MARVisT: Authoring Glyph-based Visualization in Mobile Augmented Reality

Zhutian Chen, Yijia Su, Yifang Wang, Qianwen Wang, Huamin Qu, Yingcai Wu

APPENDIX A
EXPLANATION OF TECHNICAL DETAILS

We present an example to demonstrate the technical details of **Visual Scales Synchronization** and **Virtual Glyphs Auto-layout** and how a designer can use MARVisT to create a keyboard frequency sculpture in a few minutes.

A.1 Visual Scales Synchronization

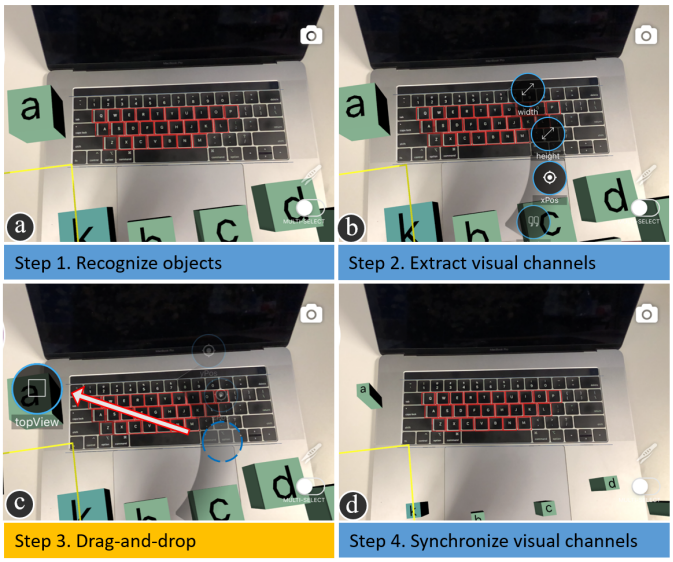

Fig. 1. Four steps to synchronizing the visual scales between real objects and virtual glyphs. The blue steps (Step 1, 2, and 4) are finished by MARVisT. The yellow step (Step 3) is done by the user.

After encoding the typing frequency using the height and color of a bar (in Figure 1), the designer needs to adjust the size of a bar to be the same as that of a keycap. Based on the basic interactions introduced in Section 4.2 of the paper, the designer has to manually modify the size of a bar in a trial and error manner. MARVisT provides an advanced method to help users finish this kind of task (i.e., synchronize visual scales between virtual glyphs and real objects) in four steps:

1) **Detect real objects**. MARVisT leverages several object detection methods provided by ARKit to recognize real objects in the current camera images as much as possible. Specifically, the methods we used include 3D object detection [2], image detection [1], and object detection based on Vision frame work [3], [5] (wherein the model was trained in Turi Create using Darknet YOLO). All of these methods are embedded in ARKit so it is easy to use them by calling their APIs. Once a real object is successfully detected, it will be highlighted with a flicker effect (in Figure 1a). All the detected real objects will be stored and passed to the next step. We further elaborate the details of this step in Figure 2 in the form of a JavaScript-style pseudo code with documentation.

```
/**
 * MARVisT leverages several object detection methods provided by ARKit to
 * recognize real objects in the current camera images as much as possible.
 * Specifically, the methods we used including 3D object detection[1],
 * image detection[2], and object detection based on Vision frame work[3,4]
 * (wherein the model was trained in Turi Create using Darknet YOLO).
 * All of these methods are embedded in ARKit so it is easy to use them
 * by calling their APIs.
 *
 * [1] https://developer.apple.com/documentation/arkit/scanning_and_detecting_3d
 *     _objects
 * [2] https://developer.apple.com/documentation/arkit/recognizing_images_in_an
 *     _ar_experience
 * [3] https://developer.apple.com/documentation/arkit/using_vision_in_real_time
 *     _with_arkit
 * [4] https://developer.apple.com/documentation/vision/recognizing_objects_in
 *     _live_capture
 *
 * @param {ImageFrame} frame. The image frame of the current perspective.
 * @returns {RealObject[]} An array of detected real objects with their
 *      positions and bounding box.
 */
function DetectRealObjects(frame) {
    // 1. Recognize objects from the current frame using the 3D Object
    //    Detection[1]. Specifically, after enabling this feature, we can get
    //    the detected 3D object in the rendere(_:didAdd:for:) of
    //    ARSCNViewDelegate whenever there is an image detected.
    const realObjects1 = objectDetection(frame)

    // 2. Recognize object from the current frame using the Image Detection[2].
    //    Specifically, after enabling this feature, we can get the detected
    //    image in the rendere(_:didAdd:for:) of ARSCNViewDelegate whenever
    //    there is an image detected.
    const realObjects2 = imageDetection(frame)

    // 3. Recognize objects from the current frame using Vision framework[3, 4].
    //    Specifically, we pass the current image frame to a Core ML (another
    //    frame work provided by Apple) object detection model, which is trained
    //    in Turi Create using Darknet YOLO, in the session(_:didUpdate:) of
    //    ARSessionDelegate to recognize objects.
    const realObjects3 = coreML(frame)

    return [].concat(realObjects1, realObjects2, realObjects3)
}
```

Fig. 2. The JavaScript-style pseudo code of the detection of real objects. The pseudo code elaborates the main process of how this function works.

- *Z. Chen is with the State Key Lab of CAD&CG, Zhejiang University and Hong Kong University of Science and Technology. A part of this work was done when Zhutian Chen was a visiting student supervised by Yingcai Wu at the State Key Lab of CAD&CG, Zhejiang University. E-mail: zhutian.chen@connect.ust.hk.*
- *Y. Su, Y. Wang, Q. Wang and H. Qu are with Hong Kong University of Science and Technology. E-mail: {yijiasu, yifangwang, qwangbb}@connect.ust.hk, huamin@cse.ust.hk.*
- *Y. Wu is with the State Key Lab of CAD&CG, Zhejiang University. Yingcai Wu is the corresponding author. E-mail: ycwu@zju.edu.cn.*

2) **Extract visual channels.** After recognizing real objects in the current camera images, MARVisT will try to extract the visual channels of each real object as much as possible. Specifically, the *position* channels (*x, y, z*) in the world coordinate of the AR environment can be extracted once the real object is detected; the *size* channels (*1D-length, 2D-area, 3D-volume*) are estimated based on the real object's bounding box, which is detected in the previous step; the *text* channel is detected and extracted using the Vision framework [4] provided by Apple and only the text with the largest area will be used. Not all the visual channels can always be extracted. MARVisT will display the available visual channels when the user taps on a detected real object (in Figure 1b). We further elaborate the details of this step in Figure 3 in the form of the JavaScript-style pseudo code with documentation.

```
/**
 * After recognizing real objects in the current camera images, MARVisT will try
 * to extract visual channels of each real object as much as possible.
 * Specifically, the position channels (x, y, z) in the world coordinate of the
 * AR environment can be extracted once the real object is detected; the size
 * channels (1D-length, 2D-area, 3D-volume) are estimated based on the real
 * object's bounding box, which is detected in the previous step; the text
 * channel is detected and extracted using the Vision framework[1]
 * provided by Apple. Only the text with the largest area will be used.
 * Not all the visual channels can always be extracted. MARVisT will display the
 * available visual channels when the user taps on a detected real object.
 *
 * [1] https://developer.apple.com/documentation/vision/detecting_objects_in
 *     _still_images
 *
 * @param {RealObject[]} realObjects The array of real objects.
 * @returns {RealObjectWithVisualChannels[]} An array of real objects with their
 *     visual channels.
 */
function ExtractVisualChannels(realObjects) {
    for (const realObject of realObjects) {
        // 1. Extract the position channels based on the position in the world
        // coordinate of the AR environment.
        realObject.channels.posX = extractPosX(realObject)
        realObject.channels.posY = extractPosY(realObject)
        realObject.channels.posZ = extractPosZ(realObject)

        // 2. Extract the size channels based on the bounding box of the object.
        // If any size channel cannot be extract, `null` will be assigned to it.
        realObject.channels.length = extractLength(realObject)
        realObject.channels.width = extractWidth(realObject)
        realObject.channels.height = extractHeight(realObject)
        realObject.channels.topView = computeTopView(realObject)
        realObject.channels.sideView = computeSideView(realObject)
        realObject.channels.frontView = computeFrontView(realObject)
        realObject.channels.volume = computeVolume(realObject)

        // 3. Extract the text channel from the real object using the Vision
        // framework[1] if possible. Only the text with the largest area will
        // be used.
        realObject.channels.text = extractText(realObject)
    }
    return realObjects
}
```

Fig. 3. The pseudo code elaborates the main process of how to extract the visual channels of real objects.

3) **Assign visual channels.** This step is finished by the user. When the user taps on a detected real object, a single-ring semi-annulus similar to the one of the virtual glyphs will pop up. The semi-annulus (in Figure.1a) consists of beads which represent the visual channels of the real objects. The user can drag a bead and drop it on a virtual glyph to assign its value to the corresponding visual channel of the virtual glyph (in Figure 1c).
4) **Synchronize visual scales.** If the visual channel has not been used to encode data attributes, MARVisT will automatically assign the value of the visual channel of the real object to all virtual glyphs of the same type. If this visual channel has already been used to encode a data attribute, MARVisT will inversely calculate the new scale based on the value of the real object's visual channel and the data bounded with the virtual glyph. Then the new scale will automatically be propagated to all virtual glyphs of the same visual mapping, leading to updates of the corresponding virtual glyphs' visual channels (in Figure 1d). We further elaborate the details of this step in Figure 4 in the form of a JavaScript-style pseudo code with documentation.

```
/**
 * After recognizing real objects in the current camera images, MARVisT will try
 * to extract visual channels of each real object as much as possible.
 * Specifically, the position channels (x, y, z) in the world coordinate of the
 * AR environment can be extracted once the real object is detected; the size
 * channels (1D-length, 2D-area, 3D-volume) are estimated based on the real
 * object's bounding box, which is detected in the previous step; the text
 * channel is detected and extracted using the Vision framework[1]
 * provided by Apple. Only the text with the largest area will be used.
 * Not all the visual channels can always be extracted. MARVisT will display the
 * available visual channels when the user taps on a detected real object.
 *
 * [1] https://developer.apple.com/documentation/vision/detecting_objects_in
 *     _still_images
 *
 * @param {RealObject[]} realObjects The array of real objects.
 * @returns {RealObjectWithVisualChannels[]} An array of real objects with their
 *     visual channels.
 */
function ExtractVisualChannels(realObjects) {
    for (const realObject of realObjects) {
        // 1. Extract the position channels based on the position in the world
        // coordinate of the AR environment.
        realObject.channels.posX = extractPosX(realObject)
        realObject.channels.posY = extractPosY(realObject)
        realObject.channels.posZ = extractPosZ(realObject)

        // 2. Extract the size channels based on the bounding box of the object.
        // If any size channel cannot be extract, `null` will be assigned to it.
        realObject.channels.length = extractLength(realObject)
        realObject.channels.width = extractWidth(realObject)
        realObject.channels.height = extractHeight(realObject)
        realObject.channels.topView = computeTopView(realObject)
        realObject.channels.sideView = computeSideView(realObject)
        realObject.channels.frontView = computeFrontView(realObject)
        realObject.channels.volume = computeVolume(realObject)

        // 3. Extract the text channel from the real object using the Vision
        // framework[1] if possible. Only the text with the largest area will
        // be used.
        realObject.channels.text = extractText(realObject)
    }
    return realObjects
}
```

Fig. 4. The pseudo code elaborates the main process of how the final step synchronizes visual scales.

A.2 Virtual Glyphs Auto-layout

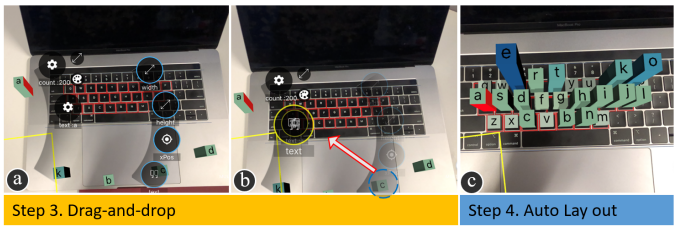

Fig. 5. Steps for auto-layout. The first and second steps are the same as those in Figure 1. a) Tap on any keycap to open the semi-annulus of a real object and tap on any 3D bar to open the semi-annulus of a virtual glyph. b) Map the keycaps to 3D bars based on the text channel of the keycaps and the name attribute of the 3D bars. c) Automatically place each 3D bar onto its corresponding keycap.

After adjusting the size of the bars to the size of keycaps, the designer needs to place each bar onto its corresponding keycap. It is quite tedious for the user to manually move each virtual bar onto its physical referent. MARVisT provides an automated method to help users finish this kind of task in four steps:

1) **Recognize real objects**. This step is the same as the one introduced in section A.1 and can reuse the results.
2) **Extract visual channels.** This step is the also same as the one introduced in section A.1 and can also reuse the results.

3) **Map visual channels**. This step is done by the user. To map real objects to virtual glyphs, MARVisT supports the user to use visual channels as the mapping key. For example, the user wants to map the 3D bars, which encode the typing frequency with the height and color channels, to their corresponding keycaps. After detecting the keycaps from the keyboard, MARVisT can extract several visual channels of the keycaps, such as width, height, and text. The user can simultaneously open the semi-annulus of the virtual glyphs and the real objects (in Figure 5a). Then the user can drag the bead of the selected visual channel of the real objects (e.g., the text channel) and drop it onto the bead of the selected data attribute (e.g., the name of the keycap) of the virtual glyphs to specify the mapping relationship (in Figure 5b). In Figure 5, a keycap will be mapped to the 3D bar whose name is equal the text channel on the keycap.
4) **Lay out virtual glyphs**. After the user specifies the mapping between real objects and virtual glyphs, MARVisT will automatically place each virtual glyph to its corresponding real object (in Figure 5c), whose position in the world coordinate of the AR environment is known after being detected. The details of this step are elaborated in Figure 6 in the form of a JavaScript-style pseudo code with documentation.

```
/**
 * To map real objects to virtual glyphs, MARVisT supports using visual channels
 * as the mapping key. For example, the user wants to map the 3D bars, which
 * encode the typing frequency with the height and color channels, to their
 * corresponding key caps. After detecting the key caps from the keyboard,
 * MARVisT can extract several visual channels of the key caps, such as width,
 * height, and text. The user can simultaneously open the semi-annulus of
 * virtual glyphs and real objects. Then the user can drag the bead of the
 * selected visual channel of real objects (e.g., the text channel) and drop it
 * on the bead of the selected data attribute (e.g., the name of the key cap) of
 * virtual glyphs to specify the mapping relationship. A key cap will be mapped
 * to the 3D bar whose name equals to the text channel of the key cap.
 *
 * @param {String} targetedAttribute The data attribute of virtual glyphs to
 *  perform the join.
 * @param {String} targetedChannel The visual channel of real objects to
 *  perform the join.
 * @param {RealObject[]} realObjects The array of real objects.
 * @param {VirtualGlyph[]} virtualGlyphs The array of virtual glyphs.
 */
function AutoLayoutVirtualGlyph(targetedAttribute, targetedChannel, realObjects,
    virtualGlyphs){
    // 1. Check whether the targetedAttribute or targetedChannel is categorical
    const isCategorical = checkCategorical(targetedAttribute)
        || checkCategorical(targetedChannel)

    // 2.1 If none of them is categorical, then map the real objects to virtual
    // glyphs based on the indices of the values of real objects' targetedChan-
    // -nel and of virtual glyphs' targetedAttribute
    if(!isCategorical) {
        const sortedVgs = virtualGlyphs.sortBy(targetedAttribute)
        const sortedRos = realObjects.sortBy(targetedChannel)
        for(let i = 0, len = sortedVgs.length; i<len; ++i){
            const vg = sortedVgs[i]
            const ro = sortedRos[i]
            vg.channels.posX = ro.channels.posX
            vg.channels.posY = ro.channels.posY
            vg.channels.posZ = ro.channels.posZ
        }
    }
    // 2.2 If both of them are categorical, then map the real objects to virtual
    // glyphs based on the vlaues of real objects' targetedChannel and of
    // virtual glyphs' targetedAttribute
    else {
        for(let i = 0, len = sortedVgs.length; i<len; ++i){
            const vg = virtualGlyphs[i]
            const ro = realObjects.find(ro => ro.channels[targetedChannel] ==
                vg.attributes[targetedAttribute])
            vg.channels.posX = ro.channels.posX
            vg.channels.posY = ro.channels.posY
            vg.channels.posZ = ro.channels.posZ
        }
    }
}
```

Fig. 6. The pseudo code elaborates the main process of how MARVisT places virtual glyphs to their physical referents automatically after the user has specified the mappings between them.

A.3 Supplemental Details

For the examples in the paper, we provide more details of the ping pong ball example of the visual scales synchronization in Figure 7 and the sugar stack example of the virtual glyphs auto-layout in Figure 8.

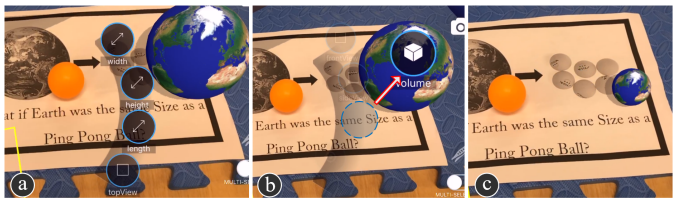

Fig. 7. a) After tapping on the ping pong ball, a single-ring semi-annulus which consists of beads with a blue border will pop up. b) The user can drag the bead out of the semi-annulus and drop it onto a virtual glyph to assign its value to the corresponding visual channel of the virtual glyph. c) MARVisT automatically synchronizes the corresponding visual channel of the virtual glyph.

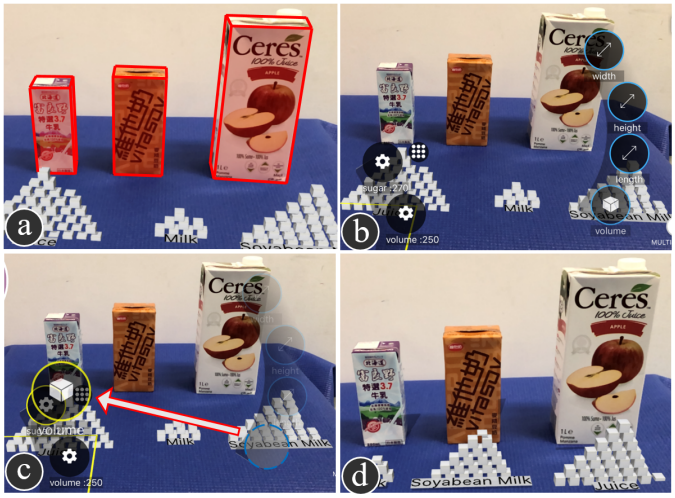

Fig. 8. a) Detect the three drinks and extract their size channels based on their bounding boxes. b) Open the semi-annulus panel to reveal the visual channels of a real object and the data attributes a virtual glyphs. c) Map the drinks to the sugar stacks based on the volume channel and the sugar content attribute: the greatest volume to the most sugar content, and so on. d) MARVisT automatically places each sugar stack in front of its corresponding drink.

APPENDIX B
PERFORMANCE TESTING

We conducted experiments to evaluate the performance of the current implementation of MARVisT. We evaluated the construction times, the frame rates of the static scenes, and the frame rates of the dynamic scenes for varying data sizes and glyph model complexities on an iPhone 8 plus (CPU with 4 processors @ 2.34GHz + 2 processors @ 1.7GHz, Apple A11 GPU, 3GB RAM) and an iPad Pro (CPU with 4 Vortex + 4 Tempest , Apple A12x GPU, 4GP RAM). The number of glyphs ranged from 10 to 1000 to 10000. We used two built-in primitive models, namely, cube and sphere, and the house and shoes models, which is the same model we used in Figure.5 in the paper, to represent simple and complex glyph models respectively (Figure 9). Each time we generated the glyphs and randomly distributed them within the field of view.

To measure the **construction time**, we imported the data and rendered the glyphs 11 times for each test case. We skipped the

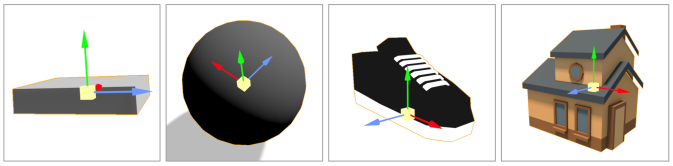

Fig. 9. Simple: Cube and Sphere. Complex: Shoe and House.

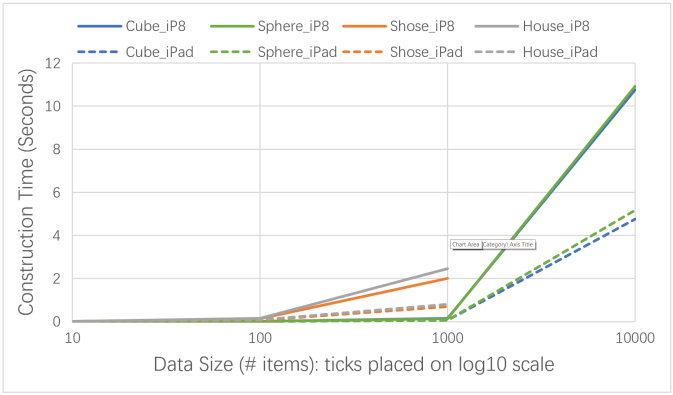

Fig. 10. The construction time of simple (cube and sphere) and complex (shoe and house) models running on an iPhone 8 Plus and an iPad Pro. The y-axis indicates the time for construction, and the x-axis indicates the data size. Given the memory limitations of the devices, we are not able to load 10, 000 complex models on an iPhone 8 Plus and an iPad Pro.

first time as a warm-up and report the average of the remaining 10 (in Figure 10). Even with 10, 000 models construction times remained below 12 seconds on both the devices. Given the memory limitations of the two devices (3GB on iPhone 8 Plus and 4GB on iPad Pro), we could not load and render 10, 000 complex models (36kb for each shoe and 60kb for each house).

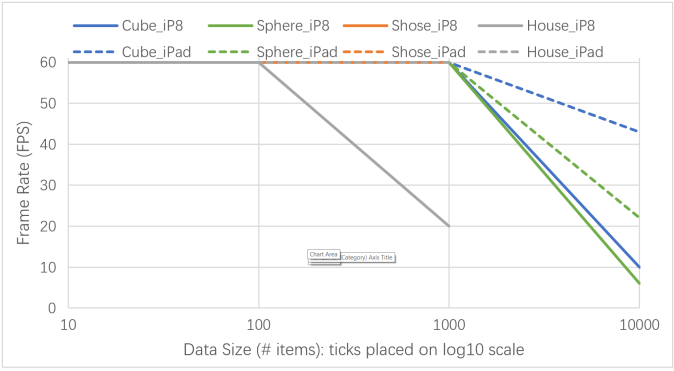

Fig. 11. The frame rates of static simple (cube and sphere) and complex (shoe and house) models running on an iPhone 8 Plus and an iPad Pro. The y-axis indicates the frame rates, and the x-axis indicates the data size. Given the memory limitations of the devices, we are not able to load 10, 000 complex models on an iPhone 8 Plus and an iPad Pro.

To measure the **frame rates of the static scenes**, we displayed all glyphs within the field of view for one minute. The maximum frame rates on the two devices were 60 FPS given the hardware restrictions. As expected, the frame rates dropped along with the increasing complexity of the scene (in Figure 11). The frame rate of the house model on iPhone 8 Plus dropped quickly because the mechanism of iOS[1] restricted the maximum frame rate to 30 FPS under heavy workloads.

1. https://developer.apple.com/documentation/scenekit/scnview/1621205-preferredframesperscond

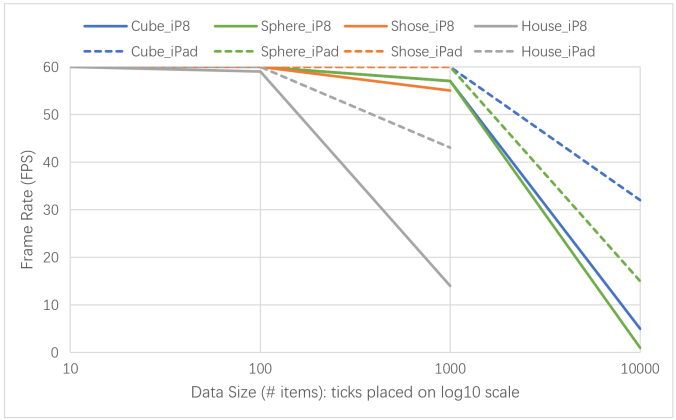

Fig. 12. The frame rates of dynamic simple (cube and sphere) and complex (shoe and house) models running on an iPhone 8 Plus and an iPad Pro. The y-axis indicates the frame rates, and the x-axis indicates the data size. Given the memory limitations of the devices, we are not able to load 10, 000 complex models on an iPhone 8 Plus and an iPad Pro.

To measure the **frame rates of the dynamic scenes**, we displayed all glyphs within the field of view and used the data-binding panel to increase their volume to twice the size. The maximum frame rates on the two devices were 60 FPS given the hardware restrictions. Compared with the frame rates of static scenes, the frame rates of the scene wherein the models are dynamically changed dropped a little bit as expected. Besides that, the frame rates dropped along with the increasing number of models in the scene (in Figure 12).

Overall, for datasets with reasonable size (1, 000 models or less), our implementation guarantee fast construction (in around 2 seconds) and real-time frame rates (over 50 FPS in most cases). Given the hardware and software limitations of the two iOS devices, we could not load 10, 000 data points and the frame rates drop quickly in heavy workloads. However, considering the usage scenario (i.e., the personal context) of MARVisT, wherein users usually do not have big datasets to visualize, we think the performance of the current implementation is acceptable. We believe there is still room for improvement of MARVisT in the future, such as optimizing the memory usage, utilizing level-of-detail techniques to improve the frame rate, and following the best practices of iOS applications to enhance the overall performance of MARVisT.

References

[1] Apple. ARKit: Recognizing Images in an AR Experience. https://developer.apple.com/documentation/arkit/recognizing_images_in_an_ar_experience. Accessed: 2018-02-21.

[2] Apple. ARKit: Scanning and Detecting 3D Objects. https://developer.apple.com/documentation/arkit/scanning_and_detecting_3d_objects. Accessed: 2018-02-21.

[3] Apple. ARKit: Using Vision in Real Time with ARKit. https://developer.apple.com/documentation/arkit/using_vision_in_real_time_with_arkit. Accessed: 2018-02-21.

[4] Apple. Vision: Detecting Objects in Still Images. https://developer.apple.com/documentation/vision/detecting_objects_in_still_images. Accessed: 2018-02-21.

[5] Apple. Vision: Recognizing Objects in Live Capture. https://developer.apple.com/documentation/vision/recognizing_objects_in_live_capture. Accessed: 2018-02-21.