DAFTAR PUSTAKA

Android Developers. (2017). *Application Fundamentals*. Retrieved from Android Developers: https://developer.android.com/guide/components/fundamentals

Azuma, R. T. (1997). A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments,* 355-385.

Bimber, O., & Raskar, R. (2005). *Spatial Augmented Reality: Merging Real and Virtual Worlds.* Natick, Massachusetts: A K Peters, Ltd.

Cushnan, D., & El Habbak, H. (2013). *Developing AR Games for iOS and Android.* Birmingham: Packt Publishing.

Google for Developers. (2022). *Fundamental concepts*. Retrieved from ARCore | Google for Developers: https://developers.google.com/ar/develop/fundamentals

Google for Developers. (2022). *Hit-tests place virtual objects in the real world*. Retrieved from ARCore | Google for Developers: https://developers.google.com/ar/develop/hit-test

Google for Developers. (2022). *Overview of ARCore and supported development environments*. Retrieved from ARCore | Google for Developers: https://developers.google.com/ar/develop

Google for Developers. (2022). *Working with Anchors*. Retrieved from ARCore | Google for Developers: https://developers.google.com/ar/develop/anchors

Google for Developers. (2023). *Depth adds realism*. Retrieved from ARCore | Google for Developers: https://developers.google.com/ar/develop/depth

Milgram, P., Takemura, H., Utsumi, A., & Kishino, F. (1994). Augmented reality: A class of displays on the reality-virtuality continuum. *Telemanipulator and Telepresence Technologies* (pp. 282-292). Boston: SPIE.

Open Handset Alliance. (2012). *Android Overview*. Retrieved from Open Handset Alliance: http://www.openhandsetalliance.com/android_overview.html

Silva, R., Oliveira, J., & Giraldi, G. (2003). Introduction to Augmented Reality. Brazil: National Laboratory for Scientific Computation.

Lampiran 1 Daftar kode program

1. Kode untuk pengujian oklusi objek

```
using System;
using System.Collections.Generic;
using Unity.Collections;
using UnityEngine.Serialization;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.Rendering;

namespace UnityEngine.XR.ARFoundation
{
    /// <summary>
    /// The manager for the occlusion subsystem.
    /// </summary>
    [DisallowMultipleComponent]
    [DefaultExecutionOrder(ARUpdateOrder.k_OcclusionManager)]
    [HelpURL(HelpUrls.ApiWithNamespace + nameof(AROcclusionManager) + ".html")]
    public sealed class AROcclusionManager :
#if UNITY_2020_2_OR_NEWER
        SubsystemLifecycleManager<XROcclusionSubsystem,
XROcclusionSubsystemDescriptor, XROcclusionSubsystem.Provider>
#else
        SubsystemLifecycleManager<XROcclusionSubsystem,
XROcclusionSubsystemDescriptor>
#endif
    {
        /// <summary>
        /// The list of occlusion texture infos.
        /// </summary>
        /// <value>
        /// The list of occlusion texture infos.
        /// </value>
        readonly List<ARTextureInfo> m_TextureInfos = new List<ARTextureInfo>();

        /// <summary>
        /// The list of occlusion textures.
        /// </summary>
        /// <value>
        /// The list of occlusion textures.
        /// </value>
        readonly List<Texture2D> m_Textures = new List<Texture2D>();

        /// <summary>
        /// The list of occlusion texture property IDs.
        /// </summary>
        /// <value>
        /// The list of occlusion texture property IDs.
        /// </value>
        readonly List<int> m_TexturePropertyIds = new List<int>();

        /// <summary>
        /// The human stencil texture info.
        /// </summary>
        /// <value>
        /// The human stencil texture info.
        /// </value>
        ARTextureInfo m_HumanStencilTextureInfo;
```

```
        /// <summary>
        /// The human depth texture info.
        /// </summary>
        /// <value>
        /// The human depth texture info.
        /// </value>
        ARTextureInfo m_HumanDepthTextureInfo;

        /// <summary>
        /// The environment depth texture info.
        /// </summary>
        /// <value>
        /// The environment depth texture info.
        /// </value>
        ARTextureInfo m_EnvironmentDepthTextureInfo;

        /// <summary>
        /// The environment depth confidence texture info.
        /// </summary>
        /// <value>
        /// The environment depth confidence texture info.
        /// </value>
        ARTextureInfo m_EnvironmentDepthConfidenceTextureInfo;

        /// <summary>
        /// An event which fires each time an occlusion camera frame is received.
        /// </summary>
        public event Action<AROcclusionFrameEventArgs> frameReceived;

        /// <summary>
        /// The mode for generating the human segmentation stencil texture.
        /// This method is obsolete.
        /// Use <see cref="requestedHumanStencilMode"/>
        /// or  <see cref="currentHumanStencilMode"/> instead.
        /// </summary>
        [Obsolete("Use            requestedSegmentationStencilMode            or
currentSegmentationStencilMode instead. (2020-01-14)")]
        public HumanSegmentationStencilMode humanSegmentationStencilMode
        {
            get => m_HumanSegmentationStencilMode;
            set => requestedHumanStencilMode = value;
        }

        /// <summary>
        /// The requested mode for generating the human segmentation stencil
texture.
        /// </summary>
        public HumanSegmentationStencilMode requestedHumanStencilMode
        {
            get        =>        subsystem?.requestedHumanStencilMode        ??
m_HumanSegmentationStencilMode;
            set
            {
                m_HumanSegmentationStencilMode = value;
                if (enabled && descriptor?.supportsHumanSegmentationStencilImage
== true)
                {
                    subsystem.requestedHumanStencilMode = value;
```

```
                }
            }
        }

        /// <summary>
        /// Get the current mode in use for generating the human segmentation
stencil mode.
        /// </summary>
        public    HumanSegmentationStencilMode    currentHumanStencilMode    =>
subsystem?.currentHumanStencilMode ?? HumanSegmentationStencilMode.Disabled;

        [SerializeField]
        [Tooltip("The   mode   for   generating   human   segmentation   stencil
texture.\n\n"
                 + "Disabled -- No human stencil texture produced.\n"
                 +  "Fastest  --  Minimal  rendering  quality.  Minimal  frame
computation.\n"
                 +  "Medium  --  Medium  rendering  quality.  Medium  frame
computation.\n"
                 +  "Best  --  Best  rendering  quality.  Increased  frame
computation.")]
        HumanSegmentationStencilMode      m_HumanSegmentationStencilMode      =
HumanSegmentationStencilMode.Disabled;

        /// <summary>
        /// The mode for generating the human segmentation depth texture.
        /// This method is obsolete.
        /// Use <see cref="requestedHumanDepthMode"/>
        /// or  <see cref="currentHumanDepthMode"/> instead.
        /// </summary>
        [Obsolete("Use             requestedSegmentationDepthMode             or
currentSegmentationDepthMode instead. (2020-01-15)")]
        public HumanSegmentationDepthMode humanSegmentationDepthMode
        {
            get => m_HumanSegmentationDepthMode;
            set => requestedHumanDepthMode = value;
        }

        /// <summary>
        /// Get or set the requested human segmentation depth mode.
        /// </summary>
        public HumanSegmentationDepthMode requestedHumanDepthMode
        {
            get         =>         subsystem?.requestedHumanDepthMode         ??
m_HumanSegmentationDepthMode;
            set
            {
                m_HumanSegmentationDepthMode = value;
                if  (enabled  &&  descriptor?.supportsHumanSegmentationDepthImage
== true)
                {
                    subsystem.requestedHumanDepthMode = value;
                }
            }
        }

        /// <summary>
        /// Get the current human segmentation depth mode in use by the subsystem.
        /// </summary>
```

```
        public      HumanSegmentationDepthMode      currentHumanDepthMode      =>
subsystem?.currentHumanDepthMode ?? HumanSegmentationDepthMode.Disabled;

        [SerializeField]
        [Tooltip("The mode for generating human segmentation depth texture.\n\n"
                 + "Disabled -- No human depth texture produced.\n"
                 +  "Fastest  --  Minimal  rendering  quality.  Minimal  frame
computation.\n"
                 +  "Best  --  Best  rendering  quality.  Increased  frame
computation.")]
        HumanSegmentationDepthMode         m_HumanSegmentationDepthMode         =
HumanSegmentationDepthMode.Disabled;

        /// <summary>
        /// Get or set the requested environment depth mode.
        /// </summary>
        public EnvironmentDepthMode requestedEnvironmentDepthMode
        {
            get       =>       subsystem?.requestedEnvironmentDepthMode       ??
m_EnvironmentDepthMode;
            set
            {
                m_EnvironmentDepthMode = value;
                if (enabled && descriptor?.supportsEnvironmentDepthImage == true)
                {
                    subsystem.requestedEnvironmentDepthMode = value;
                }
            }
        }

        /// <summary>
        /// Get the current environment depth mode in use by the subsystem.
        /// </summary>
        public      EnvironmentDepthMode      currentEnvironmentDepthMode      =>
subsystem?.currentEnvironmentDepthMode ?? EnvironmentDepthMode.Disabled;

        [SerializeField]
        [Tooltip("The mode for generating the environment depth texture.\n\n"
                 + "Disabled -- No environment depth texture produced.\n"
                 +  "Fastest  --  Minimal  rendering  quality.  Minimal  frame
computation.\n"
                 +  "Medium  --  Medium  rendering  quality.  Medium  frame
computation.\n"
                 +  "Best  --  Best  rendering  quality.  Increased  frame
computation.")]
        EnvironmentDepthMode               m_EnvironmentDepthMode               =
EnvironmentDepthMode.Fastest;

        /// <summary>
        /// Get or set the requested occlusion preference mode.
        /// </summary>
        public OcclusionPreferenceMode requestedOcclusionPreferenceMode
        {
            get      =>      subsystem?.requestedOcclusionPreferenceMode      ??
m_OcclusionPreferenceMode;
            set
            {
                m_OcclusionPreferenceMode = value;
                if (enabled && (subsystem != null))
```

```
                {
                    subsystem.requestedOcclusionPreferenceMode = value;
                }
            }
        }

        /// <summary>
        /// Get the current occlusion preference mode in use by the subsystem.
        /// </summary>
        public    OcclusionPreferenceMode    currentOcclusionPreferenceMode    =>
subsystem?.currentOcclusionPreferenceMode                                    ??
OcclusionPreferenceMode.PreferEnvironmentOcclusion;

        [SerializeField]
        [Tooltip("If both environment texture and human stencil & depth textures
are available, this mode specifies which should be used for occlusion.")]
        OcclusionPreferenceMode           m_OcclusionPreferenceMode           =
OcclusionPreferenceMode.PreferEnvironmentOcclusion;

        /// <summary>
        /// The human segmentation stencil texture.
        /// </summary>
        /// <value>
        ///  The  human  segmentation  stencil  texture,  if  any.  Otherwise,
<c>null</c>.
        /// </value>
        public Texture2D humanStencilTexture
        {
            get
            {
                if ((descriptor?.supportsHumanSegmentationStencilImage == true)
                    &&   subsystem.TryGetHumanStencil(out   XRTextureDescriptor
humanStencilDescriptor))
                {
                    m_HumanStencilTextureInfo                                =
ARTextureInfo.GetUpdatedTextureInfo(m_HumanStencilTextureInfo,

humanStencilDescriptor);

DebugAssert.That(((m_HumanStencilTextureInfo.descriptor.dimension            ==
TextureDimension.Tex2D)
                                  ||
(m_HumanStencilTextureInfo.descriptor.dimension == TextureDimension.None)))?.
                        WithMessage("Human Stencil Texture needs to be a Texture
2D, but instead is "
                                    +
$"{m_HumanStencilTextureInfo.descriptor.dimension.ToString()}.");
                    return m_HumanStencilTextureInfo.texture as Texture2D;
                }
                return null;
            }
        }

        /// <summary>
        /// Attempt to get the latest human stencil CPU image. This provides
directly access to the raw pixel data.
        /// </summary>
        /// <remarks>
        /// The `XRCpuImage` must be disposed to avoid resource leaks.
```

```
        /// </remarks>
        /// <param name="cpuImage">If this method returns `true`, an acquired
`XRCpuImage`.</param>
        /// <returns>Returns `true` if the CPU image was acquired. Returns `false`
otherwise.</returns>
        public bool TryAcquireHumanStencilCpuImage(out XRCpuImage cpuImage)
        {
            if (descriptor?.supportsHumanSegmentationStencilImage == true)
            {
                return subsystem.TryAcquireHumanStencilCpuImage(out cpuImage);
            }

            cpuImage = default;
            return false;
        }

        /// <summary>
        /// The human segmentation depth texture.
        /// </summary>
        /// <value>
        /// The human segmentation depth texture, if any. Otherwise, <c>null</c>.
        /// </value>
        public Texture2D humanDepthTexture
        {
            get
            {
                if ((descriptor?.supportsHumanSegmentationDepthImage == true)
                    &&    subsystem.TryGetHumanDepth(out    XRTextureDescriptor
humanDepthDescriptor))
                {
                    m_HumanDepthTextureInfo                                        =
ARTextureInfo.GetUpdatedTextureInfo(m_HumanDepthTextureInfo,

humanDepthDescriptor);

DebugAssert.That(m_HumanDepthTextureInfo.descriptor.dimension                 ==
TextureDimension.Tex2D
                                    ||
m_HumanDepthTextureInfo.descriptor.dimension == TextureDimension.None)?.
                        WithMessage("Human Depth Texture needs to be a Texture
2D, but instead is "
                                    +
$"{m_HumanDepthTextureInfo.descriptor.dimension.ToString()}.");
                    return m_HumanDepthTextureInfo.texture as Texture2D;
                }
                return null;
            }
        }

        /// <summary>
        /// Attempt to get the latest environment depth confidence CPU image.
This provides directly access to the
        ///     raw pixel data.
        /// </summary>
        /// <remarks>
        /// The `XRCpuImage` must be disposed to avoid resource leaks.
        /// </remarks>
        /// <param name="cpuImage">If this method returns `true`, an acquired
`XRCpuImage`.</param>
```

```
        /// <returns>Returns `true` if the CPU image was acquired. Returns `false`
otherwise.</returns>
        public bool TryAcquireEnvironmentDepthConfidenceCpuImage(out XRCpuImage
cpuImage)
        {
            if (descriptor?.supportsEnvironmentDepthConfidenceImage == true)
            {
                return
subsystem.TryAcquireEnvironmentDepthConfidenceCpuImage(out cpuImage);
            }

            cpuImage = default;
            return false;
        }

        /// <summary>
        /// The environment depth confidence texture.
        /// </summary>
        /// <value>
        ///  The environment depth confidence texture, if any. Otherwise,
<c>null</c>.
        /// </value>
        public Texture2D environmentDepthConfidenceTexture
        {
            get
            {
                if   ((descriptor?.supportsEnvironmentDepthConfidenceImage   ==
true)
                    &&            subsystem.TryGetEnvironmentDepthConfidence(out
XRTextureDescriptor environmentDepthConfidenceDescriptor))
                {
                    m_EnvironmentDepthConfidenceTextureInfo                    =
ARTextureInfo.GetUpdatedTextureInfo(m_EnvironmentDepthConfidenceTextureInfo,

environmentDepthConfidenceDescriptor);

DebugAssert.That(m_EnvironmentDepthConfidenceTextureInfo.descriptor.dimension ==
TextureDimension.Tex2D
                                    ||
m_EnvironmentDepthConfidenceTextureInfo.descriptor.dimension                   ==
TextureDimension.None)?.
                        WithMessage("Environment depth confidence texture needs
to be a Texture 2D, but instead is "
                                    +
$"{m_EnvironmentDepthConfidenceTextureInfo.descriptor.dimension.ToString()}.");
                    return  m_EnvironmentDepthConfidenceTextureInfo.texture  as
Texture2D;
                }
                return null;
            }
        }


        /// <summary>
        /// Attempt to get the latest human depth CPU image. This provides
directly access to the raw pixel data.
        /// </summary>
        /// <remarks>
        /// The `XRCpuImage` must be disposed to avoid resource leaks.
```

```
        /// </remarks>
        /// <param name="cpuImage">If this method returns `true`, an acquired
`XRCpuImage`.</param>
        /// <returns>Returns `true` if the CPU image was acquired. Returns `false`
otherwise.</returns>
        public bool TryAcquireHumanDepthCpuImage(out XRCpuImage cpuImage)
        {
            if (descriptor?.supportsHumanSegmentationDepthImage == true)
            {
                return subsystem.TryAcquireHumanDepthCpuImage(out cpuImage);
            }

            cpuImage = default;
            return false;
        }

        /// <summary>
        /// The environment depth texture.
        /// </summary>
        /// <value>
        /// The environment depth texture, if any. Otherwise, <c>null</c>.
        /// </value>
        public Texture2D environmentDepthTexture
        {
            get
            {
                if ((descriptor?.supportsEnvironmentDepthImage == true)
                    && subsystem.TryGetEnvironmentDepth(out XRTextureDescriptor
environmentDepthDescriptor))
                {
                    m_EnvironmentDepthTextureInfo                                =
ARTextureInfo.GetUpdatedTextureInfo(m_EnvironmentDepthTextureInfo,

environmentDepthDescriptor);

DebugAssert.That(m_EnvironmentDepthTextureInfo.descriptor.dimension           ==
TextureDimension.Tex2D
                                    ||
m_EnvironmentDepthTextureInfo.descriptor.dimension == TextureDimension.None)?.
                        WithMessage("Environment depth texture needs to be a
Texture 2D, but instead is "
                                    +
$"{m_EnvironmentDepthTextureInfo.descriptor.dimension.ToString()}.");
                    return m_EnvironmentDepthTextureInfo.texture as Texture2D;
                }
                return null;
            }
        }

        /// <summary>
        /// Attempt to get the latest environment depth CPU image. This provides
directly access to the raw pixel data.
        /// </summary>
        /// <remarks>
        /// The `XRCpuImage` must be disposed to avoid resource leaks.
        /// </remarks>
        /// <param name="cpuImage">If this method returns `true`, an acquired
`XRCpuImage`.</param>
```

```
        /// <returns>Returns `true` if the CPU image was acquired. Returns `false`
otherwise.</returns>
        public bool TryAcquireEnvironmentDepthCpuImage(out XRCpuImage cpuImage)
        {
            if (descriptor?.supportsEnvironmentDepthImage == true)
            {
                return          subsystem.TryAcquireEnvironmentDepthCpuImage(out
cpuImage);
            }

            cpuImage = default;
            return false;
        }

        /// <summary>
        /// Callback before the subsystem is started (but after it is created).
        /// </summary>
        protected override void OnBeforeStart()
        {
            if (descriptor.supportsHumanSegmentationStencilImage)
            {
                subsystem.requestedHumanStencilMode                              =
m_HumanSegmentationStencilMode;
            }

            if (descriptor.supportsHumanSegmentationDepthImage)
            {
                subsystem.requestedHumanDepthMode                                =
m_HumanSegmentationDepthMode;
            }

            subsystem.requestedEnvironmentDepthMode = m_EnvironmentDepthMode;
            subsystem.requestedOcclusionPreferenceMode                           =
m_OcclusionPreferenceMode;

            ResetTextureInfos();
        }

        /// <summary>
        /// Callback when the manager is being disabled.
        /// </summary>
        protected override void OnDisable()
        {
            base.OnDisable();

            ResetTextureInfos();
            InvokeFrameReceived();
        }

        /// <summary>
        /// Callback as the manager is being updated.
        /// </summary>
        public void Update()
        {
            if (subsystem != null)
            {
                UpdateTexturesInfos();
                InvokeFrameReceived();
            }
```

```
        }

        void ResetTextureInfos()
        {
            m_HumanStencilTextureInfo.Reset();
            m_HumanDepthTextureInfo.Reset();
            m_EnvironmentDepthTextureInfo.Reset();
            m_EnvironmentDepthConfidenceTextureInfo.Reset();
        }

        /// <summary>
        /// Pull the texture descriptors from the occlusion subsystem, and update
the texture information maintained by
        /// this component.
        /// </summary>
        void UpdateTexturesInfos()
        {
            var                        textureDescriptors                        =
subsystem.GetTextureDescriptors(Allocator.Temp);
            try
            {
                int      numUpdated      =      Math.Min(m_TextureInfos.Count,
textureDescriptors.Length);

                // Update the existing textures that are in common between the
two arrays.
                for (int i = 0; i < numUpdated; ++i)
                {
                    m_TextureInfos[i]                                            =
ARTextureInfo.GetUpdatedTextureInfo(m_TextureInfos[i], textureDescriptors[i]);
                }

                // If there are fewer textures in the current frame than we had
previously, destroy any remaining unneeded
                // textures.
                if (numUpdated < m_TextureInfos.Count)
                {
                    for (int i = numUpdated; i < m_TextureInfos.Count; ++i)
                    {
                        m_TextureInfos[i].Reset();
                    }
                    m_TextureInfos.RemoveRange(numUpdated, (m_TextureInfos.Count
- numUpdated));
                }
                // Else, if there are more textures in the current frame than we
have previously, add new textures for any
                // additional descriptors.
                else if (textureDescriptors.Length > m_TextureInfos.Count)
                {
                    for (int i = numUpdated; i < textureDescriptors.Length; ++i)
                    {
                        m_TextureInfos.Add(new
ARTextureInfo(textureDescriptors[i]));
                    }
                }
            }
            finally
            {
                if (textureDescriptors.IsCreated)
```

```
                {
                    textureDescriptors.Dispose();
                }
            }
        }

        /// <summary>
        /// Invoke the occlusion frame received event with the updated textures
and texture property IDs.
        /// </summary>
        void InvokeFrameReceived()
        {
            if (frameReceived != null)
            {
                int numTextureInfos = m_TextureInfos.Count;

                m_Textures.Clear();
                m_TexturePropertyIds.Clear();

                m_Textures.Capacity = numTextureInfos;
                m_TexturePropertyIds.Capacity = numTextureInfos;

                for (int i = 0; i < numTextureInfos; ++i)
                {
                    DebugAssert.That(m_TextureInfos[i].descriptor.dimension   ==
TextureDimension.Tex2D)?.
                        WithMessage($"Texture needs to be a Texture 2D, but
instead is {m_TextureInfos[i].descriptor.dimension.ToString()}.");

                    m_Textures.Add((Texture2D)m_TextureInfos[i].texture);

m_TexturePropertyIds.Add(m_TextureInfos[i].descriptor.propertyNameId);
                }

                subsystem.GetMaterialKeywords(out                    List<string>
enabledMaterialKeywords, out List<string>disabledMaterialKeywords);

                AROcclusionFrameEventArgs           args           =           new
AROcclusionFrameEventArgs();
                args.textures = m_Textures;
                args.propertyNameIds = m_TexturePropertyIds;
                args.enabledMaterialKeywords = enabledMaterialKeywords;
                args.disabledMaterialKeywords = disabledMaterialKeywords;

                frameReceived(args);
            }
        }
    }
}
```

2. Kode untuk deteksi bidang dan *raycast*

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.XR.ARFoundation;

namespace UnityEngine.XR.ARFoundation.Samples
```

```
{
    /// <summary>
    /// This example demonstrates how to toggle plane detection,
    /// and also hide or show the existing planes.
    /// </summary>
    [RequireComponent(typeof(ARPlaneManager))]
    public class PlaneDetectionController : MonoBehaviour
    {
        [Tooltip("The UI Text element used to display plane detection messages.")]
        [SerializeField]
        Text m_TogglePlaneDetectionText;

        /// <summary>
        /// The UI Text element used to display plane detection messages.
        /// </summary>
        public Text togglePlaneDetectionText
        {
            get { return m_TogglePlaneDetectionText; }
            set { m_TogglePlaneDetectionText = value; }
        }

        /// <summary>
        /// Toggles plane detection and the visualization of the planes.
        /// </summary>
        public void TogglePlaneDetection()
        {
            m_ARPlaneManager.enabled = !m_ARPlaneManager.enabled;

            string planeDetectionMessage = "";
            if (m_ARPlaneManager.enabled)
            {
                planeDetectionMessage = "Disable Plane";
                SetAllPlanesActive(true);
            }
            else
            {
                planeDetectionMessage = "Enable Plane";
                SetAllPlanesActive(false);
            }

            if (togglePlaneDetectionText != null)
                togglePlaneDetectionText.text = planeDetectionMessage;
        }

        /// <summary>
        /// Iterates over all the existing planes and activates
        /// or deactivates their <c>GameObject</c>s'.
        /// </summary>
        /// <param name="value">Each planes' GameObject is SetActive with this
value.</param>
        void SetAllPlanesActive(bool value)
        {
            foreach (var plane in m_ARPlaneManager.trackables)
                plane.gameObject.SetActive(value);
        }

        void Awake()
        {
            m_ARPlaneManager = GetComponent<ARPlaneManager>();
```

```
        }

        ARPlaneManager m_ARPlaneManager;
    }
}
```

```
/*
 * Copyright 2021 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

using System.Collections;
using System.Collections.Generic;
using System.Linq;

using Unity.Collections;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;


public class ReticleBehaviour : MonoBehaviour
{
    public GameObject Child;
    public DrivingSurfaceManager DrivingSurfaceManager;
    public ARPlane CurrentPlane;



    // Start is called before the first frame update
    private void Start()
    {
        Child = transform.GetChild(0).gameObject;
    }

    private void Update()
    {
        CurrentPlane = null;

        // Perform raycast using the center of the viewport.
        var screenCenter = Camera.main.ViewportToScreenPoint(new Vector3(0.5f,
0.5f));
        var hits = new List<ARRaycastHit>();
        DrivingSurfaceManager.RaycastManager.Raycast(screenCenter,         hits,
TrackableType.PlaneWithinBounds);

        ARRaycastHit? hit = null;
```

```
        if (hits.Count > 0)
        {
            // If you don't have a locked plane already...
            var lockedPlane = DrivingSurfaceManager.LockedPlane;
            hit = lockedPlane == null
                // ... use the first hit in `hits`.
                ? hits[0]
                // Otherwise use the locked plane, if it's there.
                :      hits.SingleOrDefault(x      =>      x.trackableId      ==
lockedPlane.trackableId);
        }

        if (hit.HasValue)
        {
            CurrentPlane                                                        =
DrivingSurfaceManager.PlaneManager.GetPlane(hit.Value.trackableId);
            // Move this reticle to the location of the hit.
            transform.position = hit.Value.pose.position;
        }

        Child.SetActive(CurrentPlane != null);
    }
}
```

3. Kode untuk pemilihan warna cat dinding

```
/*
 * Copyright 2021 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

using System.Collections;

using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.UI;


/**
 * Spawns a <see cref="WallBehaviour"/> when a plane is tapped.
 */
public class WallManager : MonoBehaviour
{
    public GameObject WallPrefab;
    public ReticleBehaviour Reticle;
    public DrivingSurfaceManager DrivingSurfaceManager;
```

```
    public WallBehaviour Wall;

    public Text debugtext;

    private int mark=0;
    Vector3 pos1,pos2;

    private GameObject obj;

    private                    Color[]                    warna                    =
{Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta,Color.w
hite,Color.black};

    private void Update()
    {

        if (Wall == null && WasTapped() && Reticle.CurrentPlane != null)
        {
            print("layar di sentuh");

            mark++;
            if(mark==1){
                pos1 = Reticle.transform.position;
                print("pos1:"+pos1);
                debugtext.text += "pos1:"+pos1+" \r\n";

            } else if(mark == 2){
                pos2 = Reticle.transform.position;
                print("pos2:"+pos2);
                 debugtext.text +="pos2:"+pos2+" \r\n";


                Vector3 position = (pos1 + pos2) / 2f;
                //set y position to 0
                position.y = -0.2f;

                Vector3 scale = new Vector3(
                    Mathf.Abs(pos2.x - pos1.x), // set X scale to a small value
                    -2f,//Mathf.Abs(pos2.y - pos1.y), // set Y scale to a large
value
                    Mathf.Abs(pos2.z - pos1.z) // set Z scale to a small value
                );

                print("scale:"+scale);
                print("position:"+position);
                debugtext.text +="scale:"+scale+" \r\n";

                obj      =      GameObject.Instantiate(WallPrefab,      position,
Quaternion.identity);     //parameter     ketiga     bisa     diganti     pake
Reticle.transform.rotation
                print("insiate wall prefab");
                Wall = obj.GetComponent<WallBehaviour>();
                Wall.Reticle = Reticle;
                // Wall.transform.position = Reticle.transform.position;
                Wall.transform.position = position;// - scale/2;
                //scale
                Wall.transform.localScale = scale;
                print("position:"+Wall.transform.position);
                // Wall.transform.rotation = Reticle.transform.rotation;
```

```
                print("recticle rotation:"+Reticle.transform.position);

                DrivingSurfaceManager.LockPlane(Reticle.CurrentPlane);
            } else {
                mark=0;
                print("mark reset:"+mark);
                debugtext.text = "mark reset";
                RemoveWall();

            }
            // Spawn our car at the reticle location.

        }
    }

    public void RemoveWall()
    {
        if (obj != null)
        {
            Destroy(obj);
            obj = null;
            debugtext.text += "wall removed";
        }
    }

    public void SetWallheight(float yScale)
    {
        if (obj != null)
        {
            Vector3 scaleChange = new Vector3(0, yScale, 0);
            obj.transform.localScale += scaleChange;
        }
    }

    public void setWallWidth(float xScale)
    {
        if (obj != null)
        {
            // Vector3 scale = obj.transform.localScale;
            Vector3 scaleChange = new Vector3(xScale, 0, 0);
            obj.transform.localScale += scaleChange;
        }
    }

    public void rotateWall(float angle)
    {
        if (obj != null)
        {
            obj.transform.Rotate(0,angle,0);
        }
    }

    public void MoveWallToward(float distance)
    {
        if (obj != null)
        {
            obj.transform.Translate(0,0,distance);
        }
    }
```

```
public void MoveWallLeftRight(float distance)
{
    if (obj != null)
    {
        obj.transform.Translate(distance,0,0);
    }
}

public void SetYScale(float yScale)
{
    if (obj != null)
    {
        Vector3 scale = obj.transform.localScale;
        scale.y = scale.y + yScale;
        obj.transform.localScale = scale;
    }
}

public void SetXScale(float xScale)
{
    if (obj != null)
    {
        Vector3 scale = obj.transform.localScale;
        scale.x = scale.x + xScale;
        obj.transform.localScale = scale;
    }
}

public void setWallColor(int colorId)
{
    if (obj != null)
    {
        obj.GetComponent<Renderer>().material.color = warna[colorId];
    }
}

public void setWallColorRed()
{
    if (obj != null)
    {
        obj.GetComponent<Renderer>().material.color = Color.red;
    }
}

public void setWallColorBlue()
{
    if (obj != null)
    {
        obj.GetComponent<Renderer>().material.color = Color.green;
    }
}

public void SpawnWall(ARPlane plane)
{
    // var packageClone = GameObject.Instantiate(PackagePrefab);
    // packageClone.transform.position = FindRandomLocation(plane);

    // Package = packageClone.GetComponent<PackageBehaviour>();
```

```
    }

    private bool WasTapped()
    {
        if (Input.GetMouseButtonDown(0))
        {
            return true;
        }

        if (Input.touchCount == 0)
        {
            return false;
        }

        var touch = Input.GetTouch(0);
        if (touch.phase != TouchPhase.Began)
        {
            return false;
        }

        return true;
    }
}
```