Browsing via Sonification

TAYLOR C. CUTLIP

West Virginia University, tcutlip3@mix.wvu.edu

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Part of the Other Computer Sciences Commons

Recommended Citation

CUTLIP, TAYLOR C., "Browsing via Sonification" (2019). Graduate Theses, Dissertations, and Problem Reports. 7394.
https://researchrepository.wvu.edu/etd/7394

This Problem/Project Report is brought to you for free and open access by The Research Repository @ WVU. It has been accepted for inclusion in Graduate Theses, Dissertations, and Problem Reports by an authorized administrator of The Research Repository @ WVU. For more information, please contact ian.harmon@mail.wvu.edu.
Browsing via Sonification

Taylor Cather Cutlip

Problem Report submitted to the Statler College of Engineering at West Virginia University in partial fulfillment of the requirements for the degree of Masters of Science in Lane Department of Computer Science and Electrical Engineering

Frances Van Scoy, Ph.D., Chair
Roy Nutter, Ph.D.
Cynthia Tanner, M.S.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2019

Keywords: sound, acoustics, browsing, Godot, GDScript, Kivy, touch, gestures, tool

Copyright 2019 Taylor Cutlip
Based on unexpected results in her previous research, Dr. Frances Van Scoy became inspired to develop a tool that allows the user to navigate spaces using auditory instead of visual cues to detect objects or anomalies in a given space in an effort to overcome the encountered obstacles. This problem report details the work of one of her grad students in exploring different open source software for developing the tool as well as research into gestures and other considerations when furthering this research into the third dimension at a future date.

“When the tool is completed the result will be, we believe, an intuitive browser for sonified spaces that will make understanding of simulation results easier.”
## Contents

- Problem Statement ................................................................. 2
- Project Context ............................................................................. 3
  - Visualization vs. Sonification .................................................. 3
  - Acoustic attenuation ................................................................. 5
- Case Study ...................................................................................... 6
- Specification of the tool ............................................................... 8
- Requirements .................................................................................. 9
  - Design Requirements ................................................................. 9
  - Annotation Skein ....................................................................... 11
  - “My vision is eventually a touch-screen based tool.” .................. 11
  - Functionality of tool ................................................................. 12
- High-Level Software Design ......................................................... 13
- Selection of Software Platform ................................................... 13
  - Kivy vs. Godot ................................................................. 13
  - Gamification of the Problem .................................................... 15
  - Choice of Software ................................................................. 16
- Choice of Gestures ........................................................................... 17
  - Gestures ................................................................................. 21
  - Suggested Gestures ................................................................. 22
- Low-Level Software Design ......................................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collision Detection with Karnaugh Maps</td>
<td>23</td>
</tr>
<tr>
<td>Layering</td>
<td>27</td>
</tr>
<tr>
<td>Movement in Fox Hunt</td>
<td>28</td>
</tr>
<tr>
<td>Data Structures</td>
<td>28</td>
</tr>
<tr>
<td>Some Thoughts on Future Usability Testing</td>
<td>29</td>
</tr>
<tr>
<td>Future Work</td>
<td>29</td>
</tr>
<tr>
<td>Transition to 3D</td>
<td>31</td>
</tr>
<tr>
<td>Experiments to Conduct</td>
<td>31</td>
</tr>
<tr>
<td>Future Modifications</td>
<td>32</td>
</tr>
<tr>
<td>Problems Overcome / Lessons Learned</td>
<td>33</td>
</tr>
<tr>
<td>Conclusion</td>
<td>33</td>
</tr>
<tr>
<td>Appendix</td>
<td>34</td>
</tr>
<tr>
<td>Source code</td>
<td>34</td>
</tr>
<tr>
<td>Using Godot</td>
<td>49</td>
</tr>
<tr>
<td>References</td>
<td>58</td>
</tr>
</tbody>
</table>
Joint research was carried out by computer scientist Dr. Frances Van Scoy and physicist Dr. Martina Bachlechner. In their paper, a metal is examined after being stretched. As metals have a crystalline structure, the molecules are typically distributed in a very meticulous structure. The stretching of the metal, however, pulls on molecules and stretches the lattice, making it have abnormalities. These gaps in molecules were examined by the professors. Instead of molecular displacements, however, broad gaps of sinusoidal-like waves appeared. These artifacts gave false positives of the molecular displacements. This spurred the idea for another way to examine the molecules.

Being a writer, Dr. Van Scoy is used to creative problem solving. She came up with an alternative method of examining the particles that is both a fresh way to look at data and possibly better. The thought is simple, but the implementation is not: instead of looking at the particles, why not listen to them?

If there were a quick method to browse through the experiment space and listen to the displacements instead of visually inspecting them, new or more reliable results might occur. This inspiration resulted in the goal of a project to develop the foundation for a tool capable of detecting differences or displacements between data in large data sets. Using these auditory clues to navigate spaces has been termed “sonification.”

Outlined in this paper is the tool created by the author along with the knowledge of the engine used and a clear direction for further work involving this project in hopes that this will aid in the improvement of the tool and its use.
in research toward furthering knowledge in this research area.

As the software is new for both the author and his committee, another main goal for this paper is to serve as an introduction to the engine itself and a guide explaining the uses and benefits of the software chosen for new users wishing to learn the engine. This paper accompanies the sonification space-browsing tool in order to fulfill the requirements in part of a M.S. Computer Science degree at West Virginia University.

**Problem Statement**

While Dr. Van Scoy was working with Dr. Bachlechner previously, several problems arose in the research. The spaces (rips) in the metal were not where they were expected to be after the stretching had occurred; they were being filled and emptied, appear to move in a very unexpected manner. Also, strange artifacts of ribbons appeared, though those were most likely due to round-off errors in calculations of the model. Lastly, the space was explored using two joysticks as a controller, which was not ideal.

Therefore, can a mobile tool with intuitive controls be developed to browse spaces by sonification that does not encounter the previous problems that occurred while browsing by visualization?
Project Context

Visualization vs. Sonification

Before detailing the manner in which one browses, it is imperative to solidify a notion of what it means to browse. The author proposes that there are two parts to browsing: both moving through a defined space and examining the area of the space that is being traversed. Think about shopping for food. One might meander down each aisle of a grocery store, looking for items on their list of things to buy there. While browsing, they search each item displayed on the shelf and look to see if it’s a match for one of the desired items. This is an example of browsing by visualization. If that same person were auditorily- or visually-impaired, how might they browse for items on their list? One way might be via the lesser explored method of sonification – using auditory cues to navigate the space around them instead of visual cues. One big difference between these methods is area of input to the user. Since human eyes are on the front of their head (like most predators), they have a relatively narrow field of vision compared to prey that needs to look more widely around themselves for threats. Owls in particular can improve their field of vision by rotating their heads more than humans can, but this is not always the case. When flying a plane for example, the pilot’s field of vision is blocked even when turning his or her head; it’s impossible to see behind their aircraft. The weather is another factor that can impair visual navigation. Sunlight can make glare on a screen rendering it temporarily unreadable (or even temporarily impairing sight as a whole of the user); fog or mist can also interfere with screen reading.
When there is contradictory perception between vision and sound, auditory cues can continue to provide information for our brains to locate sounds; if music is coming from behind a rock, we cannot see the source of the sound, but can still hear it.

Other senses have been explored to counteract the foibles of vision, such as using touch with a tactile belt. An apparatus worn around the waist would send pulses that the wearer feels by touch, giving information about their environment so that they can navigate more reliably.

Using a sonification approach, one can expand this “field of hearing” to 360 degrees. It should be noted that similar to vision’s sharpness dropping the further away the viewed object is, the sounds around a listener are not pinpointed to an exact location either. However, if a barrier is blocking view of an object, the sonification will still help locate the object, though it may be a bit muffled. While similar in many ways, each approach does have its idiosyncrasies, the question is whether sonification is more beneficial to browsing and detecting anomalies within spaces than visualization. It should also be noted that humans may suffer impairments to both vision (e.g. color blindness) and hearing (e.g. deafness), leading one method to be preferable for navigating a space despite its weaknesses. The question explored by this report is Browsing: Visualization or Sonification?
Acoustic attenuation

Much like light, sound waves also deteriorate over the distance they travel. This results in a slight difference in how each ear hears a sound, despite our ears being relatively close to each other on our head. This differential due to auditory attenuation, however, is actually beneficial as it helps us determine from where a sound is coming (though echoes from surfaces and other factors may still make it difficult to pinpoint a location). This neurophysiological adaptation, known as sound localization, has been an important biological evolution for survival, whether it’s locating food or predators.

Sound waves fluctuate the pressure in our ear with a slight delay relative to the other ear, based on location. A difference in amplitude or magnitude of the sound waves is also detected by our neurons. Ultimately, these interaural differences lead to our brains deducing the location of the sound source. The pressure and amplitude deltas are translated into “over to the left” for example and other similar thoughts when trying to perceive the location of a sound. This suggests that azimuth perception is an innate skill, whereas deciphering the elevation of the sound is more of a learned skill. When designing a tool to browse areas via auditory clues, there should be a focus on rotation-based browsing on the plane perpendicular to the user (azimuth) instead of a tilt up or down relative to the user (elevation). The Baldwin effect backs up this claim because it mentions that innate skills (i.e. azimuth detection) are more effective in stable, unchanging environments (such as static beacons that are not constantly moving).
The next step in auditory evolution is to guess where the sound will be moving to next based on its current position and other factors. Instead of a relative location from the listener, our brains need to translate to an absolute location to eliminate factors such as our current movement that would be less useful when predicting a relative location. As any student of physics might predict, speed and trajectory predictions come into play with this task.

The more unconscious observations of interaural time, amplitude differences, and spectral shape are contrasted by more subconscious measurements of azimuth, elevation, and distance. Together, our brains translate these signals into place, speed, and trajectory. Some questions still unanswered by “psychoacousticicians” are which of these observations are relevant to our brains? Even if we can identify features of sound location, are they useful? Are they all dependent on each other? Do the base instincts of survival and procreation alter the input data to just assessing based on those themes instead of other aspects of life?

Case Study

For a case study of browsing by sonification, the author examined the training exercise some amateur radio enthusiasts partake in called “Fox Hunting.” The activity is named for the sport from hundreds of years ago that inspired the activity of the same name. A group of “hunters” would follow dogs specially
trained to track down foxes by scent in hopes of catching one. The practice was eventually considered controversial as it involved the chasing, catching, and killing of real foxes, and was banned in many countries, although it may be currently still legal in the United States.

A form of fox hunting evolved among the amateur (“ham”) radio enthusiasts, also called fox hunting (or transmitter hunting). In this version of the sport, the role of the fox is played by a radio transmitter hidden in the field (designated play area). The hunters use their radio direction finding techniques in order to locate the transmitter and thereby “hunting the fox.” There are several versions of the activity that vary in how or where the hunters use their skills. When using directional receivers, a hunter can tell from which direction a transmitter is emitting by the strength of the signal received by the receivers. The signal is the strongest while facing toward the transmitter. This creates a line between the receiver and transmitter. By taking a second reading elsewhere, a second line is drawn. The intersection of the two lines is where the transmitter is located. Human ears can be thought of as directional receivers, so this method applies to fox hunting in a real-life setting. This modernized form of fox hunting is part of the inspiration for sonification.

Another technique examined was that of Triangulation. When an earthquake occurs, a monitoring station can calculate the distance to it based on several factors including the time observed and known wave propagation rates. This results in a distance measurement to the origin, or epicenter, of the earthquake. Since only a distance is known, this creates a radius around the
station of where the earthquake could have originated. By having a second station's distance calculation, the two circles formed from their radii will intersect, narrowing the location down to two possible intersections. With a third station’s calculations added to the knowledge base, the origin can be determined. This is somewhat similar to the above method using directional receivers but is more rudimentary as direction cannot be inferred from an observation and three sets of data are needed to determine the location. Ultimately this technique was not relevant enough to be helpful in solving the problem of this report since the goal is for browsing areas to find anomalies and not for pin-pointing the location of one specific object.

**Specification of the tool**

(why we need/want the tool; that is, what we want to do with i) In order to explore a sonified space, a new tool needed to be built from the ground up since it needs to provide specific functionality to aid in exploration. Thus, the two main goals of the tool are both the ability to navigate the space and to record observations while exploring. More detailed specifications arose via those two goals. In essence, two concentric circles will be moved around a space to determine the listening area of the tool. The user should be able to change the radii and control where the circles move inside the listening space. The space should also be able to re-scale or otherwise “prettify” itself after movement places the circles in an “undesirable” location (such as half of it being off the screen).
Additionally, a sector can be overlaid on top of the area in order to focus the user’s hearing onto a particular sector of the area. This area could be marked as “interesting” to the user—whether that’s good/bad data or just nice-sounding music is up to them. At these interesting regions, the user should be able to create an audio recording of his or her thoughts on the area. The recording, location, and the route traversed by the user to get to that point should all be saved within the navigable skein.

These further specifications lead to the requirements listed below.

Requirements

Design Requirements

The design functionality of the tool includes:

• Change the radius of either or both circles
  o The result may be that part of either or both circles extends beyond the display region.

• Movement by dragging (or change by clicking) the shared center of the two circles
  o The result may be that part of either or both circles extends beyond the display region.

• Rescale the display so that entire outer circle is displayed at maximum size on the screen
• select a wedge of the larger circle
  
  o Sounds rendered are those within the new segment, the region within the wedge and the outer circle but not within the inner circle.

• Mark as a segment of interest
  
  o Save coordinates of center of circle and points on outer circle that define the wedge.

• Create new circles based on current segment
  
  o Outer circle is of maximum size to fit in display and include every point in the segment, and radius of inner circle is 10% that of outer circle.

• User can change default center to be, say, 10% to the right and 10% below where it would otherwise be in order to display sounds at or near center of wedge and can change all three default percentages (including sign).

• Record voice message from user
  
  o Allow user to record message, likely commenting on observations of sounds in the segment).

  o By default, the display of sound from the simulation is muted while the user is recording.

  o The intent of allowing voice recording is to avoid causing the user to have to use a keyboard or paper and pencil to record observations, conjectures, and so forth.

• end recording voice message and save
Annotation Skein

Borrowing a term from Inform7, there should be a skein-like structure (a navigable tree of memorized steps in exploration) for replaying browsing sessions. In other words, there should be a tree structure where the tree’s nodes are the recorded actions of the user, and the user should be able to freely traverse the paths to revert to old actions. Essentially, it is as if a player created a new save file in a game every time a new action was performed.

“My vision is eventually a touch-screen based tool.”

Thinking of the technology of the time, the first envisioning of a control device for the tool involved using a joystick for navigation. Joysticks are good for movement and precise control. However, they are somewhat limited in their degrees of freedom. When it comes to movement in a 3D space, if considering an arcade-style controller then multiple joysticks would be needed. One could be for simple movement upon a plane and the other could be for rotation and turning.

An alternative if the user wanted to have complete freedom of movement in the space is to have “twist” recognition as flight control joysticks have. This would still require, however, another controller for thrust. Even if thrust were built in to the joystick controller, another hand would still need to be involved for complete control.

The idea of using touch controls for movement thus came up as it would
contain all movement controls within a single device. Not only that, but all controls would be able to be performed with one hand. Additionally, we would be able to come up with our own gestures for control; all things that are unable to be done with a joystick controller.

Functionality of tool

The overall functionality of the tool is not overly complicated; it can be split into two basic functions: movement and recording.

Movement is in two dimensions, with plans for expansion into 3D. Details on the gestures used for movement can be found later in this paper. This is the most important feature as it is what allows you to browse the space in general while recording your observations via the tool is not strictly necessary for the tool to operate in general (though the ability to record is indeed a requirement.)

Recording the observations of the user via the tool takes the form of the annotation skein mentioned previously. The user can open a menu, via button or touch (to be determined later), and “save” a point in time and space that is linked to previously saved points. Also, the user will be able to record a voice message of their observations at that particular save node if the hardware running the tool has microphone capabilities.
High-Level Software Design

Selection of Software Platform

There are several aspects of a design platform to consider when picking the best one for this project. Being open source is a must. Another is 2D modeling capabilities while maintaining the ability to be expanded into three dimensions. The ability to provide a proof of concept for further expansion via a mobile, touch-based tool is also considered. Consideration of the long-term plans of this tool are also taken into account: using music to explore models of how atoms react in metals that are stretched to both recreate and supplement previous research.

Kivy vs. Godot

Python

Python is a popular modern open-source programming language used by many companies (such as Google) for a variety of applications including the architecture of YouTube.com and the game EVE Online, the self-described #1 Massively Multiplayer Online game in space. In order to model galaxies lots of data needs to be processed and displayed, which made this a promising choice.

Kivy

The first software investigated was Kivy, a platform for making applications and GUIs, since its code is in Python. It was originally built upon PyGame, another python-based known platform used by the author.
Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.

**Kivy --- PROs**

Kivy was a good starting point in that it used a familiar language. It also was made for GUIs, though specifically for applications and not necessarily games. It also has the capability to recognize multi-touch gestures. It seemed like a good idea at the start of this report but was eventually found to have too many “con”s to prove viable as a usable language for this project.

**Kivy --- CONs**

1. It is very hard to use
2. It has bad support
3. It has a non-active community
4. It has poor documentation
5. It takes forever to install because of the above
6. Most programmers use it for a project or two, but do not stick with it (to develop a deeper understanding and not just “I know enough to get it to work…” )
Game Engines

While a nice framework to make GUIs is a big help, so is software that is intended for rendering games, especially if it has a nice way to create GUIs and HUDs.

Gamification of the Problem

Before the tool was created, the author decided to gamify the tool’s prototype as a fox hunting game, appropriately called Fox Hunt, using Python. The initial idea for Fox Hunt was as follows: foxes are generated with a particular song and song modification (or none). While some foxes may be similar-sounding red herrings, only one particular fox will have the tune playing correctly. The goal was for the player to navigate toward the correct fox using their hearing. If a wrong fox were encountered, the fox could be “muted” to eliminate one of the distractions. While this might not help the player know where the correct fox is, there would be less interference making it indirectly helpful. The score of the player was how long it takes him or her to reach the correct fox. No penalties were associated with “wrong” foxes.

Since the focus of this report is on the tool, the game was simplified before being altogether scrapped as an idea. As of the latest version of Fox Hunt, the game had the user move via arrow keys on a keyboard to try to discover the locations of the foxes. All foxes are inside the “field,” denoted by a grass-
colored Kivy TextureRectangle. There were three foxes that were “singing” the same song, but they were invisible! When the player moved closer to a fox, the song will become louder. If the player could not discern the location of the correct fox, movement via the left/right and up/down keys could be used in a trial-and-error fashion to try to tell which direction it is. When the fox was eventually stumbled upon, they would appear and stop singing, letting the player know that he or she found one.

**Choice of Software**

Godot (pronunciation is perhaps debatable, similar to the age-old debate over the pronunciation of “.gif”) offers dedicated 2D and 3D rendering engines. As a result, the base unit for 2D scenes is pixels. Even though the engines are separate, you can render 2D in 3D, 3D in 2D, and overlay 2D sprites and interface over your 3D world.

**Why Godot?**

1. Designed for particle rendering
2. Language similar to python; support for multiple languages
3. Open Source
   a. No restrictive licensing
4. Cross-platform
5. Active Community
   a. Many questions answered online
   b. Bug reports are responded to and not ignored
c. Lots of game jams to learn the engine better
6. Easier to learn than the alternative investigated, Kivy
   a. Good tutorials on their website (and on YouTube from users)
   b. Has a GUI interface for learning the system
   c. Has a scripting interface that can be used instead of, or along with, the GUI interface
7. Separate 2D and 3D engines
8. Very easy to download and view/run others’ projects!

Upon weighing the benefits between Kivy and Godot, it was decided that using Godot would be the best path forward.

**Choice of Gestures**

Many sets of illustrated touch gestures exist online. Shown below is one such design by designer P.J. Onori. While it seems that most people map their own functionality to these gestures, the gestures used themselves are usually the same between apps.
Figure 2 – Gestures by designer P.J. Onori, found at https://bigmedium.com/ideas/cue-icons-teaching-touch.html
Clearly the most numerous gestures are pressing one or two fingers down and sliding them a particular direction. However, just because these de facto industry standard gestures are commonplace doesn’t mean they are correct. We should tailor our gestures to our program and use the most intuitive gestures possible.

Luckily, Godot lets us program in our own controls.

<table>
<thead>
<tr>
<th>Movement Type</th>
<th>Movement</th>
<th>Description</th>
<th>Suggested Gesture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planar</td>
<td>Forward / Backward</td>
<td>Walking forward or backward in a straight line</td>
<td>Swipe Up or Down</td>
</tr>
<tr>
<td>Planar</td>
<td>Strafe Left / Strafe Right</td>
<td>Stepping to the left or right without turning your body</td>
<td>N/A</td>
</tr>
<tr>
<td>Rotational</td>
<td>Yaw</td>
<td>While keeping your body upright, turning to the left or right</td>
<td>Rotate</td>
</tr>
<tr>
<td>Rotational</td>
<td>Pitch</td>
<td>While driving on a hill, the angling up and down of the vehicle</td>
<td>Two-Finger Swipe</td>
</tr>
<tr>
<td>Rotational</td>
<td>Roll</td>
<td>Doing a theoretical cartwheel while staying in the same spot throughout</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2 - Movement Types and their Descriptions

While not every movement is mapped to a gesture, the ones that are sufficiently give the user movement capabilities in 3D space without sacrificing
too much freedom of movement. The overall resulting movement capabilities are similar to driving a car (no strafing or rolling) with the added benefit of yaw. An illustration of the rotational movements is shown below.

Additional functionality can be programmed on top of these gestures; zooming in and out can be mapped to the pinching gesture for example, or saving the user’s session with a three-finger slide.
Gestures

As mentioned earlier, custom gestures can be developed for innovative ways to browse spaces and record thoughts. While many touch gestures have been relatively standardized for 2D interfaces (pinch for zoom, drag for scroll, etc.), there is wide speculation on what gestures should be used for 3D/virtual environments. In fact, many papers at Human Computer Interaction (and other similarly themed) conferences last year had their own 3D gestures for navigating spaces. On one hand, the “best” controls for 3D should be used when the tool is transitioned to 3D, however having two sets of completely different controls might not be intuitive to users. Also, gestures that become more commonly used are not necessarily better or more intuitive. When looking toward Virtual Reality games for inspiration, interesting controls that do not use gestures at all could be used. For example, motion tracking of the user’s head can direct the tool to listen in particular directions. Perhaps the user could lean forward to listen more closely and “zoom in” to certain parts of the space (though users with limited physical movement capabilities might not be able to do this).

One new instance of a hand-tracking device comes from Microsoft; a product called Digits. Some main uses outlined in their video were controlling robotic arms in unsafe environments, gaming/computing from a relaxed state, such as on the couch, without the need for a controller or mouse, and classification of sign language letters and gestures for interpretation purposes.
A particular set of gestures called “surface gestures” should also be tested by users. Surface gestures are a mix of 2D and 3D gestures. Basically, a common 2D gesture is performed in 2D or 3D space. For example, a user could “pinch” a group of objects and “drag” the collection in 3D space, dropping it off where the pinch is released. Or perhaps a closed fist can be placed in front of a user, and when he or she opens the fist, the hand can then be rotated to rotate the space about the axis perpendicular to the palm of the user. Since obviously a hand cannot be rotated very much, the stipulation of the hand needing to be open allows the user to quickly close their hand, rotate to a more preferable angle, then repeat the gesture if necessary.

Altogether, a combination of 2D, 3D, and VR gestures should be used, taking the best gestures from each space, assuming they are intuitive as a whole to the user.

**Suggested Gestures**

Sliding with one or two fingers should be a gesture that enables the user to “walk” forward while listening to the sounds. If the user wants to turn left or right, they should be able to twist two fingers in a counter- or clockwise direction, respectively. Godot supports gesture recognition, along with customized gesture detection, allowing for more functions to be programmed at a later date.
It is a necessity that gestures be both easy to perform and intuitive to the user. Trying to “pinch together” four fingers at once, for instance, will never become a good, standardized gesture because of its awkwardness to perform. Also, the intended set of fingers used has an effect upon the gesture’s usability. For example, having a 3-finger gesture with the thumb, index, and middle fingers is more easily carried out than starting with the pinky side of the hand. Note, however, that this is still a relatively awkward gesture in general, despite the more easily manipulated fingers being used.

Godot sends touch event signals to its objects, which can be directly sent to a callback function of the programmer’s desire. The number of points making contact with the screen as well as their vectors are passed through.

The basic controls would need to allow movement by the user about all axes in three-dimensional space along with movement on a plane in order to be sufficient for our needs. With the possibility of excluding rolling, as the author feels this is the least useful mode of movement and can be excluded for simplicity in design of controls.

**Low-Level Software Design**

**Collision Detection with Karnaugh Maps**

One problem encountered during the design/testing of the (tool?) was when to play or not play a sound based on the position of the fox(es). Since there are two rings (outer and inner) that can be controlled, along with a sector between those rings (the orientation, so to speak), along with a “listen closely” mode
that can be toggled on or off that changes the functionality, needless to say things got complicated fairly quickly. The first approach was to write down the logic by hand, but this seemed a bulky and suboptimal approach.

In order to solve this problem, a logical complexity technique known as a Karnaugh Map (or K-map) was used. K-maps, used with a focus on switching circuits, are useful when the desired output is known but not what the simplest or most reduced logical statement would be to obtain those desired outcomes. First, a logic table is created with all the variables and the desired outcome listed for each scenario. The rows in which a sound should be heard are highlighted in green in the below.

<table>
<thead>
<tr>
<th>Outer Ring (A)</th>
<th>Inner Ring (B)</th>
<th>Sector (C)</th>
<th>Auscultate (D)</th>
<th>Play Sound?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Now that the scenarios have been dissected, they can be translated into a special table, seen further down in the report. The pairs of variables A-B and C-D are listed as the columns and rows, respectively, with a 0 representing a False value and 1 representing True. These truth values can be interpreted as whether the object is within that variable's hit box or not and if the user is listening closely or not. After placing the values in the table, the idea is to clump together any “true” groups. For this example, any “yes” scenario can be grouped up with other yeses, so long as the basic rules are followed: the table wraps around both axes (as if mapped on a torus) when clumping together truths, any group created must be of contiguous values formed by making rectangles on the torus (no jagged edges sticking out), and each group must contain a total number of cells equal to a power of 2. The rectangles may overlap, but making the largest possible rectangles reduces the logic the most, so optimal solutions will have the largest (and simplest) groups. This is shown in the table, with one group in red, a second in blue, and the overlapping cells shown in purple.

<table>
<thead>
<tr>
<th></th>
<th>AB</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>

Once these groups are decided upon, the variable configurations are noted. Each separate rectangle is an addition to the logic statement (thus making it
more complex). The consistency of the variables used in each rectangle state how the system should be configured. For example, looking at the red rectangle, it can be seen that the bottom two rows of the table are contained within the rectangle. Since the cells cover all four columns, the variables A and B vary in each case and are not helpful in determining what the red rectangle is. Looking at the rows, however, it can be seen that CD = \{11, 10\} are the only two cases covered. Since C is constant in both cases, C is the added to the logic statement for figuring out this system. The variable D is not added because it can be either true or false, depending on what row is examined. Therefore, we can not reduce the logic since we can not definitively say its value from the rectangles selected. The only thing that can be said with certainty for any cell in the red rectangle is that C is true since it is always a 1.

So far, the logic statement is:

\[ C \]

Remembering that each rectangle adds to this equation, we examine the blue rectangle and will create a logical disjunction (an “or” statement) with the current formula, effectively “adding in” the other half. The blue rectangle covers the conjunction (an intersection, or “and” statement) of the two rightmost columns and both the top and the bottom row since we can wrap around the table from the bottom to the top. Looking at the columns we see that A is true in every case but B varies. Looking at the rows, we see that C also varies, but D is false in every case. Therefore, the blue rectangle gives us:

\[ A \land \sim D \]
Putting these two pieces together, our final logic statement is:

\[ C \lor (A \land \sim D) \]

This means that we want to play a sound whenever \( C \) is true, or whenever both \( A \) is true and \( D \) is false. Substituting the variable names back in, this translates to the notion that we should play a sound any time the particle is inside the sector between the inner and outer rings, or when it is within the outer ring but the auscultate mode is OFF.

**Layering**

Sprite (texture) layering can be used to create shaders for images. An image is combined with a mask in order to remove desired parts of the image.

Layering is an important concept for video game development. Not only does an engine need proper occlusion for suspension of disbelief, but layers can facilitate object interactions. For instance, the Inner and Outer rings described above should have no collision between themselves, or else sounds would go off every second. This was achieved by placing the objects in separate layers so that they would never interact. In order for them to both collide with a fox, the fox was placed in both ring layers so that they would be individually triggered by it. This technique can also be useful for groups of objects, such as “monsters” or “NPCs.” If playing a platformer such as Mario, for instance, the game designer would most likely not want the enemies snatching up coins before the hero can get to them (or perhaps this could be a clever twist on an old game... if coded intentionally!)
Movement in Fox Hunt

Normal movement in some 2D games is player-centric; the environment is fixed and the player is the person moving around on the screen. However, the first thought of the author was that it did not make sense to have the player move about the space. Instead, the world itself moves in response to the inputs of the player, who is placed in the center of the screen and is completely stationary. In this manner, the auditory attenuation works as expected. The trick to implementing this style is to have all of the objects (foxes) in the world move with a negative velocity to how the player would normally have moved, given an input. Instead of the player moving up on the screen, everything else moves down on the screen, toward the player. This method was changed to actually moving the user around the space when designing the tool in Godot.

Data Structures

Below are some of the data structures used in Godot and what they do. The source code for all of the author-made structures can be found in the appendix.

Area2D – This is simply a designated area in two-dimensional space. It is mostly used as an abstract base class.

CollisionRect2D – This is a shape (rectangle in this case) used for collision
Particle2D – This is a created class for emitting noise in a two-dimensional space, also referenced as a “beacon” in this paper for generic dimensions. It is what emits sound for the user to find.

OuterRing2 – This is a second iteration of the Ring class, created by the author, for detecting beacons colliding within a ring shape in order to determine if a sound should be emitted or not.

User – This class that inherits from Camera is simply the representation of the user and contains controls for movement.

Some Thoughts on Future Usability Testing

Future Work

Years later in one of her papers, Dr. Bachlechner models different kinds of stresses in molecular dynamic simulations involving many millions of atoms. Although she had success at examining the stress distributions, it took 10,000,000 simulations on parallel computers to achieve these results. Also, the molecules were not displayed on a screen in real time. This paper gives hope to the project, but more improvement in current technology still needs to happen before the full tool can be realized.

Ideally, the perfect tool would be able to browse the space of all the molecules in the lattice to detect where there are some that are “out of place”
(dislocations, faults, etc.) by browsing the lattice space both via sonification and visualization in order to compare the results. This would require a very large number of pixels to be displayed and was not feasible under the technology at the time of the research.

Looking at more current technology, assuming a range of screen resolution from the author’s current home computer to that of the “4K” (Ultra HD) televisions (which are by no means a common household resolution but rather a higher end, newer TV resolution), the current number of pixels on a screen is estimated to be between ~2 million and ~8 million. Additionally, not only would the molecules have to be displayed on the screen but also the spaces between them and the orientation from the 3D projection, as the user would be using browsing these data in an effectively 3D space. Another worthwhile mention is that this tool ideally would be mobile and support touch gestures, so a laptop or tablet screen resolution would be more likely.

Modern technology has improved since the research was first performed, and putting together several large monitors attached to CPUs with enough RAM could possibly display these atoms. However, the requirement of the tool being mobile would invalidate this solution.
Transition to 3D

Since Godot is being used, transition to three dimensions will be relatively easy. Godot features a separate 3D engine built into its software that pairs perfectly with its 2D engine.

Experiments to Conduct

Several experiments can be conducted to demonstrate the capabilities of the tool. Taking a page out of the game design book, these experiments should be successively more complicated as the user gains familiarity with the system and controls. As such, I would call the first few experiments tutorial experiments that every user should follow.

The simplest experiment would be having one beacon in the space that is on the same plane as where the user starts to keep things in two dimensions. This would allow the user to get experience controlling the tool and using their ear to find a particular spot in space.

Next, I would have several beacons, all with the same melody and rhythm, that are playing at the same time. This would allow the user to get used to hearing sounds from multiple places at the same time. This allows for getting used to hearing beacons from multiple directions at the same time instead of one at a specific location. As there might be more than one location of interest to the user in the same experiment, this tutorial introduces several stimuli without distractions of incorrect signals.

Then I would have one particular beacon amongst a field of “bad” beacons
that are emitting a different sound. This would most closely resemble expected normal operation for the user in non-tutorial experiments without being an actual experiment itself. This would introduce distractions as the user learns how to tune out unwanted noises.

Lastly, I would repeat the above tutorials with the addition of using the third dimension. Adding a dimension obviously complicates the experiments, including the tool’s controls.

**Future Modifications**

By continuing to add non-programmed gestures, we can expand the functionality of the tool and ease of its use. Non-movement controls such as recording an audio observation can be mapped to custom gestures. There could also be preferences of the user selecting which gestures correspond to which movements to allow re-mapping as he or she sees fit.

Another modification is that the beacons could allow for user-selected files to be played. For example, the user could have a TwinkleTwinkleLittleStar.mp3 and select it to be used for the sound of the beacons.

The upgrade from 2D to 3D should be rather straightforward as Godot has most of its comparable classes named similarly (such as “Area2D” and “Area3D”), so figuring out which structure to change to should not be a problem.
Problems Overcome / Lessons Learned

When calculating collision, the area which overlaps other areas to trigger a collision does not always line up perfectly with the object itself. In video games, this can be seen with rectangular “hit boxes” on sprites that are not rectangular in shape. Despite the collisions not being 100% precise, the game is sped up by using a simple area instead of a complicated set of lines and vertices. The collision areas being considered good enough is a small price to pay for faster game play. Whether in a game or in a tool, the image of an object and its collision rectangle (or cube) are completely separate entities. When making a collision area for an object one has to make sure to attach that area to the object so that it moves with it. Otherwise, one will move and the other won’t, leading to some very peculiar collisions.

Conclusion

Despite a huge effort put into this problem report, its problem statement cannot currently be realized. Comparison of auditory over visual findings needs more pixels on mobile devices, and a human survey experiment would need to be performed to determine if sonification is “better.” However, this report lays the foundation for future work and delivers the tool and the knowledge base to drive the project further when technology improves and another future student continues this investigation.
Further research must be done in order to ascertain if sonification will indeed prove to be more useful than visual senses, especially in, but current acoustic research and theories make this endeavor look promising.

Since the main goal of this project is to provide a mobile tool that can be improved to further this research, the project overall is considered a success.

Appendix

Source code

Particle2D

eextends Area2D

export (AudioStream) var music

func _ready():
    if music:
        $AudioStreamPlayer2D.set("stream", music)

func outer():
    self.modulate = Color(1, 0.5, 0.5, 1)
func inner():
    self.modulate = Color(0.2, 0.6, 0.7, 1)

func out():
    self.modulate = Color(1,1,1,1)

func start_singing():
    if !$AudioStreamPlayer2D.playing:
        $AudioStreamPlayer2D.play()

func stop_singing():
    $AudioStreamPlayer2D.stop()

Sector

extends Area2D

var sector_l = PI * 5/8
var sector_r = PI * 3/8
var inner_r = 75
var outer_r = 275
var zero = Vector2(0, 0)
var center = Vector2(1024/2, 600/2)
var relative_points = PoolVector2Array([zero, zero, zero, zero])
var actual_points = PoolVector2Array([center, center, center, center])

#var hidden = true

#var arr = Array()
#var zero = Vector2(0, 0)

func _ready():
    self.update()
    self.hide()

func update():
    self.calc_rel()
    self.reposition()
    self.set_poly()

func calc_rel():
    var LL = Vector2(inner_r*cos(sector_l), -inner_r*sin(sector_l))
    var UL = Vector2(outer_r*cos(sector_l), -outer_r*sin(sector_l))
var LR = Vector2(inner_r*cos(sector_r), -inner_r*sin(sector_r))
var UR = Vector2(outer_r*cos(sector_r), -outer_r*sin(sector_r))

# print('ll: ' + str(LL))
# print('ul: ' + str(UL))
# print('lr: ' + str(LR))
# print('ur: ' + str(UR))

relative_points = PoolVector2Array([LL, LR, UR, UL])

# print('rel pts: ' + str(relative_points))

func reposition():
    for i in range(4):
        # actual_points.set(i,relative_points[i] + center)
        actual_points.set(i,relative_points[i] + zero)

        # print('adjusted: ' + str(actual_points))

func set_poly():
    # $poly.polygon = PoolVector2Array([LL, LR, UR, UL])
    $poly.polygon = actual_points
#func show_sector():
#    pass

#func hide_sector():
#    pass

**OuterRing2**

extends Area2D

export (float) var rad
#var step = float(5)
#var next_rad = 999
#var p_diff = float(0)
export (Color) var ring_color = Color(1,0,0)
export (bool) var is_outer = true

func _ready():
#    $CollisionShape2D.shape.set("radius", rad)
#    $btn_plus.connect("pressed", self, "change_rad", [5])
    pass

func change_rad(delta):
    rad = rad + delta

# draw colored circle to match actual collision radius
func _draw():
    draw_circle(Vector2(0, 0), rad, ring_color)
func _process(delta):

$CollisionShape2D.shape.set("radius", rad) #nextRad

# shape_owner_clear_shapes(self.get_instance_id())
# var new_circle = CircleShape2D.new()
# new_circle.set("radius", rad)
# var new_coll = $CollisionShape2D.
# new_coll.set_shape(new_circle)
# $CollisionShape2D.set_shape(new_circle)
# self.add_child(new_coll)
# if is_outer:
#     set_collision_layer_bit(1,1)
#     set_collision_mask_bit(1, 0)
#     set_collision_mask_bit(3, 0)

update()
# next_rad = rad + step
# p_diff = (next_rad - rad)/rad
# rad = next_rad
# print(rad, $Sprite.scale)

controls.gd

extends Panel

signal turn_left
signal turn_right
signal move_up
signal move_left
signal move_right
signal move_down

func _on_btn_turn_left_pressed():
    emit_signal("turn_left")

func _on_btn_turn_right_pressed():
    emit_signal("turn_right")

func _on_btn_mv_up_pressed():
    emit_signal("move_up")

func _on_btn_mv_left_pressed():
    emit_signal("move_left")

func _on_btn_mv_right_pressed():
    emit_signal("move_right")

func _on_btn_mv_down_pressed():
    emit_signal("move_down")
extends Camera2D

export (int) var speed = 400

var screensize

func _ready():
    screensize = get_viewport_rect().size

func _process(delta):
    var velocity = Vector2()

    if Input.is_action_pressed("ui_right"):
        # emit_signal("move")
        velocity.x += 1

    if Input.is_action_pressed("ui_left"):
        velocity.x -= 1

    if Input.is_action_pressed("ui_down"):
        velocity.y += 1

    if Input.is_action_pressed("ui_up"):
        velocity.y -= 1
if velocity.length() > 0:
    velocity = velocity.normalized() * speed

position += velocity * delta

### Clamp to field size :)
#
# position.x = clamp(position.x, 0, screensize.x)
#
# position.y = clamp(position.y, 0, screensize.y)

func move(velocity):
    if velocity.length() > 0:
        velocity = velocity.normalized() * speed

    position += velocity / 10

**FileSaver.gd**

extends VBoxContainer

func add_save():
    pass

func save(user, sector):
var txt = '\n'

#camera info

txt += 'pos: (' + str(user.position.x) + ',' + str(user.position.y) + ')

#sector info

txt += '\nsector angle: ' + str(sector.rotation)

var interest = Label.new()

$save_scroll/save_list.add_child(interest)

interest.set_text(txt)

func delete_save():
    pass

# routes instead of Skein?

func save_route():
    pass

func delete_route():
    pass
tool.gd

extends Node2D

export (float) var circle_delta = 5
export (float) var rot_delta = PI/64

var auscultate = false
#var interests = []

# Load these nodes only after readied (available)
onready var btn_mark = $HUD/controls/HBoxContainer/interest_box/mark
onready var btn_listen = $HUD/controls/HBoxContainer/listen_box/listen
onready var btn_in_plus = $HUD/controls/HBoxContainer/inner_box/inner_plus
onready var btn_in_minus = $HUD/controls/HBoxContainer/inner_box/inner_minus
onready var btn_out_plus = $HUD/controls/HBoxContainer/outer_box/outer_plus
onready var btn_out_minus = $HUD/controls/HBoxContainer/outer_box/outer_minus
onready var btn_rot_plus = $HUD/controls/HBoxContainer/rot_box/rot_plus
onready var btn_rot_minus = $HUD/controls/HBoxContainer/rot_box/rot_minus
#onready var interests = $HUD/VBoxContainer/ScrollContainer/VBoxContainer
func _ready():

    btn_in_plus.connect("pressed", self, "inner_plus")
    btn_in_minus.connect("pressed", self, "inner_minus")
    btn_out_plus.connect("pressed", self, "outer_plus")
    btn_out_minus.connect("pressed", self, "outer_minus")

    btn_rot_plus.connect("pressed", self, "rot_plus")
    btn_rot_minus.connect("pressed", self, "rot_minus")

    btn_listen.connect("pressed", self, "toggle_auscultate")
    btn_mark.connect("pressed", self, "mark_interest")

func inner_plus():

    $inner.change_rad(circle_delta)
    $sector.inner_r = $sector.inner_r + circle_delta
    $sector.update()

func inner_minus():

    $inner.change_rad(-circle_delta)
    $sector.inner_r = $sector.inner_r - circle_delta
    $sector.update()

func outer_plus():
$outer.change_rad(circle_delta)

$sector.outer_r = $sector.outer_r + circle_delta

$sector.update()

func outer_minus():

  $outer.change_rad(-circle_delta)

  $sector.outer_r = $sector.outer_r - circle_delta

  $sector.update()

func rot_plus():

  # $sector/poly.rotate(rot_delta)

  $sector.rotate(rot_delta)

func rot_minus():

  $sector.rotate(-rot_delta)

func toggle_auscultate():

  if $sector.visible:
    $sector.hide()

    btn_mark.disabled = true

    btn_rot_plus.disabled = true

    btn_rot_minus.disabled = true

    $HUD/FileSaver.hide()
else:
    $sector.show()  
    btn_mark.disabled = false  
    btn_rot_plus.disabled = false  
    btn_rot_minus.disabled = false
    $HUD/FileSaver.show()

    self.auscultate = !self.auscultate

    ### might want to move to set-checking in here instead of enter/exit signals...

    func _process(delta):
        $particle2.translate(Vector2(0,0))  # needed to update for collision-checking
        $particle3.translate(Vector2(0,0))

        for fox in get_tree().get_nodes_in_group('foxes'):
            if fox in $sector.get_overlapping_areas():
                fox.start_singing()}
elif fox in $outer.get_overlapping_areas() and not fox in $inner.get_overlapping_areas() and not self.auscultate:

    fox.start_singing()

else:

    fox.stop_singing()

#TODO: Auscultate resets song instead of pausing it

if Input.is_action_pressed("ui_cancel"):

    get_tree().quit()

func mark_interest():

    print('interest marked!')

    interests.append($sector/poly.position)  # can't do poly position bc doesn't move, just displayed elsewhere

    interests.append([$sector.rotation, $sector.rotation_degrees])

    $HUD/FileSaver.save($user, $sector)

func _unhandled_input(event):

    if event is InputEventScreenDrag:

        var ev_x = InputEventAction.new()

        var ev_y = InputEventAction.new()
Using Godot

Nodes and Scenes

Two concepts form the foundation of Godot’s philosophy: scenes and nodes. Nodes are the building blocks of the Godot architecture. They follow typical Object-Oriented Programming (OOP) techniques such as encapsulation and inheritance and represent reusable instances of the objects themselves. They can have children (or parent) nodes and be used in multiple scenes. A scene is a tree of nodes (term for single start) that can represent anything from a map or level in a video game to a shader instance. The creators of the engine like to think of these ideas as more than just code. Together, nodes and scenes form the imagery of a cook using ingredients (nodes) to follow a recipe (scene). This recipe can of course be shared with others to enjoy your creation.

Godot Design Paradigms

The Godot developers recommend using best practices for OOP:

```javascript
var vect = Vector2(0,0)
 vect.x = event.relative.x
 vect.y = event.relative.y

$user.move(vect)
```
Encapsulation

Nodes and Scenes were mentioned earlier, and this is the main way of achieving encapsulation. Every node used in a scene should be contained within the scene; scenes should not be very dependent on other scenes in order to achieve their tasks. The basic building blocks of Godot should be scenes themselves and not the nodes. (contradict earlier?) Having high cohesion and smaller, specifically-tasked objects improves maintainability and overall understanding of the project.

Coupling (Loose)

With highly cohesive, encapsulated classes, loose coupling is achieved. The scenes need a way to interact with each other while still controlling their own code. Ultimately, this cuts down on unexpected behavior and bugs. One desirable way to achieve this is through the use of dependency injection. A child node relies on a parent node (or scene) to provide some argument for its method(s). Different parents can use the child’s functionality by providing the scenario or circumstances for the child to perform its method. Similarly, if to sibling nodes need to interact, all communication should be handled via the shared parent.

The Godot website provides five different examples of this:

1. Connect to a signal (use a callback function) (separate section?)

2. Call a method
3. Initialize a function reference (FuncRef)

4. Initialize a Node or other Object reference

5. Initialize a NodePath

**Godot Special Classes**

A CollisionObject2D (extended by Area2D and the 3 Physics Bodies listed below) is used to detect collision between objects. Deciding which one to use can be tricky at first. These are the use cases for each one:

**STATICBODY2D** – Static Bodies do not respond to physics but do provide collision. A good example of the use of a Static Body is a lava pit in a platformer. The developer needs the lava to detect collision with the player (and harm them appropriately), but the lava does not “splash” or get pushed in response to the player’s cannonball into the pool so to speak. (If desired, a splash animation could be played upon collision, but still the lava pool will not move in response).

**RIGIDBODY2D** – A Rigid Body is similar to a Static Body except that it moves. This movement, however, is not controlled by the player but rather the physics engine. Forces are applied to the object and it moves accordingly. A typical use of a Rigid Body would be a projectile in an FPS (First Person Shooter) game. While the player may aim a gun (and thus the theoretical bullet inside the gun), in reality the bullet is not created until the gun is fired, and when this happens,
the bullet is not controlled by the player anymore; its trajectory is solely physics-based (no curving bullets in the air).

KinematicBody2D – Similar to Static Bodies, the Kinematic Bodies detect collisions and are not controlled by physics. The main difference is that they can still move. Since the physics engine has no effect, these bodies need to be programmed to move by the developer. This gives more control over how the objects move. A typical example would be the main character in a game. The developer links system input to the movements of the character—pressing → or D moves the character to the right, for example. These bodies would be

**Godot Editor Origin**

When applying vectors to objects on the screen, it is important to remember which direction is positive/negative on both axes. Remembering the Cartesian Plane may help with this, as positive-x is to the right and positive-y is up. However, the (0,0) coordinate (the “origin”) is at the top left corner of the (viewable) screen. This means that when moving toward the center of a screen, an object that is automatically placed at the origin would have to move in a x-positive/y-negative manner.

One thing often confusing to newer used of Godot, however, is that when creating an object or asset, one has to be careful since the object will be centered on the origin itself and does not have its upper-left corner aligned with the origin (which is how the screen is set up). This means that part of the
object’s coordinates will be negative. If desired, the programmer can make the object’s upper left corner align with the origin, but then when placing the object on the screen, the coordinates will have to be offset based on the object’s size since its upper left corner will be placed at (x, y), making calculations trickier if specifically placing the object.

**Downloading Godot**

The Godot game engine can be downloaded by visiting https://godotengine.org/ and clicking the “download” button at the top of the page.
Then, select the proper operating system and download type and the download should start automatically.

Importing and Running Godot Projects

Godot makes it easy to run projects from other people. First, open Godot. A list of your projects will be displayed. Select “Import” from the right panel.
Next, you Browse to the folder of your downloaded project. Select the “project.godot” file from the project. A green checkmark will appear to confirm a valid project file. Then, click “Import & Edit.”
Now that the project is downloaded, it is time to run it. When you click on “Run,” however, a warning will pop up. This is because the assets have not been imported yet. Simply click the “Edit” button right above instead. Once the project loads, close it and it’s ready to run.
References


A short Skein tutorial. (2019). Retrieved from Inform7:
   http://inform7.com/learn/man/WI_1_8.html

Bruder, G. S. (2013). To touch or not to touch?: comparing 2D touch and 3D mid-air interaction on stereoscopic tabletop surfaces. *1st Symposium on Spatial User Interaction*, (pp. 9-16). Los Angeles, California.


