# ­Live Streaming 360° Vision Sensors

Abstract

*This investigation aims to research and assess performance of various methods to implement 360° Vision streaming capabilities on Land Vehicles as part of DST’s Advanced Vehicle Systems STC in Land Division. Included is the documented attempts to use the FLIR Ladybug 5 360° camera and to integrate it with the Clearpath Robotics Jackal UGV. Design intends this system to provide images to be stitched and displayed by a suitable HMD enabled application.*

Introduction

The Advanced Vehicle Systems STC is part of the DST Land Division, and is responsible for developing next generation vehicle systems for all land personnel. One of the emergent technologies with a range of applications is the remote operation space is Virtual Reality, which provides immersive 360° vision to the user. This technology in conjunction with video streaming is highly desirable for the operation of land vehicles as it can provide superior situational awareness without compromising on user safety. To this end, the research conducted in this paper will investigate the early stages of implementing 360° Vision capabilities on a UGV platform, with further applications on similar AVS projects such as LOAVES. The primary hardware for this investigation is the FLIR Ladybug 5 Spherical Camera and Clearpath Robotics Jackal UGV, which served as a host platform for the camera systems.

As a proof of concept demonstration, it was desired to be able to show video streaming from the Jackal UGV to a custom HMD application created by a colleague. In order to achieve this the final camera system was required to stream .jpg image frames in real time to the host server, which would handle the process of displaying them on the HMD. How the images were acquired and processed were subject to the results of this research.

## Methodology

* Initial Compatibility Checks
* Software Development Kit and API install
* LadybugCapPro Testing
* Example API code errors and testing
* Custom C# application development
* Rejection of Ladybug 5 360° camera system
* Streaming with USB webcams
* Investigation of GoPro distributed array.

Overall, this paper documents a series of smaller research investigations on a number of alternate solutions to the 360° vision problem. Initial efforts focussed on using the Ladybug 5 Spherical camera, however research was subsequently extended to cover other systems. These included the use of USB enabled webcams and GoPros to create a stitched image for VR viewing.

**Literature Review/Initial Testing**

The investigation began with the review and analysis of all relevant documentation relating to the hardware being used. This included the reference material for the intended hardware selected by the project supervisor, as well as general background reading. This information became the foundation of all subsequent decisions regarding the hardware compatibility. By examining maximum ratings with regards to power delivery and requirements, it was found the Jackal could operate and power the Ladybug 5 without any additional modifications. The only hardware would be an open lead to dc jack to connect the Jackal’s screw terminals to the GPIO Vin and GND pins. Following the review, it was determined the manufacture of the mounting hardware necessary to secure the Ladybug to the Jackal would be outsourced to SES.

The software provided for use with the Ladybug 5 included a proprietary API and SDK, as well as LadybugCapPro, a capture and processing application for Windows 10. Following the install the camera was tested using LadybugCapPro. Image quality was excellent and the processing options made it easy to save in JPEG and bitmap formats with minimal difficulty. Video capture however required additional work, as the software only recorded to a proprietary ‘ladybug stream’ file. This could then be converted into a number of popular video formats such as .h264 and mp4, however took some time to process. At this point it became clear that whilst the camera was certainly capable of producing the image quality desired for this project, LadybugCapPro would not be suitable for the end application. This meant the research focus would need to shift to application development with the Ladybug API.

When examining the documentation for the Ladybug SDK and API it became quickly apparent that there would be some major difficulties implementing them in a custom application. The majority of documentation for the Ladybug 5 only referred to the API in passing, with brief examples on what example code was bundled with the SDK. An official help index did exist for the API, however outside simple descriptions of class names and functions, there was little to guide the development process. As a result, much of the knowledge derived about the Ladybug API would have to be found by inspecting and experimenting with example code.

**Ladybug API Testing on Windows 10**

This proved to be a very troublesome process, as of all the example code provided in C++ and C#, only two examples would compile in Visual Studio 2015 and function as described. These were the two C# examples LadybugCSharpEx and StreamProcessCSharp. The remaining two dozen example programs written in C++ were non-functional, plagued by an array of errors seeming to originate from within standard libraries such as math.h and stdlib.h. This was the first issue that was addressed when it came to further experimentation, and it was later found that by reproducing the example code in a new directory would resolve a large portion of the conflicts with standard libraries. However, a number of errors would still remain, thus rendering any apparent progress useless.

As all C++ examples were not functional, research turned towards implementing the Ladybug API in C#. LadybugCSharpEx was used as a template to modify with the goal of extracting data. By calling the pData method from the Ladybug API, a pointer to the image data memory location is returned. This then allowed access to the full image data the hexadecimal format. By utilising a string builder and logger funciton, it was possible to view this data in its raw format by dumping the contents of pData into a log file. These modifications were then added to the LadybugCSharpEx program to run as part of the image capture function. Every time an image is captured, the current image data would also be appended to a string. This string would then be written to the log file via the string builder.

Having established that the image data was being accessed correctly from viewing the log file, the next step was to establish a streaming connection between the host pc and the client running the Ladybug 5. For this, a UDP connection was chosen to be in line with the AS-GVA Defence Standard for data connections. Through collaboration with the Live Stream Oculus HMD research project, the UDP server was established as a parallel process to LadybugCSharpEx. The program would continue to write the contents of pData to the log file as before, however it would also feed the data through the server to be viewed remotely on the host PC.

**Custom Applications**

With a proof of concept for the use of the Ladybug 5 to capture and stream image data, the next step was to plan, design and implement a custom application suitable for the project needs. This required greater integration with the Live Stream Oculus HMD research project, which was responsible for developing the HMD application that would process and display the image data collected by this project. By this point in development, the interface to the Unity application displaying images required a byte stream of .jpeg images via the UDP server previously used to test the data streaming. Because of this interface, it became necessary to not only acquire and pass on the image data from the camera, but also to find a why to process and convert the data to the required format in a timely manner. Two designs were considered. The first would use client-side processing to render the jpeg files onboard the UGV before sending it to the server. This method depended on the client application managing three tasks simultaneously: Camera operation and data capture, image processing, and data streaming. Such an application would easily integrate with the existing Unity application, but could require excessive overhead on the client and impact video performance. The other design instead used server-side processing to avoid unnecessary load on the client, and simply would capture and send image data as fast as possible. Choosing between these designs was delayed to be able to research how each method would affect performance of the final application. For the sake of testing purposes, demo applications would use the client-side processing method to demonstrate the capability, but leave the option of moving that functionality to the server at a later date.

Viewing the example code and documentation outlined a specific procedure that needed to be followed in order to initialise the Ladybug 5 in C#. The API depends on a custom *LadybugContext* object to handle all API related tasks. This is required to be created for the majority of Ladybug functions to be called as it is used as the primary parameter. *LadybugStart()* is then called on the context to start up the camera. Further functions such as getCameraInfo() and *XXXXXXX*  also were required for use of later functions. It is important to note that the serial number of the Ladybug 5 camera being used is also needed to address it via the API. For this investigation it was found by using LadybugCapPro to view the information of each FLIR camera connected to the PC. After testing and debugging the start up procedure, all code was grouped into a custom *LadybugStartup()* that can be used in any future functions to initialise the camera.

**Failure of C# Implementation**

Following the completion of the *LadybugStartup(),* work moved to replicating the *pData* method used in *LadybugCSharpEx()* in the new custom application. Unfortunately, the limited documentation combined with the lack of functional examples made it difficult to determine how to implement such methods. Some examples in C++ completed tasks similar to those required but being limited to working in C# made those of little help. It was deemed that the required investment of time and resources was beyond what was achievable in the project timeframe. Thus this method was abandoned in favour of investigating other methods.

**Alternate Solutions**

After a discussion with project supervisors, research was put on hold with regards to the Ladybug 5 spherical camera, with research turning to the use of more conventional USB webcams for image capture. Without the difficult proprietary API these cameras were much more convenient to use and a stream could easily be implemented using VLC media player.

**Linux Implementation**

Following extensive attempts to solve the API issues when running the camera on Windows, one of the few avenues left was to pursue a fresh install on Linux. Ubuntu 16.04 was selected as a more stable version to work on with regards to the Ladybug software. The install was still broken until the configuration file was appropriately set, in order to meet certain memory management requirements. However, the crux was that all C++ libraries could now finally be used and modified to make custom functions as originally intended, albeit with limitations. Currently the only way to make functional code with the API is to use a modified example with its corresponding makefile. This is because the g++ compiler is unable to find the ladybug .h files it needs to call the API. This needs to be investigated further in order to speed along development.

The first modification made to the API examples was based on the *LadybugSimpleGrab* example. This was changed to take multiple frames to serve as input to the stream client and HMD display. Initial tests showed that this would produce approximately 5fps video with the current code, meaning that more work was required to make it suitable for a real time application. To achieve this, the image format was changed to .jpg, which brought the file size per image down from 120MB as a .bmp to 650kB. Next was to parallelise the image saving function.

**Install Process**

The install requires a complete Ubuntu 16.04 or similar installation. Then the Ladybug install must be downloaded from Point Grey Research. This requires an account with FLIR and specifying the correct OS to find the appropriate file. To install the Ladybug SDK, simply install the the .deb file like a regular package. The next step is to adjust the image file size limit within Linux. This process is detailed in the SDK Readme, and requires editing the /etc/default/grub file to support larger image files over USB. Following this install all example code in C++ and C# should be able to compile using the included makefiles.

**Modifying Example Code**

Of all the examples available for use within the SDK, LadybugSimpleGrab was selected as a starting point. This was due to it being capable of saving the desired image frames that could be used as a video feed for the HMD display being developed by the Live Stream Oculus HMD project. In its initial form, the example would initialise the camera, take a picture using all 6 cameras on the Ladybug 5+, then save them as a bitmap. This would result in 6 independent images corresponding to each camera on the device.

Modifications were made to this code to loop the image capturing process a desired number of times, to simulate the recording of video. The implementation was to simply place all image capturing code in a for loop specifying the desired number of frames. Due to being stored as an uncompressed bitmap, each image was extremely large at 120MB per file. This was adjusted by modifying the API specific function *ladybugSaveImage* parameters to save files in jpeg format.

## Results

Initial research into the Ladybug 5 proved to be quite encouraging. The hardware is capable of extremely high-quality images and video. The post processing is powerful, and supports a wide range of formats. However the supplied API was found to be extremely difficult to work with in some places, and in other bordering on incomprehensible. The main failing is that despite supporting development features, the majority of the documentation supplied with the Ladybug 5 pertains to the LadybugCapPro application for Windows users. Outside of brief descriptions on what example code exists and the limited comments within the code, no guides or outlines exist on how to control the Ladybug 5 with custom software. This is in addition to the fact the C++ code was broken upon install. These two major flaws essentially stopped all further development in its tracks, and at the time of writing, remain unresolved.

It is unfortunate this is the case, given the volume of example code that is provided. In fact, several examples are almost perfect for immediate integration. These functionscould easily be modified to be used in the capture and processing components of this project resulting in a powerful setup with superior image quality that integrates easily into the Live Stream Oculus HMD project as a video source.

Take *LadybugSimpleGrab()* for example

Unfortunately the plethora of build errors in the C++ code prevent this from eventuating.

**C# Development Results**

It was due to the issues with the C++ API the instantiated the need to use C# as the main development language. The work completed in this instance resulted in the creation of the *LadybugStartup()* function, which would enable any suitably referenced function to complete a standard start up procedure with the Ladybug 5. This code could prove very useful in the event of future development using the Ladybug API as it lowers the barrier of entry for a new developer to continue the work detailed in this report. Whilst not a flawless design, the code provides core functionalities that are essential to using the Ladybug successfully. The current iteration has perhaps one major flaw with regards to the device serial number. As it stands, the serial number is hard coded as an integer variable in the namespace containing *LadybugStartup()*. This means that each time a different camera is used with this code, the variable will require manual changes to work correctly. An easy fix for this would be to pass the serial number as an input parameter, changing the function signature to *LadybugStartup(int serialNumber).* This would make the code truly modular, being able to address any Ladybug camera providing it is given the appropriate serial number.

**PData Reference and append Failure**

Following the completion of the *LadybugStartup(),* work moved to replicating the *pData* method used in *LadybugCSharpEx()* in the new custom application. This was not met with much success. The stringBuilder method and log file could be setup without any error, however calling the append function did not produce the results they had before. In fact, after calling *append()* the whole function would exit without explanation. No error messages were given, and none of the internal exception handling incorporated into the code triggered either. This would imply that the function was running correctly and returning no output. However when tested in a loop, the function would exit immediately after calling *append()*, breaking the loop before it completed the first iteration. This was confirmed by the loop running as expected once the *append()* call was removed. Despite extensive debugging efforts, not solution could be found for this issue.

Given the failure to extract image data directly from the source of the image, it was decided the best alternative would be to utilise the Ladybug API’s built in image processing methods to render the jpeg files for the server. This likely would be less efficient than the original method, but would provide basic functionality.

## Discussion

**Ladybug API broken on Install**

It is unfortunate this is the case, given the volume of example code that is provided. In fact, several examples are almost perfect for immediate integration. These functionscould easily be modified to be used in the capture and processing components of this project resulting in a powerful setup with superior image quality that integrates easily into the Live Stream Oculus HMD project as a video source.

Take *LadybugSimpleGrab()* for example. This function, when run as a compiled executable, completes the start up procedure, then instructs the Ladybug to take a picture with all size cameras. It then processes these images, and saves them as six jpeg files into a designated directory. This output is perfect for the use in Unity to create a stitched VR image, and the output format meets the interface requirements of the Live Stream Oculus HMD project. To modify this for video capture, the only requirements would be to place the image grab, processing and saving methods into a loop to provide a continuous feed of frames. This process could then be placed into a separate thread, with a second thread reading the destination directory and streaming the files via UDP. Whilst the performance might be limited due to processing 6 jpeg files per frame of footage, it would serve as a bare bones version of the 360° Vision system.

Likewise other examples within the API such as *LadybugStreamProcess* and *LadybugStitchFrom3DMesh* provide valuable processing me that would be extremely useful later on in the function pipeline. However without any of these functions capable of being modified, they are next to useless. The experience with FLIR support proved a very slow process. Undoubtedly with time and a great deal of patience, the API install could be fixed with their assistance. However given the time constraints with regards to this project it was deemed a secondary matter in favour of developing a functional prototype.

One of the largest functional limitations of the Ladybug API is the restrictive nature it has when it comes to modifying code. Many times throughout the duration of this project it was found that an otherwise simple process was rendered much more difficult to implement because of this. More specifically, adding additional libraries to the C++ code was necessary to achieve the desired capabilities. This is because in order to run the Ladybug example code, a makefile must be used to compile it. This is currently the only known way to use the Ladybug API on the Linux platform, meaning any code with dependencies not included in the default example will not function. It should be possible to modify the makefile to allow the inclusion of additional libraries for further development, however this has not been attempted as of yet. Another solution would be to use an alternate compiler, however this also has been met with no success. The g++ compiler that ran the POSIX multithreading code intended to modify the example LadybugSimpleGrab was unable to compile the program with the additional Ladybug header files.

This issue would persist when it came to adding UDP functionality to the final application. As the Ladybug code could not be modified with additional libraries, the only solution that could achieve any degree of function would require running the UDP server independently of the camera control. This design has a plethora of issues, chiefly synchronisation between the two programs. Without a direct link it is near impossible to reliably ensure the correct frames are sent through the connection. More likely the system would output frames multiple times or perhaps not at all if the programs became too far out of sync.

## Conclusion and Recommendations

Having spent an extended period working with the Ladybug 5 360° Camera, it is clear that it is a highly sophisticated device which meets or exceeds many of the requirements with regards to 360° Vision Sensor hardware. However the poorly documented API is of huge detriment to ability to be implemented in a practical sense. As is clear, this single factor is responsible for the vast majority of problems faced in this project. Were the Ladybug 5 chosen to be pursued further as the hardware of choice it would require a highly specialised team to properly and reliably implement it. It would also likely require a larger time investment to properly understand the Ladybug API that drives the device.

On a more promising note, once each issue was resolved it became clear that it is functionally capable of serving the desired functions as a Land Vehicle’s omnidirectional vision sensor, provided the suitable investment on software development.

**Limitations of IPv4 and Wireless Networks**

With the provided hardware and software limitations, the design space is very restricted with regards to data transmission. After encoding, each image produced by the Ladybug5+ is approximately 600-750kB in size. Given UDP has a maximum packet size of 65kB, it is impossible to send images in a single packet. Therefore, data will need to be packetized in order to be transferred in such a manner. This greatly complicates the process as a UDP does not guarantee the delivery of individual packets, and is generally cannot be relied upon for perfect transfer of data. Such a limitation can compound as missing packets from an image can corrupt the entire image file after the packets are reassembled.

The second limitation is the transmission speed of wireless networks. Each image from the Ladybug is on average 700kB, with there being 6 images being sent per frame. That means each frame requires a total of 3.5MB of throughput, resulting in 52.5MB/s of data transmission required to achieve 15fps. This neglects any processing time required to packetize and reassemble each image, and all video processing time after the images have been received. This equates to 420 Mb/s of data throughput needed to provide this framerate. Even more ambitious is a smooth 30fps which would need 840Mb/s throughput speed. This limits the networks able to utilise this technology effectively to those with a gigabit network connection.

This seriously undermines the feasibility of this entire project as typical routers are rarely capable of such transmission speeds. Some routers can achieve this speed however a dedicated network would be required to ensure superior performance. The more serious issue is the limitation of the Jackal’s onboard Wi-Fi. This will likely need to be supplemented by a wireless connection to an additional router to provide the adequate data rate.

Another solution is to compress the image further still using a custom compression algorithm, which would require a great deal of time and effort to create for the given scope. However, at this stage it seems more reasonable to start considering changing the image source to work in these constraints. Sourcing more fit for purpose hardware could save many man-hours of work by reducing the initial file size before any compression. This could prove a far simpler fix than going to the effort to develop a new compression method.

As an attempted solution, additional networking capabilities were added to the Jackal’s PC by placing a Mesh Wi-Fi access point onboard the robot. This was wired directly to the PC gigabit ethernet port. Given the access point is rated for 867MB/s on the 5GHz band, it is possible that the Jackal will be able to stream data at the required rate.