Chapter 1

SCOUt

This project explores the reliability and flexibility of using a single intelligent controller to complete surveillance-based operations in diverse environments. The Surveillance Coordination and Operation Utility (SCOUt) is used to generalize environments, agents, states and actions into abstract data structures. This data then builds a platform for creating controllers, running simulations, and visualizing outputs.

1.1 Platform

Several coding languages and libraries are used in this project to provide a simple and expandable platform. It is laid out in a client-server architecture to allow separation of data handling and data visualization. The server portion provides full functionality to generate unique environments, build agents and controllers, run test operations, and collect results. Data structures on the server side are implemented using an object-oriented architecture of traits, classes and class instances. Traits are abstract objects that can be inherited by multiple classes. Each class that inherits a trait can add specific values and behaviors to the object. Different instances of a class can be declared for repeated usage within code. This feature is crucial when it comes to long term code handling and maintenance. All classes that inherit from the same trait can be handled using the same logic, yet each class can behave in a unique way. For example, Element is a trait that defines different types of data that an agent could detect in the environment. Elevation is a class that inherits the Element trait, and a specific instance of Elevation can be created for an area that has an elevation level of 100 feet. The trait-class-instance architecture also simplifies the addition of new class definitions. If a future project wanted to utilize the SCOUt platform with an agent that could detect ultra violet rays in their environment, they could define a new "UltraViolet" class that inherits the Element trait. The "UltraViolet" class can then effortlessly be integrated with all other pre-existing code since it will be handled as the general Element triat that it extends. This architecture's

usefulness extends into the client portion of the platform as well. The SCOUt client is a Graphical User Interface (GUI) for requesting actions to be executed by the server and visualizing the data structures returned. Because the majority of the data structures used in this platform are abstracted, the front end can be generalized to handle any new classes created without any maintenance required for the GUI.

1.1.1 Simulation Back End

The back end is written in the Scala language. Scala is a Java based, paradigm language that combines object-oriented and functional programming methods. Object-oriented programming provides the flexibility needed for the trait-class-instance architecture, while functional programming provides immutability when working on large sets of diverse data. All data storage and manipulation takes place on the back end of the platform to ensures consistency. Data is only imported or exported on the back end in two scenarios: file storage and client-server communication. In both cases, it is assumed that immutability is maintained. File storage is the only case where data is open to manipulation outside of the back end. Client-server communication only allows variables to be passed into the back end via requests, and a copy of data structures are returned to the front end for visualization only. Any alterations on the front end will have no effect on the original copy on the back end. To allow storage and communication, the back end encodes and decodes data into Json objects. When imported, Json objects are immediately decoded and parsed into Scala data structures before usage. The circe Scala library is used for the encoding and decoding of Json data. Circe provides integration of Json objects in the Scala language to allow seamless encoding and decoding. Communication for passing and receiving Json objects between the front and back end is achieved with the HTTP4s library. The SCOUt server is setup using HTTP4s' blaze-server to create a local service for handling Http communication.

Reference circe

Reference HTTP4s

1.1.2 Visualization Front End

The platform's front end is built around Electron, a framework that allows building a native desktop application with JavaScript, HTML and CSS. The GUI is written using all three of these languages. HTML structures the page within Electron, CSS provides styling and JavaScript handles all of the logic. The SCOUt platform uses D3, node-fetch and JQuerry JavaScript libraries to assist with data visualization and communication to the back end. D3 (Data-Driven Documents) is a visualization library that uses SVG (an XML-Based format for vector graphics) to create graphical representation of data sets. Node-fetch is used for Http communication with the back end through XMLHttp request and response handling. JQuerry provides integration with Json data that is passed back and forth between the client and server, as well as several functions to simplify working with DOM elements within HTML. Node Package Manager (NPM) is used to maintain all of the dependencies between Electron,

Reference Electron

reference Node packages

Reference NPM

the three languages and the JavaScript libraries on the platform's front end. In addition to dependency management, NPM has packages of its own that simplify the process for compiling code into a single file for Electron to handle. The Babel package transpiles JavaScript into a browser friendly format, then webpack integrates the resulting JavaScript into the HTML code for a single JavaScript file for Electron. The GUI can then be launched using an NPM script to compile all of the code, launch Electron and load the content.

ref Babel

ref webpack

1.2 Environment

Modeling a real-world environment in a simulation is a tricky process. Each model needs to balance simplicity and coverage. If too much is left out of the model, it won't reflect real word scenarios. On the other hand, attempting to model too much can be impractical as it consumes effort and resources that could instead be spent running real world experiments. For this experiment, environments are modeled as a high level class containing a collection of lower level classes that together form a cleverly simplified representation of a real world environment.

The **Environment** class holds an n x m 2D grid of uniformly sized s x s square **Cells**, where n x m is the total area of the environment and s x s is the area each Cell represents within the environment. Along with positional data, each **Cell** contains information about the different elements and anomalies present within the s x s area it covers. An **Element** is a generalized object that represents one specific environmental attribute, such as the elevation or temperature. An **Anomaly** represents some object present within the environment that could be of interest. Anomalies often have an effect on element values in their surrounding area which makes them "traceable".

add a diagram

```
class Environment (
  name: String
  height: Int
  width: Int
  scale: Double
  grid: Array[Array[Cell]]
)
```

Cell

A Cell holds x and y coordinate for its relative position within the Environment's grid. These coordinates do not reflect the actual size of the Cell, only an index value for the order they appear within the 2-dimensional array data structure. The Environment's scale can easily be applied to the physical location of a Cell as it is a shared global attribute within the Environment object. If an element type or an anomaly is present within the area that the cell covers in the environment grid, it will appear in its respective list.

```
class Cell (
  x: Int
  y: Int
  elements: Array[Element]
  anomalies: Array[Anomaly]
)
```

Element

An element can be any measurable attributes within an environment. For example: temperature, elevation and decibel levels are all attributes of the environment whose values can be measured. All element types are all generalized by the abstract trait, Element. The trait has a set of defined attributes that an inheriting class must define to identify the element type and how it behaves. Name and unit are used for identification and displaying the element type. The value attribute holds a numerical value for the measurement of each instance. For example, an instance of the Elevation class would store a measurement of the elevation level in a certain area. The radial flag, lowerBound and upperBound attributes guide and limit the values that can be set for the element type. These are used when procedurally generating an environment (covered in **??**).

reference environment gen section

```
trait Element {
  name: String
  value: Double
  unit: String
  radial: Boolean
  lowerBound: Double
  upperBound: Double
}
```

Anomaly

Anomalies are any object that may be of significance to an agent, such as a human or precious mineral. Anomalies have their own effects on element values in the environment around them. Like Elements, Anomalies and Effects are defined as classes that inherit from a single trait. An Anomaly class can occupy multiple Cells, but must occupy at least one cell in an environment. Anomalies can also have multiple Effects on multiple types of element values in surrounding Cells These Effects are declared in a list attribute.

```
trait Anomaly {
  name: String
  area: Double
  effects: List[Effect]
}
```

Each Effect class defines a "seed" Element class and a range of the effect. The seed attribute holds a specific instance of an Element class that will represent the value of that element type in the area that the attribute exists within the environment. The range then defines the radius of the area beyond the attribute's position that the effect will "radiate". The term radiate is used because the effect will alter the element type's values in surrounding based upon how close they are to the source of the effect (where the anomaly exists).

```
trait Effect (
  seed: Element
  range: Double
)
```

For example, Human is an anomaly that takes the area of a single cell, and Effects the Temperature and Decibel values in their Environment. If the Human is much louder than the ambient noise level in the environment, there will be a sharp spike in Decibel values in Cells nearest the Human, with a diminishing increase for values in surrounding Cells.

Show example code or photo?

Layer

One last important data structure is a Layer. While Layers are not direct members of the Environment class structure, they are crucial to building and analyzing the Environment. For this reason, instances of the Layer class are only generated on demand through method calls. A Layer is designed in the same 2-dimentional structure as the Environment grid, but holds a collection of Elements instead of Cells.

Photo of Layer

```
class Layer (
  length: Int
  width: Int
  layer: Array[Array[Element]]
)
```

1.3 Agents

Agents within this experiment have a core set of attributes and abilities, along with a set of sensors and a controller. The core attributes for an agent are health, energy level, an internal map and its current position relative to the environment grid. Because SCOUt is focused on purely observational interactions with its environment, an agent only has two categories of actions that can be performed: movement and scanning. The agent can attempt to move one cell at a time in any of the four cardinal directions. This allows the agent to reassess after each movement attempt. Scanning collects information about the agent's immediate environment and updates internal map. The list of scan actions that an agent can perform is based on the set of sensors the agent is equipped with. The agent's controller is in charge of analyzing the current state of the agent and

deciding the next action to be performed. This project focuses on creating a single controller that is highly adaptable to wide ranges of agent configurations, environments and goals titled as the SCOUt controller. This controller should be able to show adaptability when performing new tasks and when controlling different agents. To model how an Agent will interact with an Environment, Mobility and a collection of Durability factors are defined per Agent.

```
class Agent (
  name: String
  controller: Controller
  sensors: List[Sensor]
  internalMap: Array[Array[Cell]]
  xPosition: Int
  yPosition: Int
  health: Double
  energyLevel: Double
  mobility: Mobility
  durabilities: List[Durability]
)
```

1.3.1 Sensor

Sensors are created using the same trait-class-instance architecture as Elements and Anomalies. The Sensor class models a scientific instrument that could be used for gathering data measurements of a specific element type. Each class defines the element type it is able to measures, the energy it costs to perform a "scan" action, its effective range, and two flags indicating if the element type being searched for if hazardous or considered an indicator. When performing a scan, the sensor will sweep 360 degrees around the agents location and gather data within the circular area. The circular scan area is calculated with the sensors range as the radius and the agents position as the center. Any Element values that were previously unknown to the agent are then added to the internal map. Hazardous elements are flagged in a sensor Class when its element type has the potential to cause harm to the given Agent. The indicator flag is set when the element type is believed to be of importance for completing the goal. For example, if an Agent was searching for a Human, Temperature and Decibel Sensors would be flagged as indicators because their values can potentially help lead the agent to the Human.

```
class Sensor (
  elementType: String
  range: Double
  energyExpense: Double
  hazard: Boolean
  indicator: Boolean
)
```

1.3.2 Mobility

Mobility is a stand alone class that will determine the ease and limitations of the agent moving within an environment. For example, if a drone was modeled as the agent, it would have a higher range of mobility, but would likely sacrifice the amount of sensors that could be carried. A wheeled robot loaded with multiple sensors would likely have decreased mobility, but could collect a wider variety of data. Mobility is defined by maximum slope an agent can climb, the minimum slope an agent can traverse before taking "fall damage", a resistance factor and the energy cost required for movement. The resistance factor is used to scales the amount of damage that the agent may take (for example if it fell off of a cliff). Movement cost is used to calculate how much energy is used when attempting a movement action. The total cost calculated is scaled based on the distance moved, and the slope of the elevation between the agent's current position and the position that it attempts to move to.

```
class Mobility (
  movementSlopeUpperThreshHold: Double
  movementSlopeLowerThreshHold: Double
  movementDamageResistance: Double
  movementCost: Double
)
```

1.3.3 Durability

Like many other data structures in this platform, Durability factors are defined per element type. These factors model how an Agent will be effected by different Elements they come in contact with during an Operation. For example, consider an environment with pools of water in it. Most robots would be damaged when emerged in water, but an amphibious robot could be modeled to be impervious to damage when in contact with water. Durability is defined by an upper and lower value threshold and a resistance factor. The thresholds define what levels of an element type that the agent can be exposed to before it begins to take damage. The resistance factor then influences how much damage the agent will take at levels exceeding the threshold.

```
trait Duribility (
  damageUpperThreshold: Double
  damageLowerThreshold: Double
  damageResistance: Double
)
```

1.3.4 Actions

Agents interact with the Environment via actions. Actions are simplified to either movement or scanning actions. These two categories cover the exploration and

research aspects that required for most surveillance operations. The Agent's Controller is in charge of deciding what action to perform.

In simulation, movement is handled by changing the agent's current position to an adjacent cell in one of four direction. Movement to an adjacent cell is denoted as "north", "south", "west" or "east" based on the orientation of the x, y grid of cells that make up the environment. Moving a single cell at a time gives the agent the opportunity to reassess its current state before selecting the next action. Distance covered by successful movement will inherently be equal to the size of the cells within the simulated environment. Each time an agent attempts to move to a new cell, Elevation levels will be compared between the current and new cell to check if movement is possible or if it results in damage (based upon the Agent's Mobility). After the attempt has been made, changes to health and energy level are calculated based upon the Agent's Durability factors and then updated. If the movement action is successfully completed, the current position is also updated.

add photo of movement

An Agent can perform scans of the Environment using available Sensors. For each cell that fall within the Sensor's search radius, the value for the sensor's given element type is extracted. These values are then added to the agent's internal map if they did not previously exist there. Through repeated scanning, the agent will begin to map out its surrounding environment. Data collected in the internal map can be then be used by the controller to determine what actions would be most beneficial to the goal at hand.

1.3.5 State Representation

For controllers to intelligently decide when to perform what actions, they need to have sufficient data about the agent and the known surrounding environment. The agent's position, health and energy level can easily be analyzed, but the internal map containing the known environment is a very large data structure to analyze each time the controller has to decide upon an action. For this reason the data structure is simplified in order to reduce memory usage and computational effort required to analyze a state. AgentState is the minimal data structure that contains all of the useful information necessary for a controller to make intelligent decisions. Instead of a 2-dimensional array of cells, the internal map is represented as a list of ElementStates, where each ElementState is a summary of the data known about a specific element type.

```
class AgentState (
  xPosition: Int
  yPosition: Int
  health: Double
  energyLevel: Double
  elementStates: List[ElementState]
)

ElementState (
```

```
  elementType: String
  indicator: Boolean
  hazard: Boolean
  percentKnownInSensorRange: Double
  northQuadrant: QuadrantState
  southQuadrant: QuadrantState
  westQuadrant: QuadrantState
  eastQuadrant: QuadrantState
)
```

Element states contain useful information about what information is known for a specific element type during the operation. The indicator flag can cue the controller on whether the element type is being analyzed in order to progress the goal at hand. If the goal was to map out the elevation levels in an Environment, the elevation ElementState would be flagged true. If the goal was to find a human, the Temperature and Decibel ElementStates would be marked true, as irregular changes in these values could help indicate the presence of the Human. The hazard flag is used to mark any Element that could potentially cause harm to the agent. For example: the presence of water, large changes in elevation and extreme temperatures could potentially cause damage, and would be flagged as hazardous. We also track the percent of known element values that are within the range of the corresponding sensor. Technical information of each ElementState is divided into four quadrants, where each quadrant has its own state. Because agent movement is limited to north, south, west and east, we can collapse known information from the internal map into four quadrants.

Image of quadrants

```
class QuadrantState (
  percentKnown: Double
  averageValueDifferential: Option[Double]
  immediateValueDifferential: Option[Double]
)
```

The first thing that a QuadrantState looks at is the percent of values that are already known in all the cells within the quadrant. Then, the QuadrantState stores the average and immediate known values into two "Options". These values are defined as Options because there are instances where the values within the quadrant are not known or may not exist (if the Agent is at the edge of the defined Environment grid). Option is a built in Scala data type that can be None when undefined or Some(<value>) when defined. This data type preserves data immutability as all variables must be defined within Scala code. Average and immediate values are recorded as differentials relative to the value of the current cell. Average differential takes the difference between the current Cell's value and the average of all known values in the quadrant's collection of Cells. Immediate differential takes the difference between the current Cell's value and the Cell immediately adjacent to it. So when considering elevation values within the north quadrant, the immediate differential would be the difference between the elevation at the agents current position and the elevation within the adjacent cell to the North.

1.3.6 Controller

A Controller is the goal driven decision making schema that an Agent will use when navigating an Environment. The Agent will pass its current state to the controller, along with a list of valid actions that it can take. The Controller will then decide the best action that can be taken given the current state. Controllers are defined using the trait-class-instance architecture. Inheriting Controller classes are provided a setup and shutdown function to perform any initializations or final actions once an Operation has ended. The Controller's schema is defined within the selectAction function. This function takes in the list of valid action and the current AgentState and must return a single action which the Agent will attempt to perform. Specific Controllers and their schemas are analyzed and discussed in the **??** section.

```
trait Controller {
  def setup(mapHeight: Int, mapWidth: Int): Unit
  def selectAction(actions: List[String], state: AgentState): String
  def shutDown(stateActionPairs: List[StateActionPair]): Unit
}
```

1.4 Operations

To explore the usefulness and robustness of the intelligent controller, many different scenarios need to be simulated. All simulations are made up of three main components: an agent, a goal and the environment. Different combinations of each component allows the creation of a large variety of scenarios. Each simulation follows a defined process called an Operation. Each Operation will simulate an Agent's attempt to complete a Goal within an Environment. The Operation will record data for each event that occurs between the Agent and Environment and the final outcome. These data collections are denoted as short term and long term events respectively.

Operation Diagram

Short term events are collected each time the agent performs an action.

- The agent's state when the action was selected
- Any changes to the agent's internal state (health or energy reduction)
- If the action performed was successful (could it move, did it have enough energy to complete the action)
- A short term reward for the outcome of the action

A long term event is only collected once at the simulated operation ends.

- Status of internal variables (health and energy)
- Number of actions taken during operation

- Level of goal completion
- An overall long term reward
- Long term reward given to each action taken

1.4.1 Goals

Because SCOUt is designed for observation and exploration, two Goal types are analyzed: anomaly searching and element mapping. This is not say that SCOUt would be limited to Operations which only involve these types of tasks. SCOUt is intended to be coupled with other tasks. For example, if the entire task of a robot was to traverse a hazardous area to find a certain mineral for extraction. SCOUt would be used to guide exploration in the environment and detect the mineral. When the mineral is found, SCOUt's process would then be completed successfully and a separate process could take over for the actual extraction, or the location could be recorded by SCOUt and another agent could be sent in. After the other agent or process completes its task, SCOUt could continue to search for more deposits of the mineral or return to base. For anomaly searching goals, the agent is required to find a specified anomaly within an environment. This tests SCOUt's ability to use environmental clues to track down the anomaly. For example, if the agent was looking for a human after a natural disaster, it could use data such as temperature and decibel readings to locate the person. Element mapping is fairly strait forward. The agent is must map out as much data about the specified element type as possible.

1.4.2 Rewards

In addition to goal completion, an agent also has to be observant of its health and energy on a short term scale. The more efficiently an agent can complete an Operation the better. The overall performance of an agent is measured by both its ability to complete the task at hand and the safety and efficiency of the actions taken. These performance measurements are calculated on a short and long term basis, and come in the form of "rewards". In addition to measuring the performance of an agent, rewards can be used as a learning metric for an intelligent Controller ().

ref controller schema

Short Term Rewards

Short term rewards are given each time an agent performs an action to reflect the immediate outcome of the action. Energy and health depletion are major factors in this reward. If the action required an excessive amount of energy or resulted in damage to the agent, the reward is decreased. Other factors that come into play depend on the specific action taken. If the agent attempted to move to a new area and failed to move (ex: a hill was too steep to climb) a deduction is made. A small increase in reward is applied if the agent moves into an unexplored area. If the agent uses a scanner, the reward is adjusted to reflect

the amount of new data learned. This penalizes the agent from using a scanner twice in a row or after small movements as it is not efficient use of energy.

Add STR Equation

Long Term Rewards

Long term rewards are calculated once the operation is over. Operation end cases are when the agent has successfully completed its goal, or it is depleted of health or energy. To reflect these scenarios, the reward is determined by the goal completion, remaining health and remaining energy. Even if a goal is completed, the agent could receive a low score if it was "reckless" and took lots of damage or used large amounts of energy. The long term reward is then propagated backwards through all the actions that were taken. The actions performed immediately before the end of the operation are given highest score. Previous actions then receive diminishing reward based on .

LTS reward distribution equation

1.5 Environment Builder

Add LTR Equation

The SCOUt EnvironmentBuilder is a tool for creating diverse Environment models, while remaining simple to implement and understand. The tool is highly abstracted so that more details can easily be added to the model as needed while still maintaining a defined build process. Environments are procedurally generated based upon a collection of parameters called an environment template. These templates require minimal input to build a dynamic range of possible Environments. An Environment can be tweaked or even built entirely by hand, but the procedural generation process removes this overhead.

Procedural Generation Process

1. Environment Template is passed in
2. Builder initializes a grid of empty Cells
3. ElementSeeds are used to populate each present element type into the gird of cells
4. TerrainModifications are applied to manipulate their related element(s)
5. Anomalies are placed randomly within the environment
6. Anomaly effect(s) are applied to corresponding element(s) in neighboring Cells

1.5.1 Environment Templates

Each template created will act as a guide in the creation of an instance of the Environment class. A template can create similar, but unique environments each time it is used by the EnvironmentBuilder. This allows testing and training agent controllers multiple times in similar conditions, while still providing a dynamic range of scenarios that the Agent may face in each generated Environment. Each

template is comprised of the name, dimensions and scale of the environment along with lists of ElementSeeds, TerrainModifications and Anomalies to be applied.

```
class EnvironmentTemplate (
  name: String
  height: Int
  width: Int
  scale: Double
  elementSeeds: List[ElementSeed]
  terrainModification: List[TerrainModification]
  anomalies: List[Anomaly]
)
```

1.5.2 Element Seeds

The environment builder begins by procedurally generating one Layer of Elements at a time. Each Element class has a companion class called an ElementSeed which holds parameters used to produce a Layer of its element type, and a unique function defining how procedural generation will take place to produce the Layer. The generation of each Layer is modeled on how the element type's values may vary in a real-world scenario. Parameters within each Seed are set to default values that can also be overriden by creating a new instance of the ElementSeed. This can change how the values within the Layer will vary.

```
trait ElementSeed (
  elementType: String
  function buildLayer(height, width, scale)
)
```

The environment builder will use each ElementSeed to produce each Layer of Elements. Resulting Layers will be temporarily stored in a list until the end of the build process so they can easily be manipulated before being stored into corresponding Cells within the Environment grid. Some layers are far easier to generate than others. Latitude and Longitude layers can simply be generated by calculating the distance each cell is from the origin point on the Environment grid (cell (0,0)).

For an example, let's look at the ElementSeed for producing the Elevation Layer.

```
Class ElevationSeed (
  elementType: String = "Elevation"
  average: Double = 0.0
  deviation: Double = 1.0
) {
  function buildLayer(height, width, scale) {
    val layer: Layer = new Layer(AB.fill(height)(AB.fill(width)(None)))
```

```
    for {
      x <- 0 until height
      y <- 0 until width
    } {
      val value = randomDeviation(average)
      layer.setElement(x, y, new Elevation(value))
    }
    layer.smoothLayer(3, 3)
    return layer
  }

  function randomDeviation(average) {
    val lowerBound = average - deviation
    val upperBound = average + deviation
    return randomDouble(lowerBound, upperBound)
  }
}
```

The ElevationSeed's buildLayer algorithm first initializes an empty Layer. Next, it sets each (x,y) coordinate in the Layer to an a random Elevation value within a standard deviation of the average value provided [(average - deviation), (average + deviation)]. Once every Cartesian position has been set to an instance of Elevation, the layer is then smoothed. Smoothing is a function defined within the Layer class that will reduce strong variations of Element values within the Layer. For Elevation, this would equate to transforming a highly rigid surface into a smoother, more natural surface.

smoothing pseudo-code

smoothing before-after photo?

1.5.3 Terrain Modifications

Terrain modifications influence the basic landscape of the environment. After each ElementSeed has produced a layer for the Environment, TerrainModifications are applied one after the other, taking care not to overlap modifications (for example, we wouldn't want a hill to overlap with a valley and cancel each other out). Each modification represents a severe alteration of one or more element type Layers within the environment. Following a similar process laid out by Doran and Parberry**??**, desired alterations are incorporated into the environment while allowing unique variations of each to develop. Their controlled procedural generation process is used to produce landmasses that potentially have bodies and channels of water. SCOUt's environment builder generalizes this process and extends it to allow multitudes of element types to be modified. The TerrainModification trait provides an extendable template from which all types of modifications that can be applied.

```
trait TerrainModification (
  name: String
  elementTypes: List[String]
```

```
) {
  function modify(layers: List[Layer])
}
```

For an example, lets look at Elevation again.

```
class ElevationModification (
  name: String = "Elevation Modification"
  elementType: List[String] = List("Elevation")
  modification: Double
  deviation: Double
  coverage: Double
  slope: Double
) {
  def modify(layer: Layer, constructionLayer: ConstructionLayer) = constructionL
    case None => // No unmodified cells
    case Some(startCell) => {
      // Set local variables
      var modifiedCells: AB[(Int,Int)] = AB()
      val numCellsToMod = Math.round(coverage * constructionLayer.cellCount).toI
      // Initial modification
      val startX = startCell._1
      val startY = startCell._2
      layer.setElementValue(startX, startY, modification)
      constructionLayer.setToModified(startX, startY, "elevation")
      modifiedCells.append((startX, startY))
      // Move to random, unmodified neighbors and modify
      for (i <- 0 until numCellsToMod) constructionLayer.getNextUnmodifiedNeighb
        case None => // No neighbor cells to modify
        case Some((x,y)) => {
          val currentValue = layer.getElementValue(x, y).getOrElse(0.0)
          val mod = randomDouble((modification - deviation), (modification + dev
          val newValue = currentValue + mod
          layer.setElementValue(x, y, newValue)
          constructionLayer.setToModified(x, y, "elevation")
          modifiedCells.append((x, y))
        }
      }
      // Apply sloping factor to modified area through smoothing
      val effectedRadius = Math.abs(Math.round(modification / slope).toInt)
      for (i <- 0 until modifiedCells.length) {
        val randomIndex = randomInt(0, modifiedCells.length - 1)
        val c = modifiedCells.remove(randomIndex)
        val originX = c._1
        val originY = c._2
        for {
          x <- (originX - effectedRadius) to (originX + effectedRadius)
```

```
            y <- (originY - effectedRadius) to (originY + effectedRadius)
            if dist(x, y, originX, originY) != 0
            if dist(x, y, originX, originY) <= effectedRadius
          } layer.smooth(x, y, 2, dist(originX, originY, x, y))
        }
      }
    }
  }
```

turn into pseudo-code

Here we have an ElevationModification which will allow us to create hills and valleys within the environment. Again following the approach of **??**, random, unmodified (x,y) positions are selected from the Layer to begin with and updates their value to the specified modification value. The modifier then performs "walks" to random, unmodified neighboring Elements, updating their values within a standard deviation of the specified modification value. These walks continue until the specified coverage area has been modified, or until there are no neighboring cells that can be modified. A special Layer smoothing algorithm is then applied to the Elevation values in the modified area, as well as the immediate surrounding unmodified area to reduce rigidity and give a more natural change in values between neighboring cells. This type of smoothing applies a given slopping factor within the modified area, allowing the ElevationModification to generate gentle hills or valleys or sharp cliffs depending on the slope defined.

1.5.4 Anomaly Placement

Once all TerrainModifications have been applied, anomalies are placed into the environment. Each specified anomaly is randomly placed into cell(s) in the Environment. For anomalies that occupy more than one cell, neighboring cells are chosen at random until the Anomaly's coverage area is met, or there are no neighboring cells that can contain the Anomaly. The anomaly type is appended to each occupied Cell's anomalies list for reference within the simulation. After an anomaly has been placed, each of the Anomaly's Effects are applied. An Effect will alter the Element values for the occupied and surrounding Cells in the effected area. These alterations are typically applied as a "radiation". For example, a Human Anomaly might radiate heat and sound. To account for this, the radiation function is applied to the temperature and decibel values of cells in the effected radius.

radiation function

Now that all ElementSeeds, TerrainModifications and Anomaly placements have occured, the resulting Layers containing thier respective Elementss are populated into their corresponding (x,y) Cell location within the Environment grid. The resulting instance of an Environment class is then returned by the builder to the requesting party.

1.6 Visualization Tool

The environment build tool provides a Graphical User Interface (GUI) for creating and visualizing environments. Electron is used to simulate a web page contained within a standalone desktop application. This allows the front end to be written in JavaScript, HTML and CSS and handle communication to the back end via Http over a localhost network. Scala library HTTP4s is used to create a server on a localhost network for handling the http requests from the front end. Launching the GUI starts up the Scala server in a new terminal and opens the Electron window which will begin attempts to establish communication with the server.

Possibly cite Electron

Possibly cite http4s

1.6.1 Home Page

Once connection between the server and GUI has been established, the user is brought to the Home Page wher they can choose to generate a random environment, build a custom environment, load in an environment or view an operation. For a random environment, the user only inputs the name and n x m size of the environment and all other variables are selected by the server. Building a custom environment steps the user through a series of form pages to create an EnvironmentTemplate. Loading an environment allows the user to select a saved environment or a saved template to use. Selecting an operation will load the environment and log of all actions taken by an agent during a specific operation run that is saved in memory. Once an environment has been generated and/or loaded by the server, it is returned to the GUI to be displayed. The environment build tool will parse the returned environment into a graphical data representation, with interactive capabilities to explore the specific variables within the environment. In the case that the user selected an operation, the user will additionally be able to step through the event log of an Operation.

add photos

1.6.2 Template Forms Page

To create a template, the user will be presented with a series of forms with parameter input fields. The forms are generated based on the available Element, TerrainModification and Anomaly classes that are defined in SCOUt's back end. The first three forms will ask the user which element types, terrain modification and anomaly types they would like to include. Some elements (environment, latitude and longitude) are required in all environments. As the user selects what will be present in the environment, more forms will be generated for them to provide seed data (for Elements and Anomalies), or parameter definitions (for TerrainModifications). Each form within this process will save the user's input data on the front end. This allows the user to go back and edit form data while moving through the different form pages, as well as return and edit the form data after an Environment has already been generated and loaded into the **??**. Form data is also set within required bounds and checked before submission. Once the user has filled out all required form info, they can review their entire

add photos

form entry from a single page and then submit it. When submitted, the front end data is converted into Json data and sent in a request to the SCOUt server via a JavaScript fetch request. An Environment instance is then built on the back end using the template parameters provided, and returned to the front end where it is loaded into the **??**.

Template Form Sections

1. Environment Name and Size
2. Element Types Present
3. Terrain Modifications Present
4. Anomalies Present
5. Element Seeds
6. Terrain Modification Templates
7. Anomaly Seeds

1.6.3 Visualization Page

add photos

The visualization page provides an interactive overview of any given environment. The main focus is on the display section where the entire environment grid is represented using heatmaps. Different Element Layers can be viewed independently, anomaly locations can be highlighted, and specific element type values of a single cell can be viewed. A main menu is also present to allow a user to perform higher level actions. All of these interactive features are controlled by action, toggle and radio buttons within different sections of the visualization page. The primary use of the visualization page is for creating environment templates and for debugging. Debugging usage ranges from analyzing an environment that was used for testing an agent-controller setup or new features. Examples of new features would be adding new classes (ex: a new Element type), or altering the process in which environments are generated and stored on the back end.

Main Menu

add photos

The main menu provides high level functions to perform while using the environment build tool. The main menu options are displayed at the top of the visualizer as a series of buttons. The user can select buttons to return to the home page, regenerate or save the current EnvironmentTemplate (if an environment template is being used), or save the current Environment instance that is currently being viewed. If the user wants to tweak the current template that is in use, they can do so by returning to the Home Page**??** via the home button, and choosing "Custom Environment" again. Their previously set parameters will be loaded back into the form fields for editing.

Display

add photos

The display is laid out in a grid of display cells corresponding to the Environment's Cell grid. A display layer is created for each element type present within the environment using the D3 library. D3's heatmap creates a graphical representation of data values over a 2-dimentional space, providing a solution for visually differentiating each cell's value within the environment cell grid. Heatmaps display the variation of values using a color scale where higher values are indicated by darker sections and lower values as a lighter section of the map. For example, when viewing the elevation's heatmap, a hill will appear darker than a valley. The user also has the ability to select individual cells by clicking on their region in the displayed cell grid. The display will highlight a display cell once it has been selected and then load the cells data into the Legend**??**.

Tool Bar

add photos

The tool bar is divided into three subsections: Toggle Layers, Current Layer and Current Anomaly. The Toggle Layers subsection provides two toggle buttons for the user to turn on and off the display of the Elevation layer and the Grid layer. The Elevation layer is a grey-scale contour map of the Elevation layer created by D3 heatmap (a contour map is the same as a heatmap, with the distinction of boarder lines between each value layer). Because Elevation is the most fundamental piece to any environment, it is the only element type whose layer has the option of always being displayed. The grid layer displays solid black lines between each cell within the cell grid. The Current Layer subsection provides a list of radio buttons for all other element types present within the environment. When one of these radio buttons is selected, a green-scale, transparent heatmap of the selected element type will be populated into the display. This element type layer will be displayed on top of the Elevation layer (if Elevation is toggled on). Only one element type layer can be viewed at a time to prevent crowding the display. The Current Anomaly subsection is also a set of radio buttons for each anomaly type present in the environment. Selecting one of these will highlight all display cells containing the given anomaly type in red. Just as the case with element type layers, only one anomaly type can be viewed at a time.

Legend

add photos

The legend provides an overview of the environment in three main subsection: environment, layer and cell. For the environment subsection, the name of the current environment is displayed along with the dimensions and minimum and maximum elevation within the environment. The layer subsection displays the minimum and maximum values of the selected display layer (if one is selected), as well as the display layer's value at the selected cell (if a cell is selected). When a cell is selected, the values of all element types in the area covered by the cell are presented in a list, as well as the cell's relative coordinates in the grid.

Operation Log

add photos

The operation log section is a special section that only appears in the visualizer when the user loads an Operation run. This section has buttons that allow the user to step through each event that took place during the given agent's operation. The user can select to step forward or backwards by 1 or 10 events. When each event is loaded into the visualizer, the display section will update by selecting the cell where the agent is currently located. There is also a text display section that shows the index of the event that is currently being viewed, the action that was chosen, the health and energy of the agent during this event, and the long and short term rewards that were received.

Bibliography