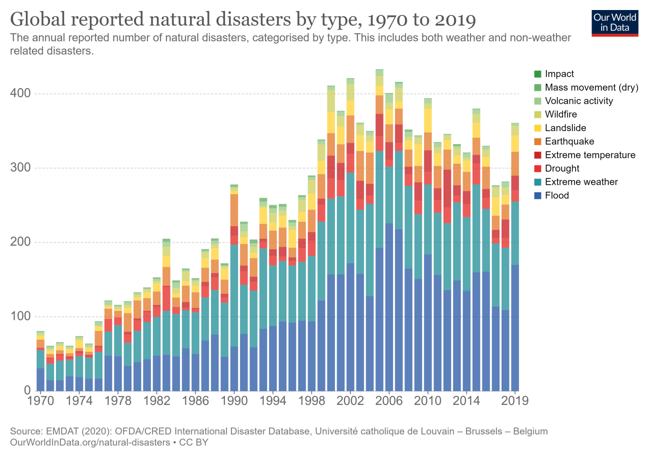
Prototyping Documentation

Hanan Korabi, Dania Al Hakim, Cristian Cernat, Kaloyan Staykov, Shehroz Bashir Malik

I. Introduction

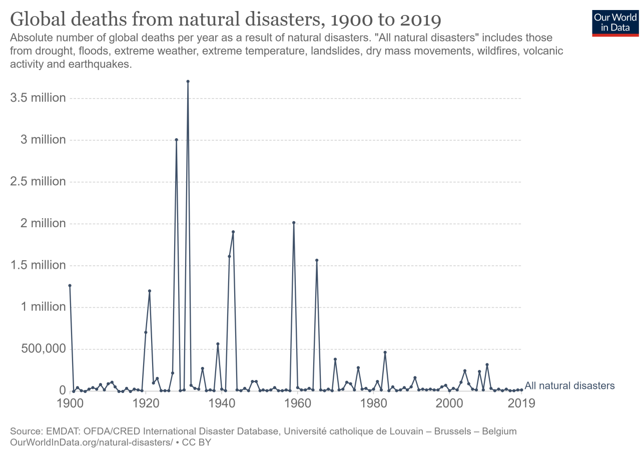
A. *Project Purpose*

Since 1970, the number of reported natural disasters has increased consistently up until 2010's and the typical amount of such events yearly is above 300 incidents.



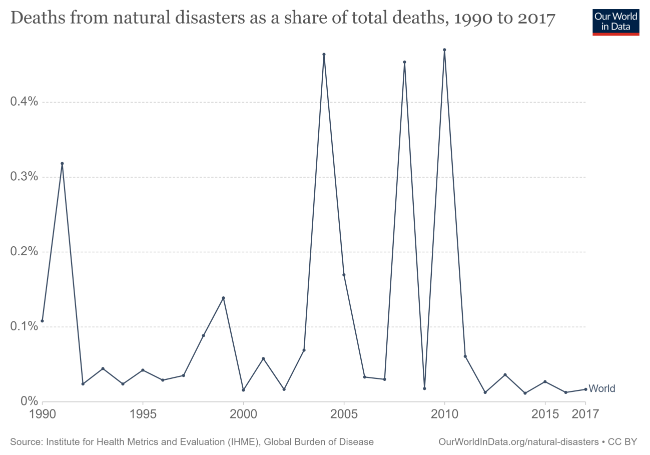
[1]

The most frequent of the events are: floods, extreme weather, drought, extreme temperature, earthquakes, landslides, wildfire, etc. Fortunately, people and organizations can prepare for such calamities to minimize the social and economical damage. For example, modern architecture in the Netherlands is very durable when it comes to floods, the tall urban buildings in Japan are built to withstand the seismic activity in the region.



[2]

Safety and health institutions are always looking for new solutions when it comes to preventing and first response if a disaster occurs. It is primordial to keep human life safe, but accidents do happen and every decision and extra time taken might be very costly.

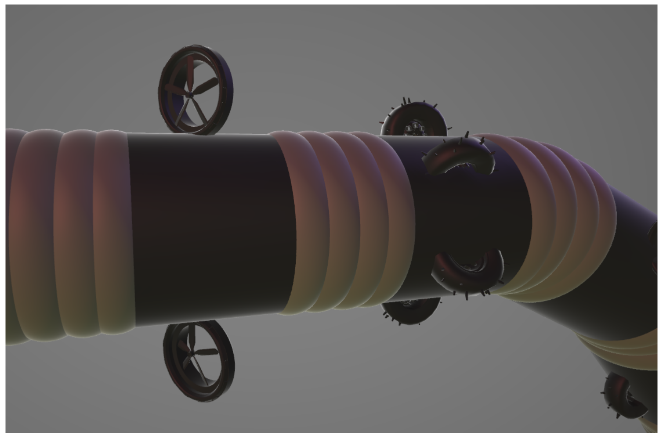


[3]

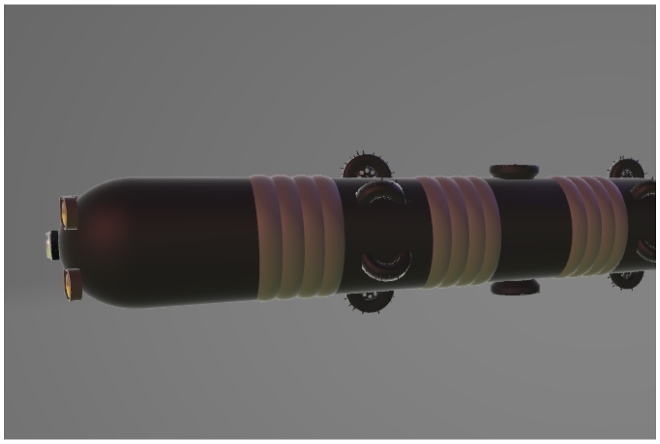
For the Prototyping semester project work, our team had the task of constructing the first steps of a ''rescue robot". With total design liberty our team thought and sketched more ideas that could work in the physical world. Only a very limited amount of requirements were forwarded to us to make sure our group had plenty of space to explore solutions that are both original and creative.

B. *Early Concepts*

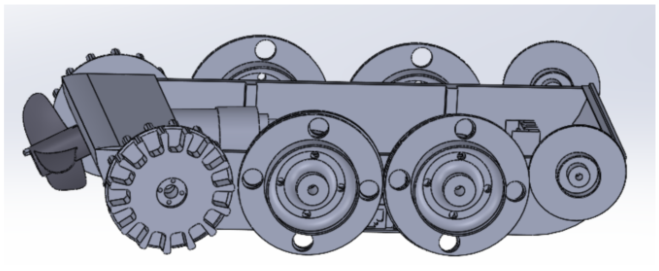
One of the first solid concepts we thought about and stopped to work on was called ''serpent", due to it's similar look.



Our team thought the snake-like design had a lot to offer, because of it's extremely narrow body that is able to function successfully in environments like: caves, rubbel, remains of a damaged building/structure and water. We specifically planned on giving the ''serpent" great abilities for navigating in the water, due to it's great potential for movement, flexibility and versatility.

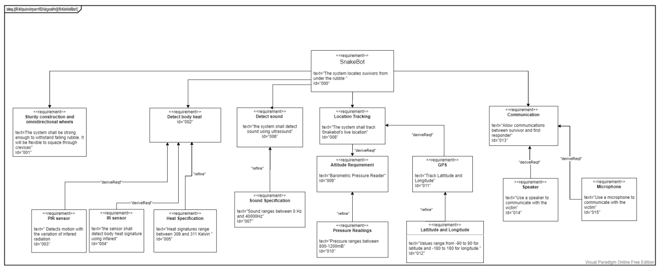


Although, on paper the ''serpent" was promising, it brought a lot of challenges along that required in-depth knowledge of software and hardware as well as a lot of resources. For example, just the process of making the model waterproof consisted of a great challenge, especially with all the propellers, wheels, cameras, lights and flexible joints that needed to be made watertight. Then we also had to solve the issue of flexibility and movement. The model proved to be quite long and thin, which made it harder to make it solid and reliable.

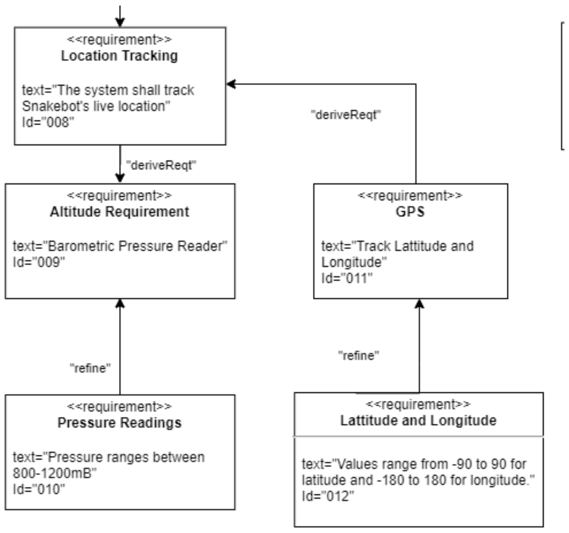


II. Requirements

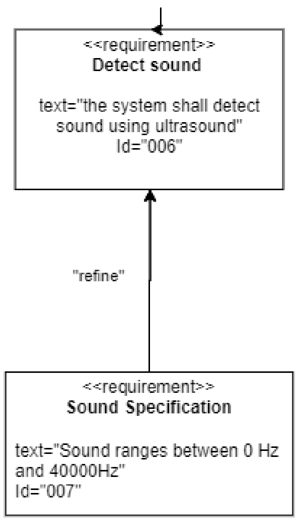
The Serpent Robot will be used to assist in the rescue after environmental disasters.


req [RequirementDiagram][SnakeBot]
<<requirement>> SnakeBot text="The system locates suvivors from under the rubble " Id="000"
<<requirement>> Sturdy construction and omnidirectional wheels text=The system shall be strong enough to withstand falling rubble. It will be flexible to squeze through crevices" Id="001"
<<requirement>> Detect body heat Id="002"
<<requirement>> Detect sound text="the system shall detect sound using ultrasound" Id="006"
<<requirement>> Location Tracking text="The system shall track Snakebot's live location" Id="008"
<<requirement>> Communication text="Allow communications between survivor and first responder" Id="013"
"deriveReqt"
"refine"
<<requirement>> Altitude Requirement text="Barometric Pressure Reader" Id="009"
<<requirement>> GPS text="Track Lattitude and Longitude" Id="011"
<<requirement>> Speaker text="Use a speaker to communicate with the victim" Id="014"
<<requirement>> Microphone text="Use a microphone to communicate with the victim" Id="015"
<<requirement>> PIR sensor text=" Detects motion with the variation of infared radiation Id="003"
<<requirement>> IR sensor text="the sensor shall detect body heat signature using infared" Id="004"
<<requirement>> Heat Specification text="Heat signatures range between 309 and 311 Kelvin " Id="005"
<<requirement>> Sound Specification text="Sound ranges between 0 Hz and 40000Hz" Id="007"
<<requirement>> Pressure Readings text="Pressure ranges between 800-1200mB" Id="010"
<<requirement>> Lattitude and Longitude text="Values range from -90 to 90 for latitude and -180 to 180 for longitude." Id="012"
Visual Paradigm Online Free Edition

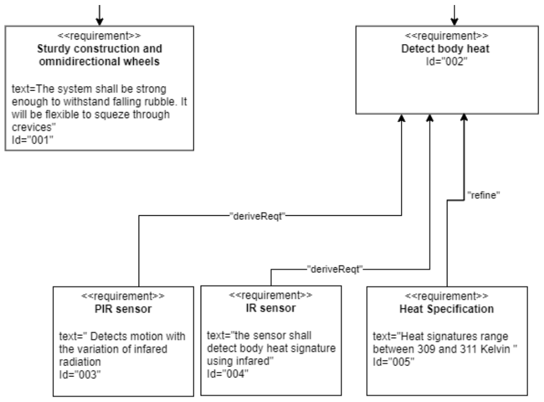
The disasters we have in mind are Earthquakes, Tsunamis, Cyclones and Hurricanes. It will traverse through rubble and find survivors while mapping out terrain. The system will be connected to a terminal that controls it. The bot will incorporate Location Tracking and transmit its live location back to the terminal. It will have to send its Latitude, Longitude and Vertical Height. It will have to capture its Altitude using Pressure Readings. These readings can vary in ranges between 800-1200mB. The Latitude Values will be ranged from -90 to 90 and -180 to 180 for Longitude.


<<requirement>> Location Tracking text="The system shall track Snakebot's live location" Id="008"
"deriveReqt"
"deriveReqt"
<<requirement>> Altitude Requirement text="Barometric Pressure Reader" Id="009"
<<requirement>> GPS text="Track Lattitude and Longitude" Id="011"
"refine"
"refine"
<<requirement>> Pressure Readings text="Pressure ranges between 800-1200mB" Id="010"
<<requirement>> Lattitude and Longitude text="Values range from -90 to 90 for latitude and -180 to 180 for longitude." Id="012"

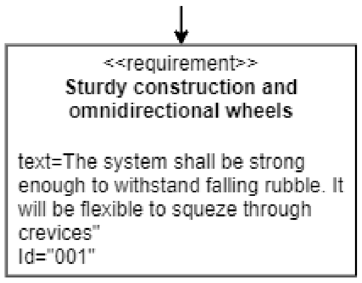
The bot needs to be able to identify points of interest. It will achieve this in three steps. Firstly, the bot will need to detect sound. A conceivable scenario is one in which the victim is calling out for help. The bot should be able to detect this by charting sounds ranging from 0 Hz to 40 kHz.


<<requirement>> Detect sound text="the system shall detect sound using ultrasound" Id="006"
"refine"
<<requirement>> Sound Specification text="Sound ranges between 0 Hz and 40000Hz" Id="007"

Secondly it needs to detect body heat. Heat Signatures for healthy humans can range from 308 to 311 Kelvin. The bot should be able to point out victims stuck underneath a building's rubble using the change in heat between surroundings. Last but not least it has to detect motion. It should be able to achieve this by detecting variation in Infrared Radiation.


<<requirement>>
Sturdy construction and omnidirectional wheels
text=The system shall be strong enough to withstand falling rubble. It will be flexible to squeze through crevices"
Id="001"
<<requirement>>
Detect body heat
Id="002"
"deriveReqt"
"deriveReqt"
"refine"
<<requirement>>
PIR sensor
text=" Detects motion with the variation of infared radiation
Id="003"
<<requirement>>
IR sensor
text="the sensor shall detect body heat signature using infared"
Id="004"
<<requirement>>
Heat Specification
text="Heat signatures range between 309 and 311 Kelvin "
Id="005"

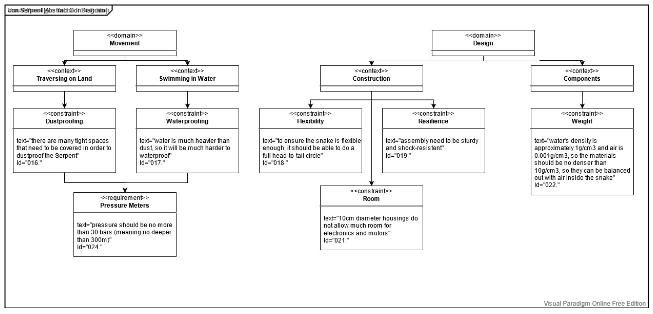
As the bot approaches the victim, it should be able to communicate with the person. This intends to establish a link between the first responders and the victim. They can better assess the situation a person is in by communication. The bot should be able to achieve this by using a speaker and a microphone. Another conceivable scenario is one where the victim is stuck underneath rubble but he/her is not critically injured. This will allow the First Responders to prioritize other victims who are in a more critical situation.


<<requirement>>
Sturdy construction and omnidirectional wheels
text=The system shall be strong enough to withstand falling rubble. It will be flexible to squeze through crevices"
Id="001"

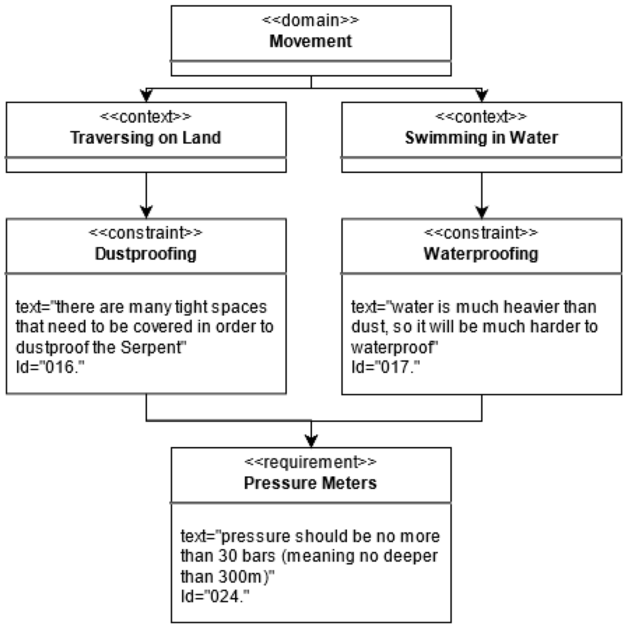
The body needs to be built of sturdy construction materials and can include Omni-directional wheels. It needs to be strong enough to withstand falling rubble. It needs to be flexible enough to squeeze through crevices. It is targeted towards Government bodies, First Responders and their needs. They need a bot that will enable them to carry out their responsibilities in a more efficient manner. They need to be able to assess the situation clearly and quickly. They need to be able to count the number of victims. They need to create a priority list of who should be rescued first. They need to identify sites of interest where a larger number of victims are situated.

A. *Constraints*

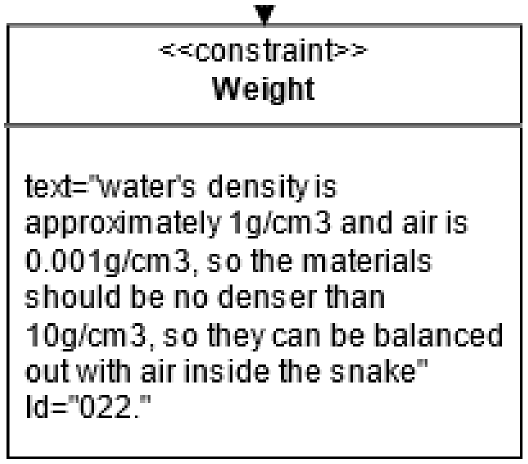
The system is a complex entity with lots of different parts. Each part warrants its own constraints and limitations.


con Serpent [Abstract ConDiagram]
<<domain>>
Movement
<<domain>>
Design
<<context>>
Traversing on Land
<<context>>
Swimming in Water
<<context>>
Construction
<<context>>
Components
<<constraint>>
Dustproofing
text="there are many tight spaces that need to be covered in order to dustproof the Serpent"
Id="016."
<<constraint>>
Waterproofing
text="water is much heavier than dust, so it will be much harder to waterproof"
Id="017."
<<constraint>>
Flexibility
text="to ensure the snake is flexible enough, it should be able to do a full head-to-tail circle"
Id="018."
<<constraint>>
Resilience
text="assembly need to be sturdy and shock-resistent"
Id="019."
<<constraint>>
Weight
text="water's density is approximately 1g/cm3 and air is 0.001g/cm3, so the materials should be no denser than 10g/cm3, so they can be balanced out with air inside the snake"
Id="022."
<<requirement>>
Pressure Meters
text="pressure should be no more than 30 bars (meaning no deeper than 300m)"
Id="024."
<<constraint>>
Room
text="10cm diameter housings do not allow much room for electronics and motors"
Id="021."
Visual Paradigm Online Free Edition

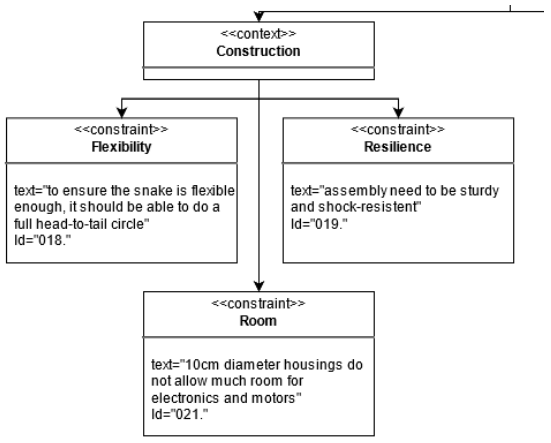
It needs to be able to drive over rocks and paths of differing heights. The terrain will almost never be a straight smooth road. Therefore, the bot needs adequate protection against the elements. It needs to protect its internal components from external dangers. These dangers include Water Damage, Fire Damage, Electrical and Dust Damage. The components have to be sealed tight and protected against aforementioned dangers. The wheels should be designed in a manner that external entities are not allowed to impede its operation.


<<domain>>
Movement
<<context>>
Traversing on Land
<<context>>
Swimming in Water
<<constraint>>
Dustproofing
text="there are many tight spaces that need to be covered in order to dustproof the Serpent"
Id="016."
<<constraint>>
Waterproofing
text="water is much heavier than dust, so it will be much harder to waterproof"
Id="017."
<<requirement>>
Pressure Meters
text="pressure should be no more than 30 bars (meaning no deeper than 300m)"
Id="024."

As the bot is amphibious in nature, it needs to be designed keeping buoyancy in mind. The density of the materials should not exceed the density of water. The closer it is to 1 Gram per Centimeter Cubed, the more of it will sit above the water level.

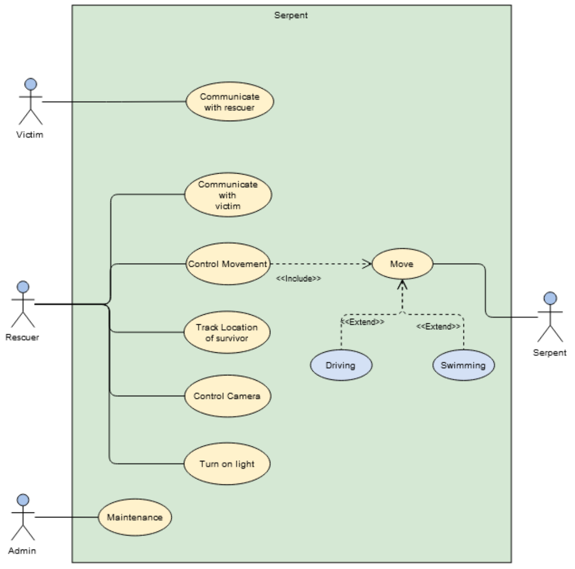

<<constraint>>
Weight
text="water's density is approximately 1g/cm3 and air is 0.001g/cm3, so the materials should be no denser than 10g/cm3, so they can be balanced out with air inside the snake" Id="022."

Furthermore, the external casing should be designed in a manner to improve aerodynamics in order to reduce drag and prevent undesired lift forces. There must be sufficient space in the internals.

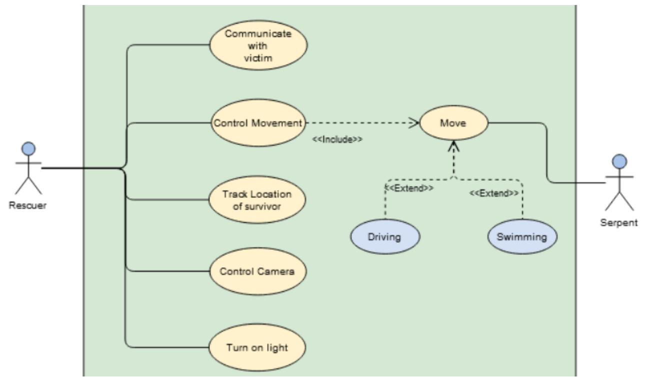

<<context>>
Construction
<<constraint>>
Flexibility
text="to ensure the snake is flexible enough, it should be able to do a full head-to-tail circle" Id="018."
<<constraint>>
Resilience
text="assembly need to be sturdy and shock-resistent" Id="019."
<<constraint>>
Room
text="10cm diameter housings do not allow much room for electronics and motors" Id="021."

For the snake design there are additional constraints: Waterproofing may be hard, especially in the joints between segments, same for dust proofing. This could be fixed by the aforementioned isolation around every joint, meant to protect the hydraulic pistons. It can be connected to the segments' endings and sealed with a strong adhesive. The parts where adhesive will be used are the rubber elements (with the function of protecting the pistons) and the main exoskeleton of the robot. Cannot bend very much (depending on the size of segments). The snake should be able to make a circle as a flexibility requirement. The ends should be able to touch each other. This is one of the goals to ensure enough flexibility during operation. Small segment housings do not allow a lot of room for electronics. They can be spread across multiple segments (or joints) to distribute free space, and therefore weight. This is as opposed to having everything in the head. Ballasts for submerging take up further valuable space. Joint spaces cannot be used as ballasts as the joints will lose flexibility. Combined design for rubble/dust and water traversing makes components for each action exclusive. For example, the water ballasts are of no use in a dry environment.

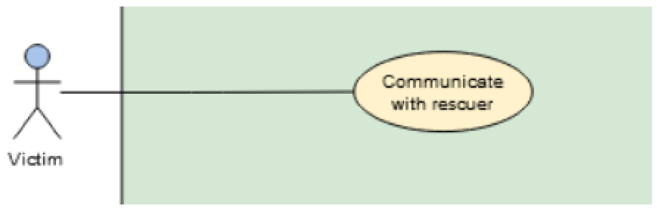
B. Use Case



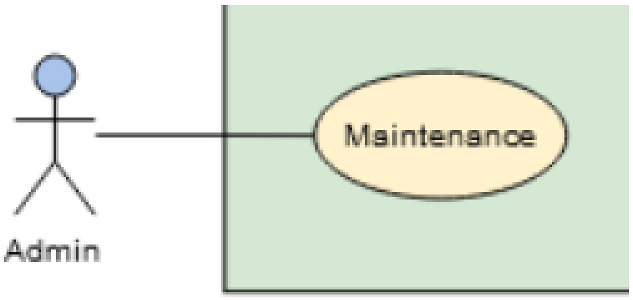
Much of the functionality goes to the Rescuer. They can use the terminal to control the movement of the bot. They can control the camera and rotate it to get a view. They can turn on its light. The terminal shows them the Latitude and Longitude of the bot. They can use this to track the location of the survivor. Then they can turn on the speaker and microphone to communicate with the Victim.



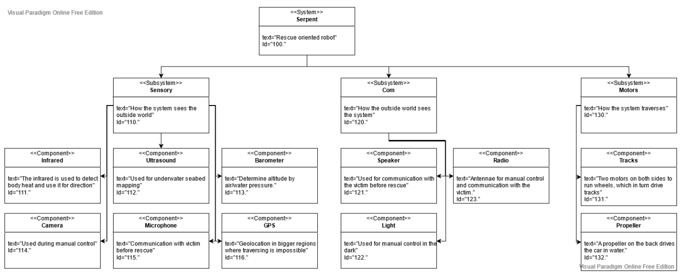
The Victim in return can also communicate with them



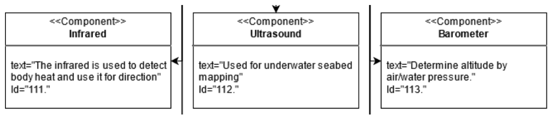
Lastly the Admin can perform maintenance on the bot.



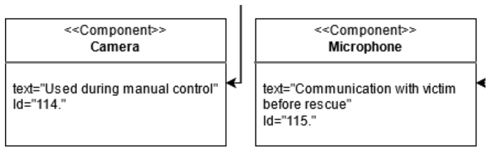
C. System Architecture

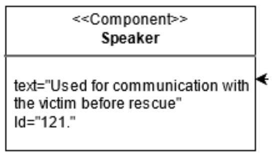

Visual Paradigm Online Free Edition
<<System>>
Serpent
text="Rescue oriented robot"
Id="100."
<<Subsystem>>
Sensory
text="How the system sees the outside world"
Id="110."
<<Subsystem>>
Com
text="How the outside world sees the system"
Id="120."
<<Subsystem>>
Motors
text="How the system traverses"
Id="130."
<<Component>>
Infrared
text="The infrared is used to detect body heat and use it for direction"
Id="111."
<<Component>>
Ultrasound
text="Used for underwater seabed mapping"
Id="112."
<<Component>>
Barometer
text="Determine altitude by air/water pressure."
Id="113."
<<Component>>
Speaker
text="Used for communication with the victim before rescue"
Id="121."
<<Component>>
Radio
text="Antennae for manual control and communication with the victim."
Id="123."
<<Component>>
Tracks
text="Two motors on both sides to run wheels, which in turn drive tracks"
Id="131."
<<Component>>
Camera
text="Used during manual control"
Id="114."
<<Component>>
Microphone
text="Communication with victim before rescue"
Id="115."
<<Component>>
GPS
text="Geolocation in bigger regions where traversing is impossible"
Id="116."
<<Component>>
Light
text="Used for manual control in the dark"
Id="122."
<<Component>>
Propeller
text="A propeller on the back drives the car in water."
Id="132."
Visual Paradigm Online Free Edition

The main system consists of different subsystems, which in turn house components. The robot needs to see the outside world, so many sensors take part in determining the actions.

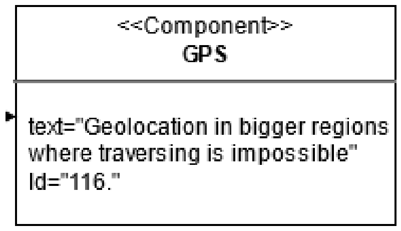

<<Component>>
Infrared
text="The infrared is used to detect body heat and use it for direction"
Id="111."
<<Component>>
Ultrasound
text="Used for underwater seabed mapping"
Id="112."
<<Component>>
Barometer
text="Determine altitude by air/water pressure."
Id="113."

First, objects need to be detected in front. For this, we use ultrasound, because it will also tell the distance between the object and the robot. We settled on a simple HC-SR04 sensor [4], which uses two speakers, one of which is re-purposed as a microphone. This saves on costs and is easier to use for our small-scale demonstration project (this will be a theme throughout the rest of the part run-through).

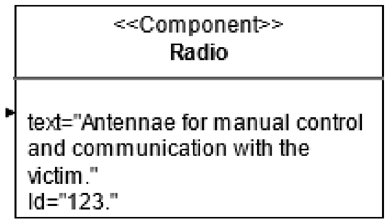

<<Component>>
Camera
text="Used during manual control"
Id="114."
<<Component>>
Microphone
text="Communication with victim before rescue"
Id="115."



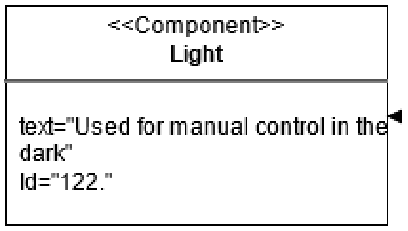
Because this is a rescue bot, the rescuers need a means of communication with the victim itself. The obvious solution to this is a simple microphone, camera and speaker setup. An electret microphone would be enough in this case, as we don't need the sound quality. Same goes for the speaker, anything above half an inch of diameter would be overkill. We decided to use a SparkFun Electret microphone [5] and IMX219 for the camera[6].


<<Component>>
GPS
text="Geolocation in bigger regions where traversing is impossible"
Id="116."

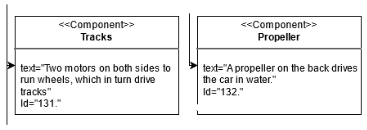
For any mode of control, whether it be auto or manual, the bot needs to be geo-located at all times. Rescue teams work in map coordinates, so they need a live feed of victim locations. Also, if anything happens to the communication line, meaning the connection to the robot is in any way is lost, we need a last known location. Parallax PMB-648 SiRF [7] is what we stopped on after some research. Geo-location uses two-dimentional plain positioning, which leaves out altitude. Therefore we use an alti-meter in the form of a barometer (SparkFun MPL3115A2, [8]).


<<Component>>
Radio
text="Antennae for manual control and communication with the victim."
Id="123."

These modules need enough channels for controlling the main two motors, plus one extra for the propeller motor, two more for microphone and speaker and most importantly, GPS and Camera transmission. In this case, much of the setup is similar to a normal drone, with the victim communication feature as bonus. With that in mind, we can use a conventional drone First-Person View interface [9].

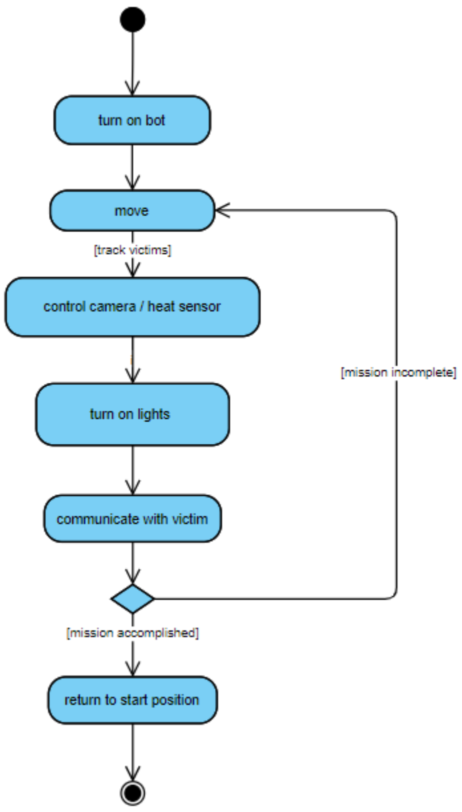


Perhaps the simplest of components so far.

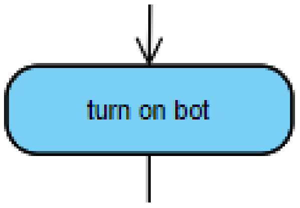

<<Component>>
Tracks
text="Two motors on both sides to run wheels, which in turn drive tracks"
Id="131."
<<Component>>
Propeller
text="A propeller on the back drives the car in water."
Id="132."

For moving the robot we use two motors attached on either side, which also allows steering in the form of differential drive. An extra motor is pushing the machine forward in water with a propeller normally suited for boats. Turning left and right in water mode is achieved by driving the respective track, essentially turning it into a paddle, to avoid using extra fragile parts for a fin. Of course, the RPi cannot provide the needed current to drive a motor, so the job is taken up by a shield (RPI SHD MOTOPI, [10]).

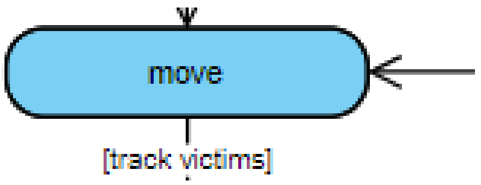
D. *Activity Diagram*



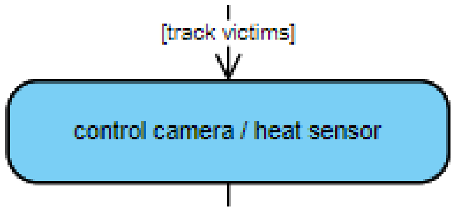
This shows a normal process of operating The Serpent.



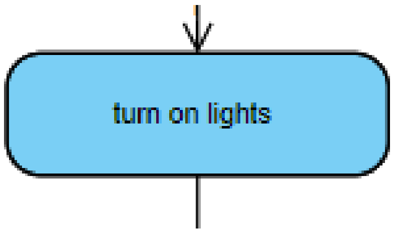
We begin by booting the RPi and establishing all connections necessary between the control center and our bot. These include syncing channels for manual control, plus audio and video.



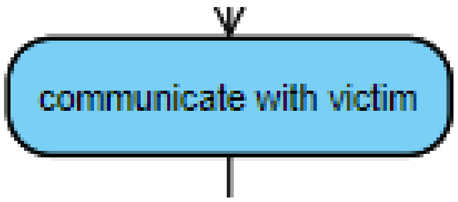
Depending on the mode chosen, the motor shield is connected either to the main algorithm for automatic search and rescue, or manual control. Small initializing movements are made to ensure all motors are functional.



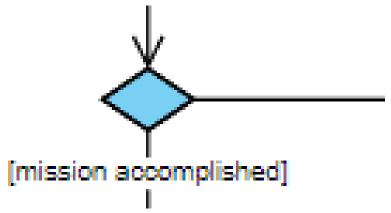
Either way, there is always a live feed of video and infrared footage to the control center. Of course, in auto-mode the infrared is also fed into the algorithm for targeting the victim by body heat.



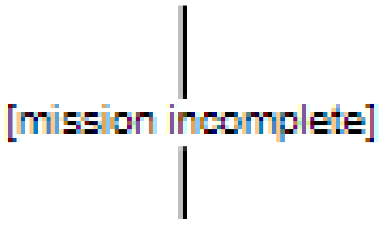
In dark, we have the option to light our way forward with a flashlight attached to the front.



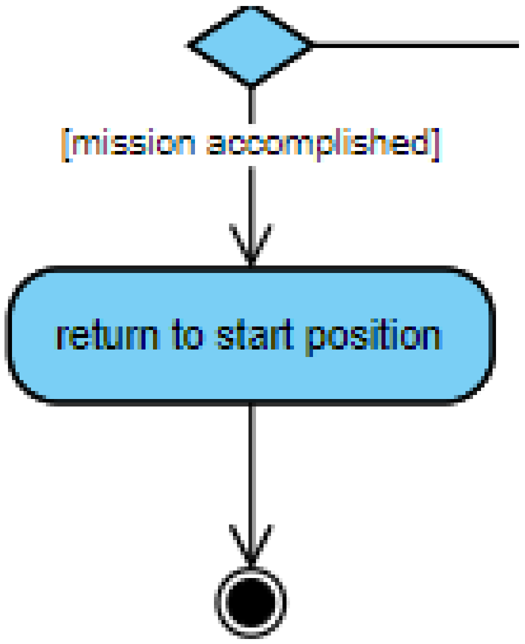
In the case of a found victim, if responsive, the rescuers can ensure them there is help on the way, bringing up spirit and in turn, increasing the chance of a successful rescue.



This represents the constant process of search, if the victim is found or not, thus determining whether to continue with the mission.

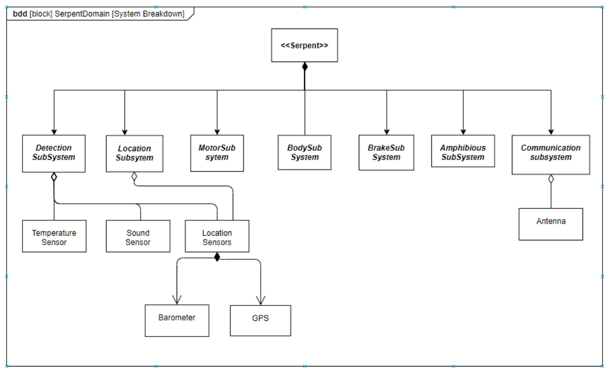


If not, we continue the search for victims under rubble.



When the victim has been located and contacted, the robot can return to the starting point.

E. *Block Diagram*

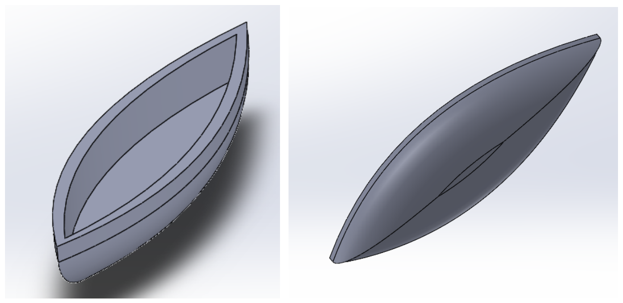


The Detection Subsystem consists of sensors that enable the bot to detect entities. These include the Temperature, Sound and Location Sensors. The Location Sensor is shared with the Location Subsystem. This is because the bot has two separate functions, one to locate points of interest. The other to keep track of the bots' location. The Location Sensors also include the Barometer and GPS. The Communication Subsystem includes the Antenna that is used for broadcasting. While separate, all of these Subsystems work in tandem with each other.

III. Design

A. *The Body*

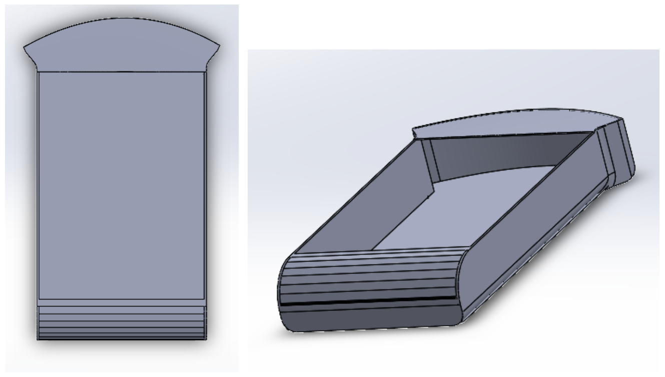
As we finally settled for a concept, we brainstormed ways of implementing what we had in mind, to start we needed the design to meet certain requirements, one of them being the ability to cruise over a water body, Our initial design has a base that was curved to minimise drag forces.



concept 1

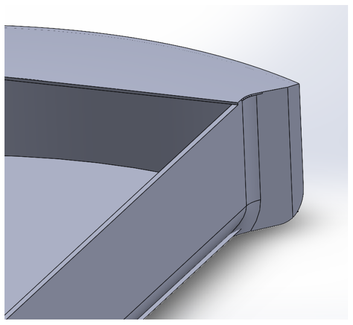
Later on, we realized that the approach we used was flawed as the curvature allowed for less space inside the model to place the components necessary to make the robot work as well as after some more in-depth research we concluded a curved base was not necessary to help our model float over water.

Our next attempt solved most of the problems we had, this design had sufficient interior space and followed aerodynamics rules, The front section of the body in concept 2 serves as a bumper, its main purpose is to move small obstacles out of the bot's way.

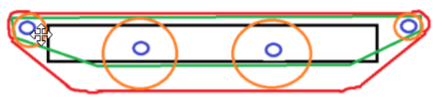


concept 2

As we started assembling the rest of the 3D parts we ran into a problem, we soon realized that the edge in the bumper would not allow us to attach the front wheel as sketched below:

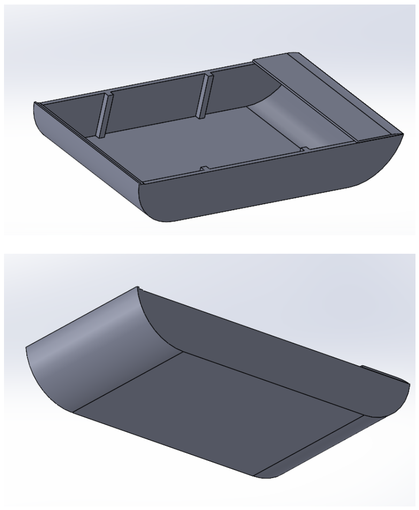


Bumper edge problem



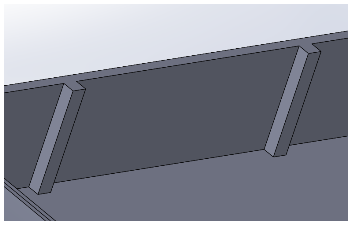
Wheel placement sketch

As it was clear to us which direction to pursue, we began to refine the model we had, we started by getting rid of the bumper and creating an elevation in the front part of the model instead, the elevation was to help support the front wheels, we achieved that by creating a slight curve upwards.



final concept

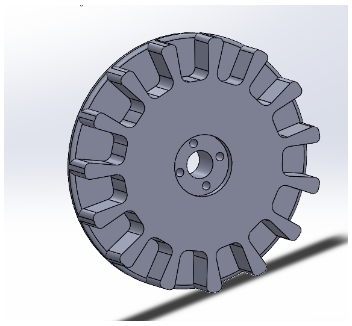
Inside the main body, we have also added four pillars to help secure the top piece in its place, We concluded that they would be necessary as we plan to use screws to attach the roof to the main body, it is also crucial for our model to be waterproof as it would be utterly damaging if water leaks into the interior part due to the presence of electronic components.



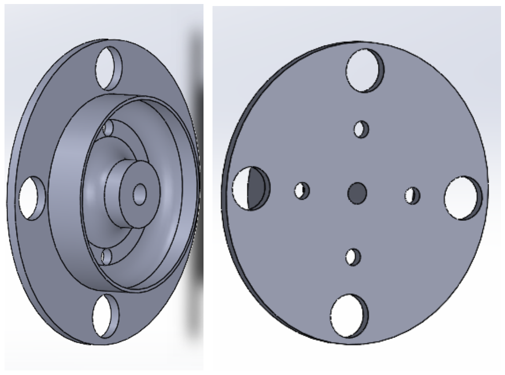
Pillars

B. The Wheels

For our model, we decided to use tracks as it can help our model cross rocky terrain more effectively, and for that we had to design two different kinds of wheels, A smooth surface wheel and a spiked wheel, the spiked wheels are used to move the belt around the wheels.

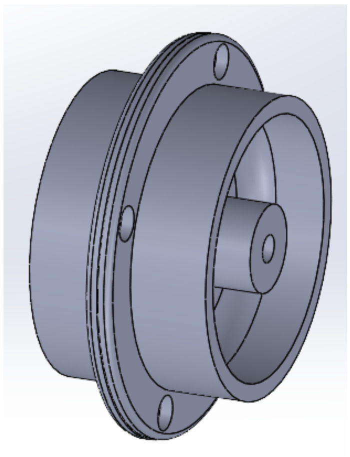


Spiked wheel



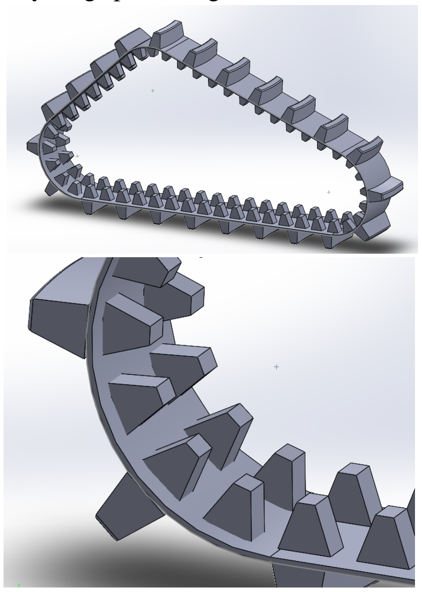
Smooth surface wheel

Keeping 3D printing time and cost of material in consideration, we created a wheel that consists of two parts that can be printed separately and later on glued together



Two wheel parts stuck together

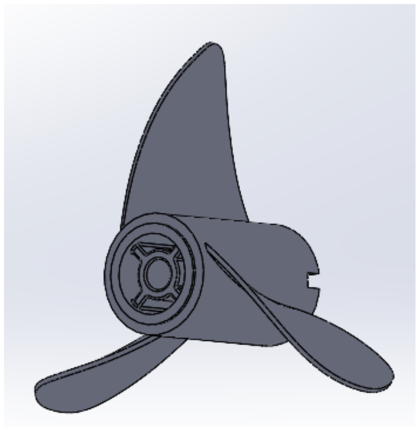
C. The Belt We added spikes on the belt to improve it's ability to grip to the ground.

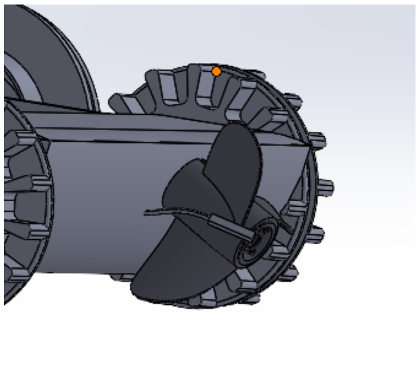


Belt close up

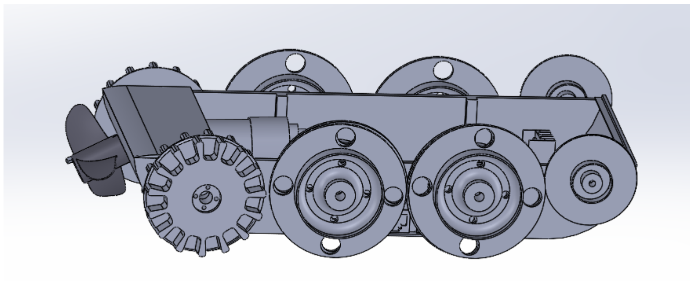
D. Propeller

The propeller is a crucial component to our bot as it allows it to accelerate while moving through a water body.


3-Blade Propeller


Propeller placed on the bot

E. Parts Assembled



IV. The code

A. Movement

Firstly, we found the indexes of T and R, using for loops that loop through the map and stop when they find T and R, and save their location as an index.
If the index of the robot is then smaller than the index of the target, we calculate the distance between them in the following way:

```
int distance = target_index - robot_index;
```

Once we know the distance, we used a for loop to find out the amount of lines, the for loop goes through the map, and counts every time it finds a newline character, but it starts at the robot index, and stops at the target index.

```
for(int i = robot_index; i < target_index; ++i) {
    if (world2[i] == '\n') {
        lines++;
    }
}
```

After counting the amount of lines between R and T, we code the movement accordingly, note that in this case, we know that R is smaller than T, which means R is above T, if the lines are more than 0. To code the movement, we used do-while loops, firstly, for the vertical movement, followed by the horizontal movement

```
//while R is on same line of T, go right.
do {
    for (int i=0; i<distance;i++) {
        return 2;
    }
}
while (lines < 1 && distance > 0);

//while R is on the same line of T, go left.
do  {
    for (int i=0; i<distance;i++) {
        return 4;
    }
}
while (lines < 1 && distance < 0);
```

The distance will be positive if T is to the right of R, and negative if to the left of it. We did the same exact thing for the rest of the code, but this time in the case that the robot index is bigger than the target index, everything would be opposite, the distance would be R-T, not T-R, the conditions for the do-while loops will be reversed (distance only).

B. Obstacles, water

1) Calculating the locations and movement

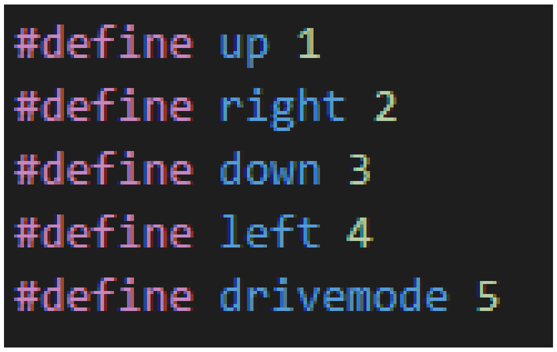
We already changed everything from the last stage, as we would have had to write code for 6 instances instead of 4. Instead of calculating which index is bigger first, and then calculate if there was vertical distance, we decided to do that directly, knowing the width is 21, we used a more simple method to calculate the positions:

```
//calculate horizontal and vertical R position
hR = robot_index % 21;
vR = (robot_index / 21) + 2;
```

Deciding the movement was done in the following way: If vR-vT > 0, it would mean that R needs to go up, as it's index is lower on the map than T, otherwise down, if 0, it is on the same line and the movement should be horizontal, then the same procedure is followed.

2) Simplifying the code

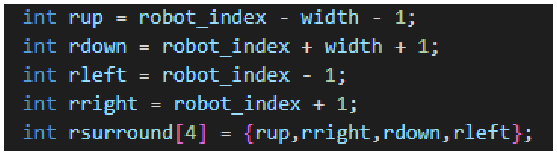
We decided to make the code more readable, and define the directions instead of using numbers



```
#define up 1
#define right 2
#define down 3
#define left 4
#define drivemode 5
```

3) *Surroundings of R*

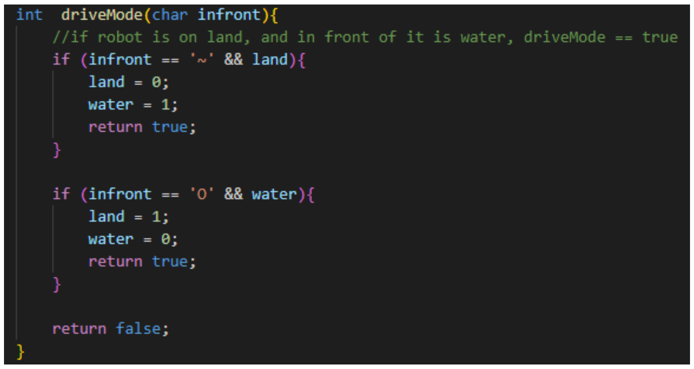
We stored the locations of the surroundings of R in the following variables:



```
int rup = robot_index - width - 1;
int rdown = robot_index + width + 1;
int rleft = robot_index - 1;
int rright = robot_index + 1;
int rsurround[4] = {rup,rright,rdown,rleft};
```

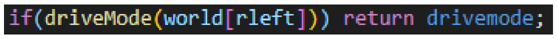
4) *Drive mode and water*

To tackle the water problem, we made a function that checks the char in front of the robot, and changes the mode if needed.



```
int  driveMode(char infront){
    //if robot is on land, and in front of it is water, driveMode == true
    if (infront == '~' && land){
        land = 0;
        water = 1;
        return true;
    }

    if (infront == 'O' && water){
        land = 1;
        water = 0;
        return true;
    }

    return false;
}
```

Before every return (movement), we check if the drive mode needs changing. For example, if we are going left, then before returning left, we always have the following if condition:



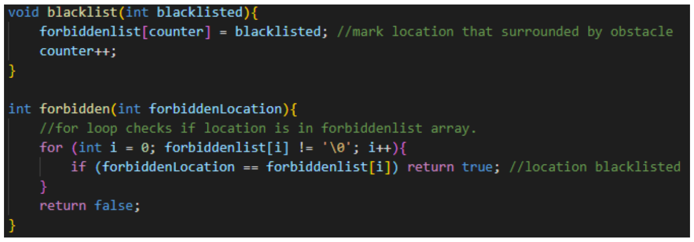
```
if(driveMode(world[rleft])) return drivemode;
```

5) *Horizontal and Vertical locks*

We introduced the idea of vertical and horizontal lock, which means that if R was going vertically, and it finds an obstacle, it will lock horizontal movement until that obstacle is solved. This is to prevent infinite loops, where R would be in the middle of avoiding an obstacle, and it would try to go horizontally once it reaches the same line as T, even though the obstacle was not avoided yet.

6) *Blacklist and forbidden locations*

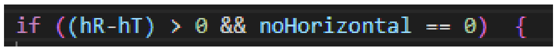
We decided on using a blacklist, as when the robot reaches a bad spot, (for example, surrounded by obstacles on 3 sides), it gets out of that spot and adds it to the blacklist, anything in the blacklist would be considered an obstacle, and the robot would not go there. These locations were stored in an array, and before every movement, we would check if it is in that array.



```
void blacklist(int blacklisted){
    forbiddenlist[counter] = blacklisted; //mark location that surrounded by obstacle
    counter++;
}

int forbidden(int forbiddenLocation){
    //for loop checks if location is in forbiddenlist array.
    for (int i = 0; forbiddenlist[i] != '\0'; i++){
        if (forbiddenLocation == forbiddenlist[i]) return true; //location blacklisted
    }
    return false;
}
```

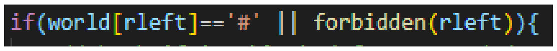
7) *Avoiding obstacles and blacklisted locations*

In this explanation, I will take the left movement as an example to how our robot avoids obstacles and moves around them.
Firstly, we know to go left by our horizontal locations



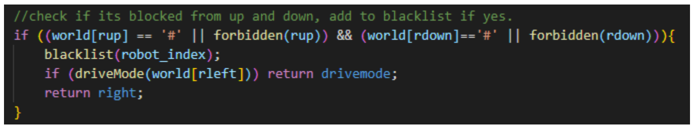
```
if ((hR-hT) > 0 && noHorizontal == 0)  {
```

If there are no obstacles to the left of R, we set the vertical lock to 0, check if we have to change the drive mode, and go left.
However, if there are obstacles to the left (or location is blacklisted), we have to avoid it in the following way:



```
if(world[rleft]=='#' || forbidden(rleft)){
```

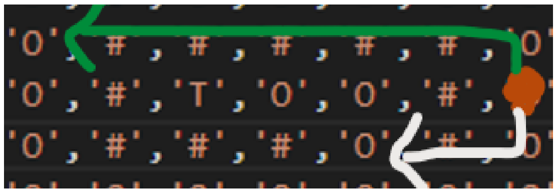
Before we do anything, we check if R is in a location surrounded by obstacles, if yes, we blacklist that location and go to the free direction.



```
//check if its blocked from up and down, add to blacklist if yes.
if ((world[rup] == '#' || forbidden(rup)) && (world[rdown]=='#' || forbidden(rdown))){
    blacklist(robot_index);
    if (driveMode(world[rleft])) return drivemode;
    return right;
}
```

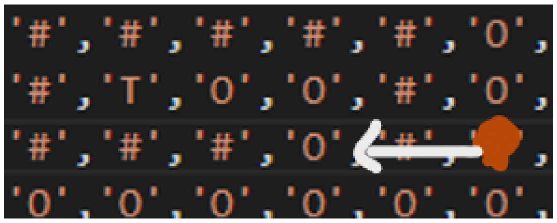
a) *The Algorithm*

The algorithm for avoiding obstacles works in the following way: in the pictures, R is red, countup is green, countdown is white.



In this case, R needs to go left, but the obstacle to its left is also surrounded by obstacles vertically, so R will calculate the amount of obstacles in the green arrow, when it reaches a free spot it will stop counting, and store that in a variable. It will do the same for the white arrow.

Once both variables are calculated, according to which one is lowest, we decide if the robot will avoid the obstacle by going up or down until there is no more obstacles on the left, followed by going left.



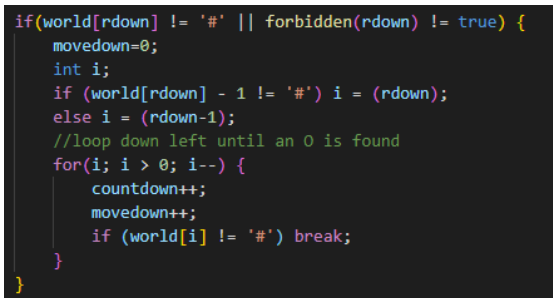
```
'#','#','#','#','#','O',
'#','T','O','O','#','O',
'#','#','#','O','#',
'O','O','O','O','O','O',
```

In the case one of the vertical directions of the left obstacle is free, then that will be prioritized, and instead of calculating down then left, we calculate straight left, then, the robot will go down until the left of it is not an obstacle (1 step down), then will go to the left, in this case 2 steps to reach an O.
After this is done, avoiding the obstacle will be done, and we will go back to calculating the movement direction using vertical and horizontal locations.

b) *The Code*

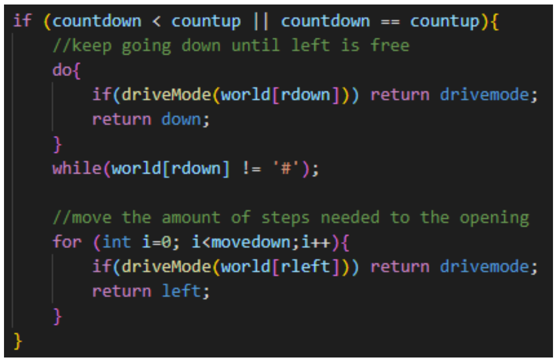
Note that the “count” variables are static variables, which means they will calculate the least amount of steps overtime, and the “move” variables are local and set to 0 before calculating movements needed.

Firstly, we check if below R is a free slot, then we set the movedown variable to 0.
We will then introduce the local variable "i", which will determine where our movement calculations start.
As mentioned in the algorithm explanation, if the position bottom left of the robot is free, then the count starts from rdown (straight below). Otherwise, it will start from the bottom left position.
We then use a for loop to count the steps needed. Our variable "i" was just determined, and it is our startpoint, our for loop goes on until we reach a free spot.



```
if(world[rdown] != '#' || forbidden(rdown) != true) {
    movedown=0;
    int i;
    if (world[rdown] - 1 != '#') i = (rdown);
    else i = (rdown-1);
    //loop down left until an O is found
    for(i; i > 0; i--) {
        countdown++;
        movedown++;
        if (world[i] != '#') break;
    }
}
```

After the steps of the movement are calculated, it is time for execution, in our case, our countdown would be smaller than our countup. The movement is again executed in do while loops, the robot will go down as long as the left of it is an obstacle, and below it is free. After it has gone down the appropriate steps, it’s time to go left, we calculated the steps needed before, and stored it in the movedown variable, so now we just use a for loop to execute these steps and go left.



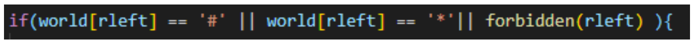
```
if (countdown < countup || countdown == countup){
    //keep going down until left is free
    do{
        if(driveMode(world[rdown])) return drivemode;
        return down;
    }
    while(world[rdown] != '#');

    //move the amount of steps needed to the opening
    for (int i=0; i<movedown;i++){
        if(driveMode(world[rleft])) return drivemode;
        return left;
    }
}
```

We are done with avoiding the obstacles!

C. *Extra obstacles, X location*

1) *New "*" obstacles*

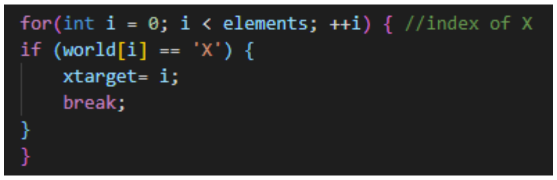
This stage was quite simple, all that was different from stage 2 was that there was a new type of obstacles, so wherever we had conditions of “#” or blacklisted locations, we added “*”.
Example: if to the left is an obstacle



```
if(world[rleft] == '#' || world[rleft] == '*'|| forbidden(rleft) ){
```

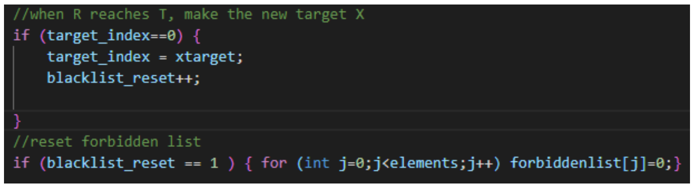
2) *Going back to the initial R location "X" after reaching T*

To do this, we only changed our “target index” to the location of X, after R has reached T. We used this loop to find the location of X



```
for(int i = 0; i < elements; ++i) { //index of X
if (world[i] == 'X') {
    xtarget= i;
    break;
}
}
```

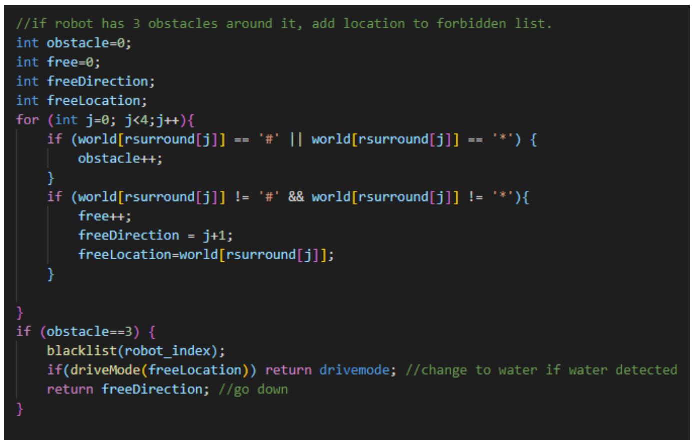
We noticed that when R reaches T, the target_index variable becomes 0, so knowing this, we changed the target_index to xtarget, when target_index became 0. We reset the blacklist and made sure it doesn’t repeatedly reset using static variable blacklist_reset.



```
//when R reaches T, make the new target X
if (target_index==0) {
    target_index = xtarget;
    blacklist_reset++;

}
//reset forbidden list
if (blacklist_reset == 1 ) { for (int j=0;j<elements;j++) forbiddenlist[j]=0;}
```

3) *Blacklist upgrade*

We upgraded our calculation of blacklisted locations in this stage, we used a for loop to count the obstacles around R, and free locations around R. If obstacles were 3, blacklist that location and go towards free location



```
//if robot has 3 obstacles around it, add location to forbidden list.
int obstacle=0;
int free=0;
int freeDirection;
int freeLocation;
for (int j=0; j<4;j++){
    if (world[rsurround[j]] == '#' || world[rsurround[j]] == '*') {
        obstacle++;
    }
    if (world[rsurround[j]] != '#' && world[rsurround[j]] != '*'){
        free++;
        freeDirection = j+1;
        freeLocation=world[rsurround[j]];
    }

}
if (obstacle==3) {
    blacklist(robot_index);
    if(driveMode(freeLocation)) return drivemode; //change to water if water detected
    return freeDirection; //go down
}
```

D. *Destroying obstacles, rescuing extra target "t"*

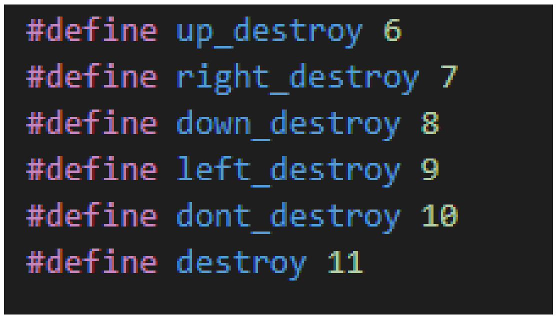
1) *Water target "t"*

For finding the water target t, we only added an OR condition in the loop that finds the index of the target T, to stop if it reaches "t".

2) *Destroying the "*" obstacles*

a) *Return values*

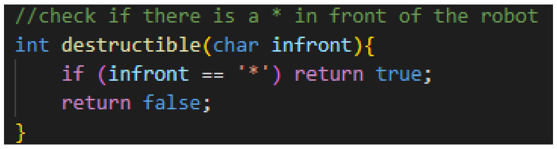
Firstly, we defined the returns for the destruction of obstacles



```
#define up_destroy 6
#define right_destroy 7
#define down_destroy 8
#define left_destroy 9
#define dont_destroy 10
#define destroy 11
```

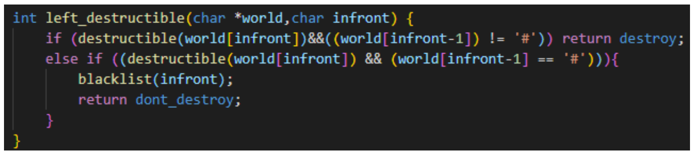
b) *functions*

To make this simple, we created a function to check if the location in front of us is a '*', similar to the drivemode function.



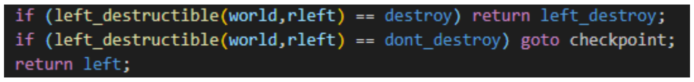
```
//check if there is a * in front of the robot
int destructible(char infront){
    if (infront == '*') return true;
    return false;
}
```

After doing this, we decided it would be simple if we create a "direction-destructible" function, that checks if it is even worth destroying that obstacle, since it costs some energy! The reason these functions are named for each direction, is because we have in front of each variable too, and see if it is blacklisted or a '#'. Again, we will take the left direction as an example



```
int left_destructible(char *world,char infront) {
    if (destructible(world[infront])&&((world[infront-1]) != '#')) return destroy;
    else if ((destructible(world[infront]) && (world[infront-1] == '#'))){
        blacklist(infront);
        return dont_destroy;
    }
}
```

We check if in front of us is a '*', AND if in front of that is free, if yes, then we return "destroy" value, which lets us know if we should destroy the obstacle or not. If in front of our '*' is an obstacle, we blacklist that location, and we return "dont_destroy" value. Now, before every return value, we have this condition:



```
if (left_destructible(world,rleft) == destroy) return left_destroy;
if (left_destructible(world,rleft) == dont_destroy) goto checkpoint;
return left;
```

Note that the checkpoint is when we start calculating the movements according to horizontal and vertical locations of R and T.

V. Conclusion

Right from the beginning of this project, I think our team did a good job when it came to making decisions and especially, to always keep an open mind while working on the concept and ideas. The pre-built part of design was the most interesting, because every design and functionality idea for the model would influence a lot the development of the project. Each in the team had their views and opinions change or at least shift overnight after our team meetings. Everyone's way of flexible thinking allowed us not to stall on the project work. The coding part also presented a good challenge, as it offered the experience of creating a pathfinder in a digital two dimensional environment. A lot of work and time was invested in the code. In the end, I think we matched the objectives of this project successfully on all tasks and exercises. We learned a great deal about working in a team and the experience we gathered here for sure will be used in next projects.

VI. Appendix

A. *Contributions and task delegations during the semester*

1) *Part One (Requirements)*

- Hanan Korabi: Requirements, physical constraints, SysML diagram, Use case diagram
- Kaloyan Staykov: Constraints diagram, Sub-system diagram
- Cristian Cernat: Componenets diagram, 5 layer diagram
- Dania Al Hakim: Activity diagram, Requirements.
- Shehroz Bashir Malik: Requirements, SysML diagram, Block Diagrams

2) *Part Two (Design)*

- Dania Al Hakim: designed all 3D parts in Solidworks.
- Cristian Cernat: Did the initial 3D design of the "Serpent snake design" before we changed it, suggesting ideas and helping with the design
- Hanan Korabi: Suggesting ideas, helping with the design and providing resources
- Kaloyan Staykov: Suggesting ideas, helping with the design, picking parts, and providing sketches and hardware knowledge.

3) *Part Three (Code)*

- Hanan Korabi: Wrote all the code for the programming tasks.
- Dania Al Hakim: Suggesting ideas, helping with creating algorithms to solve the problems, debugging the code
- Cristian Cernat: Suggesting ideas, helping with creating algorithms to solve the problems, debugging the code
- Shehroz Bashir Malik: assisting in debugging two minor bugs.

B. *Estimated percentages of work per member*

- Dania Al Hakim: 28%
- Hanan Korabi: 28%
- Cristian Cernat: 17%
- Kaloyan Staykov: 17%
- Shehroz Bashir Malik: 10%

C. *Documentation credits*

1) *Requirements*

- Cristian Cernat: Introduction, Conclusion
- Kaloyan Staykov: System Architecture, Activity Diagram
- Shehroz Bashir Malik: Requirements, Constraints, Use Case, Block Diagram

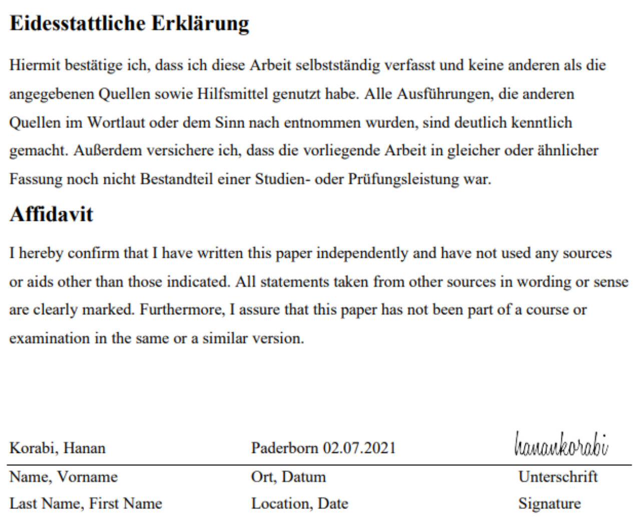
2) *Design*

Dania Al Hakim

3) *Code*

Hanan Korabi

VII. Affidavits



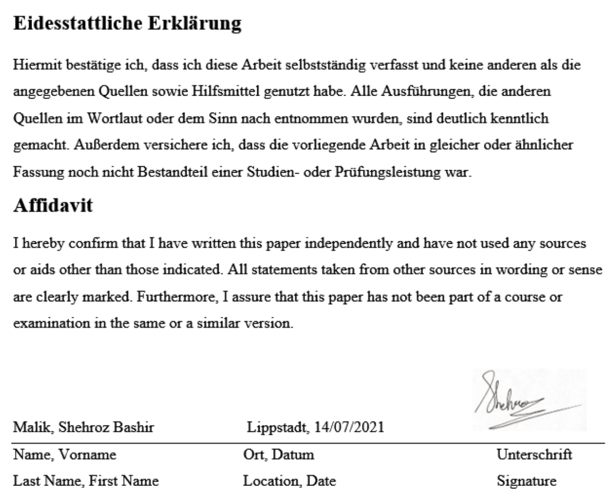
Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Korabi, Hanan	Paderborn 02.07.2021	hanankorabi
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature



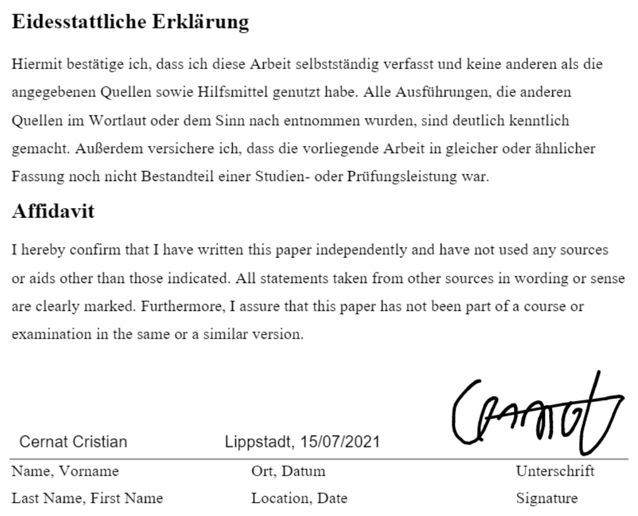
Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Malik, Shehroz Bashir	Lippstadt, 14/07/2021	[illegible]
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature



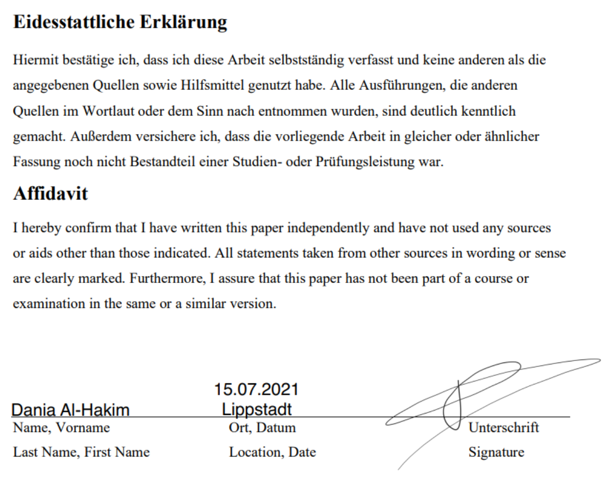
Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Cernat Cristian	Lippstadt, 15/07/2021	[illegible]
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature



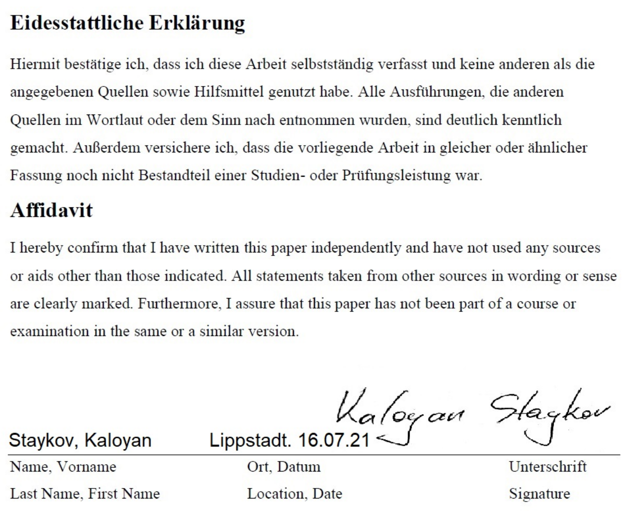
Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Dania Al-Hakim	15.07.2021 Lippstadt	[illegible]
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature



Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Staykov, Kaloyan	Lippstadt. 16.07.21	Kaloyan Staykov
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature

References

[1] Global reported natural disasters by type, 1970 to 2019. https://ourworldindata.org/grapher/natural-disasters-by-type

[2] EM-DAT | The international disasters database. http://ghdx.healthdata.org/gbd-results-tool

[3] GBD Results Tool | GHDx http://ghdx.healthdata.org/gbd-results-tool

[4] Part: Ultrasonic Sensor | https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04

[5] Part: Electret Microphone | https://www.sparkfun.com/products/12758

[6] Part: RPi Camera | https://www.newegg.com/p/1B4-06RX-07K93

[7] Part: GPS | https://www.jhongelectronics.org/2013/10/arduino-parallax-gps-module-pmb-648-sirf.html

[8] Part: Barometer | https://www.sparkfun.com/products/11084

[9] FPV Drone Technology | https://www.themodernrogue.com/articles/2018/12/8/video-fpv-drone-interfaces

[10] Part: Motor Shield | https://www.reichelt.de/raspberry-pi-shield-motopi-motorsteuerung-rpi-shd-motopi-p202551.html?CCOUNTRY=445LANGUAGE=de