Feature Article:

DOI. No. 10.1109/MAES.2020.3002000

Augmented Reality for Immersive and Tactile Flight Simulation

Jon W. Wallace, Zhengxie Hu, Daniel A. Carroll, **Lafayette College**

INTRODUCTION

Flight simulators have a history of extensive use in civilian and military aviation, providing a significant reduction in the overall cost of flight training. Simulators also allow potentially hazardous situations, such as aircraft failures and other emergencies to be safely experienced.

Enclosed-cockpit simulators with wrap-around displays and full instrumentation currently provide an experience that is very close to a real cockpit, representing a valuable tool for flight training. For example, Figure 1 depicts an enclosed-cockpit, multiscreen simulator that was developed by students and advisors of the N85 Aerospace Club, which largely inspired the research in this article. Special attention was taken to reproduce actual controls in the Cessna C172P aircraft, one of the main models used for pilot training at N85. Although duplicating a full cockpit in this way can be valuable for pilot training, the required space and cost can be prohibitive. For example, commercial general aviation (GA) flight simulators from Redbird Flight Simulations, Inc., with either a wrap-around display or a fully enclosed cockpit are listed at \$32 800 and \$42 800, respectively, and require approximately $2.7\ \mathrm{m} \times 1.8\ \mathrm{m}$ floor space [1].

Virtual reality (VR) is developing into a mature technology and offers an interesting alternative to wrap-around display or enclosed-cockpit simulators described previously. An early VR-based helicopter simulator was presented in [2], which employed a stereo head-mounted display (HMD) driven by the FlightGear open-source flight simulator, hand-tracking gloves to interact with cockpit instrumentation, and a custom designed model for flight dynamics. Later work in [3] studied the interaction of a virtual hand (or "avatar") with buttons in a VR cockpit, demonstrating the importance of the shape and configuration of the avatar for robust pilot-control interaction. The work in [4] proposes VR as a tool to enhance safety during drone pilot training. Recently, the U.S. Air Force has been investigating the ability of VR technology to accelerate flight training, where students are exposed to maneuvers in VR as preparation for in-cockpit instruction [5]. The support of VR has also become a native feature of commercial flight simulator packages, such as X-Plane from Laminar Research and Prepar3-D from Lockheed Martin.

Benefits of VR-based simulation are the fully immersive experience, low cost, and space requirements, and the ability to simulate many different aircraft with the same hardware. An important drawback, however, is the loss of natural tactile interaction with flight controls and instruments. Although this may not be a problem for HOTAS (hands on throttle-and-stick) controls in fighter aircraft, where a pilot's hands do not leave the controls, many trainer and civilian aircraft require the pilot to move the right hand to various controls, such as throttle, propeller pitch, carburetor heat, mixture, flaps, trim, landing gear, radios, etc. Fairly early in training, key controls like throttle and flaps should become familiar enough to control by touch, keeping eyes looking outside the cockpit as much as possible. A VR cockpit without the familiar touch sensation could lead to the bad habit of spending too much time looking at controls inside the cockpit. Eventually, advanced VR hardware, such as force-feedback gloves may bring natural tactile feel to VR instruments, but current solutions are bulky and will hamper pilot movement.

The purpose of this article is to explore the use of augmented reality (AR) for flight simulation, offering many of the benefits of VR realism, but without sacrificing natural tactile interaction with cockpit controls. The term AR was first used by T. Caudell and D. Mizell, two researchers at Boeing who developed a see-through, heads-up display for Boeing-747 factory workers. The headset superimposed the required assembly patterns on top of wire bundles, providing a more convenient and flexible solution than the wiring formboards that were in use [6]. Since then, AR has found significant use in other manufacturing industries [7].

Authors' current addresses: Jon W. Wallace, Zhengxie Hu, Daniel A. Carroll, Lafayette College, Easton, PA 18042 USA (e-mail: wallacjw@lafayette.edu).

Manuscript received April 1, 2020, revised May 20, 2020; accepted May 23, 2020, and ready for publication June 10, 2020.
Review handled by Peter Willett.
0885-8985/20/\$26.00 © 2020 IEEE

Credit: Image licensed by Ingram Publishing

AR is particularly interesting for space applications, allowing astronauts to execute procedures with minimal probability of error [8].

In the remainder of this article, we describe a proof-of-concept AR flight simulator that was developed by students and their faculty mentor at Lafayette College, an idea that was largely inspired by flight simulator development activities at the N85 Aerospace Club. Although the simulator has a relatively low hardware cost (approximately $4000 USD), it provides not only an immersive flight experience, but also natural tactile interaction with flight controls.

AR FLIGHT SIMULATOR CONCEPT

Before delving into our specific AR simulator implementation, we describe the basic idea of its operation. Due to its low cost, we use the *indirect view* AR approach, where a conventional VR HMD delivers video to the user and tracks the head position and pose, while a stereo camera attached to the HMD captures images of the user's surroundings. A personal computer (PC) then combines virtual scenery generated by the flight simulator software with camera video, allowing certain areas of the virtual cockpit to show VR scenery, and other areas to show the pilot's hands and controls.

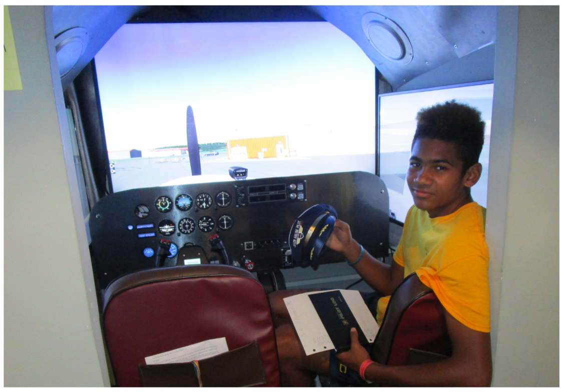


Figure 1.
Enclosed-cockpit flight simulator with wrap-around displays at Alexandria Field Airport (N85), simulating a full instrument panel for a Cessna C172P aircraft. Photo by courtesy of Sean Smith and Linda Castner of the N85 Aerospace Club.

We have explored two ways of combining virtual scenery and camera images, as depicted in Figure 2. The "geometry-based" method is appropriate for simulators, where an enclosed physical cockpit has been built, and VR scenery needs to be presented on window areas. This method requires a three-dimensional (3-D) model of cockpit window areas to be available. By combining this information with headset position and pose, the 3-D model can be properly oriented in the pilot's frame of reference, allowing perspective projections of the window areas to be computed and overlaid with VR scenery. Note that the 3-D model could also indicate positions of instruments, allowing VR instruments to be drawn on a generic (blank) instrument panel.

The simple geometry-based method has the disadvantage of requiring an enclosed physical cockpit to be built. Also, if the pilot's hand is moved over a window area, it appears to be "chopped off" as it is overlaid with VR scenery. We therefore explore an alternative "green-screen" method that uses conventional chroma keying. Here, a physical green screen is placed in the background, and pilot controls and instruments are placed in the foreground. Image processing identifies green areas in the camera video and overlays those areas with the VR scenery, allowing camera images from other areas to show through. This method avoids the need for a detailed 3-D model and allows a pilot's hand to show correctly over VR scenery. However, chroma key requires good lighting without shadows, which can sometimes be a challenge.

We expect that a combination of two methods may be the best overall solution. For example, an enclosed cockpit could simply have green screen on the window areas, avoiding the "chopped hand" problem. Or, a compact three-sided cockpit with front, left, and right green screen panels could be constructed, which only needs to extend as far as instruments, controls, and the reach of the pilot's hands. A very simple 3-D model consisting of six rectangles could then be created, which when combined with the green screen panel locations, forms a complete rectangular box. The image processor then overlays both green-screen areas and projected 3-D model areas with VR scenery.

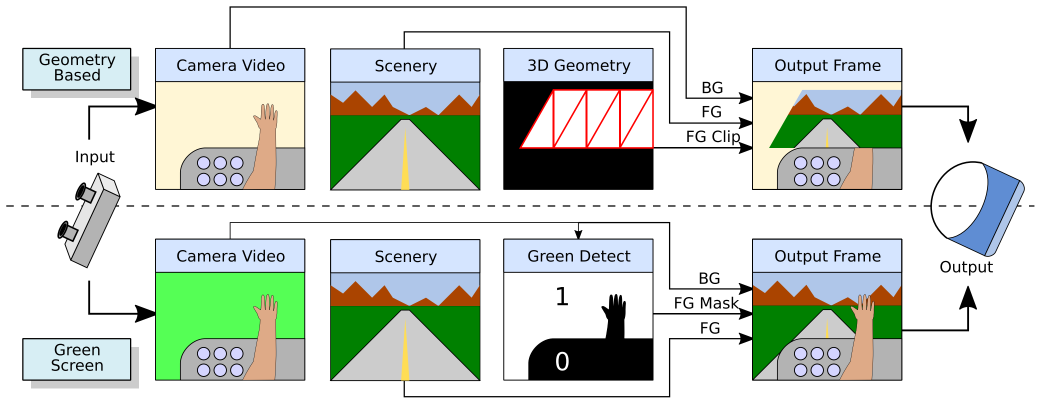


Figure 2.
Processing steps for the proposed AR flight simulator. An indirect view approach is used where input from a stereo camera is processed and displayed on a VR HMD. The top and bottom image processing chains show operations for geometry-based and green-screen overlay. BG and FG stand for a background image (drawn first) and a foreground image (drawn second), respectively.

Not only does the pilot have the illusion of being inside a complete VR cockpit, but also hands placed over window areas will look natural. Due to space constraints, we only consider the extremes of geometric and green-screen methods in the remainder of this article.

SYSTEM DESCRIPTION

HARDWARE COMPONENTS

Table 1 lists the hardware components of the AR flight simulator we used and their approximate cost (in 2020 USD), where our goal was to simulate a GA trainer aircraft. The Vive Pro headset provides 1440×1600 resolution per eye, a combined 110° field of view, and 90 frames-per-second (FPS) refresh rate. The Ovrvision Pro Stereo camera has a USB 3.0 interface and can deliver dual 960×950 color images at 60 FPS. Note that the Vive Pro has a built-in stereo camera that can be used, but the resolution is fairly low (480p), and we could only reach 30 FPS capture rate in our experiments.

A photograph of the desk-based test setup is shown in Figure 3, which is self-explanatory except for the instrument panel. The idea is to have a generic panel, where different gauges can be placed at desired positions. This is accomplished using ArUco tags, which are special binary-code markers that were developed for AR. Each unique ArUco tag can be detected by the image processing software and overlaid with a corresponding gauge. So far we have only written software to overlay an altimeter, airspeed indicator, and tachometer, but this illustrates the basic principle. The panel was made from fiberboard covered with carpet, allowing Velcro-backed ArUco gauge tags to be placed at arbitrary locations.

SOFTWARE COMPONENTS

Ubuntu Linux 18.04 was chosen due to ease of development and the availability of many open-source libraries. Figure 4 shows the software components that were required to implement the AR simulator and the basic data flow. Existing software components are shown by tan colored boxes in Figure 4 and consisted of the following.

Table 1.

System Hardware Components

Headset / Background		
HMD / Tracking	HTC Vive Pro	$900
Stereo Camera	Ovrvision Pro	$500
Green Screen, Stands	Various	$100
Personal Computer (PC) Components		
CPU	Ryzen-7 2700x	$300
Memory	32 GB	$150
GPU	Nvidia RTX-2060	$300
Motherboard, case, etc.	Various	$250
Simulator Controls/Instruments		
Pedals	Thrustmaster Rudder Sys.	$500
[1]Control Stick/Yoke	Custom Built	≈ $200
Throttle Quadrant	Desktop Aviator	$150
Instrument Panel	Custom Built	$50
[2]Switches, dials, etc.		≈ $500
Total		$3900

[1]Although we used a custom-designed two-axis stick that was available in our lab, a consumer-grade joystick or yoke control costs under $200, which is listed in the table.
[2]These are components that were not used in the proof-of-concept setup, but would be required for a full GA cockpit.

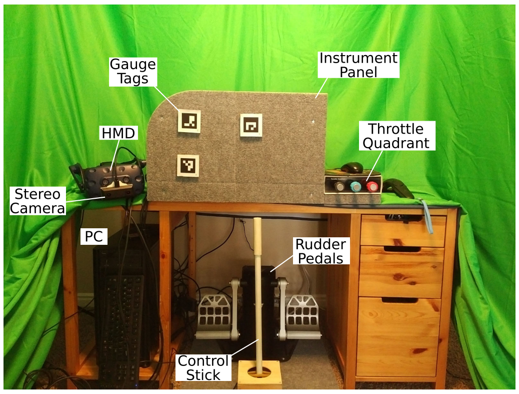


Figure 3.
Photo showing components of the desk-based setup for AR flight simulator experiments.

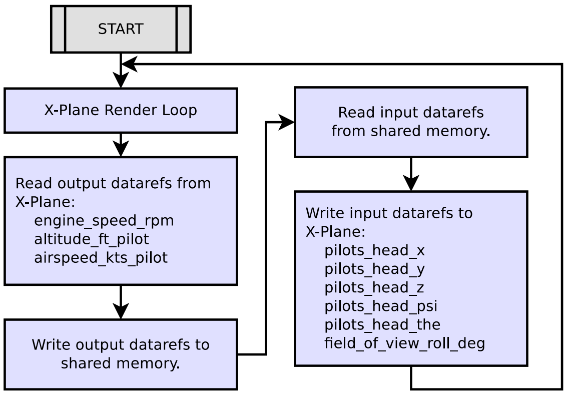


Figure 5.
Flowchart followed by the custom X-Plane plugin, indicating datarefs that are read/written to implement the virtual AR cockpit.

X-Plane Flight Simulator. X-Plane was chosen due to its low CPU footprint and Linux support, as well as the power and simplicity of its plugin interface. Program flow of the custom plugin we developed is shown in Figure 5, which outputs gauge data (altimeter reading, engine revolutions per minute, and air speed reading) to the AR Process, and receives the pilot head position and pose from the AR process, updating the required "dataref" structures in X-Plane. The plugin sends and receives data at a nominal 60 FPS. X-Plane renders scenery according to the current pose to an X11 window, which is grabbed by the AR Process.

Ovrvision Pro Camera. The stereo camera appears as a Video4Linux2 (V4L2) device. Color stereo frames at 960×950 resolution per eye are captured through the V4L2 interface at 60 FPS and passed to the AR Process.

OpenVR. The AR Process communicates with the HTC Vive Pro HMD using the free OpenVR library. For this to operate, SteamVR must also be installed and running on the system. The OpenVR application programming interface (API) allows pose information of the headset and controllers to be read, as well as video frames (OpenGL textures) to be written to the headset screen.

AR Process is a custom command-line program that was developed for this project, implementing all of the required data handling, image processing, and graphics rendering for the AR flight simulator. Below we describe the program flow and operations performed in AR Process, noting that full source code is provided at [9] for interested readers.

Capture Thread. A flowchart of the Capture Thread is shown in Figure 6(a), whose purpose is to read stereo camera frames from the V4L2 interface, and send these with a first-in first-out (FIFO) buffer to the rendering thread. Additionally, left and right video frames are placed in separate FIFOs bound for two ArUco tag detection threads.

X11 Grab Thread. Figure 6(b) depicts a flowchart of the X11 Grab Thread, which grabs video from the X11 window that X-Plane uses to render its scenery. Frame grabbing in this thread is synchronized with the capture thread to obtain nearly coincident samples of camera and rendered X-Plane scenery.

ArUco Detection Threads. The two ArUco Detection Threads follow the program flow in Figure 7, and their

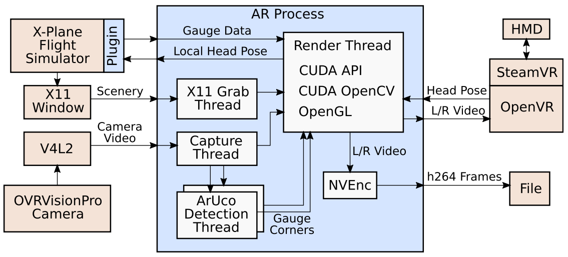


Figure 4.
Block diagram of AR flight simulator software components. The "AR Process" block and X-Plane plugin are the custom components that were developed as part of this project.

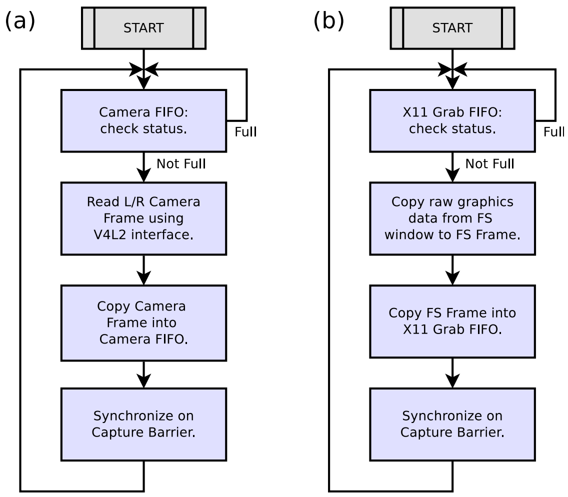
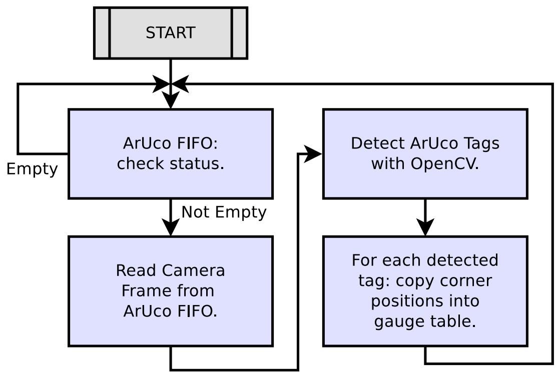

(a)
START
Camera FIFO: check status.
Full
Not Full
Read L/R Camera Frame using V4L2 interface.
Copy Camera Frame into Camera FIFO.
Synchronize on Capture Barrier.
(b)
START
X11 Grab FIFO: check status.
Full
Not Full
Copy raw graphics data from FS window to FS Frame.
Copy FS Frame into X11 Grab FIFO.
Synchronize on Capture Barrier.

Figure 6.
Flowcharts of data input threads: (a) Capture Thread for stereo camera input, and (b) X11 Grab Thread for grabbing flight simulator scenery.



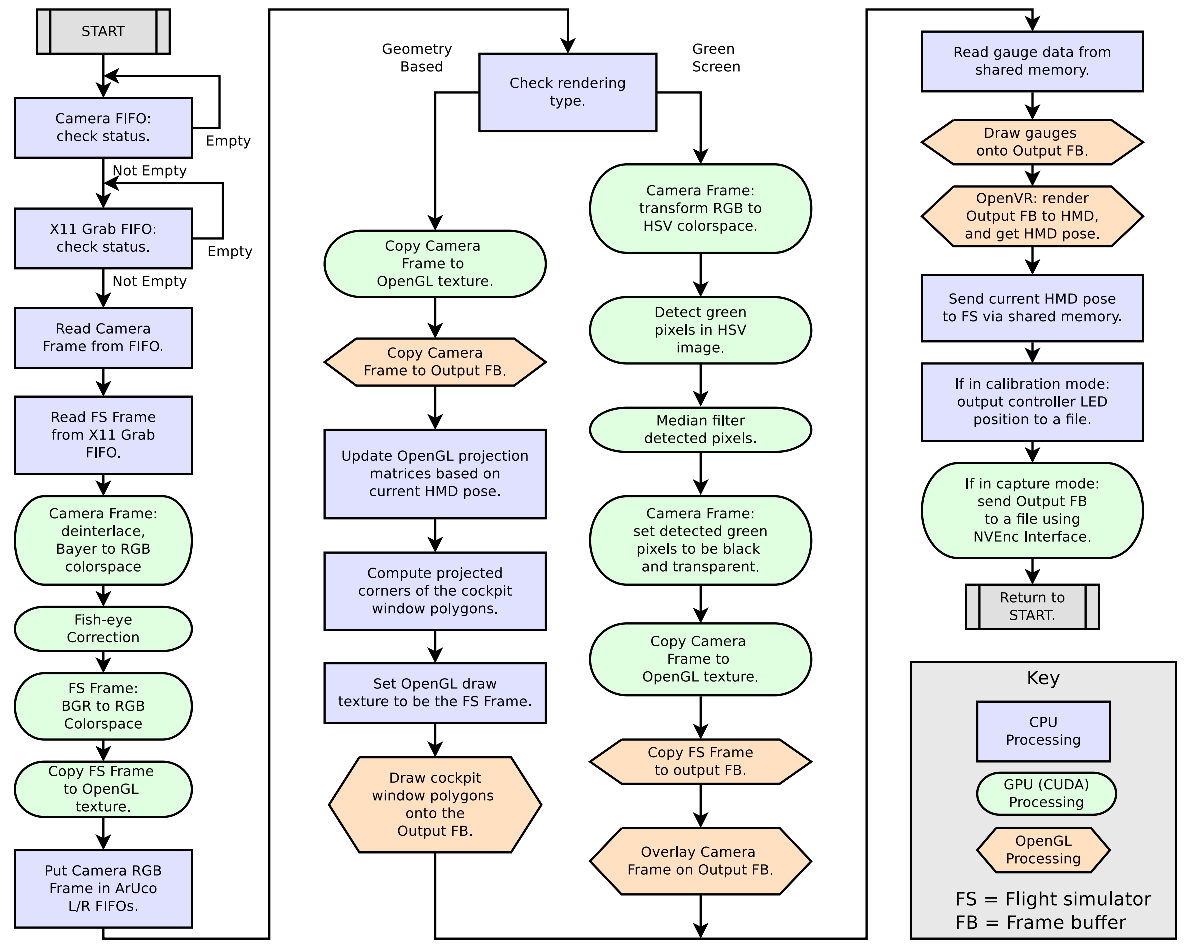
START
ArUco FIFO: check status.
Empty
Not Empty
Read Camera Frame from ArUco FIFO.
Detect ArUco Tags with OpenCV.
For each detected tag: copy corner positions into gauge table.

Figure 7.
Flowchart followed by the ArUco Detection Threads used to detect virtual gauge positions in left and right camera input frames.

purpose is to detect the position of ArUco tags on the instrument panel, allowing them to be overlaid with VR gauges. This is accomplished using the OpenCV (Open Computer Vision) library in the CPU domain, requiring two separate threads to detect tags in the left and right video frames at 60 FPS. The locations of tag corners are passed to the Render Thread via shared memory.

Render Thread. This somewhat complex thread is the heart of the system and its program flow is shown in Figure 8. The thread employs the NVIDIA Compute Unified Device Architecture (CUDA) API for GPU-based green-screen detection and the CUDA OpenCV library for color space conversions and camera fish-eye correction. OpenGL is used to compute 3-D model transformations and perspective projections, as well as to render gauges, camera video, and flight simulator scenery onto the output frames (textures) that are sent to the HMD. Note that all computationally intensive operations in this thread are performed on the GPU, minimizing processing latency and allowing frame rates of 60 FPS or higher.

NVEnc. For demonstration purposes as well as later reference, rendered video frames can be recorded to a video file in the raw H264 format. The RTX-2060 GPU has a built-in H264 hardware encoder, allowing the video stream to be encoded in real time with minimal impact on CPU or GPU load.



START
Camera FIFO: check status.
Empty
Not Empty
X11 Grab FIFO: check status.
Empty
Not Empty
Read Camera Frame from FIFO.
Read FS Frame from X11 Grab FIFO.
Camera Frame: deinterlace, Bayer to RGB colorspace
Fish-eye Correction
FS Frame: BGR to RGB Colorspace
Copy FS Frame to OpenGL texture.
Put Camera RGB Frame in ArUco L/R FIFOs.
Geometry Based
Check rendering type.
Green Screen
Copy Camera Frame to OpenGL texture.
Copy Camera Frame to Output FB.
Update OpenGL projection matrices based on current HMD pose.
Compute projected corners of the cockpit window polygons.
Set OpenGL draw texture to be the FS Frame.
Draw cockpit window polygons onto the Output FB.
Camera Frame: transform RGB to HSV colorspace.
Detect green pixels in HSV image.
Median filter detected pixels.
Camera Frame: set detected green pixels to be black and transparent.
Copy Camera Frame to OpenGL texture.
Copy FS Frame to output FB.
Overlay Camera Frame on Output FB.
Read gauge data from shared memory.
Draw gauges onto Output FB.
OpenVR: render Output FB to HMD, and get HMD pose.
Send current HMD pose to FS via shared memory.
If in calibration mode: output controller LED position to a file.
If in capture mode: send Output FB to a file using NVEnc Interface.
Return to START.
Key
CPU Processing
GPU (CUDA) Processing
OpenGL Processing
FS = Flight simulator
FB = Frame buffer

Figure 8.
Flowchart followed by the Render Thread, which implements geometry and green-screen based AR.

CAMERA-VR CALIBRATION

In order for camera video to be overlaid in the correct location on the graphics textures (output frames) sent to the HMD, some kind of calibration procedure is required. The procedure outlined below assumes that fish-eye distortion has already been removed from camera images, which was accomplished in this article using the CUDA OpenCV library.

A two-dimensional (2-D) projected VR point is denoted $\mathbf{r}$, and the corresponding 2-D point in the camera's image is denoted $\mathbf{r}'$. For simplicity, we assume the camera frame for each eye only requires scaling, rotation, and translation to align a camera pixel with a perspective-mapped VR point. This means we have the relation

$$\underbrace{\begin{bmatrix} r_x \\ r_y \end{bmatrix}}_{\mathbf{r}} = \underbrace{\begin{bmatrix} M_{11} & M_{12} & t_x \\ M_{21} & M_{22} & t_y \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} r'_x \\ r'_y \\ 1 \end{bmatrix}}_{\mathbf{r}'}. \quad (1)$$

Note that the M and t matrix entries provide scaling/rotation and translation, respectively. The unknown matrix $\mathbf{M}$ can be found by collecting many samples of coincident $\mathbf{r}$ and $\mathbf{r}'$ into the matrices

$$\mathbf{R} = [\,\mathbf{r}_1 \quad \mathbf{r}_2 \quad \ldots \quad \mathbf{r}_N\,] \quad (2)$$

$$\mathbf{R}' = [\,\mathbf{r}'_1 \quad \mathbf{r}'_2 \quad \ldots \quad \mathbf{r}'_N\,] \quad (3)$$

where N is the number of camera-VR coordinate sample pairs. We then solve (1) in a least-squares sense with $\mathbf{M} = \mathbf{R}\mathbf{R}'^{+}$, where $\{\cdot\}^{+}$ denotes matrix pseudoinverse.

The method we have chosen for finding each projected 2-D VR point $\mathbf{r}_i$ and its corresponding camera point $\mathbf{r}'_i$ uses an HTC Vive controller. The controller has a green LED on the handle that shows up well in camera imagery, and a simple peak detector can be used to find $\mathbf{r}'_i$ for that point. Next, the point $\mathbf{r}_i$ is found by obtaining the position and pose of the controller from OpenVR, computing the position of the LED in VR 3-D space, and then projecting that point with OpenGL transformation matrices to the 2-D image point $\mathbf{r}_i$.

Next we show an example of a camera-VR calibration phase, where the HTC Vive controller is moved along a random path within the camera field of view. Figure 9 shows estimated 2-D camera coordinates $\mathbf{r}'_i$ (a) and corresponding projected 2-D VR coordinates $\mathbf{r}_i$ (b) of all path points. Note that the two curves on each plot show separate information that is gathered for the left and right eyes. This procedure provides hundreds of pairs of $\mathbf{r}'_i$ and $\mathbf{r}_i$, allowing the $\mathbf{M}$ matrix to be computed for each eye. A visual comparison of Figures 9(a) and (b) indicates eye-dependent scaling, rotation, and translation, as mentioned previously, as well as some nonlinear effects, especially near camera boundaries. Error in the mapping for left and right eyes can be quantified by evaluating (1) for each point $\mathbf{r}'_i$ and computing the mean-squared error between the actual and mapped $\mathbf{r}_i$ values. Error histograms are shown in Figure 10, indicating approximate average error and peak error of 2% and 10%, respectively, over the complete visible HMD frames. Although more advanced calibration and video mapping methods for indirect-view AR could be applied, we feel this simple method has sufficiently low error for this initial treatment.

EXAMPLE AR VIDEO

In this section, we show some example imagery produced by the AR flight simulator as viewed by the user in the HMD.

ArUco TAG DETECTION ANDGAUGE OVERLAY

A close-up view of gauge detection and overlay using ArUco tags is shown in Figure 11. The OpenCV ArUco tag detection is surprisingly robust, and for a stationary headset, tags are well detected and gauges do not move appreciably. Since gauges are currently detected frame by frame without track-

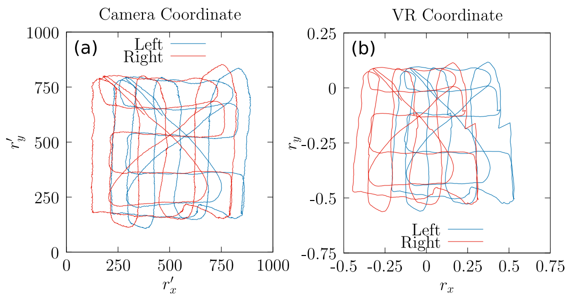


Figure 9.
Example of camera-VR coordinate pairs that are collected by moving the HTC Vive controller over a random path. The position of the Vive controller's green LED is detected in camera frames, and the resulting sequence of sample points is shown in Plot (a) for each eye in pixel units. For the same samples, Plot (b) shows the 2-D projected OpenGL coordinates where OpenVR would draw the LED for left and right eyes. Information from Plots (a) and (b) is used to obtain an approximate linear mapping from camera to VR-projected coordinates, allowing camera imagery to be displayed properly on the HMD.

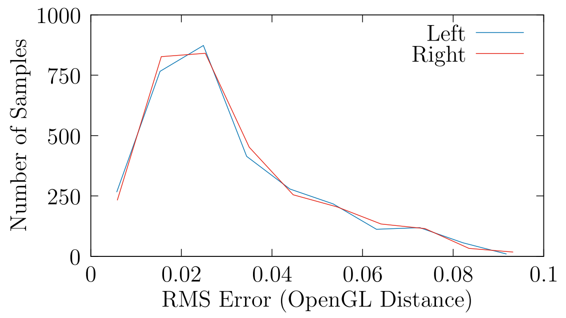


Figure 10.
Error histogram of mapped VR points using estimated $\mathbf{M}$ matrices for left and right eyes.

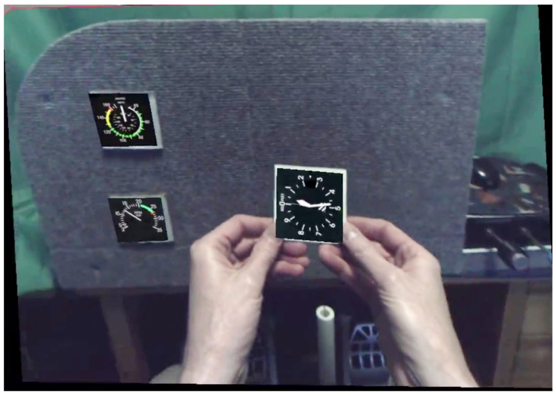


Figure 11.
Detection of ArUco tags and overlay with simulated VR gauges. Moving counter-clockwise from the bottom left, ArUco tag identifiers 0, 1, and 2 are overlaid with the tachometer, altimeter, and air speed indicator.

ing, they disappear when moving the head quickly due to camera motion blur. Also, when the head is moved slowly, there is approximately one frame (1/60 s = 17 ms) of lag, causing the gauges to be slightly misaligned with the tags.

GEOMETRIC VERSUS GREEN-SCREEN BASED OVERLAY

As described in section "AR Flight Simulator Concept," AR flight simulator imagery can be accomplished using a geometric model or green-screen detection. This section shows HMD images from the AR simulator for these two techniques.

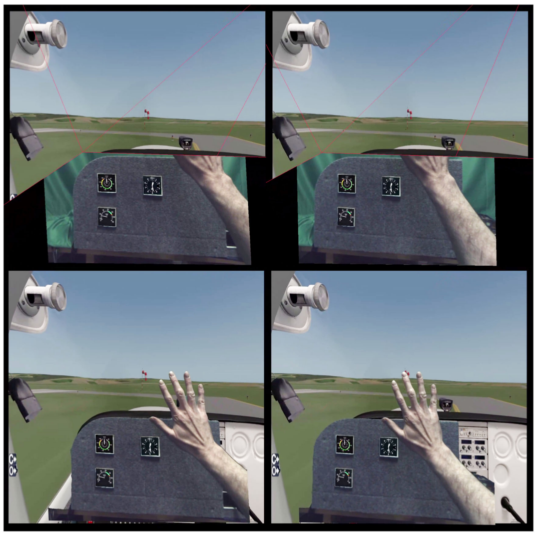


Figure 12.
Forward simulator view for geometry-based (top) and green-screen (bottom) overlay. Left and right panes show images sent to the left and right eyes with the HMD. In geometry-based overlay, the hand disappears behind window scenery, whereas green-screen overlay produces the correct result.

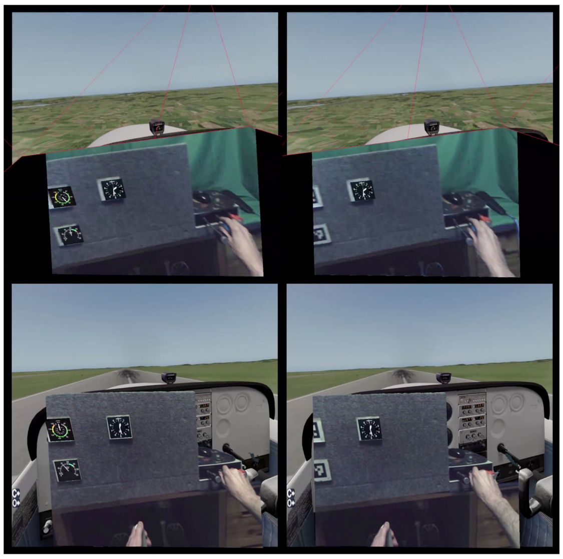


Figure 13.
AR simulator view when taking a quick glance to ensure the right hand is on the throttle control for geometry-based (top) and green-screen (bottom) overlay. Left and right panes show images sent to the left and right eyes with the HMD.

Figure 12 illustrates the basic forward view of the simulator for geometry-based (top) and green-screen (bottom) overlay. The example illustrates how geometric overlay requires a more complete physical cockpit than green-screen detection for good aesthetics. Also, whereas geometric overlay "chops off" the user's hand when placed over scenery display areas, green-screen detection allows the hand to show over correctly.

Figure 13 illustrates using the throttle with the right hand for the two overlay methods. In each case, a quick glance is all that is required to verify the right hand is touching the correct control, after which, the user can look up and actuate the throttle by feel alone.

AR PROCESS TIMING

In the AR Process code, time stamps are stored at key locations to allow for detailed profiling of the code. Table 2 shows CPU time required for each thread for a continuous 1-h and 10-min run of the AR simulator with green-screen detection enabled. Note that the CPU has eight cores, allowing all threads to run in parallel. The column "Time/Frame" indicates how much time on average is used on a CPU core to process one frame. The column "Max FPS" is the reciprocal of the first column, quantifying the maximum FPS that could be supported by that thread. Finally, "FPS" gives the average frames per second that were actually processed by the thread, which is close to the theoretical 60 FPS supported by the stereo camera input.

Table 2.

AR Process Timing

Thread	Time/Frame	Max FPS	FPS
X11 Grab	9.7 ms	102.7	58.6
Capture	0.3 ms	2944.4	58.5
ArUco Det. Left	14.4 ms	69.6	58.5
ArUco Det. Right	14.4 ms	69.4	58.5
Render	8.6 ms	116.4	58.5

This timing can also be used to estimate latency from input to output. The ArUco detection threads are very CPU intensive and are therefore run independently of the other threads to avoid impacting latency. Camera input to rendered HMD output latency is approximately 9 ms (0.3 ms + 8.6 ms), which is less than one frame period at 60 FPS (17 ms). Latency of the flight-simulator scenery from its generation in X-Plane to HMD output is about 18 ms (9.7 ms + 8.6 ms), which is significantly longer, mainly due to the simple CPU-based frame copy method. It is expected that if a GPU-only based method can be developed, grab time per frame could be reduced below 1 ms.

CONCLUSION

In this article, we described an AR flight simulator that provides the immersion of VR, but without sacrificing tactile interaction with physical controls. It was shown that conventional PC hardware can implement the required image processing functions at roughly 60 FPS, where the limitation was the input camera frame rate. Although the AR concept does place some constraints on the cockpit layout, we showed how a generic instrument panel could be used with movable instruments to easily accommodate different configurations. In general, we expect that AR-based cockpits could have much simpler and cheaper instruments than traditional enclosed-cockpit and wrap-around display simulators that must provide complete physical instrumentation.

This article presented our basic prototype AR simulator, but we envision many enhancements to this concept, including the following.

1) Combination of geometry and green-screen based methods to provide a completely immersive cockpit.

2) Autoregistration of the flight-simulator 3-D cockpit with the VR coordinate system using AR tags. The current system requires keyboard input for manual alignment.

3) Faster grabbing of flight simulator scenery using a GPU-only method.

4) Tracking of tag-based instruments to avoid lag.

5) Full time-stamping of stream data throughout the system to allow better synchronization and lower jitter.

6) Stereo capture with a higher resolution, higher field of view camera to match headset capabilities.

7) Movable gauge adjustment dials.

8) Dual headset setup for training purposes.

Although we feel that the AR flight simulator concept is promising and provides an immersive experience that is more natural than pure VR, it is unclear how it compares to wrap-around display cockpits and VR-only simulators in terms of the user experience and suitability for flight training activities. Answering this question will require research on the effectiveness of the AR method across a wide range of users and use cases, which we expect to explore in future work.

ACKNOWLEDGMENTS

This research would not have been possible without the many students, parents, and advisors in the N85 Aerospace Club, who from 2015-2019 contributed valuable time and support to the “dynamite room” (immersive flight simulator), which helped inspire the AR flight simulator concept. We especially thank Linda Castner, the club founder, for involving students and faculty from Lafayette College in that effort. We also acknowledge students Shawn Sheng and Nick Brunetti for introducing club members to VR-based flight simulation, which was the foundation of this research.

REFERENCES

[1] *Redbird Landing*. [Online]. Available: https://simulators.redbirdflight.com/products

[2] I. Yavrucuk, E. Kubali, and O. Tarimci, “A low cost flight simulator using virtual reality tools,” *IEEE Aerosp. Electron. Syst. Mag.*, vol. 26, no. 4, pp. 10–14, Apr. 2011.

[3] T. Aslandere, D. Dreyer, and F. Pankratz, “Virtual hand-button interaction in a generic virtual reality flight simulator,” in *Proc. IEEE Aerosp. Conf.*, 2015, pp. 1–8.

[4] G. R. Postal, W. Pavan, and R. Rieder, “A virtual environment for drone pilot training using VR devices,” in *Proc. XVIII Symp. Virtual Augmented Real.*, Gramado, Brazil, Jun. 21–24, 2016, pp. 183–187.

[5] E. Pennington, R. Hafer, E. Nistler, T. Seech, and C. Tossell, “Integration of advanced technology in initial flight training,” in *Proc. Syst. Inf. Eng. Design Symp.*, Charlottesville, VA, USA, Apr. 26, 2019, pp. 1–5.

[6] T. P. Caudell and D. W. Mizell, “Augmented reality: An application of heads-up display technology to manual manufacturing processes,” in *Proc. 25th Hawaii Int. Conf. Syst. Sci.*, Kauai, HI, USA, 1992, vol. 2, pp. 659–669.

[7] X. Wang, S. K. Ong, and A. Y. Nee, “A comprehensive survey of augmented reality assembly research,” *Adv. Manuf.*, vol. 4, pp. 1–22, 2016.

[8] B. Nuernberger, R. Tapella, S. Berndt, S. Y. Kim, and S. Samochina, “Under water to outer space: Augmented reality for astronauts and beyond,” *IEEE Comput. Graph. Appl.*, vol. 40, no. 1, pp. 82–89, Jan./Feb. 2020.

[9] *XWallace*. [Online]. Available: http://www.jonwallace.org