The Ultimate Display

Oliver Gabriel Milic Lemonakis
Visual Computing II
Aarhus University
201910301@post.au.dk

***Abstract*—This project uses a smartphone as the direct control surface to move elements in and between two or more displays without the need for any additional input devices, thereby allowing for the extension of the personal workspace into a collaborative setting. By leveraging existing ubiquitous technology, we reduce the necessity for more specialized equipment. Theoretically scaling to an arbitrary number of displays, the design and implementation of this project moves us closer to Sutherland's notion of "The Ultimate Display" [1], providing a dynamic environment that narrows the gap between personal and shared computing. The preliminary findings presented in this paper suggest that the interactions' affordances provide motivation for further work and research into this proposed display modality.**

***Index Terms*—Augmented Reality, Display, Visual Computing**

Python Server Repository
Unity Applications Repository
Demo Video

I. Introduction

What is the "ultimate display"? A question posed, answered by Sutherland [1] in his paper of the same name, echoing throughout the world of visual computing ever since. Many, if not most, of the ideas presented came to fruition in the decades following the publication of the paper. Sensory immersion, the "Wonderland", and interactive physics came to shape the landscape of spatial computing and VR and led to the creation of "The sword of Damocles"[2].

The CAVE [3] virtual environment came almost three decades later as a direct consequence of the "Wonderland" posed by Sutherland as a hypothetical concept of a dynamic computer-controlled environment. CAVE showed us how we could move even closer to some of the ultimate display principles outlined by Sutherland, and the visual computing world has come very far since then.

We, the Visual Computing II students, were asked to see how we would move closer to the "ultimate" display, by combining two or more existing display technologies, to create a new display modality.

As such I created this project, titled similarly to Sutherland's essay, in which I detail the design and implementation of a collaborative multi-display system. This system combines a smartphone and its display and camera as the control surface for a multi-display setup. This setup uses a personal computer and a projector as an extension of the digital environment in physical space. That is my attempt at creating the "Ultimate Display" in a collaborative setting.

II. Design and Goal

This project describes the implementation of a collaborative system, using a phone camera and screen, in combination with the user's own workstation display and a shared display.
An example scenario:

1) The user has something on their workstation display they would like to share
2) The user pulls out their phone
3) The user points it at their own screen
4) The user can now control their WIMP interface using the phone's touch-input as a control surface
5) The user can drag-drop elements from their screen, onto a shared screen in the space.

The intention behind this design being to allow for more spontaneous collaborative efforts that aren't limited by static configurations. Something that feels like a seamless transition between the user's work environment and a shared one. This project is aimed at supporting the ability to take something you would want to share with others in any space at any time, and allowing you to present it without first configuring an additional collaborative workspace.

A. Ultimate Display Principles and This Project

"The Ultimate Display" and its mentioning of the "Wonderland" concept stuck with me. A computer-controlled environment with digital objects inhabiting physical space. I thought, that by extending the desktop display into the physical space we could approach a tangible step in that direction. Albeit not an attempt at addressing sensory immersion, as it is prominently featured in this part of the original Sutherland paper, it still entails spatial input as a part of the interaction by moving the camera around the space.

Additionally, scaling to a potentially unlimited number of displays and users simultaneously means that as the CAVE [3] environment did, this project too is capable of providing an all-encompassing display experience.

The combination of limitless scaling, along with the system not needing any previous knowledge of where the screens are, but only what the users' cameras are looking at when

interfacing with the system, creates a dynamic environment, thus constituting a novel display modality.

III. Implementation

When implementing the system, initially the expectation was to have two applications. Both Unity applications, one on the smartphone and one on the desktop. However, after much experimentation and due to OpenCV unity library issues as detailed in Section V.D, a three-component architecture was achieved. It consists of:

1) Smartphone client using Unity [4], capable of reading touch-input and frame information from its camera
2) A Python server performing CV calculations with OpenCV and transmitting computed homography and touch-input data
3) A Unity [4] application driving the displays and managing the windows on them.

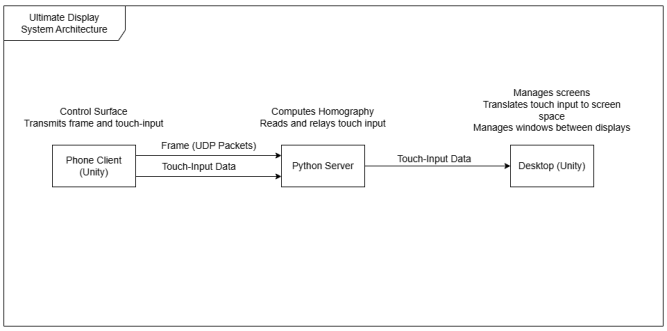

Fig. 1. System Architecture

Hardware used in this project is as follows:

1) Apple iPhone 13 Pro
2) MacBook Pro 14, M4
3) BenQ Full-HD Projector

A. Phone Client

Firstly, the phone client is implemented with Unity. Starting from the ARKit's [5] demo scene, a stripped down version laid the foundation for the phone client and its scene.
We grab the camera's frame from the camera manager, and transmit 30 frames per second.

```
if (cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
{
    frameStatusText.text = "Image acquired, encoding...";
    var conversionParams = new XRCpuImage.ConversionParams
    {
        inputRect = new RectInt(xMin: 0, yMin: 0, image.width, image.height),
        outputDimensions = new Vector2Int(image.width, image.height),
        outputFormat = TextureFormat.RGB24
    };
```

Fig. 2. Image capture code snippet

Encode it to a JPEG image, downscale it slightly, and subsequently partition it into chunks depending on the size of the image.

```
for (int i = 0; i < totalChunks; i++)
{
    var offset (int) = i * ChunkSize;
    var chunkLength (int) = Mathf.Min(ChunkSize, jpegBytes.Length - offset);

    var packet = new byte[9 + chunkLength];
    packet[0] = 0x01;
    BitConverter.GetBytes(_frameID).CopyTo(packet, index: 1);
    BitConverter.GetBytes((ushort)i).CopyTo(packet, index: 5);
    BitConverter.GetBytes((ushort)totalChunks).CopyTo(packet, index: 7);
    Array.Copy(sourceArray: jpegBytes, sourceIndex: offset, destinationArray: packet, destinationIndex: 9, chunkLength);

    _udpClient.Send(packet, bytes: packet.Length, _imageServerAddress, _macPort);
}
```

Fig. 3. Chunk partitioning snippet

This is then sent over UDP to our Python server.

Touch input is transmitted similarly, without the need for partitioning into chunks. Note that along with the normalized touch coordinates, we transmitted the touch "phase", being represented as follows:

0) Touch-down
1) Touch-hold
2) Touch-up (letting go)

```
try
{
    // Packet: [0x02][x:f32 4B][y:f32 4B][phase:1B]
    var packet = new byte[10];
    packet[0] = 0x02;
    float normX = pos.x / Screen.width;
    float normY = pos.y / Screen.height;

    BitConverter.GetBytes(normX).CopyTo(packet, index: 1);
    BitConverter.GetBytes(normY).CopyTo(packet, index: 5);
    packet[9] = phase;
    _udpClient.Send(packet, bytes: packet.Length, _imageServerAddress, _macPort);

    touchStatusText.text = $"Touch phase={phase} x={pos.x:F0} y={pos.y:F0}";
}
catch (Exception e)
{
    touchStatusText.text = $"TOUCH SEND ERROR: {e.Message}";
}
```

Fig. 4. Touch input transmission

B. Python Server

1) Frame reassembly

Initially, we define a frame buffer based on the total number of packets representing a specific frame received from our iPhone. These packets are then joined in a "raw" representation.

```
if len(frame_buffer[fid]) >= total:
    if len(frame_buffer[fid]) != total:
        del frame_buffer[fid]
        continue

    raw = b''.join(frame_buffer[fid][i] for i in range(total))
```

Fig. 5. Frame joining

We decode the frame from this raw data using OpenCV [6].

```
frame = cv2.imdecode(np.frombuffer(raw, np.uint8), cv2.IMREAD_COLOR)
```

Fig. 6. Frame Decoding

Using OpenCV again, we rotate and flip the frame, to ensure that the orientation of the frame we're processing is representative of the frame seen on the phone's camera.

```
frame = cv2.rotate(frame, cv2.ROTATE_90_CLOCKWISE)
frame = cv2.flip(frame, 1)
```

Fig. 7. Frame rotate and flip

2) Screen identification

Each display has a unique set of ArUco [7] markers, generated from online using a ArUco generation tool [8], attached to them. These markers work by assigning IDs to them, allowing them to be deciphered as numerical values with OpenCV [9]. When these markers are identified in a frame, their corresponding marker IDs are used to identify which screen we're looking at.

```
ARUCO_DICT   = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
ARUCO_PARAMS = cv2.aruco.DetectorParameters()
ARUCO_PARAMS.minMarkerPerimeterRate      = 0.01
ARUCO_PARAMS.polygonalApproxAccuracyRate = 0.08
ARUCO_PARAMS.errorCorrectionRate         = 0.6
DETECTOR = cv2.aruco.ArucoDetector(ARUCO_DICT, ARUCO_PARAMS)

DISPLAY_MARKERS = { 0: {0,1,2,3}, 1: {4,5,6,7} }
```

Fig. 8. ArUco parameters snippet

If at least two markers from one of the sets corresponding to a display are detected, the system considers the display identified. Requiring at least two markers reduces false identification from mis-detected markers, and allows for the system to not have all markers constantly visible in order to be able to perform detection.

```
ids_flat  = set(ids.flatten().tolist())
screen_id = next((sid for sid, ms in DISPLAY_MARKERS.items()
                  if len(ids_flat & ms) >= 2), None)
if screen_id is None:
    continue
```

Fig. 9. Screen ID snippet

3) Homography estimation and validation

First we pair up detected corner position in the camera image space, with physical position on the screen based on the detected markers.

```
img_pts, scr_pts = [], []
for i, mid in enumerate(ids.flatten()):
    if mid not in DISPLAY_MARKERS[screen_id]:
        continue
    sc = marker_corners_screen(mid, screen_id)
    for j, c in enumerate(corners[i][0]):
        img_pts.append([float(c[0]), float(c[1])])
        scr_pts.append(sc[j])
```

Fig. 10. Image and screen points

Using OpenCV's [10] homography estimation, we use the screen and image points and compute a 3x3 homography matrix, through which we can project touch coordinates. We use RANSAC to make the estimation more robust.

```
H, _ = cv2.findHomography(
    np.array(img_pts, dtype="float32"),
    np.array(scr_pts, dtype="float32"),
    cv2.RANSAC, 5.0
)
```

Fig. 11. Homography estimation snippet

Before we accept any homography, we validate this by projecting the centre point of the camera frame through H, and we verify whether it falls within the boundaries of the screen's dimensions. If it does not, we fail the homography, if it does we consider it valid. This helps discard invalid homographies, adding to the robustness of the system.

```
def homography_is_sane(H, fw, fh, screen_id):
    sw, sh = SCREEN_DIMS[screen_id]
    p = H @ np.array([fw/2, fh/2, 1.0])
    p /= p[2]
    return (0 < p[0] < sw) and (0 < p[1] < sh)
```

Fig. 12. Homography sanity snippet

4) Touch coordinate projection

When we detect touch input from the user's device, these coordinates are in screen normalized to the phone. Firstly, we convert these coordinates to pixel coordinates based on the camera. These are subsequently projected through our homography matrix H, in order to account for perspective warping. This result is then normalized to a 0-1 range, relative to the screen's dimensions. Lastly, we invert the Y axis to respect Unity's canvas bottom-to-top approach.

```
img_x = tx * last_fw
img_y = ty * last_fh

out    = last_H @ np.array([img_x, img_y, 1.0])
out   /= out[2]
sw, sh = SCREEN_DIMS[confirmed_screen]
nx     = float(np.clip(out[0] / (sw - 1), 0.0, 1.0))
ny     = float(np.clip(out[1] / (sh - 1), 0.0, 1.0))
ny     = 1.0 - ny
```

Fig. 13. Touch snippet

5) Drift prevention

In order to combat the problem of drift, we only use the last validly computed homography. This aids us in not being

affected as harshly by drifting while crucially still allowing us to switch screens mid-drag.

```
if is_touching:
    if homography_is_sane(H, fw, fh, screen_id):
        confirmed_screen = screen_id
        send_sock.sendto(json.dumps({
            "type":      "screen_change",
            "screen_id": screen_id,
        }).encode(), unity_addr)
        print(f"  Screen update → {screen_id}")
    continue
```

Fig. 14. Drift prevention snippet

C. Desktop Application

The desktop application is a Unity application driving all displays and performing all window management. Additionally, the Unity application manages the ArUco markers used to detect which screen we're looking at as well.

The most crucial parts of this Unity application are:

1) Image Processor

The Image Processor handles all incoming communication from the Python server. It defines two message queues which handles touch and frame information respectively, and parses the JSON formatted output of said python server. It processes these JSON informations into a "TouchEvent" struct, which is used as a definition of the user's behavior input on the control surface.

2) Window Manager

The Window Manager is responsible for defining, displaying, and managing the interactive windows of the application. It reads image processor information from its queues and uses it to perform actions on the windows, depending on the touch and coordinate output of the Image Processor. Such an output could be a "tap-down" touch input.

```
if (touch.type == "tap_down")
{
    _grabbed = FindWindowAt(touch.screenId, nx: touch.normX, ny: touch.normY);
    if (_grabbed == null)
    {
        foreach (var w :DraggableWindow in _windows)
            if (w.transform.parent == canvas.transform)
            { _grabbed = w; break; }
    }
```

Fig. 15. Grabbing snippet

Its "FindWindowAt" method, takes the normalized coordinates and the screen ID and translates these coordinates to the corresponding touch-position on the desktop or projector screen. It is by this method of hit testing that we know whether or not a window was "grabbed" in from the control surface.

```
if (w.transform.parent != canvas.transform) continue;
var rt    = w.GetComponent<RectTransform>();
float hw  = rt.sizeDelta.x * 0.5f;
float hh  = rt.sizeDelta.y * 0.5f;
if (Mathf.Abs(f: touchPos.x - rt.anchoredPosition.x) < hw &&
    Mathf.Abs(f: touchPos.y - rt.anchoredPosition.y) < hh)
    return w;
```

Fig. 16. Hit testing snippet

On release, meaning a "tap-up" touch input, we look at the image processor's confirmed screen id. If such a screen ID exists, we now know which screen is targeted by the phone's camera during user input. Depending on which screen ID is defined, we then know which canvas to target with the window. From there, we move the window to the targeted position on the corresponding canvas.

```
else if (touch.type == "tap_up")
{
    if (_grabbed != null)
    {
        int targetScreen    = imageProcessor.confirmedScreenId ?? touch.screenId;
        Canvas targetCanvas = targetScreen == 0 ? display0Canvas : display1Canvas;
        MoveWindow(_grabbed, targetCanvas, nx: touch.normX, ny: touch.normY);
        _grabbed    = null;
        _grabCanvas = null;
    }
```

Fig. 17. "Tap-up" snippet

3) Display Manager

The display manager handles display activation and resolution settings. Knowing and fixing the resolution settings is crucial to the system. The display manager also acts as the scene graph parent which holds the window manager, and its reference to the image processor.

IV. Demo and Preliminary Findings

The demo environment was purposely made simple, as to focus on showcasing the new modality and interaction. A projector and a desktop screen were setup, to mimic a personal work-space and a collaborative work-space. On each screen is a "window" titled and colored differently. These windows could be selected and moved on the displays, and moved independently to and from them.

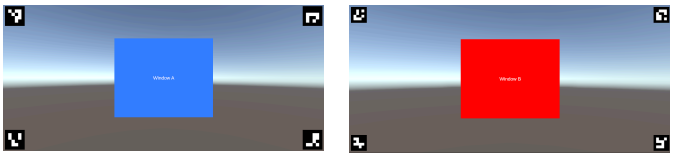

Fig. 18. Display outputs and windows with markers visible

Results presented are comments or statements made in conversation during Demo-Day 9th of May, a collaborative effort between Mixed-Reality course attendees and those of the Visual Computing II course. Though evaluation approach is

not meticulous, the results of the evaluation efforts are still worthwhile to present with these shortcomings in mind.

Demo attendees were given a brief 5-10 second introduction, informing them of the ability to point at either screen and move around the window at the screen using the touch input. As soon as attendees had moved around both windows on either screen, I asked if they would like to try and move one of the windows between the two displays, informing them that they have to drag-and-hold and then point to the other screen.

A. Interaction and Usability Impressions

Demo-attendees took a second or two to find their footing with how exactly to point and click using the camera and touch-input, but once they found it, attendees considered it intuitive. Most attendees experienced the "aha" moment as they saw whichever window they chose to move using touch-input. Many attendees found the system quite responsive, and were happy with the system's ability to recognize which screen was pointed at despite not having all markers in frame.

As they had familiarized themselves with moving the windows on their respective displays, and I had given them the prompt to try and move between displays, most attendees were able to do so on their first try. Attendees did not find it difficult to move between the displays, and for most this was the most impressive part of the interaction. They liked the ability to simply point at which screen they wanted to move the window to, wait ½ a second to ensure that the system had caught on to which display they were pointing at, and then let the window down on the display.

Following the first or second time performing the "move-between-display" interaction, respondents tried playing around with the system with much greater confidence.

B. Conceptual reception

Attendees resonated well with the collaborative approach. While some were slightly skeptical at the nuts and bolts considerations for how this project would adapt to non-flat surfaces or highly asymmetrical rooms, most were quite receptive to the idea of a dynamically expanding collaborative space. Some attendees posed the question "Why not VR?" or "Why a phone?", finding it initially difficult to understand why projectors and a phone was used as the components of this new display modality. However, most agreed that leveraging technology that people already carry daily on their person was the right approach for a system aimed at improving spontaneous collaboration.

C. Frustrations

Most attendees, while understanding the time constraints of developing university projects, were to some degree frustrated with the "offset" that is most likely caused by the system's overbearing touch-input estimation when translating between the phone's camera and the screen-space coordinates. More precise touch-input translation would greatly decrease the level of frustration in respondents.

Likewise, the lack of support for landscape interaction was a point of contention. Most attendees found themselves either starting in landscape or wanting to switch between the two in particular to try and capture all ArUco markers in frame. The lack of horizontal space when moving elements on a wide-screen aspect ratio display caused user frustration.

These findings suggest more rigorous and extensive usability evaluations of such a system in a more natural setting are warranted.

V. Discussion and Challenges

A. Phone Camera

While not ideal, sometimes we have to work with the hardware we have. As such, the phone which was available for the research presented in this project had several hardware issues. One of these was the camera. The camera quality was poor, and attempts to improve the image quality output of the phone were not possible. As such the marker-based system might be more intrusive due to the size increase necessary to perform identification to mitigate the poor image quality of the camera. Smaller markers would not only improve the subtlety of the marker-based approach but also make it easier to fit markers in-frame. As part of future work on this project, obtaining more stable and modern hardware would be a fitting start.

B. Structured light marker-based screen detection

I had initially envisioned a minimally intrusive marker-based detection system to identify the screens in the collaborative space. Structured light seemed like a perfect solution for this type of problem, and as such initial prototyping measures were made with structured light marker-based detection in mind. Once it came to implementation and testing, structured light through alpha channel manipulation (modulating the alpha channel in interpolated frames with a human-imperceptible pattern) were not successful. The interpolated pattern made with the alpha-channel manipulation were extremely difficult to capture on an already hardware-challenged iPhone, but the complexity that came with the interpolated pattern dissuaded me from further structured light experimentation.

As such, ArUco markers were chosen for a simpler albeit slightly less subtle marker-based identification approach to screen identification.

C. Homography Estimation

Homography estimation is required for translation between the relative frame position of the phone's camera, and the detected screen. We use the homography estimation to translate the control surface touch input coordinates to the on-screen coordinates for the unity scene. Re-calculating the homography estimation every frame, once we have assured that we have a screen in frame, makes a lot of sense when we are concerned with the user's manipulation of the object on screen. Ideally, we would like to recalculate for all relevant

frames in order to have the most accurate homography for the current user's frame.

With the constant movement of a handheld device, the homography calculation can cause inaccurate homography estimations. Specifically, when we're dragging the object using the phone we are moving the phone while dragging and a constant recalculation of the homography estimation will eventually cause the touch input's relative positioning to behave erratically due to the physical movement. Several measures can be taken to resolve this issue, and for this project a solution was found by quantifying the quality of the homography estimation. We do this by projecting the center of the screen through our homography matrix H and then performing a check to see if we are within the boundaries of the screen with our homography coordinates. This ensures a good homography estimation, which we lock in as the last "good" homography estimation once a hold and dragging touch input is detected. This allows for better touch-input to screen coordinate stability when dragging the windows in the scene.

D. ARM64 platform

For this project, a macOS device with an ARM64 based chip was used. Given the cost of the commercially licensed OpenCV package available for Unity a free alternative is available to use in Unity. Initially, the project did not require a python server to perform OpenCV frame calculations. This however became necessary when the ARM64 platform had gone unsupported for this OpenCV library for Unity. Several attempts were made to find a solution that did not entail another system component, but this was not possible. As a consequence of the choice of platform, an OpenCV Python server became necessary. In future work on similar efforts either swapping to a x64/86 based system or paying for the official "OpenCV For Unity"[11] package would simplify the system architecture.

VI. Conclusion

The system performance is stable, and initial feedback is very positive. A more polished demo environment with a specialized evaluation approach could yield meaningful advancements in collaborative work efforts in AR for multi-display environments. Such advancements would however require appropriate investments as outlined in this report. Such future endeavors would indeed move us closer to "The Ultimate Display".

References

[1] I. E. Sutherland, "The Ultimate Display," in *Information Processing 1965: Proceedings of IFIP Congress 65*, Spartan Books, 1965, pp. 506–508.

[2] Wikipedia contributors, "Ivan Sutherland's head-mounted 3D display — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Ivan_Sutherland%27s_head-mounted_3D_display

[3] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart, "The CAVE: Audio Visual Experience Automatic Virtual Environment," in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, in SIGGRAPH '92. New York, NY, USA: Association for Computing Machinery, 1992, pp. 65–72. doi: 10.1145/129888.129892.

[4] Unity Technologies, "Unity Game Engine." [Online]. Available: https://unity.com/

[5] Unity Technologies, "Apple ARKit XR Plugin." [Online]. Available: https://unity3d.com/

[6] OpenCV Team, "OpenCV-Python: Open Source Computer Vision Library." [Online]. Available: https://github.com/

[7] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and R. Medina-Carnicer, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[8] O. Kalachev, "Online ArUco Markers Generator." [Online]. Available: https://chev.me/arucogen/

[9] OpenCV Team, "OpenCV: Detection of ArUco Markers Module Documentation." 2026. [Online]. Available: https://opencv.org/

[10] OpenCV Team, "OpenCV: Basic Image Transformations — findHomography Documentation." 2026. [Online]. Available: https://opencv.org/

[11] Enox Software, "OpenCV for Unity." [Online]. Available: https://assetstore.unity.com/packages/tools/integration/opencv-for-unity-21088