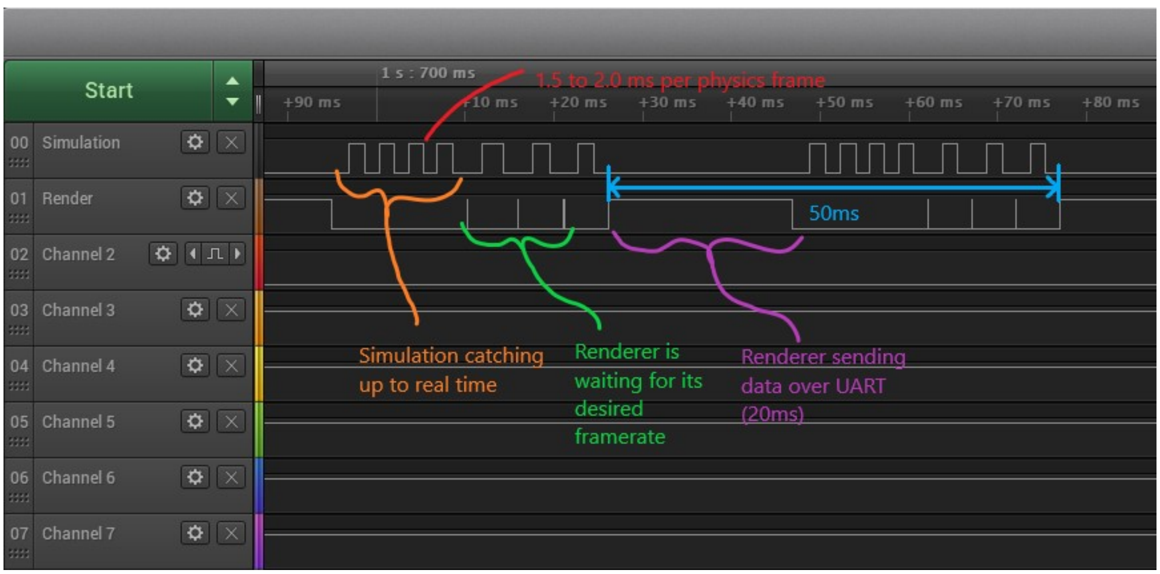

Figure 4.2: Floating-point frame rate

up to real time and the renderer chooses to avoid outputting a new frame so as to meet the requested frame rate.

Figure 4.3 shows details of a single physics step when running with fixed-point arithmetic. The collision resolution code is visible, along with various `FixedPoint` operations. Collision resolution with two objects and a single rod constraint takes 3.5ms of the 4.4ms frame time (80%) which is quite high, although nearly 20% is

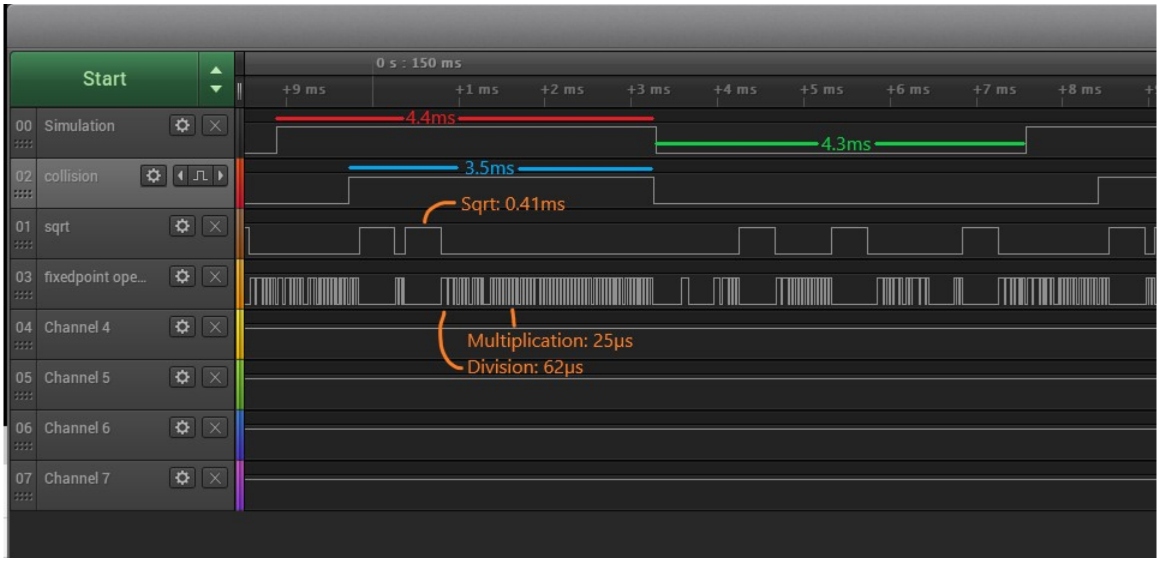

Figure 4.3: Breakdown of a single fixed-point physics frame

tied up in calculating square roots. Fixed-point multiplication matches well with the initial estimates at $25\mu s$ rather than the expected $28\mu s$, but division is actually faster than expected at $62\mu s$ instead of $120\mu s$. This lends some credence to the predicted value of 83 µs calculated in Section 3.2.1.

The haptic interface logic requires three square root calculations, and many multiplication and division operations which take nearly as long as the physics calculations. Most of the processing cost comes from normalizing various vectors (lever force vector normalization, lever magnitude calculation, and acceleration vector normalization).

4.1.2 Memory

The completed haptic interface program and attached physics engine take 18KB of memory, i.e. 60% of the 30KB available on the Arduino Nano. More critical is the available stack space. Statically allocated memory consumes 1217 bytes of the available 2048. Most of the structs used by the physics engine are statically pre-allocated to minimize changes in memory utilization during a run as much as possible, and no dynamic memory allocation is used. The only exception to this is the debug code which uses `String`s in some areas. `String`s in Arduino are allocated on the heap and can cause corruption of other variables in low memory states. Static allocation above 80% of the available memory seems to be an upper limit for reliability if debugging is enabled; any higher and stack corruption can occur in some areas of the program causing a crash, or undefined behaviour. While it is possibly to exceed this limit, any subsequent attempt to enable debugging for the program would lead to crashes. Without detailed memory analysis tools, it is very difficult to track memory use to see what areas of the code push the stack into the heap, or which debug commands inflate the heap. This limit could be improved by converting all `String` handling to use C style strings but the engine as it stands was able to support all the required objects for the haptic interface demo.

4.2 Haptic Interface Accuracy

The performance of the haptic interface is quite subjective. Many different types of simulations were experimented with. One of the most accurate feeling was a very large, heavy pendulum hanging straight down (effectively the current simulation, but rotated 90 degrees downwards). As described above, small fast objects are hard to simulate accurately, but the large ponderous movement of the pendulum felt very good. The large mass of the pendulum however resulted in the user becoming very aware of the limits of the servo motor. Because the large pendulum felt heavy, it felt natural to apply large forces to the lever. These forces could easily stall the servo motor, ruining the force feedback illusion.

The small lever on the other hand, despite its weaknesses, allows the user to get fast feedback. Changes in the setup such as lever length can be easily and quickly felt.

There is room for improvement in both the hardware and software. On the hardware side, more responsive sensors may help with the oscillations, and an improved power supply could help boost motor performance. The software oscillation damping could likely be improved further with more research into the physics of oscillations, and more data on the exact sequence of events which leads to the oscillations.

4.2.1 FSR Responsiveness and Backlash

A critical consideration for the FSRs is how quickly they can respond to user input. While the sensors themselves react instantly, there is backlash in the foam and the 3D-printed structure around it. When a force is applied to the lever, the outer structure will rotate until the force felt by the FSR is equal to the force being applied to the lever. There is approximately 2 mm of foam between the outer structure and the FSR. The sensor structure is solidly mounted to the servo and there is very little backlash in that connection. In the worst case, where the full depth of the foam is compressed, the arm will rotate 2.8°, corresponding to a movement of 0.97 cm at the end of the lever. The force increases progressively and the angle is small enough to not be noticeable to the user. Using stiffer foam together with a smaller gap in the structure would help minimize this further.

The accuracy of the force feedback is also effected by the angular resolution of the servo motor. This resolution is often included in the specifications, but not in the case of the FS0521HV[10]. Looking at a similar performance, but slightly more expensive, servo motor which uses a different protocol (Dynamixel RX-24F), gives an approximation of 0.29°. From above, the full range of motion for the foam is 2.8° of rotation. This suggests that the servo can produce nine different forces.

Clearly a thicker, less stiff foam pad gives better force resolution, but also a less accurate lever angle. An alternative hardware design could remove this trade-off at the expense of increased complexity. Instead of using a prepackaged servo, the various components could be split out. The motor would remain in place but the encoder, which determines the angle of the motor shaft, would be moved and connected to the actual lever instead of the sensor mounting. This would allow the sensor mounting to rotate independently of the desired lever angle. A very large piece of foam could be used to give many more possible forces. The foam could even be replaced with springs, giving an even larger range and more linear response. Unfortunately this change would likely make the oscillations discussed in the next section even more difficult to deal with.

4.2.2 Dealing with Oscillations

While the accuracy of the FSRs is fairly good, the backlash in the coupling described above is problematic for another reason. The haptic interface ended up being susceptible to severe oscillations during use. These oscillations were most readily apparent when holding the physical lever steady with a short virtual lever. Without better tools it is difficult to determine the exact sequence of events that lead to these oscillations, but an educated guess is possible. These types of instabilities are not uncommon in closed-loop admittance haptic interfaces [4].

Starting from a stationary state, with the lever held horizontal and the weight suspended in the air, the only force acting on the rod end is gravity. The rod end will start to fall, with the servo motor rotating the sensor mount to follow. As the sensor mount rotates, the FSR on the bottom of the mounting will begin to see increased force against the stationary lever. Ideally the force from the sensor would exactly counter the gravitational force and the rod end would stop moving. Unfortunately, in reality the rod end will often overshoot slightly, continuing to increase the force on the sensor. Quickly the force from the sensor will exceed the force of gravity and the rod end will begin to rise. The servo motor will again rotate to follow, moving the sensor mounting so that the force output is zero. The rod end then continues on a ballistic trajectory until the sensor mounting rotates enough to begin forcing it back down. The rod end now experiences both the force of gravity and the force of the lever. It will then be forced down, more quickly than the first time, until the FSR force overcomes gravity and it once again rebounds. This oscillation continues to build until the haptic interface becomes impossible to hold and it shakes itself loose, or the servo motor stalls.

Multiple solutions were explored for dealing with this. The most straight forward would be to use a torque sensor which does not have backlash, but as discussed before they are far too expensive. The next solution was to simply damp all motion; but if the damping was strong enough to bleed off energy from the oscillations, it would also negatively affect the normal operation of the physics engine. The lever moved very slowly and while the weight still felt heavy, it also felt like it was falling though water instead of air.

The next attempt involved calculating the forces exerted on the FSRs from the acceleration of the sensor structure as shown in Table D.2. The hope was that by removing the forces applied to the FSRs by the acceleration of the servo motor, there would not be enough energy added to start the oscillations. Unfortunately, this also seemed to have no discernible effect and occasionally would exacerbate the issue. This is often an approach used with closed-loop interfaces but it requires very accurate understanding and modelling of the interface dynamics.

The final solution to the oscillations was to add a selective damping force. The damping is only applied when trying to push the lever in the opposite direction it is moving. This allows the lever to drop quickly under the force of gravity, and the effect is subtle enough to be overlooked when holding the lever. It does not totally remove the oscillations, but it does reduce them enough in almost all cases that they quickly disappear.

4.3 Conclusions

Overall, the exploration into fixed-point arithmetic was a slight disappointment, due to the fact that its performance was equal or inferior to emulated floating-point for the purposes of a physics engine. Regardless, the physics engine runs well with either fixed- or floating-point arithmetic on a very resource constrained device. The physics engine was capable enough to successfully implement a one degree-of-freedom haptic interface. The haptic interface is able to accurately detect human input using FSRs, simulate the behaviour of a virtual lever, and then move the users hand in response to their input.

Future development could focus on (1) improving collision detection performance, (2) switching the debug system to use only C style strings, which would allow more free memory, and (3) improving the fixed-point division function. The extra memory could then be used to either simulate additional objects, or perhaps more interestingly add different types of pixels or a depth buffer.

References

[1] Arduino nano. [Online]. Available: https://store.arduino.cc/usa/arduino-nano

[2] V. Hayward, "Survey of haptic interface research at mcgill university," in *Proceedings of the Workshop in Interactive Multimodal Telepresence Systems*, 2001, pp. 91–98.

[3] R. E. Ellis, O. M. Ismaeil, and M. G. Lipsett, "Design and evaluation of a high-performance haptic interface," *Robotica*, vol. 14, no. 3, pp. 321–327, 1996.

[4] C. R. Carignan and K. R. Cleary, "Closed-loop force control for haptic simulation of virtual environments," 2000.

[5] J. W. Hill, K. W. Gardiner, and J. C. Bliss, "Design study of a tactile cuing system for pilot training," STANFORD RESEARCH INST MENLO PARK CA, Tech. Rep., 1969.

[6] M. A. Otaduy and M. C. Lin, "Stable and responsive six-degree-of-freedom haptic manipulation using implicit integration," in *Eurohaptics Conference, 2005 and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2005. World Haptics 2005. First Joint.* IEEE, 2005, pp. 247–256.

[7] Interlink electronics. [Online]. Available: http://interlinkelectronics.com/FSR402.php

[8] E. Herria. Regression tools - online nonlinear regression. [Online]. Available: http://www.xuru.org/rt/NLR.asp

[9] Sg5010 — tower pro. [Online]. Available: http://www.towerpro.com.tw/product/sg5010-4/

[10] Rjxhobby fs0521hv digital hv standard servo (20mmx37x40mm) black. [Online]. Available: http://www.rjxhobby.com/rjx-fs0521hv

[11] K. Turkowski, "Fixed point square root," *Media Technologies: Computer Graphics, Advanced Technology Group, Apple Computer, Inc.*, 1994.

[12] J. Vijn. Off on a tangent : a look at arctangent implementations — coranac. [Online]. Available: http://www.coranac.com/documents/arctangent/

[13] Arduino reference. [Online]. Available: https://www.arduino.cc/reference/en/language/variables/data-types/float/

[14] I. Millington, *Game Physics Engine Development, Second Edition: How to Build a Robust Commercial-Grade Physics Engine for Your Game*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[15] C. Ericson, *Real-Time Collision Detection*, ser. The Morgan Kaufmann Series in Interactive 3D Technology. Elsevier Science, 2004. [Online]. Available: https://books.google.com/books?id=0MvuykjoW_IC

[16] Midpoint circle algorithm - wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Midpoint_circle_algorithm

Appendix A

Terms and Acronyms

Glossary

FSR A Force-Sensitive Resistor reduces its resistance as more pressure is applied to it. 6–8, 10, 12, 13, 15, 42–44

Haptic Interface A haptic interface provides force feedback to a user allowing them to feel an object they are interacting with in a virtual world. 1, 3, 5, 6, 8, 11, 12, 17, 20, 37, 38, 40, 41, 43, 44

Servo Servo motors are motors which are controlled using a third control wire (in addition to +V and GND). The control wire is used to set a desired angle. The motor will attempt to reach that angle automatically. 8, 9, 15, 42

UART Universal Asynchronous Receiver Transmitter is a very basic IO interface able to transmit one byte at a time over a pair of wires. The Arduino Nano[1] can theoretically support up to 2Mbps over hardware UART. The UART is also connected to an on-board Future Technology Devices International (FTDI) UART to USB chip which can route the UART data through the USB connection on the Arduino. 1, 36, 37, 49

Acronyms

FPU floating-point unit. 1, 18

PID proportional-integral-derivative. 10

Appendix B

Debugging and Profiling

Debugging embedded systems can be challenging. The only easy way to get data out of the Arduino is over UART, or by setting the digital output pins high and low. A set of debug and profiling tools were built into the physics engine to help make debugging easier. Code Block B.1 shows the debugging tools included to output data over UART. By passing all debug output through a single system, it can be quickly turned on and off for various parts of the code. The conditional definition of `debug()` and `debugln` is required because the compiler appears to allocate memory for the strings which would be printed even if they are never actually used. By using a conditional define, the actual string can be removed by the preprocessor before compilation.

Manipulating and transmitting strings is both slow and memory intensive. This makes it useless for the purposes of profiling the code. It is much faster to toggle digital pins on and off instead. Code Block B.2 shows the profiling code used to gather the performance measurements used in this report.

```
enum DebugType {DEBUG_GENERAL, DEBUG_COLLISION, DEBUG_CONSTRAINT,
    DEBUG_FIXEDPOINT, DEBUG_FORCE, DEBUG_PARTICLE, DEBUG_RENDERER, DEBUG_SIM,
    DEBUG_VECTOR};

#define DEBUG_ON (defined( DEBUG_GENERAL_ENABLED) || defined(
    DEBUG_COLLISION_ENABLED)  || defined( DEBUG_FIXEDPOINT_ENABLED)  || defined(
     DEBUG_FORCE_ENABLED)  || defined( DEBUG_RENDERER_ENABLED)  || defined(
    DEBUG_SIM_ENABLED) ||  defined( DEBUG_VECTOR_ENABLED) ||  defined(
    DEBUG_CONSTRAINT_ENABLED))
#if DEBUG_ON
#define debug(s, c) (debug_internal(s, c))
#define debugln(s, c) (debugln_internal(s, c))
#else
#define debug(s, c)
#define debugln(s, c)
#endif

inline void debug_internal(const String &string, const DebugType &debugType,
    const bool newLine = false) {
#if DEBUG_ON
  switch (debugType) {
#ifdef DEBUG_GENERAL_ENABLED
      case(DEBUG_GENERAL):
        Serial.print(string);
        if (newLine) {
          Serial.println();
        }
        break;
#endif
#ifdef DEBUG_COLLISION_ENABLED
      case(DEBUG_COLLISION):
        Serial.print(string);
        if (newLine) {
          Serial.println();
        }
        break;
#endif
...
}
...
debug(contact.m_c1->m_penetrationAdjustment.toString(), DEBUG_COLLISION);
```

Code Block B.1: Debug code

```
enum ProfileType {PROFILE_SIM=2, PROFILE_RENDER=3, PROFILE_COLLISION=4,
    PROFILE_SQRT=5, PROFILE_OPERATOR=6};

inline void PROFILE_INIT() {
  pinMode(PROFILE_SIM, OUTPUT);
  digitalWrite(PROFILE_SIM, LOW);
  pinMode(PROFILE_RENDER, OUTPUT);
  digitalWrite(PROFILE_RENDER, LOW);
  pinMode(PROFILE_COLLISION, OUTPUT);
  digitalWrite(PROFILE_COLLISION, LOW);
  pinMode(PROFILE_SQRT, OUTPUT);
  digitalWrite(PROFILE_SQRT, LOW);
  pinMode(PROFILE_OPERATOR, OUTPUT);
  digitalWrite(PROFILE_OPERATOR, LOW);
}

inline void PROFILE_ON(ProfileType type) {
      digitalWrite(type,HIGH);
}
inline void PROFILE_OFF(ProfileType type) {
      digitalWrite(type,LOW);
}
...
inline FixedPoint fp_multiply(FixedPoint m1, FixedPoint m2) {
  PROFILE_ON(PROFILE_OPERATOR);
  ...
  PROFILE_OFF(PROFILE_OPERATOR);
  return result;
}
```

Code Block B.2: Profiling code

Appendix C

Wiring Diagram

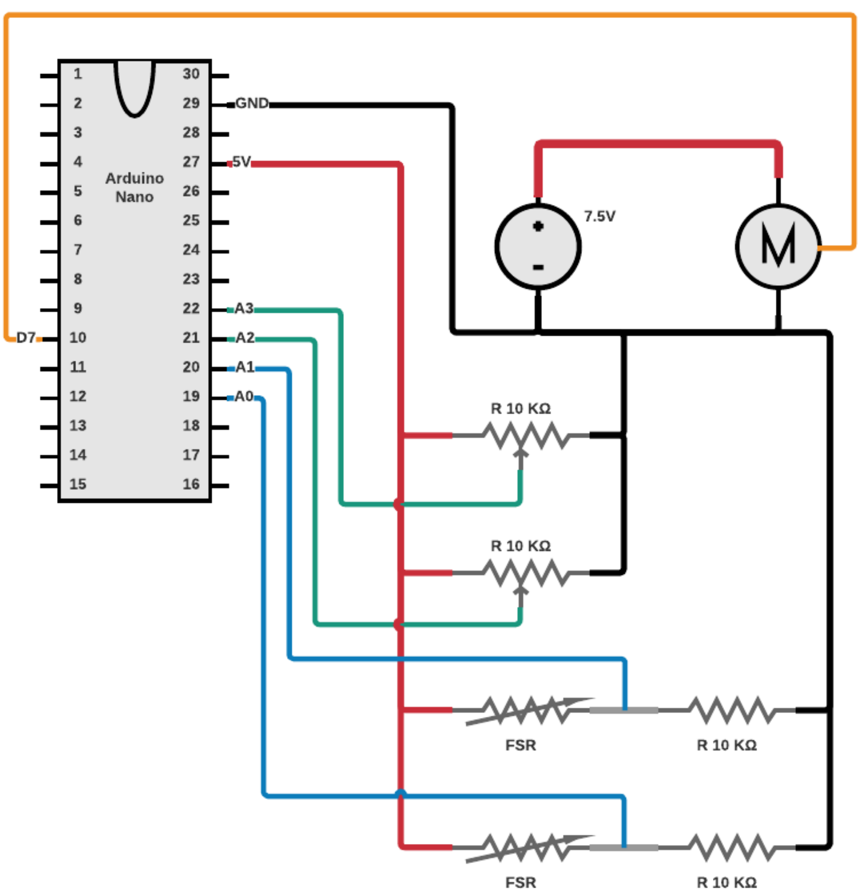

Figure C.1: Haptic interface wiring diagram

Appendix D

Additional Tables

Analog Reading	Actual weight(g)	Calculated weight(g)
0	50	0
415	100	92.8
530	150	123.8
640	200	169.5
705	250	211.9
755	300	259.6
790	350	306.7
824	400	370.8
855	450	456.3
870	500	512.9
880	550	558.9
895	600	645.3
900	700	680.1
906	800	727.2
920	900	866.5
940	1100	1190.3
950	1500	1461.9

Table D.1: FSR curve fitting results for Function 1

Acceleration(m/s^2)	N per m/s^2 Adjustment
5	0.111888194
6	0.109072801
7	0.10006929
8	0.085951109
9	0.084463968
10	0.072819874
11	0.080326486
12	0.08477272
13	0.076207648
14	0.081257382
15	0.064748871
16	0.068643572
17	0.065088944
18	0.056335634
19	0.059602573
20	0.070448792
21	0.063760947
22	0.053436257
23	0.063748326
24	0.056996927
25	0.058512895
26	0.0572307
27	0.050489067
28	0.056210608
29	0.053278013
30	0.049497032

Table D.2: Sample of force corrections