Software Framework for Mixed-Reality Multiplayer Quadcopter Gaming

Gabriel Simmons[1], Naveen Gowrishankar[2], Nelson Max, Zhaodan Kong

Abstract— **Augmented reality is seeing increasing use in video gaming and entertainment. Consumer quadcopters have likewise experienced a surge in popularity. To date, few systems have been proposed to combine these technologies into an integrated gaming experience. We propose a multi-player augmented reality quadcopter-based video game, in which the players pilot real drones from a first-person perspective while wearing a VR headset. The user sees the real environment overlaid with AR game elements through the window of a virtual cockpit, as if they are piloting the quadcopter from inside. Our game system can be utilized as a platform to implement a variety of game types.**

I. Introduction

A. Motivation

Augmented Reality (AR) is experiencing increased use in entertainment applications, as evidenced by mobile applications and games like Snapchat[1] and Pokemon Go [2]. Virtual reality (VR) technologies such as the Oculus Rift[3], HTC Vive [4], and PS-VR [5] have become available at competitive prices, allowing consumers to enjoy engaging virtual experiences and developers and artists to create software and entertainment experiences for these devices.

AR, unlike VR, allows the users perspective to be firmly anchored in a real world coordinate frame while being able to access or control visual information from a overlaid virtual frame that appears in fixed registration with the real world frame as the user moves or turns his/her head. AR requires the use of sensors and software working together to create this immersive effect. AR can also be combined with many other existing fields and technologies where a mixed perspective reality is useful, or can provide an engaging experience for the user such as in the medical industry where AR and MR tools are being developed [6] for use in surgery, patient information services and medical education.

Consumer quadcopters have seen significant advances in the past decade, both in popularity and functionality. Today's smart quadcopters provide a number of advanced features. These include multiple varieties of autopilot capability, in which the quadcopter performs actions such as hovering in the same location, or navigating to user-specified waypoints automatically. Today's consumer quadcopters also provide integrated camera support, including object following and other flight modes to facilitate cinematography.

The combination of these technologies offers the potential for exciting new gaming experiences, though relatively few existing products have attempted to take advantage of this potential. This paper proposes a multi-player game system, in which users pilot real quadcopters "from the inside", viewing the real environment overlaid with AR game elements through the virtual cockpit window.

This paper first explains the various hardware and software components of the system, and how they interact to produce the complete game platform, in the section entitled *System Summary*. We then discuss the contributions of this work in the areas of *Localization*, *Game Design*, and *Quadcopter Control*. Finally, we include some preliminary results obtained with this software framework, as well as a discussion of these results and their potential applications and future improvements.

Hardware Item	Quantity
3DR Solo quadcopter	1 per player
GoPro HERO4 camera	1 per player
Oculus Rift VR headset and touch controllers	1 per player
HDMI capture card	2 per player
HDMI splitter	1 per player
VR-ready Windows 10 workstation (required to run Unity and generate stereo video for the Oculus headset)	1 per player
VR-ready Linux workstation running Ubuntu 16.04LTS (required for ROS)	1
Local wired or wireless network	1

TABLE I
System hardware requirements

II. System Summary

A. Gameplay Environment

This system is designed for use in an open indoor environment, with dimensions approximately 30 feet long by 20 feet wide by 15 feet high.

B. Hardware Elements

The hardware elements used in this project are listed in Table II-B.

The 3DR Solo is a well built, durable quadcopter capable of withstanding hard landings, which is perfect for this use case as a gaming device for novice pilots. The quadcopter weighs about 1.7 kg with an attached GoPro HERO4 camera on a fixed gimbal. A fixed gimbal was deemed to be necessary to provide a true mixed reality experience with the user having visual feedback of the drone's movements when the drone pitches or rolls to perform its maneuvers. The 3DR Solo was designed to support the GoPro Hero 4 camera.

The Oculus VR headset, although primarily used for virtual reality applications, was the best candidate for a headset capable of rendering both the video from the drone

[1]G. Simmons thanks... `gsimmons at ucdavis.edu`
[2]N. Gowrishankar thanks... `naveen email`

and layering the virtual frame of reference containing the game objects onto the real world video view via the Unity graphics engine. The Oculus VR system has an extensive software framework for application developers to take advantage of this engine. The Oculus headset also comes with a pair of touch controllers (joysticks) that are used by the player to navigate the mixed reality landscape by controlling the movement of the Solo. The Oculus SDK is limited to supporting one headset per workstation. As a result one Windows workstation is required per player.

1) Hardware Connections: Necessary hardware connections for a two-player instance of the game system are depicted in Figure 1. One Linux workstation is required. Video is captured from the GoPro camera on-board each Solo, and streamed wirelessly from the Solo quadcopter to the corresponding Solo controller using the Solo's built-in transmission system. The Linux workstation receives these videos from each quadcopter via an external HDMI capture card between the Solo Controller HDMI-out and a USB port on the workstation.

One Windows 10 workstation is required for each player. The Oculus hardware for the player is connected to this workstation via HDMI (headset) and USB (headset and controllers). The Windows workstations also receive the video from their corresponding quadcopters, via an HDMI splitter and an added internal video capture card.

All workstations must be connected to the same local area network, by Ethernet or WiFi. All communication between the Linux workstation and the Windows workstations is mediated by *rosbridge* (described in *Software Elements*). Communication between Windows workstations is mediated by Unity's Network Manager utility.

C. Software Elements

The key software elements of this project include

- Robot Operating System (ROS)
  - rosbridge
  - MAVROS

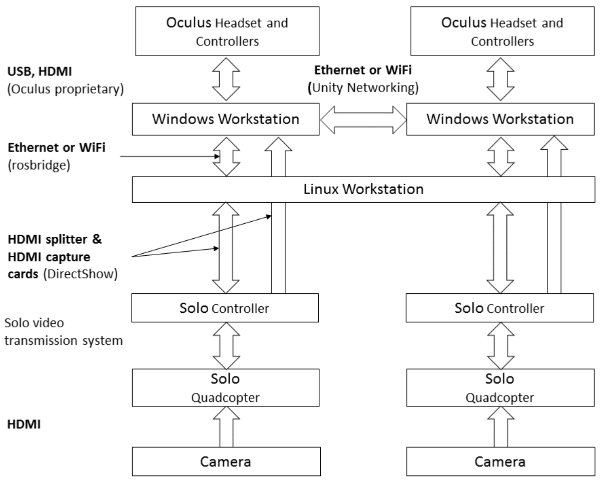

Fig. 1. Hardware and connections

- Unity game engine (one instance per player running on a VR-ready Windows 10 workstation)
- ORB-SLAM2 localization algorithm (one instance per quadcopter running on the Linux workstation)

The quadcopter control system relies heavily on the Robot Operating System (ROS). ROS is a software framework for Linux that allows for the modular development of robotics applications. Two ROS libraries of particular importance are MAVROS and rosbridge. MAVROS is a ROS package that provides a ROS interface to the MAVLink serial communication protocol. MAVLink is a standard protocol used for communication with quadcopters and other small unmanned vehicles. This communication protocol is supported by the ArduCopter firmware running on the Solo. Rosbridge is a software tool that provides a JSON interface to ROS. This software facilitates communication between ROS and Unity, necessary since ROS is available only on Linux, and Unity and the Oculus are not yet officially supported on Linux. Rosbridge supports a variety of transport layers, including WebSockets and TCP.

The gaming elements of the system are handled by the Unity game engine. Unity is responsible for game physics (simulating interactions among players and in-game objects) and rendering the view of the game for each user. The Unity game engine is one of the most popular platforms for game development.

The localization of the quadcopters is achieved by the ORB-SLAM2 algorithm, a state of the art algorithm for vehicular localization. One instance of ORB-SLAM2 is required for each quadcopter.

D. Roles and interactions of system components

Conceptual components of the system and information flow between these components is shown in Figure 2.

1) User Input: In any game, the user will typically be required to move their quadcopter within the game space to achieve some objective (i.e. maneuvering around obstacles, following another quadcopter, reaching a goal or target). Additionally, users may be required to perform other in-game actions such as firing at a virtual target, activating a power-up, etc. To perform an in-game action or manipulate the quadcopter, the user indicates their desired action by manipulating the Oculus touch controllers according to a specified control scheme as shown in Figure 3. For instance, the user may desire to move their quadcopter to the right, so they would move the joystick of the left hand touch controller to the right, since this joystick corresponds to lateral motion. This information is collected from the player's Windows workstation and forwarded to the Linux workstation for the corresponding player via rosbridge.

2) Waypoint Mapping: The controller information is used as input to the waypoint mapping system block. This conceptual element of the game system is currently implemented in ROS. (However, the logic for this system block will likely be migrated to Unity with the addition of collision avoidance and virtual object interaction described below.) The waypoint mapping module takes in the raw joystick data and translates the data to a waypoint relative to the quadcopter. See section V-B.2 for details on this translation.

This local frame waypoint is then handled by the waypoint modification system block.

3) Waypoint Modification: We use this method of waypoint mapping rather than direct velocity commands because it facilitates obstacle avoidance and the simulation of in-game physics on the real drone. These features are not yet implemented, but we propose their implementation in the waypoint modification system block. The logic for this system block would reside within the Unity game.

Each waypoint can be compared ahead of time with the known locations of physical obstacles such as the enclosure wall. The collision avoidance and virtual object interaction module modifies the waypoint generated by the waypoint mapping module to prevent collisions or to simulate the effects of in-game events. Our proposed implementation for collision avoidance and virtual object interaction is discussed further in *Quadcopter Control*.

4) Quadcopter and ORB-SLAM2: ORB-SLAM2 is a simultaneous localization and mapping algorithm for vehicles that takes in mono or stereo video frames and produces an estimate of the pose (location and orientation) of the vehicle, as well as continuously building a map of the vehicle surroundings. The Solo sends its video stream to the corresponding instance of ORB-SLAM2 running on the Linux workstation. The localization algorithm returns the pose estimate of the quadcopter. The quadcopter fuses this information with other sensor data using an Extended Kalman Filter running on its internal hardware to generate an overall estimate of its position. Equipped with an estimate of its current position, the Solo can navigate to waypoints automatically using its internal autopilot firmware (ArduCopter).

5) Unity: Unity also receives the video stream from each quadcopter. After receiving the video, Unity corrects the stream for barrel distortion, overlays virtual elements, and displays the view of the game environment to the user on the Oculus headset. Unity could potentially also be used for collision avoidance. Each instance of ORB-SLAM2 sends its position estimate to Unity, which updates the location of the corresponding virtual quadcopter. In-game events based on proximity to virtual objects are detected in Unity.

6) User Display: The user view of the game environment is generated by Unity, and consists of AR game elements overlaid on the video feed from the player's Solo quadcopter. See Figure 12 for an example of user view during a game.

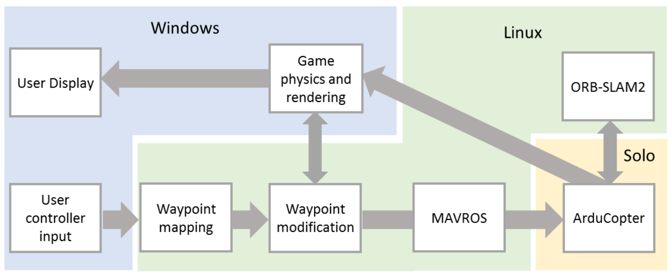

Fig. 2. Conceptual system block diagram

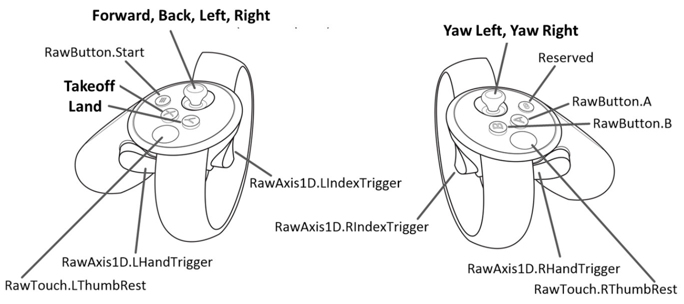

Fig. 3. Oculus touch controller button layout. Buttons not labeled with specific actions are used for in-game actions that vary between games.

III. Localization

A. Motivation and Background

To enable the quadcopter to navigate to the waypoints specified by the game system, the quadcopter must have an estimate of its current pose (position and orientation) in the local environment. The task of obtaining this estimate is termed *localization*. Most large consumer drones including the 3DR Solo are capable of outdoor localization using GPS as the primary sensor. As this system is intended for indoor use, we were required to find a localization method that functions indoors to replace GPS as the primary sensor.

B. Choosing a Localization System

To allow for simple and flexible system setup, we limited our consideration of localization methods to those only requiring on-board sensors. Several factors inform the choice of localization method. Affordability, as well as power consumption and weight increase associated with added hardware are important concerns. Since the game system requires the use of a camera to produce the user view of the game environment, localization via a monocular camera provided a localization solution that required no additional hardware, power consumption, cost, or setup time.

C. Choosing Localization Algorithm

1) Integrated AR and localization libraries: Initially, we hoped to use a commercially available AR library that would be capable of both localization and augmented reality. We initially considered four commercially available AR libraries: ARToolkit, ARkit, Vuforia, and Wikitude. After comparing videos demonstrating the performance of these libraries online (such as [7]), ARToolkit and Vuforia appeared most promising, and we conducted our own comparison of these two libraries.

Library	Platform	Open Source	Unity Support
Vuforia	Desktop and Mobile	no	yes
ARToolkit	Desktop and Mobile	yes	yes
ARkit	Mobile	no	yes
Wikitude	Mobile	?	?

TABLE II

COMPARISON OF INTEGRATED AR AND LOCALIZATION LIBRARIES

Library	Computer	Camera	Distance	Static Object	Moving Object	Comment
Vuforia	Laptop	Built-in Webcam	1m	stable	stable	
			>1m	not detected	not detected	
	VR-ready computer	GoPro	various	not stable	not stable	w/o camera calibration
ARToolkit	Laptop	Built-in Webcam	1m	stable	stable	
			2m	stable	stable	
			3m	stable	sudden change of orientation	
	VR-ready computer	GoPro	2m	stable	stable	camera calibrated
			3m	sudden change of orientation, stable position	sudden change of orientation, stable position	

TABLE III
OUR COMPARISON OF ARTOOLKIT AND VUFORIA SOFTWARE PACKAGES

2) Dedicated SLAM libraries: As shown in Table III-C.1, neither Vuforia nor ARToolkit were stable at large enough distances to be suitable for our use. This led us to consider software libraries dedicated to SLAM only. While examining the algorithms compared in [8], we encountered SVO[9] and ORB-SLAM[10], two feature-based state-of-the-art SLAM algorithms. We also considered AruCo marker-based SLAM[11].

According to the authors of ORB-SLAM:

> ORB-SLAM is a versatile and accurate SLAM solution for Monocular, Stereo and RGB-D cameras. It is able to compute in real-time the camera trajectory and a sparse 3D reconstruction of the scene in a wide variety of environments, ranging from small hand-held sequences of a desk to a car driven around several city blocks. It is able to close large loops and perform global relocalisation in real-time and from wide baselines. It includes an automatic and robust initialization from planar and non-planar scenes.[10]

According to the authors of SVO:

SVO uses a semi-drect paradigm to estimate the 6-DOF motion of a camera system from both pixel intensities (direct) and features (without the necessity for time-consuming feature extraction and matching procedures), while achieving better accuracy by directly using the pixel intensities. [12]

In a qualitative comparison of the SVO, ORB-SLAM, and AruCo SLAM algorithms, it was clear that ORB-SLAM provided the highest degree of pose stability.

- Marker-based AruCo
- ORB-SLAM v2

D. *Sensor Fusion/Localization Utilization by Solo*

Localization information is sent to the Solo via MAVROS, a ROS package that wraps the MAVLink standard communication protocol for UAVs. The Solo uses an onboard Extended Kalman Filter to combine this information with information from other sensors, including an IMU and barometer, to produce a combined estimate of its current pose. The ability to use external localization data as a substitute for GPS as the primary localization sensor is a relatively new feature in the ArduCopter firmware. To take advantage of this feature, the Solo quadcopters were updated from their stock firmare to ArduCopter master version *3.6-dev*. This firmware update required upgraded hardware - namely the “Green Cube” Pixhawk 2 flight controller.

E. *Modifications to the ORB-SLAM algorithm*

F. *Unity/ROS interface*

The combined pose estimate produced by the Solo quadcopter is used to update the position of the virtual quadcopter in the game environment in Unity. This is done using rosbridge, a utility that provides a JSON interface to ROS, allowing non-ROS programs to interact with ROS [13].

IV. Game Design and User Display

A. *Distortion Correction*

1) Motivation and Background: The wide field of view of the GoPro camera naturally imparts barrel distortion on the captured image. Barrel distortion is an optical distortion in which straight lines appear curved, as shown in Figure 5. To create a more realistic view for the user, it was necessary to compensate for this distortion.

2) Methods: Two methods were considered, color sampling, and pincushion distortion. Both methods of distortion correction require the calibration parameters for the GoPro Camera.

Camera Calibration To determine the calibration parameters for our camera, we used the a set of utility functions provided in OpenCV. A tutorial for this procedure is available at [14]. Calibration parameters are defined as follows:

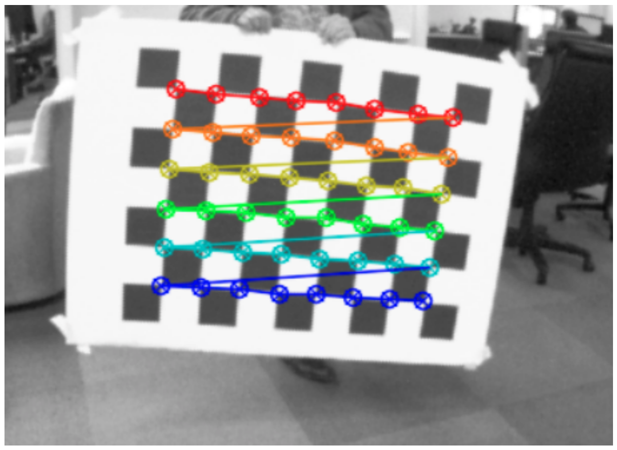

Fig. 4. Camera calibration using OpenCV utility

k_1	first coefficient for the radial factor
k_2	second coefficient for the radial factor
k_3	third coefficient for the radial factor
p_1	first coefficient for the tangential factor
p_2	second coefficient for the tangential factor
f_x	x component of the focal length in pixel coordinates
f_y	y component of the focal length in pixel coordinates
c_x	x component of the optical center expressed in pixel coordinates
c_y	y component of the optical center expressed in pixel coordinates

f_x, f_y, c_x, andc_y are intrinsic parameters of a camera. With a camera of resolution of 1920x1080, (c_x,c_y) should be approximately (960, 540).

Using the OpenCV utility, we determined the calibration parameters for our GoPro Hero 4 to be the values listed below.

k_1 = -2.44450627e-01
k_2 = 7.19937237e-02
k_3= -1.02982879e-02
p_1 = 5.14066681e-04
p_2 = 2.35650087e-04
f_x = 885.94368422
c_x = 948.48891401
f_y = 896.37970921
c_y = 515.39652463

k_1, k_2, and k_3 are the coefficients for the radial factor of the distortion. The negative sign of k_1 implies that the images produced by this camera will have barrel distortion.

p_1 and p_2 are the coefficients for the tangential factor of the distortion.

Method 1: Pincushion Distortion Pincushion distortion uses the calibration parameters to define the camera's applied barrel distortion, then applies the inverse distortion to recover the undistorted image. Distortion consists of a radial and tangential component, with both tangential and radial distortion defined in the x and y direction as follows:

$$radial_x(x, r, k_1, k_2, k_3) = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$radial_y(y, r, k_1, k_2, k_3) = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$tangential_x(x, y, r, p_1, p_2) = 2p_1 xy + p_2(r^2 + 2x^2)$$

$$tangential_y(x, y, r, p_1, p_2) = p_1(r^2 + 2y^2) + 2p_2 xy$$

The pincushion distortion algorithm can be described as follows:

Given the intrinsic camera parameters $k_1, k_2, k_3, p_1, p_2, f_x, f_y, c_x, c_y, f'_x, f'_y, c'_x, c'_y$ and the pixel at coordinate (X, Y) in the original distorted image, pincushion distortion attempts to find the new location $(X_{corrected}, Y_{corrected})$ where the pixel should be placed in the corrected image.

We find normalized coordinate (x, y) from (X, Y) by subtracting the optical centers and dividing by the focal length in each direction.

$$x = (X - c_x)/f_x$$

$$y = (Y - c_y)/f_y$$

$$r^2 = x^2 + y^2$$

We then perform the inverse distortion on the normalized coordinate, to obtain the corrected normalized coordinate $(x_{corrected}, y_{corrected})$.

$$x_{corrected} = radial_x(x, r, k'_1, k'_2, k'_3) + tangential_x(x, y, r, p'_1, p'_2)$$

$$y_{corrected} = radial_y(y, r, k'_1, k'_2, k'_3) + tangential_y(x, y, r, p'_1, p'_2)$$

The parameters for inverse distortion were calculated using the following formula, from [15]. The resulting coefficients k'_1, k'_2, and k'_3 will provide the pincushion distortion to correct the barrel distortion and recover the undistorted image.

$$k'_1 = -k_1$$

$$k'_2 = 3k_1^2 - k_2$$

$$k'_3 = -12k_1^3 + 8k_1 k_2 - k_3$$

Since we did not experience significant tangential distortion, coefficients p'_1 and p'_2 were set to 0.

We then convert this coordinate back to pixel coordinates by multiplying by the new focal lengths and adding the new optical centers. $(X_{corrected}, Y_{corrected})$ will be the location for pixel (X, Y) in the corrected image.

$$X_{corrected} = x_{corrected} * f'_x + c'_x$$

$$Y_{corrected} = y_{corrected} * f'_y + c'_y$$

The new focal lengths and optical centers f'_x, f'_y, c'_x, c'_y are used to scale and recenter the corrected image.

For an output image of the same size as the input, with the same center, f'_x, f'_y, c'_x, c'_y are equal to f_x, f_y, c_x, c_y.

Method 2: Color Sampling Color sampling finds the color for a pixel at coordinates (u, v) in the corrected image (output), by performing a barrel distortion on this corrected image coordinate. The barrel distortion is an image transformation defined by the calibration parameters of the camera. The color of the pixel at the corresponding coordinate (u, v) in the distorted image is sampled, and we then use this color for the coordinate (u, v) in the corrected image.

Given (u, v)

$$x = (u - c'_x)/f'_x$$

$$y = (v - c'_y)/fy'$$

$$r^2 = x^2 + y^2$$

$$x_{distorted} = radial_x(x, r, k_1, k_2, k_3) + tangential_x(x, y, r, p_1, p_2)$$

$$y_{distorted} = radial_y(y, r, k_1, k_2, k_3) + tangential_y(x, y, r, p_1, p_2)$$

$$X_{distorted} = x_{distorted} * f_x + c_x$$

$$Y_{distorted} = y_{distorted} * f_y + c_y$$

$(X_{distorted}, Y_{distorted})$ will be the coordinate that you want to sample the color from in the distorted image.

3) Results: Qualitative results from both methods are shown in Figure 5. Since both methods were successful, and produced largely indistinguishable results, the color sampling method was chosen due to its slight advantage in ease of implementation. The algorithm for this method was implemented in the Unity shader.

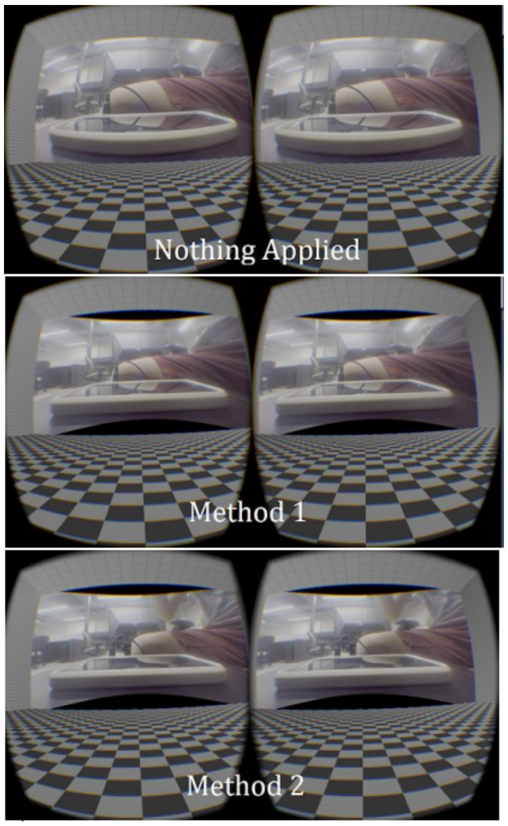

Fig. 5. Distortion correction results using Method 1 (pincushion distortion) and Method 2 (color sampling)

B. DirectShow and Unity Integration

To create an augmented reality game environment in the user display, Unity must render the augmented reality game objects on top of the video feed obtained from the GoPro camera. This video feed is streamed wirelessly from the Solo quadcopter to the Solo controller, and it can be viewed using an HDMI output on the controller. To access this video from Unity, we connected the Solo controller to the corresponding workstation using a PCIe video capture card. No readily-available solution existed to read this video data directly into Unity, so a custom driver was implemented to read data from the capture card.

We used DirectShow, a Windows media library, to read the raw data from the video capture card. A dynamic library was written to access the video frames using C++. This .dll file was then used in C scripts in Unity to access the video frames.

C. Game object rendering

3D models for all in-game objects were created in Autodesk Maya. Objects created for our example games include

- cockpit
- ball
- maze structure

D. Augmented Reality Rendering in Unity

The AR game elements were superimposed on the camera view using the depth and culling mask attributes of the game's Unity camera objects.

In Unity, a camera's depth decides the order in which it will be rendered into display devices. The culling mask attribute limits the objects to be showed in the camera. For example, if we set some objects in the scene to belong to the "virtual" layer, and configure a camera so that its culling mask only includes the "virtual" layer, then this camera will only show those objects that belong to the "virtual" layer

Using these properties of Unity cameras, we can configure our scene so that the AR game objects will always be rendered "on top" of the real-world view obtained from the GoPro camera. To achieve this, we first create two layers, *real* and *virtual*. We then configure two Unity cameras for each player. The video feed from the GoPro is placed in the real world layer, and the rest of the scene (all other in-game objects) are placed in the virtual layer. The "real camera" for each player is given depth -1, and a culling mask which only includes the real world layer. The "virtual camera" is given depth 0, and a culling mask which only includes the virtual layer.

Thus, the real world view is rendered under our virtual objects.

E. Games implemented

Two games have been implemented in Unity.

1) 3D Pong: The first is a 3D version of the classic video game Pong. In this game, two players use their quadcopters as "paddles" to manipulate a virtual ball. Similar to table tennis, the goal of the game is to strike the ball in such a way that the opponent is unable to return the ball. This game demonstrates the multiplayer capabilities of the platform, as well as interaction with dynamic in-game virtual objects.

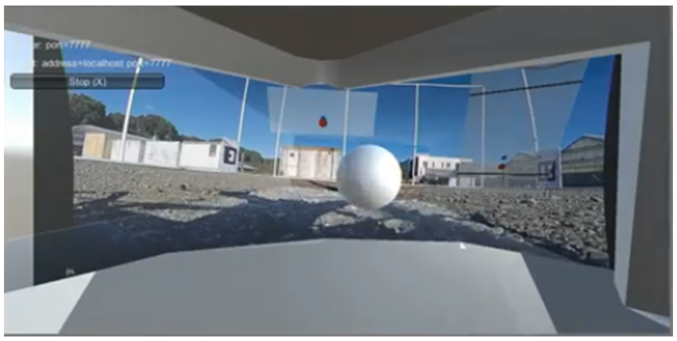

Fig. 6. User view of pong game

2) Maze: The second game is a 3D maze navigation game. The goal of this game is simple, to navigate from the starting point to the ending point of a virtual maze. This game demonstrates real-world feedback of interaction with virtual objects by stopping the quadcopter when it collides with the maze walls.

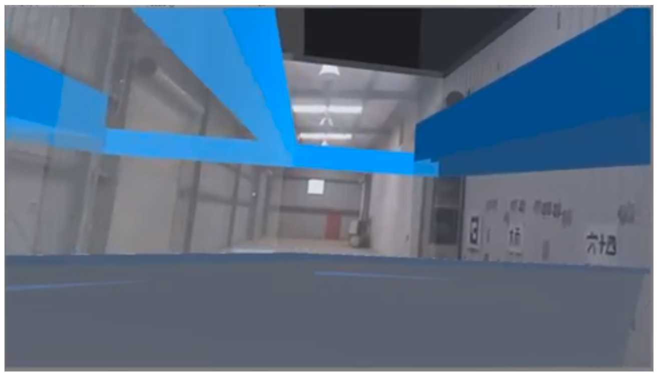

Fig. 7. User view of maze game

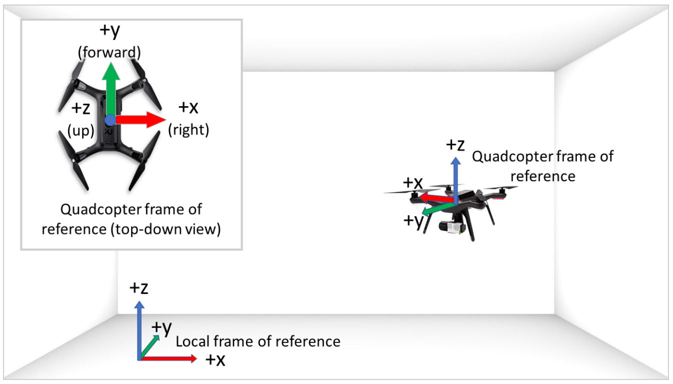
+y
(forward)
+z
(up)
+x
(right)
Quadcopter frame of reference (top-down view)
+z
Quadcopter frame of reference
+x
+y
+z
+y
Local frame of reference
+x

Fig. 8. Quadcopter and local frames of reference

V. Quadcopter Control

A. Motivation

Controlling the Solo in an indoor environment presents several challenges. Manual flight using the stock Solo controller takes several hours of practice to master and is not intuitive for the first-time flyer. In manual flight, the user exerts low-level control over the quadcopter: to maintain a constant position (hover), the user must constantly be correcting for drift and disturbances. The Solo offers multiple other flight modes that make use of the internal proportional-integral-derivative (PID) controller. However, these flight modes are not directly utilizable using a video game controller instead of the stock Solo controller. Additionally, indoor flight using the stock controller is unsafe, as the Solo does not provide any safeguards against collision. We desired a control system for the Solo that would satisfy the following three qualities:

- *Safety* Since most large consumer drones like the Solo are intended for outdoor flight, these platforms do not provide any safeguards against collisions. Since our game is intended for indoor use by multiple players, we desired a control system that prevents the user from colliding with the game enclosure and other quadcopters.
- *Ease of Flight* We desired that the user experience be focused on gameplay, rather than learning the controls of the game. To make controlling the quadcopter easier, we desired a control layout (button mapping) that would be intuitive and familiar to video gamers.
- *Immersion* To enhance the immersive link between the virtual and real game environments, we desired that the effects of in-game events could be simulated on the real drone. For instance, if a player experiences an in-game collision with a wall in the virtual environment, their quadcopter should also be stopped in the real environment. This type of control would be more difficult to implement using the stock Solo controller.

We have implemented a control system for the quadcopter using MAVROS that allows for movement of the quadcopter using joystick inputs familiar to gamers. This system (described below) acts as a layer on top of the Solo's internal autopilot firmware. The GUIDED mode of the autopilot firmware is used for flight, while TAKEOFF and LAND mode are used for landing. The control system can be subdivided into three conceptual components: user input, waypoint mapping, and waypoint modification.

B. Basic Movement: User input and waypoint mapping

1) User input: User input is captured from the Oculus touch controllers on each Windows workstation and forwarded to the Linux workstation via rosbridge. The controller information is represented as two vectors, one corresponding to the state of the buttons, and another corresponding to the state of the joysticks. The state of the buttons is indicated as 0 (unpressed) or 1 (pressed). The state of the joystick is indicated as a pair of real values between -1 and 1, each value corresponding to one axis of the joystick.

2) Waypoint mapping: Two frames of reference are considered in the control system: the quadcopter frame (fixed with respect to the Solo) and the local frame (fixed with respect to the game enclosure). Both of these frames of reference are illustrated in Figure 8. We first map the joystick data to a waypoint in the quadcopter frame. We allow for control of the quadcopter along three translational axes (forwards/backwards, up/down, left/right) as well as one rotational axis (yaw). Each of these axes is mapped to an axis of joystick input on the Oculus touch controllers (see Fig. 3).

Considering the current quadcopter pose

$$x = [location_x,\ location_y,\ location_z,\ yaw]^T$$

and joystick input

$$j = [input_x,\ input_y,\ input_z,\ input_{yaw}]^T$$

we generate the waypoint in the quadcopter frame x_q by the following:

$$x_q = x + \alpha j$$

where α is a user-defined diagonal matrix of input sensitivities. Larger values in the diagonal entries of α correspond to a waypoint that is further in the corresponding direction for the same input.

A coordinate transformation is applied to x_q to obtain the waypoint coordinates x_l in the local frame of reference as follows

$$x_l = Tx_q$$

where T is the matrix representing the transformation from the quadcopter frame of reference to the local frame. There are two reasons for this transformation. The waypoints are communicated to the quadcopter using MAVROS, and MAVROS requires that these coordinates be in the local frame of reference. Additionally, the waypoint locations in the local frame will be required for the features described in *Collision Avoidance*. In practice this transformation is implemented using the ROS tf tool, a utility in ROS that automatically tracks the transformations between coordinate frames centered on various objects.

Because the quadcopter uses PID control to correct the error between its current pose and the waypoint, the quadcopter moves faster towards a waypoint that is further away. The waypoint is continuously updated, so that continuous joystick input results in movement at a constant velocity. For the user, this provides the feeling of direct velocity control. Greater input magnitude (moving the joystick further) results in greater velocity in the corresponding direction.

C. Waypoint Modification: Collision avoidance and simulation of in-game events

We propose modification of the waypoints produced by the above method for two primary reasons: collision avoidance and simulation of in-game events on the real quadcopter.

1) Collision Avoidance: We first designate a ”safety radius” around the quadcopter. The safety radius is equal to the maximum physical radius of the quadcopter itself, plus an additional length that represents the quadcopter stopping distance when travelling at its maximum allowed speed. This maximum speed is an arbitrary limit set by the game developers for a particular game.

We suggest that the safety radius be equal to the quadcopter size plus the stopping distance of the quadcopter at its maximum allowed speed, as opposed to the stopping distance at its current speed. This is done for two reasons. First, it simplifies the calculations necessary for collision avoidance by maintaining a safety radius of fixed size. Second, it is a conservative approach that is less likely to result in crashes due to inaccuracy in the quadcopter pose estimates or other errors.

These safety radii can be represented as spheres in Unity, and the intersection of these spheres can be used to check for and predicts collisions. The safety radii would remain hidden from the user view of the game.

Quadcopter-Object interaction The simplest case of collision avoidance is that of a potential collision between a quadcopter and a stationary object, such as a wall. This scenario is illustrated in the top portion of Figure 9. As illustrated, the user has provided joystick input, and this joystick input has been mapped to a waypoint. The arrow represents the vector between the current quadcopter position and the waypoint, with the waypoint located at the arrow tip. We will call this vector the quadcopter *trajectory*.

In Unity, a sphere representing the safety radius around the waypoint is maintained, and its position is constantly updated to match the quadcopter's current waypoint. We predict the collision illustrated in Figure 9 by detecting that the waypoint safety radius intersects with the representation of the game enclosure in Unity.

We then reduce the magnitude of the quadcopter trajectory to avoid the collision. We label the point along the wall surface that intersects with the direction of the quadcopter trajectory as the *intersection*. We then find the location along the quadcopter trajectory that is closest to the intersection without causing the waypoint safety radius to actually overlap the obstacle. This point occurs where the safety radius is tangential to the surface of the obstacle. When a collision is predicted, the calculations to determine the modified waypoint will be performed in Unity, and the modified waypoint will be sent to ROS via rosbridge.

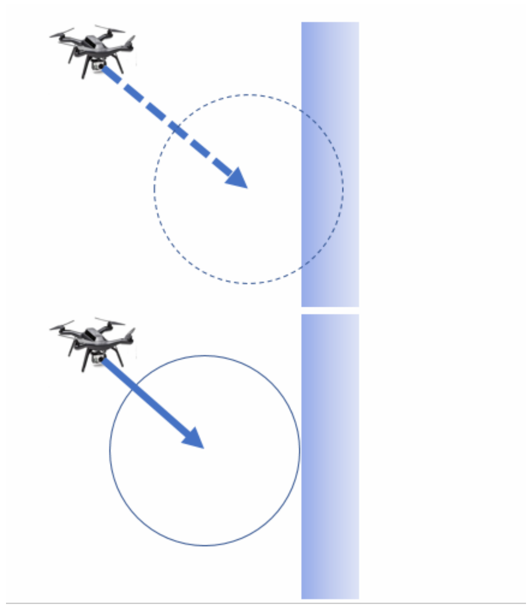

Fig. 9. Waypoint modification for collision avoidance between quadcopter and stationary object

Quadcopter-Quadcopter interaction Figure 10 shows the case of a potential collision between two quadcopters. In this case, one or both quadcopter trajectories must be modified such that the safety radii are no longer intersecting. Among the many possible approaches to this problem, we desired a simple solution that does not interfere excessively with the gameplay experience. To accomplish this, we multiply the length of each quadcopter trajectory by λ, the fraction of the distance that each quadcopter can travel between its current location and its specified waypoint where the safety spheres around each quadcopter first intersect. This fraction λ can be found by solving a quadratic equation.

2) Simulation of in-game events on real quadcopter: Our method of quadcopter control allows for simulation of the real quadcopter's reaction to in-game events. Stopping the quadcopter when it reaches a virtual wall can be accomplished directly using the collision prevention system de-

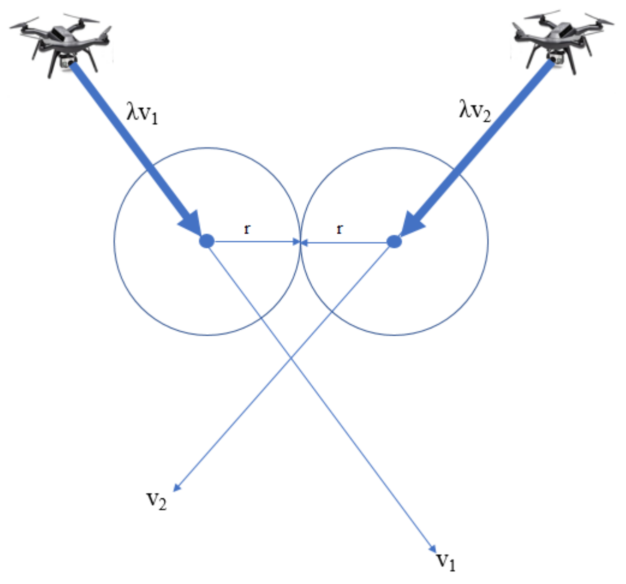

Fig. 10. Collision avoidance between two quadcopters. The original quadcopter trajectories v_1 and v_2 are scaled by the fraction λ to produce the modified waypoints.

scribed above. Many other types of effects can be potentially created using this system. We will describe two in particular: simulated crash (forced landing), and slowing.

Forced landing is the simpler of the two effects. It can be accomplished by switching the quadcopter from GUIDED to LAND mode via MAVROS. No waypoint modification is necessary, since waypoints are only condsidered in GUIDED mode. The potential risk of landing on top of another player's quadcopter can be mitigated by the collision prevention system. Since the landing quadcopter would ignore waypoints, priority would be given to the landing quadcopter in the event of a potential collison, and the free-flying quadcopter would have its waypoints modified.

Another useful form of real-world feedback for in-game events is slowing a player's quadcopter. In games such as racing or maze navigation, it is desirable to have areas of the game environment where the player suffers a penalty in speed for entering the area. In typical racing games, if a player goes off-track (through patch of grass or sand), the player's movements are slowed while they remain in this off-track area. This feature prevents cheating by cutting the track.

Slowing in all directions could be accomplished simply by scaling down the input sensitivities α. However it would be desirable to slow in only some directions in certain instances. For example, consider the case of friction against walls. In the case that a player is flying against a wall, they should receive a penalty to their movement speed to simulate friction with the wall. However, this penalty should not be applied if the player is moving away from the wall, otherwise the wall will feel "sticky".

Slowing in a particular direction can be accomplished by a mechanism similar to the one used for collision avoidance. We can define slowing zones as in-game objects in Unity, either visible or invisible to the player. Slowing is applied by adding a multiplier less than one to the quadcopter waypoint mapping. We apply slowing only if the quadcopter's current position is inside or on the boundary of the slowing zone. Furthermore, we can limit slowing to only be applied when the quadcopter trajectory points towards the interior of the slowing zone, or is parallel to the slowing zone surface. This is useful in the case of friction against walls, so that movement away from the wall is not penalized (see Figure 11.)

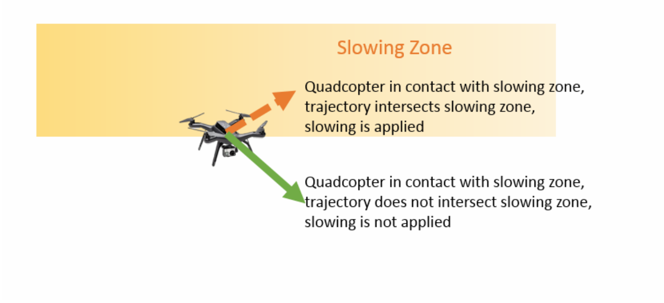
Slowing Zone
Quadcopter in contact with slowing zone, trajectory intersects slowing zone, slowing is applied
Quadcopter in contact with slowing zone, trajectory does not intersect slowing zone, slowing is not applied

Fig. 11. Waypoint modification to simulate friction with a virtual wall object

D. Control system limitations

The proposed control system has several limitations. One possible limitation of the waypoint modification method is that it may cause the quadcopter to decelerate too soon before reaching an obstacle. Some slowing before reaching an obstacle is necessary - it is impossible for the quadcopter to decelerate from some constant velocity to zero instantaneously. However, if deceleration as the quadcopter approaches an obstacle is too gradual and begins too soon, the obstacle will feel "spongy" to the player.

To minimize this effect, deceleration to avoid collision should only occur once distance between the quadcopter and the obstacle is equal to the quadcopter safety radius, and the quadcopter should maintain its current velocity when the distance from the waypoint is greater than or equal to the safety radius. Furthermore, a smaller safety radius is desirable. The quadcopter cannot go from forward movement to stopping instantaneously, so this issue will always be present. However, whether the deceleration will be fast or slow will determine the "sharpness" of the gameplay experience.

Another current limitation of the quadcopter control algorithm is that it does not model elastic collisions. When the player quadcopter approaches an obstacle, the waypoint is adjusted such that the quadcopter stops just short of the obstacle. However, we do not implement any "rebound" effect so that the player quadcopter bounces back from the obstacle. Similarly, the proposed system for avoiding collisions between two quadcopters will result in collisions where (if the user input is unchanged) the quadcopters will come to rest at a minimum distance of two safety radii away from each other without rebounding away from each other. Depending on the game context, a rebounding feature may better match what players would expect based on real-world physics and the physics of other video games. Implementing this feature is possible through further waypoint modification, and we intend to experiment with this after the basic collision avoidance system has been demonstrated to work.

The safety radius size also presents a potential limitation. We propose that the safety radius be based on the stopping distance of the quadcopter at its maximum speed. This provides the benefits of simplicity and added safety as described above. However, it does also limit the game dynamics, since the collision avoidance system will restrict the movements of slow-moving quadcopters as if they were moving much faster. In other words, generally larger safety radii will produce a heavy-handed collision avoidance system that restricts quadcopter movement to a greater degree. Once the proposed collision avoidance system is validated for safety, we plan to gradually increase the amount by which the safety radii scale with the quadcopters' current velocities.

Another potential issue is latency between control input and quadcopter movement, as well as between in-game events and real-world feedback.

VI. Results

Each aspect of the game system has produced promising results individually. The localization system produces pose estimates that are accurate enough to play a simple 1-player pong game, as well as a maze navigation game using the stock 3DR Solo controller. However, pose estimates are not always smooth, which makes in-game objects appear as if they are jumping or shifting quickly in position with respect to the real-world video. This will be a much more significant problem in later games with a more developed AR environment. Stationary overlays on real-world objects will appear to be moving, which would create a disorienting effect for the user. Nonetheless, it is currently possible for skilled flyers to hit the ball (using the stock 3DR Solo controllers), and interaction with the ball appears generally as expected.

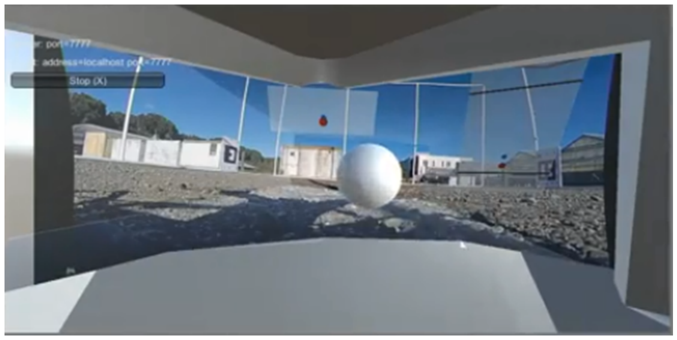

Fig. 12. User view of pong game

Basic movement using the quadcopter control system has also shown promising results, although the collision avoidance system is not yet implemented. A novice pilot is able to reliably takeoff, land, and maneuver the quadcopter using a standard gaming controller. However, flight tests have only been conducted using pose estimates produced by an external motion capture system. We have not yet integrated the ORB-SLAM2 localization algorithm. The positions of in-game objects can be updated based on the real quadcopter position, but this link has not yet been tested in a full-scale game.

VII. Discussion

A. Gaming Applications

We anticipate use of this system primarily in large entertainment venues. The game system could also be sold for private use, however the size of the game play area is rather large, and the system cost would likely be prohibitive for most consumers.

B. Applications outside of gaming

The quadcopter control system provides an intuitive way for novice flyers to operate the Solo quadcopter. The results in the localization and control aspects of this project have potential for use outside of gaming, in autonomous and semi-autonomous applications in industries such as package delivery and site inspection.

The AR elements of the system also have applicability as a training system for drone use as a first-response or security technology. Training scenarios could be developed in-game, and novice pilots could use the game system to practice operating the quadcopter in these training scenarios without the risk of crashing, with AR emergencies added to the real world environment.

C. System Limitations

1) Cost and Hardware Requirements: The system cost is another potential limitation in addition to the limitations discussed in *Quadcopter Control*. The hardware requirements of the system are necessitated by software restrictions, and by the computational demands of performing localization and rendering the game. The Oculus SDK currently only supports one set of headset and controllers per workstation - as a result, our game system requires one Windows workstation per player to run the game in Unity.

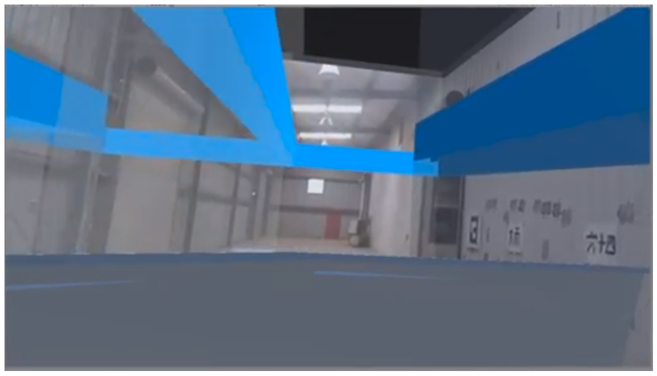

Fig. 13. User view of maze game

D. Future Improvements

There are a wide range of improvements that could be made to this system. Adding a smartphone app that allows spectators to view the AR elements of the game through their phone would improve attraction quality for spectators. The variety of available games could be greatly improved as well. In addition to the maze and pong games, we are currently developing a shooting game, in which players fire objects at each other. A number of other games could be developed, including tag, soccer, billiards, and racing. The aesthetics of the existing games could also be improved by adding color and texture to the virtual object models.

VIII. Conclusion

Augmented Reality (AR) is an increasingly popular technology for entertainment and information display. AR, unlike VR, allows the users to remain firmly anchored in the real world environment while interacting with visual elements that appear in fixed registration with the real world coordinate frame. One of the future areas where AR can offer a paradigm shifting experience is in combination with quadcopters. We propose a multi-player video game platform that combines AR and quadcopter technology, which can be used to implement a wide variety of games.

This system relies on multiple 3DR Solo quadcopters, equipped with GoPro Hero cameras. Each player's quadcopter is connected to the Linux workstation, which is responsible for localizing each quadcopter using the ORB-SLAM2 algorithm. The Linux workstation is networked to multiple Windows workstations (one per player) which are responsible for running the AR game and producing the user display in Unity.

We propose a control system that uses player input to produce waypoints relative to the quadcopter, and allows the quadcopter to navigate to these waypoints automatically using its autopilot firmware. This system will facilitate collision avoidance - waypoints can be modified in the case that a collision is predicted. This system also facilitates the simulation of in-game events on the physical quadcopter.

Discrete components of the game system have obtained promising preliminary results. The localization system produces pose estimates that are accurate enough to play a simple 1-player pong game. The control system has also shown promising results, as we are able to navigate the quadcopter using a standard video game controller, and take off and land automatically. However, these two systems have not yet been integrated. In addition, the proposed collision avoidance system has not been implemented.

After further development and refinement of this system, we anticipate its use in exhibition settings or entertainment venues. Our results lay the groundwork for an exciting new form of quadcopter-based multi-player AR gaming.

Appendix

Acknowledgment

References

[1] Snapchat. [Online]. Available: https://whatis.snapchat.com/

[2] Pokemon Go. [Online]. Available: http://origin.pokemongo.com/

[3] Mobidev. Augmented reality in healthcare: Way to medicine's digital transformation. [Online]. Available: https://mobidev.biz/blog/augmented-reality-in-healthcare-digital-transformation

[4] HTC Vive-VR. [Online]. Available: https://www.vive.com/eu/

[5] Playstation. PS-VR. [Online]. Available: https://www.playstation.com/en-us/explore/playstation-vr/

[6] Oculus Rift. [Online]. Available: https://www.oculus.com/rift/

[7] Thoughtfulmonkey. Augmented reality with unity (and artoolkit, vuforia, and wikitude). Youtube. [Online]. Available: https://www.youtube.com/watch?v=mZAg222mzA8

[8] E. Marchand, H. Uchiyama, and F. Spindler, "Pose estimation for augmented reality: A hands-on survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 12, pp. 2633–2651, Dec 2016.

[9] C. Forster, M. Pizzoli, and D. Scaramuzza, "SVO: Fast semi-direct monocular visual odometry," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[10] M. J. M. M. Mur-Artal, Raúl and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[11] F. J. Romero-Ramirez, R. Muoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers," *Image and Vision Computing*, vol. 76, pp. 38 – 47, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0262885618300799

[12] M. G. M. W. D. S. Christian Forster, Zichao Zhang. Svo 2.0: Fast semi-direct visual odometry for monocular, wide angle, and multi-camera systems. Robotics and Perception Group, University of Zurich, Switzerland. [Online]. Available: http://rpg.ifi.uzh.ch/svo2.html

[13] R. Toris, J. Kammerl, D. V. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills, and S. Chernova, "Robot web tools: Efficient messaging for cloud robotics," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2015, pp. 4530–4537.

[14] A. Mordvintsev and A. K. Camera Calibration. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py$_tutorials/py_calib3d/py_calibration/py_calibrati$

[15] P. Drap and J. Lefvre, "An exact formula for calculating inverse radial lens distortions," *Sensors*, vol. 16, no. 6, 2016. [Online]. Available: http://www.mdpi.com/1424-8220/16/6/807