**2012**

***Multiple, Independent, Automated Robots***

**By:**   
Mayank Gureja

Abstract

This report describes a Cyber-Physical System (CPS) that uses a Microsoft Kinect sensor to provide input and a series of simulated robots that react to the provided input. The underlying code is built entirely in C# using various Microsoft Frameworks that allow for asynchronous concurrency and the building of services that interact with each other, in order to create rich, human controllable applications.

In this project, multiple robots are created in a simulated environment and controlled via voice commands through the Kinect audio sensor and speech recognition software. This particular application allows the various robots to be controlled individually and independently, without any interference in the backend functioning of the robots.

Introduction

This project is rooted in the decade’s old pursuit to build robots that react to human commands and perform tasks autonomously. Research in this area has been ongoing for decades and many, many sophisticated systems have been built in recent years. Some of the most intriguing involve the use of robotics in military and also some that accomplish more cerebral objectives such as playing chess.

An example of a Cyber-Physical System (CPS) developed for use in the military is Big Dog, a “rough-terrain robot that walks, runs, climbs and carries heavy loads”[1], built by Massachusetts-based Boston Dynamics. This 4-legged machine reacts to changes in its environment and adjusts its activities accordingly, in order to accomplish its goals. Another great example of a CPS are the Quadrotor swarm robots designed by the University of Pennsylvania’s GRASP (General Robotics, Automation, Sensing and Perception) Lab[2]. They are able to work together or alone. They can fly in formation and adjust their individual trajectories to avoid collisions or obstacles. One of the CPSs that I’ve personally drawn inspiration from in the past is ChessKA[3] designed by former Russian chess world champion Konstantin Kosteniuk. ChessKA is a software/hardware solution to the problem of building autonomous chess playing systems. It has a robotic arm that moves the chess pieces and the chessboard itself is modified with sensors that input the relevant data to the computer.

Building a robust and useful CPS is not a trivial endeavor. Hard coding predetermined activities is relatively easy to do, so long as one has enough information about the conditions where the system will be run and assurance that nothing in the environment will change. However, to build a true CPS system that reacts to its surroundings and manages multiple tasks in real time requires a few more tricks. For one thing, any such project will require multiple components to be spinning continuously. This includes performing the primary task, waiting for environmental stimulus, pausing long-running activities to react to any environmental inputs and maintaining the previous ‘trajectory’ once any interrupts are handled.

The Microsoft DSS (Decentralized Software Services) and CCR (Concurrent and Coordination Runtime) are 2 frameworks that allow for all the above. The DSS is a service-based paradigm, where applications act as services that can be ‘subscribed’ to, in order to extend the functionality of an application without the need to expand on the application itself. For example, earlier in this term we combined a Kinect sensor’s gesture capabilities[4] to a ticking clock, in order to build a tick counter that reacts to user input. This was done by subscribing a central application service to the Gesture Recognition service and a clock tick service. Many creative applications can be built using the same concept, with relative ease and without the need to rebuild the same functionality on different applications.

Another important piece to this puzzle is an environment that allows for the simulation of the system, without the hassle and cost involved in actually building it for testing. This problem is solved by Microsoft’s Robotics Developer Studio (RDS) that provides a host of objects that can be simulated virtually. The objects act as they would in real world conditions, with a Physics Engine providing the approximate simulation of a physical environment.

In my project, I combined the Kinect’s audio capabilities and the RDS simulation environment, the DSS framework to interface the above 2 and added CCR into the mix to control the concurrent threads running between the frameworks.

When the project is launched, DSS is executed along with the RDS simulation environment and the Kinect’s speech recognizer. The simulation that is run contains 5 ‘motor bases with drive’, which are basically a set of wheels and a motor. Whenever a recognized voice command is received, the speech recognizer’s handler launches the appropriate method that contains code to perform the action on a robot. The commands include what to do and which robot should do it. The DSS framework interfaces the speech recognition with the simulated robots. The CCR framework takes on the task of making sure that the speech recognizer is running concurrently and accepting inputs in real time without interfering with the graphics of the simulation.

Results

In my project, the simulation that is run contains 5 ‘motor bases with drive’, which are basically a set of wheels and a motor. These 5 ‘entities’ are colored differently from each other to aid in identification. These are controlled through voice and can perform a limited set of tasks together, or independently. They can move in a circle, a square or a triangle in either clock or anti-clockwise direction and can be ordered to stop. Each entity is given its own thread to perform the ordered action, in order to avoid interference and boost performance.

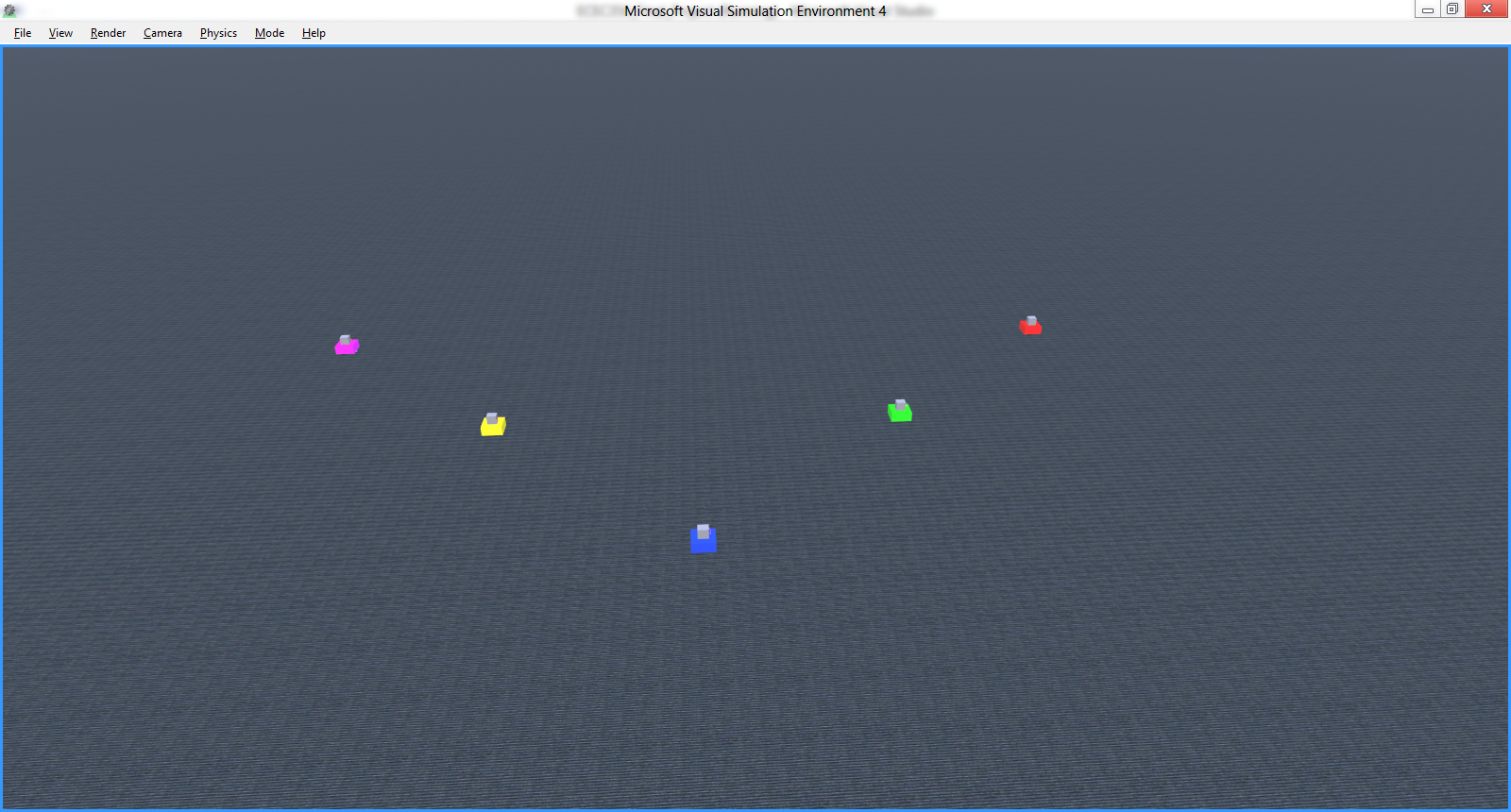


Figure The 5 robot entities. Each robot is colored differently to aid in identification

When the project is launched, DSS is executed along with the RDS simulation environment and the Kinect’s speech recognizer. The building blocks for the simulation environment is located in an XML-based ‘manifest’ file that contains information relating to any and all objects that are placed inside this virtual playground. This manifest was built using the sample EntityUI application and modified using the DSS Manifest Editor. This manifest file contains instruction to build several other objects, which have manifests of their own. A manifest, thus, can be compared to DNA, in that it contains the information required to correctly build the object in question.

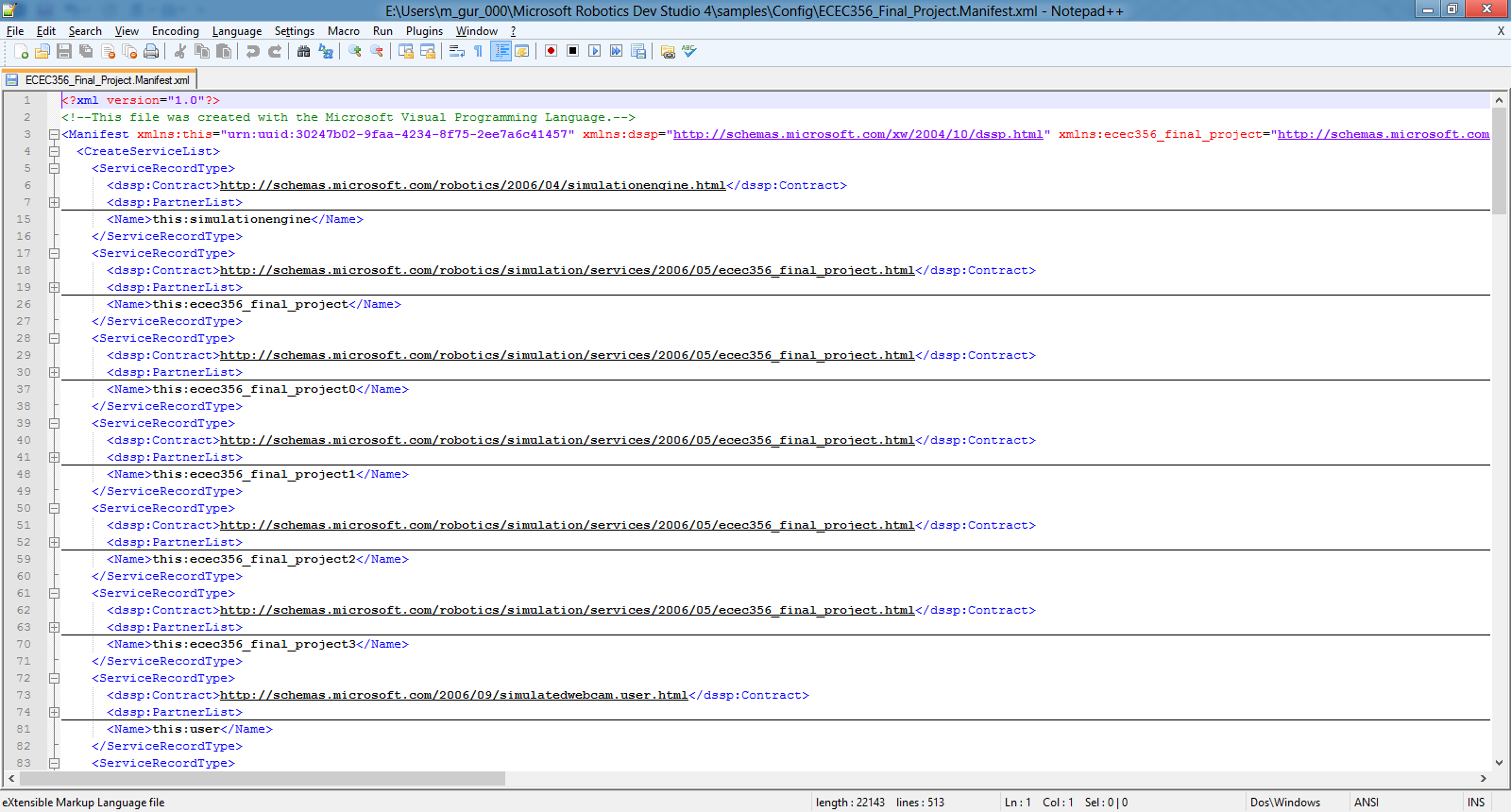


Figure The manifest file

While the manifests are being loaded and the environment is being built, concurrently the audio services are being loaded. In this project, the only physical input provided is voice. This is sent through a speech recognition engine that comes with the Kinect. The speech recognizer is provided a series of multi-word commands that are added to the dictionary, for sentences that it is supposed to recognize and act upon. This is done programmatically. A XAML window is also loaded during this time. This window is updated along with the robots, to display what the current status of each robot is.

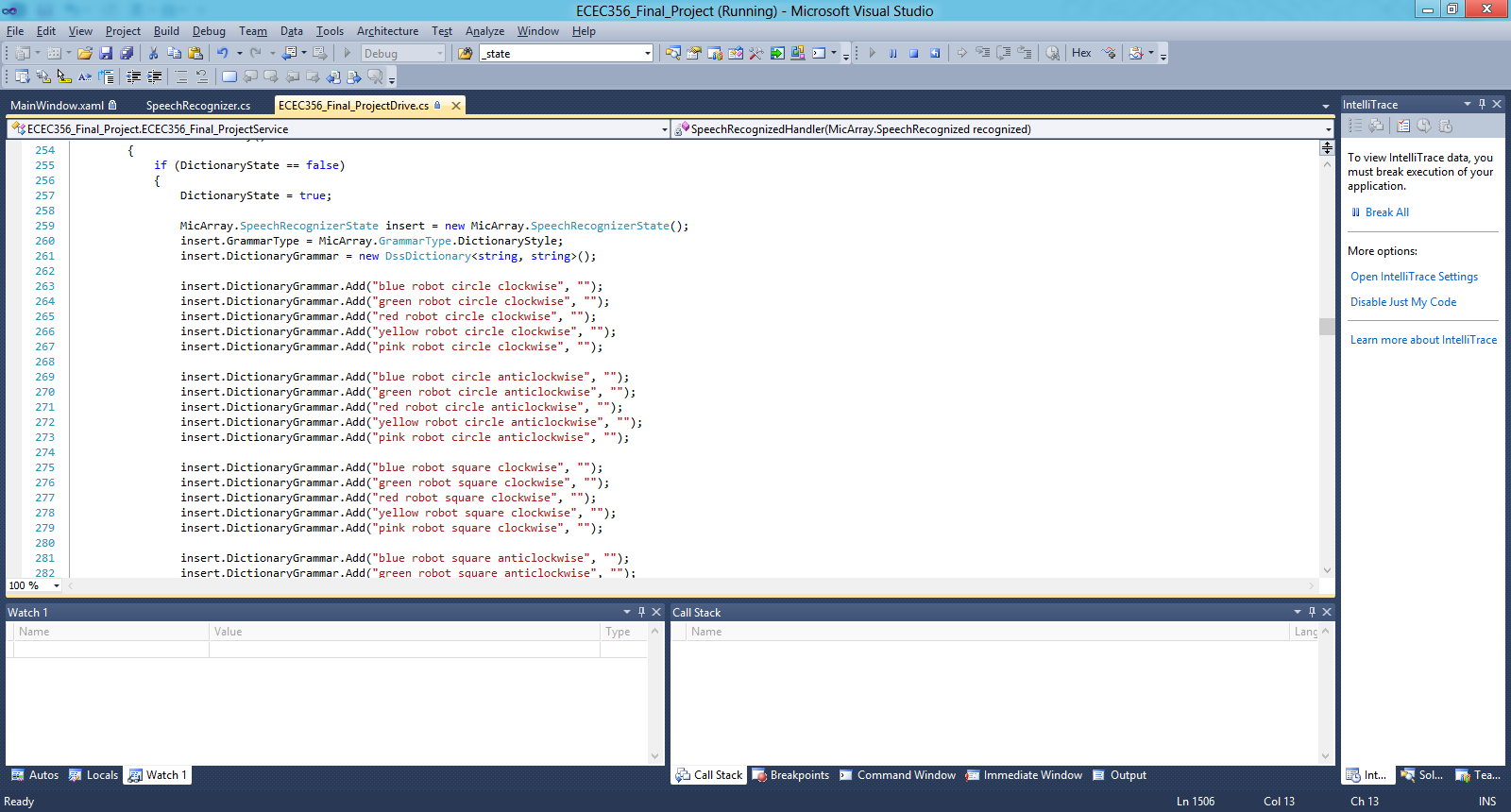


Figure Some of the available commands

Each of these services is given its own port, through which they will accept input and provide outputs about their current states.

I used a sample project called SimulatedDifferentialDrive from the RDS folder in order to make my project. This particular application was capable of controlling a single motor base with drive. However, I needed to provide a way to control multiple such entities. In order to do that, I leveraged the simulation environment’s DifferentialDriveEntity class.

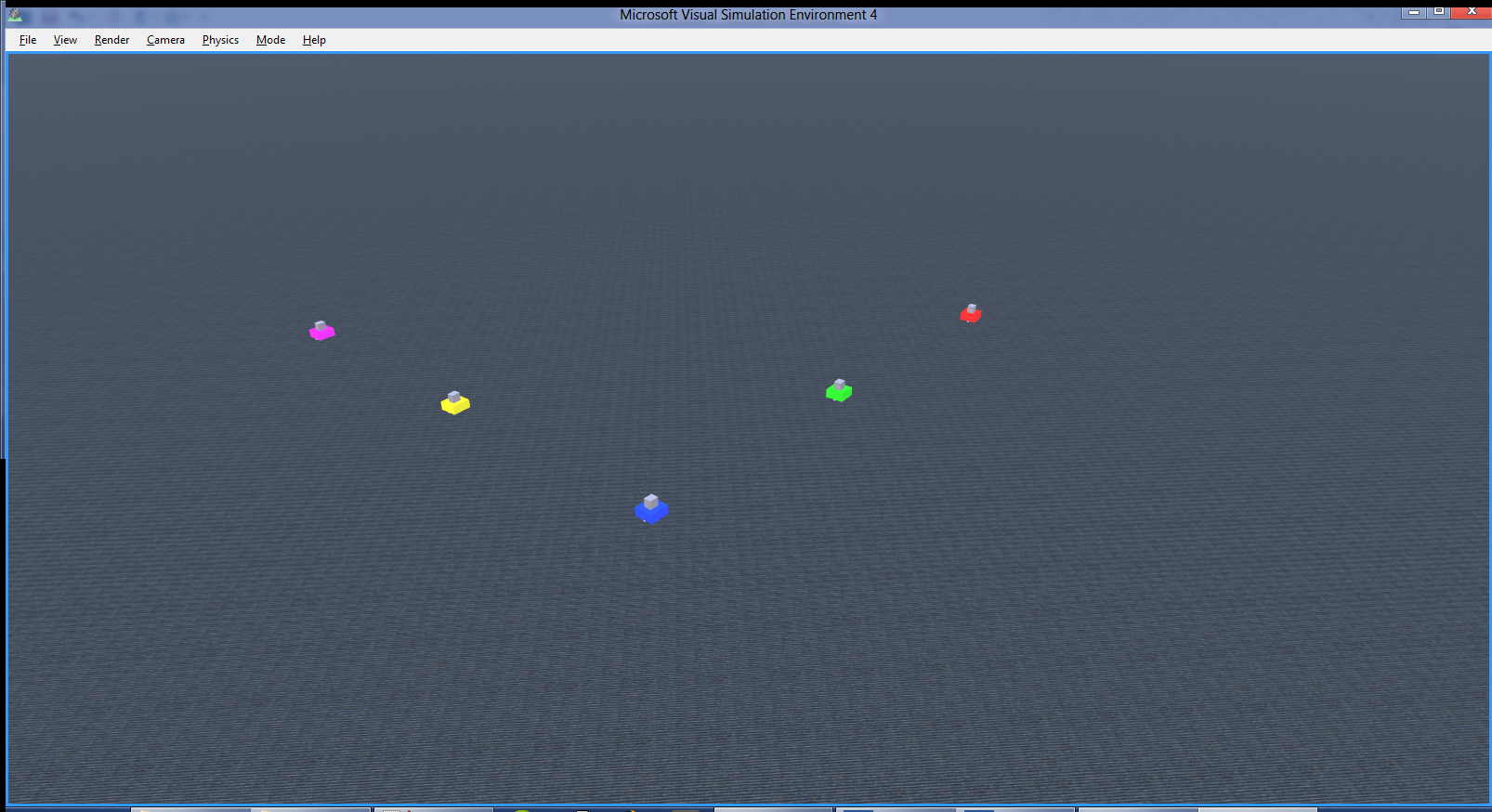


Figure All robots circling clockwise

When the SimulatedDifferentialDrive project is launched and the simulation environment is built, the single DifferentialDriveEntity is stored in a variable called \_entity. This variable stores the object and through this, many actions can be performed on this entity. However, through trial and error I found out that the method that fills the \_entity variable is called once for every motor base that is found in the environment. Thus, for 5 robots, this method was called 5 times. However, each time the previous \_entity was overwritten with the new one that was found. Therefore, with each call, the earlier \_entity information was being lost. I decided to store each of these entities in a global, static Dictionary (i.e. Hash map). This way, I could call each entity individually as I had stored the object when it was being created.

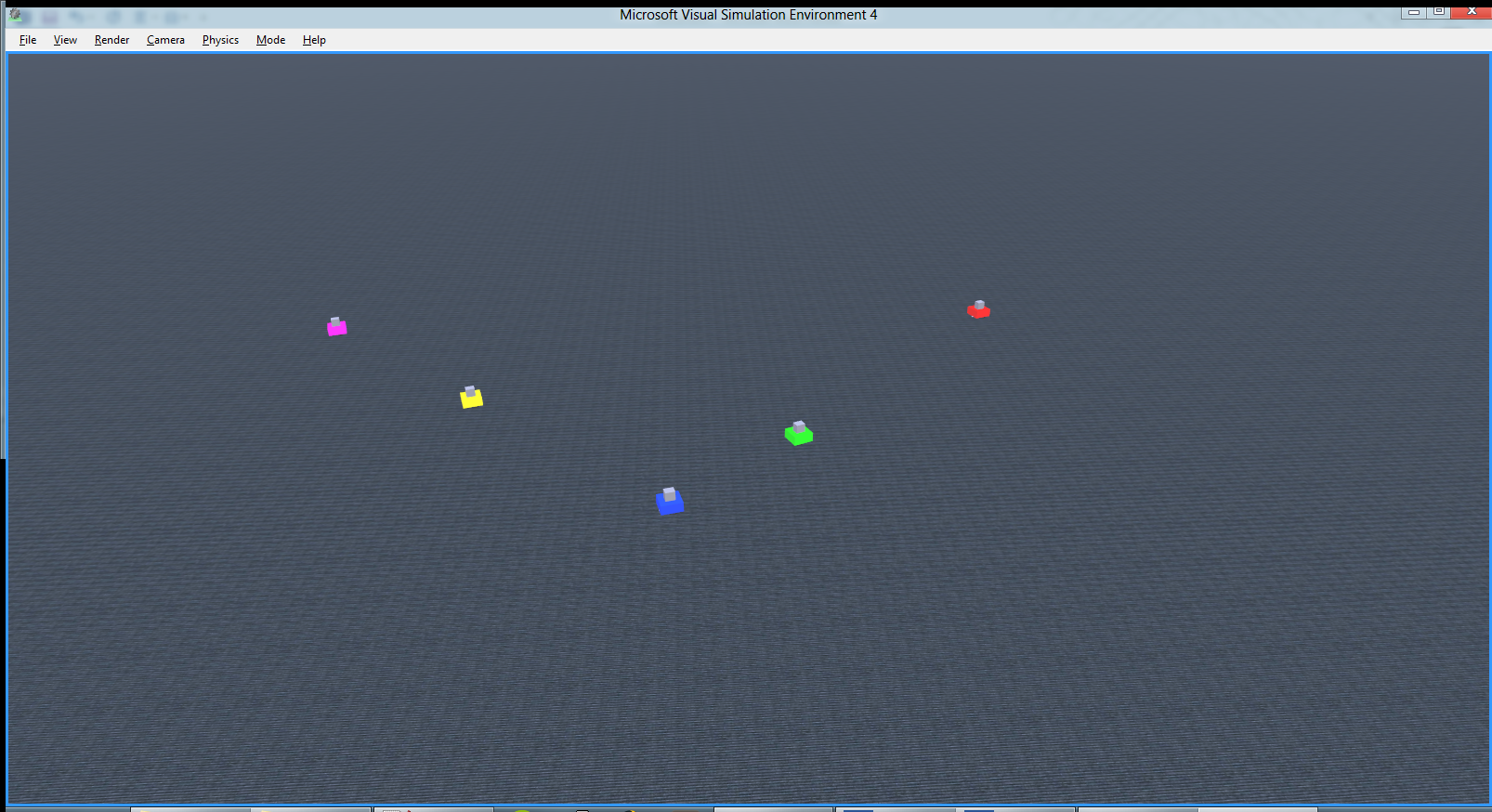


Figure Green robot moving in a square while others are circling

Thus, whenever a recognized command was received by the speech recognizer handlers, I had a way to reference the robot that the command was to go to. For example, “blue robot circle clockwise” would send a command to the blue-colored robot to start moving in a circle in a clockwise direction. Similarly, I could call the red, green, yellow or pink robots as well. As mentioned before, whenever the status of any robot changes, the XAML window updates to reflect this.

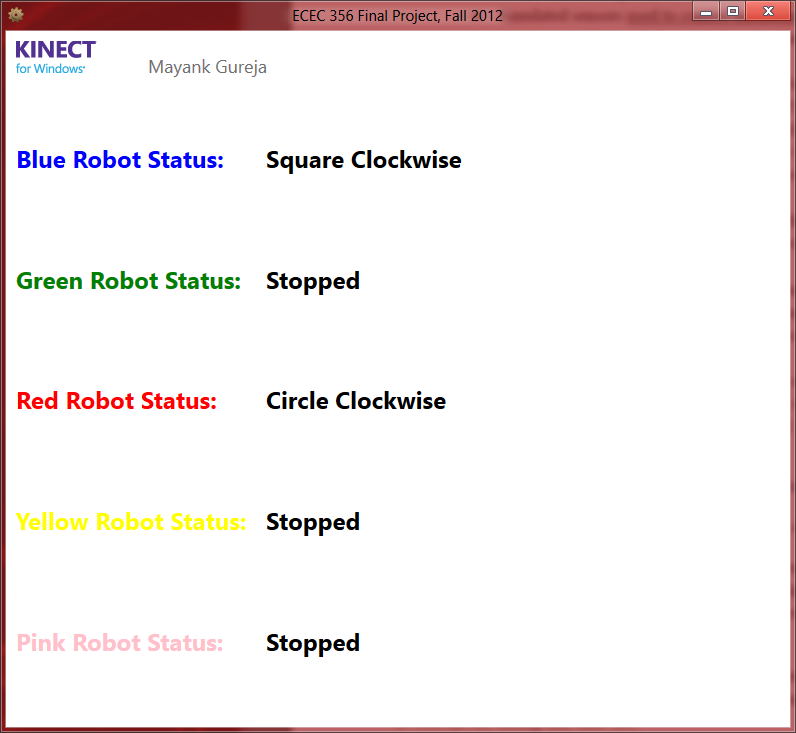


Figure AML window that reflects the robots’ status

In order for the robots to move as they do, I wrote my own methods instead of using the sample’s built-in handlers. This again had to do with the fact that those handlers were meant to control just the 1 robot, but I had many. There is a separate series of commands for each movement type, the most complicated one being the square. This movement is made precise by the use of timers that are tuned to the motor speed of the robot, so that it does not overshoot or miss a drive command.

Another noteworthy item in this project is the use of threads. Initially, I was having issues with performance. This had to do with the way I had implemented the aforementioned movement methods, where they were not taking full advantage of CCR’s threading capabilities. However, using the CCR way of doing things was causing some issues with the robots confusing commands between each other. Therefore, I built my own set of 5 threads that are exclusively assigned to each robot. This eliminates any commands from being fired by the wrong robot and makes sure that the graphics performance does not degrade due to multiple robots trying to access the same thread.

Conclusions

One of the most important lessons I learnt throughout this course is the dangers of working with newly released software. Although RDS, DSS and CCR are extremely versatile, feature-packed toolsets, the fact that they are so brand new poses a challenge. Whenever you get stuck on a problem, there is a high likelihood that no one has ever faced the same problem before and therefore there is no help to be found. You can either figure it out on your own, or look for a different solution.

When I was trying to mess around with creating entities during Lab 5 on my laptop, I hit a major snag. Whenever I tried to add an entity to the simulation environment, I got this error:

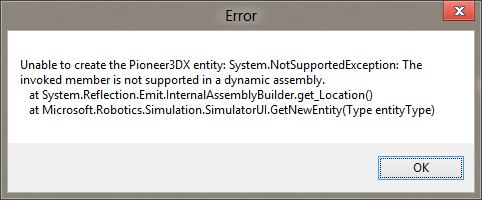


Figure Error received when creating entities in the simulation environment

This is a known issue in the world of RDS and the only solution is to reinstall RDS. I tried to do a repair of the installation. When that failed, I did a full reinstall, but that didn’t work either. In the end, I went to one of the lab computers and created the manifest there. Then I copied it over to my computer. I spent a good hour and a half troubleshooting a problem that I was unable to resolve in the end.

Another known problem relates to the use of both the Speech Recognizer and the Swipe Gestures Recognizer services. When both services are partnered with, only one of them launces and the other fails. After spending hours trying to figure why this was happening, I got a hint from a friend in the class who said that this had something to do with the order in which the services launched. Sometimes, the services are launched together and due to some block on the backend, only one of them is able to get through. A workaround to this problem was to add a wait, and hope that one of them would launch after a few seconds of the other. I believe this problem was faced by many students and no solution has been found. What is peculiar, though, is that some people never had this issue at all, while using almost identical logic.

One of the many challenges in this project was related to the fact that the sample code I used was not meant to run multiple robots. As described before, for every robot entity that was initialized, the handlers were called each time. This caused an issue with the speech recognizer, where even its handlers were being called 5 times. This slowed down the speech recognizer considerably, as it was trying to process 5 of the same messages concurrently. We only need 1 speech handler per message type. To work around this, I had to create some global, static Boolean variables that would make sure the speech handlers were initialized only once.

There was also an issue I faced with the Speech Recognizer, where this error would pop up at random intervals:

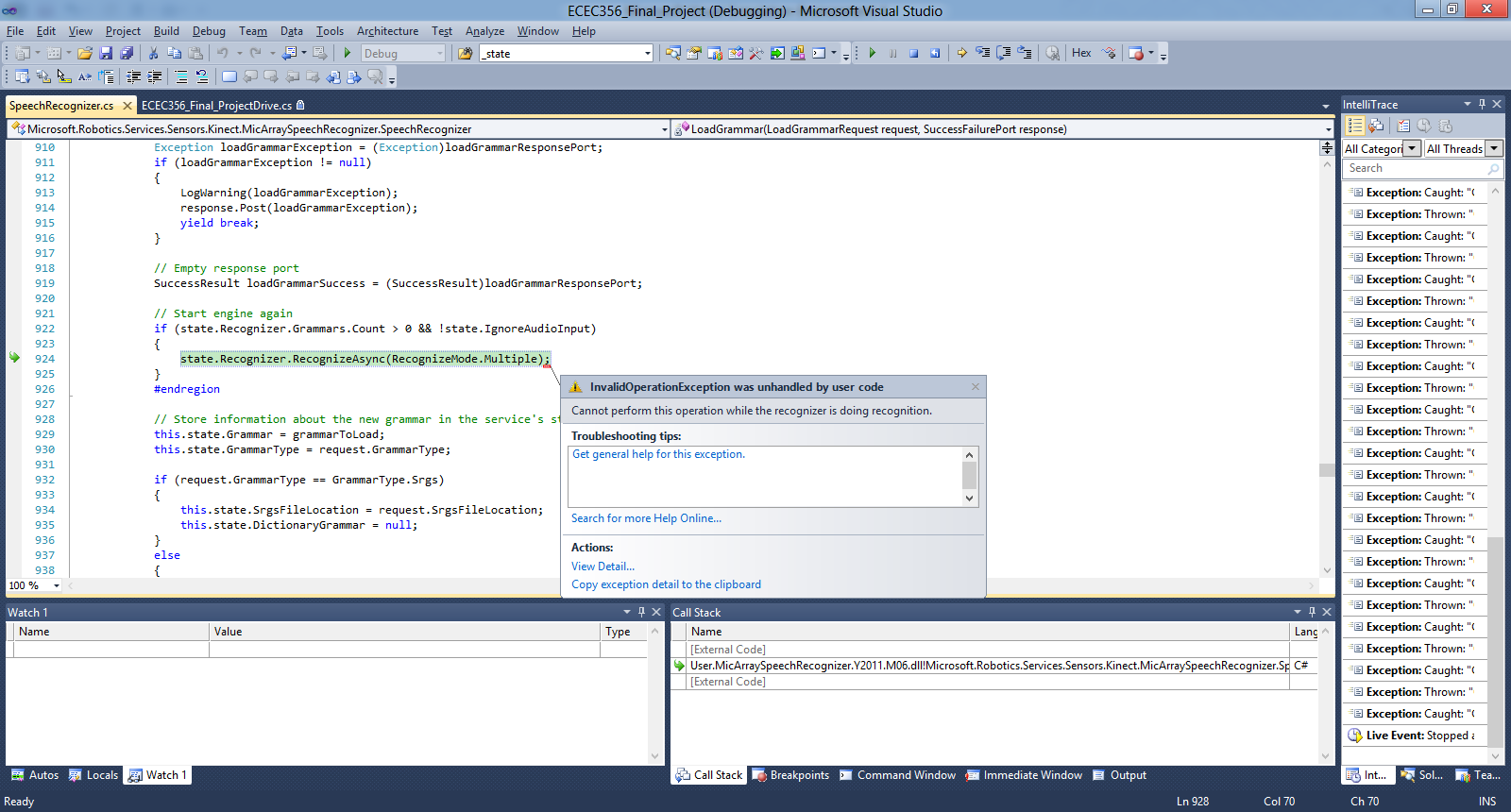


Figure Random error using Kinect's Speech Recognizer

I never found out the reason for this error, and it was intermittent. There were a few articles on this specific error but none of them helped me sort out the problem. The only thing I can assume is that this is once again related to timing and that something is being launched by a thread before the rest of the code is in place. Also, this could be specific to my case because the speech recognizer was being launched multiple times (even though I was trying to prevent that).

While working on making multiple robots work independently, I realized that the solution I devised using DifferentialDriveEntities had one major flaw: it didn’t have access to the DriveStage of the motors. This DriveStage is used to find out whether the robot is in motion or not, and if it has processed any previous requests or not. When commanding the robot to drive or rotate, a message handler is called. This approach has a problem: when multiple requests are made sequentially, only the latest one ends up being executed. This is because the handlers don’t wait for the previous commands to complete. They simply overwrite the previous requests with the latest one. So, if I wanted to move the robot in a square, I would send multiple drive and rotate requests. However, only the last request would actually end up being performed because the previous ones would get superseded before they had finished. DriveStage helps with this, because you can pause for the DriveStage to hit “complete” before you send the next command. My workaround to the lack of a DriveStage was to use timers instead, which is a crude but working solution. I realized later on that I could have accessed the DriveStage by saving the \_state of the differential drives the same way I did for entities. For each entity, I could have saved the state variable and used that. However, I got to that epiphany too late and did not get time to implement it.

Whilst working on the project, at one point I had a different idea for what I wanted to do. My idea was to build a race track (straight line) and have multiple robots race against each other. Their speeds would be randomly reduced/increased and they would have collision avoidance and simulated bumpers so that they did not crash into the side of the race track’s walls or into each other. This would’ve used all the different components that RDS provides. However, I got to the idea a little too late. I had been working on the project for a while and had faced so many issues that I did not have enough time to go back and rethink the whole plan. Also, the fact that I could not create entities on my own computer was a big hurdle as it limited my freedom to make and scrap parts as necessary. I also wanted the ability to change the colors of the robots in real-time as the simulation was running (for eye-candy, nothing more) but I found out that because this is a simulation that is not a possibility. Obviously, in the real-world, tiny robotic boxes don’t spontaneously change colors. So that makes a whole lot of sense.

One idea that came to me during demo day related to how I was referring to the robot entities. Earlier, I had been using numbers instead of colors (blue robot was robot one). However, during the demonstration someone pointed out that using colors instead would make the entire system more user friendly. It’s an elegant idea that I had not even thought of until that point. I was too busy just getting things to work and the colors had been added very recently to the mix, so I never arrived at that thought.

As for whether I would use this framework again, the answer depends on how far the help files and tutorials get. At this point, there are a good number of tutorials available, but the fact that there are so many bugs in the system diminishes their usefulness. Also, not all tutorials are complete and some require earlier tutorials to be implemented, but that is not mentioned. The whole system needs to be more organized and the various bugs need to be sorted out. Other than that, it is a rather fun and enjoyable toolset to work with. The only part that is a bit of a pain to understand is CCR, but once the initial learning is done, it proves to be quite useful.

References

[1] “BigDog – The Most Advanced Rough-Terrain Robot on Earth” Internet: <http://www.bostondynamics.com/robot_bigdog.html> [Dec. 12, 2012]

[2] “A Swarm of Nano Quadrotors” Internet: <http://www.youtube.com/watch?v=YQIMGV5vtd4> [Dec. 12, 2012]

[3] D. Garcés. “Konstantin Kosteniuk’s ChessKA is 2012 World Chess Robot Champion!” Internet: <http://chess-king.com/konstantin-kosteniuks-chesska-is-2012-world-robot-champion> [Oct. 14, 2012]

[4] “Kinect for Windows SDK” Internet: <http://msdn.microsoft.com/en-us/library/hh855347.aspx> [Oct. 23, 2012]