

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Szokoly-Angyal Armand

BuildingSketcher

Hideg Attila

BUDAPEST, 2025

# Bevezetés

A BuildingSketcher projekt egy olyan, Androidra fejlesztett alkalmazás, melynek célja épületek formájának vizualizálása pusztán egy papír és tollvonások felhasználásával. Az alkalmazás kiterjesztett valóságot használ a cél elérésére, és használatához csupán egy Android telefon szükséges, mely kompatibilis a Google AR Core-ral, valamint egy legalább A4 méretű lap és toll.

Az alkalmazás detekálja a papír jelenlétét és helyzetét a térben, és az arra rajzolt vonalakat extrapolálja falakká, mely alkalmas lehet egy egyszerűbb épület vázlatának háromdimenziós megjelenítésére.

Az alkalmazás jelenlegi formájában még sajnos nem alkalmas teljes alaprajz megjelenítésére, azonban további finomítással (melyet a jövőbeli tervekben részletezek) megvalósítható az is, hogy komplexebb alaprajzokat interaktívan módosítsunk.

# Technológiai háttér

Az alkalmazás Unity 6 (6000.0.43f1) editor használatával, az AR Foundation (6.2.0-pre.4 - May 06, 2025) és a Google AR Core XR plugin (6.2.0-pre.3 · May 01, 2025) használatával készült. Azért a legfrisebb pre verziókat alkalmazom, mert a korábbi verziókban egy olyan kritikus kompatibilitási probléma merült fel, amely miatt nem lehetett az XrCpuImage (lásd később) komponenst az AR Camera-ról lekérdezni, ezért kiemelem, hogy erre külön figyeljen oda az, aki ezt az eszközt fejleszteni szeretné. Ezen túl a projekt alapbeállításai a hivatalos AR Foundation dokumentációja[1] szerint leírtakat követik.

A lap- valamint a vonaldetektálás a nyílt forráskódú OpenCV C++ könyvtár segítségével valósult meg. Létezik Unityben egy (2025-ben legalábbis) fizetős bővítmény is, ez a projekt azonban saját c++ megvalósításra fókuszál a projekt használati esetének egyedisége okán. Ezt a c++ kódot majd CMake és Gradle segítségével .so fájlba kell csomagolni, hogy a Unity tartalmazhassa a natív kódot a buildben, és C#-ban meghívhassuk (a részletes **ajánlott** developer flow-ról később).

A „mérnöki kihívás” ebben a feladatban 2 fő problémára osztható fel:

* A megfelelő OpenCV logika megvalósítása,
* Majd annak megfelelő hozzákapcsolása a Unity csővezetékéhez, mely alatt ezúttal a Unity AR Foundation beépített AR grafikai render csővezetékét értem.

Ez utóbbi probléma nem triviális, ugyanis egy olyan **koordináta konverziós csővezetéket** kell megvalósítanunk ehhez a Unity AR Foundation beépített koordináta-konverziós csővezetékével párhuzamosan, mely tökéletesen „másolja” azt. Ennek pontos folyamatáról később írok. A koordináta konverzió alatt ezúttal pontosan a 2D kamera képének az Android készülék képernyőjére, majd aszerint a Unity világba való transzformálását értem. A későbbiekben ennek a folyamatnak a fenti eszközökbe beépített részét az egyszerűség kedvéért „AR render csővezetéknek”, vagy csak „Unity csővezetéknek” fogom hívni. Azt fontos előrevetítenem, hogy ez a feladat jelenleg **nincsen teljesen jól implementálva** ebben a programban, ami torzított AR megjelenítést eredményez, de az app még így is használható.

# Implementáció

## Az OpenCV bővítmény implementációja

A cél egy olyan egyszerűen használható API létrehozása volt, mely képes egy kapott képről kinyerni egy papír sarkainak koordinátáit a kép koordinátarendszerében (bal felső az origin), valamint az összes fellelt vonás két végének koordinátáit.

### LineDetector.cpp

Ez a modul a képen található egyenes fekete vonalak detektálását és összevonását végzi, tipikusan papírlapra rajzolt vonalakból kiindulva. Az OpenCV segítségével, többlépéses képfeldolgozási folyamatot alkalmaz, majd az egymással csaknem párhuzamos, egymáshoz közeli vonalszakaszokat összevonja. Az így kapott vonalakat később AR környezetben lehet hasznosítani, például falak vagy más objektumok vizualizációjára.

#### Főbb függvények és feladatuk

static std::vector<cv::Vec4i> MergeColinearClusters(

const std::vector<cv::Vec4i>& segs,

double angleTolDeg = 7.0,

double distTolPx = 15.0

)

**Feladata:**

* Az OpenCV HoughLinesP által detektált vonalszakaszokat csoportosítja és egyesíti.
* Azokat a szakaszokat vonja össze, amelyek majdnem párhuzamosak (**angleTolDeg**), és amelyek egymáshoz térben is közel esnek (**distTolPx**).
* A csoport minden tagjából egy „spanning segment”-et készít, amely lefedi az összes szakasz által kijelölt tartományt.
* Az eredmény: rövidebb, szaggatott vonalak helyett hosszabb, egybefüggő szakaszokat kapunk, a közeli szakaszok egyetlen egyenest alkotnak.

extern "C" \_\_declspec(dllexport)

int FindBlackLines(

unsigned char\* imageData,

int width, int height,

float\* outLines, int maxLines

)

**Feladata:**

* Ez a függvény a fő belépési pont Unity vagy más külső rendszer számára.
* Bemenetként egy RGBA vagy BGR kép pixeladatait várja, OpenCV-mátrixként alakítja.
* Lépések:
  1. **Szürkeárnyalatosítás**: RGBA/BGR → Grayscale.
  2. **Elmosás**: Gauss-elmosás a zaj csökkentésére.
  3. **Adaptív thresholding**: Az adaptív thresholding kiemeli a sötét vonalakat.
  4. **Canny élkeresés**: Detektálja a képen az éleket.
  5. **HoughLinesP**: Megkeresi az egyenes vonalszakaszokat.
  6. **MergeColinearClusters**: Összevonja az egymással majdnem párhuzamos, közeli vonalakat.
  7. **Exportálás**: Az eredményül kapott vonalszakaszokat a outLines tömbbe tölti, négyesével (x1, y1, x2, y2).

**Megjegyzés**

Ez a függvény mobileszközök nyers kamera inputjára lett optimalizálva, **irodai fényviszonyok között, tiszta fehér papíron, sötét tollvonásokra.** Ez azt jelenti, hogy a minőség drasztikusan romlik, ha fehér papír helyett sötétebbet, vagy főleg vonalazott, vagy négyzetrácsos papírt használunk. Ez a függvény fine-tuning nélkül nehezen újrahasználható más használati esetekre, kifejezetten ennek az alkalmazásnak lett elkészítve.

### PaperPlugin1.cpp

Ebben egyetlen függvény található:

bool FindPaperCorners(

unsigned char\* imageData,

int width, int height,

float\* outCorners

)

**Feladata:**

* Egy RGBA képadatot vizsgálva megkeresi a legnagyobb, konvex, négyszögű kontúrt (egy papírlapot).
* A talált négyszög négy sarkának (pixelbeli) koordinátáit adja vissza, az outCorners tömbön keresztül.
* Ha talál ilyen négyszöget, akkor **true**-val tér vissza, ellenkező esetben **false**-szal.

**Megjegyzés:**

Ez a függvény is, akárcsak az előző, mobileszközök nyers kamera képére lett optimalizálva, irodai fényviszonyokra. Ennek a függvénynek az egészséges működéséhez nélkülözhetetlen a megfelelő kontraszt a papír és az asztal között, tehát ajánlott sötét háttér, terítő használata.

### A Tesztprojekt a natív kód verifikálására : PaperPluginTester.cpp

A NativePlugins/PaperPlugin mappában két projektet találhatunk egy solutionben, a PaperPlugin1 a fenti implementációkat tartalmazza. A PaperPluginTestert az alábbiakban tagoljuk.

Ez a fájl egy **önálló, konzolos Visual Studio tesztprogram** a natív papír- és vonaldetektor pluginek kipróbálására és vizuális ellenőrzésére. Segítségével fejlesztői környezetben, Unity-től függetlenül lehet tesztelni, hogy a C++-os OpenCV-alapú algoritmusok helyesen működnek-e képeken.  
A program betölt egy bemeneti képet, meghívja a plugin függvényeket, majd az eredményt vizuálisan ábrázolja (pontokkal, vonalakkal) és fájlba is menti.

A repository tartalmaz pár gépileg előállított, illetve valós képet tesztelésre mind vonal, mind papírdetektáláshoz.

#### EnsureBGR

cv::Mat EnsureBGR(const cv::Mat& img)

**Feladata:**

Segédfüggvény, amely bármilyen csatornaszámú képből (pl. RGBA, grayscale) BGR képet készít megjelenítéshez vagy mentéshez.

#### PlaceDotsOnImage

void PlaceDotsOnImage(cv::Mat& img, const std::vector<cv::Point2f>& points, const cv::Scalar& color, int radius)

**Feladata:**  
Az átadott képen kitöltött köröket (pontokat) rajzol a megadott koordinátákra (pl. sarokpontok, végpontok vizualizálása).

#### DetectPaperCorners

std::vector<cv::Point2f> DetectPaperCorners(cv::Mat& img)

**Feladata:**  
Wrapper függvény: Meghívja a natív DLL-ben található FindPaperCorners függvényt, majd a kimenetet cv::Point2f vektorrá konvertálja. Visszaadja a detektált négyszög sarkait.

#### DetectBlackLines

std::vector<std::pair<cv::Point2f, cv::Point2f>> DetectBlackLines(cv::Mat& img, int maxLines)

**Feladata:**  
Wrapper: Meghívja a DLL-ből a FindBlackLines függvényt. Az eredményt (pont, pont) párokba (szakaszok) rendezi.

#### TestAndVisualizePaperDetection

void TestAndVisualizePaperDetection(const std::string& inputPath, const std::string& outputPath)

**Feladata:**

* Betölt egy képet (alapból RGBA formátumban).
* Meghívja a papírsarok-detektáló függvényt.
* A detektált sarokpontokra piros pontokat rajzol.
* Ment egy BGR (JPEG/PNG) képet a megadott útvonalra.
* Kirajzolja az eredményt ablakban, vizuális ellenőrzéshez.

#### TestAndVisualizeLineDetection

void TestAndVisualizeLineDetection(const std::string& inputPath, const std::string& outputPath)

**Feladata:**

* Betölt egy tesztképet.
* Meghívja a vonaldetektáló plugint.
* A megtalált vonalakat zöld színnel kirajzolja.
* A végpontokra kék pontokat helyez el.
* Az eredményt elmenti, és opcionálisan megjeleníti.

#### main

int main()

**Feladata:**  
A fő belépési pont. Teszteseteket indít.

* A bemeneti és kimeneti fájlokat itt lehet módosítani (TestFiles/, stb.).
* Csak konzolból futtatható; a vizuális ablakok automatikusan bezárhatók.

#### A tesztprogram használata

A projekt Visual Studio-ból fordítható és futtatható, de ügyeljünk arra, hogy a DLL-ek a megfelelő (pl. Debug vagy Release) mappában legyenek a dinamikus betöltéshez. A tesztképeket célszerű a TestFiles/ mappába tenni, az elkészült eredményképek pedig gyorsan, vizuálisan ellenőrizhetők. A wrapper függvények automatikusan gondoskodnak arról, hogy a bemenetek csatornaszáma megfelelő legyen.

## A Unity program implementációja

A programot megvalósító szkriptek az Assets/Scripts mappában találhatók.

### PaperPlugin.cs

Ez a statikus osztály Unity alatt biztosít managed C# interfészt a natív **PaperPlugin** könyvtárhoz, amely OpenCV-alapú papírsarok- és vonaldetektálást végez RGBA képeken. Android platformon P/Invoke-on keresztül hívja a valódi natív függvényeket, míg az Editorban stub (üres) implementációkat használ a fejlesztés zavartalansága érdekében.

public static bool FindPaperCorners(byte[] rgbaImage, int width, int height, out Vector2[] corners)

**Feladat:**  
Egy RGBA formátumú képen detektálja egy papírlap négy sarkát. Hibás bemenet esetén kivételt dob, különben feltölti a sarkok koordinátáit egy négy elemű Vector2 tömbbe. Siker esetén true-t ad vissza.

public static int FindBlackLines(byte[] rgbaImage, int width, int height, out Vector2[][] lines, int maxLines = 32)

**Feladat:**  
Legfeljebb maxLines fekete vonalat keres RGBA képen, a találatokat szegmensekként (két végpont koordinátájával) egy tömbben adja vissza. Hibás bemenetnél kivételt dob. Visszaadja a talált vonalak számát.

### PaperDetector.cs

public class PaperDetector : MonoBehaviour

Ez a Unity MonoBehaviour osztály valós időben detektálja egy papírlap sarkait az AR kameraképen, a natív PaperPlugin C++ függvényeit használva, majd átalakítja ezeket Unity világ- vagy viewport-koordinátákba, és előkészíti a vizualizációhoz vagy AR objektumgeneráláshoz.

Alapvetően debugoló célokból a segítségével megjeleníthető egy XrCpuImage kép a kijelző jobb felső sarkában, hogy azon tesztelhessük, a papír észlelése megfelelőképpen történik-e a csővezeték lefutása előtt.

Függvényei:

void OnFrame(ARCameraFrameEventArgs args)

Minden új kamera frame-nél beolvassa az aktuális képet, elvégzi a sarokdetektálást, elmenti a display mátrixot, majd a detektált papírsarkokat átalakítja viewport-koordinátákra, és továbbadja a vizualizációs komponenseknek.

private bool TryDetectPaperCorners()

Kinyeri a kamera RGBA pixeladatait és meghívja a PaperPlugin natív sarokdetektorát. Siker esetén eltárolja a sarkokat; ha nincs találat, kikapcsolja a vizualizációs vonalakat.

private Vector2[] ConvertImageCornersToViewport()

A detektált képi sarkokat átkonvertálja Unity viewport (képernyőfüggetlen) koordinátákra a display mátrix felhasználásával.

private void FetchDisplayMatrix(ARCameraFrameEventArgs args)

Kinyeri és eltárolja az aktuális AR kamera display mátrixát a további koordináta-átalakításhoz.

private void ExecuteDebug(Vector2[] viewportCorners)

Ha a debug mód aktív, a pipelineDebugger segítségével kiírja és naplózza a sarokkoordinátákat, valamint rájelöl a képre, ezzel segítve a fejlesztői hibakeresést.

private void UpdateTexture(XRCpuImage img)

Az AR kamera által szolgáltatott képből RGBA Texture2D-t hoz létre.

private void MarkCpuCornersOnTexture(Color32 dotColor, int dotSize = 7)

A detektált papírsarkokat vizuálisan, színes pontokkal bejelöli a textúrán, fejlesztői ellenőrzés céljából.

### PaperEdgeLines.cs

public class PaperEdgeLines : MonoBehaviour

Ez az osztály a papírlap sarkainak detektálása után a sarkokat összekötő éleket jeleníti meg AR környezetben LineRenderer-ek segítségével. A detektált sarokpontokat Unity világkoordinátákra konvertálja, kirajzolja a széleket, és opcionálisan debug módban fehér pontokat is megjelenít a képernyőn.

void Awake()

Inicializálja a LineRenderer komponenseket az élek megjelenítéséhez, beállítja a vonalak anyagát, és előkészíti a debughoz szükséges fehér pont textúrát.

public void InitLines()

Létrehozza és inicializálja a négy LineRenderer-t, amelyek a papírlap éleit ábrázolják Unity-ben.

public void DisableLines()

Letiltja (elrejti) az összes élvonalat, például amikor nincs érvényes detektálás.

public void PlaceLinesFromViewport(Vector2[] vpCorners)

A detektált sarkok viewport-koordinátáit Unity világpozíciókká alakítja, majd a sarkok között vonalakat húz a LineRenderer-eken keresztül.

private void DrawSegment(int i, int j, Vector3[] worldPos)

Két világpozíció között húz meg egy élvonalat.

### PipelineDebugger.cs

public class PipelineDebugger : MonoBehaviour

Ez az osztály a debugging folyamatához nyújt segítséget pár printelő függvény biztosításával.

public void SetupPrinted()

Egyszeri naplózás esetén beállítja a printed flag-et, így a többi debug függvény csak egyszer hajtódik végre.

public void printConverterCornersDebug(Matrix4x4 D, Vector2 xrCpuSize, Func<Matrix4x4, Vector2, Vector2, Vector2> converter)

Kiszámítja és kiírja a kamera kép négy sarkának viewport-beli pozícióit, majd a keletkezett téglalap szélességét és magasságát is naplózza. A megfelelő viewport konverziók ellenőrzésére alkalmas például.

public void logTransormedCorners(Matrix4x4 D, Vector2 xrCpuSize, Func<Matrix4x4, Vector2, Vector2, Vector2> converter)

A kamera kép sarkait végigiterálva naplózza, hogy azok CPU-beli koordinátából hogyan kerülnek átalakításra viewport-koordinátákká.

public void printDisplayMatrix(Matrix4x4 D)

Kiírja a display mátrixot mind transzponált, mind normál alakban a debug logba, vizsgálati célból.

public void printPaperCornerViewportCoords(Vector2[] viewportCorners)

A detektált papírsarkok viewport-koordinátáit naplózza, így ellenőrizhető, hogy minden sarok 0–1 tartományban van-e.

### WallGenerator.cs

public class WallGenerator : MonoBehaviour

Ez az osztály gondoskodik arról, hogy a papíron detektált vonalakat AR világban falakként (prefab) jelenítse meg. A detektált vonalakat csak akkor vizualizálja, ha azok mindkét végpontja a papírlap négyszögén belül van, majd a vonalakat AR világpozíciókra konvertálva helyezi el a jelenetben.

public void VisualizeLines(Texture2D tex, Matrix4x4 displayMatrix, ARRaycastManager raycastManager, Vector2[] paperQuad = null)

A papír detektálása után meghívható metódus, amely a képben található fekete vonalakat keresi, szűri, majd minden, a papírlapon belül eső vonalból világbeli falat generál a megfelelő pozícióban.

private List<(Vector2, Vector2)> DetectLines(Texture2D tex)

A PaperPlugin segítségével detektálja a fekete vonalakat a megadott textúrán, és visszaadja a vonalszakaszok végpontjait.

private void CleanupOldLines()

Törli a korábban generált falprefabokat a szülőobjektumból, hogy a vizualizáció mindig friss legyen.

private void InstantiateLine(Vector3 p1, Vector3 p2)

A két megadott világbeli pont között létrehoz egy új fal-prefabot, helyesen beállítva annak pozícióját, irányát és hosszát.

private static Vector3 ViewportToARWorld(Vector2 viewport, ARRaycastManager raycastManager)

Egy viewport-koordinátából (0–1 tartomány) AR világpozíciót számol raycast segítségével, síktalálat esetén visszaadja a sík pozícióját, különben Vector3.zero-t ad vissza.

### Scripts/Util/Converter.cs

public static class Converter

Ez a segédosztály a papírdetektálási pipeline koordináta-konverzióit valósítja meg. Segítségével a natív képpontok (CPU pixelkoordináták) Unity viewport-koordinátákká, majd az AR világpozíciókká alakíthatók.

public static Vector2 FromRawCpuToViewport(Matrix4x4 displayMatrix, Vector2 cpuPx, Vector2 XRCpuSize)

Egy adott képpont (CPU pixelkoordináta) pozícióját átkonvertálja 0–1 tartományú Unity viewport-koordinátává a display mátrix felhasználásával, beleértve a perspektivikus osztást is.

public static Vector3 ViewportToWorld(Vector2 viewport, ARRaycastManager raycastManager, List<ARRaycastHit> hits, out bool usedFallback)

Egy viewport-koordinátából AR világpozíciót számol raycast segítségével, síktalálat esetén visszaadja a találati pontot; ha nincs síktalálat, egy előre meghatározott távolságban visszaad egy “fallback” pozíciót, és ezt egy flag-gel jelzi.

### Scripts/Util/QuadHelper.cs

public static class QuadHelper

Ez a segédosztály a pipeline-ban használt konvex négyszögekhez kínál matematikai műveleteket.

public static bool PointInQuad(Vector2 p, Vector2[] Q)

Eldönti, hogy egy adott pont egy konvex négyszögön (négy csúcspontú sokszögön) belül helyezkedik-e el. A négyszög pontjai tetszőleges sorrendben (órairányú vagy ellentétes) lehetnek.

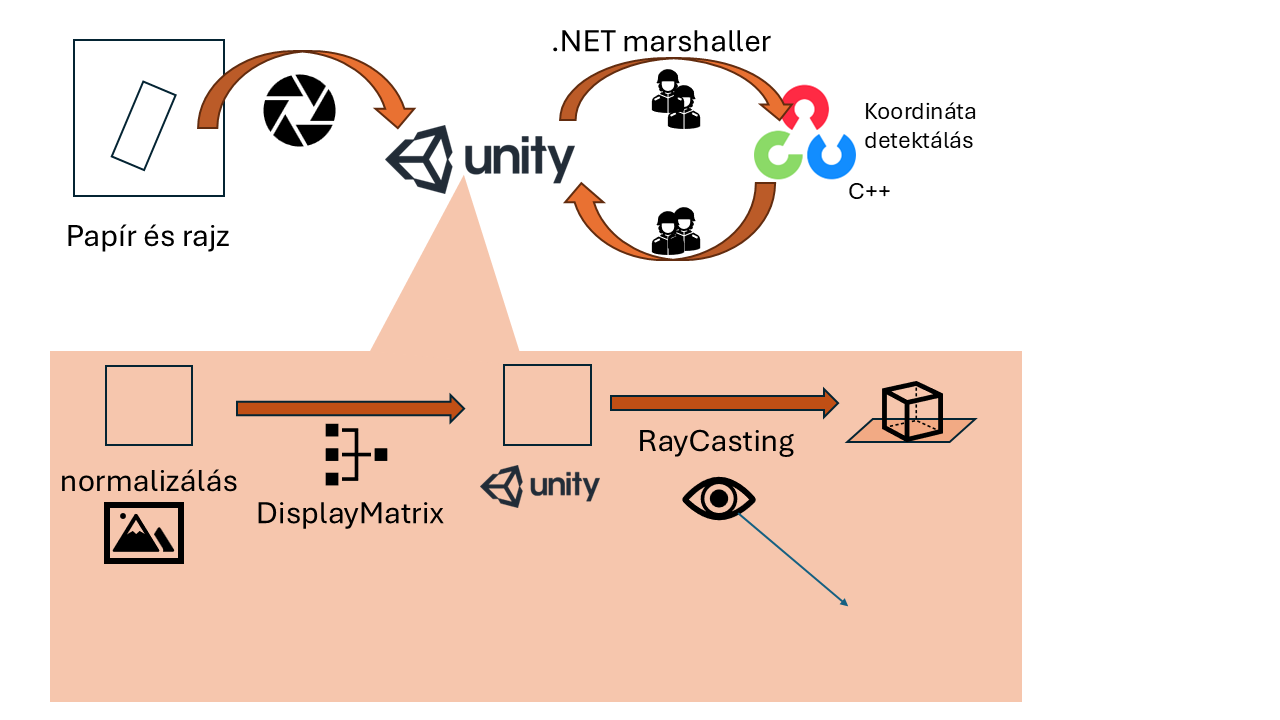
public static Vector2[] InsetQuad(Vector2[] Q, float insetPx)

Egy konvex négyszöget minden oldalon befelé "megszűkít", azaz minden sarokhoz olyan belső pontot számol, amely adott pixeltávolságra van az eredeti oldalszélektől. Az eredmény egy ugyanúgy rendezett négyszög, amely kisebb, de arányos az eredetivel.

# Működés madártávlatból

## Az alkalmazás működése

Az alábbiakban áttekintem a program működését, azt, hogy az adat milyen stádiumokon megy át a feldolgozás során, hogy végül eljussunk a megjelenítésig. Ez a szakasz a konverzió könnyebb megértésére szolgál (kutatásaim során ilyen magyarázatokból nem találtam elegendőt az interneten, így ezt hiánypótolni kívánom), de teljesen visszafejthető ez a folyamat az implementációs részletekből is.



ábra 1 Az alkalmazás működése

Mint ahogyan az az ábrán (ábra1) is látható, az adatfeldolgozás a kamera szenzorról beérkezett nyers kép (YUV\_420\_888 formátumban) „fogadásával” kezdődik (ennek pontos mikéntjéről, hogy ez pontosan hogyan történik, az implementáció fejezetben olvashatunk). A kapott kameraképet byte-okra bontva átadjuk a natív OpenCV pluginünknek, mely visszaadja a papír koordinátáit, valamint a kimutatott tollvonások egyeneseit egy tömbben. Ezeket a koordinátákat a kép koordináta rendszerében kaptuk meg, így ezeket normalizálnunk kell, hogy alkalmazhassuk a szükséges transzformációkat, hogy megkapjuk, ezek a pontok hol lesznek a készülék képernyőjén.

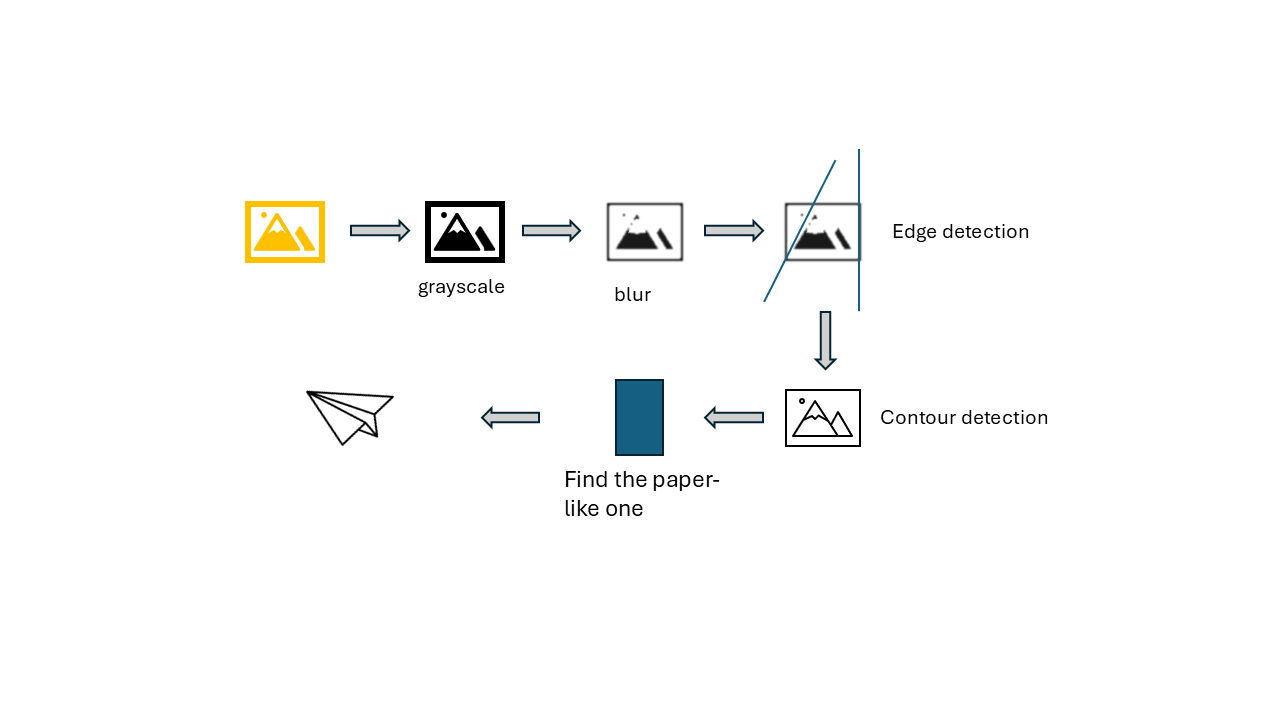
Itt fontos előrevetítenem a technikai kihívást ebben a feladatban. A dolgunk az, hogy „lemásoljuk” azokat a transzformációkat, amiket az ARCameraBackground mintázatát előállító shader tesz az XRCpuImage-el (pontosabban fogalmazva az abból nyert textúrával, de az egyszerűség kedvéért ezt értem alatta), hogy a tartalmunkat hitelesen jeleníthessük meg a kamerakép feletti overlayben.

Ahhoz, hogy a kamerakép úgy nézhessen ki, ahogy a Unity AR Android alkalmazásunkban háttérként látjuk, számos transzformációt kell végrehajtatnunk, melynek megértése a fent említett shader tanulmányozásával érhető el. A DisplayMatrix a dokumentáció alapján [2] tartalmazza ezeket a transzformációkat.

Mint ahogy azt a kódban és az implementációs részletekben is láthattuk, az alkalmazásunk pontosan erre a mátrixra épít a koordináták előállításához, azonban egyben ez is okozza a Unity worldspace-ben a megjelenített tartalmunk torzított megjelenését (erről a problémák részben még írok).

Miután az XRCpu kamera koordinátarendszeréből átkerültünk a Unity kamerájának view koordinátarendszerébe, használhatjuk a RayCastingot [3] hogy megkapjunk egy olyan Unity világbeli koordinátát, mely a kamerából kilőtt sugár legközelebbi metszéspontja egy detektált AR plane-nel, melyből már elő tudjuk állítani a 3D koordinátákat a megjelenítéshez.

## A képfeldolgozás OpenCV-vel



ábra 2 A lapdetektálás

Ahhoz, hogy a képen megfelelőképpen kimutathassuk az OpenCV eszközökkel a kontúrokat, először fekete-fehérré kell alakítanunk, majd a zaj eltávolítása érdekében 5-ös erősségű Gaussianblurt használunk. Ezután Canny edge detectiont hajtunk végre, kísérletezéseim szerint az 50 lower edge thresholdnak megfelelő, 150 pedig upper edgenek szintén az. Ez pont elég a papír széleinek kimutatásához, a zajra még nem túl érzékeny.

A cv::findContours-al kinyerjük a képből a különböző alakzatokat, a RETR\_EXTERNAL paraméter segítségével. Ez azért fontos, mert nekünk ebben a függvényben a belső részletek nem kellenek, vonalakat különállóan dolgozzuk fel, mely megfelelő.

A képen szöveg, diagram, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.Ezután végigválogatjuk a kinyert kontúrokat, megtartva csak azokat, melyek konvex négyszöget jelentenek, ezek közül is a legnagyobbat, feltételezve, hogy az jó eséllyel a papír lesz.

ábra 3 A vonaldetektálás

A fekete vonalak OpenCV-alapú detektálásához először a képet szintén szürkeárnyalatossá alakítjuk, majd GaussianBlur szűrőt alkalmazunk az apróbb zajok, textúrák eltüntetése érdekében (jelen esetben is egy 5x5-ös ablakkal dolgozunk). Ezután az adaptív küszöbölés következik, amely minden képrészletet a helyi átlagtól függően tesz feketévé vagy fehérré; ez a papír különböző megvilágítási viszonyai mellett is kiemeli a rajzolt vonalakat.

A bináris képen Canny edge detectort alkalmazunk, ahol a 50-es és 200-as küszöbértékekkel állapítjuk meg, mely élek számítanak tényleges kontúroknak. A jól látható élekből a HoughLinesP algoritmus segítségével vonalszegmenseket keresünk, melyek hosszát, összefüggését (minLineLength = 40 px, maxLineGap = 20 px) és találati küszöbét (houghThreshold = 40) külön paraméterekkel szabályozzuk. Az így kapott sok rövid, gyakran egymással párhuzamos, közel elhelyezkedő szegmenst a MergeColinearClusters függvénnyel összevonjuk. Itt a 7 foknál kisebb szögkülönbségű és 15 pixelnél közelebb eső szakaszokat egy csoportba rendezve, minden klaszterből egyetlen, hosszabb összekötő szakaszt generálunk. Végül a megmaradt, letisztított vonalszakaszokat a pipeline visszaadja további feldolgozásra vagy AR-vizualizációhoz.

# Eredmények és problémák

A projekt eredménye az volt, hogy egy stabil OpenCV C++ programot raktam össze, mely képes Android készülékek kameraképeiről egy papírt következetesen detektálni, valamint a rajta található vonalakat egy finomított OpenCV csővezeték alkalmazásával kimutatni, a zaj minimalizálásával. Ezt a létrehozott OpenCV bővítményt integráltam a Unity programba AR feldolgozás céljából úgy, hogy feltérképeztem a Unity AR pipeline koordinátakonverziójának mikéntjét, és összeállítottam egy olyan prototípust, mely az így kinyert adatokból képes az Android készülékére a detektált vonalakból falakat extrapolálni.

**A nehézség**, amelybe ütköztem, az volt, hogy hogyan konvertáljak pontosan a kapott XrCpuImage kép koordinátarendszeréből a Unity Camera View koordinátarendszerébe. Ezt a folyamatot, mint azt korábban említettem, az AR Camera Background egy shader segítségével végzi el, hogy azt a hátteret jelenítse nekünk, melyet a Unity AR android használata során látunk. Hosszas kutakodás után sem találtam egyértelmű, hiteles vagy teljesen pontosan működő leírást arról, hogy ezt pontosan hogyan teszi, illetve hogyan tudnám ezt a konverziót Unity-ben megoldani. Magának a shadernek a megvizsgálása során az alábbi sort találtam:

textureCoord = (\_UnityDisplayTransform \* vec4(gl\_MultiTexCoord0.x, 1.0f - gl\_MultiTexCoord0.y, 1.0f, 0.0f)).xy;

Ebből arra a következtetésre jutottam, amelyet a dokumentáció [2] is állít, miszerint a DisplayMatrix, melyet a ARCameraFrameEventArgs-ból nyerhetünk, tartalmazza a szükséges konverziókat ahhoz, hogy azokat a koordinátákat kapjuk meg, amiket az Android készülék képernyőjén láthatunk, miután a nyers kamera képről átkonvertáltuk őket. Ez a konverziós módszer azonban nem teljesen a várt eredményt hozta, egy ismeretlen mennyisséggel csökkenti a detektált papír méretét, illetve gyanúm szerint el is tolja. A papír pozíciója, orientációja megfelelő, a mérete az, amely nem tökéletes.

A projektben ez a legnagyobb kihívás, jelenleg nem találtam erre pontosan működő módszert. Sajnos ezt a konverziót nem tudjuk kikerülni, mert az OpenCV működéséhez csak az XrCpuImage-et tudjuk felhasználni, a már létező ARCamerBackground másolására nem találtam hosszas kutakodás után sem ismert módszert.

További probléma, hogy zárt, négyszög alakú rajzot jelenleg nem tudunk a papírra rajzolni úgy, hogy azt stabilan falakként detekálja, ugyanis az erős, markáns kontúrokat az arra rajzolt papír felettinek érzékeli a fentebb tárgyalt módszer miatt, ezért magát a rajzot érzékeli papírnak. Jelenleg egyéb alakzatok extrapolálhatóak a tollvonásokból.

# Jövőbeli tervek

A projekt jövőbeli tervei közé tartozik a fenti problémák megoldása elsősorban, valamint a szoftver továbbfejlesztése arra, hogy stabilan felhasználható legyen teljes alaprajzok extrapolálásra is. Terv, hogy a falak paraméterei szerkeszthetőek legyenek, ajtókat, ablakokat helyezhessünk beléjük interaktív UI felületen keresztül. A lapdetekálás stabilitásának megoldásához megfelelő megoldás lehet markerek használata az instabil OpenCV detektálás helyett, csupán a vonalak detektálását hagyjuk az OpenCV „vállán”. A natív kódot tovább lehetne finomítani potenciális, stabilabb eredmények elérése érdekében más fényviszonyok között is, bár ez esetben technológiai korlátok is szóba jöhetnek.