LiU-ITN-TEK-A--24/003-SE

The State of Live Facial Puppetry in Online Entertainment

Lisa Gren

Denny Lindberg

2024-01-31

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--24/003-SE

The State of Live Facial Puppetry in Online Entertainment

The thesis work carried out in Medieteknik

Lisa Gren
Denny Lindberg

Norrköping 2024-01-31

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

Abstract

Avatars are used more and more in online communication, in both games and social media. At the same time technology for facial puppetry, where expressions of the user are transferred to the avatar, has developed rapidly. Why is it that facial puppetry, despite this, is conspicuous by its absence?

This thesis analyzes the available and upcoming solutions for facial puppetry, if a common framework or library can exist and what can be done to simplify the process for developers who wants to implement facial puppetry.

A survey was conducted to get a better understanding of the technology. It showed that there is no standard yet for how to describe facial expressions, but part of the market is converging towards a common format. It also showed that there is no existing interface that can handle communication with tracking devices or translation between different expression formats.

Several prototypes for recording and streaming facial expression data from different sources were implemented as a practical test. This was done to evaluate the complexity of implementing real-time facial puppetry. It showed that it is not always possible to integrate the available tracking solutions into an existing project. When integration was possible it required a lot of work. The best way to get tracking right now seems to be to implement a standalone program for tracking that streams the tracked data to the main application.

In summary it is the poor integrability of the solutions that makes it problematic for the developers, together with a wide variety of facial expression formats. A software that acts like a bridge between the tracking solutions and the game could allow for translation between different formats and simplify implementation of support.

In the future, instead of working towards making all tracking solutions output standardized tracking data, research further how to build a framework that can handle different configurations.

Contents

List of Figures

List of Tables

1 Introduction

The avatar has become an integral part of the lives of many internet users. It is the virtual version of their online presence on websites, games, VR applications, conference calls and live streaming services. For some, the avatar allows them to express themselves in a way that would not be possible in the real world. Being able to present and articulate their avatar live is fundamental to their online experience.

The avatars within these virtual worlds get more sophisticated by each year. Some of the applications offer motion capture to make the avatar mimic the facial expressions and body motions of the user, which is a form of puppetry. Sometimes the avatar simulates speech by moving the mouth based on the voice of the user. Despite the advancements in technology and increased interest in motion capture, it is still a complex issue for developers to implement support for such a feature, especially when it comes to capturing face expressions and applying them to avatars.

1.1 Motivation

The need for real-time facial puppetry arose while Lindberg, one of the thesis authors, was working on Mortal Online 2[1]. The goal was to let players capture their face in real-time and enabling them to express themselves in game to other players. The character setup and network communication were ready for such a feature, but the available options for capturing the expressions of a user were either lacking or put unnecessary hardware requirements on the end user. The amount of work needed to get any usable result, more than a basic prototype, was beyond what the schedule allowed. The question arose why there was no fundamental library or framework available for capturing face expressions on desktop devices, considering that face related apps on mobile devices had become such a widespread phenomenon.

While several related subjects will be covered in this thesis the main area of interest remains within facial expression capture and head tracking. Facial puppetry, motion capture, head tracking, expression estimation and non-rigid tracking are a few examples of the nomenclature used to describe the area of interest.

[1]This thesis was conducted independently from Star Vault AB.

1.2 Methods

A survey was needed to evaluate existing hardware requirements, methods, solutions, trends and standards. From facial tracking to receiving output data, like landmarks and expression coefficients. From a developer perspective the goal was to determine how to implement a solution that players would like. From a scientific perspective it was necessary to build a deeper understanding of the technology. Furthermore we decided to implement various test applications to evaluate the complexity of implementing facial puppetry, and to figure out how an application can be built to simplify the integration of facial puppetry into other applications.

1.3 Aim

This thesis will research if there exists a common framework or library for game developers to access standardized face capture data formats from various sources, or if such a thing could possibly exist. Regardless of if a user has a web camera, a VR/AR device, or some other motion capture device, the data should ideally be accessible in one or multiple common data formats that the developer can rely on without risk of near-future deprecation. Perhaps there could be a way to transform data using an intermediate canonical form to support new formats. Research will determine which solutions exist for facial expression estimation, which capture devices are available, what data formats the expressions are defined in, if there is homogeneity between them, if a common framework can be made and how a proposed software architecture could look like for transferring data between devices and applications. The aim is to determine if and how it would be possible to simplify the development process for those that want to capture and display facial expressions of players in their applications.

1.4 Research questions

1. What challenges appear when developers decide to implement facial puppetry? Why is this type of motion capture not more commonly used? Is it a lack of time, resources, priorities or lack of expertise?
2. What are the fundamentals needed for face motion capture and what methods exists? Which methods are currently relevant for developers?
3. What data structures are used in existing methods of face motion capture and can these data structures be transformed to a homogenized structure? Does a structure already exist or is there a way to standardize the data?
4. What methods are suitable for sending a stream of motion data between devices or processes? What are the common approaches for streaming audio, image or tracking data in real time?
5. How can the software architecture be designed, based on existing methods for facial motion capture and animation, to simplify the process for developers who want to implement the feature in their application?

1.5 Delimitations

Since we do not have access to all consumer grade software and hardware solutions studied in this thesis we will only look at the associated API's and documentation for solutions we do not have access to. In implementation tests we will only work with placeholder data for the unavailable solutions.

2 Theory about Characters in Game Development

Facial puppetry is made possible by driving a virtual character using live data. To understand why the transfer of facial expressions from a person to a virtual character can be complicated, it is important to understand how characters are usually built and driven. A summary follows for related methodologies such as skeletal deformations and blendshapes and how those are used to simulate body motions and facial expressions to achieve facial puppetry.

2.1 Simulating Real-Time Virtual Characters

Characters in computer games appear in two major categories, 3D and 2D. A 3D character is mainly built as a skeletal mesh, a polygonal structure that is simulated with a hierarchy of joints and deformed using linear skin weights. Characters in the 2D category are often an animated sequence of frames, be it drawn, pixel-art or some layered composite. Pseudo-2D (2.5D) characters are built using multiple flat mesh layers in 3D that are deformed and animated just like the 3D counterpart. They just appear 2D during rendering. Figure 2.1 shows a 3D and 2D example of two characters.

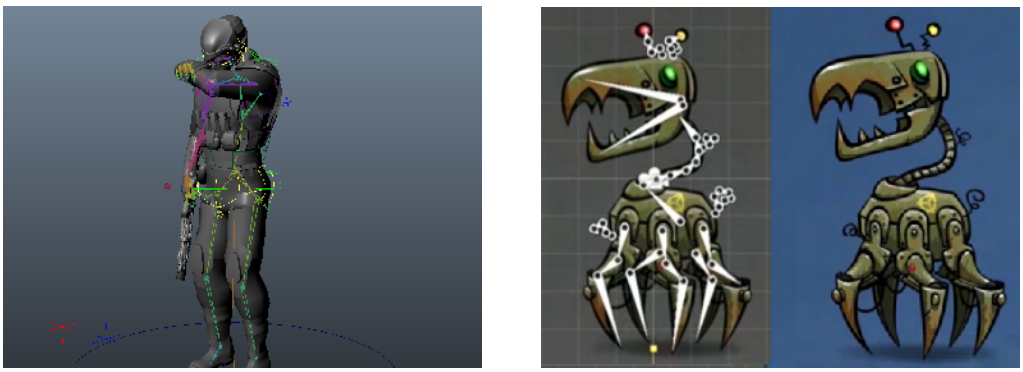

Figure 2.1: Examples of 3D and 2D characters from *Project Stealth* and *Youtube - Bringing 2D characters to life with sprite rigging - Unite Copenhagen*

The mentioned techniques are a recurring standard in the industry and are widely used in popular game engines like Unity 3D, Unreal Engine and Godot. The literature on real-time rendering [1, 2] explains the technical details but is not a prerequisite for creating characters. The mentioned game engines and 3D applications like Blender include artistic tools and asset pipelines that hide the underlying complexity.

2.2 Range of Expressions

People articulate and expresses themselves using both body language, facial expressions and voice. It is said that there are a set of universal emotions: neutral, anger, disgust, fear, surprise, happiness, sadness, and contempt. Ekman et al. [3] have done multiple studies on the subject. Facial expressions are encoded by Action Units in the Facial Action Coding System (FACS) by Hjortsjö [4] and Ekman and Friesen [5]. Ozel [6] shares several animated examples visualizing the descriptors, one example can be seen in Figure 2.2. While most video game characters have a range of expressions, they tend to be limited within the eyes and mask region. A fully expressed head needs to also include the less obvious peripheral regions like the neck, ears, scalp and hair. The most obvious being the platysma muscle that covers the neck and becomes visible under stress.

A simulated character needs to fulfill a base set of requirements to achieve a fundamental expression set. It must be able to rotate its head, open its mouth, look around with its eyes, and be able to express some of the mentioned universal emotions. Wrinkles and peripheral details are optional nice-to-haves to strengthen those expressions.

Figure 2.2: The AU1 action descriptor from FACS demonstrated by Ozel [6]

2.3 Rigging and Deformation

Real-time characters can be split into the following components: mesh, texture, skeleton, deformation, animation, sound, physics simulation and GPU shaders for rendering. A character structure can be conceptually split into the body and head. This is done for both practical and runtime performance reasons, as a fully expressed face is complex to simulate. Refer to Orvalho et al. [7] for an in depth face rigging survey. Rigging refers to the process of building a runtime skeletal structure that deforms a mesh. It can also refer to creating a set of extra controls on top of a skeletal structure, that helps animators pose and keyframe a character. The neutral pose of a mesh, where a skeleton applies no deformation, is referred to as the *bind pose*.

Osipa [8] and Orvalho et al. [7] covers industry standard rigging techniques and good animation practices for the head and face. Transformation structures and deformers like joints (bones), Free-Form Deformers (FFD), blendshapes and physically based structures are mentioned. All four are relevant in game development but skeletal deformation using bones and blendshapes are the most common. There are more deformers like Delta Mush [9], machine learning deformers [10], and GPU surface shader techniques like wrinkle maps [11].

2.4 Blendshapes

Blendshapes belong to the category of per-vertex animation. They are sometimes called morph targets, blend keys or shape interpolations. The non-deformed mesh in the bind pose is referred to as the neutral blendshape $\boldsymbol{b}_0$. Additional blendshapes $\boldsymbol{b}_k$ are created by deforming the neutral shape and computing difference vectors $\boldsymbol{d}_k = \boldsymbol{b}_k - \boldsymbol{b}_0$ (where $k = 1..N$). The difference is sometimes referred to as a delta shape. The blendshape should preferably include the modified tangent and normal to improve shading. A fully expressed face can then be simulated as a linear combination of these shapes on top of the neutral pose. The coefficient for each shape, sometimes called blendshape factor or strength value, is often a $[0, 1]$ floating point value to blend the shape (hence the name). The value is technically not limited to that range but is often done to maintain control over the deformation range. For a more formal definition of blendshapes see Möller et al. [1] or Nguyen et al. [2]. For a deep dive on blendshapes for facial models see the report by Lewis et al. [12].

As the blendshape coefficients are individual floating point values it is possible to animate them using curves, keyframes or other procedural means. The coefficients are equivalent to facial expression coefficients when the blendshapes are shaped as Action Units from FACS.

2.4.1 Pros and Cons

Blendshapes have a downside. As they are a combination of linear interpolations the vertices are only translating between shapes. They do not rotate in an arc. This is why the eyes and jaw are commonly rigged using joints to support rotations around various pivots. Expressing all possible orientations with a single joint, e.g. for the eyeball, is more efficient and correct than doing a large set of blend shapes. It is still possible to pose these parts using blendshapes due to their angle of rotations being small, but they will still deform slightly due to the linear interpolations.

Arguably the greatest strength of blendshapes is that characters of completely different shapes and styles can be animated using the same facial expression coefficients. This allows for characters ranging from realistic humans to cartoony animals to be driven with the same coefficient values. Figure 2.3 shows a character posed with blendshapes using expression coefficients. Figure 2.4 shows how Epic Games is using blendshapes to animate different characters in Fortnite using the same expression coefficients from an animation sequence.

Figure 2.3: A face using a mix of bones and blendshapes where the values in the image represent the blendshape coefficients - from the Unreal Engine 5.1 documentation pages.

Figure 2.4: Three Fortnite characters having the same expression coefficients, resulting in the same facial expression due to the advantage of blendshapes - from Unreal Engine 5.2 documentation pages [13].

Setting up blendshapes for characters can be a tedious process and is often done by professionals in the industry. The reader can refer to the online course by Schnapp [14] for an in-depth workflow. Figure 2.5 shows an example of how different characters can make use of the same blendshapes from a source mesh with proper transfer of data.

Figure 2.5: Multiple characters use blendshapes from a single source mesh by a blendshape transfer workflow - from the CGCircuit course by Schnapp [14].

2.5 Animation and Motion Capture

A character can be animated in many ways. The structure is a set of transforms and the individual floating point values can be driven by different means. Keyframing, the act of manually posing and keying a character, is used to animate a character by letting the computer interpolate the inbetween frames. Keyframing is a process that demands artistic skill and is typically done by using an animation rig on top of the deformation rig as shown in Figure 2.6. Software like Cascadeur [15] can assist animators in the keyframing process to simulate physically accurate motions. For realistic animation one can resort to motion capture, which is a method to record the performance of a human, animal, or prop, and retarget

its motion to a character rig. Motion capture often demands a cleanup or adjustment pass by an animator.

A set of animation frames is referred to as an animation sequence. Animations can be mixed by for example playing separate sequences on the head and body during a dialog shot. Sequences can also be mixed using interpolations, for example during a transition between a walk and a run cycle.

A form of motion capture is now available in mobile consumer devices. We see it in the form of virtual props or face filters being applied to the subject in a video feed. There exists expression transfer (a.k.a. facial puppetry) in some mobile device frameworks, but the availability of such features in PC games are still lacking. A few notable PC games that enables a player to use a camera to express their character are EverQuest II [16], Star Citizen [17] and VR Chat [18]. Some of those solutions are however proprietary or outsourced and are only used internally.

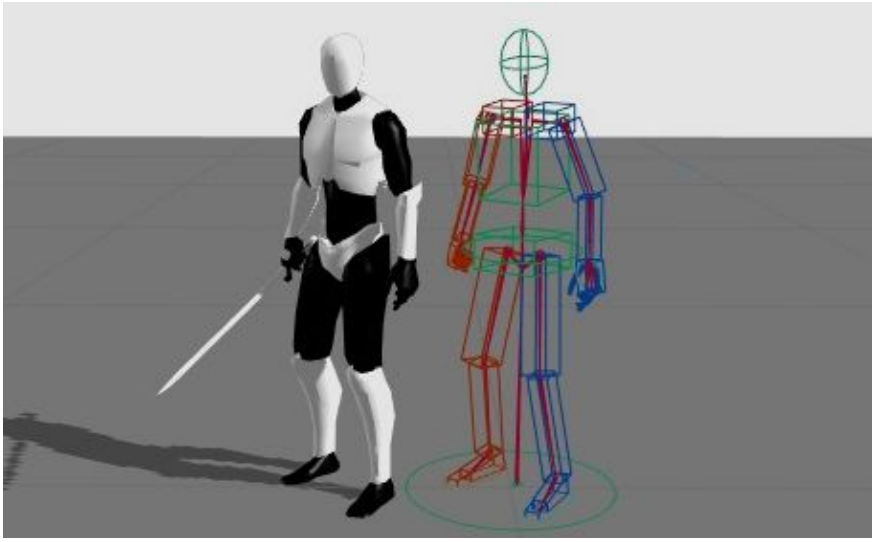

Figure 2.6: A character for prototyping from a personal project by Lindberg with its skeleton in red and controls in red, green and blue. The controls are used to assist an animator to pose and keyframe the character.

2.6 The Uncanny Valley

The term *uncanny valley* was coined by Masahiro Mori in 1970 and is explained in the foreword in the translated report by Mori et. al. [19]. Paraphrased it reads: a person's response is hypothesized to abruptly shift from empathy to repulsion when a robot approaches, but fails, to attain a lifelike appearance. This eeriness is known as the *uncanny valley*.

The uncanny valley is a known issue in both the movie and game industries when developers attempt to achieve realism. It can become an issue when motion capture is applied to a simulated character. Instead of chasing the highest fidelity possible in a character simulation, and running the risk of making a character unappealing, it can instead be a good idea to reduce fidelity and exaggerate motions to intentionally get a more cartoony effect to ensure an appealing character. This is important to keep in mind when working with facial puppetry, the disconnect of applying hyper-realistic motion onto a stylized appearance can make the character creepy.

3 Survey of Face Tracking Solutions

A survey was conducted in order to discover complexities related to implementing facial puppetry in games. The goal was to determine what solutions are available for developers, what methodologies those solutions are built upon, and what standards and commonalities there might be.

3.1 Survey Method

We used Google's search engine. First we searched for available market solutions, using keywords such as "Face Tracking in Games" and "Live Face Motion Capture Avatar". The results were stored in a shared list where we also noted information about the software such as user base or output type. After this we searched for research papers using keywords like "Facial Expression" together with "Tracking", "Transfer", "Cloning" and "Coefficients". We clicked on results that seemed relevant and excluded results with titles indicating that the report concerned finding and tracking a face over consecutive frames in a video. We extracted the information by reading through the reports individually as we found them. We decided beforehand who should focus on which research area in order to not end up finding the same reports. Sometimes we also split the work so one person worked on the prototype or report while the other did research. All reports were stored in a shared library. When we found a report that was significant we sent it to the other person so both could read it. The content of the other reports were briefly shared with the other person at the end of each day, when we also discussed our findings. As we found new relevant terms we altered our searches to include those words.

3.2 Hardware

A range of devices exist, from single cameras to advanced multi-sensor array layouts. Most consumers have access to standard RGB color channel cameras that can stream video between 24-60 FPS (web cameras, smartphone cameras, DLSR etc). Most of these cameras lack a depth channel, which is needed to increase robustness in some computer vision algorithms. A stereo camera pair, a sensor array or machine learning algorithms can estimate depth information when a depth sensor is missing [20].

Cameras that include a depth sensor are sometimes referred to as having RGBD color channels, where D stands for depth. Depth sensors were initially popularized by the now discontinued Microsoft Kinect [21]. The Azure Kinect DX [22] is its successor that is mainly used for AI and cloud computing. Intel manufactures their RealSense [23] cameras that are sometimes embedded in laptops. The most commonly used depth sensors exist in Apple's iOS devices with their TrueDepth [24] cameras.

Infrared cameras are used in motion capture and relies on markers attached to the subject. The Tobii [25] markerless eye tracker exists as a separate device or embedded in some VR headsets. Eye trackers in VR headsets are often used for implementing foveated rendering [26]. Ultraleap offers the Leap Motion Controller [27] but it only tracks hands.

The HTC Vive Face Tracker [28] is a VR headset add-on that tracks facial expressions in the mask region which excludes eyes and eyebrow movements. It can be used without a VR headset. The Meta Quest Pro [29] is a VR headset that includes both facial expression and eye tracking abilities. Other headsets exist but none of which focuses on facial expressions.

3.3 Software

Tracking and pose estimation relies on a mix of computer vision and machine learning algorithms. Refer to Szeliski [20] for fundamentals on the subject. There is a whole market of tools for face detection, tracking, pose estimation and facial puppetry. The list is filtered to exclude solutions not particularly useful in facial puppetry.

Developers that have the expertise to implement computer vision libraries can use OpenCV [30], dlib [31], CI2CV Face Analysis SDK [32], Google MediaPipe [33] and Google ARCore [34]. These have varying levels of support for face tracking and expression estimation. Tensorflow [35], Pytorch [36] and scikit-learn [37] are related machine learning libraries.

Dedicated hardware such as the HTC Vive Face Tracker, Meta Quest Pro, Tobii Eyetracker and Apple's TrueDepth cameras come with existing tracking and pose estimation solutions. HTC and Tobii have their SDK's [38, 39] and Apple has the ARKit framework [40]. The NVIDIA Maxine SDK [41] allows NVIDIA RTX GPU owners to use their webcamera for tracking facial expressions, this is also included in the NVIDIA Broadcast AR SDK [42]. These are ready-to-go but vendor-locked solutions that developers can use.

Meta has voice-based system with Oculus Lipsync [43] which generates viseme blendshapes; visual mouth shapes that are resolved from vocal phonemes through real-time audio recording.

A subset of devices exist that only tracks the head for free-look mechanics in games: TrackIR [44], OpenTrack [45], AITrack [46] and FaceNoIR [47].

Commercial or closed source products supporting RGB or RGBD cameras are Avatary [48], Visage Technologies [49], Banuba [50], MoodMe [51], Luxand [52], Faceware Studio [53] (used in Star Citizen [17]), Animaze by FaceRig [54], MocapForAll [55], VTubeStudio [56], VNyan [57], Warudo [58], Brekel Face [59] and iFacialMocap [60].

Open source projects are DeepFace [61], FaceTracker [62], MeFaMo [63] and OpenSeeFace [64]. OpenSeeFace is the most complete open source solution.

3.3.1 Derivatives of Apple ARKit

Apple acquired both FaceShift [65] and PrimeSense [66], both which likely contributed directly to the iOS face tracking features. The ARKit SDK [40] supports both face landmarks and 52 blendshape coefficients for facial expressions. These features have resulted in several derivative solutions for face tracking. Notable examples are Rokoko Face Capture [67], iClone Motion LIVE [68], MocapX [69], FaceCap [70, 71], FaceIt Blender Plugin [72], iFacialMocap [60] and Epic Game's LiveLink Face [73] for MetaHumans [74]. Some of the apps send data from the iOS device to a workstation over the network.

The model name Perfect Sync appears in some VTuber applications [75, 76]. The name refers to a character setup that supports all 52 blendshape coefficients defined in ARKit. A website exists [77] as a visual reference for sculpting the blendshapes for a character so that it can achieve Perfect Sync. There are services like Polywink [78] that can generate these shapes.

3.3.2 Derivatives of Google MediaPipe

Google MediaPipe recently added a model for solving ARKit blendshape coefficients from its own landmarks. MediaPipe has given rise to several derivatives. KalidoKit [79], Hallway [80], and PhizMocap [81] are web browser applications that access the web camera and stream tracking data to other applications over WebSockets. KalidoFace [82] uses KalidoKit. BlendArMocap [83] is a Blender plugin. ProjectGameFace [84] is a project for using facial expressions as input data.

3.3.3 Derivatives of NVIDIA Broadcast

While NVIDIA Broadcast [41, 42] is still new compared to Apple ARKit and Google MediaPipe, it might gain traction due to NVIDIA's GPU market share. At the moment of writing we could only find iFacialMocap [60] as a derivative on the Windows Store.

3.4 Commonalities and Existing Standards

For code libraries in the VR space there is the Virtual Reality Peripheral Network (VRPN) [85], a device-independent library for virtual reality peripherals. Open Source Virtual Reality (OSVR) [86] extends upon VRPN with more features. Both have a large set of supported peripherals and use the VRPN network interface. Despite their feature sets these libraries do not seem to be used much within game development. More popularized API's exist such as SteamVR [87], OpenVR [88], and Windows Mixed Reality (WMR) [89], that are congregating to be replaced by the Khronos OpenXR API standard [90]. Despite all these API's having wide peripheral support there is little mention of facial expression support. OpenXR has optional extensions for the HTC Vive and Meta Quest Pro face trackers under *XR_HTC_facial_tracking* and *XR_FB_face_tracking*. Both seem to follow their own standard. There is at least the *XR_EXT_eye_gaze_interaction* extension for eye tracking. The OpenXR extensions are a starting point but there is no standard available yet for facial expressions. The OpenXR Toolkit [91] is an OpenXR API layer that adds extra functionality between OpenXR and the target application. It does not have any facial expression related features except from eye tracking.

3.4.1 OSC and VMC

The Virtual YouTuber community (VTubers for short) have managed to build their own niche of tools for animating their avatars, using various software and devices. Two recurrent network protocols are Open Sound Control (OSC) [92] and Virtual Motion Capture (VMC) [93], that has organically become standards for the community and is used in VR Chat [18] and Animaze. VR Chat has open network protocols for animating different parts of characters in the game. It can for example use data directly from a VIVE Facetracker or the the Meta Quest Pro. Data is streamed from and to VR Chat over OSC using applications like VSeeFace [94]. More about OSC in Section 5.7.

VRCFaceTracking (VRCFT) [95] is a program that can receive blendshape coefficients from different devices, and convert the coefficients with a module system to its own standardized set of Unified Expression coefficients [96]. It then sends the coefficients over the network using OSC to VRChat. VRCFTtoVMCP [97] is a program that intercepts the OSC data from

VRCFT and forwards it over VMC if the user does not intend to use VR Chat. A complete list of VMC related software can be found on the VMC reference page [98].

3.4.2 Commonalities Growing into Standards

There are some commonalities that are organically growing into standards. The network protocols OSC and VMC are recurring when it comes to sending data. The 52 blendshape coefficients in Apple's ARKit, sometimes called Perfect Sync, also occur often. Google's MediaPipe and Nvidia's Maxine supports an approximation of the ARKit coefficients. Apart from the blendshape coefficients it does not seem as if there is any homogeneity for face landmarks or face meshes. The translation and transfer of data between standards and applications only happens at the blendshape coefficient level.

4 Common Methodologies Found in the Survey

We identified a large variety of solutions in the survey, ranging from head-only tracking to extraction of facial features and expression coefficients. The findings in the survey were studied further to determine possible commonalities and standards that could ease the adoption of facial puppetry. The methods in the survey used a mix of computer vision and machine learning algorithms. Some of the methods used physical markers for tracking in the infrared spectrum, or additional devices like inertia sensors, but still used similar algorithms to solve the pose.

4.1 A Typical Expression Transfer Algorithm

We found that the algorithms typically started with face detection to identify what region of an image occupied a face. Tracking was then used to follow an individual over a sequence of frames. Facial features were detected within the tracked region and represented as individual points, like the corners of the mouth, and are called landmarks. Expression, mood or identity can be determined directly from the landmarks, but it was common to first fit a canonical face mesh to the landmarks. The canonical mesh is used to ensure the values are invariant to the uniqueness of a face. It was common to calculate some kind of expression coefficients from the canonical face or the feature points. After this the expression coefficients were used to drive the blendshapes of an avatar.

4.1.1 Detection and Head Tracking

The first step, face tracking, is when the face of a subject is identified and followed in a series of images. Pose estimation of a head can be solved already in this step but can be improved with additional models. For example Bazarevsky et al. [99] claims that their BlazeFace face detector does both in as little as sub-millisecond performance on 2019 flagship devices. Google claims in their model info cards [100] that their face detection and landmarker models based on BlazeFace are fair across regions, gender and skin tones.

4.1.2 Face Landmarks

When a face was found it was common to detect landmarks. Landmarks correspond to surface points on the face, such as the corners of the mouth or eyes. These are either defined as 2D or 3D coordinates. Coordinates vary between spaces, ranging from image to world space, and can be expressed in pixels, normalized values, or distance units like meters. Not all methods in the survey detected landmarks. For those that did, landmarks were used as the basis for establishing an identity of a person and estimating their facial expressions.

A predefined set of landmarks are called a landmark configuration or scheme. We found a multitude of landmark configurations. For example the Multi-PIE [101] data set defines 68 landmarks that map to an individual's face. Other examples are BlazeFace [99] that solves 6 landmarks, NVIDIA's Maxine [41] with either the Multi-PIE 68 or their own 126 point set and MediaPipe landmarker [100] with 478 landmarks by Grishchenko et al. [102]. Landmark configurations change for each method and there is no standard landmark configuration. The points in each configuration have semantically different locations on the face. For a comprehensive overview on landmarks and databases see the surveys [103, 104, 105]. Some work has been done in an attempt to convert, transfer or standardize the configurations (see Bulat and Tzimiropoulos [106] or Sagonas et al. [107]). Chandran et al. [108] presented a landmark detector that used a 3D canonical face mesh as a template during prediction to allow any landmark configuration without retraining the network.

Zoss et al. [109, 110] shows in their empirical jaw rig that landmarks are sometimes not enough to represent anatomically correct motion. Landmarks are points on the surface level of the skin which slides over bones, this results in an inaccurate pose estimation of the jaw.

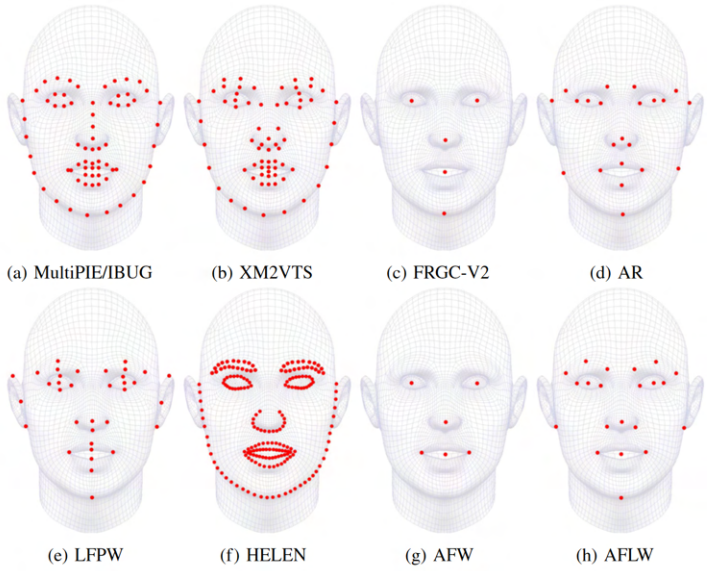

Figure 4.1: *Fig 1.* from the report by Sagonas et al. [107] showing landmark configurations from existing databases

4.1.3 Canonical Face Model

It was common to use a canonical face model to increase robustness during tracking. A canonical face model refers to a standardized, idealized or normalized model of a human face. It can be seen as a predefined mesh template. Based on the canonical faces seen during the study, it seems the shape of the canonical model is not important, as long as it is in a neutral pose and can be sampled properly. It serves multiple purposes in face tracking. It can aid in normalizing facial variances, make a model more invariant to uniqueness (shape, lighting, camera angle), help estimating landmark locations when part of the face is occluded, be used to extract facial features and to determine both rigid and non-rigid poses. Both Chandran et al. [108] and Cao et al. [111] show the benefits of using a canonical face model to increase the robustness of their algorithms. Colbry and Stockman [112] show how a 3D face scan from various angles can be defined in a normalized canonical face depth map. One of the earlier models for a canonical face is the CANDIDE face model by Rydfalk in 1987 and later updated to CANDIDE-3 by Ahlberg in 2001 [113].

Common Canonical Face Workflow

A brief overview of a face tracking process involving a canonical face can be found on MediaPipe's face mesh documentation pages [114]. For the actual methodology see the paper by Grishchenko et al. [102]. In this process, most effects related to the canonical face is done in metric 3D space, using their face transform module. The effects are then projected back into the original camera view. The effect renderer component is used to apply the visual effects to the user. The process relies on UV mapping to apply textures to the canonical mesh. This is a workflow often used by mobile apps that apply face filters and is not exclusive to MediaPipe. Some canonical face meshes are available either through MediaPipe's GitHub repository [115], Apple's AR example files [116] or by installing NVIDIA's Broadcast AR SDK [42].

Canonical Faces and UV Coordinates

UV coordinates can be useful for transferring properties between meshes that do not share the same vertex ordering, topology, shape or detail level. Figure 4.2 shows the UV map for the canonical face from MediaPipe.

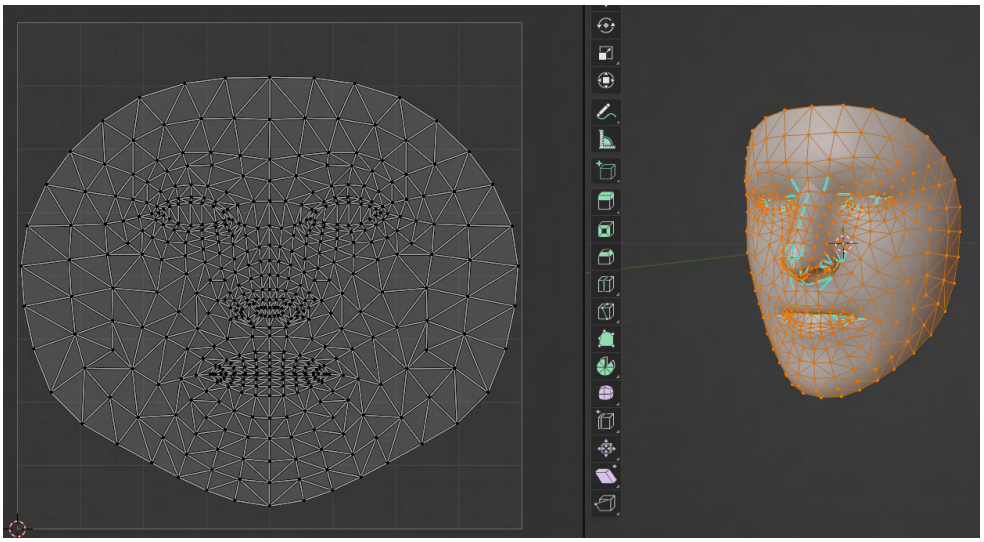

Figure 4.2: The canonical face mesh from MediaPipe's GitHub repository [115] showing existing UV coordinates when opened in Blender 3.6

The UV space can be seen as a shared space between unrelated meshes given that the UV layout is similar. Figure 4.3 shows how a shared UV layout can be used to apply the same tattoo across different faces. Chandran et al. [108] demonstrated how using a canonical face mesh in their landmark predictor enabled them to use a UV layout to apply artistic effects to faces in images, regardless of identity or expression. They also suggested that the canonical head shape for landmark prediction could be defined in the UV space for future work.

Figure 4.3: A shared UV layout can be used to apply the same face tattoo on different faces, a technique common in video games and here shown in the character creation screen from Mortal Online 2

Deformable Model Fitting

Deformable Model Fitting (DMF) was sometimes used to fit the canonical face mesh to the subject. DMF is a technique for reconstructing geometry by fitting an existing mathematical shape or template mesh to target data, such as feature points. Szeliski [20] covers various reconstruction techniques including image-based modeling for fitting a generic head model to a face, by Blanz and Vetter [117]. Both touch on the subject of using Principal Component Analysis (PCA) and Active Shape Model (ASM) to either do face recognition or shape analysis. Saragih et al. demonstrates an implementation of DMF by regularized landmark mean-shift [118].

Mesh Reconstruction

The canonical face is often a low resolution mesh and doing mesh reconstruction can conserve more details. This step was not used in the typical expression transfer methods, but reconstructing extra details can be useful when wrinkles and similar needs to be transferred to the avatar. Mesh reconstruction is usually done from landmarks but can be improved with other methods. Cao et al. [119] show this by using the low resolution mesh from a global face tracker and enhancing it with extra detail using local regressors.

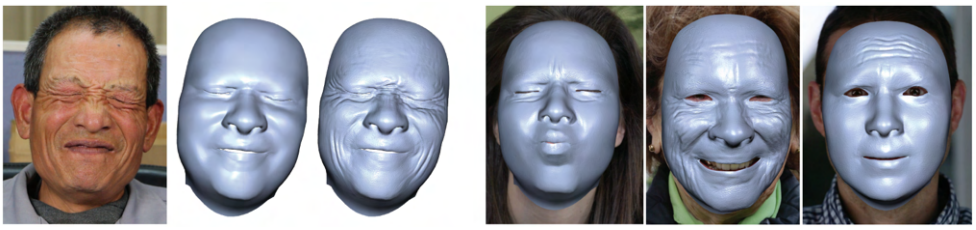

Figure 4.4: Cao et al. [119] show in *Figure 1* from their report how much extra detail can be reconstructed from a low resolution mesh

4.1.4 Expression Coefficients

After aligning the canonical face with the subject, facial expression features can be calculated from it. Features can also be inferred directly from an image or from landmarks. In a sense, a canonical face can be seen as a set of feature points, where each vertex is a point. Grishchenko et al. [102] extracted features from landmarks after predicting the face mesh. Baltrusaitis et al. [120] did Action Unit detection for FACS [4] and highlighted the importance of per individual *calibration* to determine the neutral expression (some individuals have a more smiley or frowney expression in their neutral state). Cao et al. [111] on the other hand demonstrated a way to infer the neutral expression without the need for a calibration step by using a linear regressor based on a large data set.

Expression coefficients do not need to be based on the action units from FACS. They can be arbitrarily chosen based on emotional state (happy, sad, frown, smile), the direction of the gaze (up, down, left, right) or they can define a joint orientation like jaw open. These coefficients are often defined as floating point values in the $[0, 1]$ or $[-1, 1]$ ranges.

Frameworks like ARKit and MediaPipe expose the expression coefficients intended as blendshape values, as mentioned in Section 2.4. See Figure 2.3 for an example. Ozel [121] has a reference for converting between ARKit coefficients and FACS action units.

4.1.5 Transferring the Expressions

Once the shape of the face has been determined and described the expression are transferred to the target avatar. It was most common that the descriptors were expression coefficients corresponding to a predefined blendshape setup, like the ARKit or FACS setups. Saragih et al. [122] and Cao et al. [111] both presented methods for facial puppetry by tracking a person with a single camera and animating an avatar with it. Cao et al. used the common FACS setup, but Saragih et al. built their own database with images of matching expressions for the user and target. Supervised learning was used to match the expressions with each other. Both works emphasize that their solutions are suited for broad expression transfer and are not suited for realistic puppetry. Weise et al. [123] presented a method where a blendshape library was built containing 3D scans of the users different expressions. Deformation transfer, a technique described in Section 4.3, was then used to create a library of the equivalent expressions for the avatar. This method was suited for realistic human models rather than cartoony characters. Newer work such as Epic's MetaHuman animator [124] offers a realistic expression transfer but needs offline post-processing to achieve it. It is likely that realistic real-time facial puppetry is not far off into the future.

4.2 The Camera

Cameras come with many different optical properties such as light sensitivity, lens distortion and focal length. This adds additional complexity to the tracking problem. None of the methods that we looked at had any issues with image quality unless it was severely degraded. Issues did arise however when the subject was partially out of view or being occluded by say hands or a mask. Some methods like the one by Chandran et al. [108] is robust even when the subject is partially occluded.

There were multiple types of spaces and coordinate systems involved in the process of face tracking and expression transfer. Some of the fundamentals are presented in Appendix A.

4.3 Deformation Transfer

There are other ways to do facial puppetry that does not have to involve blendshapes or canonical faces. One such way is deformation transfer. Deformation transfer is the act of transferring the deformation of a source mesh to a target mesh, while maintaining the identity of the target. There has been some work exploring this technique as an alternative to blendshape based deformation. Part of the motivation is that blendshape modeling is time consuming and requires experienced artists. Blendshapes also abstract away details of the face deformations that lie outside what the blendshapes are designed to convey. So far the work we found can not be used with high precision in real-time.

Some early work by Sumner et al. [125] resulted in an algorithm that could transfer deformation between meshes despite the meshes having different number of vertices connectivity. Figure 4.5 shows two meshes from their report where deformation transfer was successful . This method laid the foundation for many upcoming deformation transfer methods.

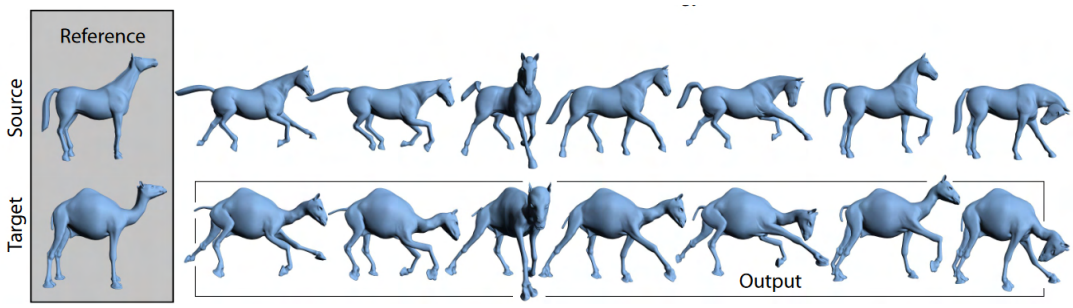

Figure 4.5: Despite different vertex count the transform can be transferred.

4.3.1 Correspondence Map

Many of the techniques created a correspondence map between the source and target mesh. The user has to select a few feature points on the source and corresponding feature points on the target to guide the mapping. The source mesh could be a 3D reconstruction of the tracked person, an avatar or a canonical face. Radial Basis Functions (RBF) were often used to establish this correspondence, using Euclidean distance. This has been shown problematic, since some points that are close to each other in Euclidean space are not supposed to be affected strongly by each other. For example the vertices of the upper lip are not supposed to be displaced downward as the character opens their mouth. To solve this Wan and Jin used Laplacian deformation, which improved this problem but induced other limitations such as being slow when the meshes contain many vertices [126].

Dutreve et al. performed deformation transfer only between feature points of the source and target, instead of deforming the whole mesh. The avatar mesh was rigged with skin weights and then driven by the feature points' movements [127].

By establishing correspondence in 2D and then translating it to 3D Bian et al. avoided the user selection step [128]. Wang et al. proposed a technique that relies on deep learning and does not need a correspondence map at all [129]. For a better overview of current and future work within deformation transfer, see the survey by Roberts et al [130].

5 Inter-Process Communication

As previous chapters have shown, the tracking data for facial puppetry is often done in a process or device separate from the application where the simulated character appears. Sending data across processes, sometimes between devices over the network, is referred to as Inter-Process Communication (IPC). So far the focus has been on the tracking and animation methods themselves, but how to send the data between the processes is also important, since it introduces latency that can affect the user experience.

In order to determine which methods are suitable for sending tracking data over the network we studied common transport and application level protocols both in general and in facial puppetry applications. This chapter establishes requirements for sending tracking data using network packets, what latencies are introduced during IPC and how much latency users tolerate.

5.1 About IPC

IPC can be simplified into three main categories: writing to a file on disk, using shared memory, and sending network packets. Refer to Stevens [131] for a detailed explanation. Accessing storage is a high latency operation. Shared memory, which includes various OS-specific IPC methods, is the fastest path but limits applications to run on the same device. This leaves network packets as the remaining practical option.

5.2 Different Types of Delays

A base requirement for face tracking is that data must be generated and transferred in real-time, which in this case means a non-interrupted continuous stream of data with little perceived delay. In the transmission chain for a real-time game scenario there are multiple steps to consider, for example: voice and image capture, tracking processing time, transfer and processing in the game application main thread, sending over the internet, transit through an authoritative server, then received and interpolated on the receiving game application thread. The chain could be reduced if players are allowed to send data Peer-to-Peer, which is possible in interactions that are similar to a video call, but not reasonable in some multiplayer settings.

5.2.1 Some Delay Comparisons

A ballpark estimate can be made from existing media and devices. Movies use 24 frames per second (41.6 ms), webcams record at 30 (33.3 ms) and games are preferred from 60 and above (16.7 ms). Input devices like a computer mouse can have up to 25 ms latency when used as a wireless Bluetooth device [132]. A ballpark estimate then for latency is around 40-50 ms before the user starts to perceive delay between their own performance and what they see on the screen. This would roughly equate to a 2-3 frames delay in a 60 frames per second game.

5.2.2 Network Delay

Network packets can have as little as microsecond latency between processes on the same device (known as sending over localhost) and single digit millisecond latency over a local area network. A wireless connection can have a degraded signal that can induce occasional latency spikes when congestion is high, but this is of little concern on modern hardware with a strong signal. Many online games operate in the 0-100 ms range and players often use wireless connections while doing so. This suggests that sending a stream of motion capture data using network packets will also be feasible even over long distances.

5.2.3 Delay from High Bandwidth

High bandwidth of data will likely have minimal impact on latency as long as the connection and network buffers are not saturated, since the bandwidth is unlikely to reach problematic levels. An exaggerated example for motion capture data would contain 1000 transforms with nine values each, that would give 9000 floating point values. At 60 Hz and 32 bit floating point precision that would result in a transfer rate of around 17.28 kbps uncompressed, which is negligible compared to a typical video stream between 0.5 Mbps to 20 Mbps. This indicates that bandwidth is of little concern when transferring motion capture data even if the values are uncompressed and string encoded.

5.3 The Experience of Delay in Vocal Communication

Facial puppetry is often combined with voice overs in live streams and voice chats. Humans are sensitive to latency in voice communications. The International Telecommunication Union (ITU) recommends in ITU-T G.114 [133] that one-way latency in "mouth-to-ear" communication should stay below 150 ms for full user satisfaction and states that the majority of users are dissatisfied beyond 400 ms. The ITU does not state a specified number, but it seems an acceptable upper bound is 300ms. This is applicable in all forms of vocal communications, such as video calls, voice chats and telephone calls.

Humans are also sensitive to disconnect between image and audio in video streams, especially lip-sync errors. The ITU claims that the threshold for detectability is -125 ms to +45 ms and the acceptability thresholds are about -185 ms to +90 ms in ITU-R BT.1359 [134]. A positive value indicates that sound is advanced with respect to vision.

We can then assume that for full user satisfaction voice can be delayed by 150 ms, and that the video can be delayed by an extra 45 ms before being noticeable. This means that live stream performances, where there is no two way conversation happening, which is already delayed by several seconds, only needs to consider lip-sync errors. In two way communication, such as when the voice is streamed inside a game, the latency of the voice should not exceed 300 ms, and the lip movement should follow within 90 ms. This allows for a few frames of interpolation given that the tracking solution is fast enough to process each frame.

5.4 Multithreading

Multithreading can be beneficial if the main thread is likely to be maxed out. Sleeping a thread is common to avoid maxing out a CPU core when no work is done. Sleep is a minimum interval command but does not guarantee that interval. Timing resolution, system clock resolution, and OS scheduling priorities can make the sleep wait as much as 15 ms, as described in the Win32 API documentation [135]. A single-threaded approach is valid until its performance degrades beyond the overhead of multi-threaded scheduling and synchronization.

5.5 Transport Layer Protocols

The choice of transport layer protocol dictates connectivity, packet loss, fragmentation of data and packets arriving in order or not. The two main ones are TCP/IP and UDP/IP (simply TCP and UDP from here on) with a few extended variants like Reliable UDP [136]. For a detailed explanation refer to Stevens and Wright [137]. UDP use datagrams and TCP use segments to refer to how the data chunks are being sent and we will refer to both cases as *packets* for simplicity. A comparison between TCP and UDP can be simplified to UDP being a connection-less protocol with likely duplicates, out-of-order messaging and occasional packet loss while TCP establishes a connection with congestion control that ensures packet ordering and delivery. Latency and throughput varies for both protocols depending on load and packet sizes. For TCP to ensure delivery of packets it has to trigger a re-transmission on packet loss which slows down the data stream by up to several hundreds of milliseconds. UDP can be reliable enough if the application has a good tolerance for dropped or duplicate packets and can handle out-of-order delivery. Basic techniques involve timestamps and sliding windows. Improving reliability above the basics will likely cause the developer to reimplement an RUDP variant and using an existing solution like GameNetworkingSockets [138] might be preferred instead. Both TCP and UDP are valid options for sending tracking data if the data streams are handled properly.

5.6 Application Layer Protocols

Sending packets using raw TCP or UDP can be inconvenient for multi-user sessions. These protocols do not include features such as user authentication, encryption, channeling or message parsing. This is what application level protocols handle for us. There is an abundance of application level protocols due to the rapid development of web technologies and many are already being deprecated or replaced. For a deep dive see Roesler et al. [139]. It is therefore important to focus on stable protocols that are easy to implement to ensure low risk of deprecation so that developers adopt the technology.

Web browser applications can be of interest but have a niche set of protocols. Browsers do not allow TCP or UDP directly for security reasons [140, 141]. Web browsers instead rely on HTTP/S, FTP, Web Real-Time Communication (WebRTC) [142], Quick UDP Internet Connection (QUIC), WebTransport or WebSockets. WebSockets is the only viable alternative as the others are either not suited for real-time transfers, need a dedicated server to connect to, or are not ready for use yet. WebSockets act like a duplex connection over TCP with an initial HTTP handshake.

Broker based pub-sub protocols have been popularized with the rise of Internet-of-Things (IoT). MQTT [143] and AMQP [144] are two widely used examples. These protocols allow devices to connect to a server called a broker to publish or subscribe to messages in topics. These topics are defined by URIs, for example *some/path/to/topic*. Several of these protocols support WebSockets for transport to allow web applications to connect to the broker. Using a protocol like MQTT is relatively easy but the downside is that one must host a broker, there is

no peer-to-peer. The upside however is that any number of clients can send and receive data to a topic. This would allow a data source to stream data to multiple recipients, for example a motion capture stream could be consumed by multiple applications.

Developers are unlikely to adopt a broker based protocol when a dedicated server is necessary for IPC. There exist messaging protocols similar to MQTT and AMQP that does not rely on a broker. ZeroMQ [145] is a notable broker-less library. The long term viability of ZeroMQ might be a cause for concern as it has several project forks and rewrites over the years; namely Crossroads.io (now defunct), Nanomsg [146] and Nanomsg Next Generation (NNG) [147].

5.7 Open Sound Control

Open Sound Control (OSC) [92] is a transport independent application layer protocol that was initially proposed back in 1997 [148] as an alternative to MIDI for streaming device data. OSC uses a URI-based scheme similar to the broker protocols and supports multiple data types [149] (integers, floats, timestamps, strings, data blobs). OSC can send time tags for receivers to eliminate jitter in transport (the developer has to implement a time synchronization routine separately from OSC). OSC has a large set of open source libraries and plugins and is commonly used in creative coding circles [150, 151, 152]. Unity3D and Unreal Engine support it and games like VR Chat [153] use OSC as a way to control avatars within the game using custom devices.

Virtual Motion Capture (VMC) [98] is a protocol built on top of OSC for sending motion capture data for avatars defined using the VRM file format [154]. VMC is used to send data between many applications that are listed on the reference page [155]. OSC together with VRM and VMC are commonly used alternatives in the VTuber scene as mentioned in the survey in Section 3.4.

6 Application Prototypes

In order to evaluate the complexity of implementing real-time facial puppetry we implemented a few prototypes for collecting, sending and receiving facial expression data. We wanted to determine how we could set up a workflow for facial puppetry that was both practical and performant for developers. The goal was also to determine a suiting architecture for a possible application that could act as a bridge between tracking solutions and applications that want to consume the tracking data.

The choices of tools and configurations were based on our findings in the survey and what tools were available for free or at low cost. We chose to primarily work with the 52 ARKit coefficients, accessed through MediaPipe.

The source code for the prototypes is available in the public GitHub repository[1].

6.1 FacePipe

We created a basic one-way UDP protocol that we named FacePipe, with similarities to OSC and iFacialMocap, to determine the viability of using a connection-less stream of data. The layout is listed in Listing 6.1. The protocol includes scene, camera, subject, timestamps and supports landmarks, blendshapes and matrices. Datagrams were sent in human readable ASCII encoding but could have been serialized more efficiently using a different encoding.

6.2 FacePipe Python

A Python application, that we refer to as FacePipe Python, was implemented to simulate an arbitrary tracking source in a separate process or device. It uses MediaPipe with a webcam stream to generate landmarks, expression coefficients and a face transformation matrix. It sends data over UDP using the FacePipe protocol. The full code is listed in Appendix B.

6.2.1 Latency Test

A test in FacePype Python was made to determine the latency from requesting a webcam capture to receiving tracking data as an output for a single frame. This was done using the

[1] `https://github.com/DennyLindberg/FacePipe`

difference between two timestamps. Both steps combined varied between 40-60 ms. The webcam recorded at 30 frames per second, roughly 33 ms per capture, which meant the remaining duration was spent in MediaPipe to extract the landmarks and facial coefficients.

6.3 FacePipe C++

We implemented a C++ OpenGL application that we refer to as FacePipe C++. It was both used as a 3D preview for tracking data but also for testing the viability of being an intermediate application for processing and forwarding the data. It was also developed as a UI prototype for how a potential application could look like. A screenshot of the prototype can be seen in Figure 6.1 where it was in the middle of receiving data from FacePipe Python.

FacePipe C++ uses UDP sockets on a separate thread with a basic receive, sleep, repeat pattern. The data is shared with the main thread through a *std::queue*, using *std::lock_guard* and *std::mutex* to ensure thread safety.

6.3.1 Sending Data to Third-Party Applications

We tested receiving FacePipe data in both Blender and Unreal Engine, both directly from FacePipe Python and FacePipe Python via FacePipe C++. This was done to evaluate the final step of applying tracking data to achieve live facial puppetry. Figure 6.2 shows landmarks received in Blender. Figure 6.3 shows the result in Unreal Engine when applying blendshape coefficients to one of the characters in Mortal Online 2. The code in Blender used Python with a UDP socket and a timer to update the scene. For Unreal Engine we created a plugin to receive and unpack the data using the same FacePipe protocol code from FacePipe C++. The socket was implemented using the engine's own interface. Delegates were exposed so that the blendshape coefficients from the FacePipe data could be applied on the character in Blueprint as animation curves. Blueprint is a visual scripting system in Unreal Engine. The Blueprint nodes are shown in Figure 6.4 and 6.5.

Similar tests were made by using the OSC protocol between FacePipe Python and Unreal Engine's OSC plugin. There was no notable difference when changing protocol.

```
# FacePipe protocol packet layout
type|protocol|source|scene,camera,subject|time|contenttype|contentdata

# Example
a|facepipe|mediapipe|0,0,0|42.3312|bs|...

[type] # first byte declares how to read the data
    a=ascii, b=bytes, s=string, w=wstring, e=encoded

[protocol] # is a name. Allows the same socket to potentially be shared by
    other sources than FacePipe that has their own data structure. This
    would likely change the remaining packet structure.

[source] # is the source generating the data, in FacePipe it signifies the
    tracking data source (e.g. mediapipe, ios_arkit, nvidia_broadcast).
    Landmarks, blendshapes, and transforms can differ based on the source.

[scene,camera,subject] # are indices. These exist to potentially allow
    multicamera setups to track the same subject. The scene would allow
    different recording setups or locations.

[time] # is a 64 bit floating point value in seconds determined by the
    source - it could for example be from epoch or application start.

[contenttype|contentdata] # is where the specific data begins. Landmark
    coordinates are x,y,z serialized in sequence. Landmarks also need the
    image dimensions so that normalized coordinates can be converted to
    pixel space. The names for matrices are hypotheticals and only the face
     is used so far.
 Landmarks2D: l2d|imagewidth,imageheight|0.1,0.2,0.3,0.4,...
 Landmarks3D: l3d|imagewidth,imageheight|0.1,0.2,0.3,0.4,0.5,0.6,...
 Blendshapes: bs|mouthShrugUpper=0.5|eyeSquint_R=0.2|...
 Matrices:  mat44|face=0.1,0.2,0.3,...|eyeL=...|eyeR=...|jaw=...

# Below are examples of packets sent from the Python application. Content
    is truncated for readability.
Landmarks3D:
a|facepipe|mediapipe|0,0,0|14.469|l3d|640,480|0.4427582621574402,...|...

Blendshapes:
a|facepipe|mediapipe|0,0,0|10.133|bs|browDownLeft=0.02666594833135605|...

Matrices:
a|facepipe|mediapipe|0,0,0|2.31|mat44|face=0.9259510636329651,...|...
```

Listing 6.1: FacePipe protocol data layout for UDP packets

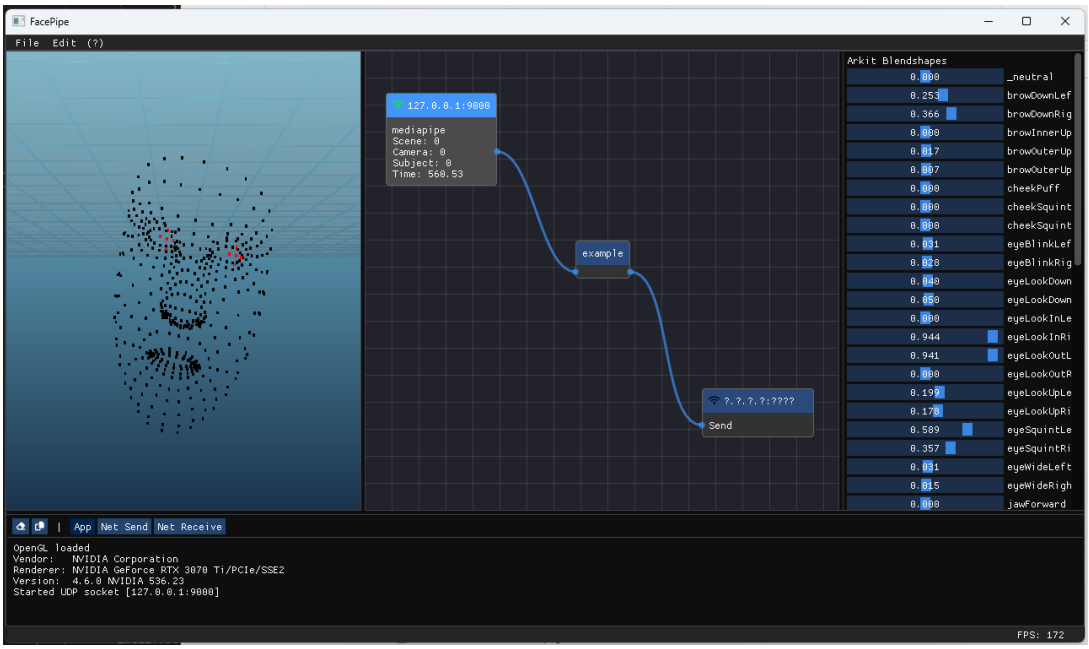

Figure 6.1: The prototype FacePipe C++ application acting as a preview and mixer of facial tracking data.

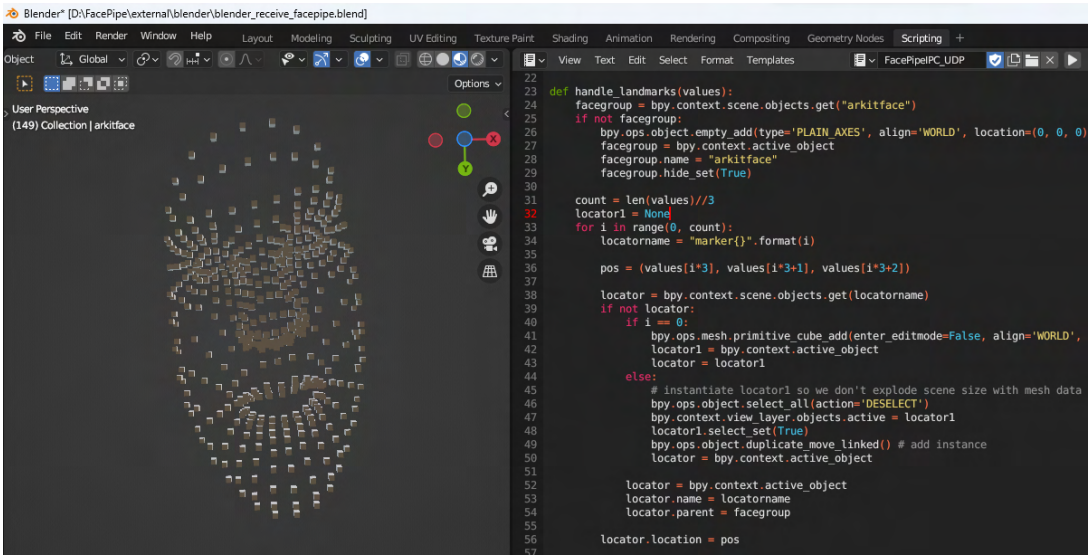

Figure 6.2: Receiving landmark data in Blender using the FacePipe protocol.

Figure 6.3: ARKit blendshape coefficients applied on a Huergar character from Mortal Online 2 in Unreal Engine using the FacePipe protocol.

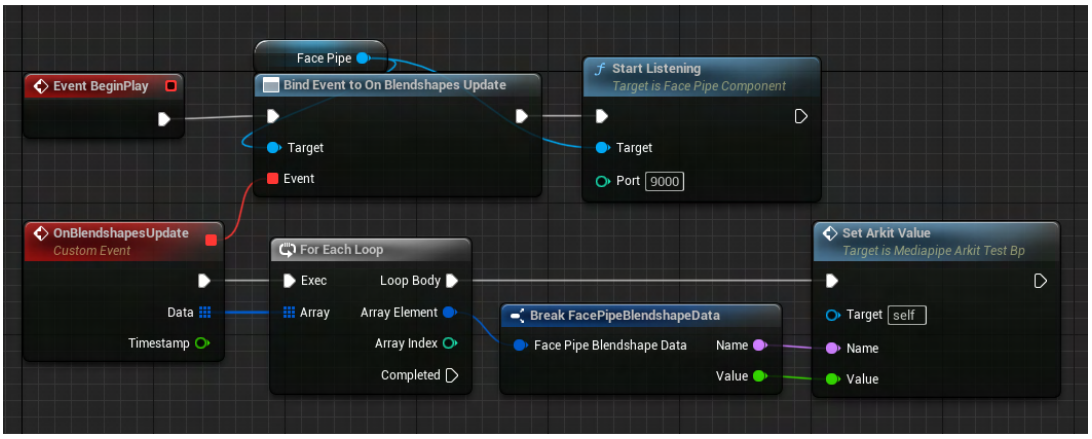

Figure 6.4: The event structure in Unreal Engine for getting the FacePipe data over a UDP socket. SetArkitValue stores the value internally for Figure 6.5.

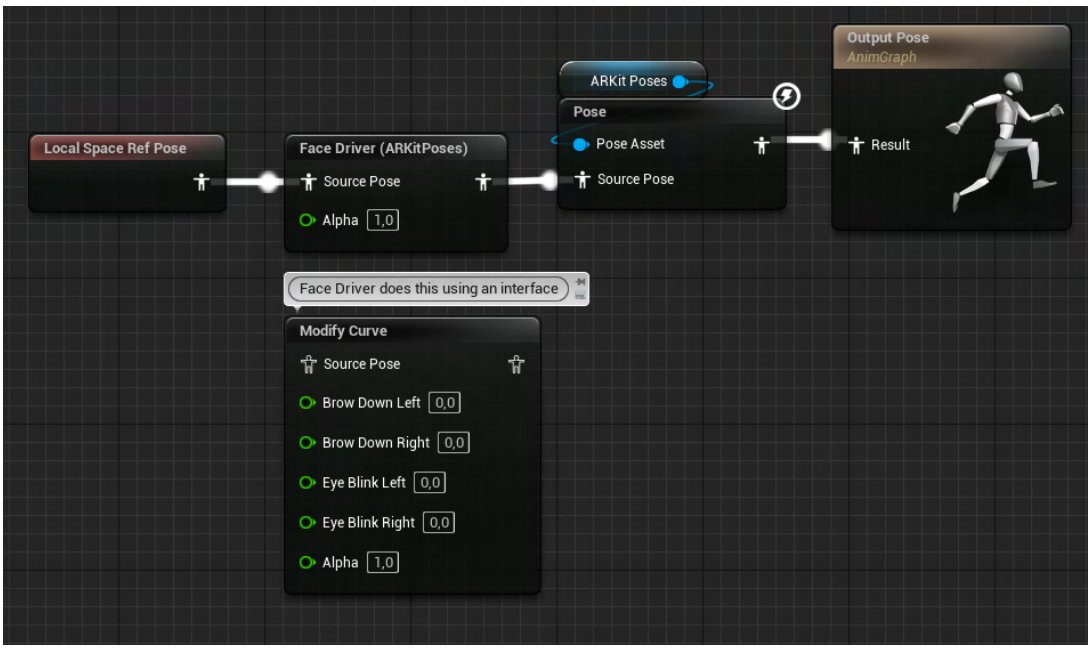

Figure 6.5: Applying the ARKit coefficients from FacePipe in Unreal Engine's Animation Blueprint. The face is driven using both a pose asset and blendshapes. We used an existing C++ node from Mortal Online 2 to avoid applying all the ARKit curves by hand.

6.4 Results from Prototypes

This section contains an integrated analysis of the results, discussion about the results and some conclusions drawn from the work on the prototypes. In the end you will find a summary of the key points. The discussions and results here have a MediaPipe perspective, since that is what was used in the prototypes.

6.4.1 Using MediaPipe

It turned out that working with a framework like MediaPipe in C++ was troublesome to such an extent that it was better to rely on the Python or JavaScript implementations. Using OpenCV to access the web camera was only a few lines of code, but seemed to interfere with the audio devices of the PC as a side-effect. This makes us believe that the likely best approach for MediaPipe tracking is to make use of the web applications mentioned in Section 3.3.2. The only issue here is receiving the WebSocket data on a desktop application. From this we can conclude that the MediaPipe framework has a lot left to offer in terms of accessibility, but it is still very relevant for developers, since it is in constant development and since it outputs the common ARKit coefficients.

We did a few tests where we sent ARKit data instead of MediaPipe data, using NVIDIA Broadcaster via the iFacialMocap implementation and a few iOS apps. We see little need to make a custom input solution here.

6.4.2 Transmitting Tracking Data

Once the MediaPipe Python application was created and the FacePipe protocol was defined, the serializing and transmitting of face tracking data proved trivial. The data itself needed little processing. Most of the work was spent in generating the data in Python, setting up the UDP sockets and processing the data in the target application.

Using UDP directly proved to be a plug-n-play solution. Little work was necessary for bi-directional communication. The OSC protocol however turned out to have a lot of layers that made the process more complicated than we deem necessary. The fact that other solutions use WebSockets, which is similar to TCP, suggests that it is safe to use either UDP or TCP to send data. If implementing a standalone tracking solution that broadcasts tracking data, OSC is a relevant option to consider since this protocol is already employed by other applications. It is also a good choice if the target application already has support for OSC.

Handling Tracking Data

We expected to do more post processing work on the face tracking data once the datagrams arrived in the target application, but that was not necessary. Apart from the limited frame rate of the webcam the network stream of landmarks and face coefficients were steady with no perceived jitter, loss, or duplication of frames. The low framerate (30 fps) somewhat affected the user experience. We believe applying interpolation could smooth out the low frame rate, but we did not test that due to time constraints.

This lead us to the conclusion that UDP, TCP and OSC are all relevant protocols for streaming tracking data, and that handling the network data stream should be of little concern to the developer. This is based on the observations that implementation was very straight forward and minimal post processing was needed.

6.4.3 Processing and Forwarding Data via an Intermediate Application

Sending data from FacePipe Python over UDP to FacePipe C++ before forwarding it to a third application showed no perceivable latency increase or quality loss despite each step having its own application loop. This means that having an external application for receiving, mixing

and processing data is a viable solution, given that the latency from the application processing is so small that the overall latency of data transfer is less than the 300 ms limit mentioned in the IPC chapter.

6.4.4 Applying the Data to Characters

The expression coefficients for blendshapes needed no processing. Once the values were applied to the character in Unreal Engine it just worked. This of course depended completely on that the character was properly set up for the ARKit coefficients. The subjective results, according to us, were good even though no smoothing or other post processing was applied.

Transferring and displaying landmarks and head transforms were tested in three applications: Blender, FacePipe C++ and Unreal Engine. No issues appeared, apart from the need to transform the landmark data to another coordinate frame matching the application. The landmarks were only previewed and not used to drive the avatar, since no solution for this was available.

6.4.5 Prototype Conclusions

Transmission of the tracking data was easy. Applying the data to an avatar was non problematic as long as the avatar was designed to be driven by that type of data. Acquiring the data was more complicated because of the problems we ran into when implementing support for the tracking.

UDP, TCP and OSC all showed to be suitable protocol for transmitting face tracking data. Because of the low latency influences when sending data between processes, it is possible to create a software that receives tracking data from another application, process this data, and then forward it to a target application.

6.5 Proposed Workflow

Based on the findings in our surveys and the experiments with our prototypes we propose a workflow for the developer who wants to implement facial puppetry in their game today, using the available tools, rather than developing a solution like an API or similar intended to make implementation easier for others in the future.

6.5.1 Pick an Expression Standard

We recommend starting with the ARKit standard of 52 expression coefficients (Perfect Sync) as that is supported by ARKit, MediaPipe and NVIDIA Broadcaster after a few value conversions. The results of the survey suggest that this will be a stable medium quality foundation for the near future.

6.5.2 Creating the Character

Create the character using a skeleton of bones but pose the face primarily using blendshapes. The neck, head, jaw and eyeballs are good candidates for bones. So are peripheral details like hair and ornaments. The jaw and eyeballs can be posed using blendshapes but consider the implications if there is need to procedurally adjust those parts using look-at constraints for the eyes or let the jaw hang during ragdoll physics. The head and face might not always be driven by facial puppetry in a game. Details like teeth, beards, piercings, and glasses, have different challenges when posed with blendshapes or bones so plan the work accordingly.

There are three choices when creating the blendshapes for a character: A) do it by hand while following one of the sites that show a reference for each pose, B) use a software like Faceit for Blender to automate it, or C) use a service like Polywink.

6.5.3 Determine How Facial Puppetry Should Apply

Consider the body, head and face separately. For a VTuber application it might make sense to have the whole torso follow along with head tracking movement across the camera frame to have a 1-1 puppetry experience. In a game however one might only need facial expressions without any head movements at all, or rotation at most, to not disturb other gameplay aspects.

6.5.4 Determine Platform Requirements

VR devices should rely on existing built-in hardware support, like in the Meta Quest Pro, or use a hardware add-on like the VIVE Face Tracker. Refer to the OpenXR specification for how to access the extensions. For the rest of the proposed workflow we focus on a PC setup with either a webcam or a remote Apple device that supports ARKit expression coefficients.

6.5.5 Implementation Aspects

NVIDIA Broadcaster, MediaPipe and ARKit are the three foundational tracking solutions. You need to decide if you implement the tracking solution directly into your game or run it as a separate process. We recommend running it separately as it is easier to maintain and standardize. Sending network data to your application on the same machine is practically instant and running a separate process makes it easier to decouple tracking from the main game thread. The only challenge here is the end-user aspect of connecting the applications.

You then decide if you should implement a tracking application yourself or rely on an existing derivative application supporting the ARKit curves. The challenge then is to send and receive the tracking data to your application. Either create a raw TCP or UDP socket in your application on a separate thread or use an existing protocol like VMC or OSC. Refer to

our FacePipe prototype for a raw UDP example. Refer to the survey for existing solutions like iFacialMocap or OpenSeeFace.

Once you receive the data in your application you need to consider basic issues like packet loss, duplication and jitter. A raw stream of UDP data on the same machine is pretty much flawless in practice and you might only need to consider smoothing the movements using interpolation. Some facial movements are snappy in reality, especially the stochastic movements of the eyes, which means interpolation might only be necessary for head poses but not the facial expressions.

With the ARKit curves, and potentially some transforms from MediaPipe, your data is almost exclusively face expression coefficients. Driving the blendshapes should be trivial. Be wary of coordinate system changes if you try to map a head transform to your character.

6.5.6 Additional Considerations

Consider supporting multiple trackers at once to improve the user experience. Dedicated eye trackers might be necessary for proper eye contact in a virtual environment.

Have a fallback pose when tracking is lost, for example when the subject leaves the camera view. A blend back and forth from a default idle animation state can be useful. Depending on the tracking source you might need to remap or clamp the expression coefficient values to stabilize poses. For example the eyelids might not close completely for users when they blink. Be careful though as some expression coefficients work together and modifying some values might break some poses. Be wary of the uncanny valley. Smoothing, reduction of detail and exaggerated poses might be necessary.

Ensure that live facial puppetry with voice-over stay within the latency thresholds as mentioned in Section 5.3.

7 Discussion

In this chapter we discuss our results, methodology and the work in a wider context. With the results from both our survey and our practical tests, we can now discuss and interpret our most relevant results in relation to our research questions. The first research question concerned what challenges developers face when implementing facial expression transfer. The second and third question regarded expression formats and if there is any homogeneity among these formats. The last questions regarded architecture that enables real-time capture and transfer of the data.

7.1 Developer Challenges

We found that developers are faced with the challenge of implementing support for a wide range of hardware devices in order to not exclude some users. Doing this was found very impractical, which left the developers with two choices. The first choice was to target their product with a specific device in mind, for example an iOS device, and integrate the library or API directly into their source code as seen in Section 3.3.1. Accessing the face tracking data this way would be straight forward but lock the developer to a particular platform, see Section 3.2. The second choice would be to use the platform as capture source and transfer data to their application using methods described in Chapter 5 on Inter-Process Communication. There could have been a third choice, using an API like OpenXR in Section 3.4, but there is no standardized extension yet for face tracking and expressions apart from vendor-specific extensions for HTC and Meta devices.

7.1.1 Integration into Main Application

Even if opting for the first choice and picking one of the most common frameworks, MediaPipe, the developer is likely to encounter a multitude of challenges. For starters, the majority of the solutions in MediaPipe are only available in C++ [33]. Running the C++ version of MediaPipe on Windows is experimental and requires prerequisites like MSYS2 (a collection of tools and libraries), some necessary packages, Python, Visual C++ Build Tools 2019 and WinSDK, Bazel and OpenCV. For Linux Debian and Ubuntu it is enough with Bazel, OpenCV and FFMPEG. Bazel is an open source port of Blaze, which is the build tool Google uses internally. Developers who use C++ are forced to build their project using Bazel if they want

to access the MediaPipe solutions. This is not possible in all projects, for example projects in Unreal Engine. Accessing MediaPipe using Python is easier, but offers less functionality.

This type of problems were not exclusive to MediaPipe. In general, API's and libraries do not exist for all programming languages and the information and functionality that can be accessed varies depending on the chosen language.

7.1.2 Standalone Tracking over IPC

The second choice of using an external application for tracking is the easier alternative. The developer can either develop a standalone application that performs the tracking and streaming of data, like we did in Chapter 6, or use existing apps mentioned in Section 3.3. Similar challenges to integrating tracking directly in the application, as in the first choice, will appear when implementing a standalone app. There are not many existing applications that stream data in a general sense. Usually they stream specifically to another application, like Unreal Engine or Unity, using a protocol like OSC as seen in Section 3.4.1. Apart from challenges in acquiring the tracking data, the main challenges are around sending, synchronizing and interpolating it.

7.1.3 Conclusions on Developer Challenges

Developers do not need specific knowledge in the computer vision field in order to use the expression transfer solutions that are available on the market today, as listed in Section 3.3. They do however need knowledge in certain programming languages. It can be time consuming and bloating to implement support even for only a few specific solutions. Available solutions are not very versatile and won't work for all projects. A developer with infinite time resources could implement support for many tracking solutions as long as the solutions are integrable in the project, or develop their own solution based on existing research in Chapter 4. This points to that it is rather a lack of time than expertise that prevents developers from using facial expression transfer. Though the need for specific programming languages can be seen as a lack of expertise in one way. However, the poor integrability seems to be the bigger reason expression transfer is not used more often.

7.2 The Problem of Homogenization

Even if the problem with accessing the tracking data would be solved, there would still be problems. The tracking data from different devices have different formats. The two most common solutions presented in Section 3.3, ARKit and MediaPipe, would deliver the 52 ARKit coefficients, while the other solutions deliver different data.

7.2.1 Incompatible Formats

The lack of homogenization means in practice that even if support for receiving tracking data from all available solutions is implemented, only tracking data from some of them will be able to drive the avatar. The avatar has to be rigged in accordance with one of the tracking formats. From a practical standpoint this means picking a good standard early on. For applications in desktop mode it can be assumed that the user will have access to one of the popular devices in Section 3.2 on Hardware: either a webcam, Android or iPhone device. Choosing the ARKit coefficients would probably be the best choice as there is a way to generate them for each device, using an existing solution from Section 3.3 on Software, or implementing a custom solution with MediaPipe or the ARKit SDK. We can see in the same sections on hardware and software that applications that can be played in both VR and desktop mode will be problematic since there is no hardware that generates the 52 ARKit values for VR headsets. For example the Meta Quest Pro [29, 90] generates 63 values that likely can be mapped to

the ARKit setup, but it doesn't have tongue tracking, which ARKit has [40]. Oculus Lipsync uses 15 viseme blendshapes [43], this is however generated based on voice and not a visual recording. The VIVE Facial Tracker has 38 blendshapes that covers the mouth and jaw region, including tongue [38].

Homogenization in VR Body Tracking

When it comes to general VR hardware, like body trackers, controllers and headsets, OpenXR has already started to solve the problem of accessing tracking data from different devices [90]. It has also started developing a unified framework, so controller output from different manufacturers can be mapped to the same keywords. This unified framework does not yet exist for blendshapes. As mentioned before in Section 3.4, OpenXR has optional extensions for the 63 Meta Quest Pro curves and the 38 VIVE Facial Tracker curves, but there is no way to directly convert between the different blendshape setups without some form of custom mapping.

Blendshape Superset

It should be possible to convert between all blendshape setups, as long as it is possible to find a superset of blendshapes that is capable of describing all blendshapes within all methods. The superset would act as an intermediate step for the translations. One such superset has been created in the previously mentioned VRCFT [95]. It is possible that similar features will appear in commercial solutions in the future.

A blendshape converter does not solve the cases where the facial expression transfer does not include blendshapes. Blendshapes are by design problematic and can cause physically implausible distortions, has problems with preserving volume, causes anatomically impossible face expressions and self-intersections of surfaces [156].

7.2.2 Blendshape Free Formats

Other tracking methods that involve more direct transfer, like deformation transfer presented in Chapter 4, might be more beneficial for realistic characters. In applications where the user is driving a realistic rig or maybe even a close approximation of themselves as a 3D model, blendshapes could remove some of the realism. Especially if using the 52 ARKit blendshapes setup, which does not have enough fidelity to represent all facial expressions.

Deformation Transfer as a Potential Future Alternative

We believe that deformation transfer from Chapter 4 can gain popularity in the future as an alternative to blendshapes. For now there needs to be more research on deformation transfer techniques that can handle the holes for eyes and mouth in the mesh. Calculating the correspondence map is so far expensive and has to be done offline. It also requires the user to manually and accurately select corresponding feature points.

7.2.3 ARKit's Impact on the Market

We saw that the 52 ARKit blendshape coefficients are converging into an unofficial standard in Section 3.4.2 and that is having an impact on the market in several ways. Other solutions that output their own tracking data and have their own method for expression transfer have started to support the ARKit coefficients simultaneously, since this is the most likely setup that already existing characters will use. Nvidias protocol is almost identical to the ARKit blendshapes, indicating it was probably heavily influenced by that. The only difference is that ARKit's "cheekPuff" and "browInnerUp" has been divided into left and right, and there is no support for tongue movement [40, 41, 42]. In general, the convergence towards the 52

blendshapes makes it easier for developers to find a common standard and make their products compatible with each other, at the same time as it slows down the development of better and higher detail solutions. On a developer and user level, this means character creators are being forced into the limits of this setup when rigging and animating their characters. In the scientific field it has motivated research on automatic blendshape generation, with the 52 blendshapes as target [157]. On the commercial market it has given rise to services for automatic blendshape generation [78] and software that helps as reference when rigging by hand [77].

MetaHuman [74] has also started to make its way into the market with its more detailed tracking and rigging, and solutions like Avatary [48] offers tracking output profiles for both ARKit and MetaHuman.

7.2.4 New Methods

Since facial expression transfer is a field on the rise it is likely that new methods will spring up with more delicate tracking and less rigidity and errors. Probably methods with various blendshape setups and methods with other types of expression transfer. It is likely we will see different standards in the future. For example a blendshape standard for simpler cartoony characters and a more sophisticated technique for more realistic characters. In the near future we expect the simple setup to be the 52 blendshapes, but further on we expect a blendshape setup with higher detail. Especially tracking of the tongue can be expected to improve, which might require special depth-sensing hardware at first, but later will be solved with machine learning techniques.

7.2.5 Homogenization of Feature Points

As feature points, or landmarks, are not used to drive rigs yet outside of the professional movie industry, there is little motivation to find a homogenized landmark setup at this moment (see Section 4.1.2). The landmarks are mainly used internally within the tracking solutions for face tracking, to solve for blendshapes and for object alignments. There could be a reason to find the most efficient landmark setup in the context of face tracking and expression classification, but it goes beyond the scope of this paper. Within the scope of this paper the interest lies in finding the optimal setup for run-time expression transfer, meaning driving a rig in real-time. Since that is not done in practice yet with landmarks, there are no landmark setups to analyze in this context.

7.3 Data Transfer and Architecture

When we looked at inter-process communication in Chapter 5 we studied both methods for the actual data transmission, and what values are acceptable in terms of delay. We found that sound has to reach the receiving user within 300 ms, and that accompanying animations need to follow within 90 ms of the sound. In a general pipeline there will be delay from the camera and mic, delay from the tracking algorithm, delay from the transmission and so on. Network delay and delay from high bandwidth were found to be low or negligible in this pipeline. Our surface level tests in our prototypes in Chapter 6 seem to confirm this. What seems more important is the efficiency and speed of the tracking algorithm itself from input to output.

We state in Chapter 5 that TCP and UDP are both valid options for transmitting facial expression data. However UDP seems to be favored in real-time interactions. VR Chat [18] for example relies on UDP because fast is key when interacting with others. The web-based solutions in Chapter 3 use TCP since the only viable option of using websockets are based on TCP. Both UDP and TCP are therefore viable for real-time facial puppetry.

7.3.1 Architecture

One of our research questions concerned how the architecture of a software could be designed to make face tracking more plug 'n' play in PC games. We found in Chapter 3 that only few PC games had support for face tracking without running an extra program on the side that streams the tracking data. In mobile platforms there is a wide variety of apps where tracking is supported, running directly inside the app.

PC vs Phone Environment

This made us wonder why it seems easier to build a face tracking app in a phone environment than a PC environment. One reason we found was that the only input option will be a camera, either a regular one or one with a depth sensor. Another reason is that the app is usually of the nature that it displays the users face with a filter applied, or overlays an avatar that is controlled by the users gestures. There is no gameplay involved, or at least, the main purpose of the app is not gameplay, which makes the architecture design simpler.

A Facial Expression Converter

After finding such a variety in the tracking devices and format of the output we thought it would be good to have a program that could take in any tracking format and then convert it to any other tracking format. That way the game developer could decide to implement tracking according to any of the formats, and not be concerned about what tracking output the users tracking solutions would generate. As seen in Chapter 7.2 about homogenization, this is not a simple task. Any case involving translation between tracking formats requires a mapping between the formats.

Our findings in Chapter 4 showed that it is possible to create a mapping if both tracking formats are blendshape based and similar enough, as is the case with NVIDIA and ARKit [41, 40], although the mapping would be approximate and not an exact translation. In the cases where the tracking is not blendshape based it could be possible to have an intermediate canonical face that gets controlled by the user, and then the target tracking format can be derived from the deformed intermediate face. But this is merely a hypothetical based on the findings in Chapter 4. It is a unlikely that this scenario will be available in the near future without further research. Instead a program similar to VRCFT [95], that has support for many different tracking methods and can translate between blendshape setups, is more likely to be helpful.

Vendor Locks and Profits

One thing hindering the development of easy to use face tracking is that the development is in an early stage and companies benefit more from making their solutions exclusive and forcing games into being vendor locked. We saw for example in Section 3.4 that Meta and HTC developed their own expression sets in the extensions for OpenXR with no commonalities. Once the market progresses it is likely more value will come from having a solution that supports other tracking formats than the own specific format, and this might drive the vendors to develop solutions that deliver a wider variety of outputs.

7.4 Method Choices

Without the survey in Chapter 3 we would not have known what the market looks like or where it is heading so we think that was a good choice to do a survey. We could have made the results from the survey more interpretable for the reader if we included statistics over what solutions or research papers focused on a specific facial expression output. This would also have supported our claims and conclusions, such as that the ARKit blendshape coefficients is

the most common output. During our research we came across more solutions and research papers that are not mentioned in our report, but they have given us a more nuanced view which aligned with the results of the survey.

Doing a survey should be a reliable method for gaining an overview of the available solutions and facial expression formats. What could be problematic is if we searched for similar keywords because they were common in our results, and this made us miss other keywords that would have led us to other types of answers. This can be seen as a low risk given the variety of solutions we found.

In the work on the prototypes in Chapter 6 we draw conclusions on problems related to implementation. This despite that we only used MediaPipe and did not implement other solutions. The problems concerned incompatibility with existing project environments, resulting in us deciding to implement the tracking in a standalone program designed to communicate with the target program. We understand that our results are influenced by the fact that we only have tested implementing MediaPipe solutions in practice, but have reason to believe that similar problems with implementation and integration would occur using other framworks, after studying their SDK's and API's, and after studying the market as a whole. Working with the prototypes and accessing raw data from a tracking method was a very good way to get an understanding on how data can and can not be converted between formats. The prototypes did not give an answer to how we could translate from a blendshape based expression format to any other arbitrary format, which we had hoped for.

7.5 Chosen Delimitations

No quality comparison or evaluation was made between motion capture devices, methods or data formats unless it was relevant from a real-time application developer perspective (i.e. game development). The work was primarily done in a PC environment but branched out to mobile devices when necessary, since the problem arouse in a PC development environment. Methods such as professional grade motion capture studio setups were excluded because very few consumers will have access to these tools. We speculated that it is more likely that affordable camera based AI solutions are going to reach customers at large before an equivalent to professional grade hardware ever will. Methods that could not run in real-time were also excluded with exception for methods under development that might run in real-time in soon in the future, since we wanted to investigate real-time facial puppetry. We were only concerned about run-time performance if a method ran in real-time within a reasonable performance range for consumer devices. By real-time we mean fast enough for the user to perceive it as such. With reasonable performance we mean a negligible impact on the target application. We did not attempt to search for every single device and method available, since evaluating a large enough subset of capture devices and methods were assumed to be enough to reach the goals of the thesis. We did not cover privacy, data ownership and security as that is a layer above the topics in Chapter 5 on IPC. It is a valid concern but is not of great importance for the aim of the thesis.

7.6 The Work From an Ethical Perspective

Working towards accessibility for real-time facial puppetry means making the technology available for all types of people, not only game developers and gamers. On social platforms it will be easier to build a following based on an imaginary character, while staying completely anonymous. Anonymity can increase the safety for the person running the social media, but it can also be associated with risk for impersonation etc. It can also be a risk for the followers, who do not know who they are actually talking to. If facial puppetry get so realistic that it is difficult to distinguish from reality it will give rise to more fake news of celebrities or leaders,

which is already happening with AI pre-rendered video footage or live face swap technology like DeepFaceLive [158].

When working with models that have been trained on large data sets of images there is a risk of using a biased model that might not handle situations where the user is wearing glasses, has a beard, has a specific skin color or has some kind of deformation in their face. We did not take any measures to address this issue in our work since we were interested in analysing the data after it had already been successfully acquired.

7.7 Future Work

As we saw in Chapter 4 on commonalities it appears that face tracking output is based on landmarks, expression coefficients and canonical meshes but each implementation seem to invent its own scheme. Instead of us declaring a need for a shared standard, a framework for handling different configurations would be desirable. We have seen examples like OpenXR [90] acting like a bridge for various peripherals and extensions. This is however in isolation from some platforms like non-XR PC gaming. Face tracking is slowly becoming an expected feature but how to implement it is far from a standardized process.

7.7.1 Standardization of Facial Puppetry

We suggest looking into the possibility of a framework or system defined based on existing schemes in Chapter 4 that not only enables old and new methods to coexist, without the risk of deprecation, but also for finding methods for translating between schemes for long term support. Such a framework would not only deal with raw data like landmarks and expression coefficients, but also offer guidelines for achieving facial puppetry. Such as level of detail, minimum requirements and achieving universal emotions.

Some early results can be seen with the standardization of the 52 ARKit expression coefficients, this can potentially offload developers to focus on their creative efforts (see see Section 3.4.2). The creative value in achieving a shared facial puppetry framework for both the video game and online streaming industries should not be underestimated.

7.7.2 Calibration and Normalization

Throughout the report we saw a few examples of user face calibration, with a few specific methods mentioned in Chapter 4. The neutral pose was resolved to be able to estimate expressions. Sometimes a canonical face was used as an intermediate for other actions like mapping landmarks.

We believe that resolving the neutral bind pose of a user's head and exposing that as data to the user would be valuable, similar to a canonical head. Landmark configurations are most often resolved and returned in relation to the camera view, which can be a bit troublesome to work with. It is often easier to work with such data when it occurs in relation to a bind pose. Getting landmark poses in relation to the bind pose of the head instead of the camera view would be valuable.

Several existing methods presented in this report includes such an intermediate stage but rarely seem to expose it as useful data to the end user. We believe that more focus on the canonical head and data in such a bind pose space need further exploration. Any data that ends up in a canonical space, be it in 3D volumetric space or 2D UV space, significantly eases the ability to introduce new algorithms. The combination of a user calibrated head, the canonical head and landmark poses in the bind pose, is likely a useful normalized space for future work.

8 Conclusion

This thesis aimed to find if there exists a framework that can take in and standardize face tracking data from different devices or if such a solution can be created. Regardless of the users tracking device, the developer should be able to access the tracking data in some common format. The aim was also to summarize available hardware and software and compare the data formats that these solutions output, to find out if there is any homogeneity. We wanted to determine if and how it would be possible to simplify the implementation process for those that want to use facial expression transfer in their applications.

8.1 Research Question 1

The first question that appeared was why facial puppetry is not used more often in PC games, seeing that it is already pretty well developed on mobile devices. We found that one reason is because it is easier to develop it in the phone app directly. In the phone there is usually no gameplay involved, so the architecture design of the app can revolve around the tracking and applying the data directly in the app. The available tools for streaming facial mocap data from these simpler phone apps to a computer usually stream exclusively to another software like Unity or Unreal Engine. Integrating available PC-based solutions into projects turned out to be problematic because of compatibility issues. This leads to the need for the tracking to be done in a separate standalone process that can stream the tracking data to the game, which seems to be the best solution at the moment. VR hardware all differed in their output formats. This means that an avatar that works with many of the camera based tracking methods will not work with the VR tracking. An avatar that works with one VR solution will also not work with another VR solution. This makes it extra problematic when implementing games that are VR based or can be played in both desktop and VR mode.

8.2 Research Question 2 & 3

We also wanted to answer the question of which methods are currently relevant for developers. Is there a standard for facial expression data, or can there be a standard format that all tracking methods can output?

We did not find a common standard for representing facial expression data, but found that the most common format found was the ARKit blendshapes. It is likely that blendshapes

became popular because of the simplicity to use them compared to other techniques. The fact that ARKit had such an early start has shaped the market, with many solutions offering tracking in the ARKit format. MetaHuman has recently entered the scene and is already gaining traction, with some commercial solutions now offering both ARKit and MetaHuman facial expression format as output.

One of the reasons there is no standard format is because the ways to transfer expressions differ a lot, to the point where they are barely comparable. Some formats use direct transfer of the mesh deformation according to a correspondence map, some use a combination of blendshapes coefficients and bone angles and some use only blendshapes. It should be possible to create a mapping between the different formats with for example machine learning, but the motivation for this is low. This is because most of the methods use regular cameras as input, and if a different format is desired as output, it is possible to just use the original camera and pick another method that generates the desired output. The solutions that require specific hardware mostly have blendshape based output, and translating between different blendshape setups is easier. In the end we believe a few different standards will develop, accommodating the different needs for cartoony and realistic animation.

8.3 Research Question 4 & 5

This leads us to our final questions. Is there a way to make it easier to incorporate facial puppetry in games? How could the architecture for a software for this be designed?

It is possible to develop a software that acts like a bridge between different tracking solutions and the target game. The bridge software could allow translation between different blendshape setups. This would simplify the process as developers would only need to implement support for the bridge software, and not for the tracking solutions themselves. A simple Application Layer Protocol could be used to forward the tracking data from the bridge to the target application. This could be a connection-less ALP over UDP, which has been shown to work in both our solution and other solutions. UDP is preferable in applications with real-time interactions since TCP can slow down the data stream several hundreds of milliseconds during re-transmissions, which is a lot in vocal communication. At the moment it is more profitable for companies to develop their own vendor locked solutions, which hinders the development of both facial expression standards and the development of tools that allows for translation between different formats.

A Spaces and Coordinate Systems

This appendix is intended as an overview for the spaces involved in face tracking and what kind of transformations might be needed before the data is useful. The reader can refer to the free to read book by Dunn and Parbery [159] for mathematical details. While it is possible to use tracking data as-is, the vectors are just values after all, the user might run into trouble with wrong scale, mirrored behavior, reversed rotations, and similar oddities when the data is used in its raw form.

Positions and transforms of tracking data are mapped in various spaces using a Cartesian coordinate system. Face expression coefficients are trivial to handle as the values modify the magnitude of vectors in a set. Landmarks or face mesh data on the other hand are represented as sets of vectors, defined by each application in $\mathbb{R}^n$, that correspond to 2D or 3D spaces. The origin and basis vectors in the tracking data might not align with that of the intended application that is going to use the data. Complexity increases when the tracking data contain transformations that represent orientations for features like the head and eyes, especially when the handedness of the coordinate systems differ. Even if the basis vectors and handedness align it does not mean that each implementation use the same basis vector for up, forward or sideways.

It is possible to still get unexpected behavior even when all the details are known as some implementations break convention, for example Unreal Engine with its left handed coordinate system has a mixed set of clockwise and counterclockwise rotations for its Euler angles inputs. Yaw rotation around the Z axis behaves according to the engine's left handed convention while Roll and Pitch around the X and Y axis are reversed.

Table A.1 lists an example of coordinate systems from various software as a frame of reference. The coordinate systems tend to appear in a "view based" or "CAD based" mindset. The view mindset think of the screen as a view into the world where Y represents up and Z aligns with the depth of the scene while CAD based see the world from a top-down perspective where Z aligns with height to match architectural layouts. This trivia is relevant when face tracking is to be applied to a character in its local space as the basis vectors are important for proper alignment, at least for landmarks or transforms. Face expression coefficients are independent from the coordinate system as previously discussed.

One must be careful with what is considered a direction like Left or Right in a coordinate system. Some define Right based on the view frame, like image space, while others define Right based on the scene or object orientation. See Figure A.1 for an example.

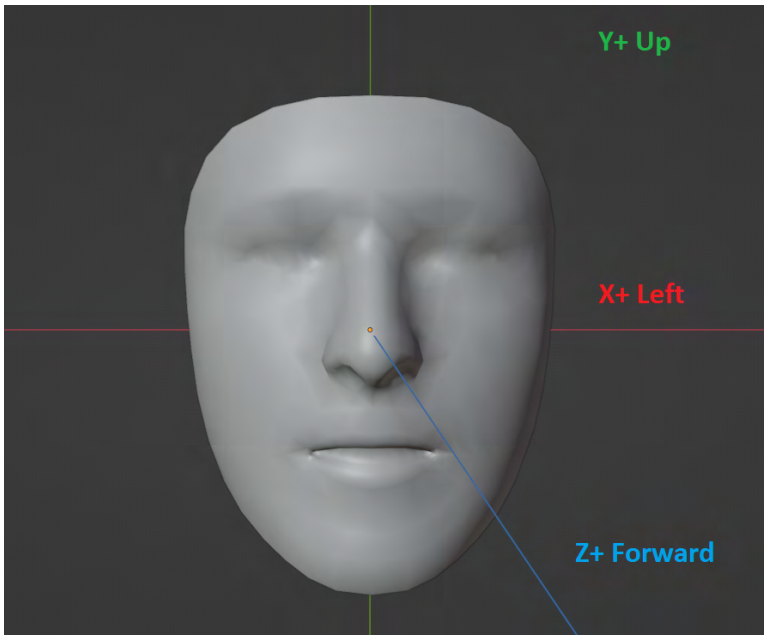

Figure A.1: In some software like game engines the sideway direction is based on the frame of the object and not the viewer. Here the positive direction of the side axis is considered left even though it is right in the camera view. Forward is considered facing the viewer.

Software	Handedness	Left/Right	Up/Down	Forward/Backward	Linear units
Mediapipe	Right	X	Y	Z	Meter
ARKit	Right	X	Y	Z	Meter
Houdini	Right	X	Y	Z	Meter
glTF	Right	X	Y	Z	Meter
Autodesk Maya	Right	X	Y	Z	Centimeter
PyBullet	Right	X	Y	Z	Unitless
Ammo.js	Right	X	Y	Z	Unitless
Three.js	Right	X	Y	Z	Unitless
Verge3D	Right	X	Y	Z	Unitless
Pygfx	Right	X	Y	Z	Unitless
Blender	Right	X	Z	Y	Meter
Autodesk 3ds Max	Right	X	Z	Y	Inch
Unity	Left	X	Y	Z	Meter
Cinema4D	Left	X	Y	Z	Centimeter
Unreal Engine	Left	Y	Z	X	Centimeter
OpenGL	Left/Right	X	Y	Z	Unitless
WebGL	Left/Right	X	Y	Z	Unitless
WebGPU	Left/Right	X	Y	Z	Unitless
DirectX	Left/Right	X	Y	Z	Unitless

Table A.1: An overview of coordinate systems used across different software. Directions are written without a sign (+ or -) as there is a mix in view vs object frame conventions, see Figure A.1 for an example. The handedness for rendering APIs like OpenGL is listed as Left/Right as it is up to the developer to decide how the depth is handled. The linear units are defined based on the default setting of the software and is only listed as unitless when that is the sole convention (applications like Blender have a mix of linear units and meter is the default). Unitless means the developer decides what distance a value represents.

A.1 Image Space

A picture is produced from a capture device in image space using pixel coordinates. The origin is in the upper left corner with the X axis pointing right and the Y axis pointing down. Some rendering APIs like OpenGL draw textures from the bottom left corner and the developer has to flip the image in the shader or the buffer so it does not appear upside down.

Detecting faces, extracting feature landmarks and tracking individuals across frames are typically done in image space. For these computer vision tasks refer to Szeliski [20].

A.2 Normalized Coordinates

The extracted features can be represented in normalized coordinates, sometimes called Normalized Device Coordinates (NDC). Normalized means that points on the image exist within either a $[-1, 1]$ or $[0, 1]$ range, similar to Clip Space in rendering APIs. The name NDC means the normalized range matches the screen of a device regardless of size or proportion. Note that the center of the image is either $(0.5, 0.5)$ or $(0.0, 0.0)$ depending on the normalized range.

A.3 Depth Component

Tracking solutions that solve a depth component for tracked features do so using either depth data or a multi-camera setup where computer vision is used to estimate a 3D location using projective geometry in homogeneous coordinates, see Szeliski [20] or Nordberg [160] for that. It is also possible to use prior knowledge of a canonical face mesh to estimate the depth by fitting it to tracking data that exist on a plane. The depth component then gives landmarks in XYZ coordinates. MediaPipe resolves the Z-coordinate using a canonical head by minimizing the difference to the XY landmarks. The center of the face remains on the XY-plane and the Z-component is returned from the optimized fit. The head is then proportional to what is seen on the screen. Resolving the actual size and Z-position of the head needs additional steps like adjustment based on a known inter-eye distance. ARKit resolves the actual size, proportion, position and orientation of the face mesh in metric units.

A.4 Camera Space / Device Space / World Space

Some tracking solutions use the device to establish a world reference frame where the camera or device is used as the origin, hence the Camera Space or Device Space name. This space is typically in XYZ coordinates. In some cases the Camera Space and Normalized Device Coordinates are equivalent.

The XY plane typically matches the device screen and Z+ points out of the screen towards the viewer. Some solutions like ARKit make use of device sensors to establish real world alignment, for example aligning with gravity.

The target application is likely to use a world XYZ coordinate system to simulate characters for facial puppetry. Going from Camera Space or NDC to the target application world space need additional transforms when the spaces do not align.

A.5 Tracking Internals

Tracking solutions have a large variety of other spaces not mentioned, such as Feature Space, Disparity Space, Latent Space, etc. These are beyond the scope of the report.

B Prototype Example Code

This appendix contains example code for some of the prototype implementations mentioned in Chapter 6.3.1. Listing B.1 is a basic application to print UDP packets. Listing B.2 contains Python code for generating face landmarks, blendshapes and the face transformation matrix using MediaPipe with a webcam feed. The data is then sent over UDP with our FacePipe protocol.

```python
import socket

udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_socket.bind(('127.0.0.1', 9000))

print(f"Listening...\n")

try:
    while True:
        data, addr = udp_socket.recvfrom(65507)
        print(data.decode('utf-8'))
except KeyboardInterrupt:
    pass

udp_socket.close()
print("\nApplication ended")
```

Listing B.1: Python code for printing UDP packets sent by Listing B.2

```python
import cv2
import numpy as np
import mediapipe as mp
import time
import socket
import json

udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_socket.bind(('127.0.0.1', 0))

def to_array_string(arr):
    return json.dumps(arr, separators=(',', ':'))[1:-1] # [1,2,3,4,5] => "1,2,3,4,5"

def on_mp_facelandmarker_result(result: mp.tasks.vision.FaceLandmarkerResult, image: mp.Image, timestamp_ms: int):
    global udp_socket
    protocol, source, scene, camera, time = ("facepipe", 0, 0, 0, float(timestamp_ms)/1000.0)

    for subject in range(0, len(result.face_landmarks)):
        values = to_array_string(np.array([(lm.x, lm.y, lm.z) for lm in result.face_landmarks[subject]]).flatten().tolist())
        content = f"l3d|{image.width},{image.height}|{values}"
        message = f"a|{protocol}|{source}|{scene},{camera},{subject}|{time}|{content}".encode('ascii')
        udp_socket.sendto(message, ('127.0.0.1', 9000))

    for subject in range(0, len(result.face_blendshapes)):
        content = "bs"
        for i in range(0, len(result.face_blendshapes[subject])):
            bs = result.face_blendshapes[subject][i]
            content += f"|{bs.category_name}={bs.score}"
        message = f"a|{protocol}|{source}|{scene},{camera},{subject}|{time}|{content}".encode('ascii')
        udp_socket.sendto(message, ('127.0.0.1', 9000))

    for subject in range(0, len(result.facial_transformation_matrixes)):
        values = to_array_string(result.facial_transformation_matrixes[subject].flatten().tolist())
        content = f"mat44|face={values}"
        message = f"a|{protocol}|{source}|{scene},{camera},{subject}|{time}|{content}".encode('ascii')
        udp_socket.sendto(message, ('127.0.0.1', 9000))

# https://developers.google.com/mediapipe/solutions/vision/face_landmarker/python
options = mp.tasks.vision.FaceLandmarkerOptions(
    base_options = mp.tasks.BaseOptions(model_asset_path='face_landmarker.task'),
    running_mode = mp.tasks.vision.RunningMode.LIVE_STREAM,
    output_face_blendshapes = True,
    output_facial_transformation_matrixes = True,
    result_callback = on_mp_facelandmarker_result)

with mp.tasks.vision.FaceLandmarker.create_from_options(options) as landmarker:
    cv2_webcam_capture = cv2.VideoCapture(0, cv2.CAP_DSHOW) # 0: first webcam in device list

    time_start = time.time()
    while cv2_webcam_capture.isOpened():
        success, cv2_webcam_image = cv2_webcam_capture.read()
        cv2_webcam_image = cv2.flip(cv2_webcam_image, 1)
        mp_webcam_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=cv2_webcam_image)

        frame_timestamp_ms = round((time.time()-time_start)*1000)
        landmarker.detect_async(mp_webcam_image, frame_timestamp_ms)

        cv2.imshow('image', cv2_webcam_image)
        if cv2.waitKey(1) == 27:
            break

udp_socket.close()
```

Listing B.2: Python code for capturing face data with MediaPipe and sent over UDP with our FacePipe protocol. The face_landmarker.task file need to be downloaded from the URL in the code comment for the code to work.

Bibliography

[1] T.A.M. ller, E. Haines, and N. Hoffman. *Real-Time Rendering, Fourth Edition*. CRC Press, 2018. ISBN: 978-1-351-81615-1. URL: https://books.google.se/books?id=0g1mDwAAQBAJ.

[2] H. Nguyen and NVIDIA Corporation. *GPU Gems 3*. Lab Companion Series v. 3. Addison-Wesley, 2008. ISBN: 978-0-321-51526-1. URL: https://books.google.se/books?id=ylNyQgAACAAJ.

[3] Paul Ekman. *Are There Universal Facial Expressions? - Paul Ekman Group*. URL: https://www.paulekman.com/resources/universal-facial-expressions/ (visited on 07/17/2023).

[4] C.H. Hjortsjo. *Man's Face and Mimic Language*. Studentlitteratur, 1969. URL: https://books.google.se/books?id=BakQAQAAIAAJ.

[5] P. Ekman and W. Friesen. "Facial action coding system: a technique for the measurement of facial movement". In: 1978. URL: https://www.semanticscholar.org/paper/Facial-action-coding-system%3A-a-technique-for-the-of-Ekman-Friesen/1566cf20e2ba91ca8857c30083419bf7c127094b (visited on 04/28/2023).

[6] Melinda Ozel. *FACS study guide*. en-US. Aug. 2022. URL: https://melindaozel.com/facs-study-guide/ (visited on 07/12/2023).

[7] Verónica Orvalho, Pedro Bastos, Frederic Parke, Bruno Oliveira, and Xenxo Alvarez. "A Facial Rigging Survey". en. In: (2012). Accepted: 2013-11-08T10:29:33Z Publisher: The Eurographics Association. ISSN: 1017-4656. DOI: 10.2312/conf/EG2012/stars/183-204. URL: https://diglib.eg.org:443/xmlui/handle/10.2312/conf.EG2012.stars.183-204 (visited on 07/11/2023).

[8] J. Osipa. *Stop Staring: Facial Modeling and Animation Done Right*. EBL-Schweitzer. John Wiley & Sons, 2010. ISBN: 978-0-470-60990-3. URL: https://books.google.se/books?id=yPGyBwAAQBAJ.

[9] Binh Huy Le and J P Lewis. "Direct delta mush skinning and variants". In: *ACM Transactions on Graphics* 38.4 (July 2019), 113:1–113:13. ISSN: 0730-0301. DOI: 10.1145/3306346.3322982. URL: https://doi.org/10.1145/3306346.3322982 (visited on 04/29/2023).

[10] Unreal Engine. *MetaHuman Framework & Machine Learning for Next-Gen Character Deformation | GDC 2023*. Apr. 2023. URL: https://www.youtube.com/watch?v=OmMi6E0EkQw (visited on 04/29/2023).

[11] W. Engel. *GPU Pro 2*. EBL-Schweitzer. CRC Press, 2016. ISBN: 978-1-4398-6560-6. URL: https://books.google.se/books?id=zfPRBQAAQBAJ.

[12] J. P. Lewis, Ken Anjyo, Taehyun Rhee, Mengjie Zhang, Fred Pighin, and Zhigang Deng. "Practice and Theory of Blendshape Facial Models". en. In: (2014). Accepted: 2014-12-16T07:12:53Z Publisher: The Eurographics Association. ISSN: 1017-4656. DOI: 10.2312/egst.20141042. URL: https://diglib.eg.org:443/xmlui/handle/10.2312/egst.20141042.199-218 (visited on 07/11/2023).

[13] UE5Docs. *Facial Animation Sharing in Unreal Engine | Unreal Engine 5.2 Documentation*. URL: https://docs.unrealengine.com/5.2/en-US/facial-animation-sharing-in-unreal-engine/ (visited on 08/26/2023).

[14] Noah Schnapp. *Auto Blendshape Transfer Different Topo Course on CGCircuit*. URL: https://www.cgcircuit.com/course/auto-blendshape-transfer-different-topo (visited on 08/26/2023).

[15] Cascadeur. *Cascadeur - the easiest way to animate AI-assisted keyframe animation software*. 2023. URL: https://cascadeur.com/ (visited on 09/29/2023).

[16] EverQuest 2. *SOEmote is Live!* Aug. 2012. URL: https://www.youtube.com/watch?v=cde01HNKQVw (visited on 07/12/2023).

[17] starcitizen. *FOIP, VOIP, and Freelook Guide – Roberts Space Industries Knowledge Base*. URL: https://support.robertsspaceindustries.com/hc/en-us/articles/360009579674-FOIP-VOIP-and-Freelook-Guide (visited on 07/12/2023).

[18] vrchat. *VRChat*. URL: https://hello.vrchat.com/ (visited on 07/12/2023).

[19] Masahiro Mori, Karl MacDorman, and Norri Kageki. "The Uncanny Valley [From the Field]". en. In: *IEEE Robotics & Automation Magazine* 19.2 (June 2012), pp. 98–100. ISSN: 1070-9932. DOI: 10.1109/MRA.2012.2192811. URL: http://ieeexplore.ieee.org/document/6213238/ (visited on 09/29/2023).

[20] Richard Szeliski. *Computer Vision: Algorithms and Applications*. en. Google-Books-ID: QWOOzwEACAAJ. Springer International Publishing, Jan. 2023. ISBN: 978-3-030-34374-3.

[21] hickeys. *Kinect for Windows - Windows apps*. en-us. Aug. 2022. URL: https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows (visited on 07/12/2023).

[22] Microsoft. *Azure Kinect DK – Develop AI Models | Microsoft Azure*. en-US. URL: https://azure.microsoft.com/en-us/products/kinect-dk (visited on 07/12/2023).

[23] intel. *Intel RealSense Computer Vision - Depth and Tracking cameras*. URL: https://www.intelrealsense.com/ (visited on 07/12/2023).

[24] Apple. *Streaming Depth Data from the TrueDepth Camera*. en-US. URL: https://developer.apple.com/documentation/avfoundation/additional_data_capture/streaming_depth_data_from_the_truedepth_camera (visited on 07/12/2023).

[25] Tobii. *Eye trackers for research - See all our models here*. en. URL: https://www.tobii.com/products/eye-trackers/ (visited on 07/12/2023).

[26] Foveated Tobii. *What is foveated rendering?* en. Mar. 2023. URL: https://www.tobii.com/blog/what-is-foveated-rendering (visited on 07/12/2023).

[27] Ultraleap. *Digital worlds that feel human | Ultraleap*. en. URL: https://www.ultraleap.com/ (visited on 07/12/2023).

[28] HTC. *VIVE Facial Tracker | VIVE United States*. en-US. URL: https://www.vive.com/us/accessory/facial-tracker/ (visited on 07/12/2023).

[29] Meta. *Meta Quest Pro: Our most advanced new VR headset | Meta Store*. en. URL: https://www.meta.com/se/en/quest/quest-pro/ (visited on 07/12/2023).

[30] OpenCV. *Home*. en-US. URL: https://opencv.org/ (visited on 07/12/2023).

[31] dlib. *dlib C++ Library*. URL: http://dlib.net/ (visited on 07/12/2023).

[32] ci2cv. *Documentation | Face Analysis SDK*. en-US. URL: https://face.ci2cv.net/doc/ (visited on 07/12/2023).

[33] MediaPipe. *google/mediapipe*. original-date: 2019-06-13T19:16:41Z. July 2023. URL: https://github.com/google/mediapipe (visited on 07/12/2023).

[34] ARCore. *Build new augmented reality experiences that seamlessly blend the digital and physical worlds, ARCore*. en. URL: https://developers.google.com/ar (visited on 07/12/2023).

[35] TensorFlow. *TensorFlow*. en. URL: https://www.tensorflow.org/ (visited on 07/12/2023).

[36] PyTorch. *PyTorch*. en. URL: https://www.pytorch.org (visited on 07/12/2023).

[37] scikit. *scikit-learn: machine learning in Python — scikit-learn 1.3.0 documentation*. URL: https://scikit-learn.org/stable/ (visited on 07/12/2023).

[38] vivesdk. *SDK for developers*. en. URL: https://www.vive.com/eu/support/facial-tracker/category_howto/sdk-for-developers.html (visited on 07/12/2023).

[39] tobiisdk. *Tobii Customer Portal*. en-US. URL: https://connect.tobii.com (visited on 07/12/2023).

[40] Apple arkitsdk. *ARKit 6 - Augmented Reality*. en. URL: https://developer.apple.com/augmented-reality/arkit/ (visited on 07/12/2023).

[41] maxinesdk. *README*. original-date: 2020-03-16T16:07:33Z. July 2023. URL: https://github.com/NVIDIA/MAXINE-AR-SDK (visited on 07/12/2023).

[42] broadcastapi. *NVIDIA Broadcast Software Integrations: Download Resources*. en-us. URL: https://www.nvidia.com/en-us/geforce/broadcasting/broadcast-sdk/resources/ (visited on 07/12/2023).

[43] Lipsync Oculus. *Oculus Lipsync for Unity Development: Unity | Oculus Developers*. URL: https://developer.oculus.com/documentation/unity/audio-ovrlipsync-unity/ (visited on 12/17/2023).

[44] trackir. *TrackIR*. en. URL: https://www.trackir.com (visited on 07/12/2023).

[45] opentrack. *opentrack/opentrack*. original-date: 2013-04-29T17:03:00Z. July 2023. URL: https://github.com/opentrack/opentrack (visited on 07/12/2023).

[46] Alvaro. *AIRLegend/aitrack*. original-date: 2020-07-20T07:43:06Z. July 2023. URL: https://github.com/AIRLegend/aitrack (visited on 07/12/2023).

[47] facetracknoir. *Home*. URL: https://www.facetracknoir.nl/home/default.htm (visited on 07/12/2023).

[48] avatary. *Avatary*. URL: https://world.avatary.com/livedrive (visited on 07/12/2023).

[49] visage. *Face tracking software*. en-GB. URL: https://visagetechnologies.com/facetrack/ (visited on 07/12/2023).

[50] banuba. *Face Tracking Software Video Editor SDK, Banuba Development*. URL: https://www.banuba.com/technology/face-tracking-software (visited on 07/12/2023).

[51] moodme. *Home - MoodMe*. en-US. June 2023. URL: https://www.mood-me.com/ (visited on 07/12/2023).

[52] luxand. *Luxand - Face Recognition, Face Detection and Facial Feature Detection Technologies*. URL: https://www.luxand.com/ (visited on 07/12/2023).

[53] faceware. *Mocap Software - Faceware Technologies, Inc.* en. URL: https://www.facewaretech.com/software (visited on 07/12/2023).

[54] animaze. *Animaze by FaceRig on Steam*. en. URL: https://store.steampowered.com/app/1364390/Animaze_by_FaceRig/ (visited on 07/12/2023).

[55] Akiya Research Institute Akiya. *MocapForAll*. ja-jp. Section: products. URL: https://vrlab.akiya-souken.co.jp/products/mocapforall/ (visited on 07/12/2023).

[56] vtube. *VTube Studio*. en-US. URL: https://denchisoft.com/ (visited on 07/12/2023).

[57] vnyan. *VNyan by Suvidriel*. en. URL: https://suvidriel.itch.io/vnyan (visited on 07/12/2023).

[58] warudo. *Warudo*. URL: https://docs.warudo.app/warudo/ (visited on 07/12/2023).

[59] brekel. *Brekel Face v2*. en-US. URL: https://brekel.com/face_v2/ (visited on 07/12/2023).

[60] ifacialmocap. *HOME - iFacialMocap*. URL: https://www.ifacialmocap.com/ (visited on 07/12/2023).

[61] Sefik Ilkin Serengil. *deepface*. original-date: 2020-02-08T20:42:28Z. July 2023. URL: https://github.com/serengil/deepface (visited on 07/12/2023).

[62] facetracker. *FaceTracker*. URL: https://facetracker.net/ (visited on 07/12/2023).

[63] Marco Pattke. *MeFaMo - MediapipeFaceMocap*. original-date: 2022-03-19T19:43:49Z. July 2023. URL: https://github.com/JimWest/MeFaMo (visited on 07/12/2023).

[64] Emiliana. *Overview*. original-date: 2020-01-01T12:49:35Z. July 2023. URL: https://github.com/emilianavt/OpenSeeFace (visited on 07/12/2023).

[65] faceshift. *Performance driven facial animation*. Aug. 2015. URL: https://www.fxguide.com/fxfeatured/performance-driven-facial-animation/ (visited on 07/12/2023).

[66] Matt Sayward. *Whatever happened with Apple's PrimeSense acquisition?* en. Dec. 2016. URL: https://medium.com/matt-sayward/whatever-happened-with-apple-s-primesense-acquisition-bb7dd950d911 (visited on 07/12/2023).

[67] rokoko. *Quality real-time face capture mocap with your iPhone*. URL: https://www.rokoko.com/products/face-capture (visited on 07/12/2023).

[68] iclone. *iPhone Face Mocap for Facial Motion Capture*. URL: https://mocap.reallusion.com/iclone-motion-live-mocap/iphone-live-face.html (visited on 07/12/2023).

[69] mocapx. *MocapX - Facial Motion Capture App for iPhoneX*. URL: https://www.mocapx.com/ (visited on 07/12/2023).

[70] facecap. *Face Cap - Motion Capture on the App Store*. URL: https://apps.apple.com/us/app/face-cap-motion-capture/id1373155478 (visited on 07/12/2023).

[71] bannaflak. *Face Cap Web*. URL: https://www.bannaflak.com/face-cap/index.html (visited on 07/12/2023).

[72] faceit. *Faceit : Facial Expressions and Performance Capture - Blender Market*. URL: https://blendermarket.com/products/faceit/ (visited on 07/12/2023).

[73] ue5iosdevice. *Recording Facial Animation from an iOS Device | Unreal Engine 4.27 Documentation*. URL: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/FacialRecordingiPhone/ (visited on 07/12/2023).

[74] epiclivelink. *Animating with Live Link | Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/metahuman/animating-metahumans-with-livelink-in-unreal-engine (visited on 07/12/2023).

[75] perfectsync. *About Perfect Sync :: Documentation for Luppet*. URL: https://luppet-document.web.app/en/faq/perfect-sync/ (visited on 07/12/2023).

[76] vmagicmirror. *Perfect Sync | VMagicMirror*. URL: https://malaybaku.github.io/VMagicMirror/en/tips/perfect_sync/ (visited on 07/12/2023).

[77] arkitrefpage. *ARKit Blendshapes Reference Page*. URL: https://arkit-face-blendshapes.com/ (visited on 07/12/2023).

[78] Polywink Polywink. *Polywink - Automatic Expressions Blendshapes and Facial Rigs*. URL: https://polywink.com/en/ (visited on 12/17/2023).

[79] kalidokit. *yeemachine/kalidokit: Blendshape and kinematics calculator for Mediapipe/Tensorflow.js Face, Eyes, Pose, and Finger tracking models*. URL: https://github.com/yeemachine/kalidokit (visited on 07/12/2023).

[80] hallway. *Hallway | Express your true self*. URL: https://joinhallway.com/ (visited on 07/12/2023).

[81] phiz. *SpookyCorgi/phiz: Phiz is a tool that allows you to perform facial motion capture from any device and location*. URL: https://github.com/SpookyCorgi/phiz (visited on 07/12/2023).

[82] kalidoface. *Kalidoface Lab*. URL: https://lab.kalidoface.com/ (visited on 07/12/2023).

[83] blendarmocap. *cgtinker/BlendArMocap: realtime motion tracking in blender using mediapipe and rigify*. URL: https://github.com/cgtinker/BlendArMocap (visited on 07/12/2023).

[84] gameface. *google/project-gameface*. URL: https://github.com/google/project-gameface (visited on 07/12/2023).

[85] vrpn. *VRPN - Virtual Reality Peripheral Network*. URL: https://vrpn.github.io/ (visited on 07/12/2023).

[86] osvr. *OSVR Developer Portal | Home*. URL: https://osvr.github.io/ (visited on 07/12/2023).

[87] steamvr. *SteamVR on Steam*. URL: https://store.steampowered.com/app/250820/SteamVR/ (visited on 07/12/2023).

[88] openvr. *ValveSoftware/openvr: OpenVR SDK*. URL: https://github.com/ValveSoftware/openvr (visited on 07/12/2023).

[89] wmr. *Microsoft Virtual Reality – Experiences and Devices*. URL: https://www.microsoft.com/en-us/mixed-reality/windows-mixed-reality (visited on 07/12/2023).

[90] OpenXR. *The OpenXR™ Specification*. URL: https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html (visited on 07/12/2023).

[91] OpenXR. *Quickstart | OpenXR Toolkit*. URL: https://mbucchia.github.io/OpenXR-Toolkit/ (visited on 07/12/2023).

[92] oscmain. *OSC index*. URL: https://opensoundcontrol.stanford.edu/ (visited on 07/12/2023).

[93] VMCmain. *VMC Protocol specification | VirtualMotionCaptureProtocol*. URL: https://protocol.vmc.info/english.html (visited on 07/12/2023).

[94] vseeface. *VSeeFace*. URL: https://www.vseeface.icu/ (visited on 07/12/2023).

[95] vrcfacetracking. *benaclejames/VRCFaceTracking: VRChat OSC App to allow AV3 Avatars to interact with Vive SRanipal Eye and Lip Tracking SDK*. URL: https://github.com/benaclejames/VRCFaceTracking (visited on 07/12/2023).

[96] unifiedexpressions. *Unified Expressions | VRCFaceTracking*. URL: https://docs.vrcft.io/docs/tutorial-avatars/tutorial-avatars-extras/unified-blendshapes (visited on 07/12/2023).

[97] vrcfttovmcp. *tkns3/VRCFTtoVMCP: VRCFaceTracking*. URL: https://github.com/tkns3/VRCFTtoVMCP (visited on 07/12/2023).

[98] vmc. *VMC Protocol specification | VirtualMotionCaptureProtocol*. URL: https://protocol.vmc.info/english (visited on 07/17/2023).

[99] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. *BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs*. arXiv:1907.05047 [cs]. July 2019. DOI: 10.48550/arXiv.1907.05047. URL: http://arxiv.org/abs/1907.05047 (visited on 07/05/2023).

[100] mediapipelandmark. *Face landmark detection guide | MediaPipe | Google for Developers*. URL: https://developers.google.com/mediapipe/solutions/vision/face_landmarker (visited on 07/12/2023).

[101] Ralph Gross, Iain Matthews, Jeffrey Cohn, Takeo Kanade, and Simon Baker. "Multi-PIE". en. In: *Image and Vision Computing* 28.5 (May 2010), pp. 807–813. ISSN: 02628856. DOI: 10.1016/j.imavis.2009.08.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0262885609001711 (visited on 07/10/2023).

[102] Ivan Grishchenko, Artsiom Ablavatski, Yury Kartynnik, Karthik Raveendran, and Matthias Grundmann. *[2006.10962] Attention Mesh: High-fidelity Face Mesh Prediction in Real-time*. URL: https://arxiv.org/abs/2006.10962 (visited on 07/16/2023).

[103] Nannan Wang, Xinbo Gao, Dacheng Tao, and Xuelong Li. *Facial Feature Point Detection: A Comprehensive Survey*. arXiv:1410.1037 [cs]. Oct. 2014. DOI: 10.48550/arXiv.1410.1037. URL: http://arxiv.org/abs/1410.1037 (visited on 07/11/2023).

[104] Yue Wu and Qiang Ji. "Facial Landmark Detection: a Literature Survey". In: *International Journal of Computer Vision* 127.2 (Feb. 2019). arXiv:1805.05563 [cs], pp. 115–142. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-018-1097-z. URL: http://arxiv.org/abs/1805.05563 (visited on 07/11/2023).

[105] Kostiantyn Khabarlak and Larysa Koriashkina. "Fast Facial Landmark Detection and Applications: A Survey". In: *Journal of Computer Science and Technology* 22.1 (Apr. 2022). arXiv:2101.10808 [cs], e02. ISSN: 1666-6038, 1666-6046. DOI: 10.24215/16666038.22.e02. URL: http://arxiv.org/abs/2101.10808 (visited on 07/11/2023).

[106] Adrian Bulat and Georgios Tzimiropoulos. "How far are we from solving the 2D & 3D Face Alignment problem? (and a dataset of 230,000 3D facial landmarks)". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. arXiv:1703.07332 [cs]. Oct. 2017, pp. 1021–1030. DOI: 10.1109/ICCV.2017.116. URL: http://arxiv.org/abs/1703.07332 (visited on 07/10/2023).

[107] Christos Sagonas, Epameinondas Antonakos, Georgios Tzimiropoulos, Stefanos Zafeiriou, and Maja Pantic. "300 Faces In-The-Wild Challenge: database and results". en. In: *Image and Vision Computing* 47 (Mar. 2016), pp. 3–18. ISSN: 02628856. DOI: 10.1016/j.imavis.2016.01.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0262885616000147 (visited on 07/10/2023).

[108] Prasanth Chandran, Gaspard Zoss, Paulo Gotardo, and Derek Bradley. *Continuous Landmark Detection with 3D Queries | Disney Research Studios*. URL: https://studios.disneyresearch.com/2023/06/04/continuous-landmark-detection-with-3d-queries/ (visited on 07/10/2023).

[109] Gaspard Zoss, Derek Bradley, Pascal Bérard, and Thabo Beeler. *An Empirical Rig for Jaw Animation | Disney Research Studios*. URL: https://studios.disneyresearch.com/2018/07/30/an-empirical-rig-for-jaw-animation/ (visited on 07/10/2023).

[110] Gaspard Zoss, Thabo Beeler, Markus Gross, and Derek Bradley. *Accurate Markerless Jaw Tracking for Facial Performance Capture | Disney Research Studios*. URL: https://studios.disneyresearch.com/2019/07/12/accurate-markerless-jaw-tracking-for-facial-performance-capture/ (visited on 07/10/2023).

[111] Chen Cao, Qiming Hou, and Kun Zhou. "Displaced dynamic expression regression for real-time facial tracking and animation". en. In: *ACM Transactions on Graphics* 33.4 (July 2014), pp. 1–10. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2601097.2601204. URL: https://dl.acm.org/doi/10.1145/2601097.2601204 (visited on 07/10/2023).

[112] Dirk Colbry and George Stockman. "Canonical Face Depth Map: A Robust 3D Representation for Face Verification". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition* (June 2007). Conference Name: 2007 IEEE Conference on Computer Vision and Pattern Recognition ISBN: 9781424411795 9781424411801 Place: Minneapolis, MN, USA Publisher: IEEE, pp. 1–7. DOI: 10.1109/CVPR.2007.383108. URL: http://ieeexplore.ieee.org/document/4270133/ (visited on 07/14/2023).

[113] CANDIDE. *CANDIDE on Wayback Machine*. URL: https://web.archive.org/web/20220412030346/http://www.icg.isy.liu.se/candide/ (visited on 07/12/2023).

[114] mediapipe. *Face Mesh documentation for MediaPipe*. en. URL: https://github.com/google/mediapipe/blob/master/docs/solutions/face_mesh.md (visited on 07/14/2023).

[115] mpcanonicalfacemesh. *mediapipe/mediapipe/modules/face_geometry/data at master · google/mediapipe · GitHub*. URL: https://github.com/google/mediapipe/tree/master/mediapipe/modules/face_geometry/data (visited on 07/14/2023).

[116] applecanonicalfacemesh. *Tracking and Visualizing Faces*. en-US. URL: https://developer.apple.com/documentation/arkit/arkit_in_ios/content_anchors/tracking_and_visualizing_faces (visited on 07/14/2023).

[117] Volker Blanz and Thomas Vetter. "A morphable model for the synthesis of 3D faces". en. In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*. Not Known: ACM Press, 1999, pp. 187–194. ISBN: 978-0-201-48560-8. DOI: 10.1145/311535.311556. URL: http://portal.acm.org/citation.cfm?doid=311535.311556 (visited on 07/17/2023).

[118] Jason M. Saragih, Simon Lucey, and Jeffrey F. Cohn. "Deformable Model Fitting by Regularized Landmark Mean-Shift". en. In: *International Journal of Computer Vision* 91.2 (Jan. 2011), pp. 200–215. ISSN: 1573-1405. DOI: 10.1007/s11263-010-0380-4. URL: https://doi.org/10.1007/s11263-010-0380-4 (visited on 07/05/2023).

[119] Chen Cao, Derek Bradley, Kun Zhou, and Thabo Beeler. *Real-Time High-Fidelity Facial Performance Capture*. en-US. URL: https://la.disneyresearch.com/publication/realtimeperformancecapture/ (visited on 07/10/2023).

[120] Tadas Baltrušaitis, Marwa Mahmoud, and Peter Robinson. "Cross-dataset learning and person-specific normalisation for automatic Action Unit detection". In: *2015 11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition (FG)*. Vol. 06. May 2015, pp. 1–6. DOI: 10.1109/FG.2015.7284869.

[121] Melinda Ozel. *ARKit to FACS cheat sheet – Face the FACS*. URL: https://melindaozel.com/arkit-to-facs-cheat-sheet/ (visited on 07/17/2023).

[122] Jason M. Saragih, Simon Lucey, and Jeffrey F. Cohn. "Real-time avatar animation from a single image". In: *2011 IEEE International Conference on Automatic Face & Gesture Recognition (FG)*. Mar. 2011, pp. 117–124. DOI: 10.1109/FG.2011.5771383.

[123] Thibaut Weise, Hao Li, Luc Van Gool, and Mark Pauly. "Face/off: Live facial puppetry". In: *Computer Animation, Conference Proceedings* (Aug. 2009). DOI: 10.1145/1599470.1599472.

[124] metahumananimator. *Animating MetaHumans | Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/metahuman/animating-metahumans-in-unreal-engine-5 (visited on 07/17/2023).

[125] Robert W Sumner and Jovan Popovic. "Deformation Transfer for Triangle Meshes". en. In: ().

[126] Xianmei Wan and Xiaogang Jin. "Data-driven facial expression synthesis via Laplacian deformation". In: *Multimedia Tools and Applications* 58 (Jan. 2010), pp. 109–123. DOI: 10.1007/s11042-010-0688-7.

[127] Ludovic Dutreve, Alexandre Meyer, and Saida Bouakaz. "Feature points based facial animation retargeting". In: *Symposium on Virtual reality software and technology (VRST)*. Bordeaux, France, 2008, pp. 197–197. DOI: 10.1145/1450579.1450621. URL: https://hal.science/hal-01494990 (visited on 09/29/2023).

[128] Shaojun Bian, Anzong Zheng, Lin Gao, Greg Maguire, Willem Kokke, Jon Macey, Lihua You, and Jian J. Zhang. "Fully Automatic Facial Deformation Transfer". en. In: *Symmetry* 12.1 (Jan. 2020). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 27. ISSN: 2073-8994. DOI: 10.3390/sym12010027. URL: https://www.mdpi.com/2073-8994/12/1/27 (visited on 09/29/2023).

[129] Jiashun Wang, Chao Wen, Yanwei Fu, Haitao Lin, Tianyun Zou, Xiangyang Xue, and Yinda Zhang. *Neural Pose Transfer by Spatially Adaptive Instance Normalization*. arXiv:2003.07254 [cs]. May 2020. URL: http://arxiv.org/abs/2003.07254 (visited on 10/03/2023).

[130] Richard A. Roberts, Rafael Kuffner Dos Anjos, Akinobu Maejima, and Ken Anjyo. "Deformation transfer survey". en. In: *Computers & Graphics* 94 (Feb. 2021), pp. 52–61. ISSN: 00978493. DOI: 10.1016/j.cag.2020.10.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0097849320301552 (visited on 09/29/2023).

[131] W. Richard Stevens. *UNIX Network Programming: Interprocess communications*. en. Google-Books-ID: CoQ_AQAAIAAJ. Prentice Hall PTR, 1998. ISBN: 978-0-13-081081-6.

[132] Gregory Vodden. *Our Mouse Control Tests: Click Latency*. en-US. URL: https://www.rtings.com/mouse/tests/control/latency (visited on 06/11/2023).

[133] International Telecommunication Union. *G.114 : One-way transmission time*. URL: https://www.itu.int/rec/T-REC-G.114 (visited on 10/01/2023).

[134] International Telecommunication Union. *BT.1359 : Relative timing of sound and vision for broadcasting*. URL: https://www.itu.int/rec/R-REC-BT.1359/en (visited on 10/01/2023).

[135] microsoft. *Sleep function (synchapi.h) - Win32 apps | Microsoft Learn*. URL: https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-sleep (visited on 07/15/2023).

[136] Jun-Ho Huh. "Reliable User Datagram Protocol as a Solution to Latencies in Network Games". en. In: *Electronics* 7.11 (Nov. 2018). Number: 11 Publisher: Multidisciplinary Digital Publishing Institute, p. 295. ISSN: 2079-9292. DOI: 10.3390/electronics7110295. URL: https://www.mdpi.com/2079-9292/7/11/295 (visited on 06/11/2023).

[137] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated: The protocols*. en. Google-Books-ID: 7e9SAAAAMAAJ. Addison-Wesley, 1994. ISBN: 978-0-201-63346-7.

[138] GameNetworkingSockets. *ValveSoftware/GameNetworkingSockets: Reliable & unreliable messages over UDP. Robust message fragmentation & reassembly. P2P networking / NAT traversal. Encryption.* URL: https://github.com/ValveSoftware/GameNetworkingSockets (visited on 08/26/2023).

[139] Valter Roesler, Eduardo Barrére, and Roberto Willrich, eds. *Special Topics in Multimedia, IoT and Web Technologies*. en. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-35101-4. DOI: 10.1007/978-3-030-35102-1. URL: http://link.springer.com/10.1007/978-3-030-35102-1 (visited on 07/03/2023).

[140] Glenn Fiedler. *Why can't I send UDP packets from a browser?* en-us. Feb. 2017. URL: https://gafferongames.com/post/why_cant_i_send_udp_packets_from_a_browser/ (visited on 06/11/2023).

[141] Casey Muratori. *No really, why can't we have raw UDP in JavaScript?* en. Feb. 2023. URL: https://www.computerenhance.com/p/no-really-why-cant-we-have-raw-udp (visited on 06/11/2023).

[142] webrtc. *WebRTC: Real-Time Communication in Browsers*. URL: https://www.w3.org/TR/webrtc/ (visited on 07/15/2023).

[143] mqtt. *MQTT V3.1 Protocol Specification*. URL: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html (visited on 07/15/2023).

[144] amqp. *Home | AMQP*. URL: https://www.amqp.org/ (visited on 07/15/2023).

[145] ZeroMQ. *ZeroMQ*. URL: https://zeromq.org/ (visited on 09/07/2023).

[146] nanomsg. *About Nanomsg*. URL: https://nanomsg.org/ (visited on 09/07/2023).

[147] nng. *nanomsg/nng: nanomsg-next-generation – light-weight brokerless messaging*. URL: https://github.com/nanomsg/nng (visited on 09/07/2023).

[148] Matthew Wright and Adrian Freed. *OSC 1997 Open SoundControl A New Protocol for Communicating with Sound Synthesizers*. May 2021. URL: https://opensoundcontrol.stanford.edu/publications/1997-Open-SoundControl-A-New-Protocol-for-Communicating-with-Sound-Synthesizers.html (visited on 07/04/2023).

[149] oscspec. *OSC spec 1_0*. URL: https://opensoundcontrol.stanford.edu/spec-1_0.html (visited on 07/12/2023).

[150] nannou. *Home | Nannou*. URL: https://nannou.cc/ (visited on 07/17/2023).

[151] openframeworks. *openFrameworks*. URL: https://openframeworks.cc/ (visited on 07/17/2023).

[152] openrndr. *OPENRNDR*. URL: https://openrndr.org/ (visited on 07/17/2023).

[153] vrchatosc. *OSC Overview*. URL: https://docs.vrchat.com/docs/osc-overview (visited on 07/17/2023).

[154] vrm. *VRM documentation*. URL: https://vrm.dev/en/ (visited on 07/17/2023).

[155] vmcreference. *VMC Protocol | VirtualMotionCaptureProtocol*. URL: https://protocol.vmc.info/Reference (visited on 07/17/2023).

[156] Nicolas Wagner, Ulrich Schwanecke, and Mario Botsch. *Neural Volumetric Blendshapes: Computationally Efficient Physics-Based Facial Blendshapes*. arXiv:2212.14784 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2212.14784. URL: http://arxiv.org/abs/2212.14784 (visited on 07/05/2023).

[157] Jingying Wang, Yilin Qiu, Keyu Chen, Yu Ding, and Ye Pan. "Fully Automatic Blendshape Generation for Stylized Characters". en. In: *2023 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*. Shanghai, China: IEEE, Mar. 2023, pp. 347–355. ISBN: 9798350348156. DOI: 10.1109/VR55154.2023.00050. URL: https://ieeexplore.ieee.org/document/10108478/ (visited on 09/25/2023).

[158] DeepFaceLive DeepFaceLive. *iperov/DeepFaceLive: Real-time face swap for PC streaming or video calls*. URL: https://github.com/iperov/DeepFaceLive (visited on 12/17/2023).

[159] Fletcher Dunn and Ian Parbery. *3D Math Primer for Graphics and Game Development*. URL: https://gamemath.com/ (visited on 08/25/2023).

[160] Klas Nordberg. "Introduction to Representations and Estimation in Geometry". eng. In: (2018). Publisher: Linköping University Electronic Press. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-140019 (visited on 04/29/2023).