REPORT

응용프로그래밍

전기전자공학부

2017142136 이관희

연세대학교
YONSEI UNIVERSITY

1. 주제

개인보고서

2. 담당 영역

내가 담당한 영역은 android studio에서 할 수 있는 모든 것이다. 각 xml(UI) 별로 구현되어 있는 코드를 살펴보는 식으로 진행할 것이다.

3. login.kt

첫 번째로 소개할 장면은 로그인 화면이다. 이 로그인 화면은 최종 구현 화면에서 제외하였다. 시간상의 이유로 회원가입에 대한 정보를 서버에 저장하고 그 정보를 받아오는 과정을 구현하지 못했기 때문이다. 그래서 보고서에도 제외시킬려고 했지만 이미 만들어 놓고 스타일에 시간을 들인 것이 아까워서 적게 되었다.

```
class login : AppCompatActivity() {

    val binding by lazy { LoginBinding.inflate(layoutInflater)}
    val binding_sub by lazy { TestBinding.inflate(layoutInflater)}

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(binding.root)

        //액티비트 이름의 파일내에 저장한다.
        var pref = this.getPreferences( mode: 0)

        // 로그인 정보 불러오기
        with(binding)
        {  this: LoginBinding
            playerSelect.setText(pref.getString( key: "id", defValue: ""))
            password.setText(pref.getString( key: "password", defValue: ""))
        }

        binding.password.setOnEditorActionListener { v, actionId, event ->
            if(actionId == EditorInfo.IME_ACTION_DONE){
                Login(v)
                true  ^setOnEditorActionListener
            }
            else false  ^setOnEditorActionListener
        }
    }
```

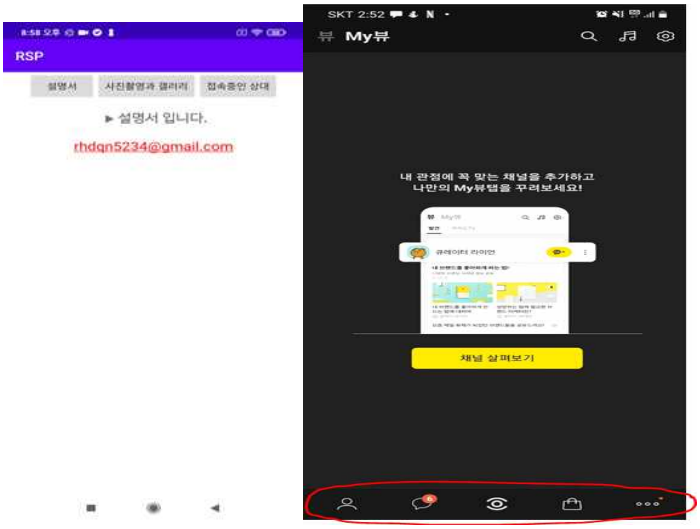
사소해 보일 수 있지만 사용자의 편의에 대해 신경을 많이 썼다. 한번 로그인하면 그 로그인한 정보가 유지되고 마지막 password도 치게 되면 자동적으로 login이 (로그인버튼을 누르지 않아도) 되도록 설계하였다.

```
fun Login(v : View){
    //키보드 숨기기
    var imm = getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
    imm.hideSoftInputFromWindow(v.windowToken,  flags: 0)

    //입력정보 비교 -> 백엔드 서버와 같으면 성공으로 바꿔야함.
    if(binding.playerSelect.text.toString() == "player1" && binding.password.text.toString() == "1234") {
        Toast.makeText( context: this,  text: "로그인 성공!!", Toast.LENGTH_SHORT).show()

        var intent=Intent( packageContext: this@login,Test:: class.java)
        intent.putExtra( name: "id",binding.playerSelect.text)
        startActivity(intent)
    }
    else if(binding.playerSelect.text.toString() == "player2" && binding.password.text.toString() == "4321") {
        Toast.makeText( context: this,  text: "로그인 성공!!", Toast.LENGTH_SHORT).show()

        var intent=Intent( packageContext: this@login,Test:: class.java)
        intent.putExtra( name: "id",binding.playerSelect.text)
        startActivity(intent)
    }
    else Toast.makeText( context: this, text: "로그인 실패",Toast.LENGTH_SHORT).show()
}
```

여기서 로그인 정보는 위와 같이 구현하였다. 현재는 백엔드 서버에서 받아온 회원가입 정보와 일치하게 구현하지 못하고 이미 정해둔 id와 password만으로 로그인할 수 있게 구현하였다. 이렇게 로그인 버튼을 누르게 되면 다음 화면으로 넘어간다.

3. Test.kt

여기서는 하고 싶은 말이 많다. 일단 최종 작품전의 이미지를 살펴보자.



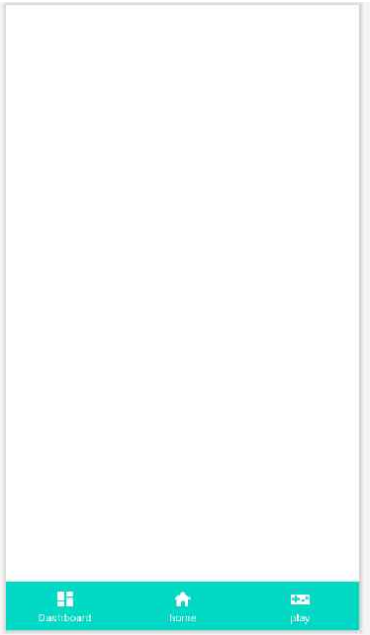
원래는 위와 같이 구현하고 저기에 설명서, 사진촬영과 갤러리, 접속중인 상대의 버튼을 누르고 그 버튼이 누르면 나머지 layout이 `android:visibility="invisible"` 이걸 이용하여 보이지 않도록 설계하였다. 즉 하나의 activity로 구현을 하였다. 여기서는 2가지 문제점이 있다. UI가 난잡하다는 점과 하나의 activity를 이용하여 만들어서 전역변수, intent 등 신경쓰지 않아도 될 것들이 많아져 편해질 수 있지만 너무 많은 변수들이 하나의 activity에 선언되어 있어서 이를 제어하는데 어려움이 있고 특히나 kotlin의 view binding을 사용할 때 id가 겹칠 수 있다는 단점이 있다. 물론 버튼을 누를 때 마다 activity를 이용할 수도 있지만 사실 이것도 좋은 설계 기법은 아니다. 왜냐하면 버튼을 누를 때마다 새로운 activity가 뜨게 되어서 효율적인 코드가 아니라는 단점 때문이다. 그럼 어떻게 해야 하는가? 구글링 결과 최근에 가장 많이 쓰이는 방법은 fragment를 이용한 방법이다.

우리가 카카오톡을 사용할 때 위의 그림처럼 빨간색 바를 본적이 있을 것이다. 이걸 네비게이션 바라고 부르는데 이 네이게이션 바를 기준으로 위의 부분만 바뀌는 것을 볼 수 있을 것이다. 이렇게 한 activity에서 바뀌는 부분을 fragment라고 한다. 즉 이를 이용하면 효율적인 코드로 구현할 수 있다. 즉 이제 우리가 할 것은 fragment을 만들고 그에 맞는 UI만을 구현하면된다. 이는 acrivity라는 개념에서 fragment라는 개념으로 넘어왔기 때문에 사실 많은 것을 공부해야 한다. (가장 기본적으로 life cycle이라는 개념에 차이가 있다. 또한 view binding을 설정하는 방법도 차이가 있다.)

3.1 test.xml과 text.kt

test.xml은 네비게이션 바만을 구현해주면 된다. xml은 간단한 반면에 text.kt는 앞으로 별어질 통신 등의 변수를 전역적으로 관리하기 위한 선언과 몇 가지 필요한 function을 정의해야 한다. 먼저 fragment는 아래처럼 keyword 상수값으로 선언하여 왔다갔다가 가능하도록 설계하자.

```
private const val TAG_HOME_FRAGMENT = "home_fragment"
private const val TAG_DASHBOARD_FRAGMENT = "dashboard_fragment"
private const val TAG_GAME_FRAGMENT = "game_fragment"
```




```
// 바인딩 초기화
val binding by lazy { TestBinding.inflate(layoutInflater) } // main xml에 접근 가능
private  val dashboardFragment = DashboardFragment()
private  val homeFragment=HomeFragment()
private  val gameFragment=GameFragment()

companion object{
    var socket = Socket()
    lateinit var writeSocket: DataOutputStream
    lateinit var readSocket: DataInputStream
    lateinit var cManager: ConnectivityManager
    var ip = "210.205.221.44"
    var port = 9999
    var mHandler = Handler()
    var closed = false

    lateinit var Home : Test
    var send_file_number : Int = 0
    lateinit var text_hello : String
}
```

후에 다시 볼 수 있지만 sockt과 writesocket readsocket cManager ip port mHandler 은 통신을 위해 activity에 선언해두었다. 이 값을 fragment에서 가져와서 사용할 것이다. 여기서 kotlin과 java의 가장 큰 차이가 나오는데 binding 변수를 사용한다는 점이다. 이는 view binding을 의미하는 것으로 이를 통해 activity에 있는 item의 click을 효과적으로 관리할 수 있다.

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    //레이아웃(root뷰)표시
    setContentView(binding.root)
    cManager = applicationContext.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    socket.close()

    // Default Fragment of MainActivity
    setFragment(TAG_DASHBOARD_FRAGMENT, DashboardFragment())

    binding.navigationView.setOnNavigationItemSelectedListener { item ->
        when (item.itemId) {
            R.id.ic_home -> setFragment(TAG_HOME_FRAGMENT, HomeFragment())
            R.id.ic_play -> setFragment(TAG_GAME_FRAGMENT, GameFragment())
            R.id.ic_dashboard -> setFragment(TAG_DASHBOARD_FRAGMENT,DashboardFragment())
        }
        true  ^setOnNavigationItemSelectedListener
    }

    mHandler = object : Handler(Looper.getMainLooper()){  //Thread들로부터 Handler를 통해 메시지를 수신
        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            when(msg.what){
                1-> Toast.makeText( context: this@Test,  text: "IP 주소가 잘못되었거나 서버의 포트가 개방되지 않았습니다.", Toast.LENGTH_SHORT).show()
                2-> Toast.makeText( context: this@Test,  text: "이미지 전송에 실패하였습니다.", Toast.LENGTH_SHORT).show()
            }
        }
    }
    //if(intent.hasExtra("id")) text_hello = intent.getStringExtra("id")+"님 환영합니다.~"
}
```

mhandler는 단순히 어떤 오류가 떴는지 살펴보기 위해서 선언한 것이고 그 외에는 쓰지 않으니 상관하지 말고 먼저 볼 것은 navigationView에서 네이게이션의 어떤 item을 선택하게 되면 일어나게 되는 부분인 가운데 부분이다. 여기서 각 id의 이름을 가져오고 그 값과 선택한 값이 일치하면 setFragment를 실행한다. setFragment은 내가 구현한 함수로 원래는 life cycle에 의해 네이게이션 바를 누르게 되면 여태한 작업이 초기화가 된다. 이를 방지하고 네비게이션바를 돌아가면서 누르더라도 기존의 작업이 유지할 수 있게 다음과 같이 선언하였다. 여기서 핵심은 hide와 show를 이용하여 fragement를 종료하지 않는다는 점이다.

```kotlin
78      /* Fragment State 유지 함수 */
79      private fun setFragment(tag: String, fragment: Fragment) {
80          val manager: FragmentManager = supportFragmentManager
81          val ft: FragmentTransaction = manager.beginTransaction()
82
83          if (manager.findFragmentByTag(tag) == null) {
84              ft.add(R.id.frameLayout, fragment, tag)
85          }
86
87          val home = manager.findFragmentByTag(TAG_HOME_FRAGMENT)
88          val game = manager.findFragmentByTag(TAG_GAME_FRAGMENT)
89          val dashboard = manager.findFragmentByTag(TAG_DASHBOARD_FRAGMENT)
90
91          // Hide all Fragment
92          if (home != null) {
93              ft.hide(home)
94          }
95          if (game != null) {
96              ft.hide(game)
97          }
98          if (dashboard != null) {
99              ft.hide(dashboard)
100         }
101         // Show  current Fragment
102         if (tag == TAG_HOME_FRAGMENT) {
103             if (home != null) {
104                 ft.show(home)
105             }
106         }
107         if (tag == TAG_GAME_FRAGMENT) {
108             if (game != null) {
109                 ft.show(game)
110             }
```

```
111         }
112         if (tag == TAG_DASHBOARD_FRAGMENT) {
113             if (dashboard != null) {
114                 ft.show(dashboard)
115             }
116         }
117         ft.commitAllowingStateLoss()
118     }
119
120     class Connect:Thread(){
121         override fun run(){
122             try{
123                 //socket = Socket("192.168.40.11", 9997)
124                 socket = Socket(ip, port)
125                 writeSocket = DataOutputStream(socket.getOutputStream())
126                 readSocket = DataInputStream(socket.getInputStream())
127             }catch(e: Exception){    //연결 실패
128                 mHandler.obtainMessage( what: 1).apply {  this: Message
129                     sendToTarget()
130                 }
131                 //socket.close()
132             }
133         }
134     }
135     // fragmentManager.beginTransaction().remove(fa).commit();
136 }
```

connet함수가 나오는데 이는 socket통신을 위한 준비과정이라고 생각해두자. connect은 나중에 fragment_dashboard에서 다시 나오니 그때 알아보기로 하고 지금은 ip와 portnumber을 받아서 서버와 연결하는 함수라고만 생각해두자.

3.2 fragment_dashboard의 xml과 kt 파일

여기서 navigation var가 없는데 당연하게 여기서부터는 fagment이고 test.xml파일에서 네비게이션바 위에 정의되어 있는 UI이다. 여기서는 기본적으로 설명서 text를 눌러 설명서를 볼 수 있고 ip와 port number을 적어서 서버와 연결을 진행한다. 또한 빨간색 이메일을 누르면 사용자의 불만 사항이 운영자의 이메일로 갈수 있도록 설계하였다. 동작 과정은 최종 발표 자료를 확인하자.

```
class DashboardFragment : Fragment(), View.OnClickListener {
    private var _binding: FragmentDashboardBinding?=null
    private val binding get() = _binding!!
    override fun onAttach(context: Context) {
        super.onAttach(context)
        Home = context as Test
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // 1. 뷰 바인딩 설정
        _binding= FragmentDashboardBinding.inflate(inflater,container, attachToParent: false)
        val view = binding.root
        return view
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.manual.setOnClickListener (this)
        binding.buttonConnect.setOnClickListener(this)
    }
```

먼저 view binding이다. 이를 보면 activity와는 다른 형태로 정의 되어 있다. 여기서 class는 Fragment()를 상속받고 onClick함수의 orride를 위해 view.onclicklistenser을 implement 한다. fragment에서 중요한 점은 lifecycle을 명확히 알아야 한다는 것이다. 이에 대해 잘 설명되어 있는 링크를 첨부해둔다.

https://readystory.tistory.com/199

간단히 말하자면 onCreateView는 activity에서 onCreate와 비슷하고 onViewCreated는 그 후 오는 cycle이다. 어쨋든 OnClick 함수를 따로 만들었기 때문에 manual을 클릭했을 때 어떻게 되는지와 connect버튼을 눌렀을 때 어떻게 되는지는 따로 알아보도록 하고 여기선 button을 누르거나 설명서 text를 누르면 어떤 독장이 작동한다는 것만 파악해두자.

```
    override fun onClick(v: View?) {
        when(v?.id){
            binding.manual.id -> {
                binding.manual.text = "이 프로그램은 가위바위보 하나삐기를 구현한 앱입니다. \n" +
                        "기본적으로 사진을 찍고 그 사진으로 게임이 진행됩니다.\n"+
                        "사진을 찍고 play 버튼을 눌러주세요\n"+
                        "한번더 반복합니다.\n"
                binding.manual.textSize = 12.0F
            }
            binding.buttonConnect.id -> {
                if(binding.etIp.text.isNotEmpty()) {
                    ip = binding.etIp.text.toString()
                    if(binding.etPort.text.isNotEmpty()) {
                        port = binding.etPort.text.toString().toInt()
                        if(port <0 || port >65535){
                            Toast.makeText(Home,  text: "PORT 번호는 0부터 65535까지만 가능합니다.", Toast.LENGTH_SHORT).show()
                        }else{
                            if(!socket.isClosed){
                                Toast.makeText(Home,  text: ip + "에 이미 연결되어 있습니다.", Toast.LENGTH_SHORT).show()
                            }else {
                                Test.Connect().start()
                            }
                        }
                    }else{
                        Toast.makeText(Home,  text: "PORT 번호를 입력해주세요.", Toast.LENGTH_SHORT).show()
                    }
                }else{
                    Toast.makeText(Home,  text: "IP 주소를 입력해주세요.", Toast.LENGTH_SHORT).show()
                }
            }
        }
    }
}
```

앱에서도 사용자가 어떤 오류인지 알수 있도록 toast를 이용하여 출력하도록 설계하였다. 처음을 보면 설명서를 버튼을 누르면 binding.maual.id가 나오고 이제 이 값이 원래 설명서 부

분에 대신 출력된다. connect버튼을 누르면 적은 ip와 portnumer를 전역변수에 저장해두고 test의 connect함수를 호출한다. 여기서는 이미 본 것처럼 socket을 이용하여 소켓통신이 가능하다. (TCP/IP) 그리고 dataoutputstream과 datainputstream이 있는데 이는 각각 내가 서버에 데이터를 보냈을 때 .서버에서 데이터를 받을 때 사용한다.

3.3 fragment_home kt와 xml 파일

여기서는 일단 사용자의 편의를 생각한 것이 있다. 먼저 내가 몇장을 보냈지는 알 수 있도록 출력할 수 있도록했고 사진을 찍고 play button을 눌러야만 사진이 전송될 수 있는데 이때 한번 보낸 사진을 실수로 두 번 보내지 않도록 사진을 다시 찍을 때까지 play버튼을 defalut를 해서 click할 수 없도록 설계하였다. 여기서 중심적으로 구현해야 하는 것은 사진 찍기, 사진 불러오기, 사진 전송이다. 먼저 수업시간과 다른 점을 설명하자면 사진촬영에서 권한을 물어보도록 설계하였다. 이렇게 설계함으로써 사용자가 클릭했을 때 원래 권한을 설정하지 않으면 꺼지지만 그 꺼지는 이유를 알도록 설계할 수 있다. 매니페스트도 변경해야 하지만 이는 수업중에 했으므로 대충 첨부파일만 남기겠다.

```
5     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/> <!--  파일쓰기 권한 -->
6     <uses-permission android:name="android.permission.CAMERA"/>                 <!--  카메라 권한 -->
7     <uses-feature android:name="android.hardware.camera2"/>                     <!--  카메라 권한 -->
8     <uses-permission android:name="android.permission.INTERNET"/>               <!--  인터넷 권한 -->
9     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>   <!-- 네트워크 권한 -->
10    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>      <!-- WIFI 권한 -->
```

```
class HomeFragment : BaseActivity(), View.OnClickListener {

    var _binding: FragmentHomeBinding? = null
    val binding_home get() = _binding!!
    val PERM_STORAGE = 100
    val PERM_CAMERA= 105
    val REQ_CAMERA=110
    var realUri : Uri? =null     // 원본 이미지의 주소를 저장할 변수

    override fun onAttach(context: Context) {
        super.onAttach(context)
        //공용 저장소 권한이있는지 확인
        requirePermissions(arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE), PERM_STORAGE)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // 1. 뷰 바인딩 설정
        _binding= FragmentHomeBinding.inflate(inflater,container, attachToParent: false)
        val view = binding_home.root
        binding_home.imgButton.setEnabled(false)
        return view
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding_home.imgButton.setOnClickListener(this)
    }
```

여기서 일단 볼 것은 setEnabled을 이용하여 playbutton 동작 유무 설정하였다. 그리고 여기서는 frame()이 아닌 baseActivity()를 상속받는데 이는 구현한 class로 아래와 같다. 권한 설정을 위한 class라고 생각하면 된다.

```
abstract class BaseActivity : Fragment() {
    abstract fun permissionGranted(requestCode: Int)
    abstract fun permissionDenied(requestCode: Int)

    //권한 검사
    fun requirePermissions(permissions: Array<String>, requestCode: Int) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.M) {
            permissionGranted(requestCode)
        } else {
            requestPermissions(permissions, requestCode)
        }
    }

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
        if (grantResults.all { it == PackageManager.PERMISSION_GRANTED }) {
            permissionGranted(requestCode)
        } else {
            permissionDenied(requestCode)
        }
    }
}
```

```
    override fun onDestroyView() {
        super.onDestroyView()
        _binding==null
    }
    //권한 거부 시
    override fun permissionDenied(requestCode: Int) {
        when(requestCode){
            PERM_STORAGE -> Toast.makeText(Home, text: "공용저장소 권한을 승인해야 앱을 사용할 수 있습니다.", Toast.LENGTH_SHORT).show()
            PERM_CAMERA -> Toast.makeText(Home, text: "카메라 권한을 승인해야 카메라를 사용할 수 있습니다.", Toast.LENGTH_SHORT).show()
        }
    }
    // 권한 승인 시
    override fun permissionGranted(requestCode: Int) {
        when (requestCode) {
            PERM_STORAGE -> initViews()
            PERM_CAMERA -> openCamera()
        }
    }
    fun initViews(){
        binding_home.cameraBtn.setOnClickListener(this)
    }
    //카메라에 찍은 사진을 저장하기 위한 uri을 넘겨준다.
    fun openCamera(){
        val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        createImageUri(newFileName(),  mimeType: "image/jpg")?.let{ uri ->
            realUri =uri
            intent.putExtra(MediaStore.EXTRA_OUTPUT, realUri)
            startActivityForResult(intent,REQ_CAMERA)
        }
    }
```

각각 큰 주석이 있으므로 따라가면 된다. 카메라를 설정하는 부분은 수업 내용과 유사하다.

```
    //원본 이미지를 저장할 Uri를 MediaStore(데이터 베이스)에 생성하는 메서드
    fun createImageUri(filename:String,mimeType:String): Uri?{
        val values = ContentValues()
        values.put(MediaStore.Images.Media.DISPLAY_NAME,filename)
        values.put(MediaStore.Images.Media.MIME_TYPE,mimeType)

        return Home.contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,values)
    }

    fun newFileName() : String{
        val sdf = SimpleDateFormat( pattern: "yyyyMMdd_HHmmss")
        val filename = sdf.format(System.currentTimeMillis())
        return "${filename}.jpg"
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        super.onActivityResult(requestCode, resultCode, data)
        if(data!= null) binding_home.ivBasic.setImageURI(data?.data)
        if(resultCode == AppCompatActivity.RESULT_OK){
            when(requestCode){
                REQ_CAMERA ->{
                    realUri?.let{ uri ->
                        val bitmap = loadBitmap(uri)
                        binding_home.ivBasic.setImageBitmap(bitmap)
                        realUri = null
                    }
                }
            }
        }
    }
```

```
//원본 이미지를 불러오는 메서드
fun loadBitmap(photoUri : Uri): Bitmap?{
    var image: Bitmap?=null
    try{
        if(Build.VERSION.SDK_INT> Build.VERSION_CODES.O_MR1){
            val source : ImageDecoder.Source = ImageDecoder.createSource(Home.contentResolver,photoUri)
            image = ImageDecoder.decodeBitmap(source)
        }else{
            MediaStore.Images.Media.getBitmap(Home.contentResolver,photoUri)
        }
    }catch (e: Exception){
        e.printStackTrace()
    }
    return image
}
```

위에 빨간줄이 뜨기는 하는데 실행에는 문제가 없다. 그러니 무시해도 상관없다. 이제 볼 것은 서버로 전송하는 파트이다.

```
//이미지 보내기
class SendImage:Thread() {
    private lateinit var rotatedBitmap: Bitmap
    fun setimage(m1: Bitmap) {
        rotatedBitmap = m1
    }
    override fun run() {
        var byteArray: ByteArrayOutputStream = ByteArrayOutputStream()
        rotatedBitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 30, byteArray)
        var bytes = byteArray.toByteArray()
        try {
            writeSocket.writeUTF(Integer.toString(bytes.size))
            writeSocket.flush()
            writeSocket.write(bytes)
            writeSocket.flush()
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

여기서는 중요한 것이 찍은 사진을 rotatedBitmap(Bitmap으로 변형시켜서 받아야 한다. 그냥 binding으로 접근하면 view image이다.)으로 받고 이를 byteArray로 바꾸어서 보낸다. 먼저 이미지의 길이를 보내고 이미지를 byteArray로 바꾼 값을 보내준다. 여기서 flush는 버퍼를 비우는 작업이다.

```
override fun onClick(v: View?) {
    when(v?.id) {
        binding_home.imgButton.id ->{    //상대에게 메시지 전송
            if(socket.isClosed){
                Toast.makeText(Home,  text: "연결이 되어있지 않습니다.", Toast.LENGTH_SHORT).show()
            }else {
                val mThread = SendImage()
                mThread.setimage(binding_home.ivBasic.getDrawable().toBitmap())
                mThread.start()
                binding_home.imgButton.setEnabled(false)
                send_file_number+=1
                if(send_file_number==1)
                    binding_home.numberSend.text="한 장 더 보내야 합니다."
                else if(send_file_number==2)
                    binding_home.numberSend.text="두 장 다 보냈습니다."
            }
        }
        binding_home.cameraBtn.id ->{
            requirePermissions(arrayOf(Manifest.permission.CAMERA), PERM_CAMERA)
            if(send_file_number!=2)
                binding_home.imgButton.setEnabled(true)
        }
    }
}
}
```

여기서는 무슨 버튼을 클릭했을 때 어떤 동작이 일어날지 적혀있다. 먼저 play button (imgButton)을 누르면 이미지를 서버에 보내게 되는데 여기서 getDrawable().toBitmap()을 이용하여 setimage 함수에 bitmap 값을 전달 해줄 수 있다. send_file_number은 사진을 2장 보내면 더 이상 보낼 수 없도록 playbutton을 defalut값으로 만들어주기 위한 flag이다. 두 번째로 카메라 버튼을 누르면 opencamera전에 권한을 물어보게 되는데 권한을 물어보면서 자연스럽게 opencamera로 이어진다.(위에 코드를 잘 살펴보면 알 수 있다.)

3.4 fragment_game

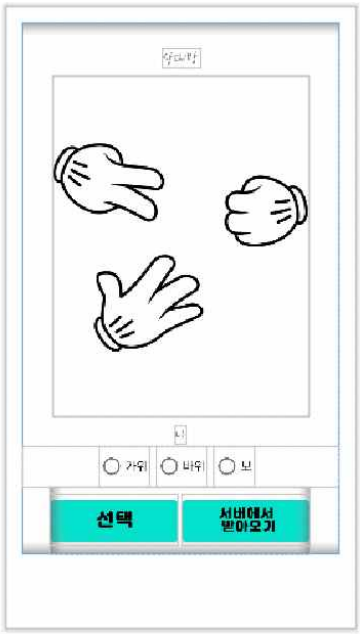
가장 핵심이자 시간이 제일 많이 든 파트이다. 이 전까지는 서버에 보내기만 했지만 이제부터는 서버에서 정보를 받아야 한다. 서버가 하는 일은 일단 사용자 2명이 전송한 2장의 파일을 가지고 유니티를 이용하여 씬을 받아오고 서버에서는 찍은 사진중 가위바위보중 없는 요소를 전달해주고 또 나에 대한 화면이 아래에 오도록 flip값을 전달해준다. 즉 순서는
이미지 길이 -> 이미지 -> 가위바위보 요소중 없는 값 -> filp 여부 순으로 데이터를 전달해준다. 이를 순서대로 잘 받는 것이 목적이다. 먼저 살펴볼 것은 이에 대한 여부를 확인하기 위한 임시 서버 구현을 살펴보자.

```python
def send_img(sock, img_path):
    img_f = open(img_path, "rb")
    data = img_f.read()
    data_len = len(data)
    sock.sendall(data_len.to_bytes(4, byteorder="big"))
    print("data length:", data_len)

    step = 1024
    loop = int(data_len / step) + 1

    while len(data) > 0:
        if len(data) < step:
            sock.sendall(data)
            data = []
        else:
            sock.sendall(data[:step])
            data = data[step:]
```

```python
while True:
    server_sock = socket.socket(socket.AF_INET)
    server_sock.bind((HOST, PORT))
    server_sock.listen(10)
    print("Wait")
    client_sock, addr = server_sock.accept()
    print('Connected by : ', addr[0], ':', addr[1])
    socket_list.append((client_sock, addr))
    print("Send")
    send_img(client_sock, path)
    client_sock.send(d.to_bytes(4, byteorder='big'))
    client_sock.send(a.to_bytes(4, byteorder='big'))

    client_sock.send(k.to_bytes(4, byteorder='big'))
    print("Done")
```

간단하게만 설명하자면 sen_img가 img길이와 img를 전할해주고 48번줄 이 d의 값을 전달해주고 (d= 0 -> 찍은 사진중에 묵이 없다는 의미이다.) 49번줄이 a(a=20 flip해라는 의미이다.) 51번 k는 승무패를 전달해주는 데이터로 (k=10 승리를 의미한다.) 이를 모두 전달해주고 android는 적절한 순간에 받아야 한다.

이제 android로 돌아와보자.




```
21  class GameFragment : Fragment(),View.OnClickListener {
22      private var _binding: FragmentGameBinding? = null
23      private val binding get() = _binding!!
24      private var number = -1
25      private var win_lose = -1
26      override fun onCreateView(
27          inflater: LayoutInflater, container: ViewGroup?,
28          savedInstanceState: Bundle?
29      ): View? {
30          _binding = FragmentGameBinding.inflate(inflater, container,  attachToParent: false)
31          val view = binding.root
32          return view
33      }
34
35      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
36          super.onViewCreated(view, savedInstanceState)
37          binding.btnRecServer.setOnClickListener(this)
38      }
39
```

여기서는 그냥 btnRecServer(서버에서 받아오기)을 누르면 onclick에서 일정한 행동이 진행된다고만 생각하자.

```
override fun onClick(v:View?){
    when(v?.id){
        binding.btnRecServer.id -> {
            if (socket.isClosed) {
                Toast.makeText(Home,  text: "연결이 되어있지 않습니다.", Toast.LENGTH_SHORT).show()
            } else {
                var mThread = getImage()
                mThread.start()
                try {
                    mThread.join()
                } catch (e: Exception) {
                    e.printStackTrace()
                }
                var img_byte = mThread.img_bytearry()
                var non_int = mThread.img_int()
                if (img_byte != null) {
                    binding.playerCapture.setImageBitmap(
                        BitmapFactory.decodeByteArray(
                            img_byte,
                             offset: 0,
                            img_byte.size
                        )
                    )
                    var a = mThread.flip_int()
                    Log.d( tag: "로그", a.toString());
                    if(a!=null) {
                        if (a == 20) binding.playerCapture.scaleY = -1.0F else binding.playerCapture.scaleY = 1F
                    }
                    Log.d( tag: "로그",  msg: "  오케이");
                } else {
                    Log.d( tag: "로그",  msg: "실패");
                }
                 if (non_int != null) {
                     number = non_int
                     Log.d( tag: "로그",  msg: "non_int : " + number.toString());
                  }else 0
                }
                if (number == 0) {
                    binding.rock.visibility = View.GONE
                }
                if (number == 1) {
                    binding.scissors.visibility = View.GONE
                }
                if (number == 2) {
                    binding.paper.visibility = View.GONE
                }
            }
```

여기서는 서버에서 받아오기를 누르면 동작하는 일련의 과정이 저장되어 있다. 여기서 보면 중요한 부분은 다 함수로 따로 만들어두어서 다 같이 보도록 하자. 먼저 getImage()라는 thread()를 상속받는 class를 mThread로 정의하고 이를 start한다 start의 의미는 run이 돌아간다는 의미이다. 아래를 보면 getImage() class가 정의되어 있는데 먼저 readInt로 이미지의 길이를 전달받고 이미지 길이를 1024로 나누어서 1024를 기준으로 끊어서 정보를 전달 받는다. for 문 마지막에 readfully는 길이를 1024로 나눈 나머지 값을 마저 전달 받는 것이다. 그다음 찍은 사진 중 없는 요소를 전달받고(하나빼기라서 두장을 찍어서 보내는데 여기서 없는 것을 받으면 그 없는 것에 대한 선택은 없애 버리도록 동작한다.) 마지막으로 사진의 filp 여부를 전달 받는다. 왜냐하면 서버에서는 player 0을 기준으로 게임씬을 만들기 때문이라 player 1은 그대로 전달 받는다면 상대방쪽에 나의 이미지가 오기 때문에 이 정보를 전달 받아야 한다. 이렇게 받은 다음 join함수를 이용하여 새로 만든 Thread가 마칠때까지 기다린다. 이게 중요한 이유가 thread.join()이 없으면 main과 thread가 같이 돌아가서 뭐가 일찍 끝날지도 모르고 만약 main이 먼저 끝난다면 원하는 결과를 얻지 못할 가능성이 높기 때문이다.(이걸 알기 위해서 많은 시행착오를 겪었다.) 그 다음에는 전달 받은 값으로 이미지를 바꾸는 작업이 이어진다.

```
146     class getImage : Thread() {
147         private var resbytes: ByteArray? = null
148         private var resInt: Int? = null
149         private var flipInt: Int? = null
150 
151         override fun run() {
152             try {
153                 var length_get_to_server = readSocket.readInt()
154                 Log.d( tag: "로그", length_get_to_server.toString())
155                 val loop = (length_get_to_server / 1024)
156                 Log.d( tag: "로그",  msg: "loop" + Integer.toString(loop))
157                 resbytes = ByteArray(length_get_to_server)
158                 var offset = 0
159                 try {
160                     for (i in 0 until loop) {
161                         readSocket.readFully(resbytes, offset,  len: 1024)
162                         offset += 1024
163                     }
164                     readSocket.readFully(resbytes, offset,  len: length_get_to_server - loop * 1024)
165                     resInt = readSocket.readInt()
166                     Log.d( tag: "로그",  msg: "resInt :" + resInt.toString())
167                     flipInt = readSocket.readInt()
168                     Log.d( tag: "로그",  msg: "resInt :" + flipInt.toString())
169                 } catch (e: IOException) {
170                     e.printStackTrace()
171                 }
172             } catch (e: Exception) {
173                 e.printStackTrace()
174             }
175         }
176         fun img_bytearry():ByteArray?{
177             return this.resbytes
178         }
179         fun img_int():Int?{
180             return this.resInt
181         }
182         fun flip_int():Int?{
183             return this.flipInt
184         }
185     }
```

이제 볼 것은 서버에 받은 정보로 내가 내고 싶은 가위바위보 요소중 선택한 후 서버로 보내면 연산 결과 승 무 패를 전달 받아야 한다.

```
            binding.btnServer.id ->{
                if (socket.isClosed) {
                    Toast.makeText(Home,  text: "연결이 되어있지 않습니다.", Toast.LENGTH_SHORT).show()
                } else {
                    var select_number: Int = -1
                    when (binding.radioGroup.checkedRadioButtonId) {
                        binding.rock.id -> select_number = 0
                        binding.scissors.id -> select_number = 1
                        binding.paper.id -> select_number = 2
                    }
                    Log.d( tag: "로그",  msg: "select" + select_number.toString());
                    var sendThread = SendInt()
                    sendThread.setint(select_number)
                    sendThread.start()
                    try {
                        sendThread.join()
                    } catch (e: Exception) {
                        e.printStackTrace()
                    }
                    if (sendThread.img_send() != null) {
                        win_lose = sendThread.img_send()!!
                    } else 0
                    when (win_lose) {
                        10 -> {
                            binding.win.text = "WIN"
                            binding.win.textSize = 40F
                        }
                        11 -> {
                            binding.win.text = "무"
                            binding.win.textSize = 40F
                        }
                        12 -> {
                            binding.win.text = "Lose"
                            binding.win.textSize = 40F
                        }
                        null -> {
                            binding.win.text = "Null"
                            binding.win.textSize = 40F
                        }
                        -1 -> {
                            binding.win.text = "-1"
                            binding.win.textSize = 40F
                        }
                    }
                }
            }
        }
    }
```

여기서 보면 내가 선택한게 뭔지 값을 받아 서버로 sendint()를 이용해서 전달하고 승무패를 전달 받아서 출력하면 된다.

```
187     class SendInt : Thread() {
188         private var sendInt : Int =-1
189         private var resWin: Int? = null
190
191         fun setint(m1: Int) {
192             sendInt = m1
193         }
194         fun img_send(): Int? {
195             //run()
196             Log.d( tag: "로그",  msg: "Win : " + resWin.toString());
197             return this.resWin
198         }
199         override fun run() {
200             try {
201                 Log.d( tag: "로그", sendInt.toString());
202                 writeSocket.writeUTF(Integer.toString(sendInt))
203                 writeSocket.flush()
204
205                 resWin = readSocket.readInt()
206                 Log.d( tag: "로그",  msg: "Winrun"+resWin.toString())
207             } catch (e: Exception) {
208                 e.printStackTrace()
209                 Test.mHandler.obtainMessage( what: 2).apply {  this: Message
210                     sendToTarget()
211                 }
212             }
213         }
214     }
```

여기서보면 먼저 write로 내가 선택한 요소를 전달하고 승무패 결과를 readInt로 받는다. 이를 모두 포함한 동영상을 첨부로 보고서를 마무리 한다.

4. 최종결과(안드로이드와 서버)

여기서는 간략하게 구현한 서버로 동작된 게임을 볼 것이다. 여기서는 구현 동작만 살펴보기 때문에 받아오는 사진은 원래 유니티 씬 사진이 아니라가 서버에 저장된 임의의 사진이고 filp 정보도 일단은 filp됐다는 가정하에 구현되어져 있다. 일단 여기서 중요한 것은 서버에 정보를 전달하고 서버가 보낸 정보를 잘 받는다는 것이다.

https://youtu.be/IcnCZFeRvRk

정확한 동작 동영상(서버 + 유니티 + 안드로이드)은 최종 발표 자료를 참고

5. 그 외. 글꼴 디자인 등

당연하지만 글꼴과 디자인에 신경을 많이 썼다. 버튼을 눌렀을 때의 변화를 사용하기 위해서 처음에는 custom button을 만들었다.

이 버튼은 누른것과 누르지 않을 때 다른 모양이 나오도록 설계되어 있다. 그런데 이 디자

인에 시간을 쏟았지만 중요한 건 사람의 친숙성이다. 우리는 어떤 버튼에 익숙해져있을까? 이렇게 만든 버튼이 사용자에게 친숙하게 다가갈 수 없다. 구글에서는 이런 문제를 해결하기 위해 버튼과 몇 개의 UI를 통일해서 제공한다는 사실을 알게 되었다. 그래서 통일성을 위해 기존에 만든 custom design을 모두 버리고 구글에서 제공하는 기본 object를 사용했다. 구글 UI를 사용하기 위해서는 아래처럼 gradle을 바꾸어야 하고 themes의 값 역시 아래처럼 바꾸어야한다. 그리고 style에 저런 식으로 필요한 값을 추가해야한다.

```
dependencies {

    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

:ml × AndroidManifest.xml × build.gradle (:app) × themes.xml × login.xml × Login.kt × HomeFragmen

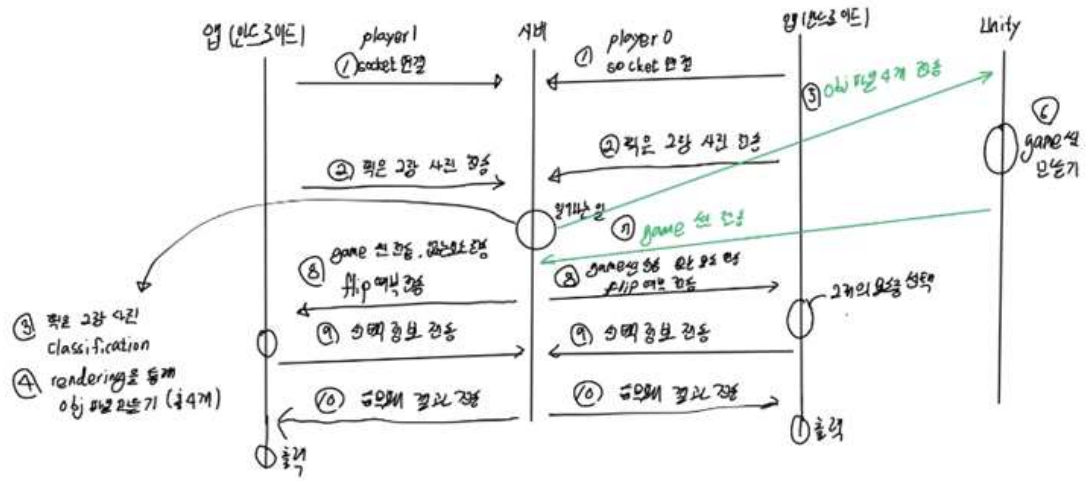
```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Test" parent="Theme.MaterialComponents.Light">
        <!-- Primary brand color. -->
        <item name="colorPrimaryVariant">@color/teal_200</item>
        <item name="colorOnPrimary">@color/black</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_200</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/teal_200</item>
        <item name="colorPrimaryDark">@color/teal_200</item>
        <item name="colorAccent">@color/teal_700</item>

        <!--custom font-family 적용-->
        <item name="android:textViewStyle">@style/customTextViewFontStyle</item>
        <item name="android:buttonStyle">@style/customButtonFontStyle</item>
        <item name="android:editTextStyle">@style/customEditTextFontStyle</item>
        <item name="android:radioButtonStyle">@style/customRadioButtonFontStyle</item>
        <item name="android:checkboxStyle">@style/customCheckboxFontStyle</item>
    </style>
</resources>
```

```
style="@style/Widget.MaterialComponents.Button"
```

위의 themes를 보면 font도 custom font로 모두 바꾼 것을 볼 수 있다.

6. 전체 프로젝트 framework

이건 그림으로 설명해 보겠다. 아래 그림을 살펴보자.



총 11가지의 과정으로 볼 수 있다. 차근차근 알아보자면

1. 먼저 서버는 켜져 있는 상태에서 두 대의 안드로이드가 접속 요청을 하고 socket connection이 이루어진다.
2. 안드로이드는 찍은 2장의 사진을 전송한다.
3. 2대의 안드로이드가 찍은 2장의 사진을 전송하면 서버는 multiThread을 이용해서 정보를 따로 따로 받는다. 이 받은 사진을 mediapipe opensource에 각 point의 거리에 따른 정보로 가공하여 이 정보를 학습시켜 classification을 진행한다.
4. 또한 mano 모델을 이용하여 rendering을 통해 obj파일을 만든다.
5. 총 4개의 obj파일을 unity에 전송한다.
6.unity에서는 받은 obj파일로 game 씬을 만든다.
7. 게임 씬 이미지를 서버로 전송한다.
8. 서버는 unity에서 받은 게임씬 이미지와 classification으로 각 player별로 없는 요소와 flip 정보(flip 정보는 특별한게 아니라 player0을 기준으로 게임씬이 아래가 player 0 위가 player 1로 구현되기 때문에 그냥 player0이면 filp하지 않고 player1이면 filp하도록 설계하여 항상 내가 player1이든 player2이든 나에 대한 가위바위보 정보는 아래에 오도록 설계한다.)를 전송한다.
8. 안드로이드는 전송 받은 정보를 출력하고 2개의 요소중 하나를 선택한다.
9. 2개의 요소중 선택한 하나를 서버에 전송한다.
10. 서버에서는 player0 와 player1이 준 정보로 승무패를 결정하고 각자 승무패 정보를 (10,11,12 각각 승 무 패) int로 변환해서 android에 전달해준다.
11. 안드로이드는 받은 정보를 출력하는 것으로 마무리한다.