Chris Elder

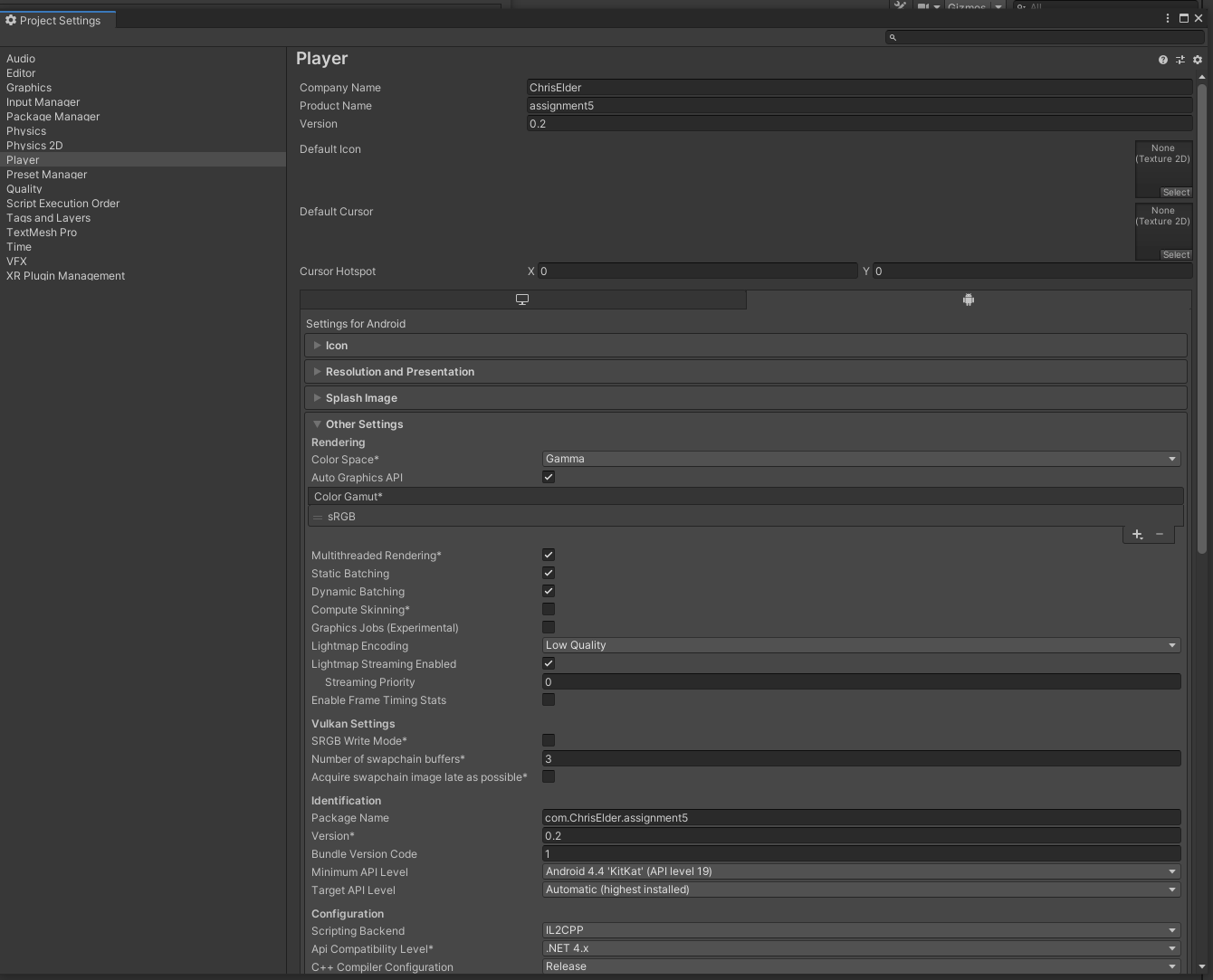
A00804804

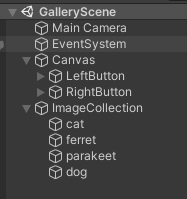
COMP 7031

# **Project Report: Android Studio & Java vs Unity & C#**

Development for Android in Unity was significantly more difficult than development in Java and Android Studio, but Unity offers some unique benefits. Given the differences in focus and structure between the two platforms, this wasn’t surprising.

## **Configuration and Development Environment**

To start, the tools and configuration were more difficult to set up and use. I actually abandoned my first project file, because of some subtle configuration that was wrong. I believe it was a graphics setting. It caused my app to display only some assets that I loaded, and to display those incorrectly (pink splotches instead of different coloured buttons). Android Studio, on the other hand, will create an app that works on Android as-is after setup. Below is a screenshot of some of the configuration that Unity provides for the Android platform. It is an overwhelming amount of options. 



Similarly, it took some time to set up the visible area, adjust the camera, create and scale working buttons, and place the images in the scene as game objects. Unity is, of course, designed for making video games, and is substantially more flexible for likely game use cases than Android Studio. For a more static interface like this project, Android Studio is simply more appropriate. The Unity scene structure is shown to the left.

Unity + Android Studio + Android Emulator is a heavy system load for development and my laptop fans were working overtime. Building the Unity project took noticeably longer than building a small Android project, and any updates to the source meant re-running the entire build. Android Studio can sometimes apply code changes and restart the current Activity without re-installing the application.

Another small but nice feature of Android Studio is that it can start the right emulator automatically. Unity will look for a compatible running emulator and give up if one is not found.

Another advantage of Unity is that I could very easily port my application to another platform. Simply by checking a box and waiting for a rebuild, I was able to use the buttons and view the images on an executable built for my Windows laptop. Of course, the swiping functionality doesn’t work on a non-touchscreen laptop, but that’s a limitation of the platform/hardware.

## **Coding**

Unity generally uses C# instead of Java, and this is not a significant difference. I am personally a fan of C#. In my opinion, it has learned from some of the mistakes and difficulties of Java, and C# is the better language for it. Overall, though, the languages are substantially similar, and no major adaptation is needed. Most other game engines use C++ primarily, and this would be a significant departure from the high-level C# and Java. Where Unity does differ majorly from Android is in structure. Objects in Unity can exist and be interactive with essentially no code added. Code in Unity is added as Scripts, which are files directly attached to a game object. In Android, however, some level of code is necessary for an app to function.

One interesting aspect was that in Unity, I could place an image on top simply by changing its z-index. In Android Studio, this would be possible, but probably an inefficient way to achieve the outcome. Unity is based on UI items primarily, with code attached. Java and XML layouts are essentially the opposite. Unlike an Android Studio project, a Unity project can be set up essentially without any code.

Android Studio Java code:

void updateImageToIndex() {  
 View f = findViewById(R.id.*imageView*);  
 ((ImageView)findViewById(R.id.*imageView*)).setImageResource(imgList.get(index));  
}  
  
public void goRight(View view) {  
 index += 1;  
 if(index >= imgList.size()) {  
 index = 0;  
 }  
 updateImageToIndex();  
}  
  
public void goLeft(View view) {  
 index -= 1;  
 if(index < 0){  
 index = imgList.size() - 1;  
 }  
 updateImageToIndex();  
}

Unity C# code:

void Update() {  
 for(int i = 0; i < images.Length; i++) {  
 int newZ = baseZ + (i == selectedIndex ? 0 : 1);  
 images[i].transform.position = new Vector3(images[i].transform.position.x, images[i].transform.position.y, newZ);  
 }  
}  
  
public void setNext() {  
 if (selectedIndex == (images.Length - 1)) {  
 selectedIndex = 0;  
 } else {  
 selectedIndex += 1;  
 }  
}  
  
public void setPrevious() {  
 if(selectedIndex == 0) {  
 selectedIndex = images.Length - 1;  
 } else {  
 selectedIndex -= 1;  
 }  
}

The two code snippets above show plenty of similarity. The biggest difference is the actual image changing line. In Android, this is setting the resource file of an ImageView, from a Drawable asset. In Unity, this is telling a GameObject, which I know to be an image, to set its z-index as appropriate. The Unity approach was a bit more work for the same result, since I had to set up a custom GameObject to hold the list of image objects and set each initial image z-index. In the Java code, I simply made a list of the names/IDs and grabbed the next one from the list when needed.

In terms of touch input, there were more noticeable differences. For one thing, Android native provides callback functions for a lot of common scenarios. Unity provides a few specific lifecycle hooks, mainly Start() and Update(). Update() is called once per frame, and is where the bulk of processing is usually done. To implement custom gestures on Android, I simply added the plugin, drew and named the gestures, and added OnGestureListener functionality and the appropriate callback function to the Activity.

What Unity provides is a relatively friendly Touches object in Update(). To check for a horizontal swipe gesture, I had to find the initial touch point, where **touch.phase == TouchPhase.Began**, and compare its X component to the final touch point, where **touch.phase == TouchPhase.Ended**. This is a much simpler approach than what I made in Android, which was a set of two custom gestures. But the Unity code was more complicated, and handled only the two horizontal swipe cases. I could have easily added new gestures to the Android GestureDetector, and plugged in other methods to change the state of the app. Adding additional calculations for custom gestures in Unity would have been much more difficult and the code would have grown in complexity very quickly.

Unity touch code:

foreach(Touch touch in Input.touches) {  
 if (touch.phase == TouchPhase.Began) {  
 lastX = touch.position.x;  
 touchStored = true;  
 } else if (touch.phase == TouchPhase.Ended) {  
 float xDist = lastX - touch.position.x;  
 if(xDist == 0) {  
 return;  
 }  
 if (System.Math.Abs(xDist) > 100.0f) {  
 if(xDist > 0) {  
 setPrevious();  
 } else {  
 setNext();  
 }  
 }  
 touchStored = false;  
  
 }  
 }

Java touch code:

public void onGesturePerformed(GestureOverlayView overlay, Gesture gesture) {  
 ArrayList<Prediction> predictions = customGestures.recognize(gesture);  
 for (Prediction curPred : predictions) {  
 if(curPred.score > gestureThreshold) {  
 if(curPred.name.contains("right")) {  
 goRight(overlay);  
 }  
 else if(curPred.name.contains("left")) {  
 goLeft(overlay);  
 }  
 }  
 }  
}

All in all, development in Unity with C# scripting was more cumbersome to start, less adapted to common Android use cases like touch gestures, and generally more difficult than development in native Android with Java. Unity does offer some unique advantages and flexibility, like near-instant porting across platforms. Unity was a stimulating challenge for this assignment, and I am interested to work more with it.