DepthLab: Real-time 3D Interaction with Depth Maps for Mobile Augmented Reality

Ruofei Du, Eric Turner, Maksym Dzitsiuk, Luca Prasso, Ivo Duarte, Jason Dourgarian, Joao Afonso, Jose Pascoal, Josh Gladstone, Nuno Cruces, Shahram Izadi, Adarsh Kowdle, Konstantine Tsotsos, David Kim[†]
Google LLC

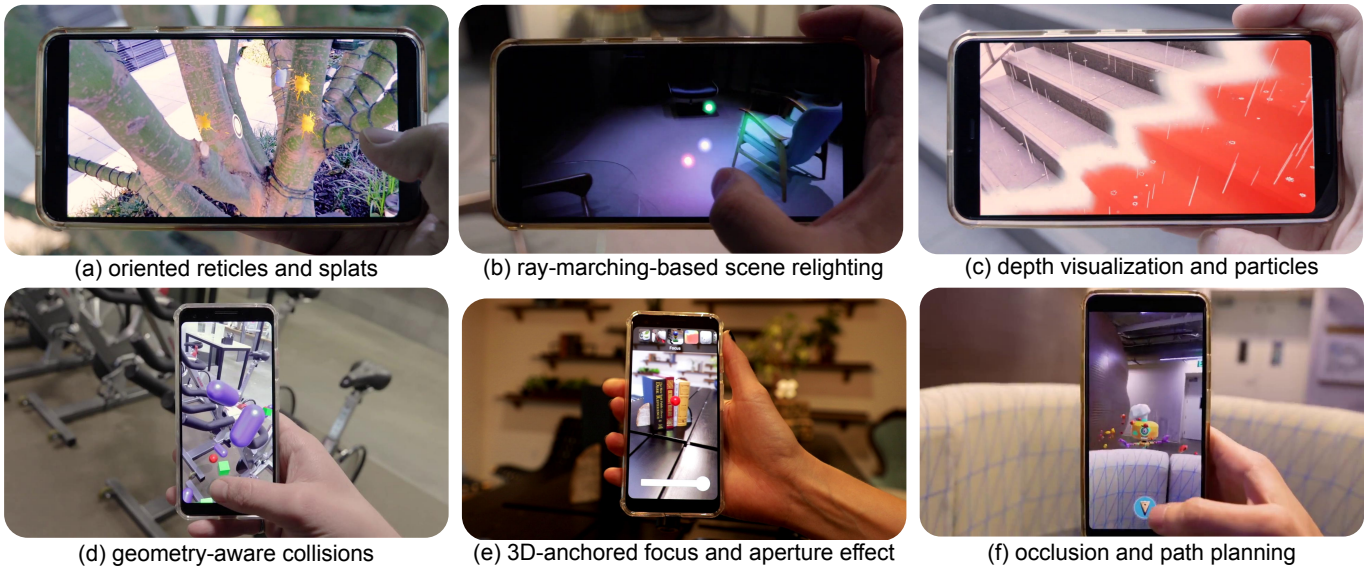


Figure 1. Real-time interactive components enabled by DepthLab: (a) virtual texture decals "splatting" onto physical trees and a white oriented reticle as a 3D virtual cursor; (b) relighting of a physical scene with three virtual point lights; (c) AR rain effect on dry stairs on the left and false-color depth map on the right; (d) virtual objects colliding with physical exercise equipment; (e) "Bokeh"-like effect putting focus on a physical 3D anchor; (f) occlusion and path planning in a mobile AR game. Please refer to the accompanying video captured in real time for more results.

ABSTRACT

Mobile devices with passive depth sensing capabilities are ubiquitous, and recently active depth sensors have become available on some tablets and AR/VR devices. Although real-time depth data is accessible, its rich value to mainstream AR applications has been sorely under-explored. Adoption of depth-based UX has been impeded by the complexity of performing even simple operations with raw depth data, such as detecting intersections or constructing meshes. In this paper, we introduce DepthLab, a software library that encapsulates a variety of depth-based UI/UX paradigms, including geometry-aware rendering (occlusion, shadows), surface interaction behaviors (physics-based collisions, avatar path planning), and visual effects (relighting, 3D-anchored focus and aperture effects). We break down the usage of depth into localized depth, surface depth, and dense depth, and describe our real-time algorithms for interaction and rendering tasks. We present the design process, system, and components of DepthLab to streamline and centralize the development of interactive depth features. We have open-sourced our software at https://github.com/googlesamples/arcore-depth-lab to external developers, conducted performance evaluation, and discussed how DepthLab can accelerate the workflow of mobile AR designers and developers. With DepthLab we aim to help mobile developers to effortlessly integrate depth into their AR experiences and amplify the expression of their creative vision.

[†]Corresponding author: kidavid@google.com.

ISBN 978-1-4503-6708-0/20/04…$15.00
DOI: https://doi.org/10.1145/3313831.XXXXXXX

Author Keywords

Depth map; interactive 3D graphics; real time; interaction; augmented reality; mobile AR; rendering; GPU; ARCore.

CCS Concepts

•**Human-centered computing** → **Mixed / augmented reality;** *User interface toolkits;*

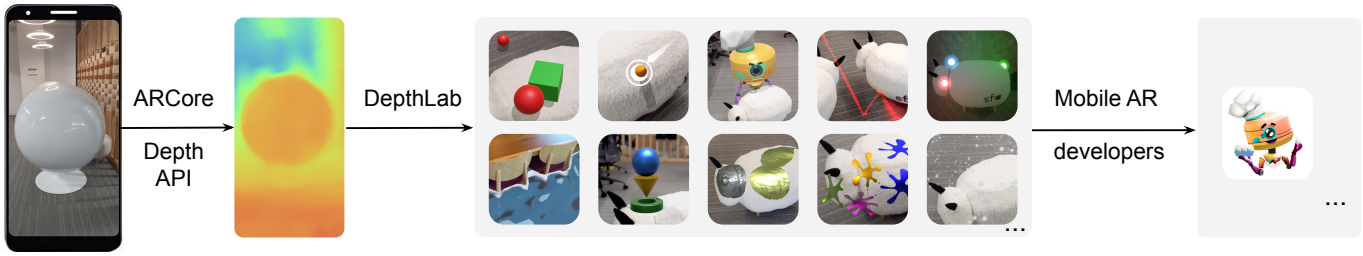


Figure 2. A high-level overview of DepthLab. We process the raw depth map from ARCore Depth API and provide customizable and self-contained components such as a 3D cursor, geometry-aware collision, and screen-space relighting. The DepthLab library aims to accelerate mobile app developers to build more photo-realistic and interactive AR applications.

INTRODUCTION

Augmented Reality (AR) has gained mainstream popularity, as evidenced by Pokemon Go[1], Snapchat[2], and IKEA Place[3] mobile AR experiences, among others. AR features have become a commodity via wide platform-level support with Google's ARCore[4] and Apple's ARKit[5]. These features empower applications to place virtual objects anchored on flat surfaces of the physical space or to invoke an experience as a reaction to detected AR markers. More advanced features are demonstrated on dedicated wearable AR devices, such as Microsoft HoloLens[6] and Magic Leap[7], which include active depth sensor hardware. Experiences on these devices use the output of continuous environmental surface reconstruction to enable geometry-aware object occlusions, shadow mapping, and physics simulations. Our goal is to bring these advanced features to mobile AR experiences without relying on dedicated sensors or the need for computationally expensive surface reconstruction.

Recent advances in mobile computer vision, demonstrated by Valentin *et al.*[46], enable hundreds of millions of compatible Android devices running the ARCore Depth API[8] to estimate depth maps from a single moving camera in real time. However, these depth maps have to be further processed to be useful for rendering and interaction purposes in the application layer. There is a large gap between this raw data and the typical expertise of mobile application developers who are not experienced in handling depth data (*e.g.*, for collisions, occlusion, shadows, and relighting).

To bridge this gap, we assembled and analyzed an exhaustive list of 39 geometry-aware AR features, and found that by applying alternative representations of depth data and a simple depth development template, we could enable over 60% of these features on mobile devices. More importantly, we could do so through efficient and easy-to-use high-level interfaces.

Our contributions are summarized as follows:

- Analysis of geometry-aware AR features and their required environmental representations.
- A depth development template leveraging three different data representations *without* surface reconstruction.
- Real-time techniques for enabling geometry-aware shadows, relighting, physics, and aperture effects in AR on general smartphones, even with a single RGB camera.
- Open-sourced code library[9] enabling AR developers with depth-based capabilities with modular geometry-aware AR features.

We believe our contributions will inspire the next generation of AR applications, where scene-aware interactions, enabled by accurate 3D information, are the key to seamless blending of the virtual and the real world.

RELATED WORK

Our work is built upon existing mobile AR capabilities and is inspired by prior art in mobile AR interactions, depth-based libraries, use cases, and algorithms for head-mounted AR displays.

Mobile AR Capabilities

Since the debut of the seminal ceiling-mounted AR system in 1968 [42], AR has gradually diverged into head-mounted displays and mobile devices. As portable computers in the backpack and wearable AR displays emerge in 1990s [11, 45, 19], a line of research further investigated outdoor navigation [14, 21], urban planning [36], tourism [5], social media [7, 8], medical surgery [3], and AR games [44, 17, 50] in mobile AR settings.

However, rendering and interaction capabilities in mobile devices are traditionally limited by tracking feature points [2], patterns [25], or detecting planes from the camera images [12]. Consequently, virtual objects may suffer from the "anchor drift" problem [48] and the detected virtual planes may go beyond the boundaries of the physical surfaces [31].

Motivated by these existing issues, DepthLab pushes the boundary of physical-virtual rendering and interaction by offering interactive modules including ray casting, per-pixel

[1]Pokemon Go: https://www.pokemongo.com
[2]Snapchat: https://www.snapchat.com
[3]IKEA Place: https://www.ikea.com/us/en/customer-service/mobile-apps
[4]ARCore: https://developers.google.com/ar
[5]ARKit: https://developer.apple.com/documentation/arkit
[6]Microsoft HoloLens: https://microsoft.com/hololens
[7]Magic Leap: https://www.magicleap.com
[8]ARCore Depth API: https://developers.googleblog.com/2020/06/a-new-wave-of-ar-realism-with-arcore-depth-api.html
[9]ARCore Depth Lab - Depth API Samples for Unity: https://github.com/googlesamples/arcore-depth-lab

occlusion test, collision detection, and more. Our work focuses on the interactive rendering space with passive depth maps computed from a single RGB camera [46] in real time.

Recently, Xuan *et al.*[30] presented a novel offline deep-learning pipeline to estimate depth maps from a single video and render video-based visual effects. In contrast, our work focuses on real-time algorithms and performance on a mobile phone.

Depth-based Libraries and Use Cases

In recent years, commercial products such as Microsoft Kinect[10], Leap Motion[11], and Project Tango have enabled active depth sensing and boosted a line of research in 3D reconstruction [22, 32, 33], semantic segmentation [47], body tracking [39, 57], indoor positioning [28], activity recognition [1, 52], collaborative work [40], hand tracking [26, 38, 43], touch detection [51, 53, 54], mixed-reality rendering [16, 55], and gesture recognition [41, 27, 35].

Interactive systems, such as HoloDesk utilize depth [18] to enhance AR experiences with direct 3D interaction on a desk and Illumiroom [24] demonstrates room-scale AR effects, such as snow and bouncing physics objects with depth and projection. Along this line of research, RoomAlive Toolkit for Unity[23] enables developers to leverage real-time depth sensing capabilities with multiple Kinect sensors in projection mapping experiences. Mixed Reality Toolkit[12] provides a set of components and features that leverage active depth sensors and semantic understanding of the environment, including spatial mesh reconstruction and hand tracking.

Our work differentiates from the prior art in scope. We explore a different interaction modality on a piece of widely-available commodity hardware (Android phones with a single color camera). We demonstrate a general development pattern enabling the direct use of depth data to merge the real and virtual environments. We further demonstrate concrete implementations of popular features, such as relighting, 3D-anchored aperture effect, and environmental re-texturing and offer open-source modules for designers and developers to use.

AR for Head-Mounted Displays

Today, commercial AR head-mounted displays (HMDs) such as HoloLens and MagicLeap use dedicated depth sensors to track hands and to continuously reconstruct real-world surfaces. However, these systems take time to scan the environment and to create a mesh reconstruction before interaction and rendering can happen.

In contrast, our system does not depend on dedicated depth sensors and can instantly run using the input depth maps. With live depth maps provided by ARCore Depth API [46], we are the first to demonstrate a number of geometry-aware AR interaction and visual effect features on smartphones without surface reconstruction to the best of our knowledge.

[10]Kinect: https://en.wikipedia.org/wiki/Kinect

[11]LeapMotion: https://leapmotion.com

[12]Mixed Reality Toolkit: https://github.com/microsoft/MixedRealityToolkit-Unity

SYSTEM SCOPE AND OVERVIEW

We propose a depth development template, which includes three different scene representations and ways to access depth data that enable a broad range of commonly used features in a 3D scene. We restrict our scope to features that can run immediately on a variety of devices by focusing on real-time depth map processing, rather than techniques requiring a persistent model of the environment generated with 3D surface reconstruction. Next, we describe our design process and system architecture.

Geometry-Aware AR Features Elicitation

We conducted a sequence of three brainstorming sessions with a total of 18 participants including researchers, engineers, and UX designers who have worked on AR or VR-related projects to elicit a wide range of geometry-aware AR features. We outline our brainstorming framework and summarize our ideas to inspire future researchers to build upon our approach.

The first brainstorming session focused on collecting all depth-map related ideas in one hour. We separated all participants in two groups (one focusing more on using passive depth data, and the other focusing more on future use cases with persistent or dynamic voxels) to add more structure. The main ideas can be grouped into the following three categories: geometry-aware rendering and actions; depth interaction interfaces and gestures; and visual effects of static and dynamic scenes (Figure 3).

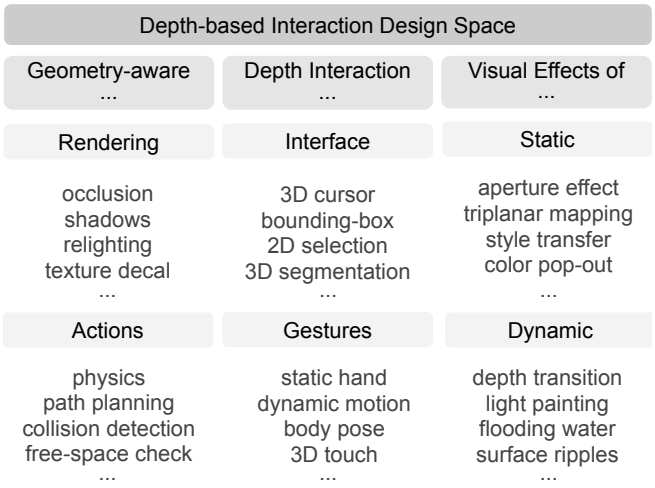


Figure 3. Classification of high-level depth component ideas from the brainstorm sessions. Please refer to the supplementary material for more items.

Each participant generated individual ideas in a 30-minute session. These ideas were then collected in a shared document and briefly presented to the group by each participant. We collected a total of 120 ideas this way. After this, the session organizer clustered similar ideas, initiated an offline voting and then summarized the ideas with the most votes.

In the second session, we assigned key attributes, discussed and ranked the technical feasibility, compelling use cases, relevance of depth, whether any form of machine learning (ML) models are required, and type of depth data as a minimum

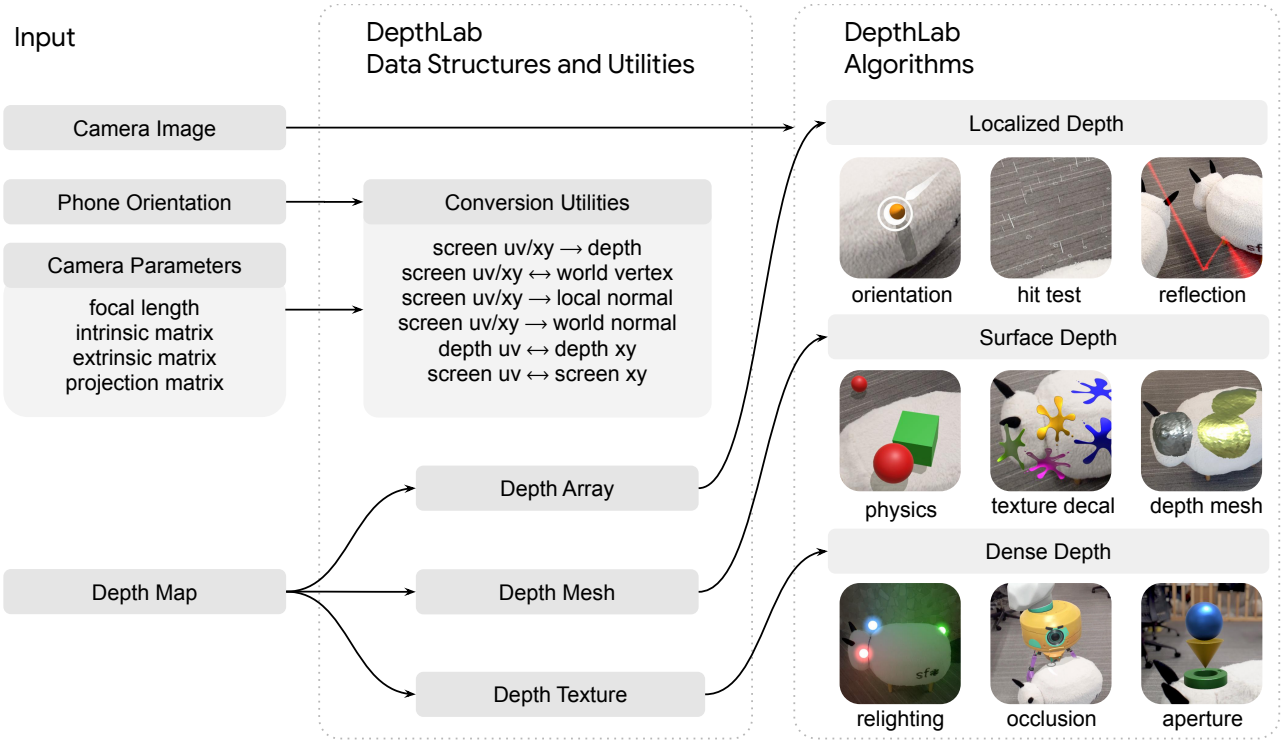


Figure 4. System architecture of DepthLab. Our input consists of the RGB camera image, depth map from ARCore Depth API, camera parameters, and phone orientation. For each frame, we update the depth array (CPU), depth mesh (CPU+GPU), and depth texture (GPU) from the raw depth buffer. We offer a set of conversion utilities to improve the workflow of developers and a set of algorithms that can be drag & dropped into their applications.

requirement for each idea. Based on the depth requirement, we scoped DepthLab to cover localized depth, surface depth, and dense depth, rather than surface reconstruction with voxels or triangles. We further explain this categorization in Table 1.

In the final session, we discussed the top priorities based on the overall rating of the ideas, organized weekly meetings, and assigned tasks to collaboratively develop DepthLab in a six-month period. We summarize 39 aggregated ideas in the supplementary material and indicate which ones DepthLab implements without 3D surface reconstruction.

System Architecture

DepthLab consists of four main components (Figure 4): tracking and input, data structure generation, conversion utilities, and algorithms for the presented effects.

Tracking and Input

DepthLab uses real-time depth maps provided by ARCore Depth API, which only requires a single moving RGB camera on the phone to estimate depth. A dedicated depth camera, such as time-of-flight (ToF) cameras can instantly provide depth maps without any initializing camera motion. Additionally, DepthLab uses the live camera feed, phone position and orientation, and camera parameters including focal length, intrinsic matrix, extrinsic matrix, and projection matrix for each frame to establish a mapping between the physical world and virtual objects. We provide extensive conversion utilities and interaction modules to facilitate higher-level mobile AR development.

Data Structures of DepthLab

The depth data is typically stored in a low-resolution depth buffer (160×120 in our examples[13]), which is a perspective camera image that contains a depth value instead of color in each pixel. For different purposes, we generate three categories of data structures:

1. **Depth array** stores depth in a 2D array of a landscape image with 16-bit integers on the CPU. With phone orientation and maximum sensing range (8 meters in our case), we offer conversion functions to access depth from any screen point or texture coordinates of the camera image.

2. **Depth mesh** is a real-time triangulated mesh generated for each depth map on both CPU and GPU. In contrast to traditional surface reconstruction with persistent voxels or triangles, depth mesh has little memory and compute overhead and can be generated in real time. We detail its generation procedure in Algorithm 2.

3. **Depth texture** is copied to the GPU from the depth array for per-pixel depth use cases in each frame. We filter the depth texture with depth-guided anti-aliasing methods (Figure 11) in additional to hardware-accelerated bilinear filtering to reduce visual artifacts.

Conversion Utilities and Algorithm

The slow adoption of depth on mobile device applications may lie in the complexity to process depth for end-user experiences.

[13]The depth map resolution may be different depending on different phone models.

Depth data becomes more useful when it is mapped to the camera image and the real-world geometry. However, even these steps require technical knowledge outside the domain of many application developers. Additional factors that can complicate depth processing include adapting to the change of the phone orientation, conversion of points between local and global coordinate frames, and the lack of examples. Our conversion utilities are detailed in the next section. Based on the three classes of depth data structures, we provide a series of algorithms and techniques for developers to directly apply high-level concepts such as physics, shadows, texture mapping, relighting in their applications using popular game editors, such as Unity or Unreal. We detail these techniques in the next section.

ALGORITHMS AND IMPLEMENTATION

DepthLab enables users to interact with a seamless blend of the physical environment and virtual renderings. To achieve this, we architect and implement a set of real-time algorithms and reusable components for mobile AR developers. Based on the data structures, we classify our DepthLab components into three categories: localized depth, surface depth, and dense depth. We provide an overview of their key traits in Table 1 and explain each term as follows:

	Localized Depth	Surface Depth	Dense Depth
CPU	✓	✓	✗ (non-real-time)
GPU	N/A	✓ (compute shader)	✓ (fragment shader)
Prerequisite	point projection normal estimation	depth mesh triplanar mapping	anti-aliasing multi-pass rendering
Data Structure	depth array	depth mesh	depth texture
Example Use Cases	physical measure oriented 3D cursor path planning	collision & physics virtual shadows texture decals	scene relighting aperture effects occluded objects

Table 1. Comparison of CPU/GPU real-time performance, key prerequisite techniques, underlying data structures, and example use cases between localized depth, surface depth, dense depth.

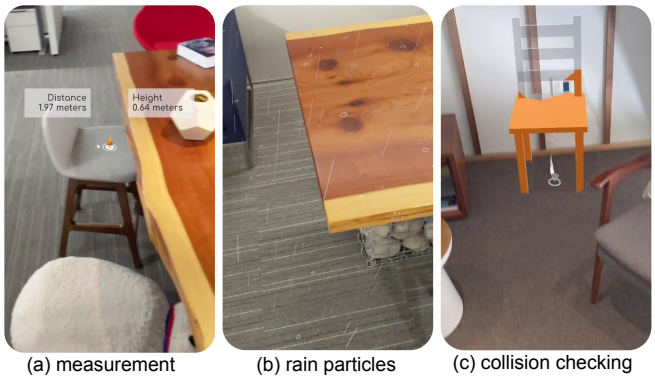
1. **Localized depth** uses the depth array to operate on a small number of points directly on the CPU. It is useful for computing physical measurements, estimating normal vectors, and automatically navigating virtual avatars for AR games.

2. **Surface depth** leverages the CPU or compute shaders on the GPU to create and update depth meshes in real time, thus enabling collision, physics, texture decal, geometry-aware shadows, etc.

3. **Dense depth** is copied to a GPU texture and is used for rendering depth-aware effects with GPU-accelerated bilinear filtering in screen space. Every pixel in the color camera image has a depth value mapped to it, which is useful for real-time computational photography tasks, such as relighting, 3D-anchored focus and aperture, and screen-space occlusion effects.

Interaction With Localized Depth

In comparison to DepthLab, widely-used AR frameworks, such as ARCore, ARKit, and AR Toolkit [25] provide hit testing functions that allow applications to get a real-world point based on the intersection between a camera ray and detected AR planes. Unfortunately, this method often yields errors due to inaccurate measurements around edges and non-planar objects on surfaces [31]. In this section, we introduce fundamental techniques and interactive use cases with localized depth (Figure 5), which yield more accurate hit tests and enable finer-grained use cases than plane-based or anchor-based AR interaction.



(a) measurement (b) rain particles (c) collision checking

Figure 5. Example use cases of localized depth. (a) shows a 3D cursor oriented according to the normal vector of the physical surface and details about its distance to the ground and to the camera. (b) shows a rain particles demo where each rain drop tests for a hit with the physical environment and renders a ripple upon a collision. (c) shows a collision checking example where a virtual chair is occluded by a physical wall.

Screen-space to/from World-space Conversion

Given a screen point $\mathbf{p} = [x, y]$, we look up its depth value in the depth array $\mathbf{D}_{w \times h}$ (in our case: $w = 120,\ h = 160$), then re-project it to a camera-space vertex $\mathbf{v_p}$ using the camera intrinsic matrix $\mathbf{K}$ [15]:

$$\mathbf{v_p} = \mathbf{D}(\mathbf{p}) \cdot \mathbf{K}^{-1} [\mathbf{p}, 1] \tag{1}$$

Given the camera extrinsic matrix $\mathbf{C} = [\mathbf{R}|\mathbf{t}]$, which consists of a 3×3 rotation matrix $\mathbf{R}$ and a 3×1 translation vector $\mathbf{t}$, we derive the global coordinates $\mathbf{g_p}$ in the world space:

$$\mathbf{g_p} = \mathbf{C} \cdot [\mathbf{v_p}, 1] \tag{2}$$

Hence, we have both virtual objects and the physical environment in the same coordinate system. Hit tests can be directly performed with ray casting from the camera location (translation) $\mathbf{t}$ to the screen point $\mathbf{p}$, then to a vertex $\mathbf{g_p}$ in the world space.

The reverse process is simpler. We first project 3D points with the camera's projection matrix $\mathbf{P}$, then normalize the projected depth values and scale the depth projection to the size of the depth map $w \times h$:

$$\begin{aligned} \hat{\mathbf{p}} &= \mathbf{P} \cdot [\mathbf{g_p}, 1], \\ \mathbf{p} &= \left[w \cdot \frac{\hat{\mathbf{p}}_x + \hat{\mathbf{p}}_w}{2\hat{\mathbf{p}}_w}, h \cdot \frac{\hat{\mathbf{p}}_y + \hat{\mathbf{p}}_w}{2\hat{\mathbf{p}}_w} \right] \end{aligned} \tag{3}$$

Through close communication with partner developers, we identified that adapting the depth processing steps to dynamically changing screen orientation and resolution is complicated and time consuming. We simplified these steps and provide convenient conversion utilities, which ensure that every pixel on the screen has a corresponding world vertex measured in meters.

Computing Normal Vectors

Computing usable normal maps out of low-resolution and coarse depth maps can be challenging. With reliable depth values we could compute a normal vector $\mathbf{n}$ with the cross product of vectors formed by adjacent depth values re-projected to 3D vertices. [22]:

$$\mathbf{n_p} = \left(\mathbf{v_p} - \mathbf{v_{p+(1,0)}}\right) \times \left(\mathbf{v_p} - \mathbf{v_{p+(0,1)}}\right) \tag{4}$$

However, such methods may yield noisy or invalid results due to depth discontinuities, holes, and outliers in the estimated scene depth, as shown in Figure 6(b). We provide two real-time algorithms to compute a more stable normal map in real time, on both CPU and GPU (fragment shader). Both components estimate the average normal from 4-ring neighborhoods and cull outliers:

Algorithm 1: Estimation of the Normal Vector of a Screen Point in DepthLab.

Input : A screen point $\mathbf{p} \leftarrow (x, y)$ and focal length f.
Output : The estimated normal vector $\mathbf{n}$.

Set the sample radius: $r \leftarrow 2$ pixels.
Initialize the counts along two axes: $c_X \leftarrow 0, c_Y \leftarrow 0$.
Initialize the correlation along two axes: $\rho_X \leftarrow 0, \rho_Y \leftarrow 0$.
for $\Delta x \in [-r, r]$ **do**
  for $\Delta y \in [-r, r]$ **do**
    Continue if $\Delta x = 0$ **and** $\Delta y = 0$.
    Set neighbor's coordinates: $\mathbf{q} \leftarrow [x + \Delta x, y + \Delta y]$.
    Set $\mathbf{q}$'s distance in depth: $d_{\mathbf{pq}} \leftarrow \|\mathbf{D}(\mathbf{p}), \mathbf{D}(\mathbf{q})\|$.
    Continue if $d_{\mathbf{pq}} = 0$.
    if $\Delta x \neq 0$ **then**
      $c_X \leftarrow c_X + 1$.
      $\rho_X \leftarrow \rho_X + d_{\mathbf{pq}} / \Delta x$.
    end
    if $\Delta y \neq 0$ **then**
      $c_Y \leftarrow c_Y + 1$.
      $\rho_Y \leftarrow \rho_Y + d_{\mathbf{pq}} / \Delta y$.
    end
  end
end
Set pixel size: $\lambda \leftarrow \frac{\mathbf{D}(\mathbf{p})}{f}$.
return the normal vector $\mathbf{n}$: $\left(-\frac{\rho_Y}{\lambda c_Y}, -\frac{\rho_X}{\lambda c_X}, -1\right)$.

Collision-aware Placement

Collisions can be computed with both localized depth and surface depth. Localized depth allows developers to project a world-space vertex to the depth map to check for a collision. On the other hand, surface depth enables features beyond simple collision checking, such as physics simulations and shadows. For simple tasks, such as placing a virtual object, we recommend using localized depth for better performance.

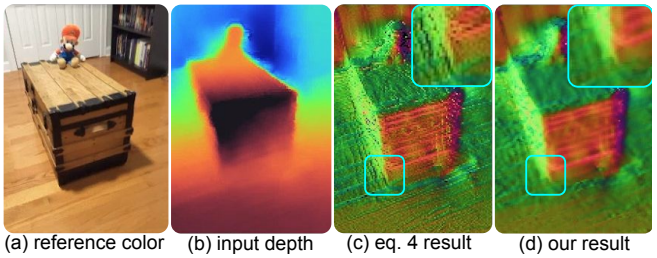


Figure 6. Comparison of the output normal maps between (c) computed by Equation 4 and our result (d) yielded by Algorithm 1. (a) shows the reference color image and (b) shows the input depth map computed from (a) with [46].

We use a majority voting approach to check for collision given a noisy depth map. For example, with the collision mesh (*e.g.*, a simplified mesh such as a bounding box or a capsule) of the virtual object, we can transform the eight corner points into screen space, then test whether its depth value is larger than the physical environment's depth value. If the majority of the corner points are visible, the user may safely place the virtual object in the environment, as shown in Figure 5(c) and the supplementary video for a live demo.

Avatar Path Planning

Figure 7. With localized depth, DepthLab can automatically plan a 3D path for the avatar that avoids a collision with the statue by making the avatar hover over the statue.

AR applications without access to a dense depth map rely on gravity-aligned AR planes to digitally represent the real-world environment. Since these planes only coarsely represent flat horizontal or vertical surfaces, existing AR applications show most virtual character simply moving along a flat ground plane, even when the real world has uneven terrain, or with user's guidance [56]. With localized depth, we can allow AR characters to respect the geometry of physical environments as shown in Figure 7 and in the supplementary video. First, the character is moved parallel to the ground plane. Then the final position of the character is calculated by casting a ray starting at the top of the character down along the gravity vector. At each ray-casting step, the ray's current position is projected to the depth map. If the projected point has greater depth value than that of the depth map, a physical surface has been intersected with the virtual avatar. If so, we set the intersection point as the new character position to avoid the obstacle along the way. We apply the €1 filter [4] to reduce avatar jitters.

Other Use Cases

Localized depth can also enable many interesting visual effects, such as virtual ray reflections and rain drops hitting arbitrary surfaces. Given a starting vertex in the world space, a direction, and a marching speed, we can estimate when and where a ray will hit a physical surface. We can also compute a ray's new reflected direction based on the surface normal at the collision point. We showcase an example of rain particles in Figure 5(b) and ray reflection in the supplementary material.

Interaction With Surface Depth

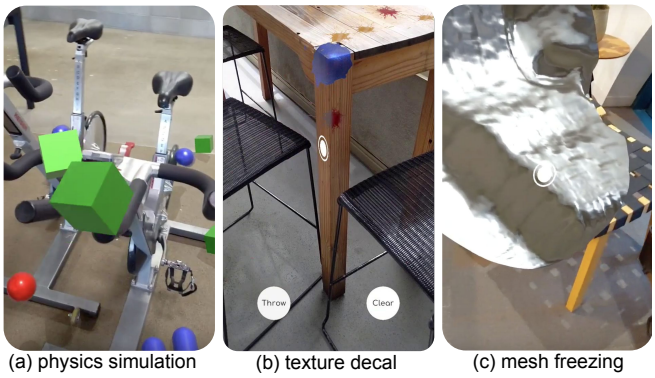


Figure 8. Example use cases of surface depth. (a) shows an AR physics playground, which allows users to throw bouncy virtual objects into the physical scene. (b) shows color balloons thrown on physical surfaces. The balloons explode and wrap around surfaces upon contact with any physical object, such as the corner of a table. (c) shows a material wrapping demo that covers arbitrary shapes with various virtual materials.

Most graphics and game engines are optimized to process mesh data composed of interconnected triangles. Features, such as shadow mapping and physics collision rely on the surface information to compute occlusions and intersections from the perspective of a light source or a rigid body physics object.

AR systems, such as HoloLens or Magic Leap use a time-of-flight depth sensor and a surface reconstruction component to create a persistent volumetric model of the physical environment. Applications receive a mesh-presentation of this volumetric model to compute shadows or physics simulations. Although a persistent volumetric model of the environment offers many benefits, it requires some time for the environment model to build up and become stable. Furthermore, surface reconstruction systems often have high memory requirements and/or high compute.

In our work, we forego surface reconstruction and directly represent environment depth measurements as meshes.

Many phones allow AR content to be rendered on planes and tracked key points anchored in the physical environment. However, the virtual 3D content often looks just pasted on the screen and doesn't show strong visual or behavioral interactions with the real world, *i.e.* virtual objects don't get occluded by real objects and don't collide with real surfaces.

A number of phones have a dedicated time-of-flight (ToF) depth sensor, stereo cameras, or a software algorithm that estimates depth from images of a monocular camera, which can add a detailed understanding of the environment geometry to the AR experience.

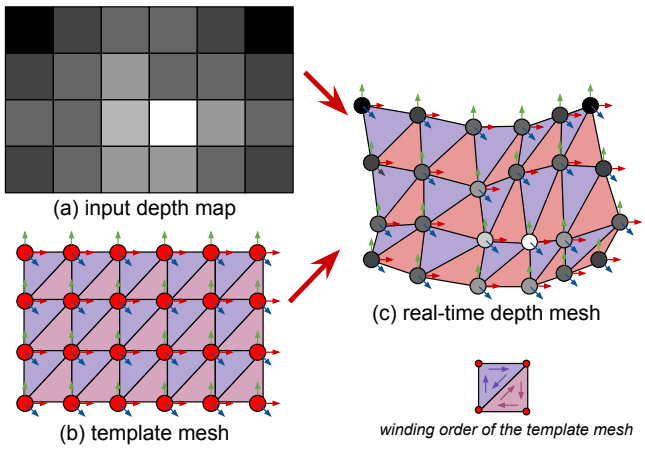


Figure 9. Overview of depth mesh generation. (a) shows an example of input depth map in which brighter pixels indicate farther regions. (b) shows the tessellated template mesh with its vertices arranged in a regular grid and displaced by re-projecting corresponding depth values readily available in the shader. (c) shows the resulting depth mesh consisting of interconnected triangle surfaces.

Real-time Depth Mesh

More sophisticated features, such as shadow calculation and physics collision often use a mesh representation of 3D shapes instead. A mesh is a set of triangle surfaces that are connected to form a continuous surface, which is the most common representation of a 3D shape.

Game and graphics engines are optimized for handling mesh data and provide simple ways to transform, shade, and to detect interactions between shapes. The connectivity of vertices and the surfaces these 3D points form are especially useful when geometric computations are not performed along the projection ray, such as shadow mapping and physics collisions.

We use a variant of a screen-space depth meshing technique described in [18]. This technique relies on a densely tessellated quad (see Figure 9 and Algorithm 2) in which each vertex is displaced based on the re-projected depth value. No additional data transfer between CPU and GPU is required during render time, making this method very efficient.

Tri-planar texture mapping

The appearance of real-world surfaces can be digitally altered with depth meshes.

By computing world coordinates for the depth mesh in a compute shader, we provide customizable assets to apply triplanar texture mapping to the physical world. In this demonstration, users can touch on the screen to change the look of physical surfaces into gold, silver, or a grid pattern (Figure 8).

The depth mesh provides the 3D vertex position and the normal vector of surface points to compute world-space UV texture coordinates. In the simplest case, axis-aligned UV coordinates can be derived from the 3D vertex position. However, these often create stretching artifacts on planar surfaces. Tri-planar UV mapping [13] is a simple technique that yields compelling

Algorithm 2: Real-time Depth Mesh Generation.

Input : Depth map $\mathbf{D}$, its dimension $w \times h$, and depth discontinuity threshold $\Delta d_{\max} = 0.5$.
Output : Lists of mesh vertices $\mathbb{V}$ and indices $\mathbb{I}$.
In the initialization stage on the CPU:
for $x \in [0, w-1]$ **do**
  for $y \in [0, h-1]$ **do**
    Set the pivot index: $I_0 \leftarrow y \cdot w + x$.
    Set the neighboring indices: $I_1 \leftarrow I_0 + 1$, $I_2 \leftarrow I_0 + w$, $I_3 \leftarrow I_2 + 1$.
    Add a temporary vertex $(x/w, y/h, 0)$ to $\mathbb{V}$.
  end
end
In the rendering stage on the CPU or GPU:
for *each vertex* $\mathbf{v} \in \mathbb{V}$ **do**
  Look up $\mathbf{v}$'s corresponding screen point $\mathbf{p}$.
  Fetch $\mathbf{v}$'s depth value $d_0 \leftarrow \mathbf{D}(\mathbf{p})$.
  Fetch $\mathbf{v}$'s neighborhoods' depth values: $d_1 \leftarrow \mathbf{D}(\mathbf{p} + (1,0))$, $d_2 \leftarrow \mathbf{D}(\mathbf{p} + (0,1))$, $d_3 \leftarrow \mathbf{D}(\mathbf{p} + (1,1))$.
  Compute average depth $\bar{d} \leftarrow \frac{d_0 + d_1 + d_2 + d_3}{4}$.
  Let $\mathbf{d} \leftarrow [d_0, d_1, d_2, d_3]$.
  if ***any*** $\left(\textit{step}\left(\Delta d_{\max}, \left|\mathbf{d} - \bar{d}\right|\right)\right) = 1$ **then**
    Discard $\mathbf{v}$ due to large depth discontinuity.
  end
  else
    Convert $\mathbf{v}$ to the world space via Equation 1.
  end
end

real-time results by blending three orthogonally projected textures based on the contributions of each axis of the normal vector.

Grid, dot, or any artificial or natural repeating pattern can be used to texture real world surfaces. An urban landscape can be textured to look as if it is overgrown with plants.

Virtual Shadows

Shadows provide a strong depth cue and are essential for increasing the realism of AR experiences. Conventional mobile experiences without access to depth often render flat virtual shadows using AR planes on the real world, which leads to very noticeable artifacts on non-planar objects.

Real objects need to be represented as meshes to solve this issue, so that they can be treated as part of the virtual scene. With surface depth, we render a depth map of the physical scene represented as a screen-space mesh from the perspective of the light source following [49]. Any scene point that has a greater depth value than that of the light source's depth map is considered to be in the dark and shaded accordingly. This allows the real scene geometry to be rendered from arbitrary viewpoints and allows real and virtual objects to cast shadows on each other.

In a 3D AR scene, the real-world environment is often rendered as a quad texture behind all other scene objects. We modify the rendering parameters of the screen-space mesh to overlay shadows on the AR background, and to only receive shadows on an otherwise transparent mesh and optionally also cast shadows on virtual objects.

Physics Collisions

Physics simulations are an essential part of many interactive experiences and games and are often part of the game mechanism. In AR, collisions between real and virtual objects can further add realism to the entire experience. In the simplest case, physics collisions prevent virtual objects from penetrating real objects. In a more advanced scenario, virtual objects would bounce off of real surfaces in a way we would expect it from real objects.

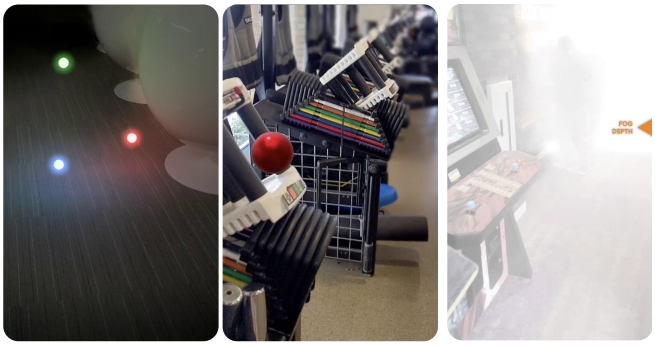
Depth maps do not represent surfaces, but rather point samples of distances, and are not suitable for calculating force vectors and collision normals from arbitrary directions. Unity's physics engine, like many others, supports converting meshes to mesh colliders, which are optimized spatial search structures for physics queries that allow virtual rigid-body objects to hit and bounce off of real surfaces.

The creation of a mesh collider (mesh cooking) happens at a lower resolution on the CPU in real time. However, we only perform it when the user throws a new dynamic object into the AR scene instead of at every frame. This operation is computationally expensive and not continuously needed as the physical environment is mostly static.

Decals and Splats

A sub-region of the depth mesh can be used to create more localized effects, such as decals, virtual graffiti, splat effects, and more on real surfaces. Please refer to the supplementary material and video for examples.

Interaction with Dense Depth



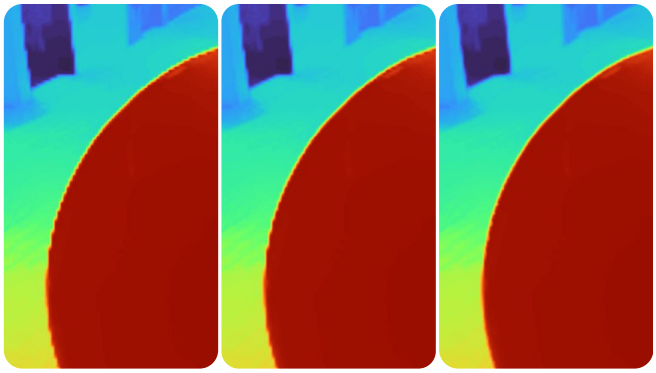
(a) relighting effect (b) aperture effect (c) fog effect

Figure 10. Example use cases of dense depth. (a) shows animating virtual light sources illuminating the carpet and spherical chairs. (b) shows a photography app where a user can anchor the focus in 3D space and the background is adaptively blurred out according to the distance to the focal point. (c) shows a fog effect example where faraway objects are more difficult to see.

Due to the compute constraints on mobile AR, we recommend interactions with dense depth to be implemented on the GPU with compute or fragment shaders. Using this dense depth

to supplement the z-buffer allows many screen-space computational photography techniques to be seamlessly applied to both real and virtual scenes.

Anti-aliasing



(a) input depth map (bilinearly filtered) (b) result depth map with FXAA (c) result with depth-guided FXAA

Figure 11. Comparison between the bilinearly upsampled depth map, post-processed depth map with FXAA, and our result with depth-guided FXAA. Although traditional FXAA smoothes the close (red) depth around curvature, it fails to straighten the lines in the far (blue) regions. With depth-guided antialising, we can adaptively smooth the edges in both near and far regions. The input color image with an egg-shaped chair can be referenced from the first image shown in Figure 2.

Since the raw depth map has a much lower resolution (*e.g.*, 160×120) compared to the phone screen resolution (*e.g.*, 3040×1040 in Pixel 4) and bilinear upsampling may yield pixelation artifacts, we provide a variation of the fast approximate anti-aliasing (FXAA) algorithm [29] with depth guidance. For each pixel on the screen, denote d as its corresponding normalized depth value ranging from 0 to 1. Considering that the closer depth pixels are typically larger, we employ an adaptive kernel size to filter the depth map $\mathbf{D}$ with FXAA:

$$\sigma = s_{\min} + \mathbb{S}_2(1-d) \cdot (s_{\max} - s_{\min}), d \in \mathbf{D} \tag{5}$$

where $\mathbb{S}_2(x) = 6x^5 - 15x^4 + 10x^3$, *i.e.*, the fast smoothstep function introduced in [9]. We empirically determined $s_{\min} = 2, s_{\max} = 3$ pixels as a good value. We show a comparison before and after anti-aliasing the low-resolution depth map on a per-pixel basis in Figure 11.

Relighting

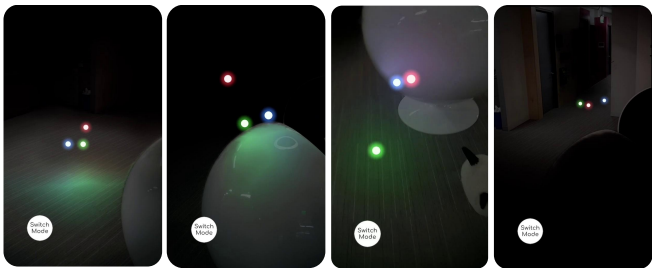


Figure 12. Given a dense depth texture, a camera image, and virtual light sources, we altered the lighting of the physical environment by tracing occlusions along the light rays in real time.

We implemented a per-pixel relighting algorithm that uses low-resolution depth maps, which is based on ray marching and a relighting model introduced in Equation 8 of [6]. Methods based on BRDFs (Bidirectional Reflectance Distribution Functions), such as Phong or Lambertian models [34] require a normal map, which can contain artifacts around object boundaries in low-texture regions as shown in Figure 6(d). Instead, we chose ray marching [10] to compute occlusion-aware relighting without normals. For each pixel on the screen, we evaluate the overall intensity by marching rays from the light source to the corresponding vertex in the world coordinate system, which naturally yields shadows for occluded areas.

ARCore Depth API provides live depth maps to make AR experiences geometry aware. Applications, such as Google's SceneViewer, Search, and a number of shopping apps allow users to preview furniture and other objects in their own space. We aim to improve the realism of such experiences beyond world-aware placement to help users answer questions such as: What will be the effect of the lamp & lighting in my room? Can I try out different light colors and configurations? Can I view my room and objects in a different simulated daytime lighting?

In order to dynamically illuminate the physical scene with virtual light sources, we need to compute the photon intensity at the points the rays intersect with physical surfaces.

Algorithm 3: Ray-marching-based Real-time Relighting.

Input : Depth map $\mathbf{D}$, the camera image $\mathbf{I}$, camera intrinsic matrix $\mathbf{K}$, L light sources $\mathbb{L} = \{\mathscr{L}^i, i \in L\}$ with each light's location $\mathbf{v}_{\mathscr{L}}$ and intensity in RGB channels $\phi_{\mathscr{L}}$.

Output : Relighted image $\mathbf{O}$.

for *each image pixel* $\mathbf{p} \in$ *depth map* $\mathbf{D}$ *in parallel* **do**
  Sample $\mathbf{p}$'s depth value $d \leftarrow \mathbf{D}(\mathbf{p})$.
  Compute the corresponding 3D vertex $\mathbf{v_p}$ of the screen point $\mathbf{p}$ using the camera intrinsic matrix $\mathbf{v_p}$ with $\mathbf{K}$: $\mathbf{v_p} = \mathbf{D}(\mathbf{p}) \cdot \mathbf{K}^{-1}[\mathbf{p}, 1]$
  Initialize relighting coefficients of $\mathbf{v_p}$ in RGB: $\phi_{\mathbf{p}} \leftarrow \mathbf{0}$.
  for *each light* $\mathscr{L} \in$ *light sources* $\mathbb{L}$ **do**
    Set the current photon coordinates $\mathbf{v}_o \leftarrow \mathbf{v_p}$.
    Set the current photon energy $E_o \leftarrow 1$.
    while $\mathbf{v}_o \neq \mathbf{v}_{\mathscr{L}}$ **do**
      Compute the weighted distance between the photon to the physical environment $\Delta d \leftarrow \alpha|\mathbf{v}_o^{xy} - \mathbf{v}_{\mathscr{L}}^{xy}| + (1-\alpha)|\mathbf{v}_o^z - \mathbf{v}_{\mathscr{L}}^z|$, $\alpha = 0.5$.
      Decay the photon energy: $E_o \leftarrow 95\% E_o$
      Accumulate the relighting coefficients: $\phi_{\mathbf{p}} \leftarrow \phi_{\mathbf{p}} + \Delta d E_o \phi_{\mathscr{L}}$.
      March the photon towards the light source: $\mathbf{v}_o \leftarrow \mathbf{v}_o + (\mathbf{v}_{\mathscr{L}} - \mathbf{v}_o)/S$, here $S = 10$, depending on the mobile computing budget.
    end
  end
  Sample pixel's original color: $\Phi_{\mathbf{p}} \leftarrow \mathbf{I}(\mathbf{p})$.
  Apply relighting effect: $\mathbf{O}(\mathbf{p}) \leftarrow \gamma \cdot |\mathbf{0.5} - \phi_{\mathbf{p}}| \cdot \Phi_{\mathbf{p}}^{\mathbf{1.5}-\phi_{\mathbf{p}}} - \Phi_{\mathbf{p}}$, here $\gamma \leftarrow 3$.
end

There are a number of challenges to enable relighting with live depth maps: High-quality normal maps are **not** available. One could compute the intensity with the Lambertian model by using the dot product between the normal vector and the lighting direction. However, in our case this method is not preferred since the normal map computed from the depth map can suffer from many artifacts including empty regions around object boundaries and over-exposed areas. The lighting condition of the physical environment is complex, and we do not know the intrinsic albedo of the materials in the physical world.

3D-anchored Aperture Effect

Figure 13. Wide-aperture effect focused on a world-anchored point on a flower from different perspectives. Unlike traditional photography software, which only anchors the focal plane to a screen point, DepthLab allows users to anchor the focal point to a physical object and keep the object in focus from even when the viewpoint changes. Please zoom in to compare the focus and out-of-focus regions.

Algorithm 4: 3D-anchored Focus and Aperture Effect.

Input : Depth map $\mathbf{D}$, the camera image $\mathbf{I}$, anchored 3D focal point $\mathbf{f}$, and user-defined aperture value γ.
Output : Post-processed image $\mathbf{O}$ with aperture effects.

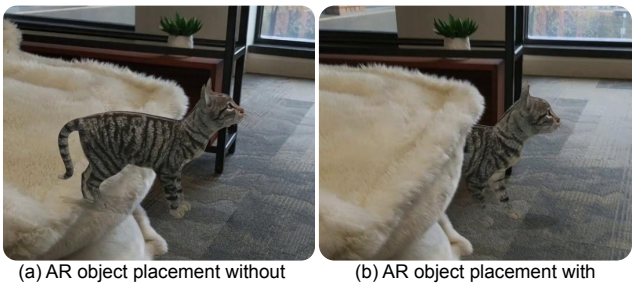
Compute the maximum d_{max} and minimum d_{min} of $\mathbf{D}$.
For $\mathbf{f}$, compute its corresponding screen-space point $\mathbf{p_f}$.
Fetch the depth of the focal point $\mathbf{f}$: $d_{\mathbf{f}} \leftarrow \mathbf{D}(\mathbf{p_f})$.
Compute its normalized depth $\hat{d}_{\mathbf{f}} = \frac{d_{\mathbf{f}} - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}}$.
for *each image pixel* $\mathbf{p} \in$ *depth map* $\mathbf{D}$ *in parallel* **do**
    Sample $\mathbf{p}$'s depth value $d_{\mathbf{p}} \leftarrow \mathbf{D}(\mathbf{p})$.
    Compute its normalized depth: $\hat{d}_{\mathbf{p}} \leftarrow \frac{d_{\mathbf{p}} - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}}$.
    Compute its distance to $\mathbf{f}$: $\Delta d \leftarrow |\hat{d}_{\mathbf{p}} - \hat{d}_{\mathbf{f}}|$.
    Compute the "aperture size":
    $\sigma \leftarrow \textbf{step}(0, \Delta d - \alpha) \cdot (1 - \cos(\beta(\Delta d - \alpha)))$, here $\alpha \leftarrow 0.1, \beta \leftarrow 3$.
    Compute the kernel size of the Gaussian filter:
    $\sigma \leftarrow \gamma_0 + \sigma \cdot \gamma$, here $\gamma_0 \leftarrow 0.1$.
    Apply a two-pass Gaussian filter with N-ring neighborhood in $O(N)$ on the GPU, here $N = 5$.
end

Depth maps allow us to simulate the style of a DSLR camera's wide-aperture picture with a small mobile phone camera. Unlike DSLR cameras or mobile camera apps, which set a focal plane at a certain distance or lock the focus on a 2D region-of-interest (e.g. face), we can anchor the focus point on a physical object and apply Gaussian blur with an adaptive kernel to simulate the Bokeh effects.

Given the user's touch position $\mathbf{p}$, we first compute a 3D vertex $\mathbf{g_p}$ to anchor the focus point. While the user moves the phone, we recompute the distance between the camera and $\mathbf{g_p}$ to set a new focal plane for the wide-aperture effect. We convert the 3D anchor to the view space, normalize the depth values the with local minimum and maximum values to emphasize the objects that are in focus. Finally, we apply Gaussian blur and render the wide-aperture effect on the 2.5D RGBD data.

Occlusion

Occlusion effects are achieved in a per-pixel manner. Each pixel of the virtual object is tested whether it is located behind surfaces of the physical environment or not using the GPU smoothed depth map as introduced in [20, 46]. Note that to reduce the blending artifacts, we perform a soft blur in depth boundary regions as shown in Figure 14(b).



(a) AR object placement without occlusion effects (b) AR object placement with occlusion effects

Figure 14. Before and after the geometry-aware occlusion effects with dense depth. By applying a single fragment shader in DepthLab, AR developers can instantly enhance the realism of their own AR experiences.

DISCUSSIONS

In this section, we discuss how our depth library differentiates from the prior art that uses persistent surface reconstruction. With DepthLab, we aim to uncover opportunities and challenges for interaction design, explore technical requirements, and better comprehend the trade-off between programming complexity and design flexibility.

Interaction with Depth Maps vs. Reconstructed Meshes

Some of our interactive components such as the oriented reticle and geometry-aware shadows are available on modern AR head-mounted displays with reconstructed meshes. However, depth maps are still often preferred in mobile scenarios. Next we list the major differences in methods using depth maps or reconstructed meshes.

- **Prerequisite sensors**: DepthLab requires only a single RGB camera and leverages the ARCore Depth API to run. Systems using reconstructed meshes typically require a depth sensor (HoloLens, MagicLeap, KinectFusion [22]). With an additional active depth sensor, such as time-of-flight sensors (*e.g.*, Samsung Galaxy Note 10+), DepthLab can offer rendering and interaction at a potentially higher quality, but this is not a requirement.
- **Availability**: Depth maps are instantly and always available for all devices while approaches with reconstructed meshes predominantly require environmental scanning before becoming available; instant meshes usually have holes.

- **Physical alignment**: Depth maps have almost per-pixel alignment with the color image while real-time reconstructed meshes only have coarse alignment with low-resolution polygons.
- **Ease of use**: By providing interactive modules in screen space, DepthLab is more directly accessible to people without advanced graphics or computational geometry background.
- **Example use cases**: Both depth maps and reconstructed meshes enable many interactive components such as oriented reticles, physics simulations, and geometry-aware shadows. However, each technique has its own advantages. On the one hand, depth maps can directly enable the wide-aperture effect and lighting effects, while real-time reconstructed meshes usually suffer from artifacts when doing so. On the other hand, depth maps are not volumetric and require advanced post-processing steps to enable 3D object scanning or telepresence scenarios.

Performance Evaluation

To evaluate the performance of key DepthLab components that use surface depth and per-pixel depth, and to gently introduce developers to the capabilities of our library, we made a minimum viable application with a point depth example (oriented reticle), a surface depth example (depth mesh generator), and a per-pixel depth example (visualization of depth map) in Unity 2019.2.

Procedure	Timings (ms)
DepthLab's overall processing and rendering in Unity	8.32
DepthLab's data structure update and GPU uploading	1.63
Point Depth: normal estimation algorithm	< 0.01
Surface Depth: depth mesh update algorithm	2.41
Per-pixel Depth: visualization with single texture fetch	0.32

Figure 15. Profiling analysis for a minimum DepthLab example application with a point depth example (oriented reticle), a surface depth example (depth mesh generation), and a per-pixel depth example (visualization of depth map).

We ran the experiments with a handheld Android Phone released in 2018 (Pixel 3). The application also records the average rendering time per frame in a sliding window of 500 frames to prevent outliers. We ran the experiments in five different locations of a typical household and report the average profiling results of timings in Figure 15 (a). The average CPU consumption is 13% and the memory usage is 223.3 MB.

In the second test, we evaluated the performance of the real-time relighting. We set the number of sampled photons to 2, 4, 8, 16, 32, 64, and 128 respectively and ran the experiment in each setting for five rounds in different locations. We report the mean and standard deviation in Figure 15 (b) and suggest a sampling rate of 4-8 photons per ray for real-time deployment on a Pixel 3 phone. To better understand the effects of our samples in Algorithm 3, we offer a comparison between 8 samples and 128 samples with a pair of input from Middlebury Stereo Datasets [37]. Based on the results shown in Figure 16, we recommend a sampling rate of 8 photons or less per ray for real-time performance on a Pixel 3 or comparable phones. For

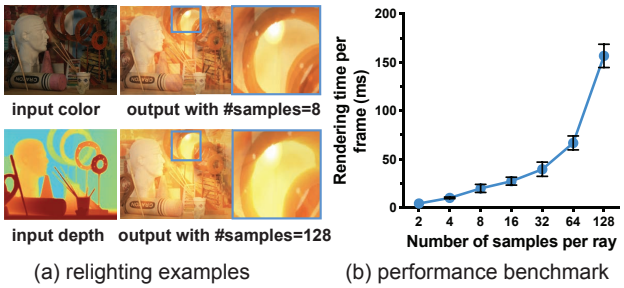


Figure 16. Examples and performance evaluation of real-time relighting. (a) shows a pair of input color and depth images and the corresponding results with 8 and 128 samples per ray in Algorithm 3. (b) shows a performance evaluation with real-time camera images on a Pixel 3. According to the results, we recommend 4-8 samples per ray to deploy our relighting module on Pixel 3 or comparable mobile devices.

computational photography applications, AR developers may leverage a small sampling rate for live preview and a large number for final processing.

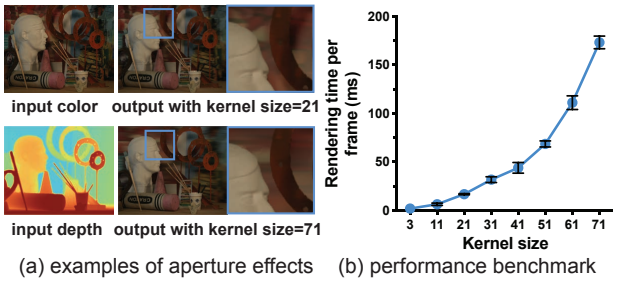


Figure 17. Examples and performance evaluation of real-time aperture effects. (a) shows a pair of input color and depth images and the corresponding results with Gaussian kernel sizes of 21 and 71 in Algorithm 3. (b) shows a performance evaluation with real-time camera images on a Pixel 3. According to the results, we recommend a kernel size of 11-21 to deploy our real-time wide-aperture effect on Pixel 3 or comparable mobile devices.

In the third test, we evaluated the performance of the wide-aperture effect. Similar to relighting, we ran 5 rounds of experiments across 8 kernel sizes: 3, 11, 21, 31, 41, 51, 61, 71. The kernel sizes are selected with odd numbers so that the receptive field is always centered at the pixel to be rendered. With larger sizes of the Gaussian kernel, the out-of-focus regions become more blurry but the performance downgrades significantly. Based on the results shown in Figure 17, we recommend a kernel size of 21 or smaller for real-time performance on a Pixel 3 or comparable phones.

DepthLab as a Reusable Library for Depth Rendering and Interaction

After solving many technical challenges for interacting with real-time depth on a mobile phone, we shared DepthLab with selected partners. In the supplementary video, we show a sped-up video of an external AR developer demonstrating how DepthLab components can accelerate mobile AR development process with Unity prefabs and reusable scripts into their AR games.

LIMITATIONS

While we present a self-contained library for rendering and interaction in mobile augmented reality, our work does have limitations.

DepthLab is designed to enable geometry-aware AR experiences on phones with and without time-of-flight sensors, hence we have yet to explore more in the design space of dynamic depth. With time-of-flight sensors available on many commercial smartphones, we would like to extend DepthLab with motion sensing, gesture recognition, and pose estimation.

We envision live depth to be available on many IoT devices with cameras or depth sensors in the future. Each pixel in a depth map could be associated with a semantic label and help computers better understand the world around us and make the world more accessible for us.

CONCLUSION AND FUTURE WORK

In this paper, we present DepthLab, an interactive depth library that aims to empower mobile AR designers and developers to more realistically interact with the physical world using virtual content. Our primary contribution is the open-sourced, reusable, real-time, depth-based Unity library DepthLab, which enables novel AR experiences with increased realism and geometry-aware features.

We described our interaction modules and real-time algorithms building upon three data structure representations of depth: localized depth, surface depth, and dense depth. On commodity mobile phones with a single RGB camera, DepthLab can fuse virtual objects into the physical world with geometry-aware shadows and occlusion effects, simulate collision and paint splatting, and add virtual lighting into the real world.

We open sourced the DepthLab library on Github (https://github.com/googlesamples/arcore-depth-lab) to facilitate future research and development in depth-aware mobile AR experiences. We believe that this library will allow researchers, developers, and enthusiasts to leverage the base interactions to build novel, realistic AR experiences on regular smartphones. With the general space of perception in AR growing as an active field, we believe there are a number of possibilities that span persistent geometric reconstructions, novel human computer interaction, and semantic scene understanding that will add to making AR experiences more delightful on modern phones or head-mounted displays.

ACKNOWLEDGEMENT

We would like to extend our thanks to Barak Moshe and Wendy Yang for providing a number of visual assets and UX guidance for our open-source code, and to Sean Fanello and Danhang Tang for providing initial feedback for the manuscript. We would also like to thank our UIST reviewers for their insightful feedback.

REFERENCES

[1] Jake K Aggarwal and Lu Xia. 2014. Human Activity Recognition From 3D Data: A Review. *Pattern Recognition Letters* 48 (2014), 70–80. DOI: http://dx.doi.org/10.1016/j.patrec.2014.04.011

[2] Ronald Azuma. 1993. Tracking Requirements for Augmented Reality. *Commun. ACM* 36, 7 (1993), 50–52. DOI:http://dx.doi.org/10.1145/159544.159581

[3] Wolfgang Birkfellner, Michael Figl, Klaus Huber, Franz Watzinger, Felix Wanschitz, Johann Hummel, Rudolf Hanel, Wolfgang Greimel, Peter Homolka, Rolf Ewers, and others. 2002. A Head-Mounted Operating Binocular for Augmented Reality Visualization in Medicine-Design and Initial Evaluation. *IEEE Transactions on Medical Imaging* 21, 8 (2002), 991–997. DOI:http://dx.doi.org/10.1109/TMI.2002.803099

[4] Géry Casiez, Nicolas Roussel, and Daniel Vogel. 2012. 1€ Filter: a Simple Speed-Based Low-Pass Filter for Noisy Input in Interactive Systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2527–2530. DOI: http://dx.doi.org/10.1145/2207676.2208639

[5] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. 2000. Developing a Context-Aware Electronic Tourist Guide: Some Issues and Experiences. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 17–24. DOI: http://dx.doi.org/10.1145/332040.332047

[6] Ruofei Du, Ming Chuang, Wayne Chang, Hugues Hoppe, and Amitabh Varshney. 2019a. Montage4D: Real-Time Seamless Fusion and Stylization of Multiview Video Textures. *Journal of Computer Graphics Techniques* 1, 15 (2019), 1–34. http://jcgt.org/published/0008/01/01

[7] Ruofei Du, David Li, and Amitabh Varshney. 2019b. Geollery: A Mixed Reality Social Media Platform. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 13. DOI: http://dx.doi.org/10.1145/3290605.3300915

[8] Ruofei Du, David Li, and Amitabh Varshney. 2019c. Project Geollery.com: Reconstructing a Live Mirrored World With Geotagged Social Media. In *Proceedings of the 24th International Conference on Web3D Technology (Web3D)*. ACM, 1–9. DOI: http://dx.doi.org/10.1145/3329714.3338126

[9] David S Ebert, F Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. 2003. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann.

[10] Thomas Engelhardt and Carsten Dachsbacher. 2010. Epipolar Sampling for Shadows and Crepuscular Rays in Participating Media With Single Scattering. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, 119–125. DOI: http://dx.doi.org/10.1145/1730804.1730823

[11] Steven Feiner, Steven Feiner, Blair MacIntyre, Tobias Hollerer, and Anthony Webster. 1997. A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. *Personal*

Technologies 1, 4 (1997), 74–81. DOI: http://dx.doi.org/10.1109/ISWC.1997.629922

[12] Martin A Fischler and Robert C Bolles. 1981. Random Sample Consensus: A Paradigm for Model Fitting With Applications to Image Analysis and Automated Cartography. *Commun. ACM* 24, 6 (1981), 381–395. DOI:http://dx.doi.org/10.1145/358669.358692

[13] Ryan Geiss. 2007. Generating Complex Procedural Terrains Using the GPU. *GPU Gems* 3 (2007), 7–37. https://dl.acm.org/doi/book/10.5555/1407436

[14] Chris Greenhalgh, Shahram Izadi, Tom Rodden, and Steve Benford. 2001. The EQUIP Platform: Bringing Together Physical and Virtual Worlds. *Mixed Reality Laboratory-University of Nottingham-UK* (2001).

[15] Richard Hartley and Andrew Zisserman. 2003. *Multiple View Geometry in Computer Vision*. Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511811685

[16] Jeremy Hartmann, Christian Holz, Eyal Ofek, and Andrew D Wilson. 2019. RealityCheck: Blending virtual environments with situated physical reality. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12. DOI: http://dx.doi.org/10.1145/3290605.3300577

[17] Iris Herbst, Anne-Kathrin Braun, Rod McCall, and Wolfgang Broll. 2008. TimeWarp: Interactive Time Travel With a Mobile Mixed Reality Game. *In the Proceedings of MobileHCI 2008, Amsterdam, Netherlands* (2008), 235–244. DOI: http://dx.doi.org/10.1145/1409240.1409266

[18] Otmar Hilliges, David Kim, Shahram Izadi, Malte Weiss, and Andrew Wilson. 2012. HoloDesk: Direct 3D Interactions With a Situated See-Through Display. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2421–2430. DOI: http://dx.doi.org/10.1145/2207676.2208405

[19] Tobias Höllerer, Steven Feiner, Tachio Terauchi, Gus Rashid, and Drexel Hallaway. 1999. Exploring MARS: Developing Indoor and Outdoor User Interfaces to a Mobile Augmented Reality System. *Computers & Graphics* 23, 6 (1999), 779–785. DOI: http://dx.doi.org/10.1016/S0097-8493(99)00103-X

[20] Aleksander Holynski and Johannes Kopf. 2018. Fast Depth Densification for Occlusion-Aware Augmented Reality. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–11. DOI: http://dx.doi.org/10.1145/3272127.3275083

[21] Shahram Izadi, Mike Fraser, Steve Benford, Martin Flintham, Chris Greenhalgh, Tom Rodden, and Holger Schnädelbach. 2002. Citywide: Supporting Interactive Digital Experiences Across Physical Space. *Personal Ubiquitous Comput* 6, 4 (2002), 290–298. DOI: http://dx.doi.org/10.1007/s007790200030

[22] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. 2011. KinectFusion: Real-Time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, Santa Barbara, California, USA, 559–568. DOI: http://dx.doi.org/10.1145/2047196.2047270

[23] Brett Jones, Rajinder Sodhi, Michael Murdock, Ravish Mehra, Hrvoje Benko, Andrew Wilson, Eyal Ofek, Blair MacIntyre, Nikunj Raghuvanshi, and Lior Shapira. 2014. RoomAlive: Magical Experiences Enabled by Scalable, Adaptive Projector-Camera Units. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. 637–644. DOI: http://dx.doi.org/pdf/10.1145/2642918.2647383

[24] Brett R Jones, Hrvoje Benko, Eyal Ofek, and Andrew D Wilson. 2013. Illumiroom: Peripheral Projected Illusions for Interactive Experiences. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 869–878. DOI: http://dx.doi.org/10.1145/2468356.2479531

[25] Hirokazu Kato and Mark Billinghurst. 1999. Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System. In *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)*. IEEE, 85–94. DOI: http://dx.doi.org/10.1109/IWAR.1999.803809

[26] David Kim, Otmar Hilliges, Shahram Izadi, Alex D Butler, Jiawen Chen, Iason Oikonomidis, and Patrick Olivier. 2012. Digits: Freehand 3d Interactions Anywhere Using a Wrist-Worn Gloveless Sensor. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. ACM, 167–176. DOI: http://dx.doi.org/10.1145/2380116.2380139

[27] Minseok Kim and Jae Yeol Lee. 2016. Touch and Hand Gesture-Based Interactions for Directly Manipulating 3D Virtual Objects in Mobile Augmented Reality. *Multimedia Tools Appl* 75, 23 (2016), 16529–16550. DOI:http://dx.doi.org/10.1007/s11042-016-3355-9

[28] Johnny Lee. 2017. Mobile AR in Your Pocket With Google Tango. *SID Symposium Digest of Technical Papers* 48, 1 (2017), 17–18. DOI: http://dx.doi.org/10.1002/sdtp.11563

[29] Timothy Lottes. 2011. Fast Approximation Antialiasing (FXAA). *NVIDIA Whitepaper* (2011).

[30] Xuan Luo, Jia-Bin Huang, Richard Szeliski, Kevin Matzen, and Johannes Kopf. 2020. Consistent Video Depth Estimation. *ACM Transactions on Graphics* 39, 4 (2020). https://arxiv.org/abs/2004.15021

[31] Paweł Nowacki and Marek Woda. 2019. Capabilities of ARCore and ARKit Platforms for AR/VR Applications. In *International Conference on Dependability and Complex Systems*. Springer, 358–370. DOI: http://dx.doi.org/10.1007/978-3-030-19501-4_36

[32] Peter Ondrúška, Pushmeet Kohli, and Shahram Izadi. 2015. MobileFusion: Real-Time Volumetric Surface Reconstruction and Dense Tracking on Mobile Phones. *IEEE Transactions on Visualization and Computer Graphics* 21, 11 (2015), 1251–1258. DOI: http://dx.doi.org/10.1109/TVCG.2015.2459902

[33] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L Davidson, Sameh Khamis, Mingsong Dou, Vladimir Tankovich, Charles Loop, Philip A.Chou, Sarah Mennicken, Julien Valentin, Vivek Pradeep, Shenlong Wang, Sing Bing Kang, Pushmeet Kohli, Yuliya Lutchyn, Cem Keskin, and Shahram Izadi. 2016. Holoportation: Virtual 3D Teleportation in Real-Time. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST)*. ACM, 741–754. DOI: http://dx.doi.org/10.1145/2984511.2984517

[34] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann. http://www.pbr-book.org

[35] Jing Qian, Jiaju Ma, Xiangyu Li, Benjamin Attal, Haoming Lai, James Tompkin, John F. Hughes, and Jeff Huang. 2019. Portal-Ble: Intuitive Free-Hand Manipulation in Unbounded Smartphone-Based Augmented Reality. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, 133–145. DOI: http://dx.doi.org/10.1145/3332165.3347904

[36] Gerhard Reitmayr and Tom Drummond. 2006. Going Out: Robust Model-Based Tracking for Outdoor Augmented Reality. In *Proceedings of the 5th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR '06)*, Vol. 6. Washington, DC, USA, 109–118. DOI: http://dx.doi.org/10.1109/ISMAR.2006.297801

[37] Daniel Scharstein and Richard Szeliski. 2002. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision* 47, 1-3 (2002), 7–42. DOI: http://dx.doi.org/10.1109/SMBV.2001.988771

[38] Toby Sharp, Cem Keskin, Duncan Robertson, Jonathan Taylor, Jamie Shotton, David Kim, Christoph Rhemann, Ido Leichter, Alon Vinnikov, Yichen Wei, Daniel Freedman, Pushmeet Kohli, Eyal Krupka, Andrew Fitzgibbon, and Shahram Izadi. 2015. Accurate, Robust, and Flexible Real-Time Hand Tracking. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, 3633–3642. DOI:http://dx.doi.org/10.1145/2702123.2702179

[39] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. 2011. Real-Time Human Pose Recognition in Parts From Single Depth Images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1297–1304. DOI: http://dx.doi.org/10.1109/CVPR.2011.5995316

[40] Rajinder S. Sodhi, Brett R. Jones, David Forsyth, Brian P. Bailey, and Giuliano Maciocci. 2013. BeThere: 3D Mobile Collaboration With Spatial Input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, ACM, 179–188. DOI: http://dx.doi.org/10.1145/2470654.2470679

[41] Jie Song, Gábor Sörös, Fabrizio Pece, Sean Ryan Fanello, Shahram Izadi, Cem Keskin, and Otmar Hilliges. 2014. In-Air Gestures Around Unmodified Mobile Devices. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, 319–329. DOI: http://dx.doi.org/10.1145/2642918.2647373

[42] Ivan E Sutherland. 1968. A Head-mounted Three Dimensional Display. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. ACM, 757–764. DOI: http://dx.doi.org/10.1145/1476589.1476686

[43] Jonathan Taylor, Lucas Bordeaux, Thomas Cashman, Bob Corish, Cem Keskin, Toby Sharp, Eduardo Soto, David Sweeney, Julien Valentin, Benjamin Luff, Arran Topalian, Erroll Wood, Sameh Khamis, Pushmeet Kohli, Shahram Izadi, Richard Banks, Andrew Fitzgibbon, and Jamie Shotton. 2016. Efficient and Precise Interactive Hand Tracking Through Joint, Continuous Optimization of Pose and Correspondences. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–12. DOI: http://dx.doi.org/10.1145/2897824.2925965

[44] Bruce Thomas, Benjamin Close, John Donoghue, John Squires, Phillip De Bondi, Michael Morris, and Wayne Piekarski. 2000. ARQuake: an Outdoor/indoor Augmented Reality First Person Application. In *Digest of Papers. Fourth International Symposium on Wearable Computers*. IEEE, IEEE, 139–146. DOI: http://dx.doi.org/10.1109/ISWC.2000.888480

[45] Bruce Thomas, Wayne Piekarski, David Hepworth, Bernard Gunther, and Victor Demczuk. 1998. A Wearable Computer System With Augmented Reality to Support Terrestrial Navigation. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers (ISWC '98)*. IEEE, 168–171. DOI: http://dx.doi.org/10.1109/ISWC.1998.729549

[46] Julien Valentin, Adarsh Kowdle, Jonathan T. Barron, Neal Wadhwa, Max Dzitsiuk, Michael Schoenberg, Vivek Verma, Ambrus Csaszar, Eric Turner, Ivan Dryanovski, Joao Afonso, Jose Pascoal, Konstantine Tsotsos, Mira Leung, Mirko Schmidt, Onur Guleryuz, Sameh Khamis, Vladimir Tankovitch, Sean Fanello,

Shahram Izadi, and Christoph Rhemann. 2018. Depth From Motion for Smartphone AR. *ACM Transactions on Graphics (TOG)* 37, 6, Article 193 (2018), 193:1–193:19 pages. DOI:http://dx.doi.org/10.1145/3272127.3275041

[47] Julien Valentin, Vibhav Vineet, Ming-Ming Cheng, David Kim, Jamie Shotton, Pushmeet Kohli, Matthias Nießner, Antonio Criminisi, Shahram Izadi, and Philip Torr. 2015. SemanticPaint: Interactive 3D Labeling and Learning at Your Fingertips. *ACM Transactions on Graphics (TOG)* 34, 5 (2015), 154. DOI: http://dx.doi.org/10.1145/2751556

[48] Jonathan Ventura, Clemens Arth, Gerhard Reitmayr, and Dieter Schmalstieg. 2014. Global Localization From Monocular SLAM on a Mobile Phone. *IEEE Transactions on Visualization and Computer Graphics* 20, 4 (2014), 531–539. DOI: http://dx.doi.org/10.1109/TVCG.2014.27

[49] Lance Williams. 1978. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. 270–274. DOI: http://dx.doi.org/10.1145/965139.807402

[50] Andrew Wilson, Hrvoje Benko, Shahram Izadi, and Otmar Hilliges. 2012. Steerable Augmented Reality with the Beamatron. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, 413–422. DOI: http://dx.doi.org/10.1145/2380116.2380169

[51] Andrew D. Wilson and Hrvoje Benko. 2010. Combining Multiple Depth Cameras and Projectors for Interactions on, above and between Surfaces. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, 273–282. DOI:http://dx.doi.org/10.1145/1866029.1866073

[52] Chi-Jui Wu, Steven Houben, and Nicolai Marquardt. 2017. EagleSense: Tracking People and Devices in Interactive Spaces Using Real-Time Top-View Depth-Sensing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, 3929–3942. DOI: http://dx.doi.org/10.1145/3025453.3025562

[53] Robert Xiao, Scott Hudson, and Chris Harrison. 2016. DIRECT: Making Touch Tracking on Ordinary Surfaces Practical With Hybrid Depth-Infrared Sensing. In *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces*. ACM, ACM, 85–94. DOI:http://dx.doi.org/10.1145/2992154.2992173

[54] Robert Xiao, Julia Schwarz, Nick Throm, Andrew D Wilson, and Hrvoje Benko. 2018. MRTouch: Adding Touch Input to Head-Mounted Mixed Reality. *IEEE Transactions on Visualization and Computer Graphics* 24, 4 (2018), 1653–1660. DOI: http://dx.doi.org/10.1109/TVCG.2018.2794222

[55] Jackie Yang, Christian Holz, Eyal Ofek, and Andrew D Wilson. 2019. Dreamwalker: Substituting real-world walking experiences with a virtual reality. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 1093–1107. DOI:http://dx.doi.org/10.1145/3332165.3347875

[56] Hui Ye, Kin Chung Kwan, Wanchao Su, and Hongbo Fu. 2020. ARAnimator: In-situ Character Animation in Mobile AR with User-defined Motion Gestures. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 83. DOI: http://dx.doi.org/10.1145/3386569.3392404

[57] Tao Yu, Kaiwen Guo, Feng Xu, Yuan Dong, Zhaoqi Su, Jianhui Zhao, Jianguo Li, Qionghai Dai, and Yebin Liu. 2017. BodyFusion: Real-Time Capture of Human Motion and Surface Geometry Using a Single Depth Camera. In *Proceedings of the IEEE International Conference on Computer Vision*. ACM, 910–919. DOI: http://dx.doi.org/10.1109/ICCV.2017.104