CalAR: A C++ Engine for Augmented Reality Applications on Android Mobile Devices

Menghe Zhang, Karen Lucknavalai, Weichen Liu, Jürgen P. Schulze; University of California San Diego, La Jolla, CA, USA

Abstract

With the development of Apple's ARKit and Google's ARCore, mobile augmented reality (AR) applications have become much more popular. For Android devices, ARCore provides basic motion tracking and environmental understanding. However, with current software frameworks it can be difficult to create an AR application from the ground up. Our solution is CalAR, which is a lightweight, open-source software environment to develop AR applications for Android devices, while giving the programmer full control over the phone's resources. With CalAR, the programmer can create marker-less AR applications which run at 60 frames per second on Android smartphones. These applications can include more complex environment understanding, physical simulation, user interaction with virtual objects, and interaction between virtual objects and objects in the physical environment. With CalAR being based on CalVR, which is our multi-platform virtual reality software engine, it is possible to port CalVR applications to an AR environment on Android phones with minimal effort. We demonstrate this with the example of a spatial visualization application.

Introduction

Recent advancements in mobile hardware and single camera-based tracking technology enable us to utilize augmented reality (AR) technology on mobile devices powered by Google's ARCore or Apple's ARKit software libraries. On Android devices, ARCore provides the functionality needed to use the Android phone for AR applications: camera-based six degree of freedom (i.e., 3D position and 3D orientation) motion tracking, as well as the recognition of flat surfaces in the physical environment. The latter can be used to place virtual objects which snap to surfaces in the physical environment. Our aim was to build a lightweight, open source software framework to easily and quickly develop AR applications for Android smartphones. Our framework includes 3D tracking and simple environment understanding. We also integrated a physics engine (Nvidia's PhysX), and implemented a lighting estimation module to illuminate the virtual objects in the scene as if they were illuminated by the light sources in the physical environment.

One of the most difficult aspects of creating AR applications for smartphones is to create appropriate techniques for intuitive interaction between the user and the AR application. The mobile nature of smartphone AR limits the user's interaction capabilities, because a 3D controller or hand tracking are not generally available. We created an interaction system, which is based on finger taps on the smartphone's display, and gives access to the advanced 3D menu system from our virtual reality (VR) development environment CalVR, which CalAR is based on. This way the programmer can quickly add menu functionality which can be accessed through the smartphone's touch screen.

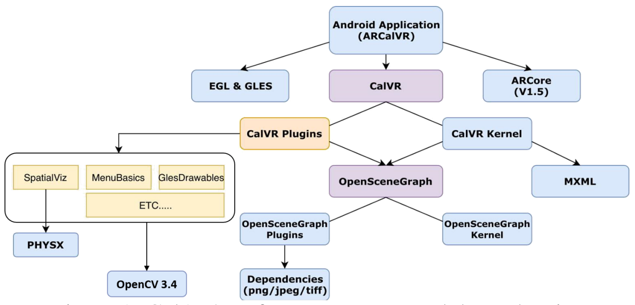


Figure 1: CalAR's software structure and dependencies

This article describes each component of the software system we built, and summarizes our experiences with two demonstration applications we created with CalAR.

Related Work

In the early days of augmented reality applications on smart phones, fiducial markers located in the physical environment were used to estimate the phone's 3D pose with respect to the environment. One of the earliest software libraries which solved this problem was ARToolkit [4,5], which used square, black and white markers similar to QR codes. Later on, the Vuforia library [6] allowed using known images of any kind for 3D markers.

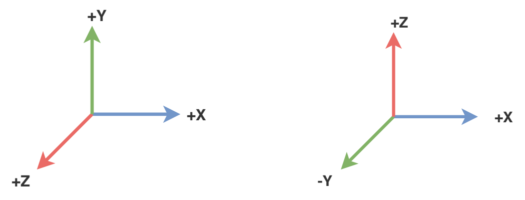
Later on, miniature 3D accelerometers and gyroscopes became widespread sensors in smartphones, and researchers found ways to use them to improve position and orientation tracking. Ben Butchart's report on AR for Smartphones [8] provides an excellent overview of the state of the art in 2011.

Today, the most common approach for AR on smartphones is to use Unity [7] along with ARKit [9] or ARCore [10] to enable AR features such as 3D pose estimation and surface recognition. The reason why we developed CalAR is that it is fully open source (Unity is closed source), allows programming in C++, and is an extension of CalVR, which with its 10 year history has produced many VR applications which are desirable to run on a mobile device.

System Overview

CalAR is a software framework for Android to create augmented reality applications. Figure 1 shows an overview of the system's software structure and its dependencies.

CalAR is targeted for Android, while using the Java Native Interface to support its parent software CalVR's C++ code. With the use of ARCore, this system requires Android 7.0 (API level 24) or later. To take full advantage of CalAR, Android 8.0 (API level 26) or later is required. For rendering, GLES 3.x is supported starting from Android 5.0 (API level 21). More powerful



(a) Tracking coordinates (b) Virtual coordinates

Figure 2: Aligning different systems.

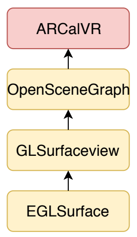


Figure 3: Rendering hierarchy.

graphics chipsets will increase the rendering rate, produce more fluid screen updates, and an overall better user experience. AR is very CPU and GPU intensive.

For development and testing, we used a Samsung Galaxy S9 smartphone with Qualcomm's Snapdragon 845 SoC and and Adreno 630 GPU running at 710 MHz, API level 26. CalAR is the AR extension of our virtual reality visualization framework CalVR [1]. CalVR is an open source software framework written in C++ which implements VR middleware functionality for VR applications running on consumer VR headsets all the way to VR CAVEs with large rendering clusters. CalAR extends it with AR pose estimation through ARCore, a physics engine, a 3D interaction concept for touch screens, and environment-based lighting.

Android-CalVR Communication

Communication between the high-level Android application and the low-level CalVR framework required a multi-layer software implementation. We use the Java Native Interface (JNI) for the main Android application, which is written in Java, to be able to communicate with the underlying C++ code from CalVR. We implemented an entry controller class as the jumping off point into C++ where we can communicate directly with the various C++ libraries which are part of CalVR.

3D Position and Orientation Tracking

Google's ARCore has the ability to track a few dozen feature points in 3D using the smartphone's built-in camera and motion sensors, and compute the 6-degree of freedom (DOF) trajectory of the device. The tracking system of ARCore is based on sparse feature detection. Once the 3D feature points are detected, and when the camera pose is known, we can overlay 3D virtual content on top of the image. ARCore's tracking system is able to detect flat surfaces within a scene with rich features. Using ARCore as the backbone, we built our own tracking system aligning the real and virtual cameras as well as a variety of coordinate systems needed for application development.

Coordinate Alignment

The first issue we needed to tackle was the inconsistency of the coordinate systems between virtual world and real world. The tracking, or real-world system utilized a y-up right-handed coordinate system, whereas the virtual system was a z-up right-handed coordinate system. Figure 2 shows the different coordinate systems for tracking (real-world) system and virtual system. The same issue arose when we integrated the physics simulation engine PhysX, OpenSceneGraph and GLES. It is crucial for CalAR to work correctly that the coordinate system transformation matrices for all coordinate systems are known.

Camera Alignment

ARCore creates the world coordinate system with the camera in its origin. As long as the system does not lose tracking, all detected feature and anchor points remain consistent. In a first person game for a regular computer, the user controls the behavior of the player character in the scene, which manipulates the virtual camera. However, since CalAR displays the real-world as seen by the camera held by the user, the viewpoint of the virtual scene should be aligned with this camera.

Rendering and Display

Android provides a system-level graphics architecture that can be used by the application framework. So we focused on how the visual data moves through the system based on Android's graphics architecture. Figure 3 shows CalAR's rendering hierarchy. The low-level component EGLSurface is provided by Android. It is a graphics-rendering API which is designed to be combined with EGL (i.e., OpenGL ES). Using EGL calls, we can create and access windows through the operating system to render our scene. GLSurfaceView provides a helper class to manage EGL contexts, inter-thread communication and interaction with the Activity lifecycle.

After creating and configuring a renderer to GLSurfaceview, we can then manipulate the GL context on both the Java and native C++ sides. CalVR utilizes the high-level graphis toolkit OpenSceneGraph (OSG), written entirely in Standard C++ and OpenGL, which enables us to write OpenGL code and GLSL shader files and adapt them as part of the scene structure.

User-Interaction Interface System

In most cases, for handheld AR applications, the non-dominant hand holds the device and points the camera towards the target. Leaving the dominant hand free to interact with the virtual environment. We applied three interaction techniques to our interaction system: multi-touch, 3D direct touch and a ray-casting manipulator for menu interaction.

Multi-Touch Interaction

CalAR being the AR extension of CalVR, is ready to use 3D controllers for user interaction, or mouse clicks on a regular monitor. But it is not prepared to interact through a touch screen with finger gestures. To translate mouse and 3D controller interactions into something more intuitive for touchscreen devices, we use multi-touch input from the touchscreen. The core mouse or controller events of clicking and dragging were mapped to multi-touch events. Swipes are interpreted as mouse drags, and taps are interpreted as mouse clicks, one finger corresponding to the left mouse button, and two fingers corresponding to the right. A

Operation	Desktop	Mobile
Drag	Left click and move	One finger touch and move
Click	Left single click	One finger single tap
Double-click	Left double click	One finger double tap
Right-click	Right single click	Two finger single tap
Right-double-click	Right double click	Two finger double taps

Table 1: Map of multi-touch events to mouse events

Operation	Description
Long press	1, 2, and 3 fingers touch for an extended time
Fling	Single finger quickly swiping across screen

Table 2: Unique events for touchscreen

breakdown of this can be seen Table 1.

In addition to providing this core functionality, we extended the interaction options to include multi-touch events that are unique to touchscreens, as well as long presses and flicks. Table 2 describes the unique events detected by our multi-touch detector.

3D Direct Touch

Mouse events, like clicking and dragging, can be more intuitive than multi-finger interactions. Often the latter interaction type requires prior knowledge to be able to correctly use the touch gestures. In an effort to make the interaction as intuitive as possible we created the ability to touch and directly interact with the objects in the AR environment on the screen.

Object selection: ARCore detects feature points in the scene that describe the camera's position and orientation in world coordinates. We attach the virtual objects to the coordinate system these anchor points are in to ensure that they maintain consistent positions and orientations. Then to select a virtual object, the user simply touches the screen where the object appears. This touch triggers a virtual ray to be cast into the environment which then determines which object the user selected.

The initial touch position p_t is in 2D screen coordinates, and is first back-projected onto the near and far planes of the virtual camera's view frustrum, as $\mathbf{P_{t_{near}}}$ and $\mathbf{P_{t_{far}}}$ This ray $\mathbf{P_r}$ can be represented by equation 1, where the scalar t is determined by the first object that the ray hits.

$$\mathbf{P_r} = \mathbf{P_{t_{near}}} + \mathbf{r} \cdot t, \quad r = \mathbf{P_{t_{far}}} - \mathbf{P_{t_{near}}} \tag{1}$$

Object Translation: If we selected an object and want to move it within our scene, we click on the object to select it and drag our finger across the screen. When we dragged our finger the touch position changed from $\mathbf{p_{t_1}}$ to $\mathbf{p_{t_2}}$. These screen space vectors can be back-projected into 3D space as $\mathbf{P_{r_1}}, \mathbf{P_{r_2}}$. The scalar value t for $\mathbf{P_{r_2}}$ can then be calculated from similar triangles using equation 2, where $\mathbf{n}$ is the normal vector of camera plane:

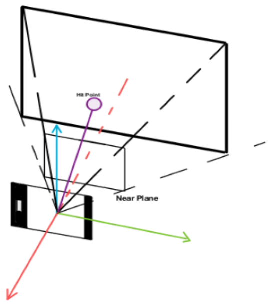


Figure 4: Ray-object intersection test in 3D space.

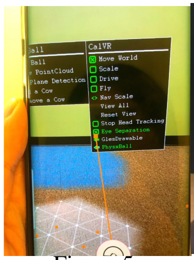


Figure 5: Raycasting

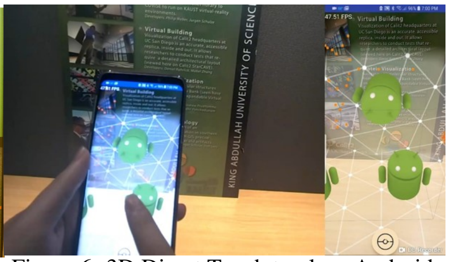


Figure 6: 3D Direct Touch to place Android robots.

$$t = \frac{\mathbf{n} \cdot \mathbf{P_{r_1}} - \mathbf{n} \cdot \mathbf{P_{t_2}}_{near}}{\mathbf{n} \cdot \mathbf{r}} \tag{2}$$

Object Rotation: We handle 3D rotations similar to translations. In order to rotate an object we first click on the object to select it, then drag our finger to rotate the object. Our rotations are about the X and Y axes of the camera coordinates, and the angle is based on the difference between the previous and current touch positions, $\mathbf{p_{t_1}}$ to $\mathbf{p_{t_2}}$.

$$\begin{aligned} &\mathbf{R} = Angle\_Axis(\delta_x, Axis_Z) \cdot Angle\_Axis(\delta_y, Axis_X) \\ &(\delta_x, \delta_y) = p_{t_2} - p_{t_1} \end{aligned} \tag{3}$$

Ray-casting Manipulator

CalAR's menu system is an extension of the CalVR's menu system. After a designated tap event occurs, the menu is placed facing the user inside the 3D environment. We then recreate the effect of a mouse cursor with a ray-casting manipulator. The ray starts at the camera position and extends into the scene. This manipulator is drawn in the scene as an orange line segment. Each frame we check the intersection between the stroke and the menu and highlight the potential menu selection. Selecting this menu item is done by tapping the main button seen at the bottom of the screen, which always remains visible. Screenshots of this interaction are shown in Figure 5

We designed this ray-casting manipulator to avoid "fat finger" errors [3] and enable precise selection. We noticed that it could be difficult to interact with the menu, so we created this ray-casting function to allow the user to more clearly see what menu option they are clicking on. We also use this ray-casting method to manipulate the menu placement. By selecting the top of the menu with the ray the menu attaches to the ray, and by moving the phone the user can move the menu around the scene. We then created this method of interaction as a alternative to the 3D Direct Touch method. The advantage of the new method is that it does not limit the manipulation of the virtual objects to the size of the mobile device's screen.

Physics Engine

To demonstrate how real-world data from the environment can be incorporated within an augmented scene, we integrated PhysX, Nvidia's multi-platform physics engine, into our software framework. Since we use the latest NDK, in order to prevent linking and runtime errors due to combining libraries built by different compilers and standards, we updated PhysX's Android makefile to build PhysX with the latest NDK with Clang instead of

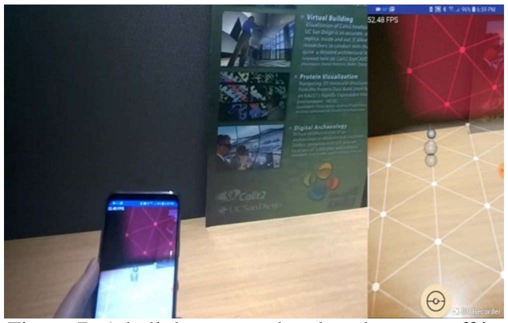


Figure 7: A ball thrown on the plane bounces off it.

GCC, and statically linked all PhysX libraries. PhysX maintains and runs simulations inside its own PhysX world, so we dynamically update real-world data coming from ARCore into the PhysX scene. Whenever ARCore detects a flat surface or plane in the environment, we create a thin box to tightly fit the detected plane. We found that this reduced the performance cost when compared to dynamically updating a convex mesh into the PhysX scene. Figure 7 shows a bouncing ball example of this feature.

Lighting

We implemented three different approaches for lighting in our Android app: ARCore Native, Single Source, and Spherical Harmonics. In our demo application, the user can select on a per object basis which of the three lighting methods to use, which allows rendering different objects with different lighting methods in the same scene and directly compare them to one another.

ARCore Native

AR Core Native comes directly from ARCore. This method calculates the average pixel intensity of the captured image, and renders the objects based on that average brightness. This is our lighting baseline to compare other lighting estimation implementations.

Single Source

Single Source builds on the functionality available in ARCore to improve the lighting of the scene. We calculate the pixel location of the brightest point in the captured image. This is then used as a 3D point light to provide additional lighting to help create a more realistic rendering for diffuse and specular surfaces.

Spherical Harmonics

Spherical Harmonics lighting projects objects and/or environment map into spherical harmonics frequency space, and then calculates lighting with spherical functions [2]. By pre-computing the complex lighting environment and per-vertex radiance transfer function into vectors of coefficients in frequency space, the lighting integration can be done in real-time. This method relies heavily on a detected environment map, so we implemented two ways to iteratively build this environment map. As the user uses the AR app and points the smartphone in different directions, we gather more and more images and create a more complete environment map.

1. We use OpenCV to stitch the input images into a panoramic image. This method is slow but accurate.

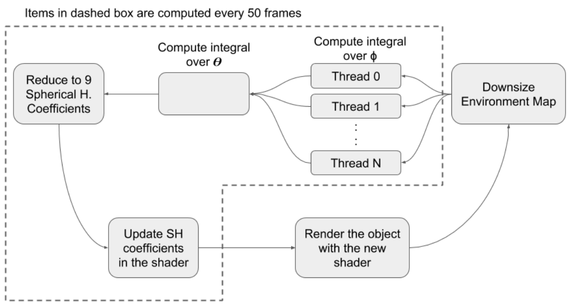

Items in dashed box are computed every 50 frames
Reduce to 9 Spherical H. Coefficients
Compute integral over θ
Compute integral over ϕ
Thread 0
Thread 1
Thread N
Downsize Environment Map
Update SH coefficients in the shader
Render the object with the new shader

Figure 8: Showing the flow of the Spherical Harmonics algorithm used within the Android app. The Spherical Harmonics are only calculated and updated every 50 frames.

2. We use projection and view matrix from ARCore to directly project the image onto a sphere. This method has a smaller performance impact, but tends to be more inaccurate in a more confined environment.

These methods are only applied once every 50 rendered frames (about once per second) so that they do not slow down the rest of the application too much. Initially we only use the current view as the environment map, assuming that the rest of the environment does not contribute to the augmented scene yet. Once we were able to stitch together enough images to fill at least 80% of the environment map we switch to the more complete environment map to make sure we use information in the calculations that will result in more accurate rendering.

To compute the Spherical Harmonics on this environment map we downsize the image, and the following calculations are completed every 50 frames:

- We compute the integral of the environment over one dimension ϕ. This calculation is parallelized and completed in multiple threads.
- Once this is complete, one thread then computes the resulting 9 Spherical Harmonic coefficients by integrating over the other angle θ.

Every frame these Spherical Harmonic coefficients are sent to a shader to calculate the diffuse colors and render objects based on the current environment map. The flow of the Spherical Harmonics Algorithm can be seen in Figure 8.

Test Applications

We created two plugins to demonstrate the features of our CalAR system. The first one *Play with Android Andy* demonstrates the following features: basic rendering, interaction, physics and lighting. The other, *SpatialViz*, is an application originally developed as CalVR plugin for VR environments. We adapted it to our CalAR in Android to show the abilities to build, use, and integrate plugins which were developed for CalVR.

Demo App: Playing with Android Robots

This application is shown in Figure 9. To start the app, the user holds the phone up and sees the 3D menus shown on in the physical environment. Then he double taps the screen and the main menu pops up in front of him 10. After browsing the menu,

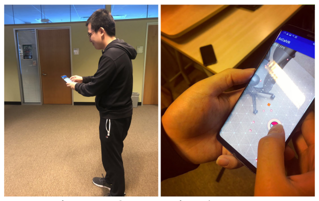


Figure 9: Typical usage scenario.

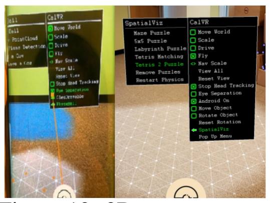


Figure 10: 3D menu system.

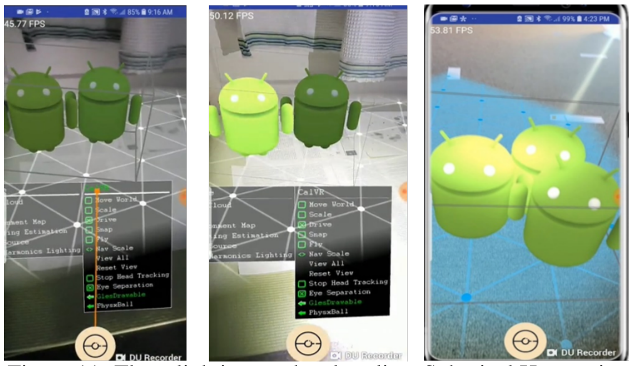


Figure 11: Three lighting modes: baseline, Spherical Harmonics lighting + baseline, and single light source.

he points the ray to check the box ”Add an Andy” with the default lighting mode. With the camera facing the floor, a flat surface is drawn on the screen in the location of the floor, and Android robots can be placed on this surface. To interact with them the user places robots in random positions and rotates one of them to face towards the others, see Figure 6.

We also implemented three different lighting modes as shown in Figure 11. The single-source-lighting estimates the real-world light source and the Android robot shows a highlighted spot on its head. Spherical Harmonics lighting mode, on the other hand, captures the current view every 50th frame and stitches them iteratively to generate a panoramic image of the environment (see Figure 12).

Demo App: Spatial Visualization Trainer

Our pre-existing Linux-based spatial visualization training application for CalVR was developed to teach engineering freshmen better 3D spatial visualization skills. The application uses a basic framework for 3D interaction within a VR environment. We thoguht that the spatial visualization trainer was a good, typical app to test how easy it would be to port a CalVR application to

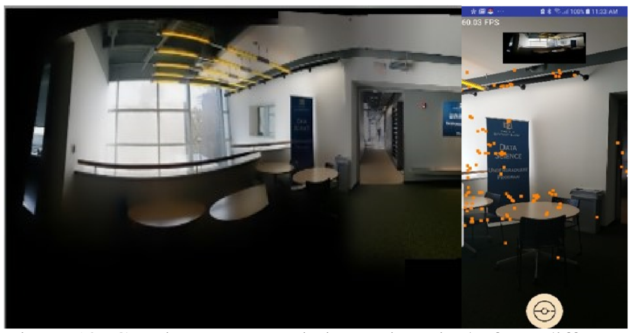


Figure 12: Creating a panoramic image iteratively from different viewing directions.

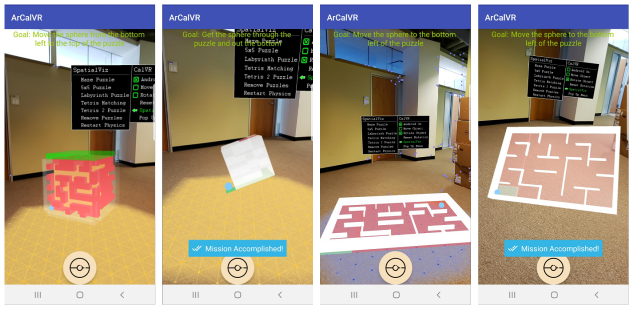


Figure 13: Showing three of the different puzzles created in the spatial visualization training app. Each involve the rotation and manipulation of different puzzle pieces to get the ball to go where it should.

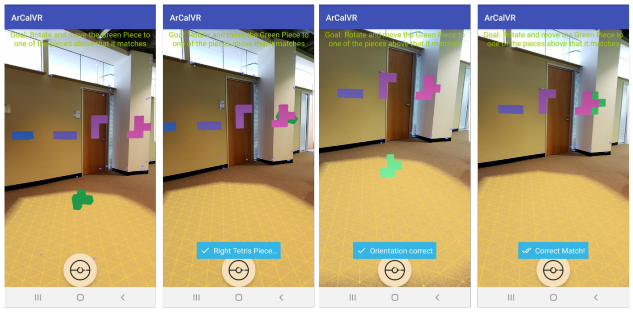


Figure 14: Tetris puzzle: the user needs to move and rotate the green puzzle piece in the AR space using taps and swipes to find and match the position and orientation of another piece.

CalAR. This application also included a variety of components we wanted to test out in CalAR, such as PhysX, as well as 3D interaction with virtual objects.

The spatial visualization trainer creates virtual 3D puzzles that the user can interact with in 3D space. Figure 13 shows three different puzzles created by the app. Each of them involves the use of the PhysX library to simulate interaction between the puzzles and a ball. Android Toast-style messages are displayed to the users when they have completed the puzzle, made possible by the two-way communication through the JNI and controller classes. Figure 14 shows our 3D Tetris puzzle and the different messages displayed to the user.

Discussion

The porting of the spatial visualization trainer app from CalVR to CalAR showed us various parts of the software which needed to still get fixed. Once it worked, we tested it out among the developers and report on our observations below.

CalAR vs. CalVR

When we initially ran the CalVR plugin on the Android phone a big issue was the scale of virtual objects. CalVR is a VR only platform and as a result the scaling of objects can be fairly flexible. However, in our Android application the camera position is fixed within the real-world, so the scale of objects becomes fixed. We needed to re-scale the objects and address some floating point issues before the application ran smoothly. Figure 15 shows a comparison between the Linux (CalVR) and Android (CalAR) environments for two of the puzzles in the trainer application.

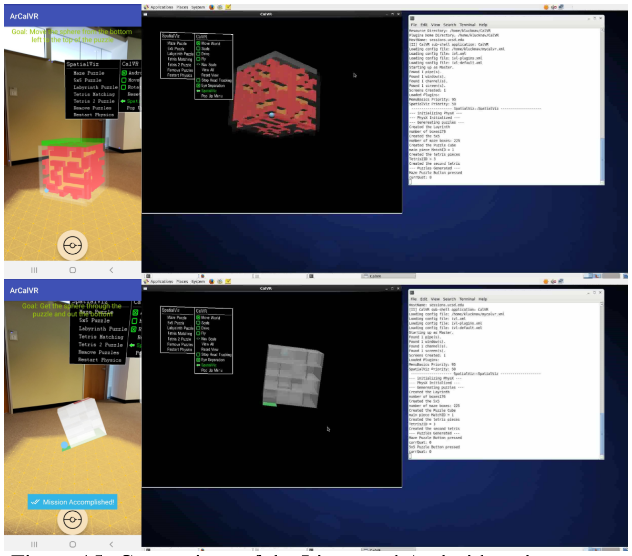


Figure 15: Comparison of the Linux and Android environments running the spatial visualization trainer app.

Real World Coordinates

ARCore is able to detect flat surfaces within a scene if there are enough features on the surface, i.e., if you are looking at a flat solid colored wall ARCore may not to detect that surface. Once ARCore does detect a flat surface, ARCore can project a 2D screen tap into a 3D feature point onto that surface. So the functionality we ended up implementing was that when a surface was detected, a single-finger double tap would then be translated into that World Coordinate feature point. The matrix that describes the translation and rotation of that feature point could then be used to place our applications objects onto the surface.

One thing that we always had to be cautious of was the units and orientation of the coordinate system. We were dealing with a lot of different coordinate systems and ensuring that they all correctly lined up was a recurring issue.

Camera Positioning

The Linux based system had a set initial camera position, and upon a mouse click and drag the camera position rotated around the stationary object. This type of interaction no longer made sense in an AR environment. We first fixed the camera location to the phone's camera location so it properly lined up with the real-world image on the screen. We then changed the click and drag interactions to change the object's position and orientation rather than the camera's.

Rendering Performance

The different puzzles were all created with boxes and spheres. As a result the amount of triangles that needed to be rendered for the puzzles ranged from between 300 and 3,650 triangles. CalAR was able to seamlessly render these puzzles while maintaining a frame rate of 60 frames per second.

Conclusions

Based on our existing CalVR middleware software, the new CalAR for Android integrates ARCore as tracking system, as well as a 3D menu system, a physics engine, and environment-based lighting. With the CalAR framework, software developers can build their own AR applications more easily than starting from scratch. If all that is needed is an AR application for smartphones, Unity may often be the better answer. But if open source or compatibility with a wide range of VR systems is desired, CalAR offers a viable alternative.

Acknowledgments

This project was funded in part by Samsung Electronics and the Helmsley Charitable Trust.

References

[1] Schulze, Jürgen P and Prudhomme, Andrew and Weber, Philip and DeFanti, Thomas A, CalVR: an advanced open source virtual reality software framework, International Society for Optics and Photonics, The Engineering Reality of Virtual Reality 2013, Vol.8649, pg. 864902 (2013).

[2] Ramamoorthi, Ravi and Hanrahan, Pat, An Efficient Representation for Irradiance Environment Maps, Proceedings of the 28th annual Conference on Computer Graphics and Interactive Techniques, ACM, pg. 497–500 (2001).

[3] Siek, Katie A., Yvonne Rogers, and Kay H. Connelly. "Fat finger worries: how older and younger users physically interact with PDAs." In IFIP Conference on Human-Computer Interaction, pp. 267-280. Springer, Berlin, Heidelberg, 2005.

[4] ARToolKit Home Page, HIT Lab, University of Washington, URL: http://www.hitl.washington.edu/artoolkit/

[5] Hornecker E., Psik T. (2005) Using ARToolKit Markers to Build Tangible Prototypes and Simulate Other Technologies. In: Costabile M.F., Paternò F. (eds) Human-Computer Interaction - INTERACT 2005. INTERACT 2005. Lecture Notes in Computer Science, vol 3585. Springer, Berlin, Heidelberg

[6] Vuforia by PTC, URL: https://www.ptc.com/en/products/augmented-reality/vuforia

[7] Unity Real-Time 3D Development Platform, URL: https://unity.com/

[8] Ben Butchart: "Augmented Reality for Smartphones - A Guide for Developers and Content Publishers", University of Bath, UK, March 2011

[9] ARKit: Augmented Reality for Apple's iOS, URL: https://developer.apple.com/augmented-reality/

[10] ARCore: Augmented Reality for Android Devices, URL: https://developers.google.com/ar

Author Biography

Menghe Zhang is pursuing a Ph.D. degree in the fields of augmented reality and immersive visualization at the University of California in Dr. Schulze's Immersive Visualization Laboratory.

Karen Lucknavalai is pursuing an M.S. degree in computer science at UCSD. Her advisor is Dr. Schulze.

Weichen Liu is a computer science undergraduate student working in Dr. Schulze's Immersive Visualization Laboratory.

Dr. Jurgen Schulze is an Associate Research Scientist at UCSD's Qualcomm Institute, and an Associate Adjunct Professor in the computer science department, where he teaches computer graphics and virtual reality. His research interests include applications for virtual and augmented reality systems, 3D human-computer interaction, and medical data visualization. He holds an M.S. degree from the University of Massachusetts and a Ph.D. from the University of Stuttgart (Germany).