Projektbericht Android-Gruppe (Montag)

Implementierung eines Non-Player-Characters

Melanie Kloss, David Krammer

Allgemein

Das von uns vorbereitete Unity-Projekt ist die Datei server.unity im Ordner UnityServer\Assets unseres Projekts. Das GameObject *ServerComponents* enthält alle notwendigen Bausteine, um unser Projekt zum Laufen zu bringen. Sollte allerdings kein *observerObject* (siehe “Vorgehen” unter “Server-Client-Struktur") referenziert sein, wirft unser *ServerStringInlet* Exceptions. Der Ordner UnityServer\Assets\LSL4Unity darf nicht verschoben werden / muss genau an dieser Stelle eingebunden sein, ansonsten könnten Referenzen nicht mehr stimmen.

Das Android-Projekt ist unter liblsl-Android/AndroidStudio zu finden. Das zu startende Modul heißt ReceiveStringMarkers - das Modul SendStringMarkers sollte automatisch nicht geladen werden und kann ignoriert werden; im Dateiordner wird immer noch automatisch eine .iml-Datei erstellt, der Ordner ist aber ansonsten leer.

Sollte das Android-Projekt nicht zusammengebaut werden können, bitte hier (https://developer.android.com/ndk/downloads/older_releases) die 14b (März 2017) Version herunterladen und als NDK einfügen.

Für weitere Informationen: siehe readme.txt des Projekts

Spracherkennung

Anforderungen

Die App soll zur Laufzeit kontinuierlich Sprache erkennen und die erkannten Worte dann im Netzwerk verschicken.
Dabei wird nur ein minimales, nicht weiter spezifiziertes konfigurations-UI benötigt.
In einem zweiten Schritt soll die offline Erkennung erzwungen werden, aber weiterhin die Möglichkeit bestehen, den aufgezeichneten Text im Netzwerk zu verschicken.

Voraussetzungen

Benötigt wird das Android SDK ab Version 23.
Zusätzlich muss unabhängig von der App zuvor ein Sprachpaket installiert werden, das zur Spracherkennung im Offline- Modus genutzt werden kann. Die Installation eines Sprachpaketes kann in den Einstellungen des Gerätes vorgenommen werden.

Vorgehen

Im ersten Schritt wurde nach Möglichkeiten und Bibliotheken recherchiert, die bei der Umsetzung der kontinuierlichen Spracherkennung sinnvoll genutzt werden können. Dabei stößt man neben DroidSpeech , welche vorgeschlagen wurde, vor allem noch auf PocketShpinx (https://github.com/cmusphinx/pocketsphinx). Da mit DroidSpeech jedoch die gewünschte Funktionalität einfacher umgesetzt werden kann und es zudem genauer in der Worterkennung ist, haben wir uns dafür entschieden.

Aufgrund der geringen Anforderungen an das UI wurde nur ein Teil der Beispiel-App implementiert. Wichtig ist vor allem, dass auf dem Bildschirm angezeigt wird, welches Wort erfasst wurde, damit die Genauigkeit überprüft werden kann. Des Weiteren musste der Signalton zum Sprechen deaktiviert werden, damit das spätere Spielgeschehen nicht gestört wird.
Dieses kann mit der AudioManager Klasse umgesetzt werden, die einen Zugriff auf die Lautstärke und den Anrufmodus des Gerätes erlaubt.
https://developer.android.com/reference/android/media/AudioManager

```
AudioManager audioManager =
(AudioManager)getSystemService(Context.AUDIO_SERVICE);
audioManager.setStreamVolume(AudioManager.STREAM_MUSIC, 0,
AudioManager.FLAG_REMOVE_SOUND_AND_VIBRATE);
```

Da die Spracherkennung die erkannten Zahlen zum Teil ausgeschrieben z.B. "vier" und zum Teil als Zahl ('4') versteht wurde eine Methode zur Analyse implementiert analyseResult(String s)
Diese überprüft das erkannte Wort und gibt immer eine Zahl als String zurück, bzw. NaN wenn keine Zahl zwischen 0 und 10 erkannt wurde. Die Methode kann recht flexibel erweitert werden, je nachdem was für Werte oder Daten gebraucht werden.

Probleme bezüglich der Sprachgenauigkeit

Die Spracherkennung funktioniert je nach Gerät unterschiedlich genau. Mit einem Samsung Galaxy S3 war die Erkennungsrate sehr hoch. Das Testen mit einem Pixel 02 brachte sehr unterschiedliche Ergebnisse:
1: und, Heinz, Hans, kannst, ein
2: Roy, was, zwar, weil
3: als, Kai, heute, bei, frei, hi, treu
4: sie, wie, ja, ihr
5: kannst, und, sims, schwanz, 12, tschüss, 1, wenn's
6: weck, rex, es
7: wie, du, im, sehen
8:
9: von, heul, moin, 0, neue
10: wie, neue, x, creme, c, Tim, du

Offline -Modus

Die Spracherkennung funktioniert auch, wenn das Gerät offline ist. Jedoch konnte keine Lösung gefunden werden, um weiterhin den Text im Netzwerk zu versenden. Sobald das Gerät Online ist, wechselt die Google Spracherkennung automatisch in den Online-Modus. Nach einiger Recherche wie dieses Problem gelöst werden könnte, haben wir einige Android Firewalls ausprobiert, um nur ausgewählten Traffic durchzulassen. Getestet wurde das mit NetGuard, AFWall, Firewall ohne Root. Jedoch konnten diese nicht ausreichend angepasst werden. Auch die Anfrage an die andere Android-Gruppe und den Dozenten hat keinen Hinweis zum weiteren Vorgehen geliefert.

Server-Client-Struktur

Anforderungen / Voraussetzungen

Unsere Anforderungen waren zu Beginn des Seminars, dass wir als Android-Gruppe neben der Spracherkennung eine Verbindung zwischen dem Android-Client (Java) und einem Unity-Server (C#,

wobei bei der Programmiersprache keine Vorgabe gemacht wurde) aufbauen, die dann miteinander auf eine nicht weiter spezifizierte Art kommunizieren und Daten in einem nicht weiter spezifizierte Format austauschen. Ein bestimmtes Kommunikationsmodell zwischen Server und Client war nicht vorgegeben und auch eine etwaige Weiterverarbeitung der übertragenen Daten durch den Server war nicht spezifiziert.

Durch die spätere Definition eines Kommunikationsmodells während des Seminars kam die Anforderung der beidseitigen Kommunikation zwischen Server und Client hinzu.

Später sollte dann LSL implementiert werden, wobei nicht festgelegt war, welche Teile der Kommunikation dies betreffen sollte.

Vorgehen

Zuerst wurde eine rudimentäre Server-Client-Verbindung[1] auf reiner Java-Basis aufgesetzt (im package *dataTransfer*), sprich sowohl Server als auch Client liefen auf demselben Handy, um zu überprüfen, ob eine TCP-Verbindung überhaupt funktionierte. Der damalige Client Code diente dann als Grundlage für eine Spezifizierung und Anpassung an die Ansprüche. Dabei mussten sowohl Server als auch Client in jeweils einen eigenen AsyncTask ausgelagert werden, da die Netzwerkaktivität nicht im Main-thread laufen darf.

Nachdem diese Tests abgeschlossen waren, wurde den Java-Server gelöscht und einen TCP-Server in C# implementiert, der erneut auf einer Vorlage[2] basierte, die angepasst wurde. Die Übertragung läuft auf Port 4444.

Da zu dieser Zeit viel getestet werden musste und es umständlich war, hardcoded IPs immer wieder zu ändern, wurde ein in den Anforderungen nicht enthaltenes UI implementiert, um die Arbeit zu erleichtern. Dieses UI erlaubt die direkte Eingabe einer IP, zu der sich der Client verbinden soll, das manuelle Starten und Schließen der TCP-Verbindung über Buttons und die stetige Anzeige des aktuellen Client-Status in einem TextView.

Da bei der Kommunikation von Java- auf C#-Code gewechselt wird, ist das Senden und Empfangen so implementiert, dass vom Client (Java) ein PrintWriter als Output dient, aber Byte-Code über einen InputStream empfangen wird. Der Server sendet und empfängt Byte-Code über NetworkStreams. Obwohl bereits hier beschrieben, wurden Teile der Kommunikation erst später implementiert und waren zu diesem Zeitpunkt noch nicht geplant.

Als die generelle TCP-Verbindung vom Client in Richtung Server stand, wurden genaue Anforderungen an Client und Server definiert.

Dabei wurde zuerst ein Kommunikationskonzept ausgebarbeitet, wie der Server und der Client miteinander kommunizieren sollen. Festlegt wurde schlussendlich in Absprache mit dem Seminarleiter, dass der Server eine “listen”-Anfrage an den Client schickt und dieser dann die Spracherkennung startet. Der String, der von der Spracherkennung zurückgeliefert wird, wird auf enthaltene Zahlen geprüft und sollte keine Zahl erkannt werden, liefert die Prüfung einen festgelegten Errorcode (in diesem Fall “NaN”) zurück. Egal was die Prüfung zurückliefert - ob Zahl

[1] Quelle: https://www.myandroidsolutions.com/2012/07/20/android-tcp-connection-tutorial/#.WuMW_JdCSUl
[2] Quelle: https://gist.github.com/danielbierwirth/0636650b005834204cb19ef5ae6ccedb

oder Errorcode – wird vom Client an den Server zurückgesendet. Das weitere Verfahren mit der Zahl auf dem Server war nicht einschätzbar, da im Seminar keine Schnittstellen ö.Ä. zwischen den Projektteilen definiert wurden. Daher wurde die Zahl einfach in einer ArrayList zwischengespeichert. Als kleine Anmerkung zu dem Begriff "Zahl", der hier verwendet wurde: Im Programmcode ist diese Zahl nicht als Zahl (int, float, …) repräsentiert, sondern als String, der eine arabische Zahl enthält (also "1" und nicht "eins"), um auch den Errorcode problemlos senden zu können. Durch die Repräsentation als arabische Zahl kann der String aber problemlos auf Integers geparst werden.

Da das Kommunikationskonzept eine "listen"-Anfrage des Servers vorsah, musst nun eine Verbindung für beidseitige Kommunikation implementiert werden (technische Beschreibung: siehe oben). Zunächst war clientseitig nur das Empfangen von sechsstelligen Strings ("listen") möglich, wurde später im Projekt dann aber auf eine dynamische Länge umgestellt.

Außerdem konnte nun nicht mehr garantieret werden, in welcher Reihenfolge welcher Teilnehmer sendet und empfängt. Es konnte also dazu kommen, dass der Server an den Client gesendet hatte, der Client aus irgendeinem Grund nicht gesendet hatte (oder der Server gerade nicht empfangen konnte) und dann beide Parteien auf den jeweils anderen gewartet haben und im Empfangen-Status feststeckten. Daher wurde für Server und Client dynamisches Senden und Empfangen implementiert, wodurch ein Feststecken in einem Status nicht mehr möglich ist.

Als die Verbindung stand, wurde sich um die internen Abläufe im Client gemäß des ausgearbeiteten Kommunikationsplans gekümmert. Dabei musste die Kommunikation zwischen den einzelnen Threads gelöst werden, da die Spracherkennung im Main-thread und der TCP Client im AsyncTask läuft. Der Client-thread löst einen Request im Main-thread aus, der dann die Spracherkennung startet. Sobald DroidSpeech ein Ergebnis liefert, wird dieses Ergebnis durch eine Analyse geschickt (siehe oben), und das Resultat jener an den Client-thread direkt zurückgeschickt.

Auf das Feedback des Seminarleiters hin, wurde die Struktur so geändert, dass das Resultat nicht direkt an den Client-thread zurückgeschickt, sondern erst in einer Queue abgelegt wird, aus der sich der Client-thread dann das Resultat holt und verschickt. Diese Queue läuft ebenfalls in einem eigenen thread. Außerdem gibt es einige Dinge zu beachten, die eventuell für die Implementierung der Verarbeitungskette in Unity wichtig sind: Da die Queue als Arraylist implementiert ist, ist es (programmtechnisch) möglich, dass mehrere Strings direkt hintereinander gesendet werden, auch wenn Unity nur einen String erwartet. Rein logisch wird aber pro "listen"-Request nur ein Ergebnisstring zur Queue geschickt. Es wurde sich trotzdem für eine ArrayList anstatt einer einzelnen String Variable entschieden, um im Ernstfall lieber mehr Daten zu speichern als sie zu überschreiben und damit zu verlieren.

LabStreamingLayer (LSL) wurde auf folgende Weise implementiert und integriert: Die TCP-Verbindung vom Server zum Client bleibt bestehen, da diese (gemäß des Kommunikationsmodells) nur für "listen"-Requests benötigt wird. Die Übertragung vom Client zum Server wird dann aber über LSL geregelt, wobei dadurch nicht nur die Daten an sich, sondern auch exakte Timestamps geschickt werden können. Das stimmt allerdings nicht ganz, da die Aufforderung, die Verbindung zu schließen (vom Client ausgehend), weiterhin über TCP läuft.

Clientseitig ist das "standard" LSL integriert worden, das über ein Outlet einen Broadcast über das Netzwerk schickt (identifizierbar durch Namen und Typ). In Unity haben wir LSL4Unity benutzt, das

einfach zu integrieren war, durchgehend auf LSL Streams im Netzwerk lauscht und eine praktische Schnittstelle für empfangene Daten bereitstellt. Wir mussten nur ein eigenes Script erstellen, das von einem bestimmten Inlet erbt und vorgegebene Methoden überschreibt.

Um die spätere Weiterverarbeitung so einfach wie möglich zu machen, wurde ein Interface *OnNumberRequested* geschrieben, das von einem Script implementiert werden muss. Das korrespondierende GameObject zu diesem Script muss als *observerObject* des von dieser Gruppe geschriebenen Inlet-Scripts *ServerStringInlet* eingetragen werden. Die übertragenen Daten, die von unserem Script abgefangen werden, werden dann an das Script, das jenes Interface implementiert, geschickt. Somit muss zur Weiterverwendung am Code dieser Gruppe nichts / nicht viel geändert werden.

Es sei außerdem gesagt, dass bei jedem Schritt / Feature sehr viele Tests gemacht wurden, die hier nicht immer extra erwähnt sind.

Limitationen

Es ist zu bedenken, dass die Spracherkennung eine bestimmte Zeit braucht, um Dinge zu erkennen. Daher ist es zwar technisch möglich zwei “listen”-Requests kurz nacheinander zu schicken, aber die Spracherkennung wird nicht ein zweites Mal gestartet, wenn sie schon läuft. Daher ist es für Unity empfehlenswert, immer auf eine Antwort des Clients zu warten (oder eine längere Zeit verstreichen zu lassen), bevor eine neue Anfrage geschickt wird. Ansonsten wird diese einfach im Sand verlaufen.

In den Tests hat sich herausgestellt, dass das Empfangen von LSL-Streams vor allem zu Beginn der Übertragung (ca. die ersten 30 Sekunden) nicht gut funktioniert. Es wird empfohlen, vor der ersten ernsthaften Anfrage an den Client mindestens diese 30 Sekunden zu warten oder noch besser dort schon ein paar Testübertragungen zu machen.

Da die TCP-Verbindung zum Server weiterhin benutzt wird (siehe oben), kann es sein, dass die Firewall des PC, auf dem Unity läuft, deaktiviert oder entsprechend konfiguriert werden muss, da diese eventuell eingehende Portanfragen blockiert.

Probleme

Es gab erhebliche Probleme, eine Verbindung zwischen TCP Client und TCP Server aufzubauen (mögliche Lösung: siehe Limitationen). Es ist aber nicht garantiert, dass das Deaktivieren der Firewall überall im Uninetzwerk die Verbindungsprobleme behebt. Am besten haben die Tests immer im Rechenzentrum funktioniert.

Die Analyse von Strings war kurzzeitig auch ein Problem. Dabei hat der Client einen String an den Server geschickt, welcher bei der Prüfung jenes Strings auf Übereinstimmung mit einem festgelegten exit-Code, der die Verbindung zwischen Server und Client beenden sollte, als Ergebnis false zurücklieferte, obwohl genau jener exit-Code auch gesendet wurde. Der Fehler lag darin, dass am Ende des Strings ein “unsichtbarer” Carriage Return saß.

Wie bereits oben beschrieben, dauerte das Debuggen und Testen immer sehr lange, da anfangs die IP-Adresse noch hardcoded vorlag und bei jedem neuen Verbindungsaufbau vom Client zum Server,

die App über Android Studio neu compiliert und auf dem Handy / Emulator installiert werden musst. Außerdem hat sich die App immer direkt beim Start zum Server verbunden – ein eigener Zeitpunkt war nicht wählbar. Diese Probleme wurden alle durch das neue UI behoben.

Strings konnten zeitweise vom Client nur in bestimmter Länge empfangen werden, was später aber behoben wurde.

LSL hat allgemein große Probleme gemacht.

Auf Android war die Integration sehr schwierig, da LSL (bzw. die master-Datei, die man sich herunterladen kann) viele Abhängigkeiten hat / Dateien an der richtigen Stelle braucht, um zu funktionieren. Daher ist es fehlgeschlagen, LSL in das Android-Projekt zu integrieren und das Android-Projekt in LSL zu integrieren, wodurch wir gezwungen waren, den Projektcode in eines der LSL Beispiele hineinzukopieren und die .gradle-Dateien für unsere Ansprüche zu erweitern. Deswegen stimmen auch z.B. Ordnernamen mit ihrem eigentlichen Inhalt nicht überein. Es sollte aber unter keinen Umständen riskiert werden, dass dort irgendeine Referenz auf irgendeinen Ordner besteht, die durch Umbenennung zerschossen wird. Ansonsten kann es eben sein, dass dieses gesamte Konstrukt irgendwann zusammenbricht. Sollte das alles erfüllt sein, funktioniert LSL auf Android – vorausgesetzt alle anderen benötigten Dateien, die LSL braucht, sind heruntergeladen und an der richtigen Stelle eingebunden.

Auf Unity-Seite wurde zuerst versucht, LSL “normal” einzubinden, aber wo auch immer die “liblsl64.dll” eingebunden wurde, der LSL-Code hat die Datei nicht gefunden, wodurch auf LSL4Unity zurückgegriffen wurde. Dieses Plugin brauchte zum Glück nur einige Anpassungen, um zu funktionieren.

Der Android-Code stand, der Unity-Code stand, aber es konnte keine Verbindungen hergestellt werden. Es bestand reger E-Mail-Kontakt mit René Maget, der einige Vorschläge hatte, wie das Problem zu lösen bzw. überhaupt zu evaluieren sei. Durch die Mouse-App (die LSL-Streams sendet) und den LabRecorder von LSL konnten festgestellt werden, dass LSL4Unity Streams empfangen kann, die Android-App aber nicht sendete, was darin half, das Problem zu spezifizieren und sicher zu sein, welche der beiden Seiten nicht funktionierte. Da dem Bearbeiter der Aufgabe nur schwer ein Android-Gerät zur Verfügung stand, wurde die ganze Zeit über im Emulator getestet und daher den Fehler in Windows-Einstellungen gesucht, sprich Netzwerkadapter gelöscht, Firewalls überprüft und ausgeschaltet, etc. Es wurde sogar Linux aufgesetzt und dort unter einigem Aufwand Android Studio und die App zum Laufen gebracht – alles ohne Ergebnisse.

Die Lösung dafür war im Endeffekt aber ziemlich einfach, denn es scheint so, als wäre der Emulator das Problem. Die App wurde einmal auf einem richtigen Android Handy laufen gelassen und ausgehende Streams sind sofort gesendet worden.

Als allgemeines Problem um LSL und LSL4Unity muss noch erwähnt werden, dass Struktur und Implementierung teilweise schwer zu durchschauen waren und die Dokumentation absolut unzureichend war. Daher wurde sehr viel Zeit damit verbracht, das Programm an sich zu verstehen, während die Integration immer wieder in Sackgassen geführt hat.