Redesigning Visualizations on the Zero Robotics Website: UAP Final Report

Miriam Prosnitz

Background

Zero Robotics

The Zero Robotics High School Competition (ZR HS Competition), a robotics programming competition, requires high schoolers to program SPHERES (Synchronized Position Hold Engage and Reorient Experimental Satellites) to complete specific challenges. The SPHERES, small autonomous satellites, can be equipped with additional tools (such as cameras or lasers) and use twelve carbon dioxide thrusters to direct themselves. The specific challenges of programming SPHERES include memory limitations, tool constraints, and the need for reliable autonomous code. Programming SPHERES teaches high schoolers how to program for the aerospace industry.

Each year the ZR HS competition challenge changes. For example, the 2015 challenge required the students to program the SPHERES to take pictures of points of interest on an asteroid while the 2016 challenge involves orbital periods and solar power. This means that the important data for each year's game is variable. These variations make for a lively competition which has grown over the past six years to include schools globally.

The ZR competition is highly structured with tests and the initial rounds of the competition run as simulations on the Zero Robotics website

(zerorobotics.mit.edu). The final rounds of the competition are run live from the International Space Station (ISS). This project will focus primarily on how the simulations are run on the ZR website.

Visualization Specific

This project was developed to replace old simulations on the ZR website. The old simulations were run using Flash, which presented issues. First, Flash requires a download to run, is slow, and is gradually being phased out by the tech community. More importantly, the old Flash website was inflexible and more simulation flexibility was needed. Ideally, the simulation should be adjusted every year to represent data specific to that year's challenge, such as how many pictures were taken or if a SPHERES is colliding with an obstacle. The new simulation can easily handle this additional data, but the old simulation could only handle minimal adjustments, such as for symmetrical obstacles (see figure 1 below). Since in the old Flash simulation the SPHERE movement representation and data are all part of a single module, adding or removing data from the display becomes a complex, time-intensive process. Instead, students were expected to monitor a large number of game variables using print statements which quickly become cluttered. This is why a new solution was developed using current technology that has increased simulation flexibility.

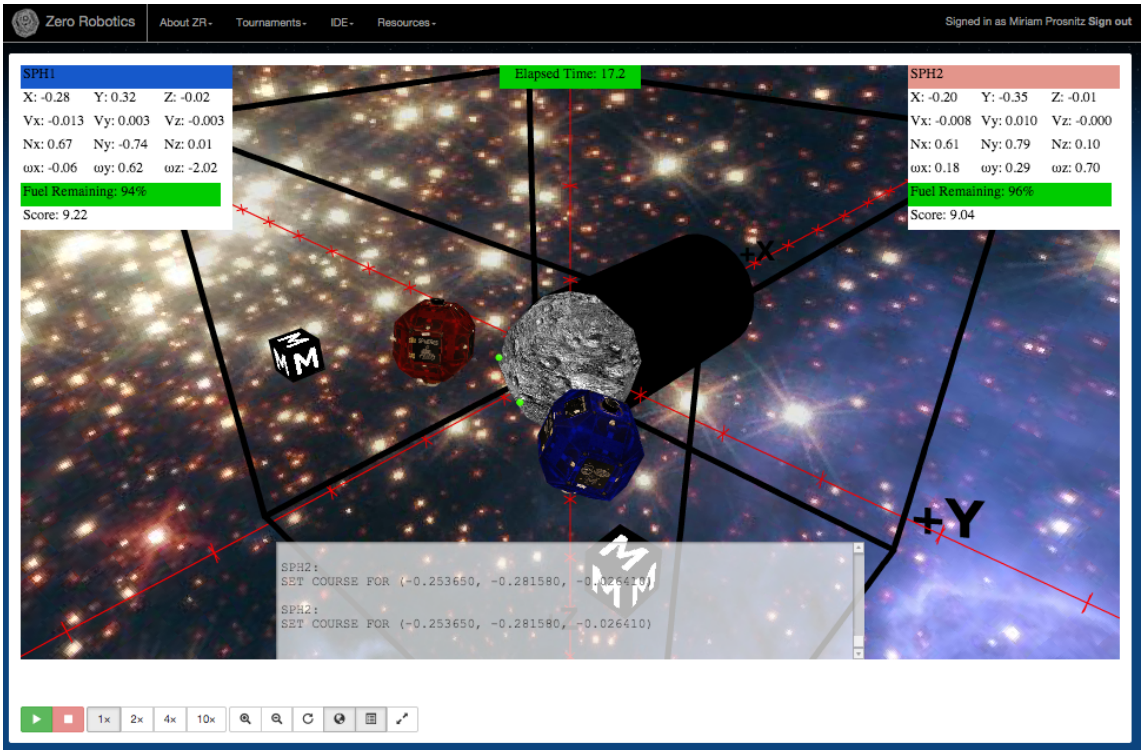

Figure 1: The old Flash simulation, though the symmetrical comet and shadow are shown, data such as whether a SPHERE is aligned with a point of Interest is unclear

Architecture

ZR System Overview

The ZR simulation system utilizes the user's machine and the ZR servers. Once the user has inputted their code and simulation configurations into the ZR website, that data is sent back to the ZR servers. The ZR servers then runs the simulation, puts all the simulation data in results.json, and sends results.json back to the user's computer. At this point the user can play the simulation. To

play the simulation, the visualization first parses and then processes results.json. This process is illustrated in figure 2.

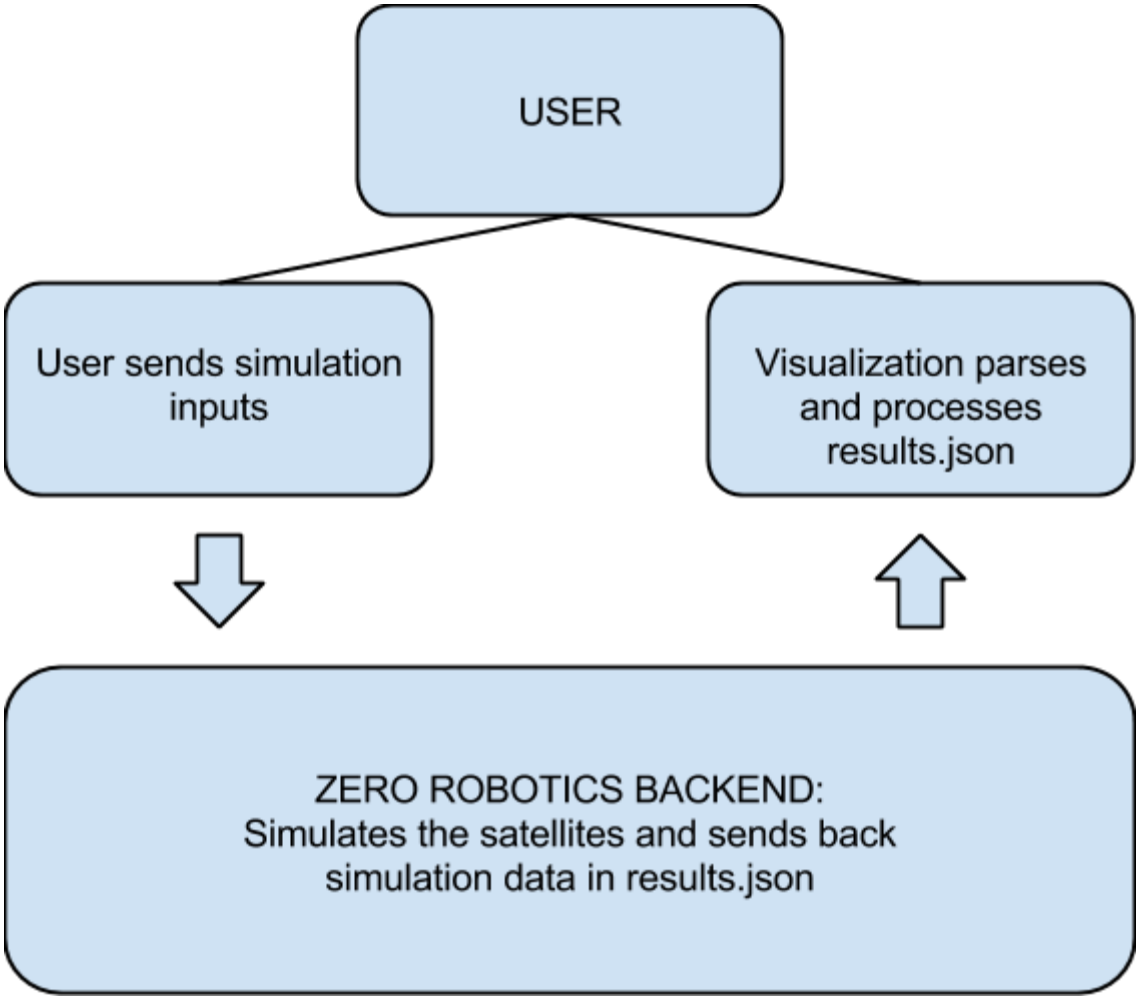

Figure 2: Diagram of the ZR simulation system

Results.js

JSON, JavaScript Object Notation, is an efficient way to store and transmit data in Javascript. Results.json contains all the simulation data generated by the ZR SPHERES simulation. Its formatting is based on the data real SPHERE

satellites can store and transmit. Therefore, its formatting is constant and contains the following:

baseSeeds			
codegen_ver			
results			
	0		
	1		
satData			Satellite Data: Both satellites can transmit a limited number of variables of three types (float, short, and unsigned short) during gameplay
	0		Satellite Zero's simulation data
		dF	Float debug: Contents change yearly
		dS	Short debug: Contents change yearly
		dU	Unsigned Short debug: Contents change yearly
		dUser	User debug: Interplayer communications
		st	Satellite movement data: -0,1,2 x,y,z (positions) -3,4,5 vx,vy,vz (forces) -6,7,8,9 quaternions (rotation)

			-10,11,12 angular velocities
		txt	Satellite Debug Information
	1		Satellite One's simulation data - See satellite zero for internals (they have identical structures)
simulationID			
standalone_ver			
tDbg			Times for dF dS and dU
tSt			Time (the number of elements should equal those in st
tTxt			Debug Information

The specific contents of the sat (satellite) section of Results.json changes every year. One place to check a given year's contents is to look at svn/ZeroRobotics_VM/Games/*GAME*/common/ZRGameInternal.cpp (where *GAME* must be replaced with a specific game such as ZRHS2014), specifically at the sendDebug method. It is important to note that the placements in results.json are the placements indicated in the sendDebug minus one (since the time step is stored elsewhere). Therefore, if the sendDebug method says that the fuel is in

Float debug spot two, it will actually be found in dF spot one. A code snipped from ZRGameInternal.cpp is shown in the figure below.

```
//Float debug packet: score, fuel, forces
DebugVecFloat[0] = (float)tstep;
DebugVecFloat[1] = game->getScore();
DebugVecFloat[2] = game->getFuelRemaining();
```

Figure 3: Code from sendDebug method of ZRGameInternal.cpp for ZRHS2014

A useful tool when looking at results.json to confirm the data is valid is a JSON parser such as the one on www.jsonparser.org (pictured below). This shows the hierarchy of the data and allows for it to be quickly checked. The figure below shows an example of this using data from a simulation run using the challenge from 2014.

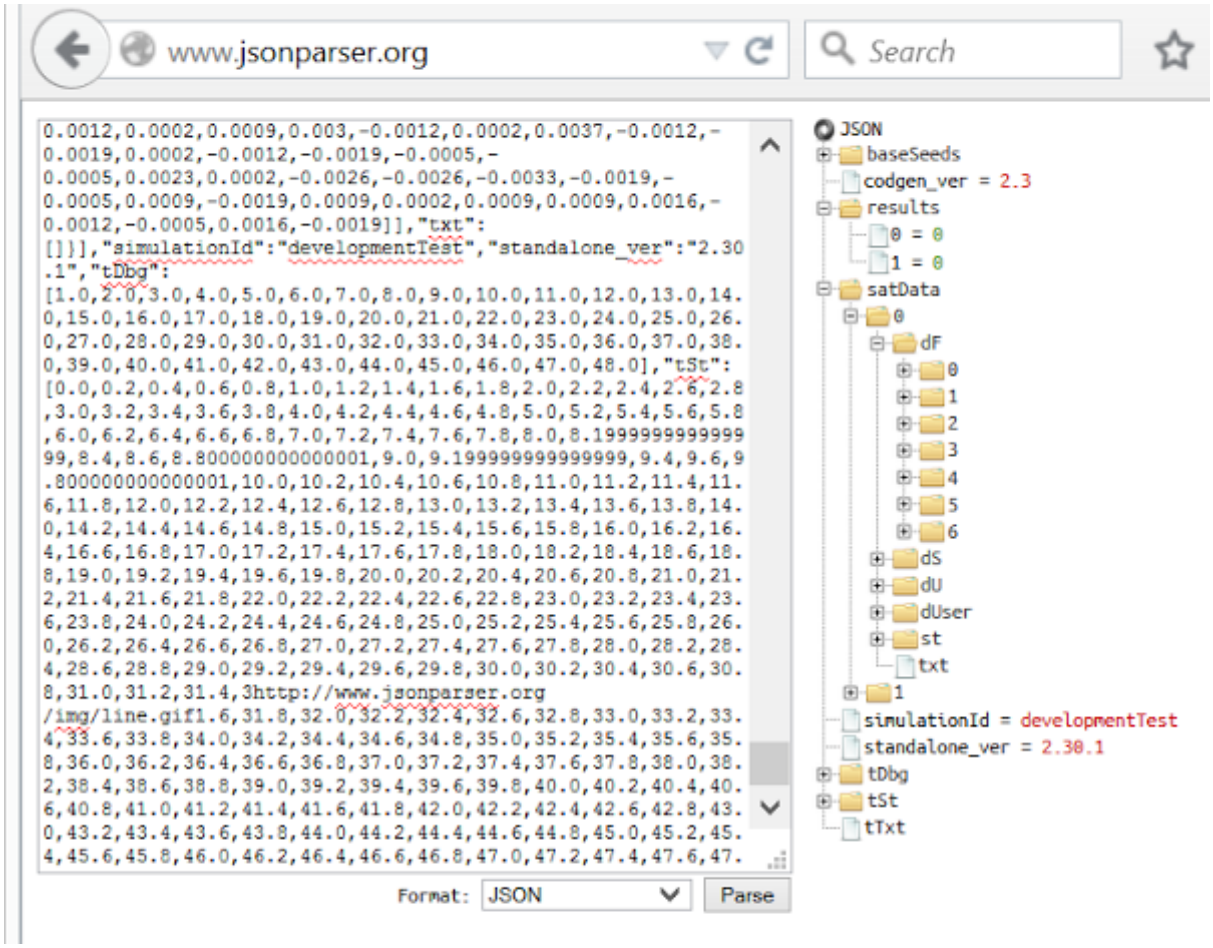

Figure 4: results.json from a 2014 simulation parsed on jsonparser.org

Implementation

The following diagram shows in pictorial form the visualization implementation:

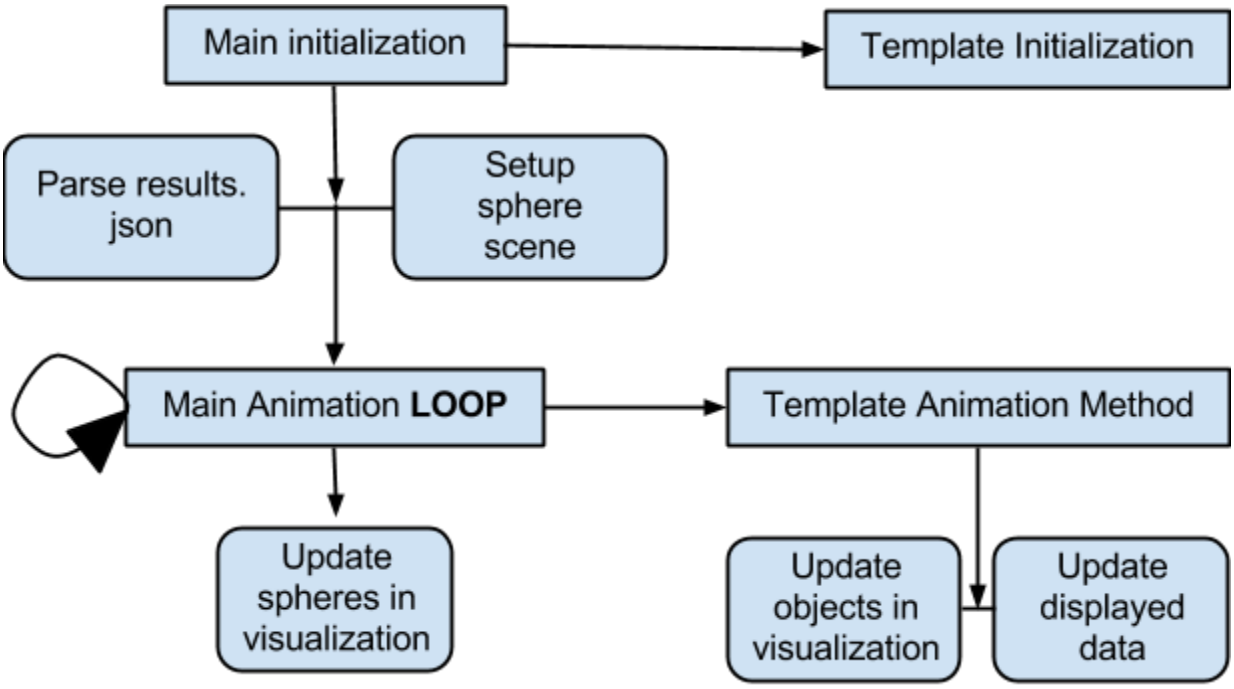

Figure 5: Overview of visualization implementation, where rectangles represent methods and template stands for the game specific code file

Before animation can occur, results.json must be parsed and the three.js scene must be setup. Parsing results.json is done using the built-in function JSON.parse(). A second function, createAnimData(), will later pull and format the necessary data for a three.js animation. It is important to note that createAnimData() depends on a separate method, getDataLoc(), found in the year specific code, in order to locate the data it needs within the parsed results.json. This decouples the the placement of the satellite data within results.json from the data; hence if items shift within results.json it will only affect one area of the code (see Figure 6). Setting up the three.js scene involves defining the space, the

camera relative to the space, and adding axis. Once this is complete alongside any challenge specific initializations, animation can begin.

```
//Tells where in the results.JSON different variables are stored
function getDataLoc(dataType){
  if(dataType == 'fuel')
    return ["dF", 1];
  if (dataType == 'score')
    return ["dF", 0];
}

//Updates data displayed on the page
function updateRelatedData(){
  var time = parseInt(sphereMeshAnimation.currentTime);
  if( time<JSONObj.satData[0]["dF"][1].length){
    var fuelLoc = getDataLoc('fuel');
    document.getElementById("fuelRed").value = JSONObj.satData[0][fuelLoc[0]][fuelLoc[1]][time];
    document.getElementById("fuelBlue").value = JSONObj.satData[1][fuelLoc[0]][fuelLoc[1]][time];
    }
}
```

Figure 6: Shows decoupling of where data is stored in results.json and the pulling of data from the parsed data

Animation functions by playing the Three.js animations for the different items, satellites included, synchronously. Changing the animation speed, pausing, and restarting is all done through the Three.js Animation class. Other data is displayed by taking the time from the animation, using it to pull the other relevant data, and updating the HTML dynamically.

Files

MiriamTest.html	Establishes the basic html components which are

(RENAME)	then modified by the Javascript
MiriamFunctions.js (RENAME)	Provides the fundamental code which is relatively static from year to year and establishes the animation and animates the spheres
template.js	Where the challenge specific visualization is implemented

API Setup

This API was set up so that adding to the visualization for a specific year's challenge could be accomplished entirely through template.js. Annually static elements of the visualization, such as the 3D scene and the animation of the two competing spheres, is handled in the file Miriam.Functions.js (RENAME). Functions in this file call and therefore depend on functions in template.js. Due to this dependency, in order for this API to function properly the following methods must be implemented in template.js for the specific year's challenge:

initTemplate()	Initialize anything needed, this function is called when the web page is first loaded
animateTemplate()	Called every animation frame, used to update other components on the page
getDataLoc(dataType)	Indicate where in the results.JSON file all the

	variables are stored, the parameter dataType is a string (example: “fuel”)
updateRelatedData()	Update data displayed on the page by dynamically changing the HTML
addStats()	Another way to display data on the page, add stats to the context.html item; may be better suited for pictorial information
addItemToAnim()	Adds items to the three.js scene

The following methods must also be implemented in template.js and should be used to update animated items in the three.js scene excluding the main spheres which are handled elsewhere.

pauseAnimationItems()	Pause the animation
changeSpeedItems(speed)	Speed up or slow down items in the animation, the parameter speed is an integer
playAnimationItems(time)	Play the animation beginning at the given start time, the parameter time is a time object

Example implementations of these functions can be found in template.js.

Future

This project was created to develop a visualization system for future ZR challenges. Looking at the final product in figure 7, clearly, this system needs to be updated making it more visually appealing, before future use.

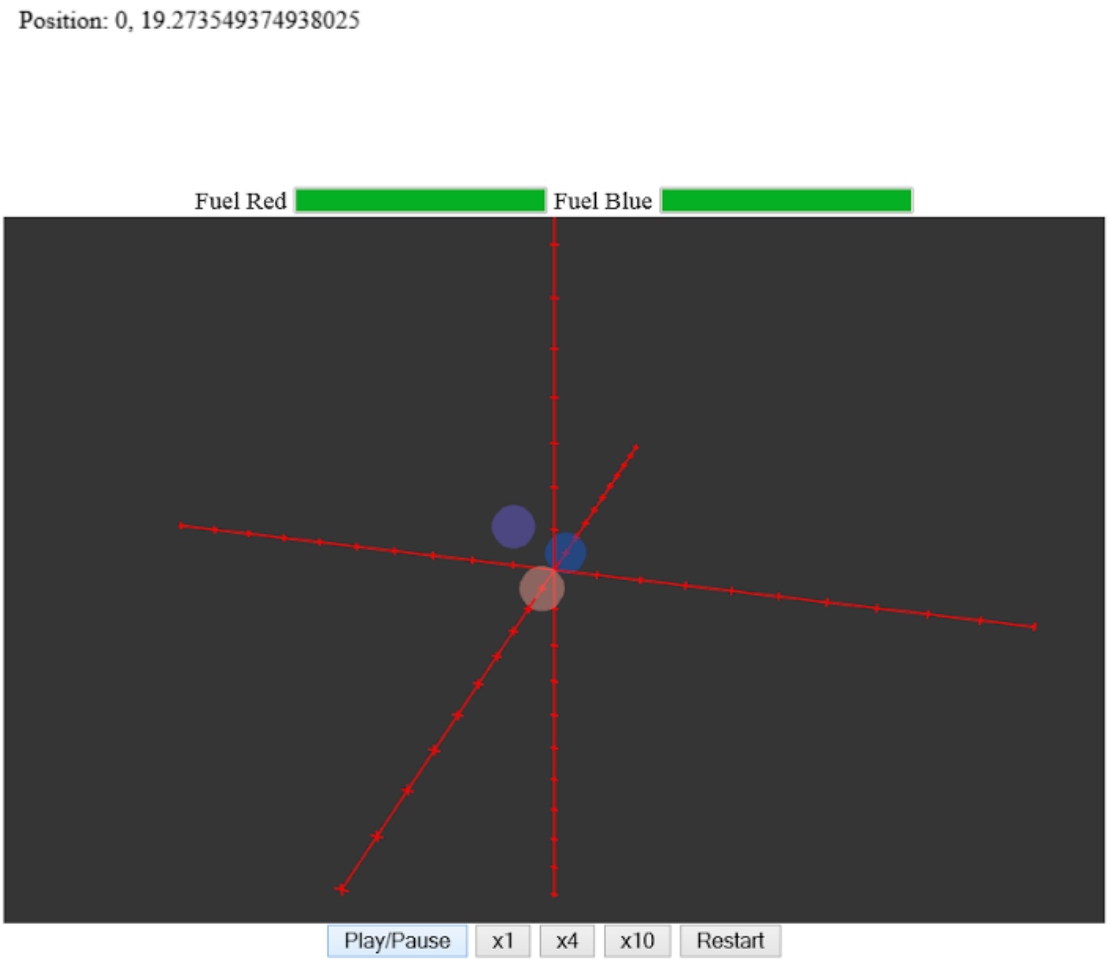

Figure 7: A visualization run in the new system

Conclusion

This ZR visualization redesign differed significantly from the old Flash implementation. Like the previous implementation, developing this simulation fundamentally depended on the transmission, parsing, and processing of data. Unlike the old Flash implementation, this visualization depends on Javascript and Three.js. Using these current technologies allows for increased future development, giving this ZR visualization a lot of potential. Hopefully, this ZR visualization will enable future ZR competitors to have better experiences while learning to program.