Senior Project

InstructAR: Building a Deliverable Infrastructure of How-to Kits for Assembly Scenarios in Augmented Reality

Linda Zhu
Advisor: Aline Normoyle, Swarthmore College

Submitted In Partial Fulfillment of the Requirements of the BA in Computer Science
Bryn Mawr College
Spring 2020
Abstract

Augmented Reality (AR) has been widely used in educational contexts, a beneficiary of the increasingly accessible hand-held devices, because of the immersive experience it creates. This project focuses on building a program infrastructure that uses AR, combining with computer animation, to simulate auto-assembly processes. Using image targets, the system recognizes construction objects and presents the simulation in AR, providing a virtual guidance for the user during assembly. For implementation, the prototype uses Soma cubes as an assembly example and is developed in Unity3D game engine with Vuforia.
Acknowledgements

I want to thank my project advisor Aline for her guidance and mentorship. She not only provided me with a lot of technical support in the development process and feedback on my progress but also saved me in a potential hardware crisis at the very last moment. I also want to thank my Senior Conference professor Deepak Kumar, who checks in with me regularly to make sure I am on track of fulfilling my goal of doing this project.
## Contents

1. Introduction .................................................................................................................. 7
2. Background..................................................................................................................... 7
3. Related Work.................................................................................................................. 8
   3.1. ARSoma ............................................................................................................... 8
   3.2. Cube AR ............................................................................................................. 9
   3.3. AssembleAR ....................................................................................................... 9
   3.4. AR Build Assist ................................................................................................ 9
4. Methodology.................................................................................................................. 10
   4.1. Software Architecture ...................................................................................... 10
   4.2. Animation Algorithm ...................................................................................... 11
   4.3. Use Case ........................................................................................................... 14
5. Results & Analysis ........................................................................................................ 14
6. Future Work.................................................................................................................. 14
7. Summary....................................................................................................................... 15
References......................................................................................................................... 16
Appendix........................................................................................................................... 17
1. Introduction

When it comes to the assembly of complicated objects such as IKEA furniture and LEGO models, people often find it frustrating and confusing to read the instruction manuals. Although the companies that sell these products have tried to make their instruction manuals as lucid as possible, it is, nevertheless, difficult for the user to digest a large amount of information. There are many video tutorials online that show the user the actual construction process. These video tutorials are helpful in terms of informative guidance but not so much in allowing the user to build objects at their own pace. Additionally, a video cannot point out objects in the world, or change its viewpoint to match the user. With the advancement of immersive technologies such as Augmented Reality (AR), people apply this type of technology for pedagogical purposes. In this project, we describe an infrastructure for creating AR assembly simulation based on the physical instruction manuals. The goal is to provide a basic architecture that can be reused in software development for various assembly scenarios. We use Soma cubes, a 3x3x3 dissection puzzle, as the experiment target to design the architecture of our system. The application was first designed on a desktop, and then converted to AR. The end product is a functional AR prototype of Soma cube assembly that recognizes objects with QR code labels and simulates the puzzle recovery solutions.

2. Background

For most people who have bought IKEA furniture, the frustration of assembly is difficult to mediate. Unlike playing with toys, the construction of complicated objects for pragmatic purposes favors a fast, accurate, and preferably pleasant process. Even though companies like IKEA have provided detail-oriented instruction manuals with concise descriptions and clear illustrations, reading through the instructions, and trying to understand them can still cause frustration and confusion that is ultimately counterproductive. Solutions such as video tutorials as a form of mutual support within the consumer community do make the installation easier but are not helpful enough. Without a “hand-in-hand” guidance or a 3D immersive experience where the user can walk around to see the layout and orientation of the components, plus automatic bookkeeping of the progress, it is hard for the user to achieve high efficiency in self-assembly. However, thanks to the emerging technologies of mixed reality and the accessibility of hand-held devices with a camera, we can use AR to enhance the learning experience of assembling physical objects, thus eliminating concerns among unskillful assemblers.

The reason for choosing Soma cube to design the infrastructure is that Soma cube represents a simplified example of assembly scenario. It is a 3x3x3 dissection puzzle composed of seven different pieces (Figure 1). Each piece consists of a unique combination of three or four unit cubes. There are 240 distinct solutions of recovering a 3x3x3 cube by assembling the seven
pieces, not counting symmetries. The seven pieces can also form unique shapes that resemble real-life figures. The Soma cube solutions are simplified versions of assembly instructions for large objects. It is reasonable for us to use Soma cube to focus on building the system architecture without worrying about digitizing the amount of information in complicated instructions.

![Figure 1: The assembled Soma cube (left) and drawings of the seven pieces (right) [1].](image)

3. Related Work

The digital community has experimented with AR for both entertainment and non-entertaining purposes. The following projects are currently proof-of-concepts to various construction scenarios using AR, from which my work draws on the software model and user interface (UI) design. ARSoma and Cube AR are related to cube puzzles. AssembleAR and AR Build Assist are case studies that serve as references for the expected end product of integrating our AR system into more complicated assembly processes e.g. IKEA furniture.

3.1. ARSoma

A mobile application that animates solutions of Soma cubes using AR [2]. The user is able to pan the camera around the room, focusing on horizontal surfaces like the floor or a tabletop. When ARSoma recognizes the flat surface, it will add the seven puzzle pieces to the scene and repeatedly assemble the cube. As this app was designed specifically for iOS devices, it was built solely with ARKit and SceneKit, Apple’s frameworks that produce AR experiences and 3D animated scenes and effects. The 240 solutions are produced by a recursive-descent algorithm. ARSoma then reads all the solutions into an array, and then plays them randomly. The concept and main functionality of this app coincide with my project idea. Its simple use cases provide an insightful reference for my system prototype. However, this app has some problems of software development and maintenance, such as high latency in recognizing flat surfaces and a malfunctioning feature of manual puzzle manipulation. To avoid these issues, my project will
use QR codes for object recognition and discard the manual rotation of each piece because we assume that in a real-life situation, if the position and orientation of a physical component changed during assembly, it is easy for the user to place it back according to the AR simulation.

3.2. Cube AR

A mobile application of AR Rubik’s Cube solver [3]. The user first scans the six faces of a Rubik’s Cube. After scanning the cube, the app guides the user with arrows how to move the layers, and repeatedly calculates the next step once the user finishes a move until completion. Cube AR provides clear instructions on the start of application and avoids manual input in the app. The easy-to-follow guidance indicated by arrows serves a good example for my UI design. The progress tracker by displaying the completion percentage on the screen is another in-app feature that we will integrate into the end-product. As both apps mentioned above are catered towards iOS devices, my project expands the compatibility to all mobile operating systems, including Android, iOS, Windows Phones, and Tizen, by building the infrastructure on Unity3D which has compatibility in deployment to all the previous mobile systems. Cube AR is built with OpenCV library with Objective-C/swift codes. Without any dependence on the AR frameworks, it requires an extensive degree of key implementation for image recognition. Considering the time constraint of this project, we use Vuforia Engine, a widely used AR platform that already encompasses a mature ecosystem, to achieve image recognition as a more convenient approach.

3.3. AssembleAR

A proof-of-concept mobile application [4]. It provides two modes of simulation: a virtual preview of the complete assembly and real-time assistance. The virtual preview simply animates all the components to its final position in the furniture as what ARSoma presents. In the real-time assistance mode, by recognizing the object to be assembled via SKU and a flat surface, the app simulates the construction on the side of the real physical process. The user swipes on the screen to proceed to the next step. AssembleAR focuses on an artistic digital production, which will be different from our project. The various furniture parts and tools were modeled by hand in Cinema 4D, a 3D software suite for artists. The main functionalities were put together using a collection of Adobe software: Premiere, After Effects, Illustrator, Photoshop, and Audition. Indeed, the cartoonish render style that referenced the original manual appearance adds a nostalgic touch on the app UI. Nevertheless, the manual modelling of objects exposes the drawback or potential problem that we need to tackle in the future when our AR system is implemented for complicated construction scenarios. Although not a furnished product, AssembleAR has an inspirational design that our project could draw on; it enlarges the animation of small components such as screws and hinges to provide more detail for the user.

3.4. AR Build Assist
Another similar IKEA furniture AR assembly app [5]. It launches the assembly experience with a scan of the product package barcode. The app’s true-to-size 3D models aid the user in the correct steps to take, with guided animation that elaborates on manual handling of the components. AR Build Assist continuously feeds in the instruction for the next step until the product is finished, while providing options to disassemble a previous component. It is also able to store a personal backlog of the user’s product builds. AR Build Assist was conceived entirely in Unity3D with ARKit. Its interface is dynamic and context-based, enabling the user to focus on what is important and follow the correct steps. AR Build Assist is currently undergoing user testing. As in the development of AR Build Assist, relying on Unity3D allows the performance of the 3D assistive animations to directly connect with the app UI and functionality. Component. This further proves the importance of a powerful development platform, in addition to compatibility with different devices.

4. Methodology

4.1. Software Architecture

We reference the design principle of Multi-Tier architecture (also referred to as N-Tier architecture) [6] to organize our program resources and major functions. A widespread use of Multi-Tier architecture is the simple Three-Tier pattern, which separates the software components into three categories listed from the top level to the low level: Presentation, Application, and Data. The Data tier performs all the data processing and management, such as storing, parsing, and retrieving information from the external data source. The Application tier encapsulates various kinds of application logic and domain concepts, such as data calculation and sorting, and decision making when interacting with the Presentation layer. The Presentation tier is a tier the user can access directly, such as a desktop UI, a web page, etc. This layer receives user input and displays output, if any, generated or updated by the Application tier. Each tier only communicates with and has knowledge of the adjacent tier.

In our system design depicted in Figure 2, classes that store and process puzzle solutions reside in the Data tier. Each puzzle solution object is composed of the end positions and end orientations of the seven block pieces. All the puzzle objects are stored in a library of solutions. The processor class Solution Player resides in the Application tier. It accesses data of the chosen puzzle object retrieved from the library and animates its solution. The Solution Player also listens to the Presentation layer for any command, and handles the pause, resume, and exit animation functionality while bookkeeping the user’s build progress. The UI elements such as button, text, and dropdown, along with the classes that handle the interaction between different screens reside in the Presentation tier. Our application contains two major screens: The Welcome Screen and the Main Menu. The Welcome Screen is the first interface displayed when the user
enters the program. The user is able to proceed to the Main Menu by clicking the Start button. In
the Main Menu, i.e. the animation screen, the user is presented with four interactive features: a
Start button, a Pause/ Resume button, a dropdown list containing all the solution choices in the
library, and an Exit button. All the updates in the UI elements are directly sent back to the
Solution Player, who responds correspondingly. Lastly, besides a complete virtual camera
simulating the controller that can be turned off, an AR Camera is attached to the scene for image
and ground plane targets detection in the physical environment to decide where to display the 3D
models and animate the assembly solutions.

Figure 2: UML diagram of the software architecture: classes in blue constitute the Data
tier; Solution Player class in green is the only member of the Application tier; classes in red
constitute the Presentation tier; classes in black below are the main components for object
recognition in AR.

4.2. Animation Algorithm
We use linear interpolation to animate puzzle components. In computer animation, interpolation is a common practice of producing transition between key frames. Key frames are manually created settings of fixed time points for which the computer generates the animation curves in between. The Unity scripting API has a built-in function called Lerp, which linearly interpolates between two endpoints by a certain fraction called interpolant $t$ [7]. The interpolant parameter $t$ is used to find a point some fraction of the way along the animation curve between the endpoints (e.g. to change properties of an object gradually based on time), thus usually clamped to the range $[0, 1]$. Figure 3 provides a simple example of linear interpolation of 2D vectors. For two positions $\mathbf{P}_0$ and $\mathbf{P}_1$, the linearly interpolated vector $\mathbf{P}(t)$ is given by the function below:

![Figure 3: Linear interpolation of 2D vectors.](image)

$$\mathbf{P}(t) = (1 - t)\mathbf{P}_0 + t\mathbf{P}_1$$

where $0 \leq t \leq 1$.

$x_t, y_t$ are parameterized in terms of $t$.

As object positions are represented in 3D vectors in Unity, we use the Lerp function to animate component movement.

For object orientation, we use quaternions to represent rotations. Compared to Euler angles and matrices, quaternions are more compact number systems to represent rotation and can be interpolated easily without suffering Gimbal Lock, the loss of one degree of freedom in a three-dimensional mechanism that usually occurs in animating rotation using Euler angles and matrices [8]. Unity has a built-in interpolation method for rotation called Slerp, a shorthand for spherical linear interpolation [9]. A Slerp path is the spherical geometry equivalent of a path along a line segment in a 2D plane [9, 10] (Figure 4). A quaternion of norm one is called the unit quaternion. Given two unit quaternions, $\mathbf{q}_1$ and $\mathbf{q}_2$, the linearly interpolated quaternion $\mathbf{q}(t)$ is given by
Figure 4: Linear interpolation of quaternions [10].

As shown in Figure 5, $q(t)$ does not maintain the unit length of $q_1$ and $q_2$, so we need to renormalize at each point by using the function

$$q(t) = \frac{(1-t)q_1 + tq_2}{\| (1-t)q_1 + tq_2 \|}.$$

We use the Slerp function in the Unity Scripting API to animate the component rotation trajectory.

Another key implementation of our animation algorithm incorporates coroutines and yield semantics. As previously mentioned, animation is basically a series of continuous updates spanning several key frames. When you call a function, it runs to completion before returning. This effectively means that any action taking place in one function must happen within a single frame update; the function call cannot be used to contain a procedural animation or a sequence of events over time. It is possible to handle situations like this by using a state machine and a variable to keep track of our state (e.g. timer) frame by frame. However, it is more convenient to use a coroutine and yield semantics for this kind of task (See Appendix for sample code of how a state machine can be implemented with a coroutine using yield semantics). A coroutine is like a function that can pause execution and return control to Unity but then to continue where it left off on the following frame [11]. The execution of a coroutine can be paused at any point using a yield statement. In Unity, IEnumerator, yield, and etc. are C# language features that are used for somewhat of a different purpose with coroutines. When a yield statement is used, the state is automatically saved for us and the coroutine will resume at the next frame. Therefore, coroutines are suitable for modeling behavior over several frames. Furthermore, coroutines have virtually no performance overhead. With coroutines and the yield semantics, we are not only able to achieve smooth animation but also to pause and resume in the middle of a simulation.
4.3. **Use Case**

Here I provide a scenario that describes a sequence of interactions between class objects and between processes.

1. The user has the image targets laid out.
   → e.g. with each image target labelling a puzzle solution
2. The AR camera scans and detects one of the image targets.
   → The application retrieves the data of the corresponding puzzle solution from the library.
3. The application projects virtual 3D models of the puzzle blocks in the physical space.
4. The app animates the assembly process piece by piece from bottom to top.
   → The app tracks the current build choice, the rate of completion and the build time.
5. The user clicks pause button.
   → The app pauses the current build process, the rate of completion and the build time.
6. The user clicks resume button.
   → The app continues the previously suspended build and counts the rate of completion and the build time.
7. The user chooses other puzzle objects to build.
   → The app stops the current build, destroys all the components and starts simulation of the newly chosen puzzle.
8. The user clicks exit button.
   → The app destroys all the current build objects and returns to the Welcome Screen.

5. **Results & Analysis**

We implemented our system on a Windows 10 Notebook PC (Intel Core i7, 240 Hz, 8 GB RAM) using the Unity3D game engine [12] and AR framework Vuforia [13]. We use 60 frames per second (fps) as the standard frame rate of animation. In the prototype software, we hardcoded ten puzzle solutions into the library to avoid latency before animation. We experimented with both images and QR codes as the target feature in the workflow of object detection. The AR application is deployed and tested on a Windows PC and an iPhone 6S. Controlled by the arrow keys and the AWSD keys, the virtual camera can walk around the scene watching the auto-assembly from perspectives as the user likes. Currently the application only supports one assembly process throughout the app execution, i.e. app reboot is required for a different assembly simulation.

6. **Future Work**

I would like to support making different shapes with the soma cube. We could also try to auto-generate puzzles for the user to solve. It would be useful to conceive a method to convert the
mathematical representation of the Soma cube solutions into a data structure suitable for the system. That way the program can load all the solutions into the library in the format we hard coded our solutions. For a vivid user experience, we can implement physical collisions between components as if in a real-life situation. Hand avatars (“ghost hands”) that mimic how the user could handle the components and tools are also helpful in making the virtual assembly guidance more user-friendly. We can use procedural animation by implementing inverse kinematics to produce this “ghost hands” effect. Additionally, more research could be done on replacing the object recognition by image target feature with shape detection. Since, the original aspiration was to apply the infrastructure to complicated objects such as IKEA furniture and LEGO models, it would be nice if this prototype could be extended in that direction. As mentioned in the motivation section, creating or accessing the 3D models of IKEA furniture components is something we need to look into. Digitizing the instructions of IKEA furniture models is also something that requires some further thoughts to standardize the data management tier.

7. Summary

To facilitate the learning experience of self-assembly, we proposed to design a software architecture that animates the assembly instructions in AR so that the user can follow the build process at their own pace. Although the previous research has reflected upon the current lack of available virtual resources online that provide help and guidance for self-assembly, there are some case studies that experiment with AR to animate the construction steps in a mobile application. One similarity of all the presented related work is that they are somewhat catered towards one type of mobile platform. Our project wants to expand the compatibility of our infrastructure to most popular desktop and mobile operating systems, so we decided to develop in Unity3D game engine with the AR framework Vuforia. For simplicity, we used Soma cubes as an example of assembly for our project development. In the software design, we applied the Three-Tier Architecture pattern by placing all the puzzle solution data into the Data Management category, the animator into the Logic tier, and the UI elements into the Presentation/UI tier. For the main animation algorithm, we used coroutines and yield semantics to perform smooth animation over a course of multiple frames and to achieve the animation pause/resume feature. We incorporated both an AR camera for image recognition and a virtual camera in the scene that is turned off when AR is in use. The prototype contains ten hardcoded puzzle solutions for the user to choose. The user is able to either move around the scene virtually using the keyboard or walk around physically with the AR camera to watch the animation. Future work could be done on the data management of instruction manuals, physics simulation, and UI design that altogether refine the user experience and extend the prototype to apply to more complicated construction processes.
References

Appendix

Sample code of comparing linear interpolation between two positions using a state machine and using coroutines with yield semantics.

With a state machine, we add a variable to keep track of our state. In this example the state is _time.

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InterpolateStateMachine : MonoBehaviour
{
    public Vector3 start = new Vector3(-10, 0, 0); // start position
    public Vector3 end = new Vector3(10, 10, 0); // end position
    public float duration = 10; // seconds

    private float _time = 0; // keep track of the state

    void Start()
    {
        _time = 0;
    }

    // manual check and update
    void Update()
    {
        _time += Time.deltaTime;
        if (_time < duration)
        {
            float u = _time / duration; // interpolant
            transform.position = Vector3.Lerp(start, end, u);
        }
        else
        {
            // stop when reaching the end position
            transform.position = end;
        }
    }
}
```
With a coroutine, the state is automatically saved when yield is called. Calling "yield return null" returns the function but saves its state. The next time the function is called, it starts where it left off.

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InterpolateYield : MonoBehaviour
{
    public Vector3 start = new Vector3(-10, 0, 10); // start position
    public Vector3 end = new Vector3(0, 10, -10); // end position
    public float duration = 10; // seconds

    void Start()
    {
        // start function AnimateObject as a coroutine
        // and wait until it is completed.
        StartCoroutine("AnimateObject");
    }

    void Update()
    {
        // do something else
    }

    // update position every frame
    IEnumerator AnimateObject()
    {
        for (float time = 0; time < duration; time += Time.deltaTime)
        {
            float u = time / duration; // interpolant
            transform.position = Vector3.Lerp(start, end, u);
            yield return null;
        }
        transform.position = end; // stop when reaching the end position
    }
}