프로젝트 최종보고서

1. 과제 개요

▷ 계획 목표 및 수정 보완 목표

당초 목표	수정·보완 목표
증강현실 상에서의 펜을 이용한 3차원 스케치	증강현실 상에서의 펜을 이용한 3차원 스케치 앱

▷ 목표의 수정 보완 사유

- 전반적인 컨셉은 같으나, 앱 위에서 트래킹을 비롯하여 많은 스케치 기능들이 수행되므로 제목에 앱을 추가하였다.

2. 관련자료 조사 및 분석

▷ Cuboid Marker Pen

증강현실 상에서 스케치 작업을 위하여 일반적으로 사용되는 펜의 형태는 그림 1과 같다. 정사각형 마커를 이용하여 펜을 추적하고 펜촉과의 거리를 실시간으로 계산하여 점을 찍는 방식이다. 버튼과 블루투스 모듈은 없는 경우가 많고, 그리기 작업은 대부분 데스크탑이나 스마트폰의 앱에서 담당한다.

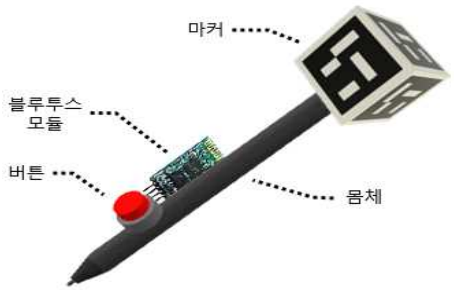


그림 1. 큐브형 마커 펜

▷ Tilt Brush

VR 환경에서 스케치 작업을 할 수 있도록 개발된 툴이다. 가상현실 상에서 3차원 스케치 작업을 하는 데에 용이하며, 많은 기능을 보유하고 있다.

▷ Just a Line

Google의 AR Experiments 프로젝트에 출품된 3차원 스케치용 앱이다. 우리가 진행할 프로젝트의 컨셉과 가장 유사하지만, 많은 단점을 보유하고 있다.

▷ Ramer-Douglas-Peucker Algorithm

선의 보정과 관련된 알고리즘이다. 새로운 점이 그려질 때마다 전체 점을 탐색하여 스케치를 보정하는 알고리즘이다.

▷ 2D Sketch-based 3D Model Retrieval

2차원 스케치를 기반으로 하여 3차원 모델을 검색하는 기법이다. 오픈소스로는 Github의 OpenSSE가 있다.

▷ Cloud Anchors

Google의 ARCore에 적용되어 있는 다자간의 실시간 증강현실 공유 기술이다. 그림 2와 같이 Firebase를 활용하여 앵커(Anchor)와 룸(Room) 그리고 호스트와 게스트들의 정보를 공유하는 방식이다.

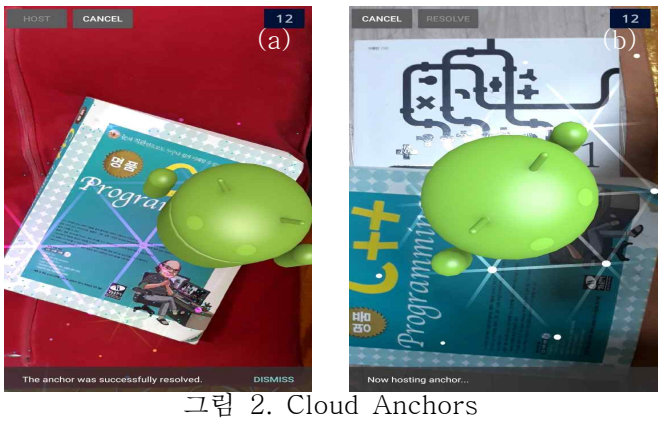


그림 2. Cloud Anchors

3. 연구내용 및 결과

▷ 펜 제작

기존에 구상한 정사각형 마커(Cuboid Marker, Multi-Target Marker) 를 펜 끝에 붙이는 방식은 문제점이 세 가지가 있는데, 첫째로는 마커가 화면에서 벗어날 가능성이 높다는 점이다. 펜의 길이가 길어질수록 벗어날 가능성은 더욱 높아지며, 더군다나 스마트폰의 화면이 작기 때문에 마커를 벗어나지 않게 하면서 그리는 것은 사용자에게 상당한 불편을 초래할 것이다. 둘째로는 아두이노 관련 부품이 겉에 배치되어 미관상 보기 좋지 않다는 점이다. 마커 안에 넣으려면 안이 꽉 채워진 마커가 아닌 내부가 빈 형태의 마커를 써야 하는데, 이런 마커는 전개도로 조립해야하는 특성상 안정성이 떨어진다. 마지막으로는 마커 때문에 펜의 균형을 맞추기 어렵다는 것이다. 이 때문에 펜을 바닥에 놓기도 불편하고, 그릴 때 마커가 스케치를 가릴 수 있으며, 끝에 쏠린 무게중심 때문에 손목이 아플 수도 있다.

그림 3. 기존 방식의 펜

그림 4. 새로운 방식의 펜

이러한 문제점들을 고려하여 새로운 방식의 펜을 고안하게 되었다. 우선 차이점을 열거하자면, 그림 3과 같은 기존 방식이 마커를 끝에 부착하여 트래킹을 수행하였다면, 그림 4와 같은 새로운 방식은 펜 전체가 하나의 마커가 되어 상당히 높은 트래킹 성능을 보인다. 새로운 방식의 펜은 화면에 보여지는 면적부터가 월등히 넓으며, 펜의 일부가 가려져도 트래킹이 유지된다. 또한 마커 자체가 펜의 디자인으로 사용될 수도 있다. 또한 아두이노 관련 부품은 실린더(Cylinder) 내부에 장착할 수 있고, 전체적인 펜의 균형도 안정적으로 유지될 수 있다.

우선 실린더 타겟의 성능을 분석하기 위하여 펜의 몸체 부분을 먼저 만들어보기로 하였다. 펜의 몸체 사이즈는 너무 커지면 사용자 입장에서 쥐기 불편하고, 너무 작아지면 트래킹 성능에 영향을 줄 수 있으므로 중요한 문제인데, 앞의 두가지 조건을 고려하여 경험적으로 생각해보았을 때 지름은 4~5cm, 길이는 10~15cm이면 적합하다고 생각하여 지름 4.5cm, 길이 13cm의 PVC 파이프를 구매하였다. 실제 제품이 아닌 개발을 위한 시제품이므로 강성이 아닌 연성 재질로 구매하였기에 사용자가 펜을 떨어뜨리거나, 펜을 강하게 쥐어 파이프가 찌그러질 때에 대해서는 대비하지 않았다. 또한 구매하였을 때 파이프가 지나치게 약한 구조여서 세 개의 파이프를 내부에 덧대어 보강하였다.

그 후 실린더(파이프)에 부착할 마커(이미지)를 여럿 검토하였는데, 그 중 가장 해상도가 높으며 다수의 Feature Point를 보유한 이미지를 사용하였다. 고수준의 AR 체험을 위해서는 이미지의 스케일을 명확히 입력 및 출력해야 한다. Target Manager에는 미터 단위로 지름은 0.045(4.5cm), 길이는 0.13(13cm)로 입력하였고 이에 따라 실린더의 종횡비(h : 2πr)은 1.087이 되어야 한다. 이 비율을 이용하여 측면 이미지의 해상도를 1024x1024 에서 1113x1024 로 변환해주고 Target Manager에 등록해주었다. 또한 실린더의 상단 및 하단의 이미지는 정사각형이어야 한다.

입력 작업이 끝난 후에는 앞에서 작업한 이미지를 출력하여야 한다. 종횡비에 의하여 측면 이미지의 너비는 14.13cm, 높이는 13cm가 되어야 하고 상하단 이미지의 너비와 높이는 4.5cm가 되어야 한다. 그림 3은 이를 출력한 뒤 실린더에 부착하여 완성된 최종 몸체이다.

그림 5. 실린더의 몸체

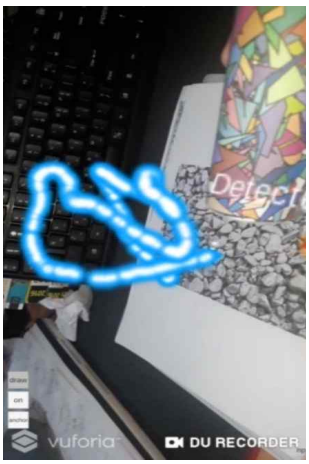


그림 6. 스케치 사진

브러쉬의 위치는 실린더의 상단이나 하단, 그 안에서 중앙이나 구석 또는 바로 앞이나 약간 떨어진 곳 등 다양한 위치에 전략적으로 배치될 수 있다. 일반적인 사용자 입장에서는 브러쉬가 상단 또는 하단에서 약간 떨어져 있으면서 구석에서 살짝 튀어나오게 배치되어야 펜을 세웠을 때도 무난하게 보일 것이다. 또한 사용자는 브러쉬가 있는 부분을 보면서 작업할 확률이 높으므로 가장 Feature Point가 밀집되어있는 쪽의 구석에 브러쉬를 배치하여 안정적인 트래킹이 가능하도록 하였다. 그림 5의 좌측 상단에 브러쉬의 위치를 표시하였다.

그림 6은 실린더를 사용하여 스케치 작업을 수행한 사진이다. 실린더 형태인데도 트래킹 성능이 꽤 좋은 편이며, 펜의 일부분이 가려지거나 기울어져도 문제없이 트래킹이 된다. 다만, 맨 처음 트래킹을 할 때 약 10초간의 보정작업이 필요하다는 점이 아쉽다.

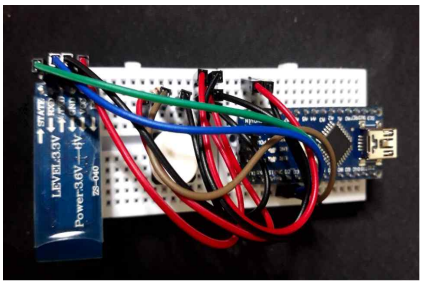


그림 7. 소형화된 아두이노 장치

기존에 구성해둔 아두이노 관련 하드웨어들의 크기가 크므로, 펜 내부에 장착될 아두이노 관련 부품들을 소형화하기위해 이전에 구매해두었던 미니 브레드보드(170홀), 점퍼케이블(수-수), 아두이노 나노 및 전용 USB를 이용하여 작업을 시작하였다. 먼저 미니 브레드보드 2개를 세로로 연결하고 아두이노 나노, 푸쉬버튼, 블루투스 모듈을 장착하였다. 브레드보드 크기가 한정되어 있으므로 최대한 집약적으로 배치하였고, 일단은 고정형 점퍼 케이블로 연결하였지만 차후에 글루건으로 접착력을 높일 수 있다.

기존 아두이노 우노에 업로드 된 소스를 나노에 컴파일하였는데, 계속해서 업로드에 실패하는 현상이 있었다. 이는 스택오버플로우의 답변을 참고하여 프로세서를 ATmega328P(Old Bootloader)로 바꾸어 해결하였다. 신형이 115200 Baudrate를 지원하는데에 비해서 구형이 57600 Baudrate를 지원하는데 그로 인한 속도차이에 의해서 발생한 문제이다. 그리고 블루투스모듈이 BTChat에서는 인식이 되는데, 앱에서는 인식이 안되는 문제도 있었다. 이는 여러 테스트를 해보며 결국 HC-06 모듈 상의 문제라고 판단하였고 새로운 모듈로 바꿔주어 해결하였다.

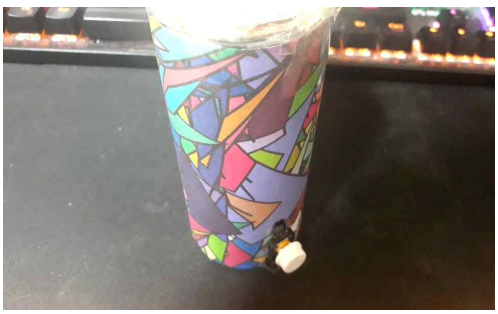


그림 8. 완성된 펜

그림 9. 펜의 내부

그림 8과 그림 9는 완성된 펜의 모습이다. 펜의 스펙은 다음과 같다.

지름 : 4.5 cm

길이 : 13 cm

무게 : 1kg 내외

마커의 종횡비 : 1.087
주요 부품 : 아두이노 나노, 푸쉬 버튼, 블루투스 모듈, 배터리 홀더

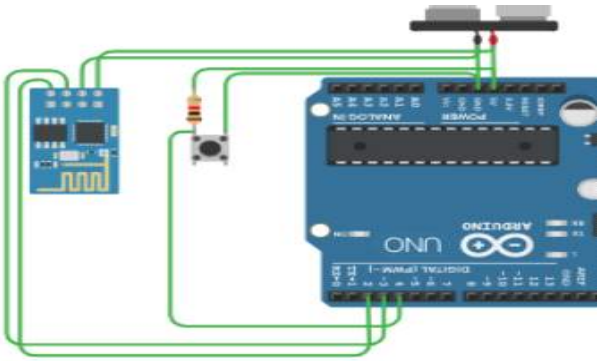


그림 10. 내부 모듈 회로도
(실제 부품과 모양이 다소 다를 수 있음)

그림 10은 펜 내부에 삽입되는 모듈의 회로도이다.

펜과 앱 사이의 블루투스 연결은 다음 매커니즘에 따라 동작한다.

푸쉬버튼을 누른다 → 펄스가 아두이노 우노로 전달 → HC-06으로 다시 전달 → 블루투스를 통하여 스마트폰의 안드로이드 OS로 전달 → (아두니티 플러그인이 탑재된 유니티로 컴파일된) 뷰포리아 앱에서 신호를 읽어 기능을 조작

```
private BluetoothHelper BTHelper;
public bool isPush = false;

void Start()
{
    BTHelper = BluetoothHelper.GetInstance("HC-06");
    BTHelper.OnConnected += OnBluetoothConnected;
    BTHelper.OnDataReceived += () =>
    {
        string str = BTHelper.Read();
        GetComponentInChildren<Text>().text = str;
        if(str[0] == 'P')
        {
            isPush = true;
        }
        else if(str[0] == 'U')
        {
            isPush = false;
        }
    };
    BTHelper.setTerminatorBasedStream("₩n");

    while (!BTHelper.isDeviceFound())
    {
    }
    BTHelper.Connect();
}

void OnBluetoothConnected()
{
    BTHelper.StartListening();
}
```

그림 11. 유니티에서 블루투스 연결을 수행하는 스크립트

```
#include <SoftwareSerial.h>

const int BUTTON = 4;
bool isPush = false; // push & release
bool isDraw = false; // draw toggle

SoftwareSerial softwareSerial(2, 3);

void setup()
{
  softwareSerial.begin(9600);
  pinMode(BUTTON, INPUT);
}

void loop()
{
  if (digitalRead(BUTTON) == LOW)
  {
    if(isPush == false)
    {
      isPush = true;

      if(isDraw == false)
      {
        isDraw = true;
        softwareSerial.println("P");
      }
      else
      {
        isDraw = false;
        softwareSerial.println("U");
      }
    }
  }
  else
  {
    if(isPush == true)
    {
      isPush = false;
      //softwareSerial.println("U");
    }
  }
```

그림 12. 아두이노 IDE 스크립트

그림 11과 그림 12는 유니티와 아두이노 상에서 블루투스 연결 작업을 담당하는 스크립트이다. 이를 통해 완성된 펜과 스케치 앱은 서로 통신이 가능하며, 결과적으로 펜의 버튼을 누를 때는 그리기 작업을 수행하고, 펜의 버튼을 뗄 때에는 그리기 작업을 수행하지 않게 하는 기능이 완성되었다.

목표했던 대로 완성하였지만 일단은 실제 상용 제품이 아닌 샘플이므로 아래와 같이 여러모로 부족한 점들이 존재한다.

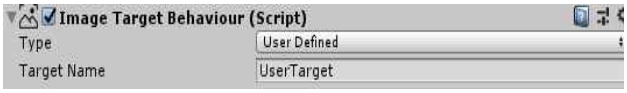
- 겉면(마커)에 일반 종이를 테이프로 부착하였으므로 아예 코팅되어 나온 것보다는 부착 상태가 불량하고 테이프의 광택때문에 트래킹 수준이 어느정도 떨어질 수 있다는 점
- 푸시 버튼의 선이 외부로 나와있고 절연 상태가 좋지 않으며 택트 스위치가 아니라서 조작감이 떨어진다는 점
- 산업용 칩인 PIC가 아닌 아두이노 나노를 사용하고 그 외의 모듈도 전부 산업용이 아니라서 부피가 크고 단가가 상대적으로 비싸다는 점
- 스위치가 별도로 존재하지 않고 배터리 홀더만이 존재하며, 배터리를 끼워서 넣을 때도 불편한데다 애초에 USB 충전방식이 아닌 배터리 충전방식이라는 점
- 뚜껑의 고정 상태가 불량하다는 점

이러한 문제점들을 고려하여 상용을 목표로 보다 개선시킬 수 있을 것이다.

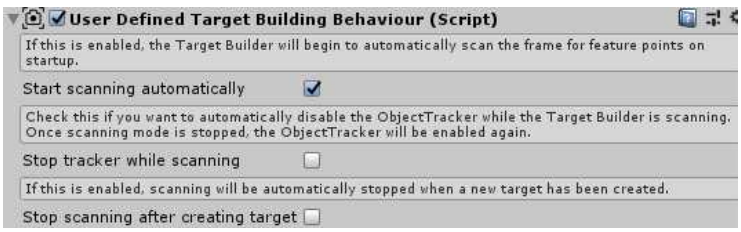
▷ Vuforia 환경 구현

– UDT 기반의 환경 구현

User Defined Target (이하 UDT) 이란, 런타임동안 화면을 지속적으로 Scanning하여 Feature Points를 검색하다가 사용자가 Target으로 적합하다고 생각하는 화면을 찍으면, 실시간으로 그 Target이 내부 DB에 저장되어 즉시 활용할 수 있게 되는 Vuforia의 기능이다. 일반적으로 Target은 컴파일 이전에 개발자에 의해 지정되는데, UDT를 사용하면 사용자가 직접 Target을 설정할 수 있다. 즉, 사용자가 Anchor로 활용할 Target을 가지고 있지 않거나 이를 활용할만한 장소가 아닌 경우, UDT로 대체할 수 있다.



UDT는 크게 세 개의 컴포넌트 즉, UserDefinedTarget, UserDefinedTargetBuilder, TargetBuilderUI로 구성되어 있다. 먼저 UserDefinedTarget은 말 그대로 User가 지정할 Image Target의 작업 영역을 정의한다. 기존의 다른 Target처럼 Scene에 배치하면 되고 다만 UserDefined 타입이기 때문에 DB에서 별도의 이미지를 import하지 않아도 된다. 또한 위의 사진과 같이 ImageTargetBehaviour 스크립트에서만 UDT 타입이 지정가능하기 때문에 실질적으로 Image UDT에 최적화되어있다.



UserDefinedTargetBuilder에는 크게 세 개의 Script가 포함되어있으며 그 중 미리 정의된 UserDefinedTargetBuildingBehaviour에서는 기본적인 설정을 담당한다. Start scanning automatically는 앱이 시작될 때 자동적으로 scan을 수행하도록 하는 설정이며 UDT 모드로 전환했을 때만 작동하도록 하는 별도의 최적화를 수행할 것이 아니라면 사용하는 것이 좋다. Stop tracker while scanning은 앱이 오로지 UDT만을 위해 동작하는 것이 아니기 때문에 사용해서는 안 되는 설정이며, stop scanning after creating target 설정도 target을 한번만 지정하도록 할 것은 아니기 때문에 사용하지 않았다.

그 외에는 GLErrorHandler와 UDTEventHandler가 포함되는데, 특히 UDTEventHandler는 UDT와 관련된 이벤트들을 작성해야하는 가장 중요한 스크립트인데 가장 중요한 메서드는 OnFrameQualityChanged, OnNewTrackableSource, BuildNewTarget 세 가지이다.

```
public void OnFrameQualityChanged(ImageTargetBuilder.FrameQuality frameQuality)
{
    Debug.Log("Frame quality changed: " + frameQuality.ToString());
    m_FrameQuality = frameQuality;
    if (m_FrameQuality == ImageTargetBuilder.FrameQuality.FRAME_QUALITY_LOW)
    {
        Debug.Log("Low camera image quality");
    }

    m_FrameQualityMeter.SetQuality(frameQuality);
}
```

일단 FrameQuality란 화면에 잡힌 FeaturePoints들의 개수, 주변 조명환경 등의 상황에 의해 결정되는 해당 frame상 품질을 의미하며 이 수치가 높을수록 고품질의 AR환경이 구축된다. OnFrameQualityChang

ed는 이러한 FrameQuality가 변경될 때마다 수행되어야하는 작업을 정의하는데 이 부분에서는 SetQuality라는 메서드에 주요 작업을 넘겨주고, 해당 메서드에선 FrameQuality에 따라 UI에 표시되는 상태 바를 사용자에게 보여주는 작업을 수행한다.

```
public void OnNewTrackableSource(TrackableSource trackableSource)
{
    m_TargetCounter++;

    // Deactivates the dataset first
    m_ObjectTracker.DeactivateDataSet(m_UDT_DataSet);

    // Destroy the oldest target if the dataset is full or the dataset
    // already contains five user-defined targets.
    if (m_UDT_DataSet.HasReachedTrackableLimit() || m_UDT_DataSet.GetTrackables().Count() >= MAX_TARGETS)
    {
        IEnumerable<Trackable> trackables = m_UDT_DataSet.GetTrackables();
        Trackable oldest = null;
        foreach (Trackable trackable in trackables)
        {
            if (oldest == null || trackable.ID < oldest.ID)
                oldest = trackable;
        }

        if (oldest != null)
        {
```

OnNewTrackableSource는 새로운 UDT가 생성되었을 때 자동 호출되는 메서드이다. 이 부분에서는 일종의 queue 구조로 저장된 UDT들을 관리하며 유한 버퍼이기 때문에 일정량의 UDT 개수를 초과하면 가장 오래된 UDT부터 삭제하게 된다. 또한 Build 명령을 통해 생성된 UDT를 저장하여 DataSet에 넣고, 다시 Scanning 단계로 진입하는 작업도 이 메서드에서 수행한다. 일반적으로 UDT는 앞서 설명한 대로 여러 개를 관리할 수 있고 이를 활용해 동시에 여러 Target을 tracking하여 AR환경을 구성할 수 있지만 우리는 오로지 하나의 앵커만 필요로 하기 때문에 MAX_TARGETS를 1로 하였다. 이를 통해 multi-tracking의 부하를 줄일 수 있고, build명령을 수행할 때마다 새로운 UDT로 갱신되어진다.

```
public void BuildNewTarget()
{
    if (m_FrameQuality == ImageTargetBuilder.FrameQuality.FRAME_QUALITY_MEDIUM ||
        m_FrameQuality == ImageTargetBuilder.FrameQuality.FRAME_QUALITY_HIGH)
    {
        // create the name of the next target.
        // the TrackableName of the original, linked ImageTargetBehaviour is extended with a continuous nu
        string targetName = string.Format("{0}-{1}", ImageTargetTemplate.TrackableName, m_TargetCounter);

        // generate a new target:
        m_TargetBuildingBehaviour.BuildNewTarget(targetName, ImageTargetTemplate.GetSize().x);
    }
```

마지막으로 BuildNewTarget은 Build 명령을 내렸을 때 호출되는 메서드이다. Build는 FrameQuality가 Medium이거나 High일때만 실행되어질 수 있다. Low일때는 저품질이므로 메시지 상자를 출력하여 사용자에게 Build할 수 없는 frame인 것을 알린다. 조건이 맞으면 이름을 지정하고 Build 작업을 수행한다.

```
void Start()
{
    button2 = GameObject.Find("Button2");
    targetBuilderUI = GameObject.Find("TargetBuilderUI");
}

void Update()
{
    if(button2.GetComponent<ChangeAnchor>().UDTChecker == true)
    {
        targetBuilderUI.gameObject.SetActive(true);
    }
    else
    {
        targetBuilderUI.gameObject.SetActive(false);
    }
}
```

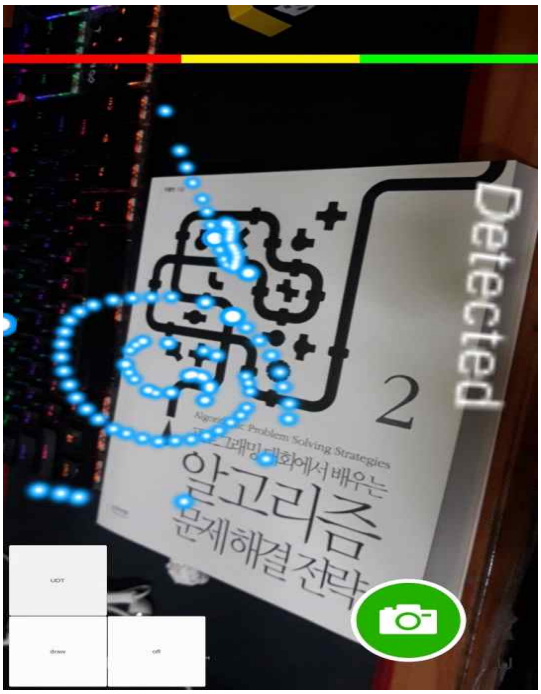
TargetBuilderUI에서는 별도의 버튼을 만들어 Build작업을 수행할 수 있도록 하였다. 해당 버튼을 누르면 앞서 말했던 것처럼 새로운 UDT가 Build되어 추가되며, 이전에 만든 UDT는 제거된다. 또한 상단 바에서는 3단계로 FrameQuality를 보여주어 사용자가 UDT를 생성할 때 참고할 수 있도록 하였다. 물론 이러한 UDT용 UI들은 UDT모드가 사용될 때만 보여져야하므로 위와 같이 외부 Object에서 SetActive를 통해 제어할 수 있도록 하였다.

```
public class ChangeAnchor : MonoBehaviour
{
    public bool UDTChecker = false; // false = anchor
    private TrackableBehaviour userDefinedTarget;
    private TrackableBehaviour anchor;
    private GameObject anchorObject;

    void Start()
    {
        userDefinedTarget = GameObject.Find("UserDefinedTarget").GetComponent<ImageTargetBehaviour>();
        anchor = GameObject.Find("anchor").GetComponent<ImageTargetBehaviour>();
        anchorObject = GameObject.Find("anchor");
    }

    public void OnClickChangeAnchor()
    {
        if (UDTChecker == false)
        {
            UDTChecker = true;
            VuforiaARController.Instance.SetWorldCenter(userDefinedTarget);
            anchorObject.GetComponent<detecting>().trackingCheck = true; // UDT 상태에선 true
            GetComponentInChildren<Text>().text = "UDT";
        }
        else
        {
            UDTChecker = false;
```

UDT모드와 Anchor모드 전환은 ChangeAnchor에서 구현하였다. UI의 특정 버튼을 통해 모드 전환을 Toggle로 수행하며, UDTChecker가 true이면 UDT모드이고 false이면 Anchor 모드이다. UDT모드일때는 SetWorldCenter를 통해 world center가 UDT가 되도록하고, 반대로 Anchor모드일때는 다시 anchor가 world center가 되도록 하였다. world center는 AR 환경의 중심점이 되는 Target을 의미하며 그림을 그릴 때 그 Target과 동기화되어 점들이 놓여진다. 또한 모드 전환시 그린 것들이 꼬이지 않도록 점들을 자동 삭제 시키는 기능도 추가하였다.



위의 사진은 UDT를 이용한 트래킹 사진이다. 좌측하단 버튼을 통해 UDT와 Anchor모드 간에 전환이 가능하고, UDT 모드일 때 위의 상태 바가 최대한 초록색(High)까지 도달한 상태에서 카메라 버튼을 눌러주면 UDT Target이 생성된다. Target에 동기화된 "Detected" 텍스트를 통해 해당 Target이 Tracking 되고 있음을 확인할 수 있다. UDT Target이 Tracking 상태이고 동시에 블루투스로 연결된 버튼이 눌린 상태이면 사용자는 UDT 위에 그림을 그릴 수 있으며 그려진 그림은 UDT에 동기화된다. 동기화되었기 때문에 일반적인 Target처럼 움직여도 그림이 따라오지만 아무래도 런타임 상에 생성되어 노이즈가 낀 이미지다 보니 미리 지정된 Image Target보다는 성능이 떨어지는 단점이 있다.

- Mid Air 기반의 환경 구현

기존에 작업하였던 방식(Image Target 또는 UDT 기반)은 모두 가상공간을 특정 매개체(Target)를 통해 생성하여야 하는 한계가 있었다. Target을 이용한 방식은 스케치를 들고 이동 또는 회전을 해야 하는 상황에서 나름의 이점을 가지고 있지만, 사용자는 Image Target이든 UDT든 간에 Feature Point가 많은 Target를 어떤 형태로든 반드시 가지고 있어야 하며, Target이 트래킹된 상태에서만 스케치 작업을 할 수 있기 때문에 이러한 단점들을 보완할, 즉 Target 없이 스케치 작업을 할 수 있는 또 다른 방식을 필요로 한다.

Ground Plane과 Mid Air는 모바일 AR환경에서 보편적으로 각 플랫폼이 제공하는 일종의 Marker-less Tracking 방식의 기능으로서 앞서 언급한 Target 방식의 한계를 극복하기에 최적화된 기능이다. 일반적인 Marker-less 방식이 그러하듯, 이 기능들 또한 다른 방식에 비해 스마트폰에 내장된 센서(자이로, 가속도 등)에 크게 의존하기 때문에 이에 특화된 ARCore와의 Fusion을 필수적으로 요구한다.

두 기능의 차이점에 대해 언급하자면 Ground Plane은 먼저 카메라로 수평면을 탐지하고 그 수평면 기반으로 가상공간을 생성하는 방식이고, 반면에 Mid Air는 수평면이 아닌 허공을 탐지하여 똑같은 작업을 수행한다. 언뜻 보면 이 기능들 또한 수평면이나 허공이라는 특정 Target을 활용하는 것처럼 보일 수 있으나 이는 Target이라기보다 Anchor라고 하는 것이 더 적합하다. 조금 더 명확히 하자면 Target과

Anchor 모두 가상공간을 생성하는 주체라고 볼 수 있으나, Target이 반드시 카메라를 통해 비춰져야 하는 물리적인 주체를 의미한다면, Anchor는 카메라와 센서의 조합을 통해 만들어지는 가상의 주체이다. 이와 같은 차이에 의해 Anchor는 Target과 달리 일정 영역 안에서는 별도의 Feature Point가 없더라도 계속 트래킹이 유지된다. 왜냐하면 Anchor 방식은 카메라에 내장된 센서를 결합하여 Motion Tracking이 가능하기 때문이다. 그림 13은 Ground Plane과 Mid Air 방식을 보여준다. 사진에선 차이가 없어 보이지만 실제로는 원리나 구동방식 면에서 꽤나 차이가 있다.

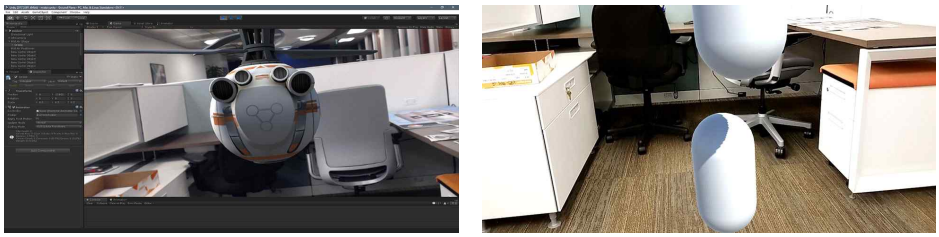


그림 13. Ground Plane과 Mid Air 사용 예시

실제 구현을 위해서는 모든 작업을 총괄할 Plane Manager를 필요로 한다. 이 컴포넌트 내의 PlaneFinder나 MidAirPositioner는 자신들이 가지고 있는 Behaviour 스크립트를 통해 수평면이나 허공을 감지한다. 이 감지 작업의 몇몇 세부 사항은 필요에 따라 조정될 수 있으나, 이 작업에 필요한 핵심 루틴은 Vuforia의 API로서 Low Level에서 작동하기 때문에 우리가 직접적으로 관여할 수는 없다. 이와 별도로 Plane과 Mid Air에 대한 앵커가 존재하는데 우리가 그리게 될 스케치의 각 점들은 이 앵커에 대한 각자의 상대 위치를 가지고 가상공간 상에 놓일 것이다. 그 외에는 Ground Plane과 Mid Air 모드 사이를 전환시킬 버튼을 가진 UI가 필요하다. 보다 고차원적인 작업을 위해서는 3차원 모델을 가상공간에 추가하는 것을 도와주는 ProductPlacement나 터치 방식을 통해 그러한 모델을 이동 또는 회전시킬 수 있는 TouchHandler가 활용되어질 수 있지만 우리의 프로젝트에서는 직접적인 활용 가치가 없다고 판단하여 일단 배제하였다.

그림 14는 유니티 상에서 Ground Plane Anchor와 Mid Air Anchor가 놓인 모습을 보여준다. 이들은 가상공간 상에서 각자의 절대위치를 가지고 놓여진다. 이들의 절대위치는 중요하지 않으며, 각 Anchor 위에 놓여질 점들의 Anchor에 대한 상대위치만이 중요할 뿐이다.

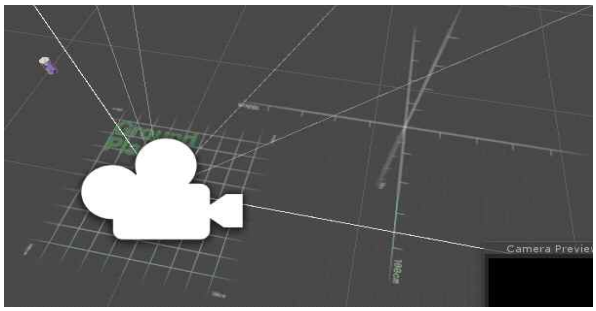


그림 14. Ground Plane과 Mid Air의 Anchor

결국 모든 Anchor와 관련된 스크립트들은 그림 15에 나타난 Draw 작업과 관련된 스크립트의 원활한 작동을 위하여 존재한다. 해당 스크립트에 대한 간략한 설명은 다음과 같다. 블루투스를 통해 펜과 연결된 버튼 또는 앱 상의 Draw 버튼을 눌렀을 때, 펜의 브러쉬가 Tracking Lost 되지 않았을 때, Ground Plane 모드이면 브러쉬 위치에 점을 생성하고 그 점의 부모를 Ground Plane Anchor로 한다. Mid Air 모드이면 브러쉬 위치에 점을 생성하고 그 점의 부모를 Mid Air Anchor로 한다.

```
if (isDrawButtonPushed || scriptBTManager_BackUp.isPush)
{
    if (!scriptPenCylScript.isBrushLost)
    {
        //if (!scriptAnchorScript.isAnchorLost)    // => 앵커 모드
        //{
        //    Instantiate(paint, brush.transform.position, brush.transform.rotation).transform.SetParent(anchor.transform);
        //}

        if(isGroundPlane)  // => GP 모드
        {
            Instantiate(paint, brush.transform.position, brush.transform.rotation).transform.SetParent(anchor_plane.transform);
        }
        else  // => MA 모드
        {
            Instantiate(paint, brush.transform.position, brush.transform.rotation).transform.SetParent(anchor_midair.transform);
        }
    }
}
```

그림 15. 수정된 Draw 스크립트

그림 16은 Ground Plane과 Mid Air를 활용한 스케치 작업의 실제 작동 화면이다.

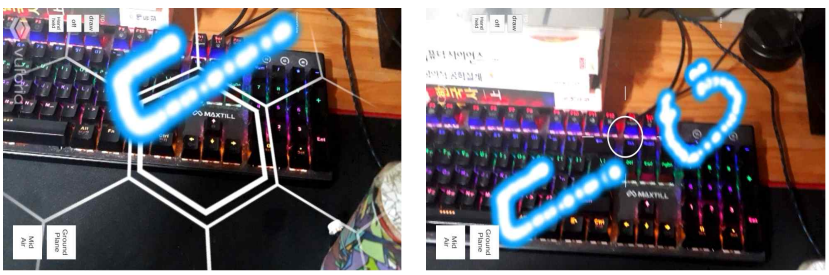


그림 16. Ground Plane과 Mid Air를 활용한 스케치

▷ 트래킹 성능 실험

완성된 펜의 트래킹 수준에 대해서 살펴보자면, 앱을 키고 일정 시간동안은 트래킹이 잡히지 않는다. 이는 Image Target이 켜자마자 바로 트래킹 되는 것과 비교해봤을 때 Cylinder Target의 고질적인 문제라고 유추해볼 수 있다. 그 이후에도 트래킹 수준이 매우 좋다고 할 만한 수준은 아니지만, 적정한 속도로 일정 영역 이상이 드러나도록 움직였을 때 Tracking Loss가 되지 않고, 브러쉬가 올바른 곳에 존재하며, 펜과의 동기화 속도가 빨라서 괴리감이 크게 느껴지지 않는다. 아래의 표 1은 앱이 시작된 이후 펜이 맨 처음 트래킹이 되기까지의 평균적인 소요 시간에 대해, 표 2는 펜이 화면상에 드러난 면적, 펜이 움직이는 속도, 펜의 구도, 토치 사용에 따른 트래킹 유지율에 대해 구체적인 실험을 통하여 정리한 표이다.

* 토치 : 스마트폰에 내장된 손전등 기능

제약 조건	평균 소요 시간 (토치 사용시)
전면	0.64s (0.42s)
후면	0.71s (0.68s)
측면(Edge가 많은 부분)	1.12s (0.97s)
측면(Edge가 적은 부분)	응답 없음 (5.42s)
측면과 후면이 보이도록 기울임(30도)	3.64s (1.38s)
측면과 후면이 보이도록 기울임(60도)	응답 없음 (응답 없음)

표 1. 앱이 시작된 후 맨 처음 트래킹 되기까지의 평균 소요시간
(단위는 s, 10회 기준, 면적은 100%, 속도는 0cm/s, 폰과 펜 사이의 거리는 20~40cm)

기준	제약 조건	트래킹 유지율 (토치 사용시)
화면에 보여지는 펜의 면적 (화면에 보여지는 2차원 이미지 기준)	10%	0% (6.67%)
	20%	23.33% (73.33%)
	30%	70% (93.33%)
	40%	96.67% (100%)
	50%	100% (100%)
펜이 움직이는 속도 (MODE_OPTIMIZE_QUALITY)	15cm/s	100% (100%)
	20cm/s	100% (100%)
	25cm/s	86.67% (100%)
	30cm/s	0% (0%)
펜이 움직이는 속도 (MODE_OPTIMIZE_SPEED)	15cm/s	100% (100%)
	20cm/s	60% (100%)
	25cm/s	40% (93.33%)
	30cm/s	0% (6.67%)
화면에 펜이 보여지는 구도 (20cm/s 기준)	전면	100% (100%)
	후면	13.33% (56.67%)
	측면 (Edge가 많은 부분)	100% (100%)
	측면 (Edge가 적은 부분)	26.67% (46.67%)
	측면과 후면이 보이도록 기울임(30도)	93.33% (100%)
	측면과 후면이 보이도록 기울임(60도)	3.33% (90%)

표 2. 트래킹 유지율

(10초간 작업시, 유지된 횟수 / 전체 시행 횟수, 단위는 %, 각 시행 횟수는 30회)

위의 실험 결과들을 모두 종합해보았을 때, 내릴 수 있는 결론은 다음과 같다.

▷ 앱을 처음 구동했을 때 토치를 켜고 펜의 전면 부분을 통해 트래킹을 시작하는 것이 유리하다.

▷ 토치를 켜고, 펜의 면적은 최소 20% 이상 유지하며, MODE_OPTIMIZE_QUALITY 상태에서 25cm/s 이하의 속력으로, Edge가 많은 부분의 측면을 바라보는 구도에서 작업하는 것이 권장된다.

▷ 다만 MODE_OPTIMIZE_SPEED는 렉이 적고, 재트래킹에 유리하므로 취사선택할 필요가 있다.

▷ 스케치 앱 제작

그림 17. 스케치 앱 UI

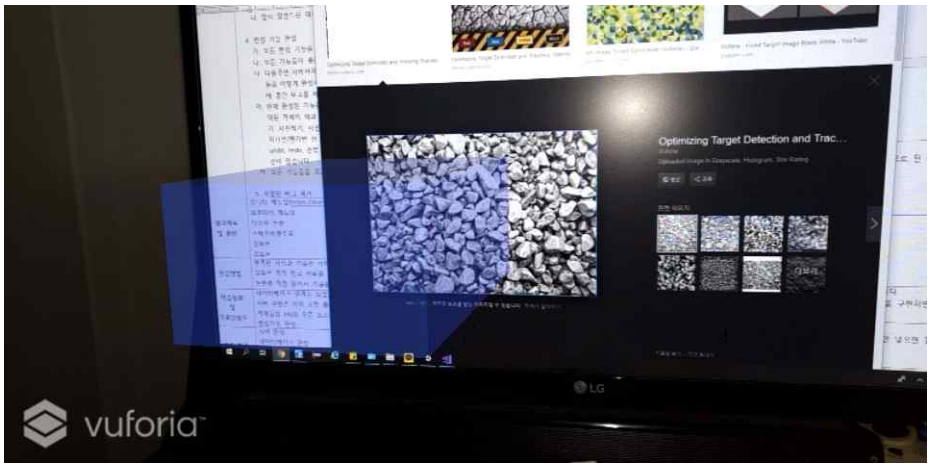
그림 17은 완성된 스케치 앱의 UI이다. 이 앱의 기능들은 다음과 같다.

1) 선단위 드로잉, 선보정 기능: 점단위 드로잉은 그려지는 동안 점 사이가 비어버리게 됩니다. 이러한 단점과 앱의 오버헤드와 지연을 줄이고자 선단위 드로잉과 선보정 기능을 만들게 되었습니다.

2) 객체의 색과 크기 변경 기능: 객체의 색을 변경하는 것은 rgb값을 변경하는 오픈소스 UI를 앱 형태에 맞게 집어넣는 것으로 구현했습니다. ui의 rgb 값을 변경하면 그 변경되는 값을 바로 메테리얼에 적용할 수 있게 하는 연결했습니다. 크기 변경 기능은 객체의 종류에 따라 서로 변경해야하는 가중치가 천차만별이라서 종류에 따라 다르게 적용되도록 만들었습니다. 또한 이후에 편집 모드가 완성되면서 이미 만들어진 객체의 색과 크기도 별도로 변경할 수 있게 재구성했습니다.

3) 객체의 선택 및 다중 선택: UI상에 편집 버튼을 클릭해서 편집 모드에 돌입하게 되면 기본적으로 객체의 선택 기능이 제공됩니다. 화면상의 객체를 터치하면 스크린 상에서 보이지 않는 일직선의 빛을 발사해 이와 물리적으로 충돌하는 콜라이더를 가지고 있는 객체와 이벤트를 발생시킬 수 있습니다. 이것이 레이캐스트이고, 이러한 원리를 특정한 객체를 편집에 이용할 수 있게 선택하도록 만들었습니다. 또한 다중선택을 별도로 가능하게 만들었는데 처음에는 화면을 3초이상 누르면 전환되게 하였으나 이는 실용적이지 않아서 그냥 다중 선택 버튼을 따로 만들었습니다. 다중 선택 모드에서는 선택된 객체를 다시 한 번 클릭하는 것으로 선택을 취소할 수 있습니다.

4) 편집 박스: 그림판이나 포토샵에 보면 존재하는 점선으로 된 편집용 투명 상자가 있습니다. 이것에서 영감을 받아 편집용 직육면체를 고안했습니다. 이 상자 안에 들어온 모든 객체는 다중 선택됩니다. 편집 박스 모드에서 펜을 따라 육면체의 크기가 커집니다. 유니티에서는 3차원 도형의 넓이를 따로 조정하는 방법이 없어서 거의 불가능해 보였지만, 부모와 자식의 관계를 적절히 이용해서 구현에 성공했습니다. 객체를 판별하는 방법은 충돌 판정을 이용합니다.



5) 사진 찍기: 사진을 찍는 것 자체는 그냥 UI를 꺼서 화면을 캡쳐하는 기능을 사용하면 됐지만 유니티에서 안드로이드로의 저장은 쉬운 일이 아니었습니다. 일단 기본적으로 저장 경로가 안드로이드마다 다르며, 공통적으로 쓰이는 저장경로도 자잘한 오류를 일으켰습니다. 가장 큰 문제는 유니티로 만들어낸 사진은 갤러리에 등록이 안 된다는 점이었습니다. 분명 사진은 존재하는데 이 사진을 안드로이드의 갤러리 앱이 인식을 하지 못하게 되는 것입니다. 이를 해결하기 위해 nativeGallary라는 오픈소스를 앱에 맞게 고쳤습니다.

6) 객체 삽입: 직선은 처음 점과 마지막 점을 잇는 방식으로 삽입됩니다. 정사각형과 구체, 정육면체가 원래 존재하는 입체를 프리팹형태로 구현해서 따로 삽입하기 쉽게 만들었습니다. 여기서 위에서 언급한 편집박스와 같이 형태를 자유자재로 바꾸는 기능을 넣었었으나, 이것은 유니티의 한계로 인해 무조건 객체가 부모를 갖게 만들어야하는 형태로 만들어야 했고, 뒤에 나오는 객체묶기라는 기능등과 연동하기에 매우 힘들어지게 되는 형태라서 그 기능은 도입하지 않았습니다. 사진 객체 삽입은 갤러리 상의 사진을 텍스쳐로 변환해서 사각면에 붙여서 불러오도록 만들었습니다.

7) 객체 묶기: 다중 선택된 객체들을 하나의 객체로 재구성하는 기능입니다. 이 기능을 이용하면 모든 편집 기능이 동시에 적용되게 됩니다. 묶었던 객체와 다른 객체를 합치게 되면 묶었던 객체를 풀면서 다른 객체까지 하나로 합치게 됩니다.

8) 객체의 이동: 화면에서 터치앤 드래그로 선택된 객체를 이동시킬 수 있습니다. 이때 사용하게 되는 것 역시 레이캐스트입니다. 선택된 객체에서 카메라에 평행인 면을 안 보이게 생성하고 그 면을 통해 빛을 충돌시키는 것으로 위치를 계산하게 됩니다. 이 위치값과 이때 나오는 실제 위치와의 차이에서 보정값을 얻게 되고, 그 이후 손으로 드래그 한 위치에서 또 한 번 레이캐스트로 면과 충돌시켜서 깊이는 고정되었지만 변화한 드래그 위치로 이동시키게 됩니다. 일반적인 객체들의 이동은 간단하지만 선객체와 묶여있는 합성 객체는 그 위치로의 이동이 아니라 하나의 점의 상대적인 위치 이동으로 나머지 객체를 모두 별도로 이동시키는 방식을 사용합니다. 합성객체 내에서 선 객체는 또 별도로 움직여야 합니다.

9) 객체의 회전: 역시 선의 회전과 일반 도형의 회전은 다릅니다. 일반 도형의 회전은 터치된 손의 움직임을 기준으로 회전을 시키면 되지만 선의 회전은 각 꼭지점 별로 따로 적용해야하고, 기준을 따로 잡아줘야 합니다. 합성객체는 적용하지 않도록 만들었습니다.

10) 2D 투영 그리기: 하나의 투명한 스케치북을 카메라에 평행하게 어느정도 떨어진 곳에 배치하고

레이캐스트를 이용해서 펜과 스케치북 사이의 직선 거리를 이용해서 허공의 펜 위치를 스케치북에 투영해서 그립니다. 공간에서 면을 그리고 싶을 때를 위한 보정기능입니다.

11) 이어 그리기: 선을 이어서 그리고 싶을 때 이미 그려져 있는 선의 끝 점에서 펜의 상대적인 움직임을 따라서 원격으로 그려지게 만들었습니다. 이때 가만히 있어도 발생하는 drift를 제거했습니다.

12) 2D에서 3D 입체로 변형: 2D 투영에서 사용되는 투명 스케치북을 통해서 선을 그리게 합니다. 이때 한 붓으로 그려진 선으로 면을 만들고 그 면에 깊이값을 더해서 입체를 만듭니다. 이때 직접 꼭지점으로 mesh를 만드는 함수를 짰습니다. 입체의 mesh를 만들 때 아직 실력이 부족해서 울퉁불퉁한 모양이 계속되는 경우 mesh가 망가집니다.

13) 펜 기반 편집: 앱이 아닌 펜을 통해서 객체 선택, 이동, 회전이 가능하게 합니다. 선택은 충돌처리, 객체 이동은 펜 위치로의 이동합니다. 여기서도 마찬가지로 객체별로 이동방법이 매우 상이합니다. 회전은 일반적인 도형은 바라보게 하지만, 선 객체는 직접 사이각 계산을 통한 벡터의 차이로 회전하도록 구현했습니다.

14) undo,redo,삭제: undo 스택에에 모든 기능을 두고, undo 작업시 redo 스택에 넣어 임시로 랜더링을 멈추고, redo시 하나씩 다시 랜더링하고 다른 작업을 하면 전부 지워버리는 기능입니다. 원래는 세세한 작업도 명령어와 함께 같이 리스트에 넣어서 기능하게 만들었지만, 별도의 기능들이 너무 많아져서 그냥 객체의 생성과 삭제에 한해서만 undo,redo가 작동하게 만들었습니다. 삭제는 모든 객체를 삭제합니다. 선택된 객체만 삭제하는 것도 가능합니다. 다만 합성 객체는 undo,redo가 먹히지 않도록 설정했습니다.

15) 기타 UI 및 모드 간의 충돌 교정: 많은 기능들이 모두 충돌하지 않도록 교정했습니다.

▷ 스케치 보정 기술 개선

기존의 Ramer-Douglas-Peucker 알고리즘은 모든 점을 반복적으로 탐색하기 때문에 실시간으로 점의 개수를 최소화하기에는 적합하지 않다. 이를 해결하기 위해 사잇각 개념을 도입하여 처리 방식을 개선하였다.

새로운 점 x를 추가하는 과정에서 이전에 작성된 점 o을 기준으로 x와 펜의 위치 p의 사잇각 θ을 계산하고, 이 값이 임의의 상수 α를 o와 p사이의 거리로 나눈 값보다 작으면 x를 삭제한다. α의 값을 조정하면 삭제 허용 범위를 조절할 수 있다. 식 (1)은 이러한 사잇각을 이용한 판별 공식을 실험을 통해 도출한 것이다.

$$if\ \theta_{(x,p)} < \alpha/dist(o,p)\ then\ Remove(x) \quad (1)$$

사이각 판별 공식 도출 과정을 보다 상세히 기록하면 다음과 같다.

1. 우선 사잇각 공식은 0~180도가 산출되는 내적을 이용한 방식을 사용하기로 결정했습니다. 가장 심플하면서 사용하기에 용이한 측면이 많았기 때문입니다.
2. 사이각 판별 공식의 경우 실험을 통해서 40개의 기준점을 사용하여 0.489/(기준점과 펜 위치의 거리)이상의 각도일 경우 새로운 선으로 변경하는 방식을 사용하게 되었습니다.
3. 또한 전체적인 틀을 잡아주기 위한 최초의 기준점(라인 렌더러의 펜 위치를 제외한 마지막 벡터)을 따로 저장하여 0.8/(기준점과 펜 위치의 거리)라는 제한을 두어 전체적인 선의 모양을 잡아주는 역할을 맡도록 했습니다.
4. 0.489/거리 라는 보정치를 결정하게 된 이유는 미세한 움직임에서의 정확도를 고려했기 때문입니다. 0.489 부분의 상수의 값이 작아지면 선의 유연성이 낮아지게 되어 선의 꼭지점이 많이 생성되게 되고, 상수의 값이 커지면 선의 유연성은 높아지지만 그리는 선의 정확도가 떨어지게 됩니다. 작은 그림을 그릴 경우의 정확도를 유지하기 위해서는 0.489라는 정도의 상수가 필요하게 되었습니다.
5. 다만 실제 펜을 사용한 실험에서 얻어진 결과가 아니기 때문에 실제 펜을 이용한 실험에서는 공식이 변할 수 있습니다.

그림 18은 스케치를 구성하는 점들을 드러나게 하여 보정 이전과 이후의 상태를 비교한 사진이다. 직선뿐 아니라 사용자의 손 떨림에 의해 미세하게 꺾어진 선에서도 보정과 동시에 점의 최소화가 수행되는 것을 볼 수 있다. 또한 완성된 객체에 사잇각에 의한 실시간 보정 방식과 기존 보정 방식을 함께 적용하면 최적의 결과가 나오는 것을 실험을 통해 확인하였다.

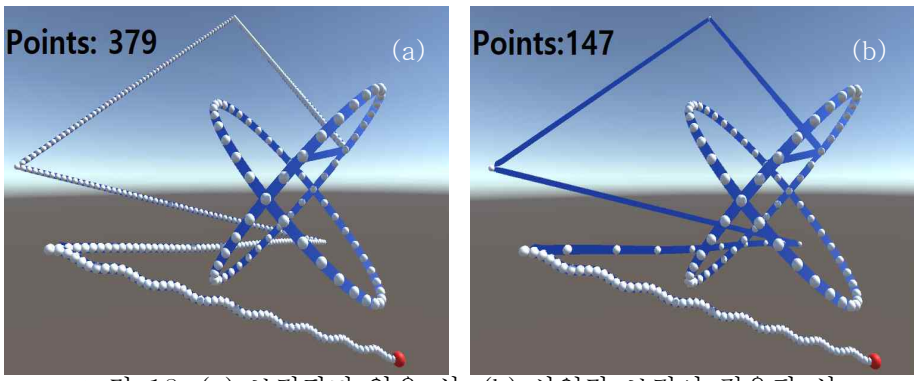


그림 18. (a) 보정되지 않은 선, (b) 사잇각 보정이 적용된 선

▷ 모델 검색 기능

Sketch-based Model Retrieval은 2D 스케치를 통해 3D 모델을 검색하는 방식이다. 담당교수님의 소개로 처음 이러한 개념을 알게 되었는데, 이 방식을 적용하면 앱의 활용성이 한층 더 높아질 것이라 생각하여 구체화시켜보기로 했다. 현재까지의 구상은 다음과 같다.

① 앱 내에서 submit 버튼을 누르면 현재의 3D 객체(스케치)를 2D로 투영시킨 이미지 파일을 저장하여 서버에 제출한다.

② 서버에서는 모델 검색을 수행하고 그 결과를 일치율이 높은 순으로 반환한다.(오픈소스 활용)

③ 해당 결과를 앱에서 받아 화면에 디스플레이하고 사용자가 원하는 모델을 선택하게 한다. 선택하면 이미지를 제출했던 위치에 기존에 있던 3D 객체를 지우고 모델로 대체한다.

물론 이상적인 것은 2D가 아닌 3D 스케치로 검색하는 것이지만, 해당 오픈소스는 사실상 거의 전무하고, 사실 2D 스케치기반 모델검색도 거의 없다시피 하다. 그나마 2D 스케치 기반 이미지검색은 꽤 많은 편인데, 대표적으로 구글의 Quick Draw가 있다. 그러나 우리가 필요한 것은 모델검색이기 때문에 이에 특화된 오픈소스가 필요하다. 2D 스케치기반 모델검색 관련 오픈소스 중 현재 가장 접근성이 좋은 것은 Github의 OpenSSE(Open Sketch Search Engine)이다. 그림 19는 OpenSSE의 사용 예를 보여준다.

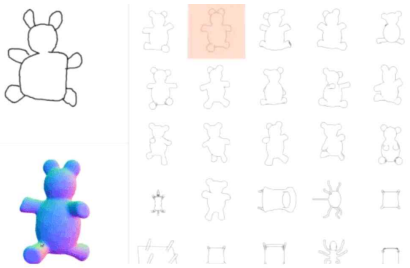


그림 19. OpenSSE 사용 예

대부분의 오픈소스가 그렇듯이, 이 오픈소스도 리눅스 환경에 최적화되어 있기 때문에 먼저 VMWare에 우분투를 설치해주었다. 그 다음 OpenCV 4.0.1을 설치하였는데 계속해서 링크 오류가 발생하였다. 원인을 알 수 없어 고생하던 중 개발자의 Gitter Chat을 발견하고 예전 글을 보던 중 OpenCV 3와 관련된 언급이 많아서 기존 버전을 삭제하고 OpenCV 3.2.0으로 다시 설치하였다. CMakeList가 설치된 폴더에서 cmake와 make를 한 뒤에 make install 할 때도 계속 에러가 났었는데 이는 단순히 sudo를 사용하지 않아서 권한이 없어서 발생한 문제여서 금방 해결하였다. 그림 20은 설치가 성공한 모습이다.

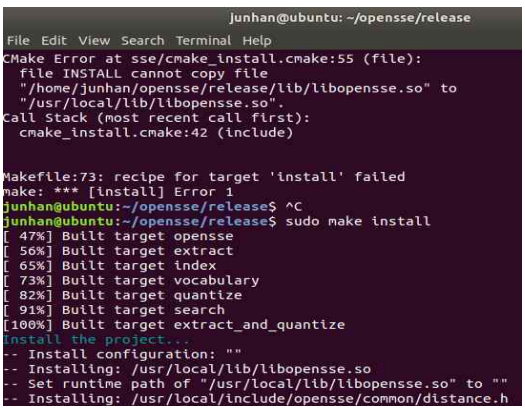


그림 20. OpenSSE 설치

이 오픈소스는 QT (C++ 기반 GUI 제작 프레임워크) 로 제작되었기 때문에 qmake로도 빌드해야했는데 *error while loading shared libraries: libopensse.so: cannot open shared object file: No such file or directory* 에러가 계속 나서 고심하다가 **sudo /sbin/ldconfig -v** 를 통해 최신 Shared Library에 대한 링크와 캐시를 만들어 해결하였다.

환경 세팅을 마치고 SSE를 실행했지만 아무런 응답없이 그저 프로세스가 꺼지는 현상이 반복되었는데, 해당 Repository의 issue 탭을 확인해보니 Dataset이 없을 때 발생하는 문제라고 하였다. 별도의 Dataset이 해당 오픈소스에는 없었고 README.md에서 샘플로 보여준 Dataset인 SHREC 2012가 소실되었기 때문에 SHREC 2009를 구하여 이를 Dataset으로 사용하기로 했다. 사실 이 Dataset은 가장 기본적인 형태의 3D Shape들이 아닌 약간 복잡한 형태를 가진 특정 분야에 편중된 Dataset이었기 때문에 우리의 목적에 완전히 부합하는 것은 아니지만, 일단 테스트를 위해 해당 Dataset을 사용하기로 했다. 다만 이 Dataset은 적은 용량을 지니고 있어 학습시키기가 빠르다는 장점은 있다.

이 3D object 파일들로 구성된 이 Dataset을 SSE의 용도에 맞게 가공하기 위해서는 TriangleMesh라는 SSE에 대한 종속 오픈소스를 사용해야 했다. 이 오픈소스를 사용하면 Dataset의 각 file을 여러 각도에서 투영한 이미지들로 분화시킬 수 있다. 기본 문법은 **gen_view_image modelfile xfdir viewnum imagedir** 이다. modelfile 경로에 있는 off(object file format) 파일을 xfdir에 있는 각 xf파일(모델을 투영하는 각도에 대한 정보를 담은 포맷. 4x4 matrix로 되어있어 OpenGL의 projection matrix와 관련있는 것으로 추정) 기준으로 투영하고, 그 결과로 나온 이미지 파일들을 viewnum만큼 만들어 imagedir에 넣는 작업을 한다. 그림 21은 실행 결과를 보여준다.

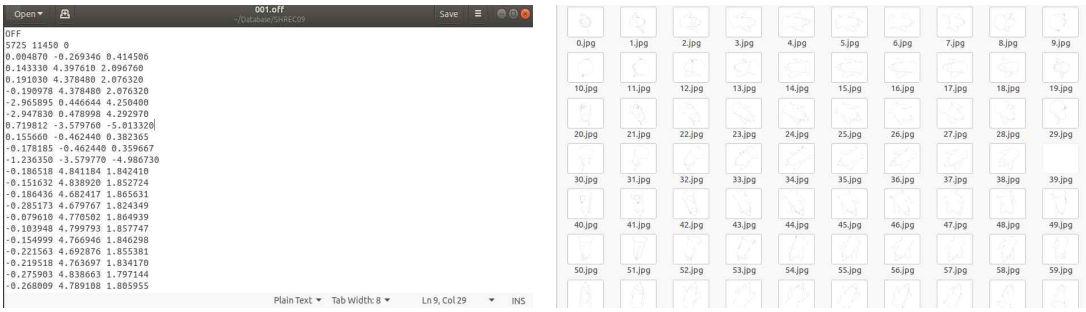


그림 21. off 파일(좌), 여러 각도에서 투영된 이미지 파일들(우)

그 후에는 다음 프로세스에 따라 Dataset을 학습시킨다.

① **$ sse filelist -d ~/Database/SHREC09/ -p "*.jpg" -o filelist** : 모든 jpg를 filelist화 시킨다.

② **$ sse extract -f filelist -o features** : filelist의 각 file에서 feature를 추출한다. (반드시 filelist가 있는 경로에서 수행해야 함) features.txt는 최소 5GB이상의 거대한 크기를 가지고 있다.

③ **$ sse vocabulary -f features -n 100 -o vocabulary** : k-means 알고리즘을 이용하여 file들을 군집화(clustering) 시키는 방식이다. 즉 features.txt에 생성된 값들을 n번의 epoch를 통해 각 file마다 가지고 있는 feature에 따라 군집화시키고, 군집화된 file들의 cluster마다 하나의 vocabulary를 대응시키는 작업을 수행한다.

④ **$ sse quantize -v vocabulary -f features -o samples** : vocabulary.txt의 float값을 samples.txt의 int값으로 일종의 color quantization처럼 양자화시켜 압축시킨다. 이 과정에서 용량이 크게 줄어든다.

⑤ **$ sse index -s samples -o index_file** : index_file을 통해 테이블을 만들어 보다 빠른 접근을 가능하게 한다.

3번 과정에서 계속하여 프로세스가 kill되는 현상이 발생하여 **dmesg | grep -i kill** 을 통해 커널로그에서 killer 메시지를 찾아보았다. **[74737.598582] Out of memory: Kill process 19563 (vocabulary)**

score 384 or sacrifice child 을 보았을 때 OOM(Out of Memory)에 의한 에러임을 알 수 있었다. 그래서 일단 학습시키는 모델의 수를 줄이고, Epoch와 관련된 n을 줄여 수행하니 문제없이 작동하였다. 만약 줄어든 n값에 의해 underfitting이 된다면 차후에 VMWare의 가용 메모리 공간을 늘려야 한다.

이제 만들어진 파일들의 경로를 config.json에 등록해주었다. 이 과정에서 터미널에서 사용되는 home의 단축 기호인 ~을 경로에 적어서 제대로 작동하지 않았었다. 이는 **"/home/junhan/Database/SHREC12/index_file"** 로 수정하여 해결하였다. 그림 22는 최종적으로 SSE가 실행된 모습이다.

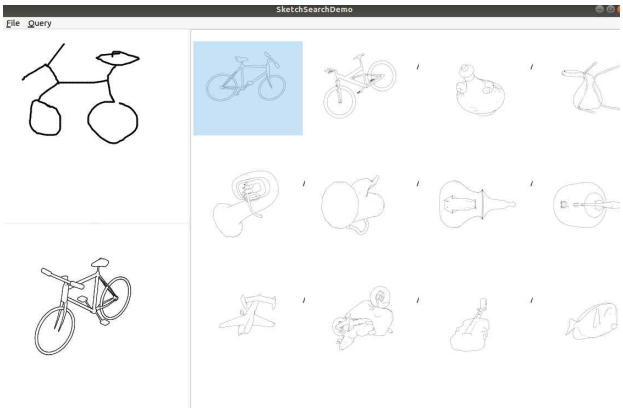


그림 22. SSE 실행 화면

모델들이 다소 복잡한 형태라 AR환경에서 사용하려면 더 간단한 형태의 모델을 사용할 필요가 있다. 또한 나중에 여유가 된다면 train 횟수에 따른 인식률을 정리해볼 수도 있을 것이다.

SSE를 우리의 프로젝트에 활용하기 위한 구상은 다음과 같이 보다 구체화되었다.

▷ 앱에서는 자신이 그린 스케치의 obj파일을 서버에 저장하고, 스케치의 정보를 로컬에 저장한다.

▷ obj파일을 한 곳에서 투영한 이미지나 여러 곳에서 투영한 이미지 세트를 SSE에 input으로 넣어 특정 모델(off)들을 반환받는다. 이들은 off to obj 작업을 통해 obj파일들로 변환된다.

▷ 해당 obj를 클라이언트(앱)에서 받고 아까 저장하였던 스케치의 좌표, 각도, 스케일에 맞춰 obj를 배치한다. 기존에 있던 스케치는 지운다.

off to obj 작업과 SSE에 input과 output을 하는 방식은 보다 구체화될 필요가 있을 것이다. 서버에 작업을 요청하고 결과를 받는 과정은 초단위의 오버헤드를 발생시킬 수 있으나 이 작업은 빈번히 수행되지는 않기 때문에 현재로선 크게 문제될 것으로 보이진 않는다.

SSE를 서버에서 사용하기 위해서는 GUI보다는 Command Line에서 다루는 것이 더 수월하다는 것을 알게 되었으므로 CLI에서의 사용법을 알아보았다.

$ sse search -i index_file -v vocabulary -f filelist -n 10 를 사용하면 특정 이미지가 있는 경로(입력 경로)를 입력하였을 때 이와 유사한 다른 이미지들의 경로를 확률 순으로 n개 출력한다.(SHREC09 폴더에서 작업해야함) 입력 경로는 클라이언트와 연결된 공유 디렉토리가 될 것이고, 여기에는 클라이언트에서 받은 이미지가 들어갈 것이다. 출력된 경로들의 문자열은 파싱된 뒤 그에 대응하는 off파일들을 찾는데 사용된다. 여기서 찾은 파일들은 입력 경로와 별도의 출력용 공유 디렉토리로 복사될 것이다. 이 프로세스를 구현하기 위해 search.cpp를 그림 23과 같이 수정하였다.

```
string pathStr[results.size()];
vector<string> parsedStr(results.size());
for(uint i = 0; i < results.size(); i++) {
    pathStr[i] = files.getFilename(results[i].second).c_str();
    cout << results[i].first << " " << pathStr[i] << " ";

    int viewIdx = pathStr[i].find("view", 0);
    viewIdx--;
    while(pathStr[i].at(viewIdx) != '/') {
        char inserted[2];
        inserted[0] = pathStr[i].at(viewIdx);
        inserted[1] = '\0';
        parsedStr[i].insert(0, inserted);
        viewIdx--;
    }
    cout << parsedStr[i] << endl;
}
```

```
int vecSize = RemoveDuplicatesKeepOrder(parsedStr);
for(uint i = 0; i < vecSize; i++)
{
    string offPath("/home/junhan/Database/SHREC09/" + parsedStr[i] + ".off");
    string destPath(output_directory + to_string(i) + ".off");
    cout << parsedStr[i] << endl;
    cout << offPath << endl;
    cout << output_directory << endl;
    copyFile(offPath.c_str(), destPath.c_str());
}

command = "rm ";
path = input_directory;
system(command.append(path).c_str());
```

그림 23. search.cpp

우선 반환된 각 경로에서 "view"부터 '/'까지의 문자열을 파싱한다. 즉, 경로가 *SHREC09/007view/19.jpg* 라면 007이 파싱된다. RemoveDuplicatesKeepOrder 함수에서는 unordered_map을 이용하여 O(vec.size) 시간에 파싱된 문자열들의 순서를 유지하되 중복된 요소를 제거한다. 순서를 유지하는 이유는 사용자가 제출한 스케치와 일치율이 높은 순서대로 모델을 반환하기 위함이고, 중복된 요소를 제거하는 이유는 같은 모델을 여러번 반환하지 않기 위함이다. 결과적으로 선별된 문자열들을 각 입력 경로에 넣어 off파일에 접근하고, 이를 일치율 순으로 이름을 붙여 출력 경로에 복사한다. 작업이 시작하기 전엔 출력 디렉토리를 비워주고, 작업이 끝난 후엔 입력 디렉토리를 지워준다.

만일 클라이언트와 서버의 작업을 서로 비동기적으로 구현했다면 뮤텍스같은 것이 필요했겠지만, 그 대신 동기식으로 구현하였다. 즉, 해당 루틴에는 isPerforming과 같은 부울 변수가 선언되어야 하고, 이미지가 입력 디렉토리에 들어온 순간에는 false일 것이다. 그 후에 클라이언트에서는 입력을 하자마자 반환된 값을 가져가려고 계속하여 시도할 것이나, 오직 isPerforming이 true에서 false로 바뀌는 순간(작업이 완료된 순간)에만 반환된 파일들을 가져갈 수 있다. 그 외의 경우에는 작업이 끝날 때까지 스핀 상태로 대기해야 한다.

이 방식에서는 사용자가 출력이 끝나기 전에 입력을 여러 번 하더라도 입력 큐(Input queue)를 만들 수는 없으나, 작업의 특성상 사용자가 연속적인 입력을 요구할 가능성이 적은데다가 응답시간이 충분히 빠를 것으로 예상되므로 문제는 없을 것이다.

서버와 클라이언트 간 연결을 위하여 포트포워딩(Port Forwarding) 작업을 해주었다.

211.172.66.177:9009 (공용 주소) →
192.168.0.4:9009 (사설 주소) →
192.168.152.128:9009 (가상머신 주소, VMNet8)

그 후에 서버 부분의 코드를 결합하여 OpenSSE와 합치는 작업을 하였다. 전체적인 작동과정은 그림 24과 같다.

① 클라이언트(유니티)에서 서버로 사용자가 그린 3D 스케치 객체를 2D로 투영한 이미지 한 장을 보낸다. (지원 형식 : JPG, PNG)

② 서버는 그 이미지를 받아서 n장(우리는 9장으로 설정함) 의 유사 이미지를 확률순으로 클라이언트에게 보낸다.

③ 클라이언트에서는 이 이미지들을 순차적으로 UI에 배치하여 사용자가 한 장을 고를 수 있도록 한다. 사용자가 한 장을 고르면 그에 해당하는 번호가 서버로 전송된다.

④ 서버는 그 숫자에 해당하는 이미지의 3D 모델(.obj)를 클라이언트에게 전송한다. 클라이언트에서는 펜의 위치에 모델을 반환한다.

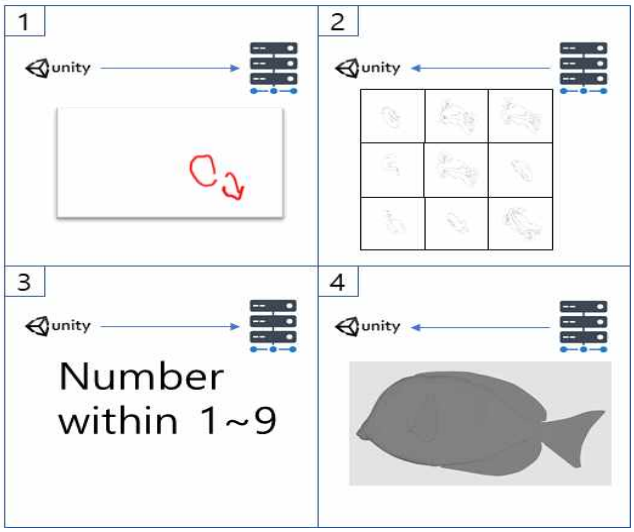


그림 24. 모델 검색 작동 과정

유니티 부분 스크립트, 서버 관련 소스 자체는 크게 문제가 없었지만 OpenSSE와 서버 관련 소스를 통합하는 과정에서 여러 작업을 해주었다. 첫째로는 char 배열을 통해 경로 문자열이 저장되어 있었는데 이를 파싱하는 과정에서 계속하여 NullPointerException이 발생하였다. 이 에러자체는 해결하였지만, 기반 언어가 C++이기도 하고 코드의 안정성이 떨어져서 String으로 다루도록 바꾸었다. 둘째로는 클라이언트에서 전달된 key를 특정 모델 파일에 매핑시키는 작업을 unordered_map을 통하여 해주었다. 셋째로는 스케치를 이미지파일로 만들 때 Texture2D의 내장함수인 EncodeToPNG를 하고나서 파일 생성하는 부분이 빠져있어서 WriteAllBytes를 통해 이진값을 PNG 파일로 생성해주었다. 마지막으로 obj를 전달받아 화면에 띄우는 과정에서 Sharing Violation 에러가 발생했는데 이는 모든 열려진 FileStream을 정상적으로 close 해주어 해결하였다.

이러한 자잘한 작업들 이외에도 해야했던 작업은 모델 파일의 포맷을 변경하는 작업이었다. OpenSSE는 모델을 off 포맷으로 반환하는데, 유니티 뿐만 아니라 대부분의 경우 obj 포맷을 사용하기 때문에 off to obj convert 작업을 수행할 필요가 있었다. 이 작업을 런타임에 수행하려면 오픈소스, 라이브러리, 직접 구현 등의 방법을 통해 코드를 추가해주어야 하는데 아쉽게도 쓸만한 라이브러리는 찾지 못하였다. 만약 직접 구현을 한다면 파일 포맷을 변경하여야 하는데, 각 파일 포맷을 이루는 구조를 명확히 알고 있다면 불가능한 일은 아니다. 예를 들면 off 파일 포맷은 정점들의 좌표(x,y,z)와 면들의 목록(면을 구성하는 정점, RGB 값 포함)으로 이루어져있고, obj 파일 포맷은 이보다 훨씬 복잡하지만 기본적으로는 obj 역시 정점과 면들의 집합으로 이루어져 있다. 우리가 다루는 모델에는 별도의 color depth가 없는데다가 애초에 off 파일은 obj 파일 포맷을 이루는 구성 요소들 중 다룰 수 있는 것이 거의 없으므로 사실상 공백의 위치, 횟수 등만 고려하여 각 문자들을 재배치해주면 될 것이다. 이런 구상은 해보았지만 우리의 주 목적은 아니므로 일단 Antiprism의 off2obj converter를 통해서 미리 변환을 해두었다.

200개의 파일에 대하여 변환을 하여야 하므로 다음과 같이 쉘스크립트의 반복문을 사용하였다.

```
$ for i in {001..200}
> do
> off2obj "$i".off > "$i".obj
> done
```

결과적으로 완성된 모델 검색 기능을 직접 수행해보았다. 아직 앱에서는 수행되지 못하므로 유니티를 이용하여 수행하였다. 먼저 가상머신에 올라온 우분투에서 서버를 켜두어 연결 대기 상태에 둔다.(그림 25) 유니티에서 play버튼을 누르고 카메라가 앵커가 있는 위치를 바라보게끔 한다.(그림 26) 그 후 펜을 가지고 실제 위에다 스케치를 그리면, Photon을 통해 실시간으로 유니티의 가상공간 내에 반영된다.(그림 27) 그 후에는 UI의 Search 버튼을 눌러주면 된다.

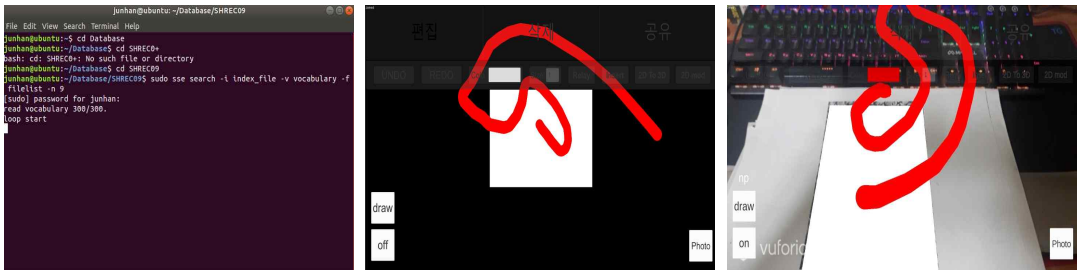


그림 25, 26, 27

서버에서는 전송된 이미지를 가지고 연산하여 유사한 이미지들을 전송한다.(그림 28) 유니티에서는 해당 이미지들을 사용자가 선택할 수 있게 UI 상에 확률 순으로 띄운다.(그림 29)

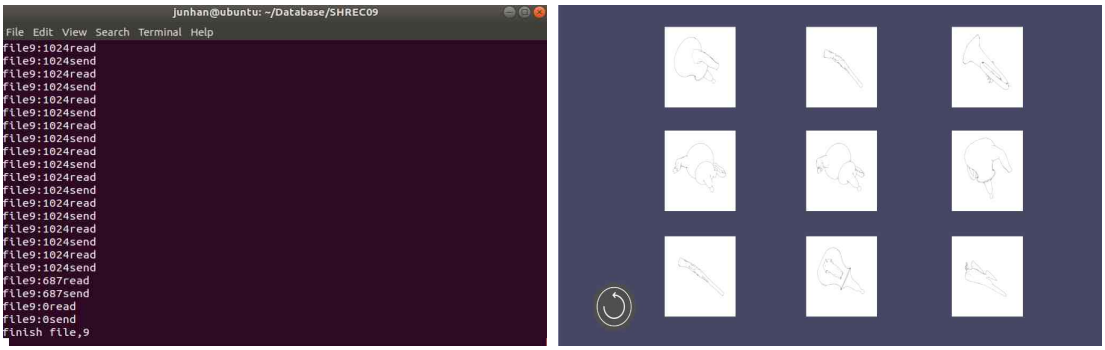


그림 28, 29

사용자가 이미지를 하나 선택하면 서버로부터 그 이미지에 해당하는 모델을 전송받는다.(그림 30) 결과적으로 전송받은 모델파일이 가상공간 상에 성공적으로 load 된 모습이다.(그림 31) 당연히 해당 모델도 객체이므로 움직이거나 회전하거나 등의 작업이 가능하다.

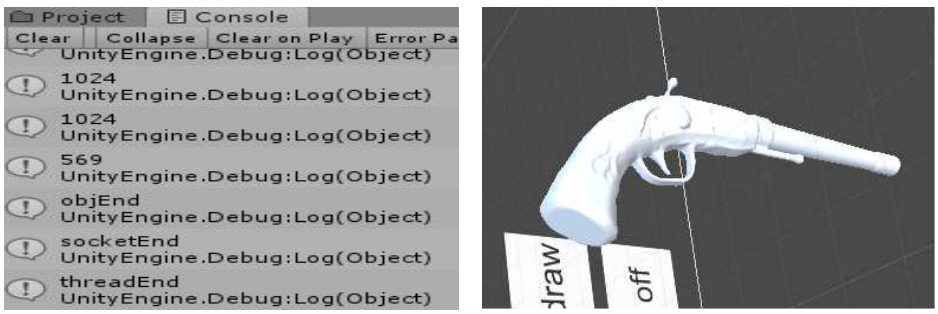


그림 30, 31

다른 문제점들은 차치하더라도 현재 가장 큰 문제점은 반환된 모델의 위치, 각도, 스케일을 잡기가 애매하고 기존에 있는 스케치를 지우기도 애매하다는 것이다. 직관적으로 생각해봤을 때는 스케치를 이미지로 변환할 때 관련 정보를 로컬에 저장해두었다가 모델이 반환되었을 때 그 정보를 그대로 반영하면 될 것이다. 그렇게 되면 모델은 기존 스케치의 위치, 각도, 스케일을 따를 것이다. 문제는 현재 방식이 객체 단위

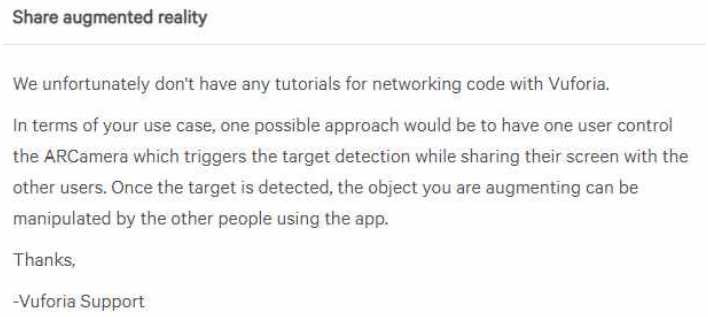
로 이미지 변환을 시키는 게 아니라, 현재 카메라가 보고 있는 모든 객체들을 2D로 투영시켜 이미지화 한다는 것이다. 즉 스케치 객체를 특정할 수 없으므로 정보를 반영하기도 애매해지는 것이다. 그나마 현재로서는 바라보고 있는 모든 객체의 평균 위치, 평균 각도, 평균 스케일에 배치하는 것으로 타협하였다. 물론 사용자가 위치와 각도를 조절할 수 있고 스케일 기능은 아직 만들진 않았지만 구현할 수 있으므로 편의성을 제외하면 크게 문제는 안 된다. 그리고 사실 오히려 이 방식이 더 사용자에게는 직관적일 수 있다. 왜냐하면 모델 검색을 수행할 때마다 사용자가 일일이 손이나 펜으로 객체를 특정 짓는 것보다는 그냥 사용자가 보고 있는 화면에 존재하는 스케치를 모델로 바꾸는 것이 훨씬 쉽고 와닿기 때문이다.

그 외의 아쉬운 점은 부족한 학습도와 2D Sketch Based 방식의 한계이다. 다만 학습도가 낮더라도 추가적으로 학습을 시키는 것은 전혀 문제가 되지 않는다. 얼굴인식 인공지능을 학습시킨다고 가정하면 가장 문제가 되는건 다양한 각도에서 찍힌 수많은 얼굴 사진을 필요로 한다는 것이다. 이는 2D의 정보를 가지고 3D를 만들어내야 하기 때문이다. 반면에 우리는 이미 모델파일을 가지고 있고 그 모델을 수많은 각도에서 찍고 살짝 변형시켜 학습을 시키기 때문에 꽤나 학습이 쉬운 편이다.

2D Sketch based 방식이 아니라 3D Sketch based 방식을 사용하면 보다 정확성이 높을 것이다. 쉽게 떠올릴 수 있는 방법은 우리가 펜을 가지고 계속해서 스케치를 하여 데이터셋을 모으는 방법이 있을 것이고 또는 요즘 대세인 점구름(Point Cloud)를 활용하는 방법도 있을 것이다.

▷ 멀티 AR 기능 (Shared AR)

Shared AR이라는 것은 사실 학계에서 사용되는 정식명칭이라기보다는 AR분야의 개발자들이 '증강현실의 공유' 라는 부분에 초점을 맞출 경우 일반적으로 사용되는 표현이다. 구글의 ARCore의 경우, Cloud Anchors라는 시스템을 통하여 개발자가 편리하게 증강현실을 공유할 수 있도록 지원해주는데, Vuforia는 안타깝게도 그런 시스템이 없기 때문에 포럼의 많은 개발자들이 어려움을 겪고 있다. 더군다나 공유에 대한 라이브러리, 레퍼런스, 튜토리얼 등 별도로 참고할 수 있는 것도 존재하지 않았다. 그럼에도 Shared AR은 구현되었을 때 응용 면에서 매우 큰 가치를 지니고 있고, 생각보다 쉬운 방법으로 접근할 수 있다고 생각하여 이 작업을 시도하게 되었다. 그림 32는 구상에 조금이나마 도움을 받은 게시글이다.


Share augmented reality

We unfortunately don't have any tutorials for networking code with Vuforia.

In terms of your use case, one possible approach would be to have one user control the ARCamera which triggers the target detection while sharing their screen with the other users. Once the target is detected, the object you are augmenting can be manipulated by the other people using the app.

Thanks,

-Vuforia Support

그림 32. Vuforia Forum의 게시글

우선 유니티 내에서 사용자끼리 네트워크 작업을 할 수 있게 해주는 플러그인 같은 것이 필요했다. 우리의 프로젝트에서는 스케치 파일(obj파일) 공유를 위한 서버가 구현될 것이지만, 아무래도 우리가 직접 만든 서버는 저레벨에서 돌아가는 플러그인보다는 성능면에서 뒤떨어질 것이라 생각했다. 처음에는 UNet이라는 플러그인을 사용할 예정이었으나, 이는 곧 없어질 기능이라 Deprecated로 지정되었기 때문에 사용할 수

없었다. UNet만큼 많이 쓰이는 플러그인으로는 PUN(Photon Unity Networking)이 있다. 이는 ARCore의 Cloud Anchors에서 사용되는 FireBase처럼 클라이언트 간 데이터의 빠른 공유를 도와주는 역할을 한다. Photon 홈페이지에서 제공하는 DB를 만들고 AppID를 할당받아 유니티에 적용한 후에 플러그인을 Import하기만 하면 된다. PUN은 최대 20명의 클라이언트까지 무료로 수용 가능하며, KR(Korea)서버가 별도로 존재하여 실시간 네트워킹에 적합하다.

먼저, PUN을 초기화할 NetworkManager 스크립트가 필요하다. 여기에는 클라이언트가 공유할 룸(Room)의 정보, 룸을 지정하는 방식, 룸 입장에 실패했을 때의 루틴, 최대 입장 가능 클라이언트 수 등 여러 콜백 메서드가 구현된다. 현재는 단지 테스트가 목적이기 때문에 최대한 단순하게 구성하고, 입장 가능 클라이언트 수는 2명으로 하였다.

PUN의 사용이 처음이었기 때문에 작동 방식을 직관적으로 익히기 위해 간단한 텍스트 공유 작업을 해보기로 하였다. 어느 하나의 클라이언트에서 draw버튼을 눌렀을 때, 해당 클라이언트가 보유한 Brush의 정보가 텍스트박스에 나타나게 하였다. 그림 33은 텍스트가 PC와 모바일 기기 사이에서 공유되는 모습이다. 물론 어떤 기기 사이에서든 인터넷만 연결되어있다면 문제없이 연결이 가능하다.

그림 33. PC와 스마트폰 상에서의 텍스트 정보 공유

텍스트가 공유된다는 것은 앵커나 스케치의 어떤 정보를 텍스트화 시킬 수만 있다면 AR환경도 공유할 수 있다는 뜻이 된다. 기존의 일반적인 접근 방식은 AR Camera를 공유하거나, 가상 객체 자체를 공유하는 방식을 사용한다. AR Camera를 공유하는 방식은 굉장히 다루기가 까다롭고, 가상 객체 전체를 공유하는 방식은 서로 공유할 정보의 양이 많은 것이 단점이다. 특히 이 방식은 마스터(Master)에서 가상 객체의 위치가 변할 때마다 그 객체의 모든 정보를 다른 슬레이브(Slave)에 갱신시키므로 사실상 사용하기가 곤란한 방식이다. 반면에 텍스트 접근 방식은 객체에 대한 정보가 적은 용량을 지닌 텍스트로만 공유되어 훨씬 효율이 좋다. 이 방식은 한번 그 객체가 그려지면 그 이후로는 마스터가 따로 갱신시키지 않으며 각 클라이언트에서 알아서 렌더링한다. 이는 타겟과 가상 객체 사이의 관계가 Low Level에서 부모-자식 간의 관계로 결합되는 Vuforia의 강점이다.

이러한 텍스트 공유 방식을 이용하여 결과적으로 구상한 방법은 다음과 같다.

① 어떤 클라이언트에서 draw버튼을 누르면 그 시점에 해당 클라이언트가 보유한 Brush의 position과 rotation을 저장한다.
② 해당 값을 인자로 하여 원격 프로시져를 호출한다.(RPC)
③ 이와 연결된 메서드를 통해 공유되는 인자를 가지고 같은 위치(타겟 또는 앵커에 대한 상대 위치)에다 점을 찍고 타겟 또는 앵커의 자식으로 한다.

그림 34에서는 사용자가 draw를 눌렀을 때 해당 브러쉬에 대한 string 정보를 인자로 하여 Send 메서드를 호출하는 모습이다.

```
Transform brushTransform = brush.transform;

// Shared AR 시작 부분.
string[] strPosition = { brushTransform.position.x.ToString(), brus
string[] strRotation = { brushTransform.rotation.x.ToString(), brus
scriptPhotonChat.Send(PhotonTargets.All, strPosition, strRotation);
```

그림 34. Draw 스크립트 부분

전달된 인자들을 다시 RPC를 통해 모든 클라이언트에 전달시키고, 결과적으로 그 인자들(점에 대한 정보)을 가지고 각 클라이언트의 기기에서 앵커를 부모로 하여 점을 등록한다.

```
public GameObject paint;
public GameObject anchor;


public void Send(PhotonTargets _target, string[] position, string[] rotation)
{
    photonView.RPC("SendMsg", _target, position, rotation);
}

// position과 rotation 인자는 모든 연결된 클라이언트에서 공유된다.
[PunRPC]
void SendMsg(string[] position, string[] rotation, PhotonMessageInfo _info)
{
    SetPointTransform(position, rotation);
}

void SetPointTransform(string[] position, string[] rotation)
{
    Vector3 pointPosition = new Vector3(float.Parse(position[0]), float.Parse(position[1]),
    Quaternion pointRotation = new Quaternion(float.Parse(rotation[0]), float.Parse(rotation
    Instantiate(paint, pointPosition, pointRotation).transform.SetParent(anchor.transform);
```

그림 35. 공유 스크립트 부분

제대로 작동되는지 확인해보기 위해 두 스마트폰(갤럭시 S3와 갤럭시 J7) 사이에서 작동시켜 보았다. Vuforia는 범용성에 강한 플랫폼이기 때문에 꽤나 구형인 갤럭시 S3에서도(심지어 안드로이드나 애플 제품이 아니더라도) 비록 열악하지만 AR 환경이 구성될 수 있다. 그림 36은 각 기기 간에 실시간으로 스케치가 공유되는 모습이다. 실행해본 결과 두 기기 사이에서의 지연시간은 측정하기도 힘들 정도로 빨랐다. 지금은 같은 타겟 상에서 다른 스마트폰으로 본 것이지만, 이 방식은 사실상 인터넷만 연결되어있다면 공간적 제약이 존재하지 않는다. 그러므로 서울에 사는 A와 뉴욕에 사는 B가 프린터로 출력을 했든 모니터에 보이는 것이든 서로 같은 타겟(Target)을 보유하기만 하다면 스케치의 실시간 공유가 가능하다. 또한 이 방식에 Vuforia의 대형 저장 공간인 Cloud Reco가 결합되면 시간적 제약 역시 사라진다. 왜냐하면 런타임에 스케치한 가상 객체를 영구적으로 타겟에 종속시킬 수 있기 때문이다. 물론 런타임 상 등록한 가상 객체를 Cloud Reco에 저장할 수 없다 해도, 완성된 가상 객체(점이나 선)의 집합을 하나의 인스턴스화 시켜 그 인스턴스 자체를 동적으로 타겟의 하위 컴포넌트로 추가시키는 등의 테크닉을 사용한다면 크게 어렵지 않을 것이다.

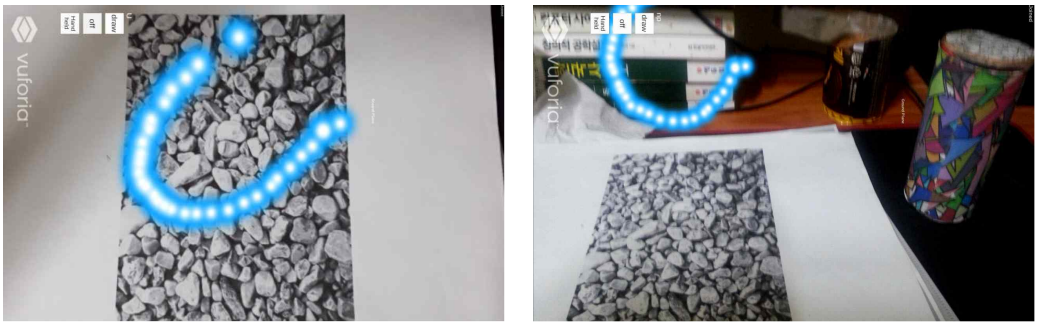


그림 36. 갤럭시 J7(좌), 갤럭시 S3(우) 사이의 실시간 스케치 공유

결과적으로 구현된 Shared AR은 어느 한 기기에서 스케치를 그리면 연결된 모든 기기에서도 실시간으로 그려지는 방식이다. 이 방식을 모델 타겟(Model Target)과 결합하면, 기업이 고객의 A/S 요청을 받을 때 보다 효과적으로 대응할 수 있을 것이다. 만일 담당자와 고객이 공유하고 있는 규격화된 제품이 있으면, 그 제품 위에서 서로 스케치를 하며 보다 효과적으로 소통할 수 있을 것이고, 특히 연령대가 높은 분들에게 어렵사리 말로 전달하는 것보다는 훨씬 효과가 좋을 것이다. 또한 미래에 상용화 될 AR안경 같은 디바이스와 결합한다면, 강의실에서 ppt에다 간단한 스케치를 하거나 칠판에 필기를 하는 문화에서 보다 발전하여, 3차원 상에서 필기를 하고(물론 이 때는 Target 방식이 아닌 허공에 스케치가 가능한 Anchor 방식을 사용하게 될 것이다) 학생들은 이를 3차원으로 보고 또 자기가 그 필기에 개입하여 훨씬 효과적인 학습과 더불어 쌍방향 학습 문화가 자리 잡을 수 있을 것이다. 그 외에도 채팅, 게임, 기타 수많은 산업 등 응용할 수 있는 범위는 무궁무진하다.

▷ 최종 결과물

최종 결과물(앱)의 대부분의 기능은 다음의 링크에서 확인할 수 있다.

https://www.youtube.com/watch?v=emyAoJ-IdPI&feature=youtu.be

4. 조원별 역할 분담

▷ 김준한

- 실린더 펜 구상 및 제작
- Vuforia를 통한 증강현실 환경 구현
- 모델 검색 및 멀티 AR 기능 구현

▷ 한제완

- 스케치 앱과 앱에 활용되는 3차원 스케치 관련 기능 구현
- 스케치 보정 기술 개선
- 멀티 AR에 사용되는 서버 구현

6. 목표 달성도

세부 목표	달성 내용	달성도(%)
Vuforia 환경 구현	기본 스케치 환경, UDT, Mid Air 구현 일정수준의 트래킹 제한점 남아있음	80%
펜 제작	실린더 타겟을 활용한 펜 제작 원활한 블루투스 통신이 가능하지만, 펜 자체의 트래킹 성능이 다소 떨어짐	70%
스케치 앱 제작	많은 수의 스케치 기능을 구현 각 기능의 완성도가 조금 아쉽고 사용자 관점에서의 UI 개선이 필요함	70%
보정 기술 개선	3차원에서의 스케치 작업에 대한 효과적인 보정 기술을 고안하였으나 실험을 통해 나온 수식의 이론적 뒷받침이 미흡하며, 사용자 경험을 통해 충분히 검증하지 못하였음	70%
모델 검색 기능	2차원 모델 검색 기능 자체는 문제가 없으나, 서버의 성능이 충분히 개선되지 못하였고, 모델의 데이터 셋 상태나 학습률이 아쉬움.	60%
멀티 AR 기능	다양한 기종에서의 멀티 AR이 원활하게 동작함.	100%
합 계		75%

7. 활용 계획 및 기대효과

▷ 활용 계획

– 원격 고객 지원 시스템

상담자와 고객은 서로 원격에서 실시간으로 증강현실 환경을 공유한다. 공유되는 환경에서 3차원 스케치를 같이 그려나가면서 직관적인 소통이 가능하다. 이러한 방식은 전화상으로 소통하였을 때 생기는 언어의 한계를 극복하는데 큰 도움을 줄 수 있을 것이다. 물론 화상통화나 팀뷰어 같은 기존의 방식도 있지만 현실의 환경을 배경으로 하여, 3차원 상에서 (모니터 상에서 구현된 3차원이 아닌) 소통을 한다면 훨씬 원활한 소통이 가능할 것이다.

– 산업 현장

산업 현장에 존재하는 다양한 부품에는 가상의 라벨링이 되어있을 것이다. 부품이 장착된

상태이던 장착되지 않은 상태이던 상관없이 트래킹이 가능하다. 어떤 부품을 어떻게 장착하고 활용해야 하는지에 대해서는 상급자나 전임자의 3차원 스케치를 통해서 쉽게 익힐 수 있다. 조립과 관련된 작업을 할 때를 떠올려 보자. 추상적으로 표현된 언어나 상징적인 2차원 그림에 의존하지 않아도 된다. 또는 고가의 컴퓨터에서 어려운 기능으로 구성된 툴을 이용할 필요가 없다. 그저 자기가 보유한 스마트폰으로 현재 눈앞에 있는 장치에다가 가상으로 그리면 된다. 지시를 전달받는 사람도 단지 자신의 스마트폰으로 그것을 보고, 그대로 따라서 조립하면 될 것이다.

– 전문적인 모델링

아직은 직접 활용할 수 없지만, 향후에 그리드 및 가이드 라인을 추가하여 전문 사용자가 모델링 작업을 할 때 보다 직관적인 방식으로 활용할 수 있도록 개선할 예정이다.

– 일반 사용자 앱

일반 사용자는 자신이 그린 3차원 스케치를 현실의 배경과 합성하거나, 이를 저장하거나 공유할 수 있으며, 실시간으로 친구와 3차원 스케치를 하며 여가 활동을 보낼 수 있다.

▷ 기대 효과

현재는 스마트폰 상에서 작업을 해야하기 때문에 상업성이 어느정도 애매한 편이지만, 향후 AR 안경이 보급화되었을 때는 이러한 툴의 접근성이 보다 높아질 것이고, 거기에 추가적으로 Hand Tracking을 통해서 펜의 역할을 대체하고, 전문가 작업을 위해 도움을 줄 수 있는 여러 기능을 추가하고, 일반 사용자를 위해서 기존 다른 앱들과 연계를 한다면 적지 않은 기대 효과를 얻을 수 있으리라고 생각한다.

이번에 제작한 실린더 타겟의 펜은 AR 환경에서의 스케치 작업을 위한 컨트롤러로서 새로운 관점을 제시할 수 있으며, 그 외에도 다양한 증강현실 환경 구현, 앱에서의 여러 3차원 스케치 관련 기능, 모델 검색 기능이나 멀티 AR 기능과의 연계는 앞으로 증강현실 상의 3차원 스케치 기술이 나아갈 새로운 관점을 부분적으로 제시하였다고 볼 수 있을 것이다.

8. 문제점 및 건의사항

– IMU 만을 활용한 트래킹을 시도하려고 했으나, 비용적 기술적 문제 때문에 구현하지 못하였다.

– 펜의 제스처를 인식하여 특정 기능들을 수행하려고 했으나, 원하는 신경망을 구현하지 못하여 제작하지 못하였다.