INHABITANCE

A DIGITAL INTERACTIVE INSTALLATION
BY SPENSER SPRATLIN

IN AFFILIATION WITH TEMPLE UNIVERSITY

TABLE OF CONTENTS

Inhabitance: A Digital Interactive Installation

Spenser Spratlin

Abstract

Inhabitance is a Digital Interactive Installation employing the use of projection design, sound design, infrared depth mapping, and complex programming in the languages Processing and Java, in order to create an experiential work that leaves a lasting impact on both participants and the work itself.

At it's core, *Inhabitance* is about spatial and sensory presence. Taking distinct inspiration from other interactive projection works such as *Snow Fall (2009)* by Italian media studio FUSE°, and *Text Rain (1995)* by Camille Utterback and Romy Achituv, *Inhabitance* encourages participants to allow the boundary between their physical bodies and digital representations to blur. Participants who enter the active area of the installation are greeted with a white silhouette of themselves in a sea of letters. These letters, as they will soon discover, are not stationary. They react and interact with the movements made within the active area of the installation. Similarly to how participants in *Text Rain* soon begin to experiment with the interaction with the text, those in *Inhabitance* begin to "play" and test the rules of this newfound sensory environment. However, there is one more layer of interaction to be discovered. As soon as a letter moves, it creates a sound. Another is added to the space when another letter is moved. Each interactive object on the screen fills the space with sound, allowing participants to orchestrate their own accompaniment to the space. As those inside come to allow the boundary between physical and digital to blur, they now also begin the process of blurring the boundary between sight and sound. Reminiscent of Synesthesia (the condition where one sense triggers the experience of a different sense, such as sound triggering a sense of touch), the blurring of senses allows participants to experience a sense of "diminished consciousness of self", and create an audiovisual experience totally unique to them. These letters and sounds will continue long after the participants have left the space, allowing messages, chords, or any other lasting impression to be passed onto those that will

come next. Each new participant's experience is a fully unique version, informed by those that have come before.

"Under the hood" of *Inhabitance* is a complex marriage of software and hardware. Kinect cameras create an accurate depth map of the space, and feed their findings directly into Processing. Processing, with the help of libraries crafted by Daniel Schiffman, takes the Kinect information and uses that to draw a silhouette of any participants in a specific range. The letters and sounds are also handled by Processing, but without the use of a pre-built library. The letter objects, sonic interaction, and collision have all been coded by hand and are specific to the use-case of *Inhabitance*.

Mechanical Art in an age of Mechanical Reproduction

In 1936, German philosopher Walter Benjamin crafted the concept of the "Aura" in relation to physical works of art as a quality of the work that could not be reproduced through mechanical techniques like Photography. Benjamin would go on to argue that the "Aura" was something to be rebelled against to aid in the democratization of art, but what if the "Aura" could be exploited *through* mechanical reproduction?

Presence is the pillar on which all art stands. Without viewers to experience, view, or otherwise interact with a work of art, the social purpose of that art cannot be fulfilled. This experiential nature of art is true regardless of reproducibility, though this changes and magnifies when the work is *designed* with reproduction in mind. The "Aura" that Benjamin writes about survives Mechanical Reproduction when Mechanical Reproduction is crucial to the work itself. Mechanical Art possesses an inescapable "Aura" that fuels its acceleration through the Digital Age. This is doubly true for Mechanical (Digital) Installation. The "Aura" *is* the work, and the work *is* the "Aura". They cannot exist without one another. This is the conceptual base on which all of my work stands, in one way or another. To capture the experiential nature of art, to exploit the Aura, and through the use of live video and sensor technology, ensure that the work can quite literally not exist without it.

Without Presence there is no Art. Without Art, there is no Presence.

TIMELINE - START TO FINISH

January 2022

- Idea Drafting in Media Arts Theory Practice class.
- Live Video used in MadMapper, Development of Core Concepts begins
- First idea drafting, “Interactive Shadow”

February 2022

- Beginning technology tests
- Troubleshooting video feed, Implementing usage of FS5 OBS Link, and Elgato HD60 to handle video feeds.
- MadMapper dropped. Projection Software Research Begin
- Begin research into Physical Interfaces.
- Purchase of Isadora 3 (Projection Designer)

March 2022

- First conclusive tests, accurate “interactive Shadow” in tandem with Alpha Channel integration.
- Visual Optimization Begin, Image Processing in Isadora 3
- Begin testing for larger scale iterations, purchase of Arduino and FSRs

April 2022

- Coding Arduino (C++) and assembly of first full iteration.
- Fabrication of FSRs into sensor mat.
- Implementation of Arduino Processor into Isadora 3
- Implementation of Albert Camus’s “Myth of Sisyphus”
- First device transfer onto Temple Mac Mini
- First full Exhibition at Diamond Screen Festival (Awarded)

June 2022

- Research into Kinect technology, integration into Isadora.
- Research and Implementation of Body Tracking in Isadora 3 (Unstable)
- Format inverted from “Interactive Shadow” to “Interactive White Silhouette”

July 2022

- Development of new features “Ghost” feature fully functional

August 2022

- Second Full Exhibition at “Perpetual Motion” show in Lancaster PA
- Revised Text element to excerpt from “A Thing Like You And Me”
- Problems documented with Exterior exhibitions
- Stability Problems Persist, Research into Optimization begin

September 2022

- Research into first iterations of Sonic Component, adoption of Overhead Cam method.
- Significant Stability Issues documented with both Body Tracking and Overhead Cam. Previous Stability Issues still unsolved.

October 2022

- Isadora instability unsolved, begin switching Body Tracking to Processing, using OpenKinect libraries from Daniel Shiffman.
- Troubleshooting sonic component

November 2022

- Processing code finalized, Sonic Component successfully added.

December 2022

- Third full Exhibition at “The Library as a Futurist Book”
- Text element dropped, Sonic component fully operational
- Stable for multiple hours

January 2023

- Begin intensive Java Learning
- Revisitation of Kinect code, fundamentals established

February 2023

- Reassessing of the Installation, integration of Physics objects
- Idea drafted for Current iteration
- Deep dive into learning the Processing language.

March 2023

- Isadora fully dropped, Sonic Component added to processing
- Adoption of Box2D Engine to handle Physics
- Sonic Component added to Box2D
- Kinect Tests with Box2D. Proved Extremely difficult.
- Abandonment of Box2D Engine to handle Physics
- Start writing Box class
- Begin work on purpose-built Physics engine
- Establish Testing Grounds on Github for tests
- Object interaction successful (though buggy), Kinect functionality working
- Multi-Kinect tests still inconclusive
- Scaling Tests inconclusive
- https://github.com/SWSpratlin/Thesis_TestingGrounds.git

April 2023

- Optimization of Physics engine
- Finalization and stability testing of Object Collision
- Integration of Sonic component to full testing grounds
- Multi-Kinect issues resolved
- “Reset” function added
- Equipment List drafted and reviewed
- Additional Equipment purchased and tested
- Presentation and Demonstration on 4/18 for New Media Survey
- Display Format decided

May 2023

- *Fourth full Exhibition at Diamond Screen Festival*
- *Defense of Inhabitance at Diamond Screen Festival*

Inhabitance Production Budget

Item	Unit Cost	AMT	Budget Cost	In-Kind Cost	Total Cost	Total Budget Cost	Subtotal
Isadora Subscription	$18.95	12	$227.40	$0	$227.40	$227.40	$227.40
Elgato HD60 Capture Card	$155.00	1	$0.00	$155.00	$155.00	$227.40	$382.40
Arduino Uno + Case	$30.66	1	$30.66	$0.00	$30.66	$258.06	$413.06
Soldering Supplies	$38.96	1	$38.96	$0.00	$38.96	$297.02	$452.02
Heatshrink Tubing	$6.35	1	$6.35	$0.00	$6.35	$303.37	$458.37
BreadBoard	$15.82	2	$31.64	$0.00	$31.64	$335.01	$490.01
Arduino Extension Cable	$22.25	1	$22.25	$0.00	$22.25	$357.26	$512.26
Force Sensitive Resistors	$15.66	4	$62.64	$0.00	$62.64	$419.90	$574.90
Foam Mats	$25.00	1	$25.00	$0.00	$25.00	$444.90	$599.90
Masonite Hardboard	$13.98	1	$13.98	$0.00	$13.98	$458.88	$613.88
Kinect 1	$46.18	1	$46.18	$0.00	$46.18	$505.06	$660.06
Kinect 2	$20.09	1	$20.09	$0.00	$20.09	$525.15	$680.15
Kinect 3	$6.99	1	$6.99	$0.00	$6.99	$532.14	$687.14
Overhead Webcam	$69.96	1	$69.96	$0.00	$69.96	$602.10	$757.10
Webcam Extension Cable	$15.00	1	$15.00	$0.00	$15.00	$617.10	$772.10
Apple MultiPort Adapter	$69.00	1	$69.00	$0.00	$69.00	$686.10	$841.10
Processing & Kinect Research	$45.00	1	$45.00	$0.00	$45.00	$731.10	$886.10
Projectors	$1,554.61	2	$0.00	$3,109.22	$3,109.22	$731.10	$3,950.32
Projection Screens	$60.00	2	$0.00	$120.00	$120.00	$731.10	$4,070.32
Sony FS5	$3,400.00	1	$0.00	$3,400.00	$3,400.00	$731.10	$7,470.32
Kinect (Library)	$30.00	1	$0.00	$30.00	$30.00	$731.10	$7,500.32
Tripods	$150.00	1	$0.00	$150.00	$150.00	$731.10	$7,650.32
Combo Stand	$348.00	1	$0.00	$348.00	$348.00	$731.10	$7,998.32
Projectors (new)	$350.00	2	$700.00	$0.00	$700.00	$1,431.10	$8,698.32
Projection Screens (new)	$200.00	2	$400.00	$0.00	$400.00	$1,831.10	$9,098.32
Speakers (new)	$450.00	1	$450.00	$0.00	$450.00	$2,281.10	$9,548.32

*In Kind from University

TOTAL $9,548.00

*Purchased OOP

*Purchased through Grant funds

SETTING UP *INHABITANCE*

Spatial Requirements

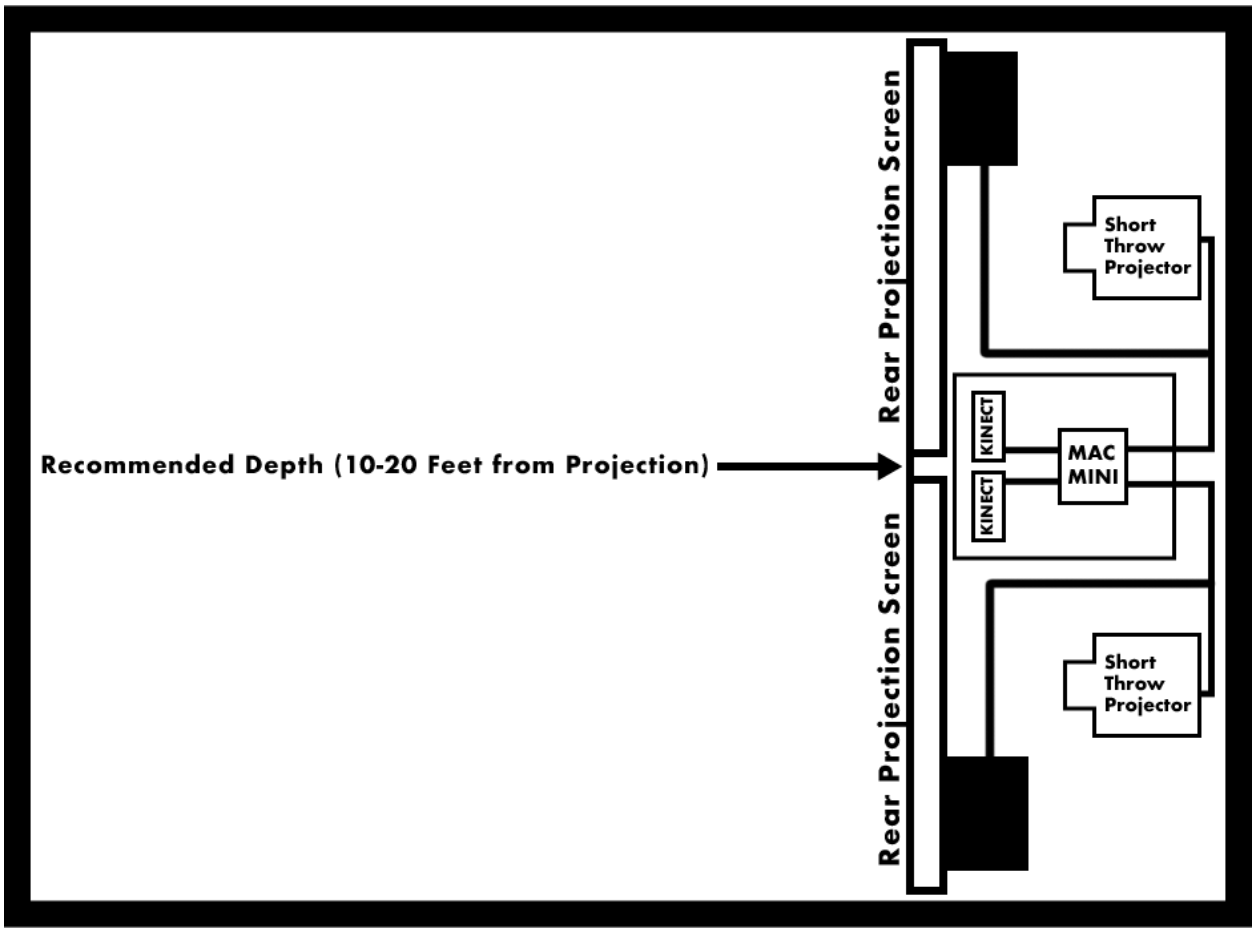
Inhabitance requires a relatively large, relatively square/rectangular space to set up.
A minimum Distance of 2 Feet is required to allow the sensors to work, with participants able to go no further than 20 feet from the sensors.

Short Throw projectors and Rear Projection is recommended for the most control but not required if the space has pre-installed ceiling-mounted projectors.

A small table or platform about 2-3 feet in height is required to house the sensors (and Computer if no pre-existing computing space exists).

An optimal hardware list can be found at the end of this guide.

General Space Setup



Instructions

After the space has been measured, confirmed to meet the spatial requirements, the platform for both sensors and Computer has been set up, and all relevant equipment has been gathered, follow the steps outlined below to set up and run *Inhabitance:*

1) Connect the Mac Mini (or equivalent Mac computer) to an external display for setup and calibration.
    a) Download all relevant files and assets from https://github.com/SWSpratlin/Thesis_TestingGrounds
2) Connect the Kinect Camera(s) to the Computer
3) Launch the included “RGB Calibration” downloaded from Github
4) Adjust the Kinect(s) to either:
    a) Produce a seamless video feed between the two of them
    b) Include as much of the space as Possible with a single sensor
5) Mark the Kinect(s) positions on the platform. Close “RGB Calibration”
6) Set Up Projection Screens and (Rear) Projectors
7) In “System Settings > Desktop and Dock > Displays Have Separate Spaces” and toggle this setting OFF.
8) Connect both Projectors to the Computer.
9) In “System Settings > Displays” adjust the Resolution and Arrangement so that the Projectors show a Resolution of “480” (Tests in progress to fix this issue) and the two Projectors are arranged next to each other. Make sure the projector seam aligns with where the Projection Screens meet in the middle.
10) Launch the “Depth Calibration” app downloaded from Github
11) With another person, Adjust the depth calibration to Desired Range. Record Calibration Settings.
12) Launch *“Inhabitance”* in Visual Studio Code, Connect Speakers.
    a) Input depth measurements, test.

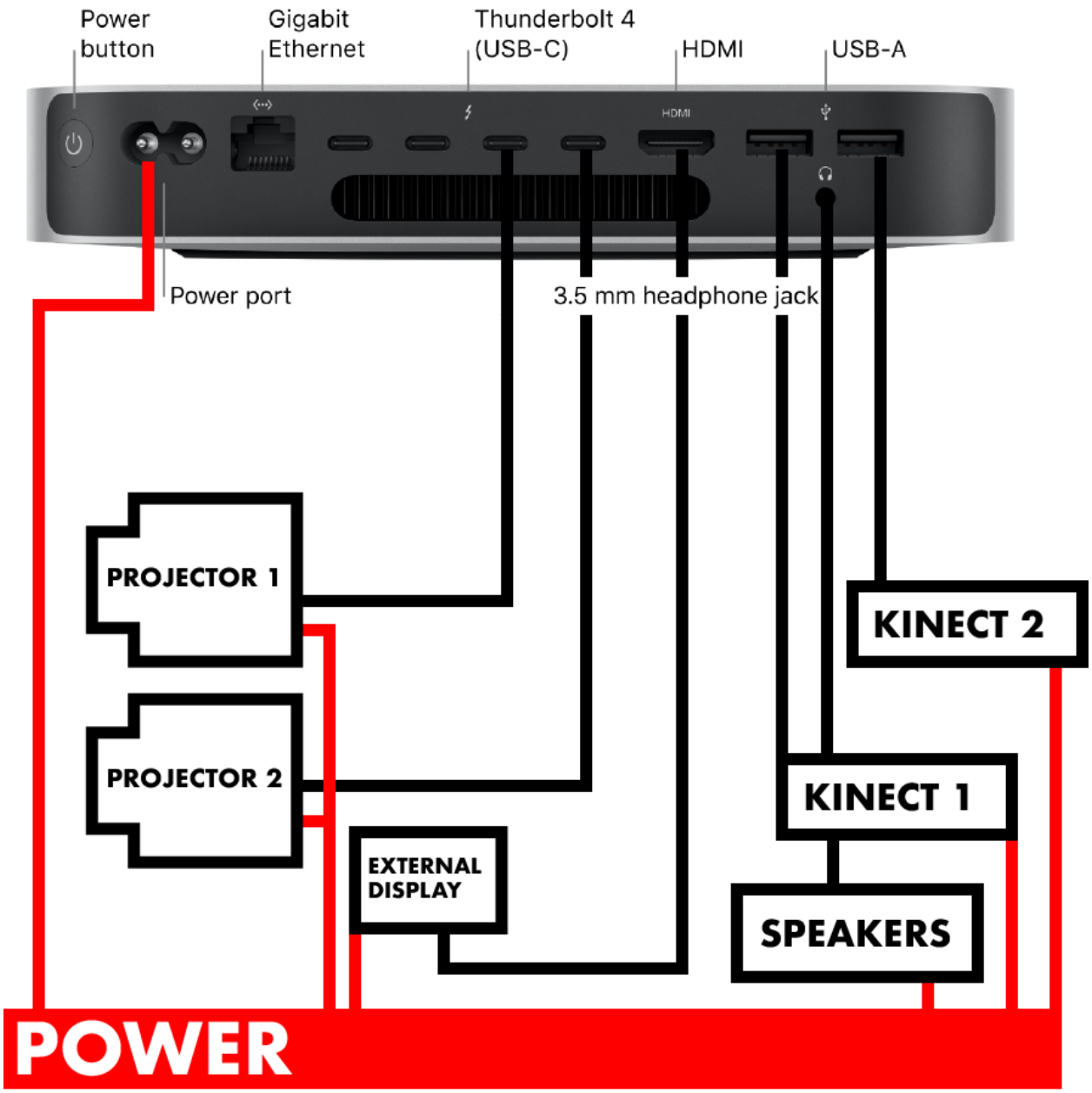
Hardware Setup

Red Lines are for Power

Black Lines are for Data Appropriate Cabling

USB C wiring to Display is necessary when using an External Display

Kinects require a USB A, or a first-party USB C Adapter.



Optimal Hardware List

Computing

- Mac Mini (INTEL ONLY)
    - Any Apple computer with at least 2 Display Outputs and 2 USB A inputs.

Kinects

- One or Two Kinect V1 Cameras
    - Models 1414 or 1473

Projectors

- BenQ MW853UST Ultra Short Throw Projector
    - Rear projection for best results
    - Rear Projector should have a throw ratio < 0.5:1 to save space.
    - Ceiling Mounted Projectors must be tested on a case-by-case basis

Projection Screen

- Any 16:9 Ratio projection Screen will work.
- 1 screen for each Kinect
    - Best results stem from at least 5’ height on the screens, with the screens raised ~2 Feet from the ground. This allows the Kinects to cover the most seamless ground.

Speakers

- Any TRS Connecting speakers will work.
    - The Computer must be able to send signal via TRS cable to the speakers. If a designated receiver is needed to power the speakers, ensure that the receiver has a TRS Input, or RCA input with the proper adapter.

TECHNICAL BREAKDOWN

Inhabitance requires a fair amount of technical knowledge to properly test and run. This section will outline the specific technologies that are being used in the classes and source code of *Inhabitance* as well as what specific software/system requirements are needed for the installation to run in both its Sketch format as well as the Packaged Application format.

At this time, *Inhabitance* only runs on MacOS Intel. Apple Silicon still contains bugs in the Kinect library, which prevent the program from compiling and running properly. For best results, use the specified hardware listed in the Setup Instructions.

Before assessing the software requirements of *Inhabitance*, it should be noted that a basic knowledge of coding will be assumed, as well as cursory knowledge of Java (and by extension, Processing).

Software Requirements

Sketch Format

- In order to run *Inhabitance* in it's sketch format, the following must first be installed:
    - Java JRE
    - Java JDK
    - Visual Studio Code
        - Visual Studio Code Relevant extensions
            - Processing Language Integration
    - Processing IDE
        - Make sure to install processing.jar for all users on the computer
    - Processing Libraries:
        - Java Point class by Oracle
        - OpenKinect by DanielShiffman
        - SimpleOpenNI by Max Rheiner
        - Sound by Processing Foundation
    - All Relevant files from https://github.com/SWSpratlin/Thesis_TestingGrounds
- These must all be run on an Apple Intel computer that meets the requirements listed in the Setup Instructions section.

Packaged Application Format

- This is unfortunately currently non-functional.
- Additional Tests required, but should be ready to go soon.

Inspection of Code

This section will break down the specifics of both the “main” as well as the “Box” class in order to establish a baseline understanding of how to troubleshoot or adapt the installation to a specific space.

MAIN CODE

This is the main code for *Inhabitance*, putting all the pieces together. There’s not much room for customization here, but this is important for troubleshooting and calibration.

```
import processing.sound.*;
import java.awt.Point;
import org.openkinect.freenect.*;
import org.openkinect.processing.*;

public PApplet master = this;

//Master Image
PImage masterImg;

//Array to store the Int arrays for the Switching block
ArrayList<int[]> currentArray = new ArrayList<int[]>();

//Depth Thresholds
float minDepth =  300;
float maxDepth = 800;

//Kinects to Iterate through
ArrayList<Kinect> kinects;

//Boxes and Notes to assign
ArrayList<Box> boxes;
ArrayList<String> notes;

//Number of Boxes to spawn
int boxNumber = 40;
```

```
//Device variables
int numDevices = 0;
```

This is the declaration of all the global variables, and Initialization of the currentArray for the switching block later in draw();

```
void setup() {
      // Size. Width has to be double the width of a Kinect
      //This lets 2 Kinect feeds populate next to each other
      size(1280,480);

      //Populate Notes Array
      notes = new ArrayList<String>();
      notes.add("A__1.wav");
      notes.add("B__1.wav");
      notes.add("C__1.wav");
      notes.add("D__1.wav");
      notes.add("E__1.wav");
      notes.add("F__1.wav");
      notes.add("G__1.wav");
      notes.add("H__1.wav");
      notes.add("I__1.wav");
      notes.add("J__1.wav");
      notes.add("K__1.wav");
      notes.add("L__1.wav");
      notes.add("M__1.wav");
      notes.add("N__1.wav");
      notes.add("O__1.wav");
      notes.add("P__1.wav");
      notes.add("Q__1.wav");
      notes.add("R__1.wav");
      notes.add("S__1.wav");
      notes.add("T__1.wav");
      notes.add("U__1.wav");
      notes.add("V__1.wav");
      notes.add("W__1.wav");
      notes.add("X__1.wav");
```

```
    notes.add("Y__1.wav");
    notes.add("Z__1.wav");
```

This is a long block for a simple thing. This populates a String ArrayList to be put into the SoundFile constructors within the Box class.

```
    //Kinect initialization
    numDevices = Kinect.countDevices();
    println(numDevices);

    //Initialize Kinect array
    kinects = new ArrayList<Kinect>();
```

This counts the devices and prints it to the console for troubleshooting purposes. Look for the single number in the console and make sure it matches the physical number of Kinects you've hooked up. This currently only works with 2, but may expand to more if the need appears.

```
    //Iterate through as many Kinects as are connected
    for (int i = 0; i < numDevices; i++) {
    //Initialize object for each Kinect detected
    Kinect tempKinect = new Kinect(this);

    //Activate the Current Kinect
    tempKinect.activateDevice(i);

    //Initialize Depth for current Kinect
    tempKinect.initDepth();

    //Add the current Kinect to the Array
    kinects.add(tempKinect);

    //Initialize rawDepth array for each Kinect
    //MUST BE IN THIS LOOP OR ONLY ONE KINECT WILL WORK
```

```
      int[] tempRawDepth = tempKinect.getRawDepth();

      //Add the rawDepth array to the Switching Array
      currentArray.add(tempRawDepth);
      }
```

This is the lynch pin for the dual kinects to work. Iterating through each of the Kinect devices and activating them/Initializing their depth. The important part of this is also adding int arrays to the currentArray ArrayList. Doing this in this loop makes the dual Kinects both activate.

```
      //Initialize Box Array
      boxes = new ArrayList<Box>(boxNumber);

      //Create Letters
      for (int i = 0; i < boxNumber; i++) {

      //Temp Boxes, spawn at random places on screen
      Box tmpBox = new Box(int(random(width)), int(random(height)),
15, 15, 150);

      // Populate the Coordinate Array for each letter
      tmpBox.getCoord();

      //Add each new Letter to the box Array
      boxes.add(tmpBox);
      }
```

Initialize the Boxes, add them to an array, and place them at random spots on the screen. This also populates the Coordinate array for each box. If you change the font, letter size, or anything else, color the box so you can keep the sizes relatively consistent. Odd Numbers work better for the coordinate arrays.

```
      //Blank image
      masterImg = createImage(width, height, RGB);
}
```

Draw the blank Master Image.

```
//Reset Function, will work out a different physical interface
//at a later time. Possibly Arduino Button.
void mouseReleased() {
      for (int i = 0; i < boxes.size(); i++) {
      boxes.get(i).bx = int(random(width));
      boxes.get(i).by = int(random(height));
      }
}
```

Reset function tied to the mouse releasing. Tying to the release keeps the reset from happening every frame the mouse is pressed. Good solution, might add another physical interface to control this.

```
//Calibration Controls
void keyPressed() {
      if (keyCode == RIGHT) {
      maxDepth += 10;
      }
      if (keyCode == LEFT) {
      maxDepth -= 10;
      }
      if (keyCode == UP) {
      minDepth += 10;
      }
      if (keyCode == DOWN) {
      minDepth -= 10;
      }
}
```

Calibration controls to adjust the depth map. This will be very useful during initial setups in new areas.

```
void draw() {
	//Load masterImg pixel Array
	masterImg.loadPixels();

	//Interating/Switching Variables
	int k = 0;
	int image = 0;
```

Load the pixel array and declare the iterating variables

```
	//Combo Loop
	for (int i = 0; i < masterImg.pixels.length; i++) {

	//Set up the current array at the start of the loop
	if (i ==  0) {
		image = 0;

		// Modulus to switch the array every other time.
	} else if (i % kinects.get(image).width == 0) {

		// check which array is selected
		if (image == 0) {

			// if it's 0, switch images that we're indexing
			image = 1;

			/* Subtract the width of the image from K so K
iterates over the same
			set of numbers twice each time. Use [k] to control
each image individually

			Only has to go back when going to the second image
(one on the right) otherwise
```

```
                    it can just keep going. */
                    if (k > kinects.get(image).width) {
                    k -= kinects.get(image).width;
                    }
```

Start of the switching loop. This loop switches which array is coloring the Master Image depending on where in the base image array we are. This causes one row of each of the base images to color onto the master at a time. It will color one row of the first, then one row of the second, then the second row of the first, then the second row of the second.

```
               } else {

                    // don't have to reset K when coming back to the
first image
                    image = 0;
               }
          }

          //reset K if it reaches the end prematurely.
          //This helps to solve OutOfBounds errors
          if (k >= currentArray.get(image).length) {
               k = 0;
          }

          //Assign depth values to Master image
          if (currentArray.get(image)[k] >= minDepth &&
currentArray.get(image)[k] <= maxDepth) {
               masterImg.pixels[i] = color(255);
          } else {
               masterImg.pixels[i] = color(0);
          }

          //Increment k
          k++;
}
```

The second half of the switching block, which includes a short statement that colors the master image directly into the Pixel Array.

```
      //Update Master Image
      masterImg.updatePixels();

      //Display Master Image
      image(masterImg, 0,0);

      //Physics for Boxes
      for (int i = 0; i < boxes.size(); i++) {
      boxes.get(i).lookUnder(masterImg);
      boxes.get(i).display();
      boxes.get(i).edgeBounce();
      boxes.get(i).collisionPoint();
      boxes.get(i).collisionVector();
      }

      // //debugging purposes and performance checks
      // fill(255);
      // textSize(20);
      // text(frameRate, 10, 10);
      // stroke(255);
      // strokeWeight(10);
      // line(11, 11, frameRate * 4, 11);
}
```

Two blocks here. The first handles all the physics for the Letters. These methods are outlined below in the CLASS METHODS section. The commented out section is used for performance checks and debugging. Check the framerate via text and a visual indicator so it is easy to see if something is drastically impacting performance, or if the hardware that is being used is up to the task of running this installation. Recommended to run this **with** the text during testing, then take it out afterwards.

BOX CLASS

This is a purpose-built class to handle the Letters in *Inhabitance*. Any adjustments that need to be made to the letters, their collision physics, or Sound Assignments should happen within this class.

```
class Box{

      //X and Y size for the collision of the box
      int bW;
      int bH;

      //Corner coordinates, determines the location of the box
      int bx;
      int by;

      //Threshold for the search to return a collision point
      int threshold = 220;

      //Center Coordinates of the Box
      int bCx = bW / 2;
      int bCy = bH / 2;
```

Instantiation of the Class Variables. These should never be changed.

```
      PImage box; //Box for collision detection area
      IntList px; //Array for collision detection
      ArrayList<Point> coord; //Coordinate array for Vector generation
      Point cPoint; //Point that feeds from collisionPoint into collisionVector.
      char letter; //Random letter variable, global so it can change
      int letterNumber; //Number associated with each letter.
      int noteNumber; // ASCII number to access the <notes> array

      SoundFile boxNote;

      float varAmp = 0;

      //objects for any movement related methods
      PVector location;
      PVector velocity;
      PVector acceleration;
```

```
      PVector friction = new PVector(0,0);

      // Boolean for sound methods.
      boolean hasMoved = false;
      boolean isMoving = false;

      float f; //friction coefficient, used in collisionVector
      float mass = 1.5; //mass, just to find out if it helps. (it doesn't really)

      PFont font;
```

The rest of the Class Variables. These also have no reason to be changed.

```
//Constructor. Called in SETUP
//Intakes spawn coordinates, size, color
Box(int x_, int y_, int sizeW, int sizeH, int bColor) {

      //Spawn Coordinate variables
      this.bx = x_;
      this.by = y_;

      //Size variables
      this.bW = sizeW;
      this.bH = sizeH;
```

Constructor. This initializes the Box object with a designated X and Y Coordinate, pixel Width, pixel Height, and Color. In the main code, the color can be changed to see exactly where the active collision area is. This helps with troubleshooting Collision Detection and Edge Bouncing. Keep these in mind if things are going wrong with collisions.

```
      //Initialize Collision variables
      location = new PVector(this.bx, this.by);
      velocity = new PVector(0,0);
      acceleration = new PVector(0,0);
```

These collision variables are CRUCIAL to the operation of the Box. Do not change these for any reason. There are other places to adjust the Velocity/Acceleration, so do not change them here. This merely initializes the variables for global usage later.

```
      //Create PImage for the Box
      imageMode(CORNER);
      box = createImage(bW,bH, HSB);

      //Color Letter
      fill(200);

      font = createFont("01_AvenirHeavy.ttf", 30);
```

The PImage for the Box, and the initialization of the letter/letter color. This is a visual change, and can be adjusted. Look into fill(); in the Processing documentation for information about how to adjust this setting. This is also where the font is called. If for any reason the font needs to be changed, simply load a .ttf file into the data folder, and replace the name in the createFont() section.

```
      //Generate random character
      letterNumber = int(random(65, 65 + 26)); //generate ASCII
values for char(). CAPS.
      noteNumber = letterNumber - 65; //convert ASCII values to ints
that can access <notes>
      letter = char(letterNumber); // Assign char() a random CAPS
letter

      boxNote = new SoundFile(master, notes.get(noteNumber));
```

This is the random generator. This generates a value between 65 and 90. These numbers then correspond with ASCII values in the char(); object. For optimization, this value is then modified and used to access the <notes> array in the main code. This is actually a clever bit of programming, as the sounds are never actually all loaded at the same time. The <notes> array contains strings of the names of each sound file. These are stored in the array in order corresponding to the letters of the alphabet. Then, the sound file is loaded in each individual box object using the Name data stored in the string.

```
      //Color Box pixels (mostly for debugging)
      box.loadPixels();
      for (int i = 0; i < box.pixels.length; i++) {
```

```
            //Make collision box transparent
            box.pixels[i] = color(bColor, 0,0,0);
      }
      //Update Box pixels
      box.updatePixels();
}
```

This is the end of the constructor. This is where any collision area color is assigned. Look at the color(); Processing Reference for more information about how to use this datatype.

CLASS METHODS

```
      void getCoord() {

      //initialize coordinate array
      coord = new ArrayList<Point>();

      //comb through the entire area of the box to assign every pixel
a coordinate
      for (int y = 0; y < this.bH; y++) {
            for (int x = 0; x < this.bW; x++) {

  //assign coordinates. USES CALCULATION TO MAKE SURE THE CENTER
  //COORDINATE IS 0,0. COLLISION IS EXTREMELY BUGGY WITHOUT THIS
                  coord.add(new Point(x - (bW / 2),y - (bH / 2)));
            }
      }
}
```

Populates the Coordinate Array with Points taken from iterating through each of the X and Y values in the Width and Height. While this doesn't return anything, it does allow for the Coordinate array to be accessed by the later collision methods.
The calculation in the add() function is actually extremely important, as the collision detection had trouble handling all positive numbers. This sets the center of the collision box as (0,0), and allows for the rest of the collisions to happen without trouble.

```
      // Display the Box(if visible) and Letter
```

```
      void display() {

      // Call box image. Necessary for loadPixels() later to work
      image(box, bx, by);

      //Call the text and character. This is where the text can be
      //customized visually

      textFont(font, 30);
      text(letter, bx,(by + bH));

      //debugging text goes here
      // String debug = "--";
      // textSize(20);
      // text(debug, bx, by - 1);
      }
```

Display Method. This is where the visual customization of Size can happen. Uncomment the debugging text to check any object-specific values (like amplitude, position, or velocity). This spawns small text above the letter that can be used to troubleshoot problems. This is often faster than debugging since there are so many complex for loops in the code.

```
      void lookUnder(PImage p) {

      //Generate PImage (and therefore a pixels array) for the space
under the box
      PImage r = p.get(this.bx, this.by, this.bW, this.bH);

      //create pixels array that can be referenced
      px = new IntList();
      px.append(r.pixels);
}
```

“LookUnder” method. This populates the <px> array and allows the collisions to work. The <px> Array is a continually updating array that contains the values from the Pixel array from the small section of the larger image that are under each box. (The larger image is PImage master; for reference). This method needs to be called in draw() in order

to properly function, as it needs to update every frame.

```
Point collisionPoint() {

     // Xand Y arrays to create a centroid coordinate
     IntList collisionArrayX = new IntList();
     IntList collisionArrayY = new IntList();

     // Xand Y sum variables that clear every loop for the average
     //calculation to take place
     int sumX = 0;
     int sumY = 0;
```

The beginning of the collisionPoint method. This returns a Point object for use in the collisionVector method.

The arrays and sum variables are intended to add stability to the collision, rather than operating on the very first pixel that meets the criteria, this takes the mean of any pixels that meet the criteria every 2 rows and uses that to calculate a “centroid”.

This centroid is a little more consistent, and results in collisions that feel more natural, and go in more intended directions.

```
 //scan the whole px array
     for (int i = 0; i < px.size(); i++) {
          // Check if any given pixel is brighter than the threshold
          if (int(brightness(px.get(i))) >= threshold) {

               //populate the X and Y arrays with the values from
the
               // coord array.
               collisionArrayX.append((coord.get(i).x));
               collisionArrayY.append((coord.get(i).y));
          }
```

Populate the X and Y arrays with any pixels that are over a certain threshold (listed earlier). Since the master image is literally only 0 or 255, this is largely useless, but again, the threshold adds some stability and consistency.

```
if (i % (this.bW * 2) ==  0) {

                //Adding the size check here to stop empty arrays
from trying to trigger a for loop
                if (collisionArrayY.size() != 0 &&
collisionArrayX.size() != 0) {

                // Add each value to the sum, to be divided for the
mean
                for (int o = 0; o < collisionArrayX.size(); o++) {
                      sumX += collisionArrayX.get(o);
                      sumY += collisionArrayY.get(o);
                }
```

Mean calculation for all entries in the X and Y arrays. A simple check for active indices in the arrays prevents a nullPointer or OutofBounds error.

```
//assign cPoint as the mean of the arrays rather than
// the first bright pixel in each pass. normalizes collision vector

                cPoint = new Point((sumX /
collisionArrayX.size()),(sumY / collisionArrayY.size()));

                //return the centroid for collision purposes
                collisionArrayX.clear();
                collisionArrayY.clear();
                return cPoint;
                }
          }
     }
```

If there **are** active indices, return a collision point of the mean of both arrays.

```
     //if there are no bright pixels, return null
     cPoint = null;
     return null;
```

```
	}
```

If there **are not** active indices, return null. This causes problems when any function tries to access cPoint when it is null, but a simple " if (cPoint != null)" solves that problem.

```
void collisionVector() {

	//Method variables.
	//Friction coefficient. Change from between 0.01 and 0.5 for
best results
	float f = 0.25;

	//Acceleration coefficient for how much speed picks up after
collision
	//Change between 8 and 20 for best results
	float aMult = 8;

	//speed limiter so things don't fly away
	//Change between 3 and 10 for best results
	float topSpeed = 4.5;

	//Method objects
	PVector force;

	//Directional Vector for collision direction
	PVector dir = new PVector();
```

Method variables for collisionVector. This is where much of the physics behavior can be modified. Change them in line with the instructions in the comments to adjust how the physics behave. Lower aMult for a lower speed from a collision. Lower or raise friction to decrease/increase the amount of time the letters take to stop. Change topSpeed to allow objects to go faster or remain slower (beware, topSpeed has adverse effects if changed too dramatically. Generally recommended to stay within the guidelines).

```
	if (cPoint != null) {

		//Get collision vector
```

```
          PVector colPoint = new PVector(cPoint.x, cPoint.y);
          PVector centerPoint = new PVector(bCx, bCy);

          //Calculate the direction between the center point and
Collision Point
          dir = PVector.sub(centerPoint, colPoint);

          //Normalize the vector, and multiply it to create
acceleration upon collision
          dir.normalize();
          dir.mult(aMult);
          isMoving = true;
```

Assign a PVector from cPoint, and a new PVector from the CenterPoint. These are then subtracted to get a new PVector in the opposite direction of the collision. This vector is normalized and multiplied to create an acceleration value.

isMoving is used later for consistency issues.

```
     } else {
          //If there is no collision, make sure the directional
vector is zeroed out.
          //Causes drift without this.
          dir.set(0,0);
          isMoving = false;
     }
```

Zero everything out if cPoint **is** null. Note that trying to call cPoint in this section results in nullPointer errors.

```
     //SET UP FRICTION
     friction = velocity.get();
     friction.mult( -1);
     friction.normalize();
     friction.mult(f);

     //Apply collision vector to acceleration
```

```
      acceleration.set(dir);

      //APPLY FRICTION
      friction.div(mass);
      acceleration.add(friction);
```

Set up the friction vector, apply the acceleration value, and then apply the friction to the acceleration value. This results in the object continually slowing down, since the CollisionVector gets called every frame.

```
      //UPDATE
      location.set(this.bx,this.by);
      velocity.add(acceleration);
      velocity.limit(topSpeed);
      location.add(velocity);
      acceleration.mult(0);

      //UPDATEPOSITION
      this.bx = int(location.x);
      this.by = int(location.y);
```

Simply continually updating all the physics information, and applying that to the position.

```
      float lowThresh = -0.04;
      float highThresh = 0.04;

      // Variable Amp
      float varAmp = map(this.by, 0, height, 0,1);
      boxNote.amp(varAmp);
```

Velocity threshold variables for use later, and a simple code to tie each sound file's volume with it's position on screen.

```
      //lotta && statements to find out if 2 values are within a
range
      if (lowThresh <= velocity.x && velocity.x <= highThresh &&
lowThresh <= velocity.y && velocity.y <= highThresh) {
            velocity.set(0,0);
```

```
            isMoving = false;
      }
```

There were issues with objects drifting rather than stopping, so this simple if statement tells each object to stop if the velocity reaches a certain threshold. 4 conditions are needed so the object stops ONLY if both X and Y velocity are within a certain range. Also changes the isMoving to false.

```
      if (isMoving == true && boxNote.isPlaying() == false) {
            boxNote.play();
            isMoving = false;
      }
```

Play the sound if the box is moving! Easy, simple.

```
      float noteThresh = 0.25;

      if (isMoving == true && boxNote.position() > noteThresh) {
            boxNote.jump(0);
            isMoving = false;
      }
}
```

There were issues with the note not playing again until after the letter stopped moving AND the sound finished playing. This simple code block solves this by jumping the playhead to 0 if the box is still moving **and** the note is through a specified number of seconds. In this case its ¼ second, but this can be changed if need be.

```
      //bounce off the edges
      void edgeBounce() {

      //Check if the box is on the edge (same for all)
      if (this.bx < 0) {

            //set location to the lower bound, invert and multiply
velocity to
```

```
            //avoid getting stuck on the edges
            this.bx = 0;
            velocity.x *= -4;
      } else if (this.bx + bW >= width) {
            this.bx = width - bW;
            velocity.x *= -4;
      }
      if (this.by < 0) {
            this.by = 0;
            velocity.y *= -4;
      } else if (this.by + bH >= height) {
            this.by = height - bH;
            velocity.y *= -4;
      }
      }
}
```

Bounce the box off the edges with 4 times their current velocity. This makes it easier to retrieve objects that have gathered in the corner.

EXTERNAL ASSETS AND DATA

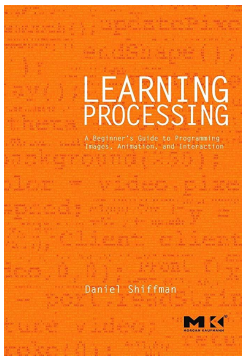
Assets

- In order to properly troubleshoot *Inhabitance*, there are a few assets that are important to have on whatever computing system is being used to run the installation.
    - Processing IDE or Visual Studio Code
        - This is the optimal way to edit any of the source codes or classes that come with *Inhabitance*. This project was created in Visual Studio Code, and can be easily edited in it, assuming the correct extensions are installed to allow integration with the Processing Language. This does not allow debugging, but the troubleshooting text blocks of code should allow for issues to be easily spotted.
    - Github or Github Desktop
        - Github is the main platform on which *Inhabitance* is distributed, so a base understanding and the proper software is important. Having GitHub Desktop makes cloning/forking the whole repository simple. This way, any updates can be easily pushed to any iteration of the installation without troublesome site visits.

Data

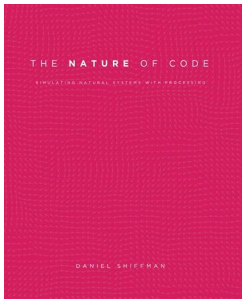
- There are two main blocks of data that are important to *Inhabitance*:
    - Audio Files
        - 16 Bit .wav files. 26 of them, names with the following convention
            - “(LETTER)__1.wav” (case sensitive)
        - If any of the files break from this structure without explicit documentation in the main code and/or github page, they will not work, and the application will not run. Because of the way the sound file retrieval is coded, they have to keep a consistent naming structure in order to smoothly retrieve, attach, and play sound files.
    - Font(s)
        - Any .ttf file can be loaded in, assuming the name is changed in accordance with the description on Page 24. If the name in the Box class cannot (or will not) be changed, loading any .ttf file with the name “01_AvenirHeavy.ttf” will work.
        - Do not load more than one font into the data folder of the installation.

FREQUENTLY USED LEARNING & RESEARCH MATERIALS



Learning Processing by Daniel Schiffman

- A book detailing the beginnings of how to utilize the processing language for animation, rendering, and interactivity. While this book does cover mostly the basics of coding, and how to get your first processing sketches up and running, it is an extremely helpful reference when doing even very complex sketches. No one keeps all every constructor, detail, and process in their head, so having this book on hand as reference was extremely helpful.



The Nature of Code by Daniel Schiffman

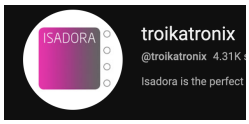
- A much more complex book written to explore the ways natural phenomena might be simulated in code. This book goes into detail about vector calculation, simulated physics, implementation of pre-built physics libraries, forces, and complex interactivity. As *Inhabitance* grew into needing its own Physics, this book was extremely helpful as a reference library for complicated topics.

Making Things See by Greg Bornstein

- A useful (though somewhat outdated) book about the fundamentals of using Kinect V1s as “computer eyes”. This book dives into exactly what kind of technology the Kinect contains, and how to implement that in different ways. While it does spend a significant amount of time on Skeleton tracking, it does contain valuable details about the data gathered by the depth mapping function.

Processing Documentation

- A full library of all included functions, classes, methods, and syntax that can be used in Processing. Does not include any Java utility packages, but those were worked out in a different research topic. Included in this is the JavaDoc for processing’s documentation, which includes readable lists of all classes and methods.



TroikaTronix Guru Sessions

- Used extensively while researching Isadora 3, though less useful after the switch to processing.

LinkedIn Learning - Learning Java

- Since Processing is an abstraction of Java, it was important to learn the basics of Java before diving into Processing. This included syntax, data types, and overall fundamentals of code. Through the entirety of the Winter of 2022 - 2023, this was singularly important to establish a base of knowledge before moving into the final steps of coding *Inhabitance*.

INHABITANCE

CONCLUSIONS AND FINAL THOUGHTS

In the 14 months of iterating through versions of *Inhabitance*, the piece has fully transformed. Starting as a simple silhouette with an Alpha Channel, it is now a complex marriage design, psychology, technology, space, and memory. During its evolution, *Inhabitance* has gone through four significant stages (including this final one). These stages are as such:

- *The Boulder (Spring 2022)*
- *The Hero (Summer 2022)*
- *The Cave (Fall 2022)*
- *Inhabitance (Spring 2023)*

In order to properly organize my conclusive thoughts on the piece and its progression, it is important to lay out the general structure of the evolution in each of the significant stages.

Spring 2022 - The Boulder

This was the first major step in the process. The foundation on which the rest of the iterations stand, and without this, none of the others could have become what they became. This all started with a singular technological idea: A Controllable Silhouette. What the silhouette would do, how it would be controlled, what kind of interactivity the piece would have, these were all still in nebulous states of "figuring it out", but the idea of a Controllable Silhouette was solid, and has been the connecting thread for each of the stages.

The Boulder also introduced the usage of Isadora, a very significant piece of software that would allow for easily manipulated images to be stacked and projected in very precise ways. Isadora, while simple on the surface, employed a node-based coding system, and provided a solid base of technical understanding for the rest of the iterations. This also brought the first of two uses of the Arduino. While this was not something that would continue through all the iterations, it did help to start thinking about the project in terms of code.

While *The Boulder* was technologically successful and exceedingly stable, the core issues with the work were with the conceptual and interactive ends of the piece. Fundamentally, *The*

Boulder actively rewarded non-interaction, with many users exploiting the use of a camera to "cheat" the system. Additionally, users would simply step off of the active area in order to circumvent the interaction designed by the FSRs and Arduino. Finally, the "reward" of reading Albert Camus's "Myth of Sisyphus" was not engaging enough on its own to keep participants from leaving once they uncovered the technological functions.

Summer 2022 - The Hero

Months later, the work would evolve into another stage, later titled *The Hero*. "Myth of Sisyphus" was removed and replaced with ten combinations of random words from a short excerpt of the Hito Steyerl essay "A Thing Like You And Me". This lowering of the barrier to entry allowed many more people to meaningfully engage with the piece, although what they gleaned from the work was not exactly "meaningful". The core text was still too esoteric for a pedestrian audience, even if the interaction was more intentionally designed and executed.

The Hero also introduced the usage of Kinect sensors to track participants, however the integration with Isadora was buggy and crashed often. While this was an important step forward technologically, the inconsistency dragged the whole of the piece down to what ended up as a half-baked idea with poorly implemented technology.

Fall 2022 - The Cave

The final iteration before *Inhabitance*, this brought together many of the poorly implemented ideas that were present in *The Hero,* and polished them into something much more consistent and stable. The Kinect sensors stayed, though Isadora still proved too unstable to run them. To handle this, the Processing language was implemented, and simple sketches (with poorly coded solutions) meant that *The Cave* was stable, unmonitored, for multiple hours. Isadora was still utilized for an added sonic component, adding an important layer of user control to the piece. Rather than operating the installation through a physical interface, participants themselves *became* the physical interface. This was the most successful usage of technology and interaction that the piece had seen. However, there was still a missing piece in regards to conceptual depth. The Controllable Silhouette was present, but it didn't have any real purpose. It was simply a visual set piece.

Spring 2023 - Inhabitance

The final version of the piece, and one that has taken the most technical work to get up and running. In preparation to get this iteration up and running, a baseline of knowledge had to be established in Java, Processing, and (unfortunately) physics. This iteration started out as a refined version of *The Cave*, with some added interactive features (such as a vocal component, or additional text), but after research and discussions about other Digital Interactive Installations, and how they presented their interactivity, it grew into something much more ambitious.

In late February, it was decided that interactive letters would be a conceptually meaningful direction to follow. This would allow users to leave their own messages, allowing them to directly influence the next participants that would come after them. Implementing this interactivity meant finding a consistent, reliable, and stable way to handle interactive objects; which in turn meant finding a suitable physics engine to handle collisions. This led down a weeks-long rabbit hole of learning Box2D (a simple pre-built physics engine that powers many mobile games such as Angry Birds), and reading Daniel Schiffman's "Nature of Code" cover to cover. After the long stint of testing, it became clear that a purpose-built physics engine was necessary, and work began on the Box class that handles all the physics in *Inhabitance*. Overall, this was a positive direction to move, as it meant a lighter weight code that could run on a wider range of machines with differing computing power.

All in all, the learning and iterating process for *Inhabitance* has been a fruitful one. Over the past year, I have been able to take a specific look at what exactly the work that I do is *about*, and why I keep coming back to this same topic. Highlighting this gave me a clear vision into exactly what I wanted from this piece, and in turn allowed me to focus on the most important parts of what makes this installation work.

Final Thoughts

I am thankful for the support given to me in the multiple classes I've iterated this piece in, and to all those that have given me meaningful feedback. Without the valuable input I've received, this installation would never have evolved past its first stages, and I would be left still wondering exactly what it is I make art about.