Simulated Locomotion with VIPLE and Unity Game Engine

Collin Christensen

Thesis Director: Dr. Yinong Chen

Second Committee Member: Dr. Yoshihiro Kobayashi

Barrett, the Honors College
Arizona State University

Acknowledgements

I would like to thank Dr. Yinong Chen and Dr. Yoshihiro Kobayashi for providing advice and guidance for this project. Additionally, I would like to thank Dr. Yinong Chen and Gennaro de Luca for their work on ASU VIPLE.

Table of Contents

Abstract

The instruction of students in computer science concepts can be enhanced by creating programmable simulations and games. ASU VIPLE, which is a framework used to control simulations, robots, and for IoT applications, can be used as an educational tool. Further, the Unity engine allows the creation of 2D and 3D games. The development of basic minigames in Unity can provide simulations for students to program. One can run the Unity minigame and their corresponding VIPLE script to control them over a network connection as well as locally. The minigames conform to the robot output and robot input interfaces supported by VIPLE. With this goal in mind, a snake game, a space shooter game, and a runner game have been created as Unity simulations, which can be controlled by scripts made using VIPLE. These games represent simulated environments that, with movement output and sensor input, students can program simply and externally from VIPLE to help learn robotics and computer science principles.

Introduction

Students learning computer science principles for the first time can find it daunting. Early classes in the computer science program, through a student's educational career can focus on robotics. An engaging way to learn about robotics-related frameworks and event-driven workflows, can be to use them for programming games and simulations. Creating new simulators for VIPLE in the Unity engine, and integrating these with VIPLE can be an interesting and educational task. This leads us to the creation of three simple minigames, programmable with VIPLE to introduce beginners to computer science concepts as well as a robotics framework,

ASU VIPLE is a Visual IoT/Robotics Programming Language Environment developed at Arizona State University, in the IoT and Robotics Education Laboratory, directed by Dr. Yinong Chen. VIPLE, which provides a visual workflow for constructing event-driven programs, is used to remote control robots and simulations through web services, WiFi, Bluetooth and USB connections.[1] It can also be hosted locally and used to control a simulation. VIPLE has significant pedagogical uses -- "VIPLE supports the integration of engineering design process, workflow, fundamental programming concepts, control flow, parallel computing, event-driven programming, and service-oriented computing seamlessly into a wide range of curricula."[2] Further, "VIPLE provides an environment with which instructors can teach students many of the fundamentals of computing and programming."[3]

Using VIPLE, one can drag and drop activities and services including Basic Activities (fundamental programming logic), General Purpose Services (Customized activities or keyboard input events), Robot/IoT services (such as Robot controllers and Robot/IoT sensors with preset interfaces), and Lego services (which have not been used in this project, but allow connection

[1] Chen and De Luca, *Introduction to ASU VPL*, Arizona State University.
[2] *Luca, Gennaro De, et al.* "Visual Programming Language Environment."
[3] Chen et al., *Introduction to ASU VPL*, ASU.

with EV3). According to its official documentation, “ASU VIPLE supports an open interface to other robot platforms. Any robot that follows the same interface and can interpret the commands from ASU VIPLE program can work with ASU VIPLE”[4]. Indeed this interfaces was used in development of the games.

Unity 3D is a cross-platform game engine developed by Unity Technologies. It provides a real-time platform with a physics engine and a UI to support development workflows[5]. Unity can also be used to create 2D games and simulations, which it is used for in this project. I have used Unity to create 3 scenes, each with their own set of scripts and objects that makes them a minigame. The Unity game engine allows one to structure a game in terms of a scene, a hierarchy of game objects within that scene that have attached components, such as C# scripts interacting with each other, and prefabs or reusable assets that can be used in multiple scenes. I have used the Unity engine in this project due to previous familiarity with the platform and due to the way that it allows for steady development.

The three Unity games are simplified environments that can be programmed from VIPLE, with the player of each game being the robot in the simulation. They can receive Robot/IoT commands from VIPLE using the Robot Input supported by VIPLE. The snake mini-game, additionally, outputs sensor readings from the snake (player) which is printed in VIPLE. The decision to make a snake game, a space side scroller, and a runner game was because these three basic game types would facilitate a manageable simulation in terms of input (sensors) and output (a limited range of movement).

Methodology

Three minigame simulations are created in Unity with common features: a snake game simulation, a space side-scroller, and an infinite runner. Each simulation has a player in an environment with a limited number of movements, a network script that allows it to communicate with VIPLE using TCP packets and a TCP listener, and some sort of goal or hazard for the player to navigate to or away from. Manual control is implemented for each minigame. While the snake game outputs sensor distance readings to VIPLE, further iteration is required for sensor based navigation.

Project Structure

The Unity project consists of three scenes, the snake game (SnakeGame.unity), the space shooter (SpaceInvaders.unity), and the infinite runner (RunnerGame.unity). Each scene has a number of game objects in the hierarchy, and C# scripts, prefabs (pre-configured game objects), and materials that are unique to each minigame. There are three VIPLE scripts for manual control of the minigames through VIPLE key press and release events.

[4] Chen et al., *Introduction to ASU VPL*, Arizona State University.
[5] https://unity3d.com/unity.

Snake Game

The Snake Game simulation is based on a relatively common genre of 2D games where a snake navigates through a 2D environment chasing apples. For each apple the snake eats, the snake grows in length by one tile, so the player must also avoid their own body, as well as the walls of the arena. In order to create a programmable simulation more suitable for VIPLE and event driven behavior, this version is modified so that the snake has only two possible actions: turning left by 90 degrees and turning right by 90 degrees. The snake has three sensors, front, left, and right, and the player must decide how to navigate based on these three distances.

The Snake Mover script handles the movement of the snake, optional direct input, and also creates and updates a list of tail blocks to simulate the snake's ever growing tail. The snake moves one tile at a time by invoking a repeated move function. The Food Spawn script creates apples or food items up to a maximum, and similarly, continually invokes a spawning function.

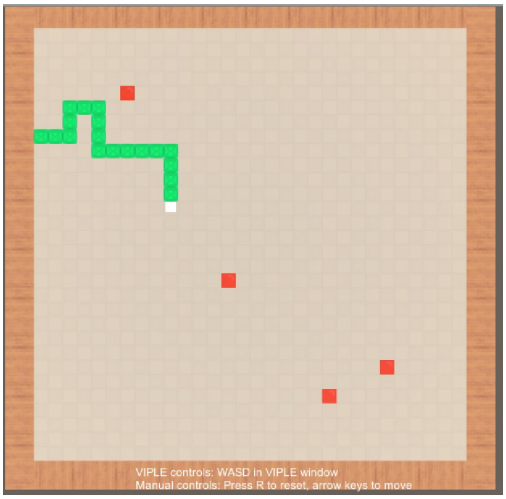

Fig. 1: Snake Game. The snake game is 3D, but it is viewed from above with an orthographic camera.

The snake moves forward at a constant speed, and cannot turn backward or speed up, so the VIPLE script is straightforward and only required detection of key-press events. Manual control of the snake from VIPLE is done through key-press events for the keys A and D, which cause the robot to use the Robot/IoT services "Turn by Degrees" for -90 degrees or 90 degrees to turn left, or right, respectively.

VIPLE documentation defines a specific ROBOT OUTPUT interface which is used in the network controller to report the snake's distance sensors.[6]

name: string (touch, distance, sound, light, color, motorEncoder)
id: int
value: For touch sensor, value will be an int (0 = not pressed and 1 = pressed).
For other sensors, value will be a double

{"sensors": [
{"name":"touch", "id":0, "value":0},
{"name": "distance", "id":1, "value":12.8}
]}

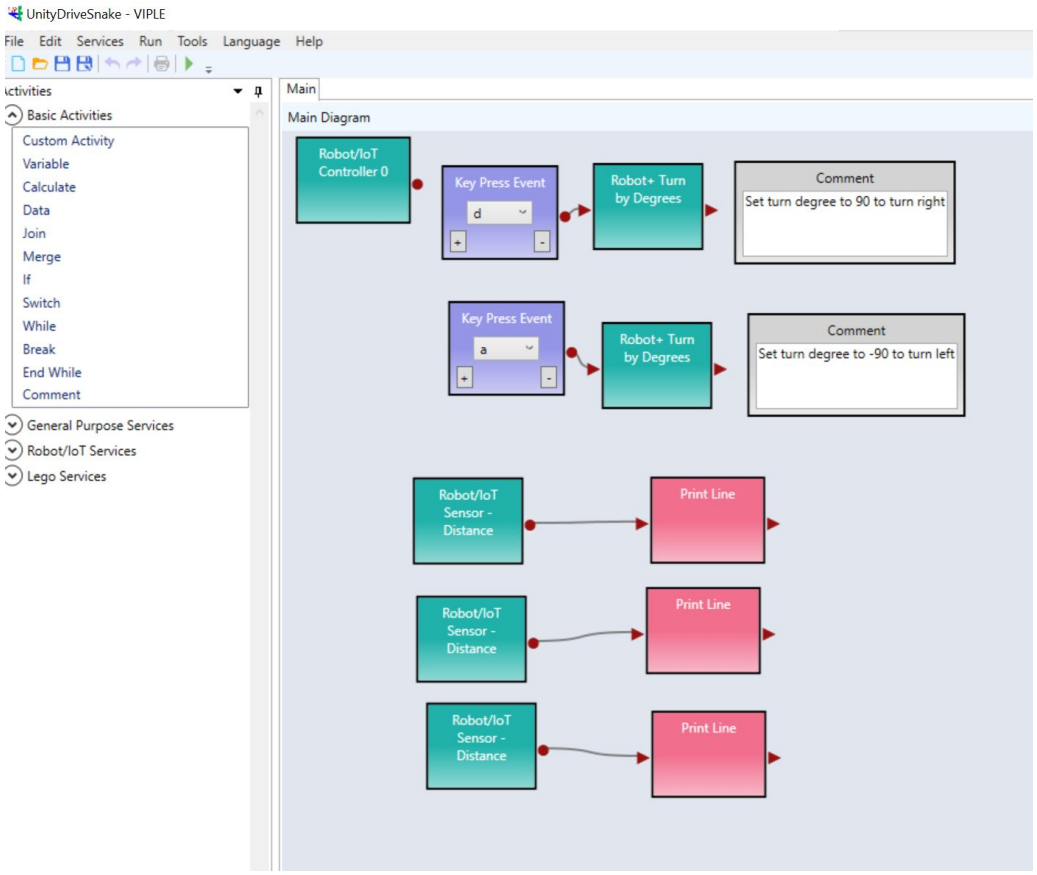

Fig. 2: Manual control script for the Snake game.

When run in the Unity editor, the snake game reports the distances detected by its front sensor in the console. These distances are sent to the VIPLE script through a callback, using the robot output interface. For manual control, the VIPLE script sends a JSON message with

[6] Chen et al., *Introduction to ASU VPL*, ASU.

information for each servo. The VIPLE Network Controller script in Unity uses a TCP listener to receive the message. It parses the JSON and uses the “isTurn” boolean, as well as “servoId” (whether or not the speed value is a turn), and “servoSpeed” (the number of degrees to turn). In this simulation, the only input events possible are turns, so this allows it to determine whether the action is a perpendicular left or right turn.

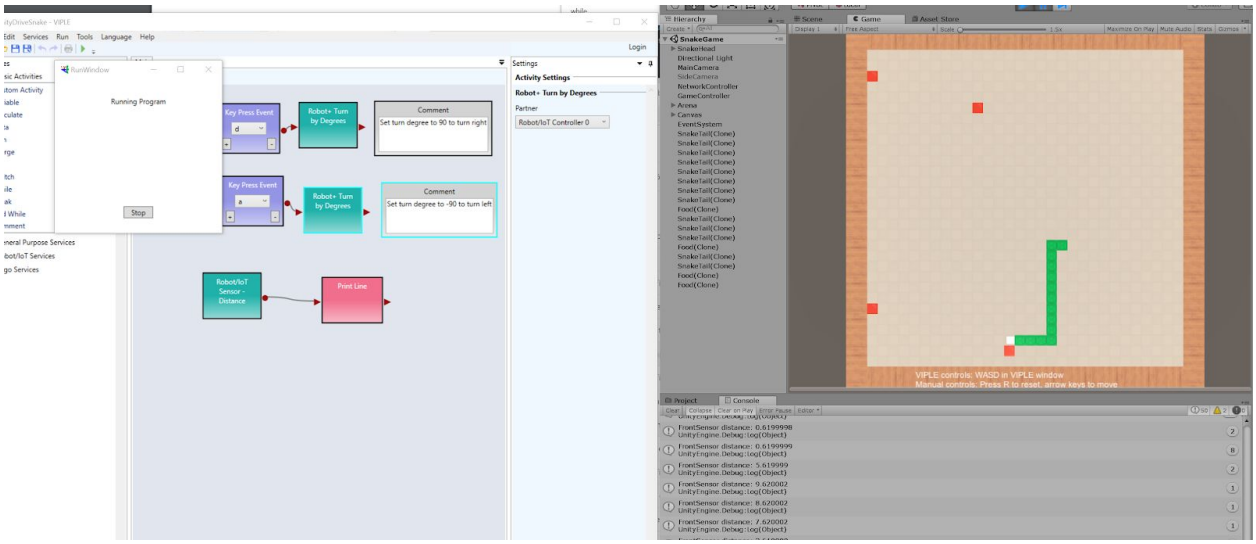

Fig. 3: Snake game control with VIPLE script

Space Side-Scroller

The second simulation is a space side-scroller or space shooter where the player must avoid oncoming space debris. The game was inspired by the “Space Shooter” tutorial[7] on the official Unity website. There are three actions possible in this minigame -- shifting to the left, to the right, or shooting the laser. This presents a relatively different challenge for VIPLE scripting in terms of its pedagogical uses. Movement is more simple and no real pathfinding is needed but the laser and oncoming debris also presents a new hazard.

In the Unity implementation, the player’s spaceship can only drift so far before being stopped by the edge of the game world. Meanwhile, spawn waves of hazards are generated continually by the SpaceGameController. Hazards are spawned up to a maximum, and they are either destroyed by contact with the player, with the player’s laser, or by exiting the game area determined by collisions with the plane representing the playable area.

[7] *“Space Shooter Tutorial.” Unity.*

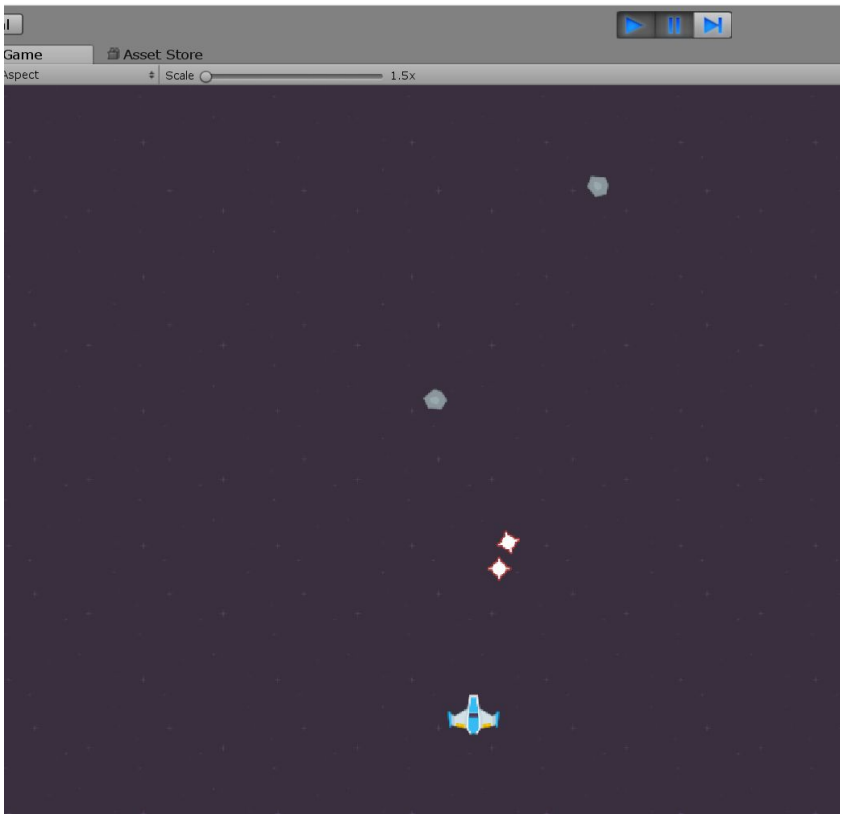

Fig. 4: Space game in play mode in Unity

On the VIPLE side, the space shooter manual control script is implemented and sensor readout remains for further iteration. Input is done by key press and release events, to allow for continuous movement rather than the tile-based blocky movement of the snake game. The VIPLE program uses key press and release to determine a boolean flag whether to go left or right, which then sends JSON message of turn by degrees to the Unity simulator. A while loop is used to maintain the movement as long as the player presses the key. On the other hand, firing the laser is a single press and fire rate is limited in the Unity simulator itself by the SpacePlayerController.

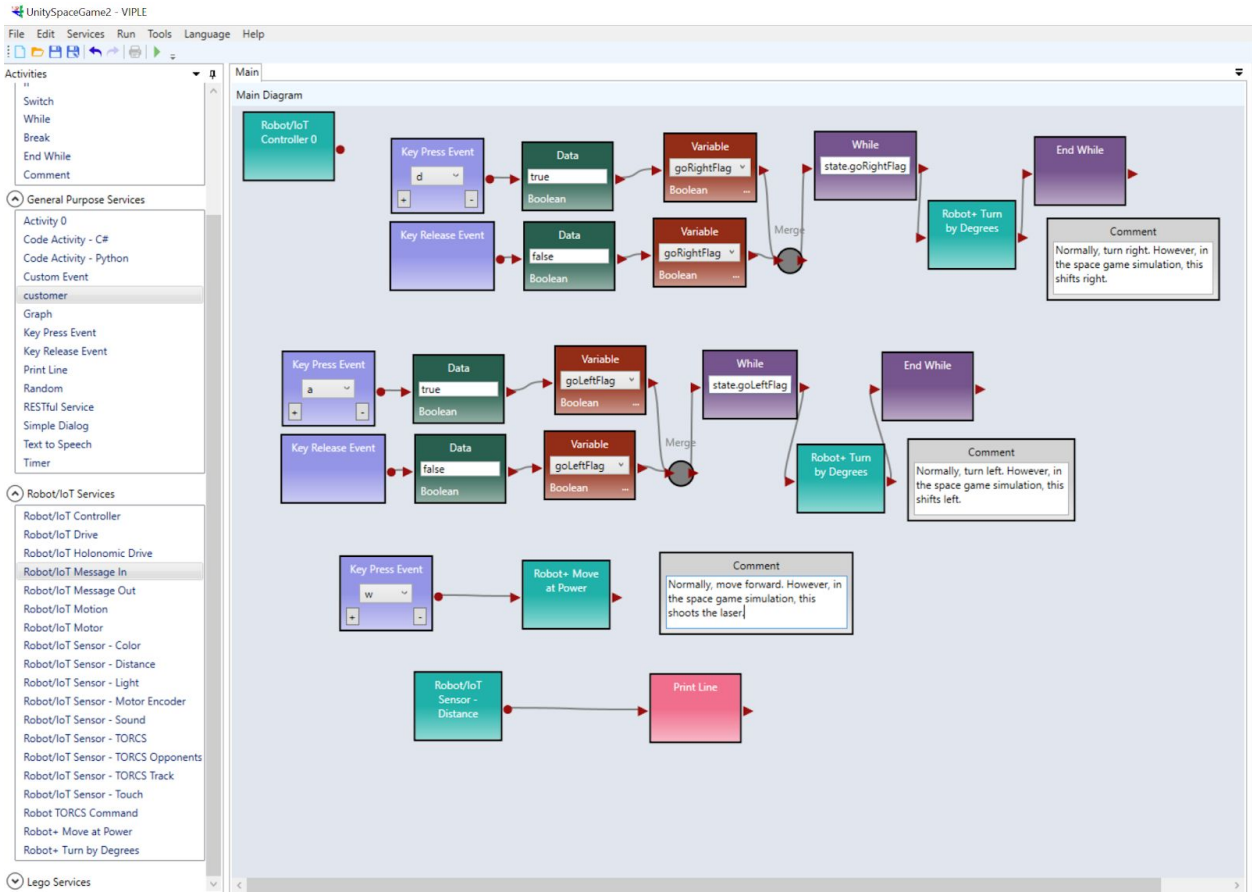

Fig. 5: Space game manual control script in VIPLE

Infinite Runner Game

The runner game presents another simple minigame. The infinite runner genre generally involves a player being able to only jump to avoid obstacles, and the simulation is developed with this in mind. With only one control this makes for a straightforward script in VIPLE. This would be the first game that students are introduced to understand how to respond to key press events and read the distance sensor. The runner game involves more of Unity built-in physics engine by making the player and the continuously spawning obstacles into rigidbodies and applying velocity to move them.

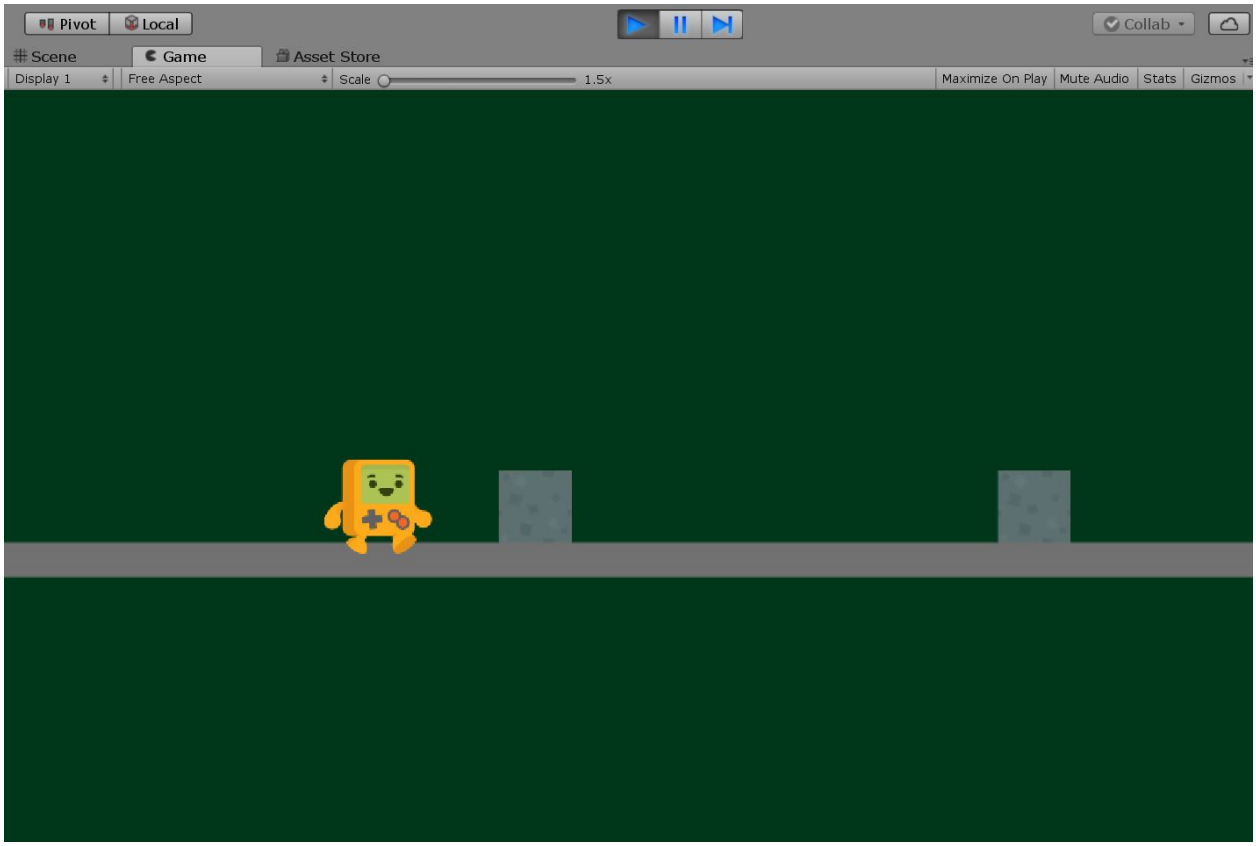

Fig. 6: Runner game in Unity

The manual control script involves only one key press event leading to a move at power command, which is read in Unity by the network script (VIPLENetworkController.cs) to cause the player to jump. Similarly to the other network scripts, it listens to TCP messages parses JSON and reads the servoId and servoSpeed. Only one movement is possible in this minigame, and it is all about timing for the player to avoid the obstacles.

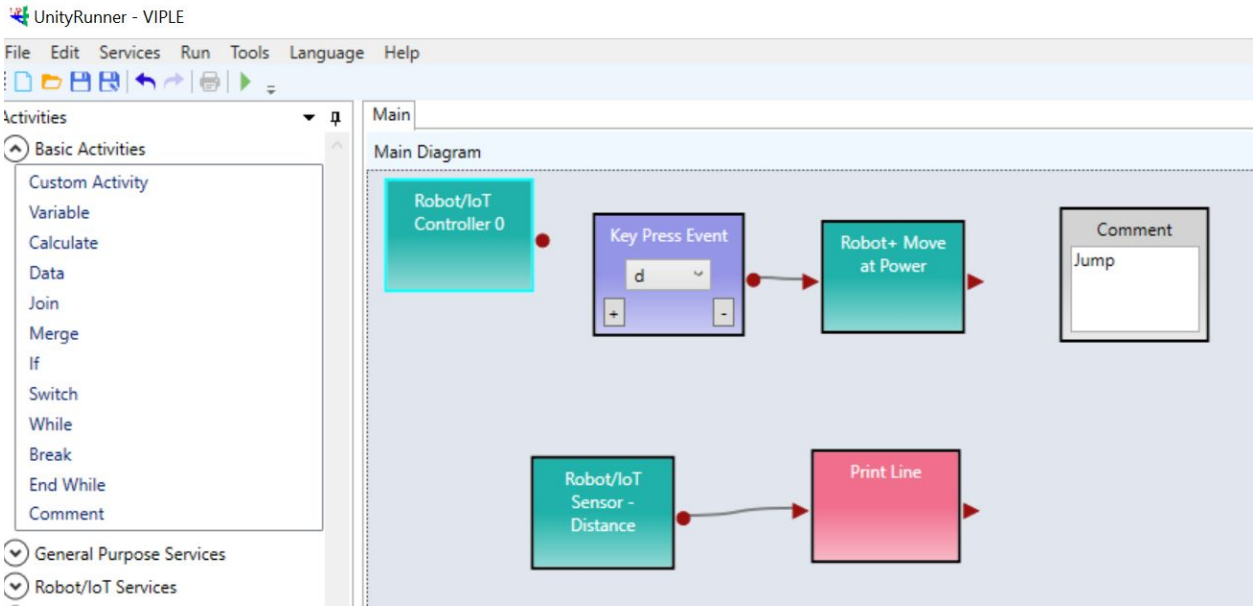

Fig. 7: Manual control script for the Runner game

Shared Network Controller

Each game shares the same VIPLE network controller script, based on a robot controller provided by Dr. Yinong Chen. The Newtonsoft JSON framework is used to read the robot input and format the robot output according to the interface used by ASU VIPLE. The network controller finds the player of any given minigame, opens a TCP connection with the VIPLE program. JSON input of “isTurn”, “servoId”, and “servoSpeed” is parsed to determine which Robot/IoT action to perform (moving at power vs. turning by degrees). JSON output writes name, id, and value for sensors to send the sensor readout to VIPLE, which is thus far used in the snake game with distance sensors in the other two simulations having room for iteration.

Unity Build

The games are bundled into one app, in which one can switch between the three games by use of keyboard commands. The app is a release for Windows, Mac, and Linux, using Unity’s default build settings except for several configuration changes in regards to input.

Analysis

The creative project was a useful learning experience and the three minigames can provide an interesting programmable simulation in VIPLE. The snake game and space shooter seem the most promising in terms of providing a dynamic challenge for event-driven programming, while the runner game seems like it could become an effective introduction and start off the course of learning with something basic but with didactic benefits. More iteration remains for the Unity simulations so that sensor-driven navigation can be done on the VIPLE

side beyond the implemented manual control. Over the course of development, the code has been more decoupled and the same network controller was used for all scenes, although further decoupling and modularity may be desired. Since VIPLE's "strength is in event-driven programming and [it] can respond to a sequence of events",[8] responding to input can be a good introduction to this platform and key goals were met.

Conclusion

I developed three mini-games in Unity, a Snake game, a side-scrolling space shooter, and an Infinite Runner with the intent that students could program the simulations in VIPLE. Each Unity simulator can be manually controlled using event-driven keyboard input both in Unity and in VIPLE. Further iteration is necessary to establish fully programmable simulation and expose the sensor data in Unity to the VIPLE programs. The games as they are implemented provide examples of simulated movement through gameplay, which can be useful for learning a robotics framework such as ASU VIPLE, and the event-driven programming mindset required for handling user input and sensors in robotics.

Limitations and Future Work

There are a number of limitations, and some future work can be recommended. With the goal of simulating locomotion using gaming and Unity, continued development could expose the sensor data to VIPLE allowing for robotic algorithms to be implemented without the human player's view of the environment. The user interface and the graphics of the three games could be tuned and improved. Additional documentation could assist in teaching students concepts such as control flow and parallel threads.

A future direction for expansion could be using machine learning, in particular a neural network and genetic algorithm, to evolve a solution to each of the three minigames, with sensors as inputs to the neural network and movement as output.

[8] *Luca, Gennaro De, et al.* "Visual Programming Language Environment."

Appendix

Repository link

https://bitbucket.org/c96/viple_gaming_unity/

External Assets and Packages used

Free-to-use visual assets, and Unity Asset Store packages, were utilized.
Licenses (public domain) are included in the project files.

- Kenney.NL sprite packs (Animal Pack, Simplified Platformer, Space Shooter). These packs were used to provide visual assets in the mini games.
- Json .Net for Unity 2.0.1, a port of the official Json .Net library (Newtonsoft). This package was used to serialize and parse JSON.

Scripts for the Robot controllers (including TCP listener function) were provided by Dr. Chen

References

Chen, Yinong, and Gennaro De Luca. “VIPLE: Visual IoT/Robotics Programming Language Environment for Computer Science Education.” 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, doi:10.1109/ipdpsw.2016.55.

Chen, Yinong, and Gennaro De Luca. “IoT and Robotics Problem Solving in Visual Programming: Laboratory Manual.”School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, 2016, http://neptune.fulton.ad.asu.edu/VIPLE/IntroductionVIPLE.pdf.

“JSON .NET For Unity.” Asset Store, assetstore.unity.com/packages/tools/input-management/json-net-for-unity-11347.

Luca, Gennaro De, et al. “Visual Programming Language Environment for Different IoT and Robotics Platforms in Computer Science Education.” CAAI Transactions on Intelligence Technology, vol. 3, no. 2, 2018, pp. 119–130., doi:10.1049/trit.2018.0016.

“Products.” Unity, unity3d.com/unity.

“Space Shooter Tutorial.” Unity, unity3d.com/learn/tutorials/s/space-shooter-tutorial.