SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2886

ANIMACIJA LJUDSKOG LICA

Maja Radočaj

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2886

ANIMACIJA LJUDSKOG LICA

Maja Radočaj

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2886

Pristupnica:	**Maja Radočaj (0036505371)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Animacija ljudskog lica**

Opis zadatka:

Proučiti osnove u izradi 3D modela ljudskog lica. Proučiti mehanizme upravljanja izrazima lica. Razraditi animirane pokrete i promjene u izrazima te upravljačke mehanizme za definiranje različitih izraza lica. Implementirati animaciju i demonstrirati ostvarivanje upravljanja izrazima lica. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski alat Unreal Engine. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 27. lipnja 2022.

Sadržaj

Uvod

Interes za računalnu animaciju lica trodimenzionalnih likova i ljudskih lica drastično je porastao u zadnjih desetak godina. Međutim, animacija ljudskog lica koristeći računala nije novi zadatak: prve računalno generirane slike trodimenzionalnih lica generirao je Frederic Parke u sklopu Ivan Sutherlandovog kolegija računalne grafike na Sveučilištu u Utah davne 1971. godine [1].

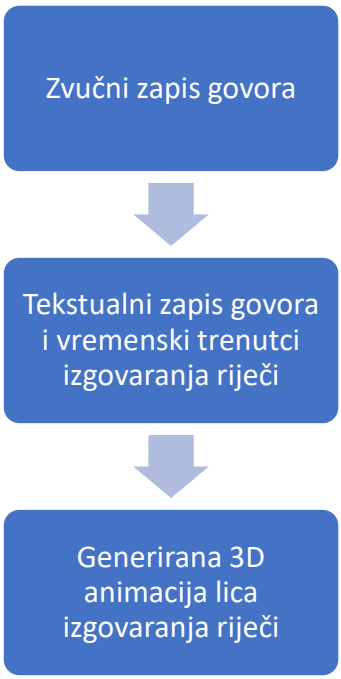
U današnje vrijeme, računalna animacija lica najviše je primijenjena u samoj industriji animacije. Osim u animacijskim studijima, područje u kojem je animacija trodimenzionalnih likova dominantno zastupljena je industrija igara. Naime, industrija igara doživjela je ubrzan razvoj omogućen naprecima u tehnologiji. Brzi procesori i jake grafičke kartice otvorile su vrata sofisticiranom, visoko kvalitetnom renderiranju u stvarnom vremenu koje do tad nije bilo moguće. Računalno generirana ljudska lica nalaze primjenu čak i u medicini. Liječnici računalne modele lica mogu koristiti u svrhu planiranja i simuliranja operacija na određenom dijelu lica.

Ljudsko lice zanimljivo je i izazovno animirati jer nam je vrlo poznato. Upravo lice je primarni dio tijela po kojem prepoznajemo ljude te smo zbog toga vrlo dobri u detektiranju najmanjih promjena u njegovom izgledu. Dapače, toliko smo dobri u prepoznavanju lica da simulacija koja nije sasvim realistična u ljudima izaziva nelagodan osjećaj poznat kao tzv. *uncanny valley* (engl. neprirodna dolina) [2]. Upravo zbog toga je pri animaciji trodimenzionalnih likova najveći fokus na animaciji lica.

U okviru ovog rada implementiran je sustav koji iz zvuka ljudskog govora generira realističnu računalnu animaciju trodimenzionalnog lica koji izgovara riječi sa zvučnog zapisa. Svrha ovakvog sustava je omogućiti jednostavno generiranje animacija govora koje mogu biti korištene u npr. animacijskim sekvencama u videoigrama. U nastavku slijedi detaljan opis implementacije sustava, korištenih tehnologija i rasprava o mogućim budućim nadogradnjama.

1. Opis sustava i korištenih tehnologija

Cilj je implementirati sustav koji će preuzeti zvučni zapis ljudskog govora (npr. MP3 kodirani zvučni zapis) i na neki način ga prevesti u tekstualni zapis sa svim bitnim informacijama (koje riječi su izgovorene i u kojim trenutcima). Koristeći taj tekstualni zapis treba generirati animaciju lica koje će izgovarati navedene riječi. Za sve ovo potreban je i odgovarajući 3D model ljudskoga lica na koje će se primijeniti animacija, ali i animacije pojedinih riječi koje će biti izgovarane u određenim trenutcima. Pojednostavljena skica ovakvog sustava prikazana je na Slici 1.1.



Slika 1.1: Skica sustava za generiranje animacije govora

Iako je idejni opis prilično jasan i jednostavan, tehnologije i modele koje u konačnici čine ovakav sustav treba pažljivo odabrati. Bez dobrog modela za prepoznavanje govora i pretvorbu u tekst nije moguće ostvariti slični sustav za generiranje animacije. Prepoznavanje govora je kompleksan i popularan problem te danas postoje razni algoritmi varirajuće kvalitete rezultata s podrškom za različite jezike.

Isto tako, za konačno generiranje animacije potrebno je iskoristiti dovoljno kompleksan 3D model lica koji omogućuje kontrolu najmanjih mišića lica koji su potrebni za generiranje realistične animacije. Za korištenje i renderiranje takvih modela potrebno je dovoljno jak uređaj.

Za potrebe ovog rada odabrane su sljedeće tehnologije:

- Google Cloud Speech-to-Text API za prepoznavanje govora
- Unreal Engine 5.0.1, Visual Studio 2019 i C++ za generiranje animacije
- Visoko realistični 3D model MetaHuman

U nastavku slijedi kratki pregled odabranih tehnologija i modela za ostvarenje sustava animacije lica.

1.1. Speech-to-Text API

Za prepoznavanje i pretvorbu teksta u govor odabran je Googleov Speech-to-Text API [3]. Kao što je iz naziva jasno, Google je svojim korisnicima 2018. godine prvi puta ponudio sofisticirano aplikacijsko sučelje za pretvorbu govora u tekst. Sučelje nudi pretvorbu govora s visokom razinom preciznosti omogućenu naprednim algoritmima dubokog učenja za automatsko prepoznavanje govora, odnosno ASR (engl. *automatic speech recognition*). Osim prepoznavanja govora, korisniku je ponuđena i mogućnost prilagođavanja i eksperimentiranja s modelima korištenima za prepoznavanje. Nudi se odabir treniranih modela za različite scenarije i domensko specifične kvalitete zvučnih zapisa poput npr. telefonskih poziva ili video transkripcija.

Speech-to-Text podržava tri glavne metode za izvođenje prepoznavanje govora, a one su:

- Sinkrono prepoznavanje – šalje se zvučni zapis na Speech-to-Text API (pomoću REST ili gRPC), vrši se prepoznavanje govora i rezultat se vraća kad se čitavi zapis obradi; duljina zvučnog zapisa u ovom slučaju je ograničena na jednu minutu ili manje
- Asinkrono prepoznavanje – šalje se zvučni zapis na Speech-to-Text API (pomoću REST ili gRPC) i inicira se posebna operacija dugog trajanja; korisnik može periodički provjeravati rezultat pretvorbe, a duljina zvučnog zapisa u ovom slučaju može biti do 480 minuta
- Tekuće prepoznavanje – engl. streaming recognition. Izvodi se prepoznavanje govora na zvučnom zapisu proslijeđenom preko gRPC dvosmjerne veze; služi za prepoznavanje govora u stvarnom vremenu

U sklopu ovog projekta korištena je prva metoda prepoznavanja govora, odnosno sinkrono prepoznavanje pomoću REST poziva. Sinkrono prepoznavanje bilo je dobro rješenje zbog dovoljno kratkog trajanja zvučnih zapisa korištenih u svrhu demonstracije animacije lica.

Pri prepoznavanju govora, Speech-to-Text trenutno nudi izbor od osam različitih treniranih modela strojnog učenja. Jedan od njih je i tzv. medicinsko diktiranje (engl. *medical dictation*) koji je posebno treniran da prepoznaje govor medicinski stručne osobe. Postoji čak i tzv. medicinski razgovor (engl. *medical conversation*) koji je treniran da prepoznaje razgovor između doktora i pacijenta. Pri implementaciji sustava za animaciju korišten je pretpostavljeni model (engl. *default*).

Kod korištenja sinkronog prepoznavanja s REST pozivima moguće je definirati niz parametara kako bi se prilagodila pretvorba govora u tekst. O kojim parametrima se točno radi, kako oni utječu na proces prepoznavanja te format odgovora Speech-to-Text API-ja bit će detaljno objašnjen nešto kasnije u opisu implementacije.

1.2. Unreal Engine

Unreal Engine jedan je od najnaprednijih i najpopularnijih alata otvorenog koda za izradu 3D računalnih igara [4]. Razvio ga je Epic Games, a prvi put je predstavljen 1998. godine. Napisan je u programskom jeziku C++ i podržava brojne desktop i mobilne platforme, konzole i platforme za virtualnu stvarnost. Samo neke od poznatih igrica napravljenih u Unreal Engine su Fortnite iz 2017. godine, Street Fighter V iz 2016. godine, Shrek 2 iz 2004. godine, itd [5].

Najnovija stabilna verzija Unreal Enginea, verzija 5, objavljena je u travnju 2022. godine. U najnovijoj verziji nalaze se brojna poboljšanja koja su implementirali i članovi Unreal Engine zajednice. Generiranje i pregled animacije lica implementirana je upravo u najnovijoj verziji Unreal Enginea.

Za korištenje Unreal Engine alata potrebni su određeni hardverski preduvjeti. Preduvjeti za rad u najnovijoj UE verziji prikazani su na Slici 1.2.

Operating System	Windows 10 64-bit
Processor	Quad-core Intel or AMD, 2.5 GHz or faster
Memory	8 GB RAM
Graphics Card	DirectX 11 or 12 compatible graphics card
RHI Version	▪ **DirectX 11:** Latest drivers ▪ **DirectX 12:** Latest drivers ▪ **Vulkan:** AMD (21.11.3+) and NVIDIA (496.76+)

Slika 1.2: Hardverski preduvjeti za rad u Unreal Engine 5

Pošto je rad u Unreal Engine vrlo računalno zahtjevan, ovo su minimalni preduvjeti za bilo kakav značajni posao u alatu. Sve druge preduvjete i upute za rad moguće je naći u službenoj dokumentaciji [6].

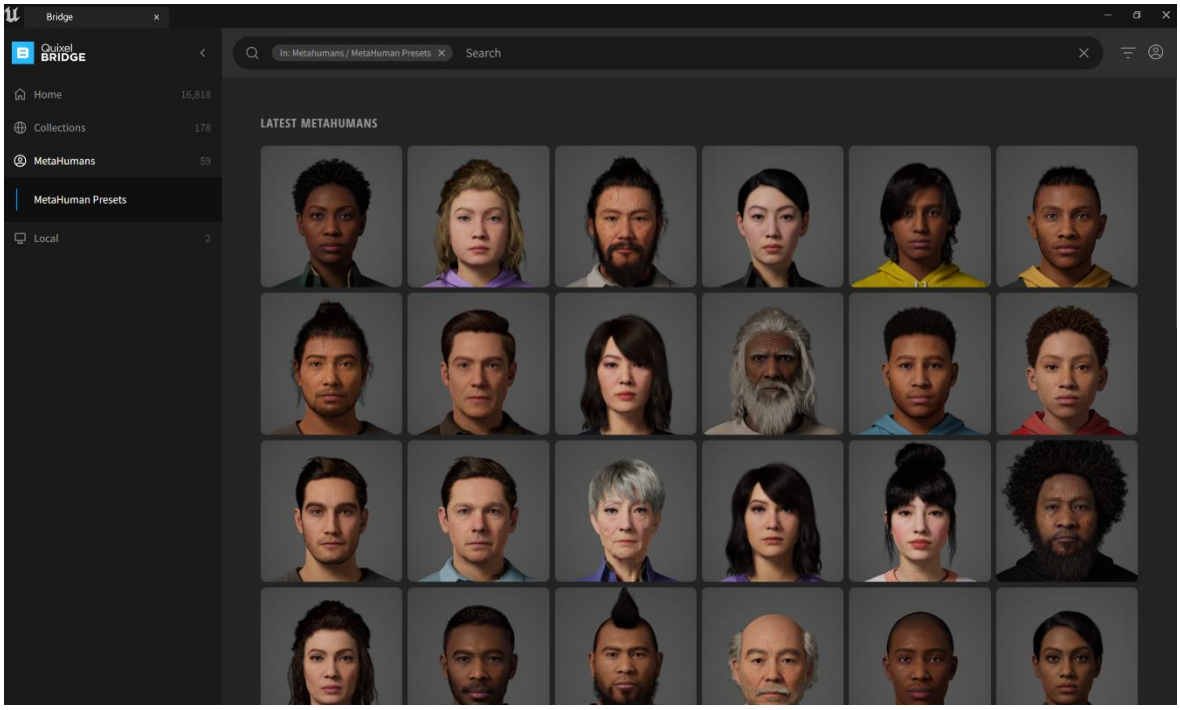
Nadogradnja logike likova u UE implementirana je u jeziku C++. Za rad sa C++ uz Unreal Engine potrebno je instalirati i IDE Microsoft Visual Studio 2019.

1.3. MetaHuman 3D model

U Unreal Engine verziju 5 ugrađen je i dodatak Quixel Bridge, alat koji nudi brojne resurse, kolekcije i 3D modele potrebne za izgradnju projekta u Unreal Engineu [7]. Među brojnim dostupnim detaljnim modelima nalaze se i tzv. MetaHuman modeli – detaljni i vjerni 3D digitalni likovi opremljeni sa svim potrebnim resursima potrebnima kako bi se lik koristio u projektu [8]. Kao što je već spomenuto u kontekstu Unreal Enginea, za najbolje iskustvo korištenja MetaHuman modela potrebna je kvalitetna grafička kartica kako bi se bez smetnja mogli pregledati detalji poput praćenja zraka, iscrtavanje kose i sl.

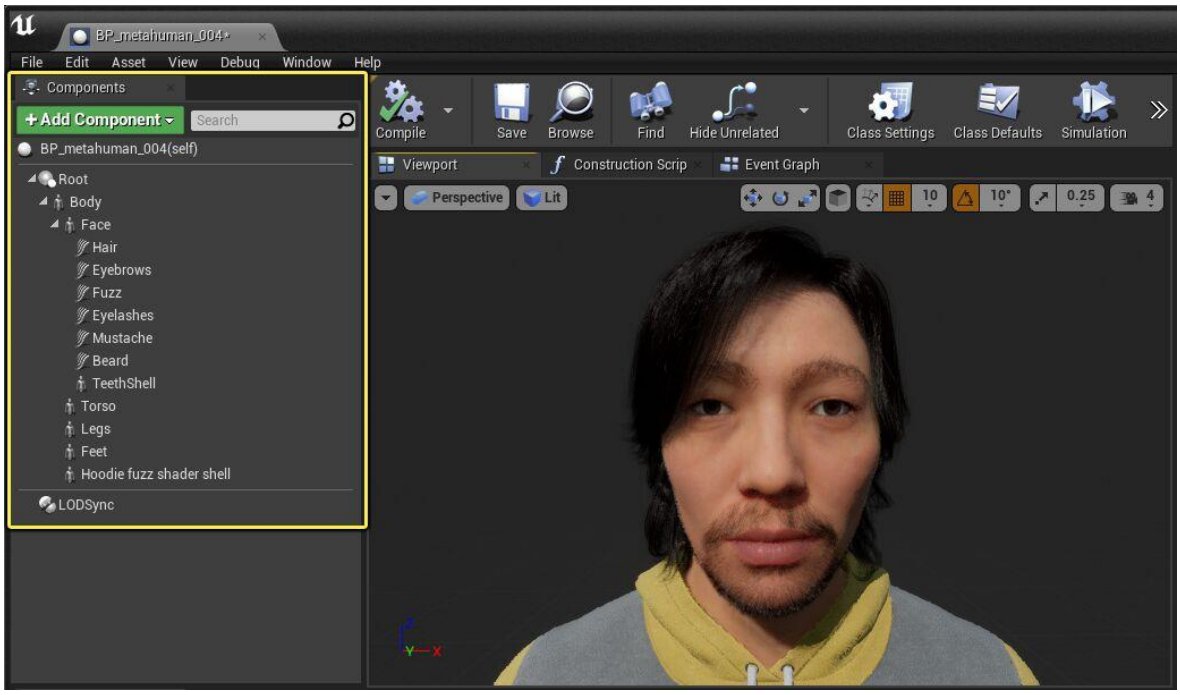
MetaHuman model moguće je preuzeti direktno iz Quixel Bridge i učitati ga u projekt. Osim što je moguće preuzeti gotove modele, kroz tzv. MetaHuman Creator moguće je na

jednostavan način kreirati vlastiti MetaHuman digitalni lik (u ovom radu se to nije dalje razmatralo). Na Slici 1.3 prikazan je skup od 59 unaprijed dostupnih besplatnih modela koje je moguće preuzeti i koristiti u vlastitim projektima.



Slika 1.3: MetaHuman modeli dostupni u Quixel Bridge

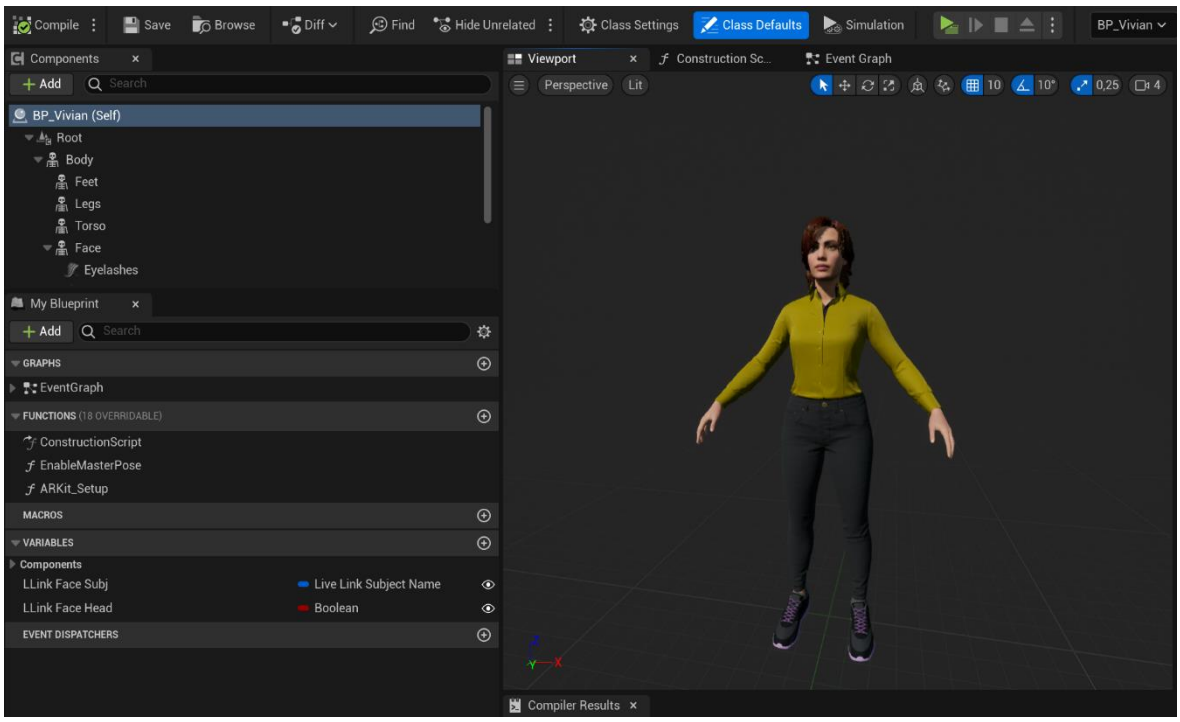
Svaki MetaHuman model sačinjen je od brojnih komponenti koje sačinjavaju njegovo tijelo, lice, kosu i odjeću. Primjer MetaHuman lika i njegovih komponenti prikazan je na Slici 1.4.



Slika 1.4: Prikaz komponenti koje sačinjavaju MetaHuman lika

Komponente MetaHuman lika prikazane na slici (konkretno, komponente *Face*, *Torso*, *Legs*, *Feet*) su komponente tipa kostur (engl. *skeletal mesh*). To su komponente koje pokreću sve animacije i pokrete tijela lika. U ovom projektu najviše se koristio kostur lica čijom se manipulacijom ostvaruje animacija lica.

Kao model nad kojim se izvode animacije odabran je MetaHuman zvan Vivian prikazan na Slici 1.5. Opis uvoza modela u Unreal Engine projekt, kao i način manipulaicje kostima lica za potrebe animacije bit će opisan u sljedećim poglavljima.



Slika 1.5: Prikaz BP_Vivian u Unreal Engine projektu

2. Implementacija

Integracijom tehnologija navedenih i opisanih u prethodnom poglavlju ostvaren je sustav za generiranje 3D animacije govora. Konačni rezultat je Unreal Engine projekt koji pri pokretanju pokreće C++ kod za parsiranje datoteke koja sadrži prijepis govora koji se nalazi u zvučnoj datoteci u resursima projekta. Datoteka s prijepisom govora rezultat je koji se dobije prepoznavanjem govora pomoću Speech-to-Text API-ja. U nastavku slijedi detaljan opis korištenja Speech-to-Text API-ja i stvaranje Unreal Engine projekta i ostalih resursa za potrebe implementacije animacije.

2.1. Prepoznavanje govora

Prvi korak u generiranju animacije govora iz zvučnog zapisa je dobivanje podataka o izgovorenim riječima iz zvučnog zapisa. Kao što je već spomenuto, za potrebe prepoznavanja govora korišteni su sinkroni pozivi prema Google Speech-to-Text API-ju.

Svaki sinkroni zahtjev prema API-ju mora sadržavati konfiguraciju prepoznavanja govora i zvučni zapis. Konkretan primjer zahtjeva u `request.json` datoteci korištenog u ovom projektu prikazan je kodom Kôd 2.1.

```
{
  "config": {
      "languageCode": "en-US",
      "encoding": "MP3",
      "sampleRateHertz": "48000",
      "enableWordTimeOffsets": true
  },
  "audio": {
      "uri":
"gs://speech_audio_0506/SpeechAudio/Hello_My_Name_Is.mp3"
  }
}
```

Kôd 2.1: Zahtjev za prepoznavanjem govora prema Speech-to-Text API-ju

Konfiguracija prepoznavanja govora nalazi se u `config` polju, a u `audio` polju je zvučni zapis koji treba obraditi.

Polje konfiguracije sadrži sljedeća polja:

- `languageCode` – obavezno polje; sadrži jezik i regiju koji će se primijeniti pri prepoznavanju govora. Pošto su zvučni zapisi korišteni u ovom radu na engleskom jeziku, odabrana je vrijednost `en-US` (engleski jezik, američki izgovor).
- `encoding` – obavezno polje; sadrži informaciju o tome na koji način je zvučni zapis kodiran. Postoji 9 mogućih kodiranja koje Speech-to-Text podržava, a u ovom primjeru korišteno je `MP3` kodiranje [9]. Međutim, `MP3` kodiranje dostupno je samo u beta inačici Speech-to-Text API-ja. Zbog toga se u kasnijim pozivima može vidjeti da se zahtjevi šalju prema `v1p1beta1`.
- `sampleRateHertz` – obavezno polje; sadrži stopu uzorkovanja proslijeđenog zvučnog zapisa. Stopa uzorkovanja ovisi o zvučnom zapisu i mora mu odgovarati. Speech-to-Text podržava stope od 8000 Hz do 48000 Hz.
- `enableWordTimeOffsets` – neobavezno polje; ako je ova zastavica postavljena na `true`, uz prepoznatu riječ u rezultatu prepoznavanja dobiva se i informacija o vremenskom trenutku početka prepoznate riječi (`startTime`) i vremenskom trenutku kraja prepoznate riječi (`endTime`). Ova zastavica je vrlo bitna jer se prema trenutku početka i završetka riječi kasnije računa brzina reprodukcije animacije izgovora riječi. Pretpostavljena vrijednost ove zastavice je `false`.

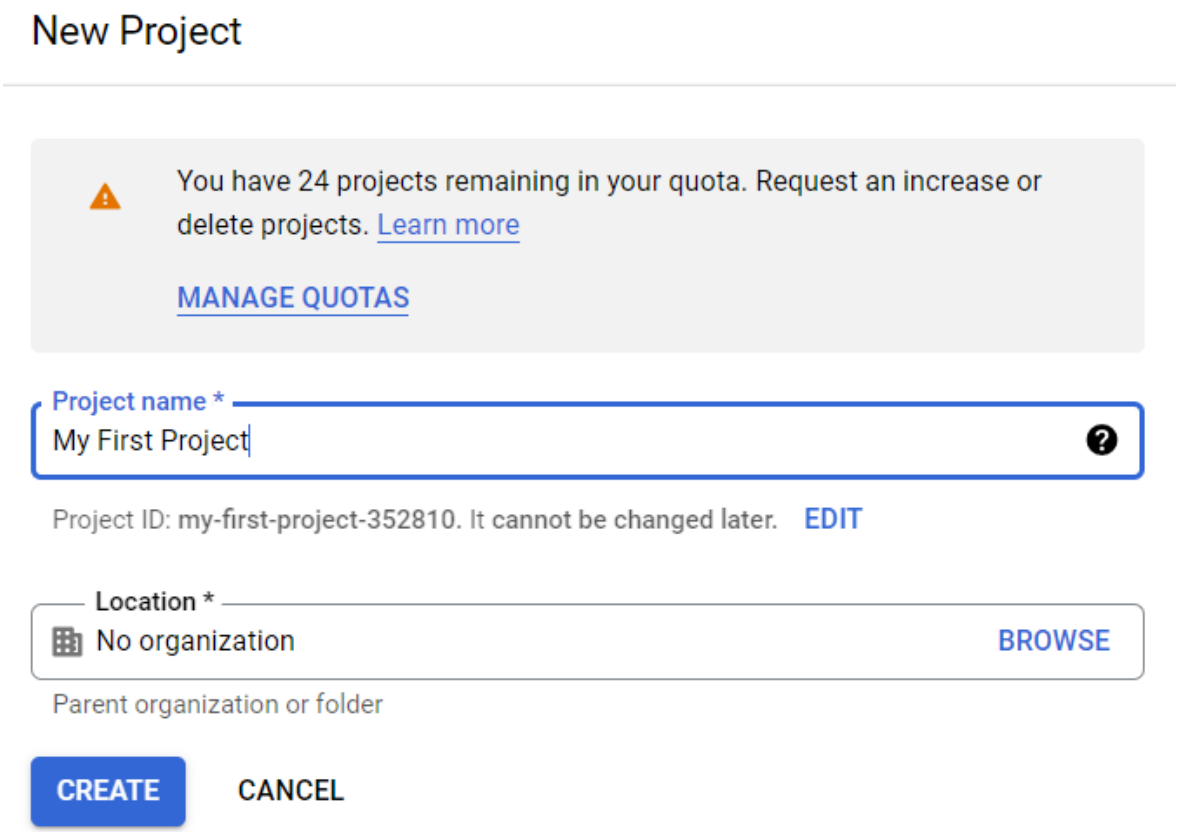
Postoje još brojne druge opcije koje je moguće konfigurirati pri prepoznavanju govora (npr. opcija filtriranja prostih riječi, mogućnost automatskog dodavanja interpunkcijskih znakova, itd.), no one nisu korištene u ovom radu te se neće dalje razmatrati.

Zvuk je u zahtjevu dan kroz polje audio. Polje audio mora sadržavati jedno od dva polja:

- `content` – sadrži zvučni zapis direktno priložen u zahtjevu za prepoznavanje govora
- `uri` – sadrži URI gdje je pohranjen zvučni zapis. Trenutno to mora biti URI prema Google Cloud Storage, Google rješenju za pohranu podataka [10].

Za potrebe ovog rada korišteno je prosljeđivanje zvučnog zapisa pomoću URI-ja. Kako bi se mogao proslijediti URI, korišteni zvučni zapisi moraju biti pohranjeni na Google Cloud Storage. Stoga, potrebno je kreirati projekt na Google Cloud platformi kako bi bilo moguće pristupiti Google Cloud Storage i Speech-to-Text API. Iako se korištenje Google Cloud platforme inače plaća, Google je svojim korisnicima omogućio besplatno probno razdoblje od 90 dana ili do dostizanje limita od 300 dolara potrošnje resursa.

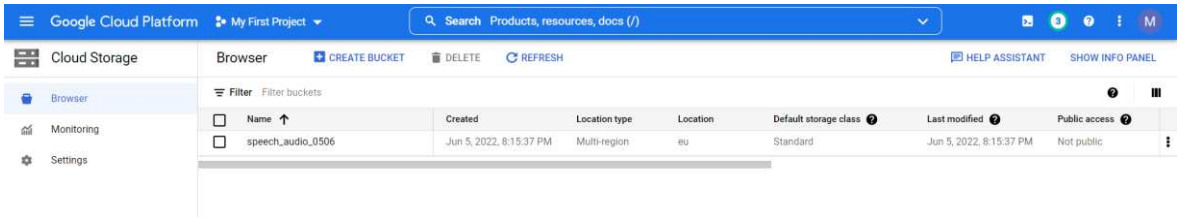
Za korištenje potrebnih API-ja kreiran je projekt naziva *My First Project* prikazan na slici Slika 2.1. Upute za kreiranje projekta moguće je naći u službenoj dokumentaciji [11].



Slika 2.1: Stvaranje projekta na Google Cloud platformi

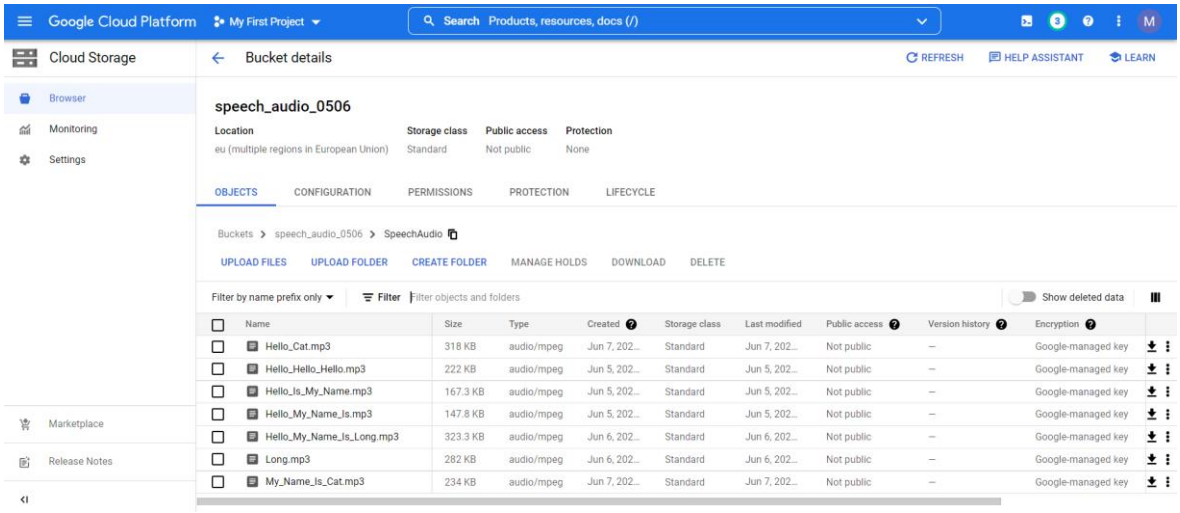
Nakon što je projekt uspješno stvoren, potrebno je kroz Google Cloud Storage konzolu pohraniti potrebne zvučne zapise. Upute za pohranu datoteka također je moguće naći u službenoj dokumentaciji [12].

Unutar kreiranog *My First Project* projekta, potrebno je stvoriti novi tzv. *bucket* (engl. *bucket* – kanta) – osnovna jedinica ili kontejner unutar kojeg se pohranjuju datoteke. Sve što je pohranjeno u Google Cloud Storage mora biti sadržano unutar jednog ili više *bucketa*. Za pohranu zvučnih zapisa stvoren je novi *bucket* `speech_audio_0506`. Na Slici 2.2 prikazan je pregled postojećih bucketa u projektu *My First Project*.



Slika 2.2: Pregled bucketa kroz Cloud Storage konzolu

Unutar *bucketa* `speech_audio_0506` stvoren je novi direktorij `SpeechAudio`. U njemu je pohranjeno ukupno sedam zvučnih zapisa u MP3 formatu. Na Slici 2.3 prikazana je hijerarhija direktorija i pohranjene zvučne datoteke.



Slika 2.3: Pregled pohranjenih zvučnih zapisa

Dakle, sada kad znamo gdje je točno pohranjen zvučni zapis, možemo tu lokaciju proslijediti u URI polju zahtjeva za prepoznavanje govora (kao što je već opisano). U gornjem primjeru zahtjeva obrađuje se zvučni zapis `Hello_My_Name_Is.mp3` koji se nalazi u *bucketu* `speech_audio_0506` i direktoriju `SpeechAudio` pa se zato u URI polju prosljeđuje:

```
"uri": "gs://speech_audio_0506/SpeechAudio/Hello_My_Name_Is.mp3"
```

Konačno, potrebno je na neki način pozvati API za prepoznavanje govora. Za dobivanje obrađenog zvučnog zapisa koristio se REST odnosno HTTP zahtjev koristeći alat curl. Tijelo zahtjeva prikazano u kodu Kôd 2.1 pohranjeno je u datoteci `request.json` koja se prosljeđuje u curl naredbi. Pri tome je naredba za poziv API-ja prikazana s kodom Kôd 2.2.

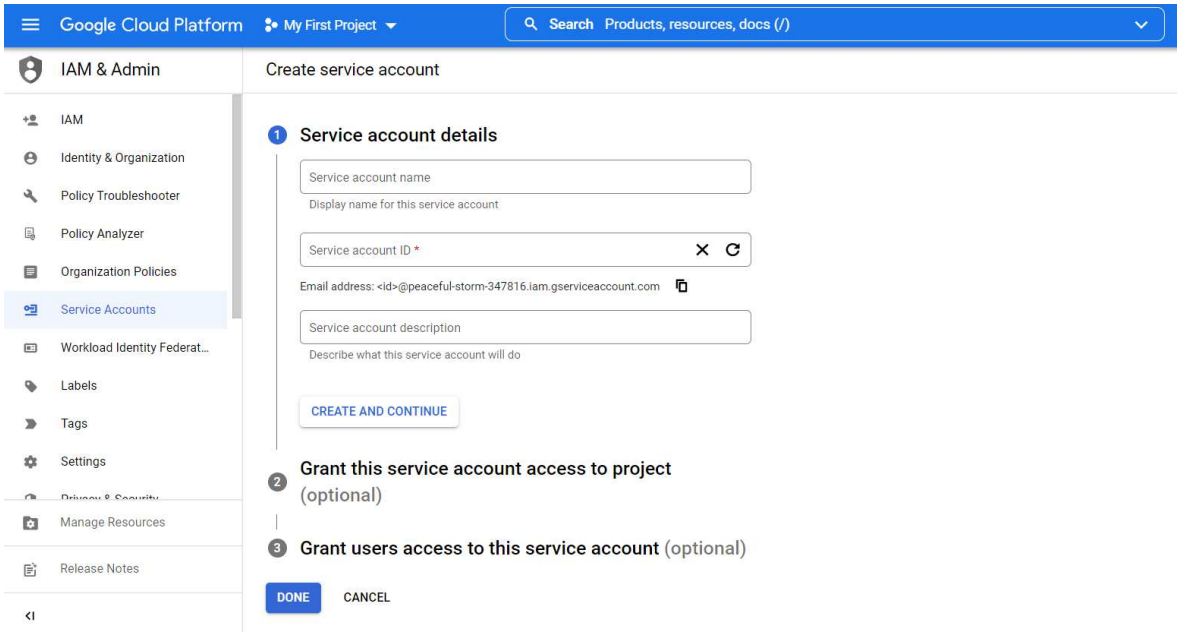
```
curl -X POST
-H "Authorization: Bearer "$(gcloud auth application-default
print-access-token)
-H "Content-Type: application/json; charset=utf-8"
-d @request.json
"https://speech.googleapis.com/v1p1beta1/speech:recognize"
```

Kôd 2.2: Curl naredba za poziv Speech-to-Text API-ja

Za poziv Speech-to-Text API-ja se šalje HTTP POST zahtjev na URL beta inačice API-ja `https://speech.googleapis.com/v1p1beta1/speech:recognize`. Međutim, API ne može bilo tko pozivati – zbog toga je u zahtjevu potrebno proslijediti i autorizacijsko

zaglavlje. Za poziv za prepoznavanje govora moramo biti autorizirani kao korisnik koji ima pravo raditi nad stvorenim *My First Project* projektom. To znači da za obradu zvučnih zapisa treba prvo stvoriti servisnog korisnika koji ima pristup projektu.

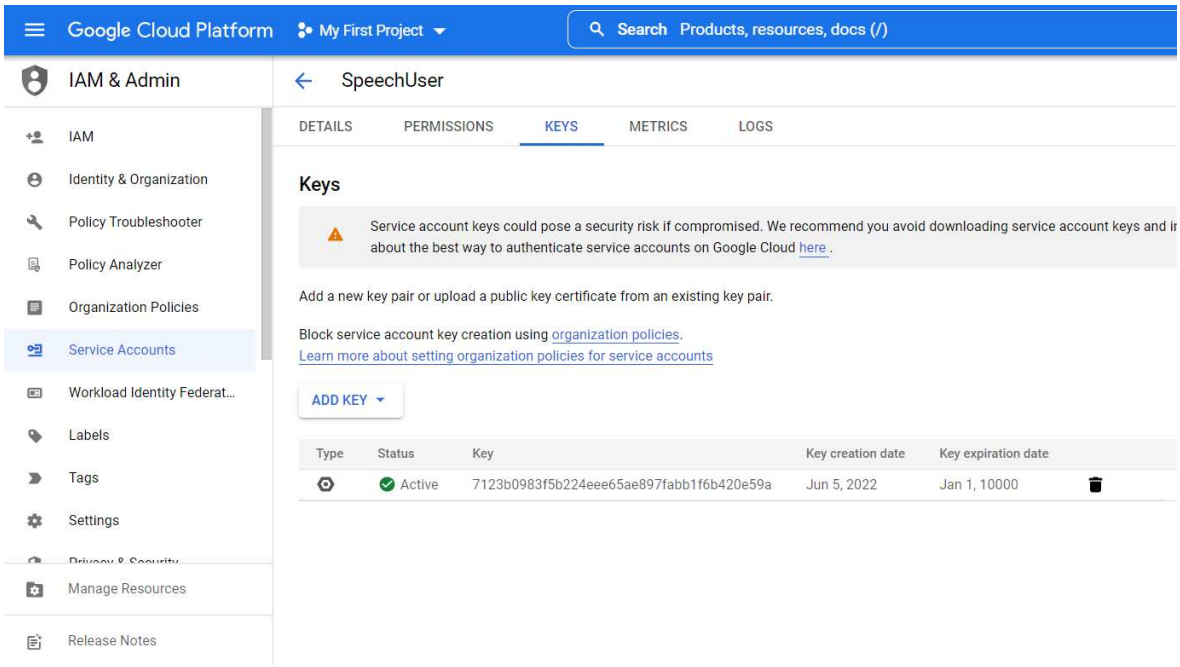
Za kreiranje servisnog računa potrebno je otići u *IAM & ADMIN* postavke projekta, odabrati opciju *Service Accounts* te upisati potrebne podatke za stvaranje servisnog računa. Na Slici 2.3 prikazana je forma za stvaranje servisnog korisnika.



Slika 2.4: Forma za stvaranje servisnog računa

U dijelu *Grant this service account access to project* odabire se opcija *Project > Owner*. Tako servisni korisnik ima jednako visoka prava pristupa resursima kao vlasnik projekta.

Za *My First Project* projekt kreiran je servisni korisnik naziva `SpeechUser`. Nakon što je kreiran, potrebno je korisniku dodati par ključeva kojima će se autorizirati u zahtjevima. Ključevi su mehanizam kojim vlasnik privatnog certifikata (privatnog ključa) dokazuje svoj identitet tako što u zahtjevu prilaže digitalni potpis generiran privatnim ključem. Javni dio ključa nalazi se učitan na Google Cloudu. Ako API uspije verificirati potpis iz zaglavlja zahtjeva, efektivno se utvrđuje da je pošiljatelj zahtjeva zaista onaj kojim se predstavlja. Ključ se može dodati u postavkama servisnog korisnika. Dovoljno je odabrati stvorenog servisnog korisnika, kliknuti na opciju *Keys* i potom opciju *Add Key*. Na Slici 2.5 prikazan je stvoreni aktivni javni dio ključa za servisnog korisnika `SpeechUser`.



Slika 2.5: Ključ za autorizaciju servisnog korisnika

Pri kreiranju ključa automatski se preuzima JSON datoteka na računalo. U toj datoteci pohranjeni su podaci o servisnom računu kojem ključ pripada, projektu kojem pripada servisni račun, identifikacijska oznaka privatnog certifikata (ključa) za digitalni potpis, privatni ključ i ostale servisne informacije. Puna putanja do te JSON datoteke mora na lokalnom uređaju biti spremljena u varijablu okoline naziva `GOOGLE_APPLICATION_CREDENTIALS`. U zahtjevu se autorizacijsko zaglavlje, odnosno token u zaglavlju kreira naredbom:

```
gcloud auth application-default print-access-token
```

Kako bi se ova naredba uspješno izvršila, na Windows uređaju treba biti instaliran Google Cloud CLI prema službenim uputama iz dokumentacije [13].

Sada kad su ostvareni svi preduvjeti za poziv API-ja, možemo dobiti HTTP odgovor koji u sebi sadrži prijepis prepoznatog govora. Odgovor za prethodno opisani HTTP POST zahtjev (zahtjev u Kôdu 2.1) nalazi se u nastavku prikazan Kôdom 2.3.

```
{
  "results": [
    {
      "alternatives": [
        {
          "transcript": "hello my name is",
          "confidence": 0.98762906,
          "words": [
            {
              "startTime": "1.100s",
              "endTime": "1.800s",
              "word": "hello"
            },
            {
              "startTime": "1.800s",
              "endTime": "2.900s",
              "word": "my"
            },
            {
              "startTime": "2.900s",
              "endTime": "3s",
              "word": "name"
            },
            {
              "startTime": "3s",
              "endTime": "3.600s",
              "word": "is"
            }
          ]
        }
      ],
      "resultEndTime": "4.350s",
      "languageCode": "en-us"
    }
  ],
  "totalBilledTime": "15s"
}
```


Kôd 2.3: Rezultat prepoznavanja govora pomoću Speech-to-Text API-ja

Odgovor u obliku JSON-a sadrži sljedeća polja:

- `results` – lista rezultata gdje svaki rezultat odgovara jednom dijelu zvučnog zapisa. Dijelovi zvučnog zapisa međusobno su odvojeni pauzama.

Svaki rezultat sadrži polja:

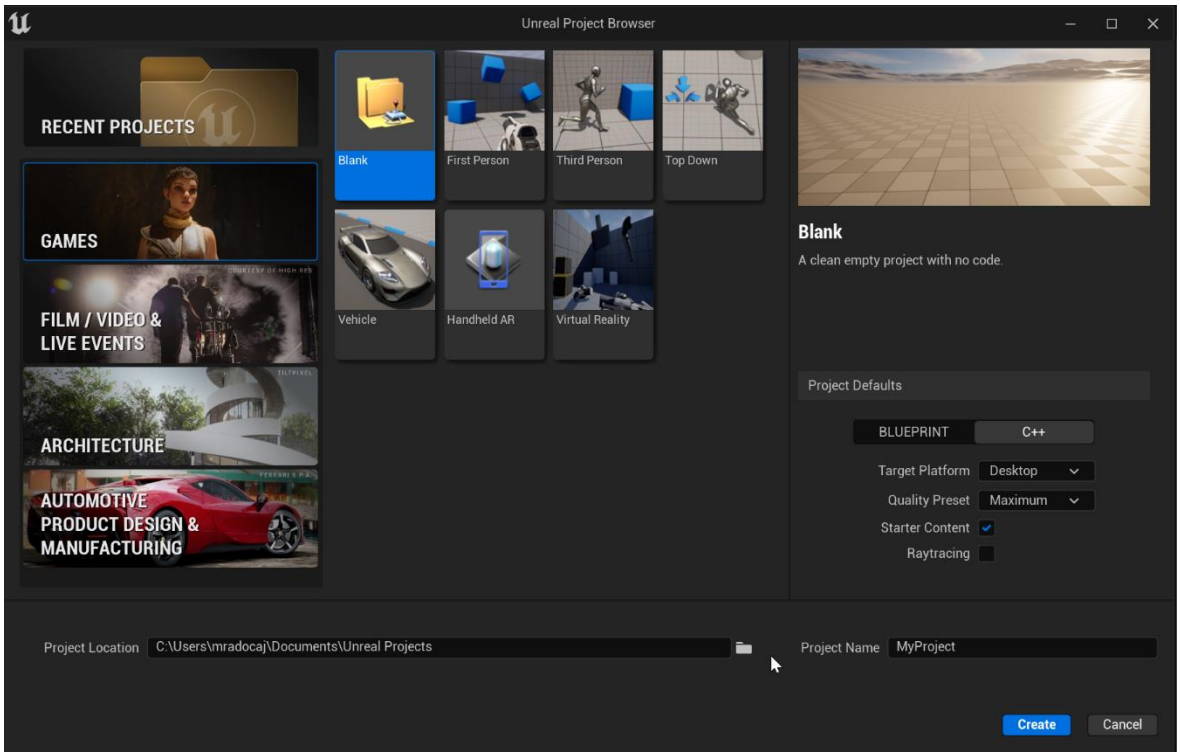
- `alternatives` – lista mogućih transkripcija zvučnog zapisa. Svaki alternativni zapis sadrži polja:
  - `transcript` – prijepis dijela zvučnog zapisa u tekst
  - `confidence` – vrijednost od 0 do 1 kojom API označuje koliko je siguran u rezultat prijepisa
  - `words` – lista pojedinih riječi u prijepisu `transcript`; svaki objekt riječi sadrži riječ koja je izgovorena (`word`), vremenski trenutak početka riječi (`startTime`) i vremenski trenutak završetka riječi (`endTime`)
- `totalBilledTime` – koliko je sekundi obrade zvučnog zapisa zabilježeno i naplaćeno

Na ovaj je način dobivene prijepise govora u tekst sada moguće iskoristiti kod generiranja animacije lica. Sve što je potrebno je uzeti JSON odgovor, parsirati ga i dobiti informaciju koja riječ je izgovorena u kojem trenutku. Za demonstraciju je svih sedam zvučnih zapisa učitanih u Cloud Storage obrađeno na opisani način i pohranjeno na lokalni uređaj. Kasnije je detaljno objašnjeno parsiranje dobivenih JSON-a s prepoznatim govorom u programskom jeziku C++.

2.2. Animiranje pojedinih riječi

Ostatak implementacije odrađen je pomoću alata Unreal Engine. Iako iz JSON odgovora Speech-to-Text API-ja imamo informacije koje riječi treba animirati, ne postoji konkretna animacija koja se treba pustiti u nekom trenutku. Srećom, MetaHuman likovi nude odličnu podršku za kontrolu kostiju lica koja na relativno jednostavan način omogućava stvaranje animacija realističnih pokreta.

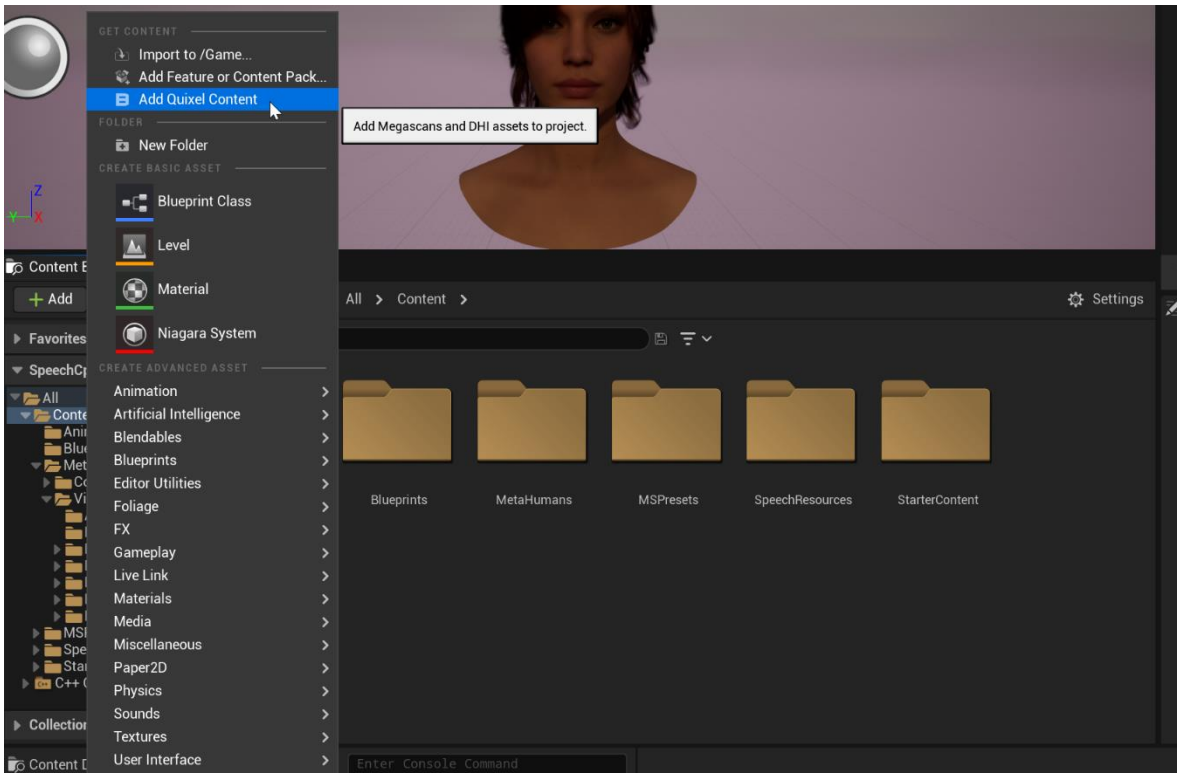
Prvo je potrebno stvoriti Unreal Engine projekt u kojem će se nalaziti lik. Kao što je već navedeno, koristi se najnovija Unreal Engine verzija 5.0.1. Na Slici 2.6 prikazan je ekran za stvaranje projekta za igricu (odabrana je opcija *Games*).



Slika 2.6: Stvaranje C++ projekta u Unreal Engine

Projekt korišten u sklopu ovog rada stvoren je kao projekt tipa C++. Moguće je stvoriti i projekt koji koristi tzv. *Blueprints* umjesto C++ projekta. *Blueprint Visual Scripting* sistem je vizualni skriptni jezik koji se temelji na tzv. čvorovima za stvaranje i obradu događaja u igri. Međutim, za svrhe ovog projekta vizualne skripte nisu bile dovoljno dobro rješenje. Ne postoji jednostavan način da se pomoću takvih skripti obrađuju vanjske datoteke i na temelju njih okidaju animacije. Zbog raznolikosti koju čisti C++ nudi kreiran je prazni C++ projekt naziva `SpeechCpp`.

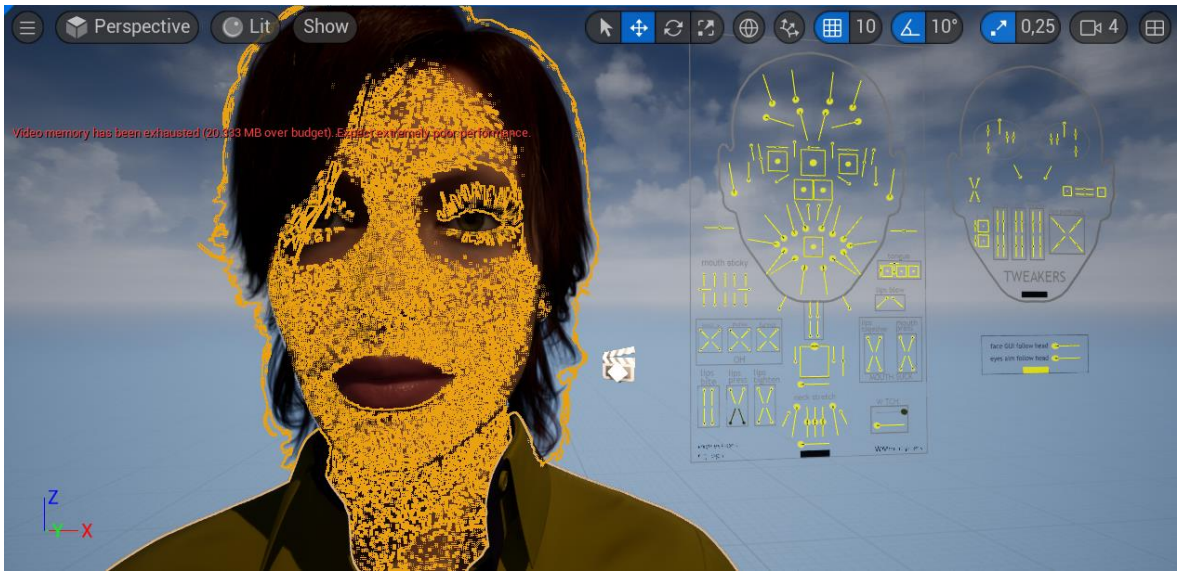
Nakon uspješnog stvaranja projekta, u scenu je potrebno uvesti željeni model nad kojim će se izvoditi animacije. Kao što je već spomenuto, u sklopu rada korišten je MetaHuman lik zvan Vivian. Model je moguće uvesti iz alata Quixel Bridge koji je ugrađen u Unreal Engine. U dijelu *Content Browser* gdje se nalaze početni resursi projekta potrebno je kliknuti na gumb *Add* i odabrati *Add Quixel Content*. Na Slici 2.7 je prikazan proces dodavanja sadržaja u projekt. U Quixel Bridge je dovoljno odabrati željenog lika, pričekati da se preuzmu sav pripadajući sadržaj i uvesti ga u Unreal Engine scenu.



Slika 2.7: Dodavanje sadržaja iz Quixel Bridge u Unreal Engine

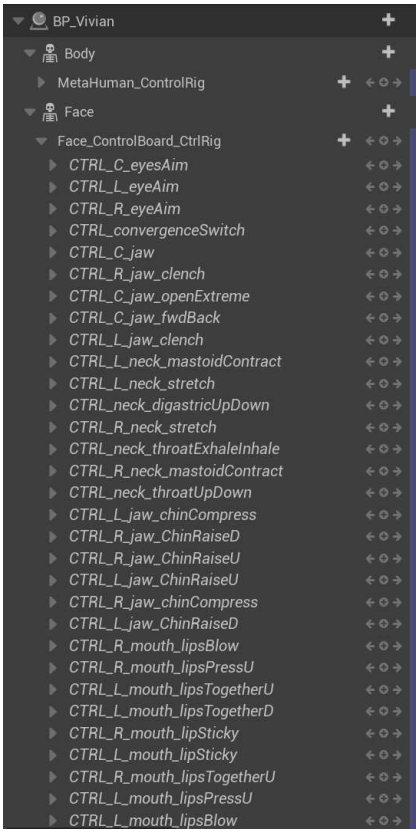
Kad se lik postavi u scenu, moguće je krenuti sa stvaranjem animacija pojedinih riječi. Stvaranje animacije manipulacijom kostiju lica radi se kroz *Sequencer*, alat za stvaranje i pregledavanje filmskih ili video isječaka u igri. Kako bi se omogućilo korištenje Sequencera, potrebno je dodati tzv. *Level Sequence*, odnosno kontejner koji će sadržavati filmsku sekvencu. U projektu je odabirom opcije *Add Level Sequence* kreirana sekvenca naziva `SpeechLevelSequence`.

Svaki MetaHuman lik dolazi sa tzv. *Control Rig* resursom – alatom koji animatorima olakšava kreiranje animacija. *Control Rig* je pojednostavljena vizualna reprezentacija kostiju koje se nalaze u licu ili tijelu lika; nešto poput kontrolne ploče na kojoj animator pomicanjem jedne poluge može liku npr. otvoriti ili zatvoriti usta. Izgled te komponente kao i kompleksni sastav lica MetaHuman lika na koje *Control Rig* utječe vidljiv je na Slici 2.8.



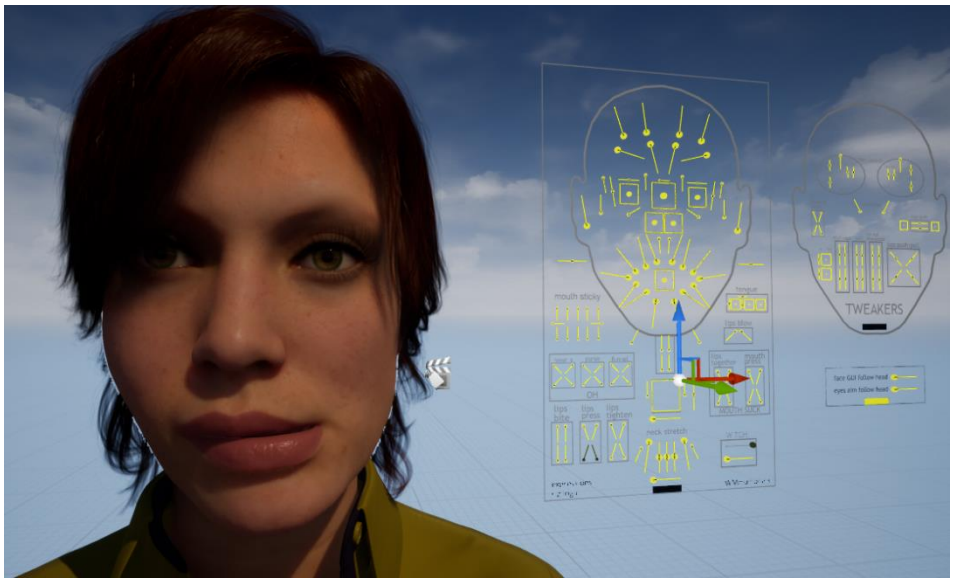
Slika 2.8: *Control Rig* lica (desno) i brojni vrhovi 3D modela lica (lijevo)

Control Rig koji kontrolira lice naziva se *Facial Rig*. *Facial Rig* ima ukupno nevjerojatnih 189 kontrola kojima je moguće manipulirati. Opcije koje je moguće kontrolirati obuhvaćaju sve od otvaranja i zatvaranja usta ili očiju pa do čak pomicanja ušiju. Na Slici 2.9 prikazane su samo neke od mnogobrojnih kontrola.

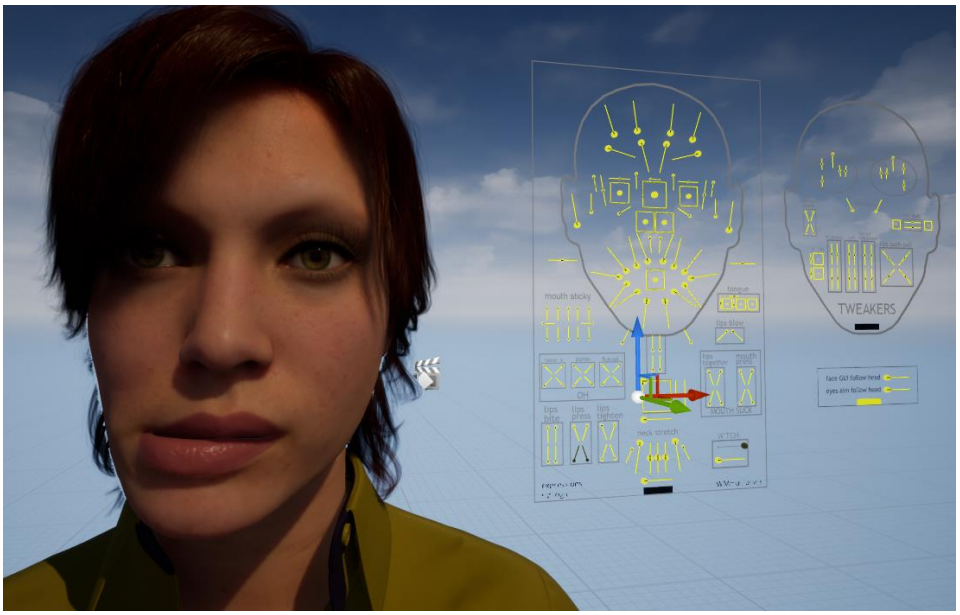


Slika 2.9: Prikaz nekih od mnogobrojnih kontrola *Facial Riga*

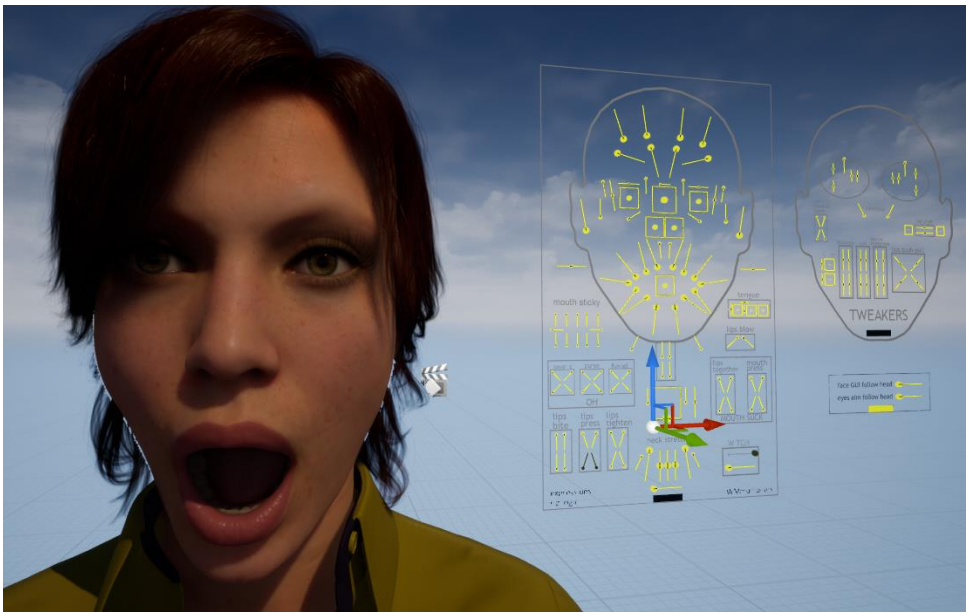
Svaka od ovih kontrola može imati vrijednost postavljenu od 0 do 1 što označava razinu do koje je neki dio lica pomaknut iz početnog položaja. Također, neke kontrole moguće je pomicati u više od jedne različite osi. Tako kontrola za otvaranje čeljusti `CTRL_C_jaw` može biti pomaknuta u dvije osi: *X* i *Y*. Vrijednost za pomicanje po *Y* osi može ići od -1 do 1 (pomicanje čeljusti u potpunosti lijevo ili u potpunosti desno u odnosu na neutralni položaj). Na Slici 2.10, Slici 2.11 i Slici 2.12 prikazan je utjecaj izmjene vrijednosti *X* i *Y* osi za kontrolu `CTRL_C_jaw`.



Slika 2.10: Izgled lica sa postavljenom vrijednosti `CTRL_C_jaw` (*X* = -1,0, *Y* = 0,0)



Slika 2.11: Izgled lica sa postavljenom vrijednosti `CTRL_C_jaw` (*X* = 1,0, *Y* = 0,0)



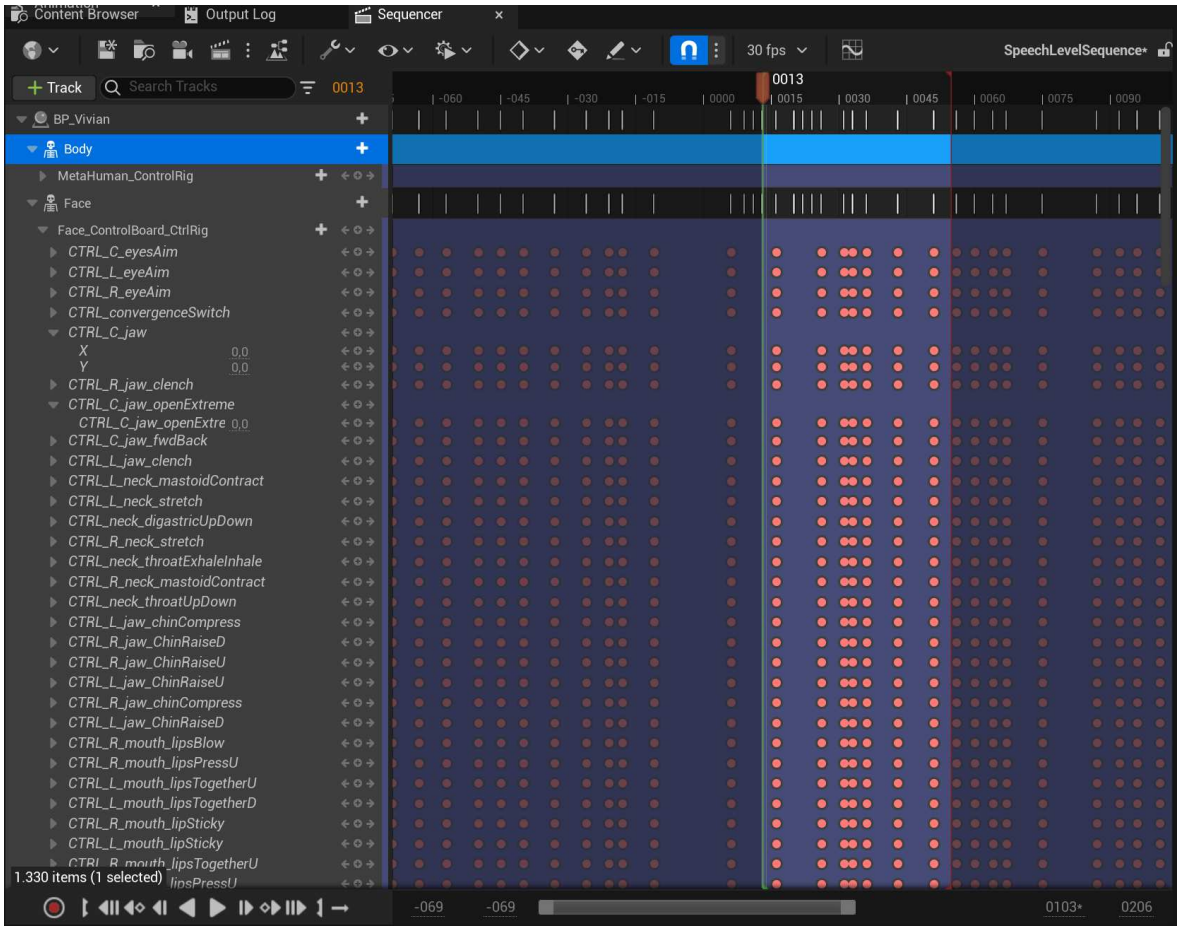
Slika 2.12: Izgled lica sa postavljenom vrijednosti `CTRL_C_jaw` (X = 0,0, Y = 1,0)

Animacije su izrađene metodom tzv. *keyframe* animacijom ili kraće *keyframing*. *Keyframe* animacija je stvaranje digitalne animacije tako što se definira početni i završni položaj kroz vrijeme [14]. Ti početni i završni položaji lika se nazivaju *keyframe*-ovi. Računalni alat za animaciju će zatim između ta dva vremenska trenutka interpolirati ključne položaje lika koji se razlikuju u početnom i završnom položaju.

U Unreal Engine, *Sequencer* alat nudi podršku za stvaranje *keyframe* animacija. Ručno animiranje pojedinih riječi svodi se na pažljivo promatranje izgovora riječi i pomno proučavanje utjecaja izmjene pojedinih kontrola *Facial Riga* na položaj lica. Nakon što se nauči koje su najbitnije kontrole za manipulacijom donjeg dijela lica (donji dio lica je ključan za animaciju izgovora riječi), kontrole je potrebno primijeniti kako bi se što vjernije reproducirao izgovor riječi.

Zvučni zapisi snimljeni za demonstraciju 3D animacije sadrže govor na engleskom jeziku (američki naglasak). Stoga je za animaciju bilo potrebno dobro proučiti izgovor engleskih riječi. Pojednostavljenje koje se načelno koristilo je sljedeće: jedan fonem u riječi – jedan *keyframe* u animaciji. Dakle, ako se animirala npr. riječ „hello“, pomoću javno dostupnog konvertera [15] je engleska riječ prebačena u fonemski zapis. Fonemski zapis riječi engleske riječi „hello“ s američkim naglaskom je „həˈloʊ“. Fonemski zapis dakle sadrži ukupno pet znakova (pet različitih fonema). Za svaki od njih je stvoren po jedan *keyframe* koji što vjernije prikazuje način izgovora tog fonema. Ako se isti fonem kasnije pojavi u nekoj drugoj riječi koja se animira, nije potrebno ponovno ga reproducirati, već se navedeni *keyframe*

jednostavno ponovno iskoristi. Na Slici 2.13 prikazani su *keyframeovi* koji sačinjavaju animaciju riječi „hello“. *Keyframeovi* unutar zelene i crvene vertikalne crte odnose se na riječ koja se promatra. Početni i završni *keyframe* nisu fonemi, nego neutralni položaj lica (prije i poslije svake izgovorene riječi lice se nalazi u neutralnom položaju).

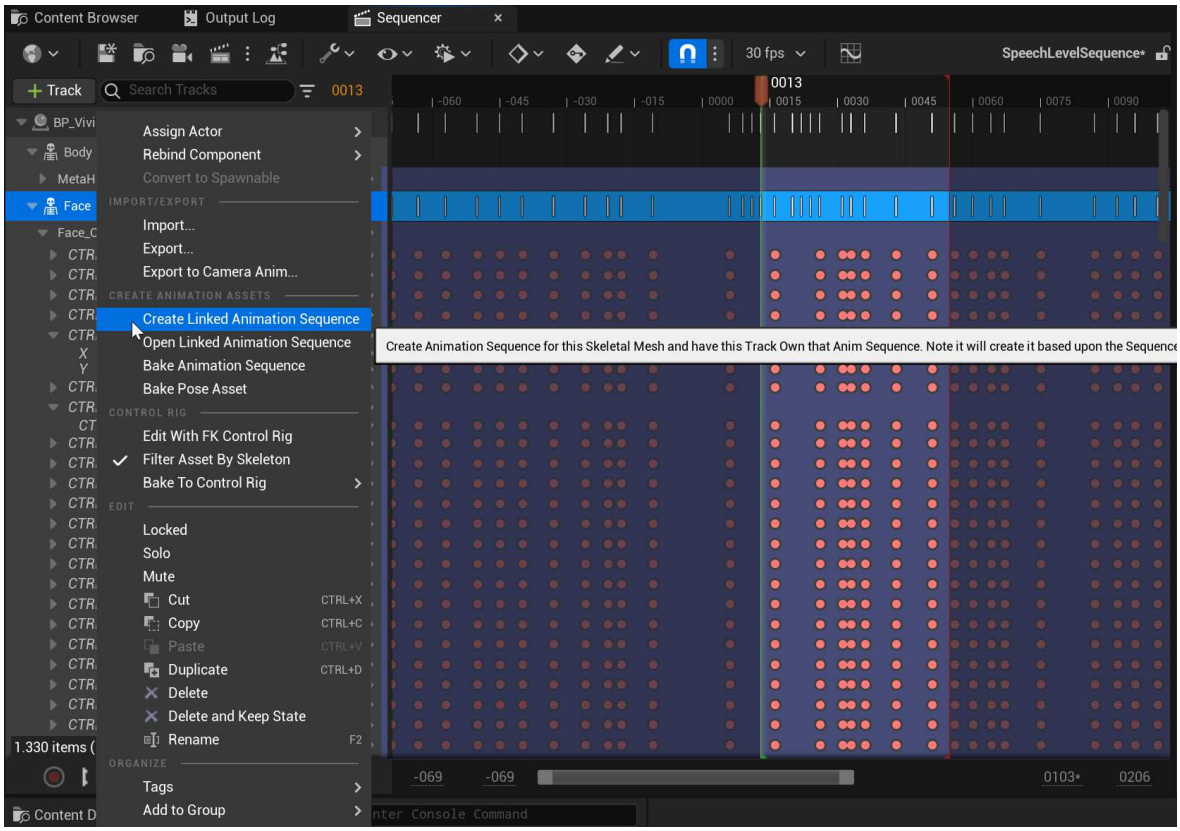


Slika 2.13: *Keyframe* animacija riječi „hello“

Osim animacije riječi „hello“, izrađeno je još sedam animacija: animacije riječi „my“, „name“, „is“, „cat“, „what“, „your“ i jedna animacija lica u neutralnom položaju s treptanjem. Animiranje izgovora pojedinih riječi jedan je od najkompleksnijih i najdugotrajnijih procesa u cijelom sustavu generiranju 3D animacije lica iz zvuka govora. Konačna kvaliteta i realističnost rezultata u velikoj mjeri ovisi upravo o kvaliteti animacija pojedinih riječi. Također, za generiranje animacija kompleksnijih rečenica s više riječi bilo bi potrebno ručno animirati mnogo više riječi (npr. u engleskom jeziku ima ukupno preko 170 tisuća riječi – to je strašno puno animiranja!). Nešto kasnije će se razmatrati na koji način se može doskočiti problemu kompleksnog procesa animiranja jedne po jedne riječi.

Nakon što su izrađeni *keyframeovi* za animaciju svake od riječi, potrebno je stvoreno sekvencu u *Sequenceru* „rascjepkati“ i izvesti pojedine animacije koje mogu biti korištene

kao resursi kasnije. Za to je potrebno zelene i crvene kursore postaviti tako da obuhvaćaju čitavu riječ kao što je prikazano na Slici 2.13, desnim klikom miša kliknuti u izborniku na kostur lica *Face* i odabrati opciju *Create Linked Animation Sequence*. Taj proces prikazan je na Slici 2.14.



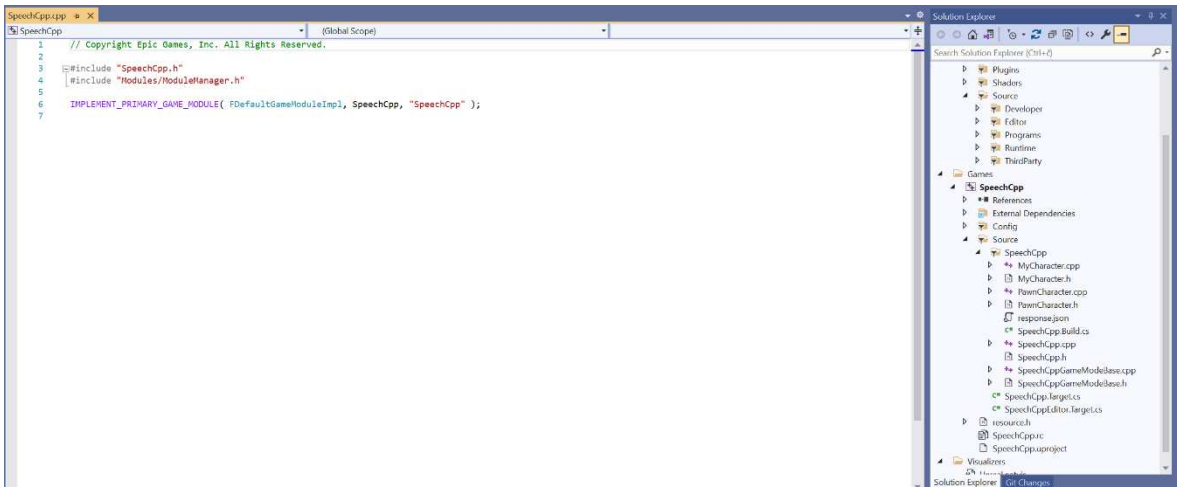
Slika 2.14: Izvoz animacije riječi „hello“

Sve što je nakon toga potrebno je odabrati putanju do direktorija gdje se želi pohraniti animacija. Time je kreirana animacija, odnosno Unreal Engine resurs tipa *Animation Sequence*. Takav se resurs dalje može koristiti kao čvor u vizualnoj *Blueprint* skripti, ili u našem slučaju kao resurs u C++ implementaciji lika.

2.3. Stvaranje lika u sceni

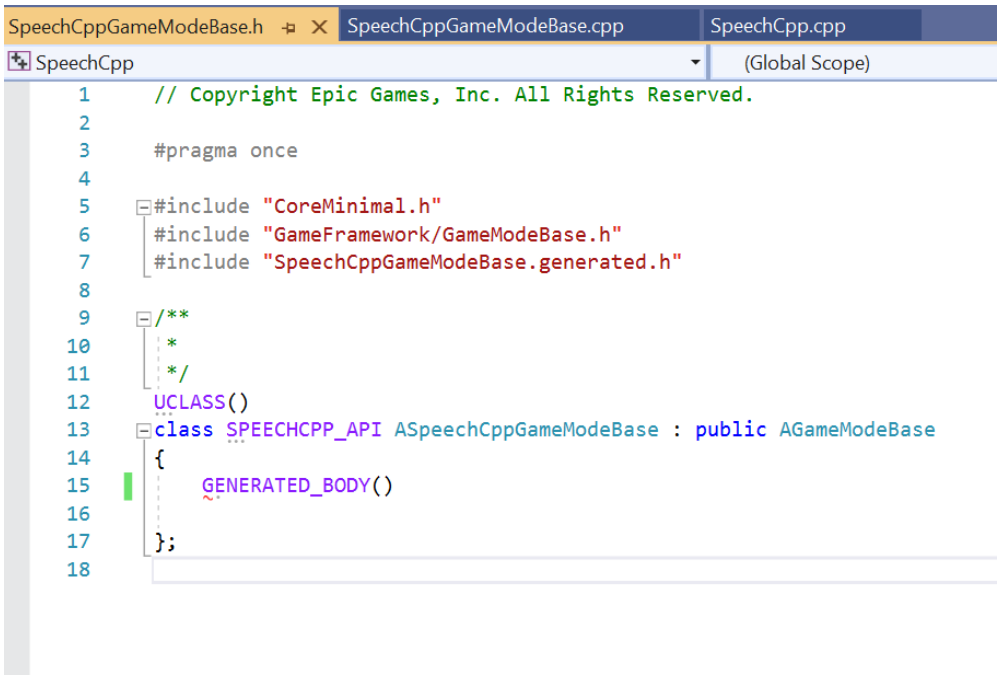
Kako bi se omogućio rad s likom u sceni i baratanje s njegovim resursima iz koda, potrebno je stvoriti implementaciju koju će taj lik naslijediti. Za početak je potrebno objasniti hijerarhiju klasičnog Unreal Engine projekta.

Pri stvaranju C++ Unreal Engine projekta odmah se stvara i Visual Studio C++ projekt istog naziva. Početno stvoreni kod `SpeechCpp.cpp` predstavlja inicijalni modul igre i prikazan je na Slici 2.15.



Slika 2.15: Inicijalna klasa `SpeechCpp.cpp` i Visual Studio projekt

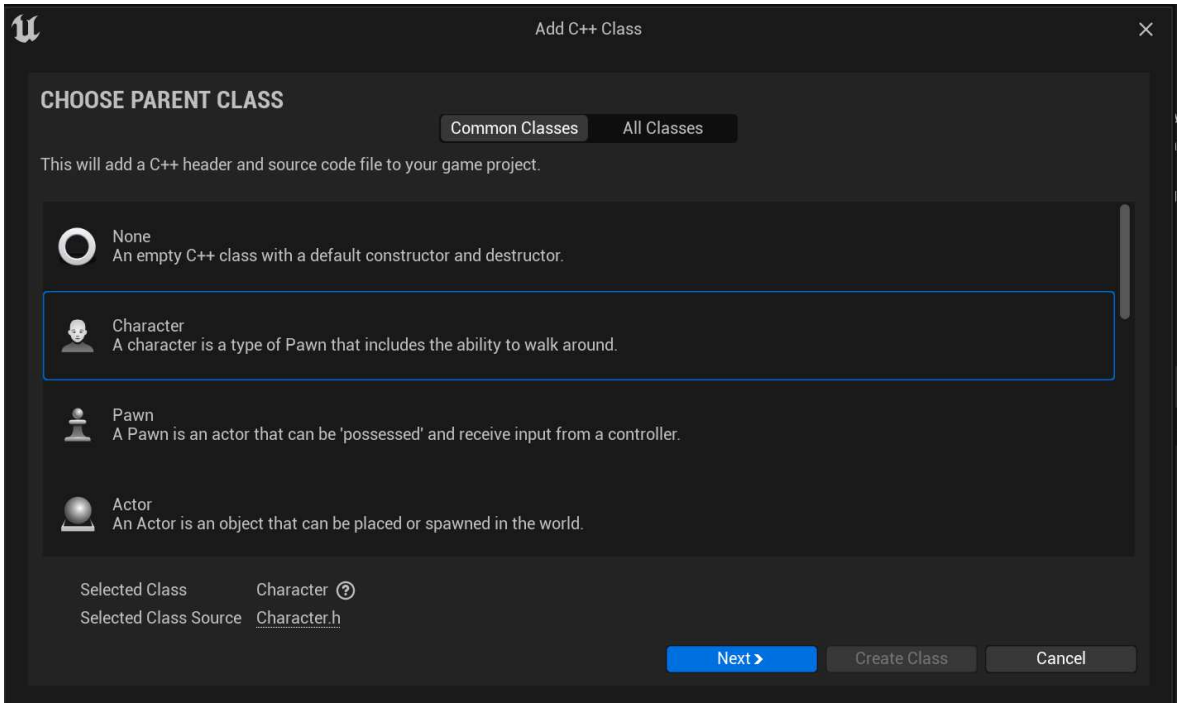
Uz `SpeechCpp.cpp`, automatski su generirane i datoteke `SpeechCppGameModeBase.cpp` i `SpeechCppGameModeBase.h`. U Unreal Engine hijerarhijskoj podjeli, *GameMode* je nešto poput *Controller* dijela u MVC arhitekturi. On definira logiku i komunicira direktno s drugom komponentom naziva *GameState*. *GameState* predstavlja *Model* dio MVC arhitekture: u *GameState* se pohranjuju podaci igre, pišu se funkcije i učitavaju i pohranjuju podaci. Na Slici 2.16 vidljiva je osnovna implementacija `SpeechCppGameModeBase.h`. Iako u okviru ovog projekta nisu rađene dorade na početnom automatski generiranom *GameMode*-u, bitno je razumjeti čitavu hijerarhijsku podjelu klasične Unreal Engine računalne igre.



Slika 2.16: Generirana implementacija `SpeechCppGameModeBase.h`

Dakle, *GameMode* služi za detektiranje i početnu obradu događaja u igri te on dalje poziva *GameState* za konačnu obradu. No tko stvara događaje u sceni? Svaki objekt postavljen u sceni naziva se *Actor*. Postoji nekoliko vrsta *Actora* – pretpostavljena vrsta *Actora* je obični statični resurs koji može imati neki pripadajući 3D model, izvor zvuka, itd. Posebna vrsta *Actora* je tzv. *Pawn* – dinamični *Actor* koji može pripadati nekom igraču. Kad *Pawn* pripada nekom igraču, on može implementirati neke događaje kao npr. kretati se u nekom smjeru na pritisak tipke igrača. Još jedna razina iznad *Pawna* je tzv. *Character*: to je *Pawn* s ljudskim osobinama. Dolazi s unaprijed definiranim resursima za definiranje kolizije, ljudske animacije, itd.

Za potrebe ovog rada potrebno je stvoriti novog *Charactera* u projektu. U njemu će biti implementirana sva logika obrade datoteke s prepoznatim govorom i generiranje animacije. Dodavanje nove C++ klase radi se kroz Unreal Engine. U željenom direktoriju potrebno je odabrati opciju *New C++ Class...* nakon čeka se otvara izbornik prikazan na Slici 2.17.



Slika 2.17: Izbornik za stvaranje nove C++ klase

U kodu Kôd 2.4 prikazana je početna generirana implementacija *Character* klase `MyCharacter.h` koja nasljeđuje `ACharacter` klasu. Svaka klasa koja nasljeđuje `ACharacter` klasu sadrži konstruktor i određene funkcije koje je moguće nadjačati. Te funkcije su `BeginPlay`, `Tick` i `SetupPlayerInputComponent`.

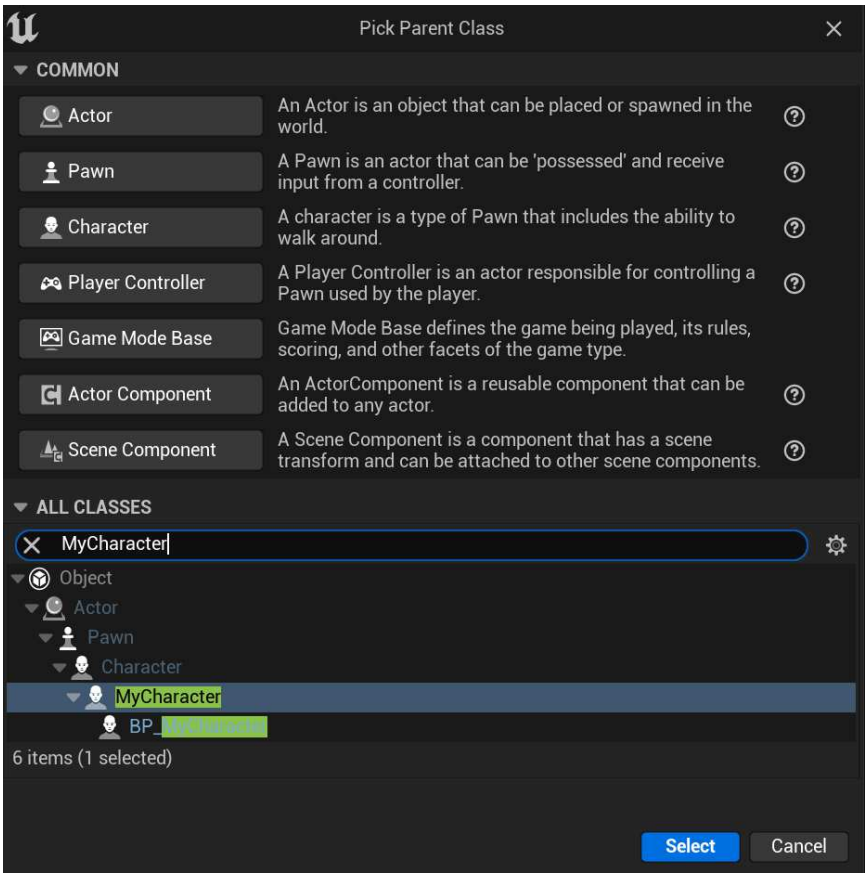
```
#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "MyCharacter.generated.h"

UCLASS()
class SPEECHCPP_API AMyCharacter : public ACharacter
{
	GENERATED_BODY()

public:
	// Sets default values for this character's properties
	AMyCharacter ();

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;

public:
	// Called every frame
	virtual void Tick(float DeltaTime) override;

	// Called to bind functionality to input
	virtual void SetupPlayerInputComponent(class
UInputComponent* PlayerInputComponent) override;

};
```

Kôd 2.4: Početna generirana implementacija datoteke zaglavlja `MyCharacter.h`

Implementacija ovih funkcija nalazi se u `MyCharacter.cpp`. U datoteku zaglavlja će se nešto kasnije dodati nove funkcije i članske varijable koje će biti potrebne za baratanje animacijama.

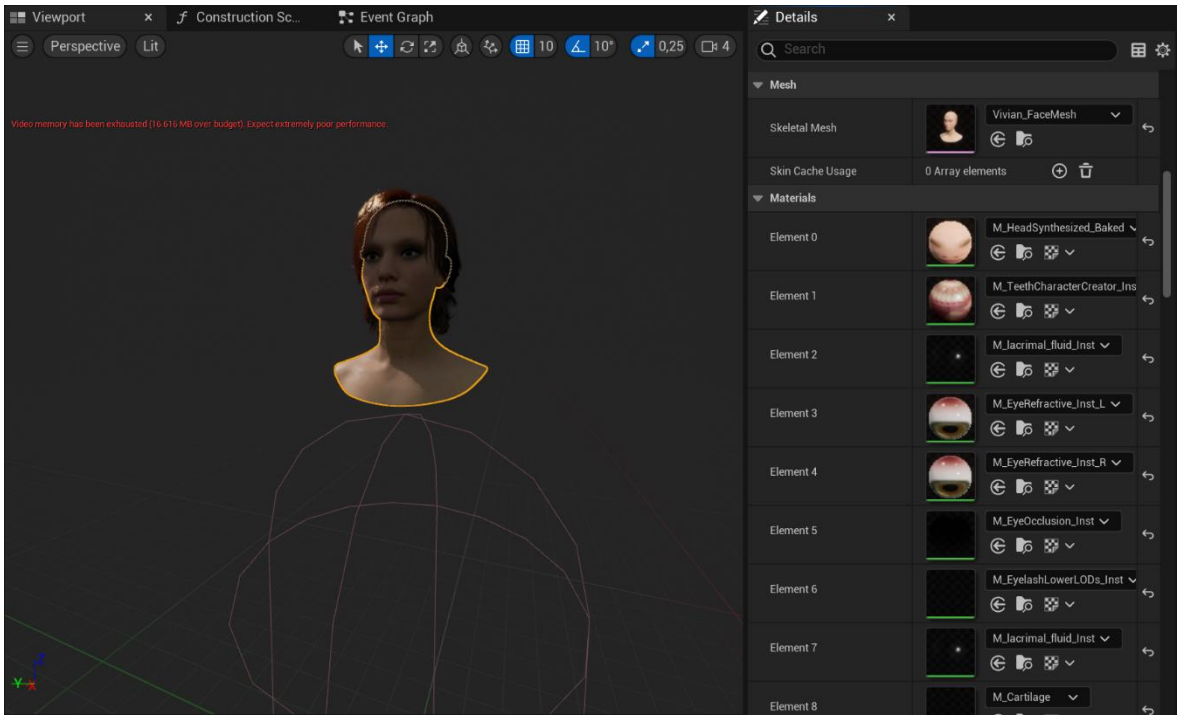
Sljedeći korak je u scenu u Unreal Engine projektu dodati konkretnog *Actora*, odnosno *Charactera* koji implementira ovu klasu. Također, želimo da navedeni *Character* ima kostur lica, 3D model i odgovarajuće resurse koje smo uvezli uz MetaHuman model Vivian. Povezivanje svega od navedenog je moguće kreiranjem nove *Blueprint* klase. Blueprint klasa omogućava korisniku da na jednostavan način dodaje funkcionalnosti i nadograđuje nove komponente na neku postojeću klasu. U našem slučaju želimo nadograditi klasu `MyCharacter.cpp` tako da joj dodamo 3D model lica, kostur i ostale resurse koje čine prikaz

lika (npr. kosu, trepavice, itd.). Pozicioniranjem u direktorij u kojem želimo stvoriti *Blueprint* klasu i odabirom opcije *Create Basic Asset* > *Blueprint Class* otvara se izbornik prikazan na Slici 2.18. Kao roditeljsku klasu treba odabrati *Character* `MyCharacter`.



Slika 2.18: Stvaranje nove *Blueprint* klase koja nasljeđuje `MyCharacter`

Na ovaj način stvorena je nova *Blueprint* klasa naziva `BP_MyCharacter`. Pri kreiranju svaki *Character* već ima stvorenu komponentu za definiranje područje kolizije. Ono što je potrebno je dodati novu komponentu kostura i modela lica. Dvostrukim klikom na novostvorenu *Blueprint* klasu `BP_MyCharacter` otvara se poseban prozor za uređivanje klase. U *Components* dijelu, odabirom opcije *Add* otvara se prozor koji nudi mnoštvo mogućih komponenti koje se mogu pridijeliti *Blueprint* klasi. Zatim se odabira opcija Skeletal Mesh. Nakon što se doda *Skeletal Mesh* komponenta, u njenim detaljima je potrebno u *Mesh* dijelu učitati `Vivan_FaceMesh` koji je prethodno bio uvezen u projekt pri učitavanju MetaHuman modela. Dodatno su uvezeni još i statični modeli trepavica, kose i obrva. Detalji stvorene *Skeletal Mesh* komponente i konačan izgled `BP_MyCharacter` klase vidljiv je na Slici 2.19.



Slika 2.19: Konačan izgled `BP_MyCharacter` klase (označen je kostur i model lica)

Sada je sve spremno za implementaciju generiranja animacije. Instanca stvorene *Blueprint* klase postavljena je u scenu igre. Pošto instanca te klase direktno nasljeđuje `MyCharacter` klasu, pri njenom stvaranju se poziva `MyCharacter` konstruktor. Isto tako, pokretanjem igre klikom na *Play* gumb poziva se metoda `BeginPlay` implementirana u `MyCharacter.cpp`. U nastavku slijedi detaljan opis implementacije.

2.4. Generiranje animacije govora

Sva C++ implementacija nalazi se u datotekama `MyCharacter.cpp` i `MyCharacter.h`. Kao što je već spomenuto, pri stvaranju instance `BP_MyCharacter` poziva se konstruktor koji generira *Charactera*. U konstruktorima *Actor* objekata općenito se moguće referencirati na druge objekte ili resurse koji se nalaze negdje u Unreal Engine projektu. Nakon što su dohvaćeni, spomenute resurse moguće je koristiti za potrebe implementacije logike lika.

Pronalazak objekata u projektu radi se definiranjem posebnih pomoćnih predložaka funkcija `ConstructorHelpers::FObjectFinder`. Tip predloška definira koju vrstu objekta treba dohvatiti. U primjeru koda Kôd 2.5 prikazan je dio implementacije funkcije konstruktora u klasi `MyCharacter`.

```
AMyCharacter::AMyCharacter()
{
	// ...
	// Izostavljeni pozivi FobjectFinder za ostalih 7
animacija
	static ConstructorHelpers::FObjectFinder<UAnimSequence>
catAnimation(TEXT("AnimSequence'/Game/Metahumans/Vivian/Anima
tions/Face_Cat'"));
	static ConstructorHelpers::FObjectFinder<USoundBase>
speechAudio(TEXT("/Game/SpeechResources/AudioCue"));

	HelloAnimSequence = helloAnimation.Object;
	MyAnimSequence = myAnimation.Object;
	NameAnimSequence = nameAnimation.Object;
	IsAnimSequence = isAnimation.Object;
	WhatAnimSequence = whatAnimation.Object;
	YourAnimSequence = yourAnimation.Object;
	CatAnimSequence = catAnimation.Object;
	IdleAnimSequence = idleAnimation.Object;
	SpeechAudio = speechAudio.Object;
}
```

Kôd 2.5: Konstruktor `MyCharacter` klase (nisu prikazani svi pozivi `FObjectFinder`)

U konstruktoru se učitavaju dvije vrste objekata iz projekta. Prva vrsta su `UAnimSequence` objekti koji predstavljaju resurse animacije. Pri stvaranju lika, želimo da su mu dostupne sve animacije pojedinih riječi kako bismo ih kasnije mogli koristiti pri generiranju animacije rečenica. Za dohvat objekta potrebno je proslijediti relativnu putanju do lokacije resursa u projektu. Svaka pojedina animacija spremljena je kao članska varijabla u `MyCharacter.h`. Tako npr. za riječ *cat* (engl. mačka) u `MyCharacher.h` se nalazi varijabla `CatAnimSequence`:

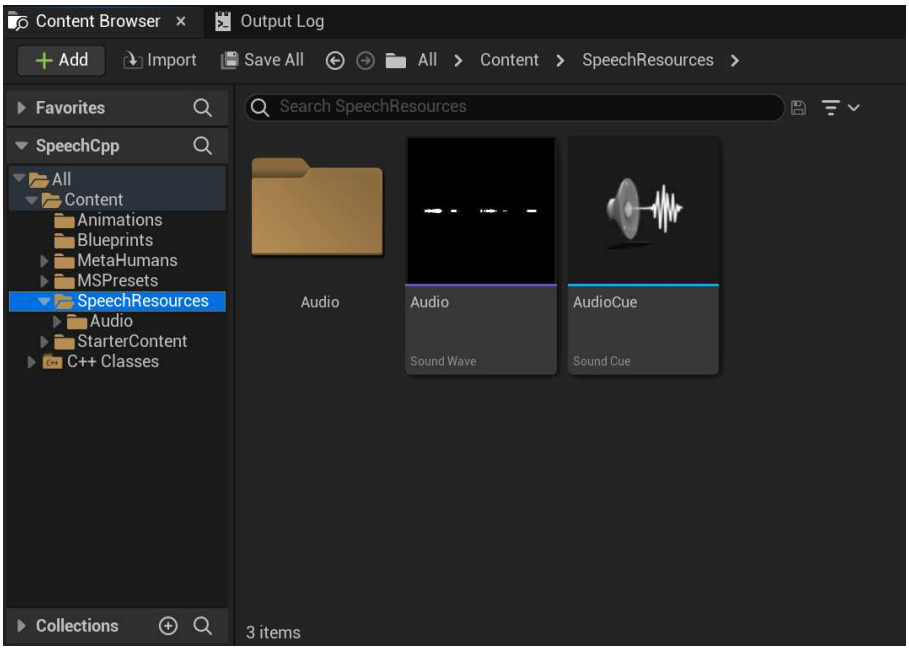
```
// 'Cat' animation sequence
UAnimSequence* CatAnimSequence;
```

U konstruktoru se svaka od varijabli za resurse animacija postavlja iz `FObjectFindera` na sljedeći način:

```
CatAnimSequence = catAnimation.Object;
```

Druga vrsta objekta koja se dohvaća u konstruktoru je `USoundBase` objekt. Kod početka igre želimo pustiti zvučni zapis koji će biti usklađen sa animacijom govora. Svaki od zvučnih zapisa pretvoren je iz MP3 formata u WAV format jer Unreal Engine podržava samo to

kodiranje. Čisti zvučni zapis u UE naziva se *Sound Wave*. Na istoj lokaciji potrebno je stvoriti i resurs naziva *Sound Cue* koji kao izvor zvuka koristi postojeći *Sound Wave*. *Sound Cue* je resurs koji se u kodu učitava kao `USoundBase` objekt. Na Slici 2.20 prikazani su korišteni *SoundWave* i *SoundCue* resursi u UE projektu.



Slika 2.20: Zvučni resursi u Unreal Engine projektu

Kao i za `UAnimSequence` objekt, za dohvat `USoundBase` objekta potrebno je proslijediti relativnu putanju do resursa koja je vidljiva i na prethodnoj slici. Kasnije će za stvaranje animacije iz nekog drugog zvučnog zapisa biti potrebno na istoj lokaciji staviti neki drugi *Sound Cue* objekt. Uz stavljanje drugog zvučnog resursa, na tu lokaciju treba premjestiti i odgovarajuću `response.json` datoteku koja sadrži prijepis govora u tekst.

Nakon što su svi potrebni resursi iz projekta dohvaćeni u `MyCharacter` liku, oni se mogu koristiti u implementaciji funkcije `BeginPlay` koja se poziva pri pokretanju igre. U funkciji `BeginPlay` prvo se dohvaća `response.json` datoteka. Ona se potom parsira koristeći vanjsku biblioteku otvorenog koda za serijalizaciju i deserijalizaciju JSON objekata autora Niels Lohmann [16]. Dovoljno je navedenu biblioteku preuzeti sa službene GitHub stranice i uključiti ju u projekt koristeći:

```
#include "nlohmann/json.hpp"
```

U primjeru Kôd 2.6 prikazano je učitavanje i parsiranje `response.json` datoteke koristeći navedenu biblioteku.

```
// Load json file
std::ifstream t("C:\\Users\\mradocaj\\Documents\\Unreal
Projects\\SpeechCpp\\Content\\SpeechResources\\response.json"
);
std::stringstream buffer;
buffer << t.rdbuf();
std::string content = buffer.str();

// Parse json file
json object = json::parse(content);
json results = object["results"];
```

Kôd 2.6: Učitavanje i parsiranje JSON datoteke koristeći biblioteku `nlohmann/json.hpp`

Sada želimo iz JSON objekta iščitati svaku izgovorenu riječ, njeno vrijeme početka i završetka te u to vrijeme pustiti odgovarajuću animaciju. Za ovaj dio potrebno je dobro razumjeti format odgovora koji se dobiva od Speech-to-Text API-ja. Format odgovora je detaljno opisan u poglavlju *2.1. Prepoznavanje govora.* Na temelju formata odgovora nastala je obrada odgovora koja je opisana pseudokodom prikazanim Kôdom 2.7.

```
Za (rezultat u results) {
      sve_alternative = rezultat["alternatives"];
      prva_alternativa = sve_alternative[0];
      rijeci = prva_alternativa["words"];

      Za (rijec u rijeci) {
            rijec_tekst = rijec["word"];
            pocetak = rijec["startTime"];
            kraj = rijec["endTime"];
            pusti_animaciju(rijec_tekst, pocetak, kraj);
      }
}
```

Kôd 2.7: Pseudokôd generiranja animacije na temelju JSON odgovora

Kao što je već objašnjeno, u odgovoru se u polju `results` nalazi niz rezultata prepoznavanja govora. Svaki rezultat jedan je dio govora odvojen pauzom pa želimo svaki od njih uključiti u završnu animaciju. Međutim, svaki rezultat u polju `alternatives` sadrži i polje mogućih alternativnih rezultata. Pošto iz samog odgovora ne znamo točno koju alternativu treba animirati, bira se prva po redu. Prva alternativa u odgovoru ujedno je i alternativa s najvećom mjerom sigurnosti. Potom se iterira po svakom objektu riječi koji se nalazi u odabranoj alternativi. Koristeći informacije iz objekta riječi (tekstualni zapis riječi, vrijeme početka i

vrijeme završetka), pušta se jedna od animacija koje su učitane prilikom konstruiranja instance `MyCharacter` klase. Nakon učitavanja podataka iz JSON objekta, obavljaju se transformacije kako bi se dobili podaci u ispravnom tipu i obliku (npr. trajanja se prebacuju u `float` tip, a zapis riječi u velika slova i bez navodnika). Navedene transformacije prikazane su u Kôdu 2.8.

```
std::string wordString = word["word"].dump();
std::string startTimeString = word["startTime"].dump();
std::string endTimeString = word["endTime"].dump();

wordString = wordString.substr(1, wordString.size() - 2);
std::transform(wordString.begin(), wordString.end(),
wordString.begin(), ::toupper);
startTimeString = startTimeString.substr(1,
startTimeString.size() - 3);
endTimeString = endTimeString.substr(1, endTimeString.size()
- 3);

float startTime = std::stof(startTimeString);
float endTime = std::stof(endTimeString);
float duration = endTime - startTime;
```

Kôd 2.8: Transformacije učitanih podataka

Sljedeći izazov je pronaći način za pozivanje funkcije u točno određenom vremenskom trenutku. Srećom, Unreal Engine nudi podršku za postavljanje tzv. *Timera*, alat koji se koristi za pozivanje neke akcije jednom ili više puta u vremenskim intervalima. Sintaksa za stvaranje *Timera* prikazana je u Kôdu 2.9.

```
if (wordString == "HELLO") {
      FTimerDelegate HelloDelegate =
            FTimerDelegate::CreateUObject(this,
            &AMyCharacter::PlayAnimation, HelloAnimSequence,
            HELLO_DURATION / duration);
      GetWorldTimerManager().SetTimer(TimerHandle[i],
            HelloDelegate, startTime + DELAY, false);
}
else if (wordString == "MY") {
      // ...
}
```

```
// Ista logika i za ostale rijeci
```

Kôd 2.9: Stvaranje *Timera* za odgođeni poziv funkcije

Postavljanje *Timera* vrši se pozivom ugrađene funkcije `GetWorldTimerManager().SetTimer`. Svaki poziv za postavljanje ove funkcije prima nekoliko argumenata. Prvi argument je objekt tipa `FtimerHandle`, jedinstveni upravitelj koji se koristi za prepoznavanje *Timera* u UE svijetu. U `MyCharacter.h` definirano je polje `FTimerHandle` objekata:

```
FTimerHandle TimerHandle[30];
```

Za svrhe demonstracije odabrana je duljina polja 30 jer nijedan zvučni zapis ne sadrži govor čiji broj riječi prelazi 30. Za svaku izgovorenu riječ postavlja se poseban *Timer*, a svaki *Timer* mora imati svoj jedinstveni `FTimerHandle`.

Osim `FTimerHandle` objekta, `SetTimer` funkcija prima i argument tipa `FTimerDelegate`. To je objekt koji omogućava stvaranje funkcije koja ima postavljene argumente. Bez korištenja `FTimerDelegate`, u `SetTimer` ne bi bilo moguće proslijediti *callback* funkciju koja prima neke argumente. `FTimerDelegate` se stvara pozivom:

```
FTimerDelegate::CreateUObject(this,
&AMyCharacter::PlayAnimation, HelloAnimSequence,
HELLO_DURATION / duration);
```

Prvi argument u `CreateUObject` funkciji je vlasnik funkcije. Nakon toga slijedi funkcija na koju želimo vezati neke argumente. U našem slučaju je to funkcija `PlayAnimation` koja prima dva argumenta: animaciju tipa `UAnimSequence` koju treba pustiti te brzinu izvođenja animacije tipa `float`. Za svaku pojedinu animaciju definirana je duljina njezinog trajanja u sekundama u statičkim varijablama na sljedeći način:

```
static float HELLO_DURATION = 1.2f;
```

Na isti način definirane su i druge statičke varijable `MY_DURATION`, `NAME_DURATION`, itd. Konačna se brzina izvođenja animacije pojedine riječi računa prema izrazu (1).

$$brzina = \frac{trajanje\ animacije}{trajanje\ izgovorene\ riječi} \tag{1}$$

Ovakvo računanje brzine animacije kasnije omogućava realistični prikaz animacije riječi bilo kakvog trajanja (npr. ako se riječ u zvučnom zapisu vrlo sporo izgovara, rezultat će i dalje biti odgovarajuć bez obzira na trajanje originalne animacije).

Konačna dva argumenta koje prima `SetTimer` funkcija su vremenski trenutak početka izvođenja funkcije (tip `float`) koje je proslijeđena u `FTimerDelegate` te `boolean` argument koji označava treba li se funkcija ponavljati ili ne. Zbog brzine obrade u Unreal Engine potrebno je učitanom vremenu početka riječi dodati iznos `DELAY` kako bi se uskladio zvuk govora i izvođena animacija. Mi želimo animaciju pustiti samo jednom u određenom trenutku pa je stoga zadnji argument u `SetTimer` `false`.

Implementacija funkcije `PlayAnimation` koja konačno pušta animaciju prikazana je u Kôdu 2.10.

```
void AMyCharacter::PlayAnimation(UAnimSequence* Animation,
float PlayRate)
{
	UE_LOG(LogTemp, Warning, TEXT("Playing animation with
	play rate %f"), PlayRate);

	USkeletalMeshComponent* faceMesh = GetMesh();
	faceMesh->Stop();
	faceMesh->
	SetAnimationMode(EAnimationMode::AnimationSingleNode);
	faceMesh->SetAnimation(Animation);
	faceMesh->SetPlayRate(PlayRate);
	faceMesh->Play(false);
}
```

Kôd 2.10: Implementacije funkcije `PlayAnimation`

U prvoj liniji funkcije poziva `UE_LOG` pomoćna funkcija za logiranje na UE konzolu. Nakon toga dohvaća se objekt tipa `USkeletalMeshComponent` pozivom ugrađene funkcije `ACharacter::GetMesh`.

Podsjetimo se, pri kreiranju `BP_MyCharacter` klase dodana joj je i *Skeletal Mesh* komponenta `Vivan_FaceMesh`. Poziv `GetMesh()` za `BP_MyCharacter` vraća upravo tu dodanu komponentu. Kad se dohvati kostur i model lica, prvo se zaustavlja bilo koja animacija koja se možda već izvodi nad kosturom pozivom funkcije `USkeletalMeshComponent::Stop`. Naime, želimo u svakom trenutku izvoditi samo jednu animaciju. Potom se postavlja jedan od mogućih načina animacije pozivom funkcije `USkeletalMeshComponent::SetAnimationMode`. Za kontrolu animacije moguće je koristiti i tzv. *Animation Blueprint* koji predstavlja skriptni opis logike okidanja animacija. Za potrebe pokretanja jedne po jedne animacije ipak se koristi način animacije

`EAnimationMode::AnimationSingleNode`. Sada je još jedino potrebno definirati animaciju koja će se izvoditi nad dohvaćenim kosturom. To se obavlja pozivom funkcije `USkeletalMeshComponent::SetAnimation` koja postavlja animaciju koja je proslijeđena kao argument funkcije `PlayAnimation`. Brzina kojom se animacija izvodi postavlja se pozivom funkcije `USkeletalMeshComponent::SetPlayRate`. Konačno, animacija se pokreće pomoću `USkeletalMeshComponent::Play`. Funkciji `Play` proslijeđen je argument `false` što označava da animaciju ne treba ponavljati, nego pustiti samo jednom.

Implementacijom ove funkcije zaključen je sustav za generiranje animacije govora. U Visual Studio odabirom opcije *Build* izgradi se programsko rješenje. Otvaranjem izgrađenog `SpeechCpp` projekta u Unreal Engine i pokretanjem igre vidljiva je scena prikazana Slikom 2.21. Rezultat je realistična animacija lica koje izgovara riječi koje se čuju na zvučnom zapisu u pozadini.

Slika 2.21: Prikaz modela u trenutku izgovaranja riječi

3. Rezultati

Konačna implementacija sustava nudi mogućnost generiranja 3D animacije izgovora bilo koje rečenice koja sadrži jednu od sedam ručno animiranih riječi. Za procjenu kvalitete rezultata razmatra se sedam različitih primjera zvučnih zapisa, odnosno animacija lica.

Ključnu ulogu u kvaliteti rezultata ima prva komponenta sustava – prepoznavanje govora i dobivanje ispravnih vremenskih trenutaka izgovaranja riječi. Čak i mala odstupanja u vremenskim trenutcima rezultiraju nerealističnom animacijom (npr. zvuk malo kasni za pokretima usta).

Iako Speech-to-Text API daje vrlo točne rezultate izgovorenih riječi, nažalost ne daje uvijek precizne vremenske trenutke izgovora. U nekim se slučajevima kao rezultat vraća obrnut redoslijed nekih riječi ili čak nemoguće kratko trajanje neke riječi. U primjeru Kôd 3.1 prikazan je dio rezultata u kojem je trajanje riječi *name* 0 sekundi što je, dakako, nemoguće.

```
...,
{
  "startTime": "2s",
  "endTime": "2.500s",
  "word": "my"
},
{
  "startTime": "2.500s",
  "endTime": "2.500s",
  "word": "name"
},
{
  "startTime": "2.500s",
  "endTime": "4.100s",
  "word": "is"
},
...
```

Kôd 3.1: Prikaz neispravnog odgovora Speech-to-Text API-ja

Zbog ovakvih grešaka neki su od odgovora Speech-to-Text API-ja ručno dorađeni kako bi za demonstraciju animacije imali ispravne vremenske trenutke.

Drugi ograničavajući faktor za kvalitetu konačne animacije predstavlja uređaj, odnosno hardver nad kojim se pokreće Unreal Engine. Kao što je već spomenuto, renderiranje

MetaHuman modela je računalno vrlo zahtjevan proces i zahtjeva jaku grafičku karticu i napredni procesor. Zbog toga u nekim trenutcima nakon pokretanja igre može doći do zastoja u renderiranju slike što također rezultira raskorakom u poklapanju zvuka i animacije. Nažalost, jedino rješenje za taj problem je korištenje jačeg računala za pokretanje alata.

Konačno, zadnji ključni utjecaj na kvalitetu rezultata imaju animacije pojedinih riječi. Iako je odabranih sedam riječi relativno kratko i jednostavno za animirati, jedan pogrešan *keyframe* u animaciji može znatno utjecati na uvjerljivost rezultata. Srećom, tom problemu se uvijek može doskočiti tako da se izradi nova i bolja animacija upitne riječi.

3.1. Moguće nadogradnje sustava

Iako sustav nudi prilično dobre rezultate na demonstracijskim primjerima koji koristi pretpostavljeni skup animiranih riječi, generiranje animacije iz novih zvučnih zapisa zahtijevalo bi *keyframe* animaciju svake nove riječi koja još nije animirana. To je, dakako, mnogo posla s obzirom na ogromnu količinu riječi koje se mogu pojaviti u zvučnim zapisima.

Možemo li na neki način izbjeći animiranje jedne po jedne riječi, nego iskoristiti neku manju gradivnu jedinicu govora za animaciju? Odgovor na ovo pitanje nametnuo se još u poglavlju *2.2. Animiranje pojedinih riječi*: pri animiranju pojedinih riječi kao pomoć se koristio njihov fonemski zapis.

Dakle, umjesto da se pušta animacija riječi u nekom trenutku, tu riječ bi se pretvorilo u fonemski zapis (npr. koristeći prije spomenuti konverter u fonemski zapis). Na temelju pojedinog fonema učitali bi se njihovi *keyframeovi* koji bi zajedno činili animaciju riječi. Pozitivna strana ovog pristupa je činjenica da postoji tek oko 160 fonetičkih simbola u IPA (engl. *International Phonetic Alphabet* – internacionalna fonetička abeceda), a od njih se u engleskom jeziku koristi oko 40 [17] [18]. To znači mnogo manje ručnog i dugotrajnog posla s animacijama. Ipak, jedan po jedan *keyframe* trebalo bi na neki način spojiti u jednu kontinuiranu animaciju riječi (bez skokova ili prekida). Srećom, Unreal Engine nudi mehanizam zvan *Blend Space* za stvaranje tečne animacije interpolirajući odabrane *keyframeove* [19].

Također, sustav je moguće dodatno automatizirati tako da se ugradi HTTP poziv Speech-to-Text API-ja direktno u C++ implementaciju. Postoje mnoge C++ implementacije biblioteka koje omogućavaju jednostavno stvaranje *curl* poziva i obradu njihovih odgovora. Osim

poziva Speech-to-Text API-ja, na isti način bi se mogao ugraditi i poziv prema API-ju koji vrši pretvorbu teksta u njegov fonemski zapis. U sklopu ovog rada nisu ugrađeni HTTP pozivi u kod zbog optimizacije (HTTP poziv dodatno bi usporio generiranje animacije), ali i zbog nedovoljne kvalitete odgovora koje API za prepoznavanje govora trenutno daje.

Zaključak

U području računalne animacije likova, animacija lica jedan je od najkompleksnijih i najvažnijih dijelova za konačno stvaranje uvjerljivog rezultata. Cilj sustava zamišljenog u ovom radu bio je olakšati proces animacije i implementirati sučelje koje će iz zvuka govora i kratkih animacija samo moći generirati kompleksniju animaciju govora.

Rezultat rada je kompletan sustav za generiranje realistične 3D animacije lica iz zvučnog zapisa govora. Demonstrirane su mogućnosti generiranja animacija izgovora rečenica koristeći skup od sedam riječi. Navedeni skup dalje je moguće relativno jednostavno proširiti dodajući nove animacije riječi, nove zvučne zapise i uz manje izmjene u samoj implementaciji. Ovakav sustav mogao bi naći primjenu u videoigrama u kojima treba npr. generirati filmske sekvence govora likova.

Glavna moguća poboljšanja sustava uključuju dodatnu automatizaciju uvođenjem pozivanja API-ja za prepoznavanje govora iz C++ implementacije te poopćenje sustava tako da se umjesto riječi animiraju pojedini fonemi. Druga od navedenih nadogradnja zahtjeva nešto više truda, ali bi u konačnici rezultirala sustavom koji doista može generirati animaciju iz zvučnog zapisa bilo koje rečenice. Također, daljnji napredci u modelima strojnog učenja za prepoznavanje govora s vremenom će dovesti do sve preciznijih informacija o izgovorenim riječima, a samim time i do sve uvjerljivije animacije lica.

Literatura

[1] Parke, F. I., Waters, K. *Computer Facial Animation*. 2. izdanje. Massachusetts: Wellesley, 2008.

[2] Tech Target Contributor, *Uncanny valley*, (2016, veljača). Poveznica: https://www.techtarget.com/whatis/definition/uncanny-valley; pristupljeno 8. lipnja 2022.

[3] Google, *Speech-to-Text*, (2018). Poveznica: https://cloud.google.com/speech-to-text; pristupljeno 1. lipnja 2022.

[4] Epic Games, *Unreal Engine*. Poveznica: https://www.unrealengine.com/en-US; pristupljeno 6. lipnja 2022.

[5] *List of Unreal Engine games*. Poveznica: https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games; pristupljeno 7. lipnja 2022.

[6] Epic Games, *Unreal Engine 5 Documentation*, (2022, travanj). Poveznica: https://docs.unrealengine.com/5.0/en-US/; pristupljeno 8. lipnja 2022.

[7] Bergsman, T., Azim, W., *Quixel Bridge*. Poveznica: https://quixel.com/bridge; pristupljeno 29. svibnja 2022.

[8] Epic Games, *Unreal Engine 4.27 Documentation: Metahumans*. Poveznica: https://docs.unrealengine.com/4.27/en-US/Resources/Showcases/MetaHumans/; pristupljeno 9. lipnja 2022.

[9] Google, *Speech-to-Text: Supported audio encodings*. Poveznica: https://cloud.google.com/speech-to-text/docs/encoding#audio-encodings; pristupljeno 9. lipnja 2022.

[10] Google, *Cloud Storage*. Poveznica: https://cloud.google.com/storage; pristupljeno 9. lipnja 2022.

[11] Google, *Resource Manager: Creating and managing projects*. Poveznica: https://cloud.google.com/resource-manager/docs/creating-managing-projects; pristupljeno 10. lipnja 2022.

[12] Google, *Cloud Storage: Creating storage buckets*. Poveznica: https://cloud.google.com/storage/docs/creating-buckets; pristupljeno 10. lipnja 2022.

[13] Google, *Install the gcloud CLI*. Poveznica: https://cloud.google.com/sdk/docs/install; pristupljeno 10. lipnja 2022.

[14] Adobe, *Keyframe animation for beginners*. Poveznica: https://www.adobe.com/creativecloud/video/discover/keyframing.html; pristupljeno 10. lipnja 2022.

[15] *toPhonetics*. Poveznica: https://tophonetics.com/; pristupljeno 10. lipnja 2022.

[16] Lohmann, N., *JSON for modern C++*. Poveznica: https://github.com/nlohmann/json; pristupljeno 9. lipnja 2022.

[17] *International Phonetic Alphabet*. Poveznica: https://en.wikipedia.org/wiki/International_Phonetic_Alphabet; pristupljeno 12. lipnja 2022.

[18] The Reading Well, *The 44 Phonemes in English*. Poveznica: https://www.dyslexia-reading-well.com/44-phonemes-in-english.html; pristupljeno 12. lipnja 2022.

[19] Epic Games, *Unreal Engine 4.27 Documentation: Blend Spaces*. Poveznica: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/Blendspaces/; pristupljleno 12. lipnja 2022.

Sažetak

Animacija ljudskog lica

U okviru ovog rada implementiran je sustav za generiranje 3D računalne animacije lica u alatu Unreal Engine. Iz zvuka govora primjenom metoda strojnog učenja obavljeno je prepoznavanje govora. Iz prepoznatog govora stvoren je tekstualni zapis na temelju kojeg se generirala animacija. Opisane su glavne značajke i korištenje Google Speech-to-Text API-ja za prepoznavanje govora. Animacije se izvode nad realističnim MetaHuman modelom dostupnom unutar Unreal Engine. Detaljno je objašnjena implementacija u programskom jeziku C++ kao i glavne funkcije potrebne za ostvarenje animacije. Na kraju su razmotreni rezultati i moguće nadogradnje i poboljšanja.

Ključne riječi: animacija lica, prepoznavanje govora, Speech-to-Text, Unreal Engine, C++

Summary

Human Face Animation

This paper describes the implementation of a system for generating 3D computer animation of faces in the Unreal Engine tool. Speech recognition was performed based on the sound of speech using machine learning methods. A text record was created from the recognized speech, based on which the animation was generated. The main features and use of the Google Speech-to-Text API for speech recognition are described. The animations are performed over a realistic MetaHuman model available within Unreal Engine. The implementation in the C ++ programming language is explained in detail, as well as the main functions required for the realization of the animation. Finally, the results and possible upgrades and improvements are discussed.

Keywords: human face animation, speech recognition, Speech-to-Text, Unreal Engine, C++