API Design and Implementation for an Augmented Reality and Robotic Prototyping System

Final Report

Phillip G. Hege
Advisor: Shih-Yuan Liu
Lab: Aerospace Controls Lab (ACL)
Professor: Jonathan How

Introduction

Although the decreasing cost of electronic hardware has made unmanned aerial vehicles an affordable platform for developing autonomous algorithms, there are still many challenges facing this area of research including planning, control, perception and learning. When designing software models that tackles these problems, it is necessary to test the model's performance on live robotic hardware. Performing live tests are a key step in transitioning algorithms from software simulation to real-world systems. These tests are important in understanding how the algorithm performs in different tasks and reacts to various kinds of environmental noise. In general, when running algorithms on physical robotics systems, new challenges that are often unrelated to the primary goals of the investigation present themselves, including issues of hardware failure and noisy environments. When running algorithms on flying robots or quads, these issues are magnified with the added concern of unsafe failure modes. When a ground robot fails,

at worst it collides with a wall or stubs someone's toe. When a quad fails, spinning propellers in the air may potentially cause more damage. With these challenges, the overhead of creating new demonstrations are time consuming obstacles that are often burdensome to researchers.

With an augmented reality simulation, we can begin to bridge the gap between the digital and the physical world thus expediting the development and research of control algorithms. These simulations can facilitate the construction and testing of new environmental configurations and real-world scenarios without actually requiring researchers to construct new environments by hand or physically changing any part of the robot. With augmented reality, performing new algorithm tests on physical hardware in various environmental simulations is only a few keystrokes away.

Project Background

The current augmented reality solution, known as the Measurable Augmented Reality for Prototyping Cyber-Physical Systems (MAR-CPS) [1], allows for real-time visualization of latent state information to aid in hardware prototyping and performance-testing of algorithms. This engine is powerful, but requires substantial software engineering overhead in order to calibrate the system for new environments, thus replacing the original hardware problem with a similarly tedious software problem.

Currently, researchers first must understand the 3D geometry of an object and then encode that using Vispy, a 3rd party visualization framework, in order to create or manipulate any elements on the field. Although the framework is useful, it is a relatively

new resource with incomplete documentation. In some cases, it requires a deeper understanding of graphics processing in order to make basic changes. With these hurdles, it still requires extra work in order to function because it is not customized for ACL. Over time this code becomes redundant yet fragile where changing something small could require a complete re-write. Two projects that suffer from this visualization overhead problem are displaying and manipulating background environment images and manipulating many thousand projected polygons simultaneously. Both require the same visualization engine but are handled in different ways.

System Background

The primary software of interest for researchers at ACL are control algorithms. Control algorithms dictate how a machine will move in its environment. At ACL, these algorithms can change, depending on the challenge that the team must solve. These algorithms are implemented on a machine that runs ROS (Robot Operating System). ROS manages every component for a robot. It allows developers to easily design modular multithreaded systems with the use of topics. A topic is a data stream that passes information between each sub-system simultaneously.

The ROS machine communicates with the quads that are on the field exchanging sensor data and movement commands. ROS is also aware of the position of all the quads through the use of feedback mechanisms like Vicon, the indoor GPS system. Quads have various onboard sensors depending on the requisite task they need to

complete. They can hold detection sensors such as cameras and infrared sensors, in addition to their flight sensors such as accelerometers and gyroscopes.

In a traditional test, obstacles must be constructed in the test area with indicators attached to them so that Vicon could register their location. In a system without Vicon, various techniques of computer vision would be applied to find objects of interest. When the task does not require physical objects, or in the case of controls simulation, we are primarily concerned with how a quad moves. The experiments are typically designed to characterize how the quad performs when certain obstacles are theoretically present or how the quad performs at high velocity. The test is not necessarily meant to characterize how well a quad detects certain obstacles. In these cases, simulating a test environment makes the most sense.

In a simulated test, MAR-CPS can generate the appropriate simulated obstacles in a virtual environment and project images onto a 2D surface to aid in controller development and when giving live demos. The pre-existing implementation was able to load images and create custom shapes using pre-packaged objects provided by Vispy. The simple shapes (disks, rectangles) produced by the visualization library could be layered to make more interesting objects, such as a simulated drone with status indicators for demonstration.

All of the simulated content for display including quads and obstacles are produced on a Windows computer that runs the projector software. This pre-packaged software stitches an image across multiple projectors in order to produce the appropriate image on the lab floor. It “stitches” the edges of the image together and

adjusts the projections brightness to offset the bright spots that appear at the edge of two frames due to two projectors shining onto one spot. Without this mechanism, the researcher would have to generate quad control algorithms as well as adjust for projector stitching.

Quad positions are sent over UDP to the projector computer, which are then translated into Vispy objects and displayed onto the projector screen. The figure below shows an example of a field that would be projected. The areas with shapes and letters superimposed on them show points of interest on this map. Below is an image of the field from the existing implementation from ROS's perspective.

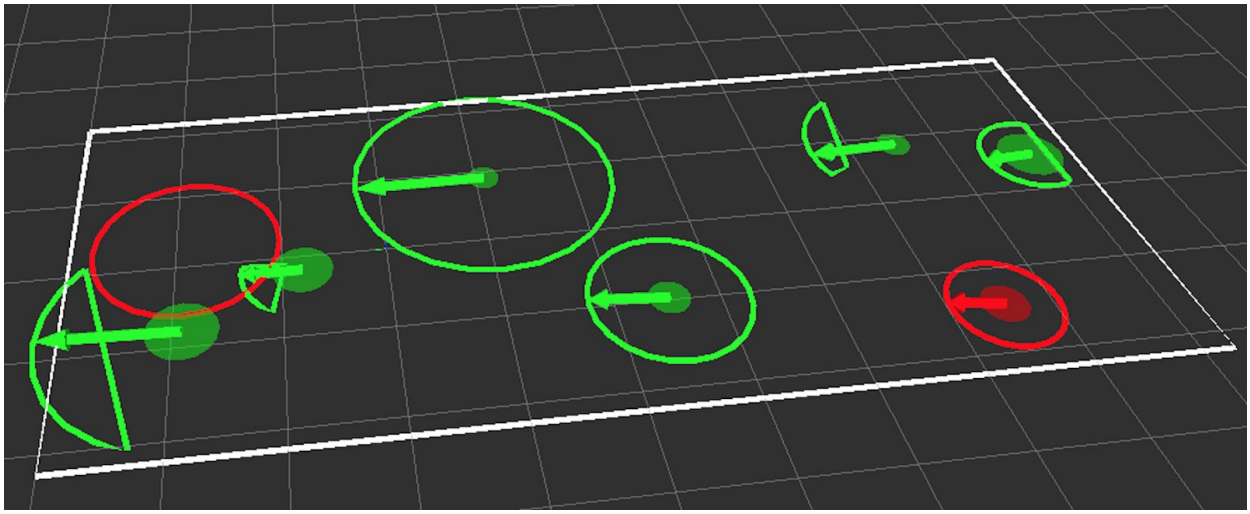

Figure 1: A view of quads using RVIZ that comes pre-packaged with ROS

Purpose

The key goal of this project is to improve the interface between simulation and projections. In the following, I discuss the overarching system design, OpenGL (a 3rd Party Graphics Library) implementation, system integration and my debugging strategy.

System Implementation

The focus of my project was to improve on the software architecture for the MAR-CPS in order to make it more efficient and developer friendly. I created image processing objects that could save images with Vispy and manage their implementation. Instead of digging through incomplete Vispy API documentation, I overloaded convenient transformations, such as rotation and translation, with my own implementations. In addition, I created polygon support to handle thousands of objects of different shapes and sizes to be constructed and manipulated.

The module I designed allows researchers to easily create environments in the simulated world, to control environmental projection equipment and to quickly integrate simulations from the software development workspace into the virtual workspace. I then integrated the newly created module into the existing control flow software. This system integrates with the other modules in the existing control software which is written for ROS.The figure below is an image that is loaded onto the Vispy canvas and displayed onto the projector screen. It is generated using the newly designed object for handling images.

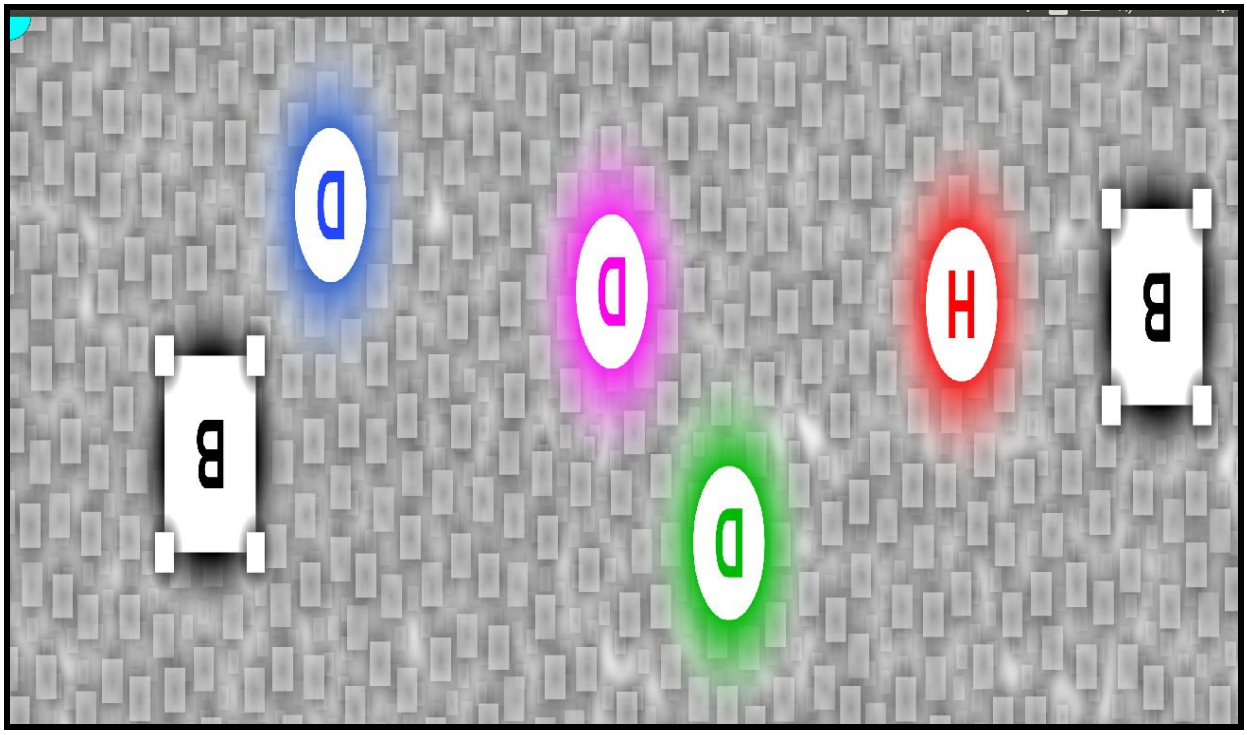

Figure 2: Image loaded into Vispy and what would be displayed by the projector

OpenGL

While MAR-CPS relies on Vispy to display information to the screen, Vispy relies on OpenGL to do the heavy lifting of creating objects. OpenGL is an open source graphics library that handles the math for manipulating geometries using the graphics card on a computer. It is a popular way of designing efficient graphics software. I designed custom objects in OpenGL to create and manipulate multiple shapes on the projector. The program could handle on the order of thousands of unique points with unique colors, shapes and sizes moving at different speeds and directions with seemingly no delay.

Vispy provides an interface for developers to write custom OpenGL code with the use of shaders. A shader is a program designed to run on some stage of a graphics processor. Its purpose is to execute one of the programmable stages of the rendering pipeline. A user describes the behavior and attributes that a shape may have. Following shader protocol involves separating the creation of elements and rendering elements onto a canvas into separate stages. Separating execution into many stages allows for many thousands of points to be created and manipulated simultaneously without slowing down your machine. Building processor logic in this manner allowed me to design an API that exploits the processing speed of OpenGL with the simplicity of Vispy. Below is an image of a shader program running through Vispy to generate disks and move them across the screen.

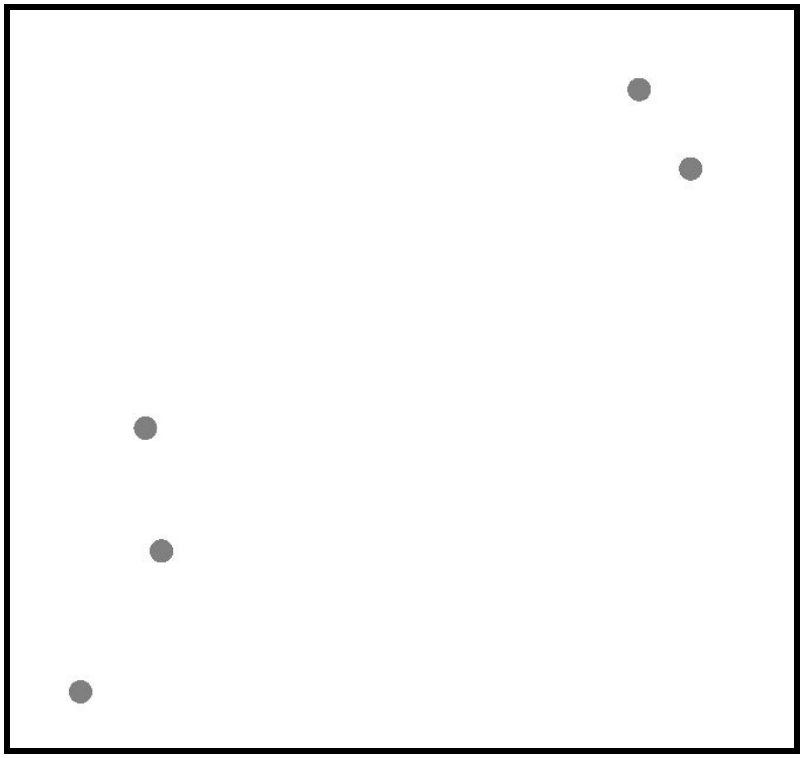

Figure 3: This shader program creates “crowds” of disks and moves them across the screen

With the shader in place, projections can be designed that are similar to the projections displayed on a ROS terminal. Below is an image generated by the projector that relates to the original image from a ROS computer.

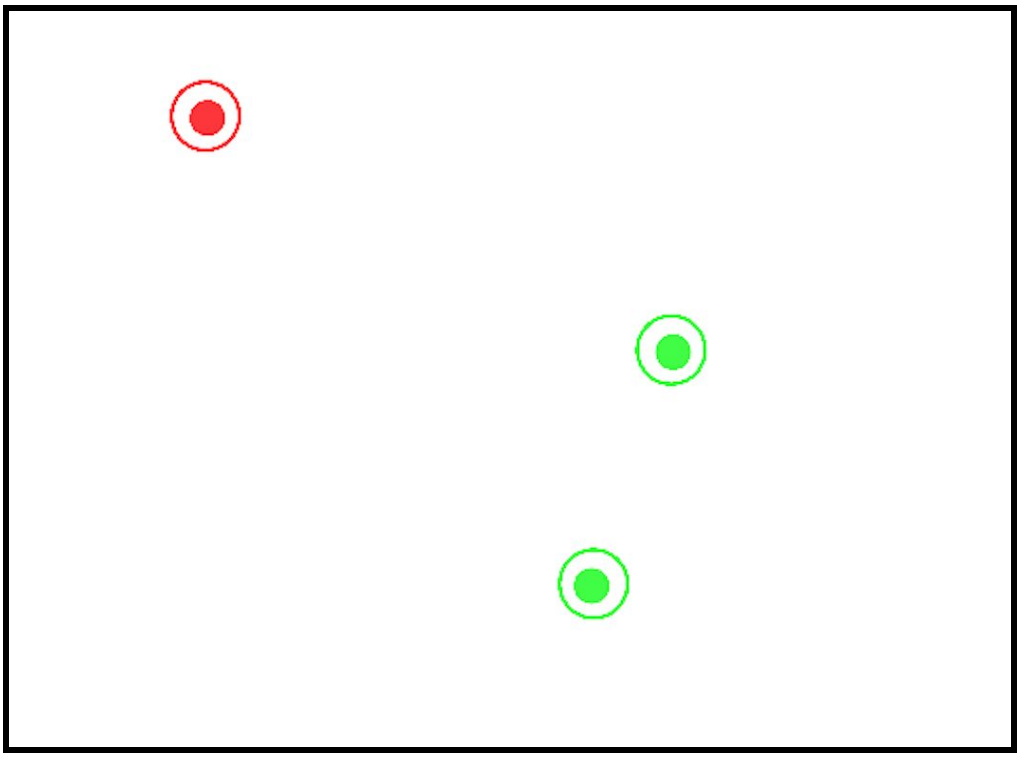

Figure 4: A simulated view of multiple elements running across the screen

Software Integration

Integrating my custom software with the pre-existing solution involved converting data types from ROS nodes into data types that OpenGL can understand. I primarily worked with crowd messages which are made up of position vectors. I added an additional node that reads the crowd messages and opens a UDP connection to send

data to the projector computer. The projector computer opens a UDP connection on the same port and receives data to be displayed on screen. Below is an example of the ROS Node graph that connects all of the different data sources with the areas in software that use them. My component connects to the rvo_vel_controller/crowd message to the ros_to_udp node. In the ros_to_udp node, position messages are converted from ROS to Vispy objects. The point of interest is highlighted in red.

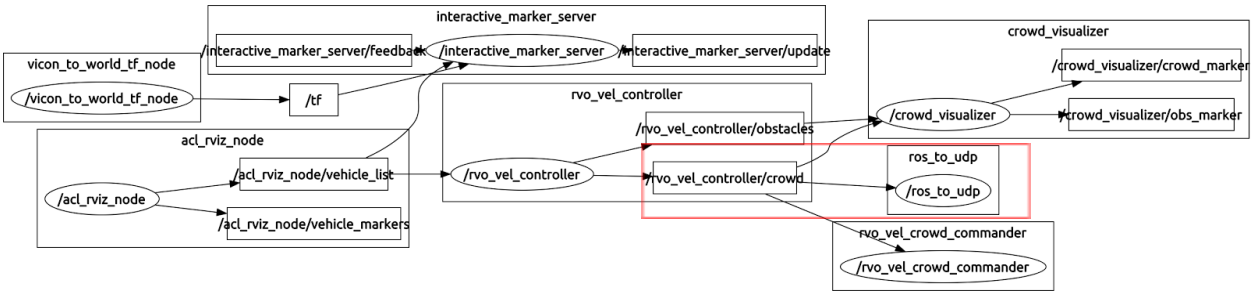

Figure 5: ROS node graph

Testing API & System Debugging

My primary testing strategy was to maintain current code functionality while adding more robustness. With that in mind, I wrote unit tests that load-tested point generation and destruction at various intervals, measuring the time lag in order to ensure that the machine would not slow down considerably while trying to project thousands of elements. I interfaced with ROS and the projector platform using python and wrote my test scripts in python as well.

Contribution

My contribution to the ACL Research team facilitates the development of new interactive demonstrations. In the old implementation, image objects and polygon object creation and manipulation would be an additional headache for researchers, which would have to be handled every time a new simulation was created. With the custom ACL API I designed and implemented, a researcher can now create new elements for display with minimal syntax and without requiring a deeper understanding of OpenGL. This system also gives researchers the confidence that their image manipulations are functioning in a logical object oriented manner. See Appendix for the Image Manager API layout.

Next Steps

My project served as a useful time saving fix for a problem; however it is still not the ideal solution. Creating more accurate simulations increases the ease of development and clarity when presenting information to a group of people. Future upgrades to this system should make existing architecture easier to use or provide new features that allow for a different level of demonstration than what has been capable before. The biggest hurdle in connecting the two systems is the different default architectures are naturally incompatible, because ROS can only run on Ubuntu while the projector software can only run on Windows. Until there is a fix for that, solutions will have to be engineered in order to account for the differences.

Apart from the architectural differences, including more elaborate visualization is an area that could provide more utility. For example, the ability to interact with visualizations by touch could allow for more engaging demonstrations compared to only interacting with the quads through a computer terminal. Allowing a researcher to touch a quad in the environment and drag it to a new start location could be a more intuitive way of conducting tests than to only enter in coordinates. This would add an additional layer of tangibility to simulation and perhaps the way we will interact with machines in the future.

Appendix

ImageManager API Design & Layout

Init (list of images)

Load_imgs (imagePath)

Calls import image for each imagePath

Import_img(imagePath, canvas, scaling, translation)

Scaling is defaulted to [1,1,1]

Translation is defaulted to [0,0,0]

Creates a transformation object and attaches the element to 'canvas'

Translate(imageIndex, translation_coordinates)

Scale(imageIndex, scaling_size)

Rotate(imageIndex, angle, axis)

Polygon Handler Formatting

To display data to the projector, the relevant data must be encoded into four arrays

Position Array : 3D position (X,Y,Z)
Element Array : A value that represents if the polygon is a disk, arrow or square
Color Array: A RGBA value representing the color of the polygon
Size Array: The radius(size) of an element in pixels

These arrays are passed over UDP in the following format

(Position, Element, Color, Size)

On the projector computer, these objects are read from UDP and unwrapped into these four arrays. Then the contents are displayed to the project using an OpenGL shader.

References

[1] Shayegan Omidshafiei, Ali-akbar Agha-mohammadi, Yu Fan Chen, N. Kemal Ure, Shih-Yuan Liu, Brett T. Lopez, Rajeev Surati, Jonathan P. How, and John Vian , “Measurable Augmented Reality for Prototyping Cyber-Physical Systems ” *To Appear*