Velt: A Framework for Multi RGB-D Camera Systems

Andreas Fender
Aarhus University, Denmark
andreasfender@cs.au.dk

Jörg Müller
University of Bayreuth, Germany
joerg.mueller@uni-bayreuth.de

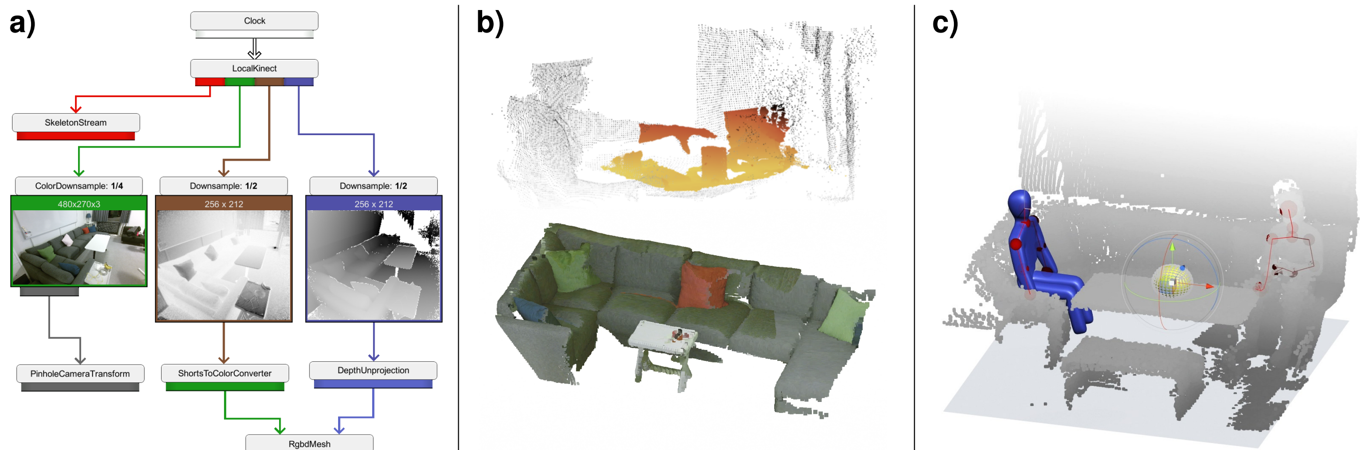


Figure 1. Velt is a modular framework that facilitates the development of multi RGB-D camera systems and applications. a) A node-based GUI makes it possible to inspect and configure local and remote camera streams at runtime. b) Velt generates configurable real-time point clouds based on modular preprocessing. c) Velt has additional functionalities like streaming synthetic data and recording streams.

Abstract

We present Velt, a flexible framework for multi RGB-D camera systems. Velt supports modular real-time streaming and processing of multiple RGB, depth and skeleton streams in a camera network. RGB-D data from multiple devices can be combined into 3D data like point clouds. Furthermore, we present an integrated GUI, which enables viewing and controlling all streams, as well as debugging and profiling performance. The node-based GUI provides access to everything from high level parameters like frame rate to low level properties of each individual device. Velt supports modular preprocessing operations like downsampling and cropping of streaming data. Furthermore, streams can be recorded and played back. This paper presents the architecture and implementation of Velt.

Author Keywords

RGB-D camera systems; point clouds; framework; pipes and filters

CCS Concepts

•**Human-centered computing** → **Interactive systems and tools;**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISS '18, November 25–28, 2018, Tokyo, Japan

© 2018 ACM. ISBN 978-1-4503-5694-7/18/11...$15.00

DOI: https://doi.org/10.1145/3279778.3279794

INTRODUCTION

Over the years, many researchers have built prototypes of 3D user interfaces that combine multiple RGB-D cameras to create point clouds or utilize skeleton tracking. Streaming and combining multiple streams, which often includes network communication, were mostly programmed by the researchers themselves. Working with RGB-D camera SDKs is not difficult in itself and device agnostic frameworks like *OpenNI* further simplify the access to streaming data from a device. However, the complexity quickly increases when the system uses multiple RGB-D cameras. The primary challenges of building multi RGB-D camera systems are:

- Streams need to be transmitted via network in real-time.
- Streams need to be configured, processed and combined.
- Good performance and stability are required.
- The large amount of real-time data makes it difficult to keep an overview and spot problems with the streams.

Especially for research prototype systems, not much time is allocated for achieving stable and efficient streaming capabilities. A naive and quick implementation might cause serious performance bottlenecks and stability issues later on in development. These might cause problems for the application and user experience. Not only are there different types of streams (typically RGB, depth and skeletons), but also multiple devices to be synchronized. Conventional debugging techniques like breakpoints and logging are not feasible due to the large amount of real-time data. Existing toolkit solutions create point clouds [12] or per-device surface reconstructions [9] in

System	Point clouds	Meshes	Skeletons	Record / Play	Pipeline	Inspection
RoomAlive [9]	No	**Yes** (GPU)	Per device	Basic	Fixed	Unity UI
LiveScan3D [12]	**Yes**	No	No	**Yes**	Parameters	3D viewer
CreepyTracker [35]	Limited	No	**Merged**	No	Parameters	Unity UI
Velt	**Yes**	**Yes** (CPU)	**Merged**	**Flexible**	**Modular**	**Inspector**

Table 1. Comparison of Velt with the three most related toolkits and systems. All systems have a server-client architecture to read from multiple camera devices. RoomAlive focuses on interactive projection mapping. LiveScan3D allows for recording point clouds. Creepy tracker is a toolkit for context-aware interfaces. Velt focuses on flexible streaming and debugging capabilities. The framework generates point clouds and mesh reconstructions based on depth streams. Skeleton streams from different devices can be merged into one world space representation. Playback functionalities include adjusting playback speed and jumping to frames. The pipeline, including preprocessing, consists of substitutable modules. An inspector interface on top of the Unity UI enables efficient setup and inspection of device streams.

real-time while allowing configuration of high level parameters like density. However, to develop novel user interfaces, configuring high-level parameters is often not sufficient.

To overcome these challenges, we built Velt, a flexible framework for developing multi RGB-D camera systems. Velt is based on a pipes-and-filters architecture that allows for configuring high level parameters as well as extending and substituting particular streaming components. The *Velt Inspector* interface provides access to each streaming component and allows viewing and configuring streams as well as adding operations. Our contributions are:

1. A framework for the fast, easy and robust implementation of multi RGB-D camera networks. Velt facilitates development of spatial user interfaces and concepts like implicit input.
2. A GUI that enables adding streams and operations as well as adjusting parameters. All of this is possible during runtime without any pre-configuration.
3. A highly flexible and extensible system architecture. Velt is applicable to RGB camera processing pipelines as well as complex multi RGB-D camera systems.

Figure 1 provides an overview of Velt's capabilities. In this paper, we present the architecture of Velt. We discuss the core functionalities, as well as more specialized capabilities, which are all implemented in the current iteration. Furthermore, we provide details about the implementation and performance measurements. Lastly, we present use cases, applications and extensions of Velt.

RELATED WORK

Many researchers have built interactive systems based on one or multiple RGB-D cameras. Using depth cameras and skeleton tracking for interaction typically frees users from wearing devices, potentially to a point where the interface becomes invisible. We focus our attention to systems that utilize more than only the skeleton streams of such devices. The general unobtrusiveness of RGB-D cameras make them useful in the context of medicine [36, 13, 3, 15]. Furthermore, RGB-D cameras are often used for projection mapping and spatial augmented reality systems [2]. For instance, Jones et al. [10] utilize a Kinect camera for reconstruction and radiometric compensation in their *IllumiRoom* system. Only comparatively few systems utilize the full capabilities of multiple RGB-D cameras in real-time. One exception is *RoomAlive* by Jones et al. [9], which combines multiple Kinect cameras and projectors. *Room2Room* by Pejsa et al. [25] uses RGB-D cameras for a projection mapping based video conferencing application. Lindlbauer et al. [16] use RGB-D cameras and a VR headset. They reconstruct the environment in real-time and render it into an immersive virtual scene with an altered appearance. Lemkens et al. [14] built a scalable multi RGB-D camera network to create point clouds. With this, the authors investigate challenges like for instance interference of multiple structured light depth cameras. Furthermore, work has been done on large RGB camera networks for distributed vision-based tracking [20, 24]. For instance, *Panoptic Studio* by Joo et al. [11] is a massive camera network consisting of 480 RGB cameras for motion capture.

The remainder of this section deals with approaches to simplify the development of RGB-D camera systems based on generalized frameworks.

Multi RGB-D camera frameworks

While there are many systems utilizing RGB-D cameras, only few provide a reusable framework. Table 1 compares Velt with the most related frameworks. Wilson et al. [38] developed the *RoomAlive Toolkit*, which generalizes from RoomAlive [9]. Their toolkit consists of three parts: The first part is a dedicated calibration application, which aligns depth cameras and projectors. The second part consists of real-time RGB-D and skeleton streaming capabilities. The third part is a projection mapping rendering pipeline, which simplifies the creation of projection mapping applications. They generate meshes based on the depth streams on the GPU during the rendering process using shaders. However, a software architecture that is optimized for projection mapping makes it difficult to utilize the framework for different use cases without changing the core system components. Our focus is on using the stream data for purposes beyond rendering, e.g., for implicit input [32]. The *LiveScan3D* open source system by Kowalski et al. [12] works very similar to the streaming part of the RoomAlive architecture, but generates point clouds instead of reconstructions. *CreepyTracker* by Sousa et al. [35] is a framework for combining multiple RGB-D cameras to enable development of context-aware systems. However, they do not focus on streaming capabilities, but only process the relatively few points around tracked users. *CreepyTracker*'s functionalities, which are specific to context-aware systems, can be seen as complementary to the streaming capabilities of our framework. The *Proximity Toolkit* [18] enables rapid proto-

typing of proxemic-aware systems. They add an abstraction layer to support different sensor technologies and generate high level proxemic context information between users in an environment.

Some of the above mentioned toolkits incorporate some display technology, but primarily they focus on tracking and interaction. Other toolkits mainly focus on enabling new display technology, but integrate RGB-D cameras for input. Examples include *UbiDisplays* [8] and *ASPECTA* [26].

While Velt also supports projection mapping and multi-projector-multi-display environments, we will only discuss the streaming capabilities of our framework in this paper. That is, the focus is on Velt's modular processing of camera streaming data. We see calibration and rendering as separate parts of a system, possibly implemented as modules or supported by different frameworks. Velt has a simple default module for calibration, but it can also import a calibration from RoomAlive.

Device abstraction

There are previous systems with a focus on device agnostic platforms. Streams and sensor data are abstracted and translated into higher level information. Rosen et al. [29] propose a standard interface called *HomeOS* for different devices in a smart home. Later on (in 2010), *Microsoft* developed an infrastructure with the same name [21] to support a wide range of devices. *ECCE* by Belucci et al. [1] combines and abstracts multiple sensor devices for easy physical programming. They aim at non-programmers or beginners to enable prototyping of simple applications. Schmalstieg et al. [31, p.389–393] provide an overview for augmented reality frameworks that utilize the pipes-and-filters pattern [22] to generalize 3D UI input devices. Reitmayr et al. [28] present the *OpenTracker* software architecture to unify data from different tracking devices. MacWilliams et al. [17] present a similar approach for distributed augmented reality systems.

These toolkits typically do not cope with the same amount of real-time data as RGB-D streaming systems. Besides adding layers of abstraction and hiding complexity, our goal is to still allow for low level access and configurability of RGB-D devices in real-time.

Dataflow GUIs

There are many user interfaces, which are organized in connected nodes to represent objects, data flow, signals etc. This type of interface has been used to process files [34], for simulation [19], in music production [27] and in media compositing [4]. Visual scripting interfaces like for instance *Unreal Engine Blueprints* [7] make use of visual nodes and edges as well. Such interfaces focus on simplifying programming by implementing logic through visual connections.

We chose a similar style for our *Velt Inspector*. However, our user interface focuses on displaying and configuring real-time RGB-D streams to manage the streaming input of an application or to implement processing pipelines for camera streams. We do not aim at fully replacing the need to program application logic or specialized processing steps.

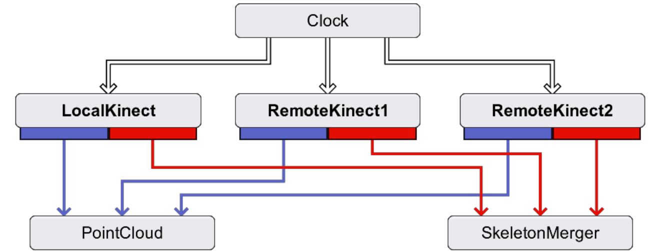


Figure 2. Overview of a system with one local and two remote Kinect devices displayed in the *Velt Inspector*. The device nodes can be expanded for detailed inspection and configuration. Each device creates local point clouds (blue) and skeleton data (red), which are forwarded to the point cloud or skeleton merger, respectively, to generate unified representations in world space. In the simplest case, applications connect to the point cloud or skeleton merger and listen for updates.

EXAMPLE SCENARIO

In the following, we describe an illustrative scenario for the development of a multi RGB-D camera system and a specific application. A developer wants to create a room-scale projection mapping system in Unity. In particular, the system should project up-to-date information about the contents of closed drawers onto them whenever the user approaches. A UI appears next to a drawer, whenever it is opened. The user puts the object that she removed or inserted from the drawer onto a dedicated highlighted area next to the drawer so that the system can identify the object and update the information.

For a system like this, a set of RGB-D cameras would fulfill multiple purposes. An approaching user could be tracked easily based on skeleton tracking. Instead of equipping the furniture with internal sensors, the drawers can be tracked externally based on depth streams. Color streams are necessary to identify the object to be inserted or removed. Multiple RGB-D cameras can be used for good coverage of all furniture in the room and to handle occlusions. These cameras are connected to different machines of the setup. To improve performance, the developer can choose to downsample the depth streams. For the color streams, only a particular region needs to be processed.

While previous toolkits like RoomAlive could easily and efficiently handle the rendering functionalities of this scenario application, they would not be suitable for other parts of the system. For instance, generating point clouds for processing is not directly supported. Inspection of streams would be difficult and preprocessing functionalities like downsampling and cropping would need to be implemented by replacing core system components. We present a framework that is flexible and generic enough to support functionalities like these in a modular way.

SYSTEM ARCHITECTURE

Velt is implemented as a Unity plugin. The Velt system architecture is primarily based on a variation of the pipes-and-filters software pattern (see for instance [33, 22]). We implemented the pattern as a directed acyclic graph, where each node is an operation and edges represent data flow. We refer to source nodes and sink nodes when talking about nodes with no input

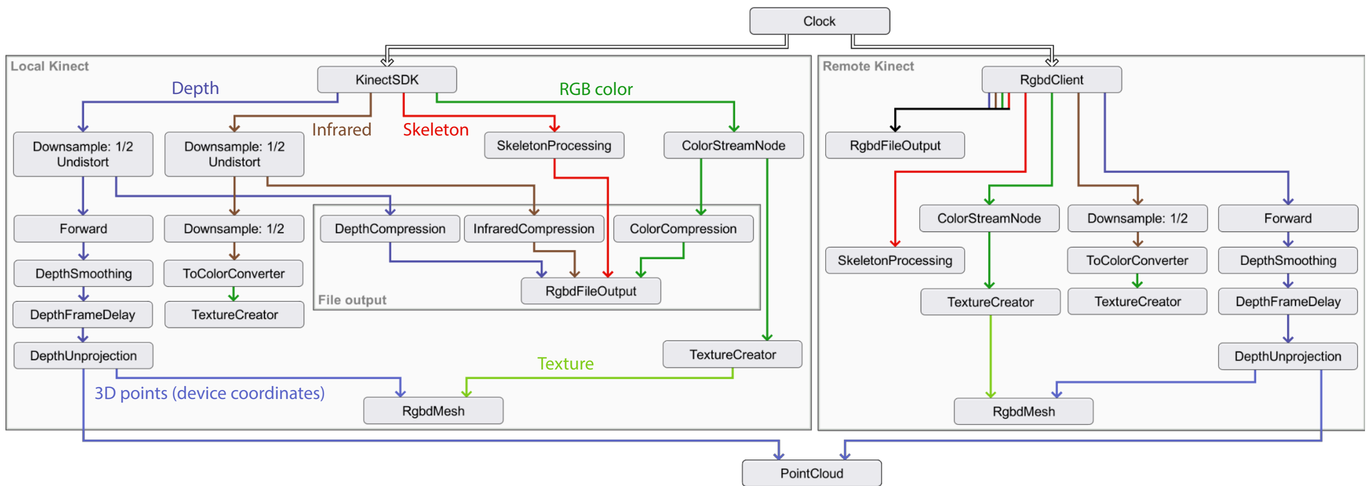


Figure 3. System with a local Kinect (left) and a remote Kinect (right). The figure is generated by the *Velt Inspector*. The clock triggers the devices' source nodes. Each node generates or processes data. Data output is depicted as colored edges. Both Kinects contribute to the point cloud. For each Kinect a mesh is generated in real-time. The textures of these meshes can be based on the RGB streams or the infrared streams. The local Kinect applies operations like downsampling. Furthermore, it compresses the data before streaming it into file. Optionally, data can be downsampled already before writing to the file stream. The local PC receives preprocessed and compressed data from remote machines via TCP through the `RgbdClient` source node. The data can be directly forwarded to the `RgbdFileOutput`. This graph can be rewired depending on the application's needs.

or no output data, respectively. A source node typically binds an RGB-D device like the *KinectV2*. Figure 2 shows a system with three Kinect devices as source nodes and the point cloud as the sink. Within each device, data is processed as can be seen in the expanded view of Figure 3. The source nodes are triggered by a clock, which ticks at a chosen frame rate. Each source node then triggers operations that transform the data and forward it until it reaches the sink nodes. The main idea of this architecture is to not only provide a simplified abstraction layer and common functionalities, but to also allow developers to implement modules that listen to data streams from any node in the pipeline or to subsequently replace nodes with their own operations.

Even though the architecture is not bound to particular RGB-D devices, we focus on setups with multiple Kinect devices for the sake of simplicity. We will discuss generalizations to other devices and setups in later sections.

Inspector GUI

Our *Velt Inspector* GUI allows for inspecting and configuring the system during runtime. The GUI displays nodes, which can be opened to configure their parameters, disable or enable functionalities and more. Nodes, which are not sink nodes, have visually attached output ports. These can be opened to inspect the data that the nodes produce (see Figure 4). Furthermore, nodes are organized in hierarchical groups that can be collapsed and expanded to allow for different levels of granularity. This way, the inspector can display on an overview level (e.g., Figure 2), as well as on the level of every operation of every device (e.g., Figure 3). To streamline working with multiple devices, groups can reference the same node structure, so that processing pipelines can be defined once and then applied to multiple devices, instead of redefining them for each device. Furthermore, variables can be defined to share configuration parameters of nodes among or within groups.

Networking

A typical multi RGB-D setup uses one main PC to run the application and each RGB-D camera device is connected to a separate PC to stream RGB-D data to the main PC [9, 12]. Alternatively, one of the devices can be connected to the main PC directly. We implemented a dedicated streaming program to process and stream data via TCP to the main PC. All streaming PCs process the RGB-D data using the previously described modular pipes-and-filters architecture. That is, the *Velt Inspector* interface on the main PC is also used to define nodes, which are then executed on every streaming PC instead of the main PC. For example, data can be downsampled before it is being transmitted. The main PC has source nodes to receive TCP data (see Figure 3 right). The streams are further processed and eventually accessed by the application.

CORE FUNCTIONALITIES

This section deals with built-in nodes and data types, which are integrated as core functionalities of Velt. Applications can connect to these nodes to receive callbacks.

Point cloud generation

Each device contains a `DepthUnprojection` node. It converts a depth stream into a point cloud in the device's local coordinate frame based on the device's camera intrinsics. The `PointCloud` node (see Figure 2 bottom left or Figure 3 bottom) receives the local point clouds from all devices. This node then combines the points from all devices into a single point cloud in world coordinate space using the cameras' extrinsics. Optionally, the points can be colored by projecting the infrared or color streams onto them. This node also contains a fast and simple voxel-based algorithm for regularizing the point clouds. Redundant points are merged and averaged, if they appear in the same voxel. This reduces the number of points depending on the chosen minimal distance between points.

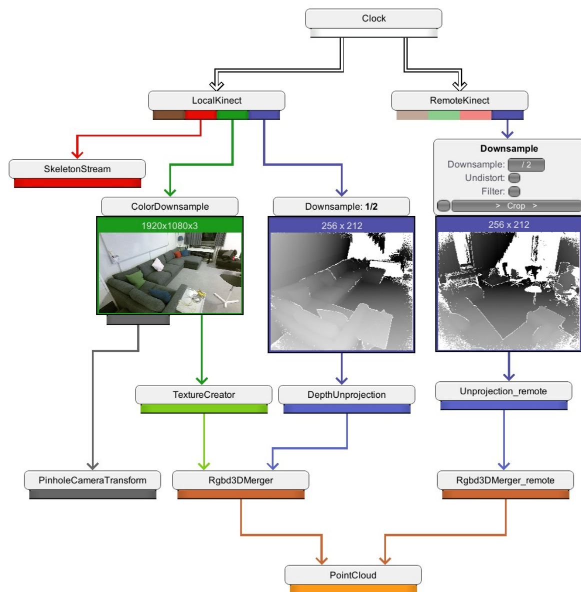


Figure 4. The inspector GUI interface of Velt showing a simplified setup with a local and a remote Kinect. Developers can add and edit nodes and edges to create camera streams and apply operations. Nodes can be expanded to configure their properties (see the remote downsample in the example). The output of nodes can be expanded to inspect the data that they produce. In the example, the output streams of the downsample operations are expanded.

Skeleton merging

Applications can connect to the skeleton output of each device to receive skeleton data in the device's local coordinate system. However, in multi RGB-D camera systems, skeletons are typically needed in a world coordinate frame. In particular, if two devices see the same user from to different view points, then the user should be registered as one user in world space. We merge the different skeleton streams based on distance into one world representation of skeletons similar to *CreepyTracker* [35] or Li et al. [15]. One problem is, that a Kinect device always assumes that users face the camera, i.e., a person tracked from behind creates a skeleton that is rotated 180 degrees. To disambiguate between the two possible directions of a user, we combine the skeleton streaming with face tracking, which is included in the Kinect SDK. Face tracking also provides a coarse head orientation.

Even though we use the term "skeleton", the used data structure is actually a generic graph, thus allowing for different topologies of different devices (e.g., hand tracking devices). However, in this paper, we focus on skeleton streaming. The graph structure is primarily used for different human-skeleton sub topologies. For instance, a Kinect skeleton has fairly many joints to represent the user. However, an application might only need the position of the head and the hands, or even only the head. Therefore, the graph for representing the skeletons can be configured to only stream necessary joints to simplify development and to save resources like file stream size.

Recording and playback

Recording and playback of RGB-D streams are generally very useful features, which is why many frameworks support this functionality. One purpose is to record sessions, e.g., user studies. Sessions can be played back with RGB-D and skeleton data. The application then reacts as if the streams were live. Furthermore, file streaming in Velt is suited for other use cases. In particular, development of room-scale applications can be sped up significantly by having the ability to repeatedly playback recorded interactions. Therefore, we added many parameters for controlling the clock interface in Velt, which in turn controls playback and the main application. Playback can be slowed down or paused with keystrokes to progress frame-by-frame. Conversely, a recording can be played back as fast as computationally possible, to quickly test out a new algorithm or different parameters on the same recording. Furthermore, we allow for jumping to any frame at any time or setting start and end frames for loop regions.

The recording functionality is also modular and provides some parameters. Operations like downsampling can be applied before writing into the file (to reduce filesize) or after reading from the file at runtime. That is, files can be recorded in a low quality and played back as is, or in a high quality and downsampled during playback or as a mixture of both. For instance, if developers are not sure about the required quality, they can record in a high quality and downsample as needed during playback. Compression can be applied as described later in the *Compression* section. There are no restrictions for changing the parameters of the nodes like for instance the factor for downsampling during recording. The parameters are included in the recording as well, i.e., Velt files are played back properly even if parameters change during recording.

If a lot of data is recorded from multiple devices then file streaming on the main PC might slow down the system performance when writing to a storage device. If this becomes an issue, developers can decide to start the recording locally on each machine, instead of the main PC. The per-device streams can then be combined as if they were recorded on the main machine. This is similar to LiveScan3D [12].

Besides recording each device, we support recording merged skeletons in their world-space representation, i.e., the output of the skeleton merger. The skeleton merger is bound to the clock as well, so it has the same playback capabilities as described above. This allows for very lightweight recording and playback of users. Additionally, different recordings can be superimposed and played back in parallel.

Mesh generation

For each streaming device, a 3D mesh can be created based on the unprojected depth data. The topology of the mesh is already given due to the 2D array structure of the depth data. Therefore surfaces can be reconstructed easily in real-time. The meshes are generated as Unity meshes and can be used for rendering and/or as colliders. Updating colliders is a very slow operation in Unity and typically not possible with a high resolution in real-time. To circumvent this, the flexibility of the architecture allows developers to easily generate two meshes depending on the needs: One mesh for rendering with high resolution at high frame rates and a collision mesh based on heavily downsampled depth data and/or at a much lower frame rate.

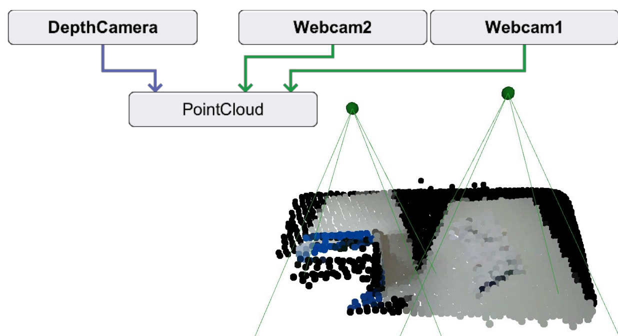


Figure 5. Point cloud with decoupled source nodes. A depth camera and two RGB cameras point at a table with a box on it. The depth camera provides data to calculate the positions of the points. Two separate close-range webcams provide color data. Using their extrinsics and intrinsics, the points can be colored. Black points are outside the FoV of both cameras and have no color.

ADDITIONAL FUNCTIONALITIES

Developers can start simply by using default systems, e.g., by placing a Kinect with default operations into their Unity scene. They can then configure, combine, rewire and extend systems. This section deals with exemplary configurations and system setups that go beyond the default case of using Kinect devices as source nodes.

Combining different camera devices

Using self-contained RGB-D cameras as source nodes for the data flow is the default approach. However, color and depth can also be completely decoupled. For instance, a depth camera might only support depth streams or developers might wish to use a different RGB camera than the one which is built in. In that case, one or multiple calibrated RGB cameras can provide the color data for the point clouds. This way, any depth camera can be combined with any RGB camera, whereas there is only little programmatic difference on the application level. Figure 5 shows an example with a depth-only camera and two RGB camera devices.

Furthermore, different source nodes that are based on different depth camera devices can be combined without restrictions. For instance, Figure 6 shows how a Kinect and a RealSense camera are combined to create a focus+context point cloud.

Synthetic data

Especially in the early parts of a development cycle, it can be very useful to have very controlled synthetic data as input, instead of actual streamed data. Instead of using depth cameras, the input for the point clouds can be synthetic data, such as static or animated 3D models. Similarly, static or animated debugging skeletons can be created in the Unity scene and are treated the same way as streaming data. Hence, developers can use clean 3D models and position skeletons in a completely virtual scene and run the application. Besides debugging, this can also be used to demonstrate a system's behavior in a stylized way or for generating descriptive figures. Lastly, synthetic data can be mixed arbitrarily with live or recorded data without any programmatic effect on the main application. For instance, in Figure 1 (c) a virtual skeleton is placed into the reconstruction of the real scene. Furthermore, a virtual object (yellow sphere) contributes to the point cloud.

Delaying streams

When streams are combined from different camera devices or transmitted via network, then these streams might have different latencies. We implemented a ring buffer node, which can delay streams by a given number of frames to synchronize them with slower streams. To synchronize a system, all delays must be set to synchronize with the stream that has the highest latency. For instance, a number of frames can be set to delay the low-latency local Kinect camera stream according to the latency of remote streams. The point cloud generator will then receive synchronized RGB-D frames of the same point in time.

Refinement operations

We implemented some functionalities to correct and refine streams. For instance, depth streams are especially noisy towards the edges. Developers can crop depth streams to remove these noisy regions and reduce the amount of data as a side effect. Another example are erroneous skeletons: The Kinect occasionally detects inanimate objects as skeletons, often due to reflections. These can be removed before the skeletons are further processed by masking these regions in the streams.

Calibration

The intrinsics and distortion parameters of the Kinect cameras can be retrieved from the SDK. The extrinsics can be adjusted manually and/or calibrated. Velt can parse the XML output of a RoomAlive Toolkit calibration to set the extrinsics of the Kinect devices, including the transformations between the internal RGB camera and IR sensor of each Kinect. However, this requires projectors and is therefore only practical, if projectors are used anyway and high accuracy is needed. Alternatively, Velt supports a simple OpenCV-based calibration method. The extrinsics can be estimated by placing a checkerboard so that it can be detected by the Kinect devices. The modular architecture allows for implementing or binding more sophisticated calibration methods.

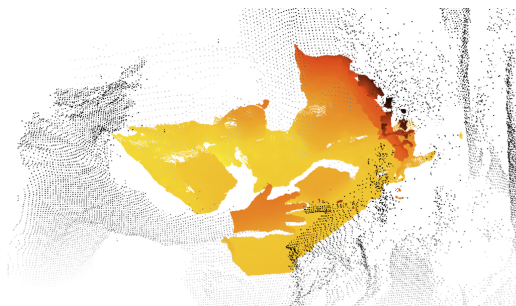
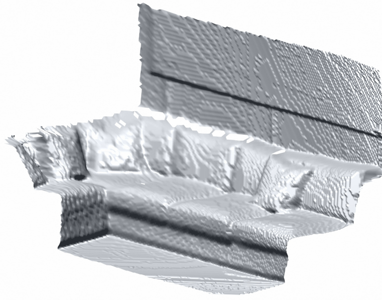


Figure 6. Focus + context point cloud as an example for combining streams from two different depth camera devices. A Kinect camera provides depth streams from a distance to create a point cloud of the desk and the surrounding area with moderate resolution (black dots). A close-range RealSense camera enables creating a high-density region within the focus area (red to orange). Depending on the needs of the application, both devices contribute to the same point cloud or two separate point clouds.

a) Lossless : Gradient (2), RVL (2.4) | **b) Half precision** : LZ4 (4 to 6.2) | **c) JPEG (90% quality)** : (13 to 16)



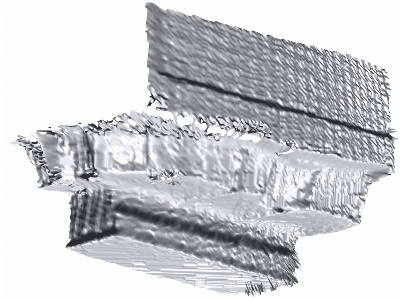


Figure 7. Mesh reconstructions of a downsampled, smoothed and compressed depth frame. The uncompressed size of the downsampled raw depth frame is 256x212 short values = 108544 bytes. Each column has different degrees of information loss due to different compression approaches. The numbers in brackets are approximate ranges of compression ratios (uncompressed size divided by compressed size). Note that these ratios differ with different dimensions of the depth frame. Developers can choose a method that suits the speed, stream data size or precision requirements of the application. (a) Lossless compressions. This category contains a simple gradient based approach and RVL. (b) The 2 bytes per depth value are reduced to byte precision before applying compression. (c) Depth data is very sensitive to JPEG fragments. JPEG compression should only be used if very coarse depth information is sufficient.

IMPLEMENTATION

This section provides some details about the implementation as well as information about how to build applications with Velt. We chose to implement the framework in the *Unity3D* game engine, since its drag-and-drop IDE and direct manipulation capabilities work well in conjunction with the purpose of Velt. For instance, extrinsic parameters of devices are represented as transformations of Unity *GameObjects*.

Creating nodes and data types

Developers can create new nodes and data types by implementing C# classes that derive from the architecture's base classes for nodes and ports. This applies to source nodes, operations and sink nodes. For instance, developers can bind new RGB-D devices or replace operations like downsampling. They can implement operations as needed by the application like for instance a sepia filter for color streams and integrate it as preprocessing. Nodes can be heavily specialized or can be unrelated to RGB-D cameras. For instance, an operation for unwarping image contents or a source node for streaming desktop contents can be implemented. When implementing new nodes or data flow types, widgets can be defined to view and configure them in the inspector.

Compression

To reduce the network usage and file size, we apply image and depth compression. We use well established image compression techniques like JPEG. For depth and infrared we bound and implemented several compression methods to choose from, where each has trade-offs between speed, stream data size and precision. For example, if high precision is not necessary, infrared data can be converted to a gray scale image to then apply JPEG compression. JPEG is often not feasible for depth compression, due to a depth stream's sensitivity to compression artifacts. If speed is the foremost objective, a depth compression that reduces precision and applies LZ4 can be chosen. For the general case, we bind the lossless *RVL* compression [37], which has the overall best trade-off for depth compression in our experience. Figure 7 provides an overview of the integrated depth compression methods.

PERFORMANCE

The architecture of Velt reduces the number of redundant operations. Wherever possible only references are passed on, instead of copying data. For instance, data is not unnecessarily copied when it is needed by multiple nodes. Executing operations and passing on data comes down to method calls and forwarding references. Hence, there is very little overhead, since the computationally expensive parts are implemented within the nodes (some of them in native C++). Elements for debug output, e.g., image representations of depth streams, are only generated whenever they are viewed in the inspector.

Multi-threading

The whole graph can be executed in a single thread or even step-by-step to make debugging easier. However, per default the graph is multi-threaded with an implicit fork and join execution. The clock triggers source nodes in separate threads. A node that has multiple output ports creates threads for each output per default. Nodes with multiple input ports are triggered as soon as data from all input sources has arrived. Each node has two essential callbacks to allow for easy multi-threading: the asynchronous callback and the render thread callback. The asynchronous callback is called whenever the clock ticks and data arrives. This callback is meant for the main computations of the nodes. The render callback is called whenever a frame is to be rendered, but only if there was an asynchronous callback before. This is meant for nodes, which need to push data into the rendering pipeline. As an example, the mesh generator calculates the meshes' vertices and indices in the asynchronous callback parallel to the main loop. However, vertex data can only be pushed to the GPU within the render thread. Therefore, if there is a finished reconstructed mesh, it is transmitted to the GPU within the render thread callback.

Performance tests

Our main setup runs on a gaming PC with an *Intel Core i7 8700K* CPU (6 cores, up to 4.6GHz across cores) and an *NVIDIA GeForce GTX 1080 Ti* GPU. We use different machines and mini PCs for streaming, but they all preprocess and send data at 30Hz.

Point cloud generation

We tested the system with 8 devices in a local network, where each is streaming depth (512x424) and color streams (downsampled to 960x540 and using JPEG compression). The reason for not using full-hd for the color streams is that the 3D points are based on depth streams, which have a lower resolution than RGB and hence many pixels would not be utilized for coloring the points. This is different when generating textured meshes, where access to color data is interpolated between points. The main machine receives approx. 400Mbps through ethernet. In total, approximately $1,700,000$ colored 3D points are generated at every frame. Streaming and calculating all world coordinates without rendering runs at approximately 28Hz in our local network including occasional frame drops. However, depending on the rendering method, the frame rate drops when displaying all points (we currently use the Unity particle system for rendering the points). If the points should be displayed, then defining a clipping volume heavily reduces the number of rendered points and increases the frame rate.

Mesh generation

To form a baseline for our measurements, we chose to compare the mesh generation performance with the release version of *RoomAlive Toolkit* [38]. We measured and compared the performance of streaming RGB-D data from 5 machines on a local network to the main application, i.e., 5 RGB-D meshes are generated. No Kinect was connected to the main PC. We configured Velt to replicate the mesh generation functionality of RoomAlive:

- Streaming depth at the native resolution without any downsampling, smoothing or similar.
- Streaming RGB at Kinect's native full-hd resolution using JPEG compression with 75% quality.
- Generating one textured RGB-D mesh per device including normals and projected texture coordinates

The frame rates are very similar when comparing Velt and RoomAlive (both between 12Hz and 14Hz). Note that the measurements are based on the release version of RoomAlive Toolkit and not more recent iterations (e.g., [16]). The processing resources are distributed differently when comparing both systems. RoomAlive generates the meshes mostly on the GPU, i.e., whenever it is being rendered. Much processing happens in the render thread, which slows down the Unity UI and the overall render frame rate. Velt's mesh generation runs entirely on the CPU and none of the calculations are executed on the render thread. The render thread is only used for transmitting the finished mesh data to the GPU and hence the main thread can be utilized fully by the main application. On the other hand, GPU based approaches for mesh generation can be beneficial for reducing latency. So far, Velt focuses on having data available on the CPU, also for the mesh generation. This makes it easier for further processing, analysis and operations like raycasts. However, in the future we plan to provide a GPU based RGB-D mesh and point cloud generation as an option for applications, which focus on low-latency rendering or further processing using compute shaders.

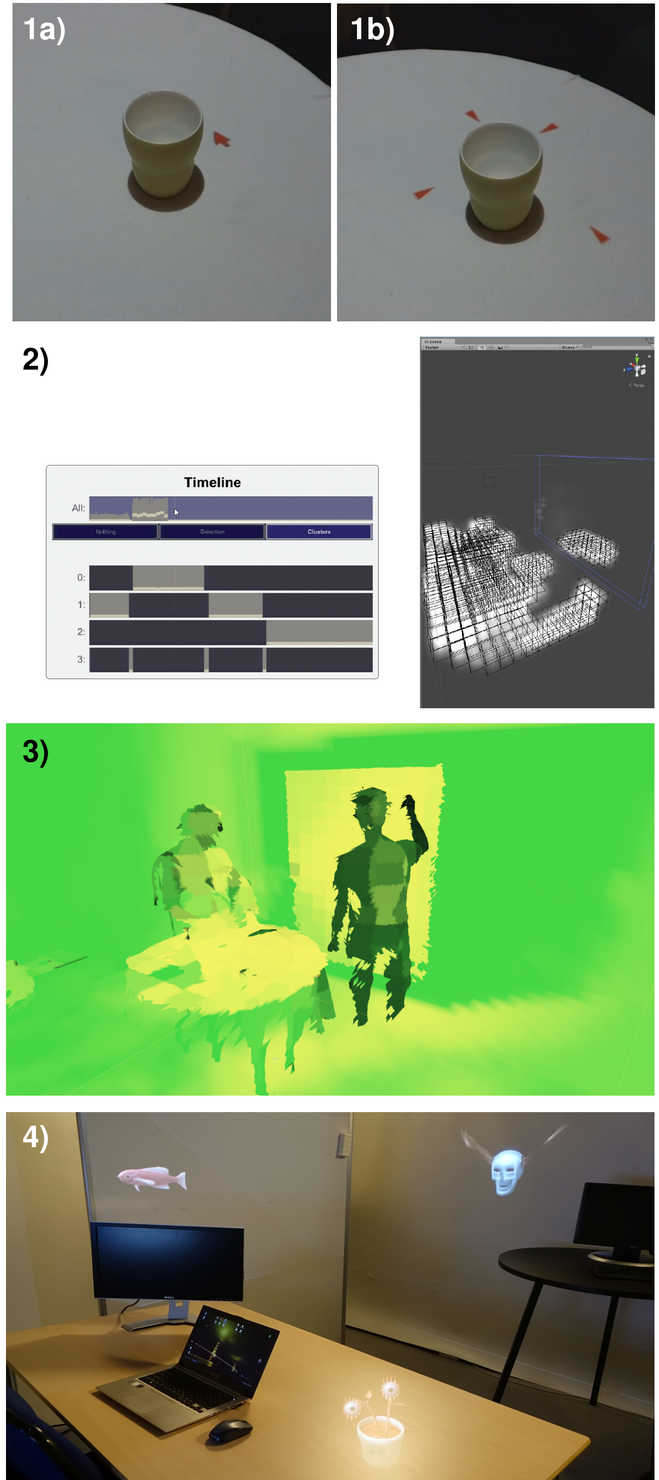


Figure 8. Selection of applications and prototypes based on Velt. (1) Rapid prototype: The combination of skeleton and depth streams makes it possible to point towards a cursor, which is occluded from the user's viewpoint. (2) An extension, which records the room and clusters room layouts over time. In the example, the room has been in three different layouts throughout the recording. The fourth timeline contains transitions. (3) Research project UIST'17: The system measures user viewing behavior and generates heat maps in space. (4) Research project CHI'18: Based on the room's geometry and surface visibility, the system optimizes placement of projection mapping content.

VELT-BASED PROJECTS

Currently and in the past, we used Velt for several projects ranging from simple rapid prototypes based on one RGB-D camera to complex research systems. For instance, Velt was utilized for a student project, in which students used the real-time point clouds of Velt as input for *Point Cloud Library* [30] to test and compare different surface reconstruction algorithms. Figure 8 shows a selection of systems and applications based on Velt. The research systems of the OptiSpace [5] (see Figure 8.3) and HeatSpace [6] (see Figure 8.4) publications are based on Velt and utilize many of its functionalities. Both systems measure the physical environment and user viewing behavior over time. The mesh reconstruction functionalities of Velt were used, e.g., to identify which surfaces can be seen from the users' viewpoints. File streaming was used to facilitate testing of different measurement approaches on the same recordings.

Example extension: room layout clustering

As an example for how to extend Velt, we implemented an extension as a set of specialized nodes. Figure 8.2 shows a screenshot. The extension first measures the room over time and converts the point cloud into a voxel grid. At every frame, a vector is created where each entry represents one voxel. The values of the entries are the number of points contained in each voxel. Afterwards, the different layouts of the physical geometry are identified by applying clustering approaches like k-means to the set of vectors of all frames. For instance, the system generates states for different chair arrangements or other room layout changes, without any prior knowledge of the space. The extension then displays a timeline that shows the different room layouts and when they where active (see Figure 8.2 left). For every state, an approximation of the room layout is visualized. One such visualization can be seen in Figure 8.2 (right). This extension can be useful, e.g., for creating 3D video annotations. Based on the extension, fast state identification is a topic of future research.

DISCUSSION AND FUTURE WORK

This work strongly focuses on RGB-D devices and described the architecture, nodes and data types for multi RGB-D camera systems. However, Velt is also applicable to other camera pipelines. For instance, nodes can be defined for applying *OpenCV* operations to one or more local or remote RGB camera streams. These streams can be displayed, recorded, debugged etc. in the same way as was described in this paper.

In the future, the system can be extended in many directions by implementing new nodes and data types. Specifically, this would mean to support more RGB-D cameras like the *Orbbec Astra* [23] and different types of sensors like the *Leap Motion* to allow for any device ecology. Furthermore, there are many possible operations to implement, such as a Kalman filter for depth streams.

The *Velt Inspector* heavily simplifies working with Velt. However, the arrangement of nodes in the 2D GUI is still done manually. Applying more sophisticated graph drawing algorithms, real-time adjustments and editing capabilities will improve and simplify arranging nodes in the inspector.

CONCLUSION

We presented Velt, a framework for multi RGB-D camera systems. While there are many research prototypes based on RGB-D streams, only comparatively few systems combine multiple RGB-D cameras, since this is difficult to implement properly and it introduces many system parameters. Our framework is designed with a typical development cycle of a room-scale spatial user interface in mind to facilitate prototyping of multi RGB-D camera systems. The configurability, inspection capabilities and playback functionalities of Velt considerably speed up the development of systems. The framework has been useful for implementing different types of multi RGB-D camera systems ranging from rapid prototypes to research projects that analyze physical spaces over time. We are planning to provide Velt as an open source framework in the near future.

ACKNOWLEDGEMENT

This work has been supported by IFD grant no. 3067-00001B for the project entitled: MADE - A platform for future production.

REFERENCES

1. Andrea Bellucci, Ignacio Aedo, and Paloma Díaz. 2017. ECCE Toolkit: Prototyping Sensor-Based Interaction. *Sensors* 17, 3 (2017), 438.
2. Oliver Bimber and Ramesh Raskar. 2005. *Spatial augmented reality: merging real and virtual worlds*. CRC press.
3. Nathan Burba, Mark Bolas, David M Krum, and Evan A Suma. 2012. Unobtrusive measurement of subtle nonverbal behaviors with the Microsoft Kinect. In *Virtual reality short papers and posters (VRW), 2012 IEEE*. IEEE, 1–4.
4. Christie. 2018. Pandoras Box. https://www.christiedigital.com/emea/business/products/media-servers/pandoras-box. (2018). Accessed: 2018-09-24.
5. Andreas Fender, Philipp Herholz, Marc Alexa, and Jörg Müller. 2018. OptiSpace: Automated Placement of Interactive 3D Projection Mapping Content. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 269, 11 pages. DOI: http://dx.doi.org/10.1145/3173574.3173843
6. Andreas Fender, David Lindlbauer, Philipp Herholz, Marc Alexa, and Jörg Müller. 2017. HeatSpace: Automatic Placement of Displays by Empirical Analysis of User Behavior. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 611–621.
7. Epic Games. 2018. Unreal Engine Blueprints. https://docs.unrealengine.com/en-us/Engine/Blueprints. (2018). Accessed: 2018-09-24.
8. John Hardy, Carl Ellis, Jason Alexander, and Nigel Davies. 2013. Ubi displays: A toolkit for the rapid

creation of interactive projected displays. In *The International Symposium on Pervasive Displays*.

9. 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 (UIST '14)*. ACM, 637–644.
10. 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.
11. Hanbyul Joo, Hao Liu, Lei Tan, Lin Gui, Bart Nabbe, Iain Matthews, Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh. 2015. Panoptic studio: A massively multiview system for social motion capture. In *Proceedings of the IEEE International Conference on Computer Vision*. 3334–3342.
12. Marek Kowalski, Jacek Naruniec, and Michal Daniluk. 2015. Livescan3D: A Fast and Inexpensive 3D Data Acquisition System for Multiple Kinect v2 Sensors. 318–325.
13. Gregorij Kurillo, Jay J Han, Alina Nicorici, and Ruzena Bajcsy. 2014. Tele-MFAsT: Kinect-Based Tele-Medicine Tool for Remote Motion and Function Assessment.. In *MMVR*. 215–221.
14. Wim Lemkens, Prabhjot Kaur, Koen Buys, Peter Slaets, Tinne Tuytelaars, and Joris De Schutter. 2013. Multi RGB-D camera setup for generating large 3D point clouds. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 1092–1099.
15. S. Li, P. N. Pathirana, and T. Caelli. 2014. Multi-kinect skeleton fusion for physical rehabilitation monitoring. In *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. 5060–5063. DOI:http://dx.doi.org/10.1109/EMBC.2014.6944762
16. David Lindlbauer and Andrew D. Wilson. 2018. Remixed Reality: Manipulating Space and Time in Augmented Reality. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3173574.3173703
17. Asa MacWilliams, Christian Sandor, Martin Wagner, Martin Bauer, Gudrun Klinker, and Bernd Bruegge. 2003. Herding sheep: live system for distributed augmented reality. In *Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on*. IEEE, 123–132.
18. Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. 2011. The proximity toolkit: prototyping proxemic interactions in ubiquitous computing ecologies. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 315–326.
19. MathWorks. 2018. Simulink. https://www.mathworks.com/products/simulink.html. (2018). Accessed: 2018-04-02.
20. Henry Medeiros, Johnny Park, and Avinash C Kak. 2007. A Light-Weight Event-Driven Protocol for Sensor Clustering in Wireless Camera Networks.. In *ICDSC*. 203–210.
21. Microsoft. 2010. HomeOS. https://www.microsoft.com/en-us/research/project/homeos-enabling-smarter-homes-for-everyone/. (2010). Accessed: 2018-03-29.
22. Microsoft. 2017. Microsoft Azure - Pipes and Filters pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters. (2017). Accessed: 2018-03-29.
23. Orbbec. 2018. Astra. https://orbbec3d.com/develop/. (2018). Accessed: 2018-04-03.
24. Johnny Park, Priya C Bhat, and Avinash C Kak. 2006. A look-up table based approach for solving the camera selection problem in large camera networks. In *Proceedings of the International Workshop on Distributed Smart Cameras (DCS'06), Boulder, CO, Oct*, Vol. 31. 72–76.
25. Tomislav Pejsa, Julian Kantor, Hrvoje Benko, Eyal Ofek, and Andrew Wilson. 2016. Room2Room: Enabling life-size telepresence in a projected augmented reality environment. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. ACM, 1716–1725.
26. Julian Petford, Miguel A Nacenta, Carl Gutwin, Joseph Eremondi, and Cody Ede. 2016. The ASPECTA toolkit: affordable full coverage displays. In *Proceedings of the 5th ACM International Symposium on Pervasive Displays*. ACM, 87–105.
27. Miller Puckette. 2018. PureData. https://puredata.info/. (2018). Accessed: 2018-04-03.
28. Gerhard Reitmayr and Dieter Schmalstieg. 2001. Opentracker-an open software architecture for reconfigurable tracking based on xml. In *Virtual Reality, 2001. Proceedings. IEEE*. IEEE, 285–286.
29. Neal Rosen, Rizwan Sattar, Robert W Lindeman, Rahul Simha, and Bhagirath Narahari. 2004. HomeOS: Context-Aware Home Connectivity.. In *International Conference on Wireless Networks*. 739–744.
30. R. B. Rusu and S. Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *2011 IEEE International Conference on Robotics and Automation*. 1–4. DOI: http://dx.doi.org/10.1109/ICRA.2011.5980567
31. D. Schmalstieg and T. Hollerer. 2016. *Augmented Reality: Principles and Practice*. Pearson Education. https://books.google.de/books?id=qPU2DAAAQBAJ

32. Albrecht Schmidt. 2000. Implicit human computer interaction through context. *Personal technologies* 4, 2-3 (2000), 191–199.

33. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. 2013. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Wiley. https://books.google.dk/books?id=rYiKY3mrrswC

34. Safe Software. 2018. FME. https://www.safe.com/how-it-works/. (2018). Accessed: 2018-03-10.

35. Maurício Sousa, Daniel Mendes, Rafael Kuffner Dos Anjos, Daniel Medeiros, Alfredo Ferreira, Alberto Raposo, João Madeiras Pereira, and Joaquim Jorge. 2017. Creepy Tracker Toolkit for Context-aware Interfaces. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces (ISS '17)*. ACM, 191–200.

36. Rong Wen, Binh P Nguyen, Chin-Boon Chng, and Chee-Kong Chui. 2013. In situ spatial AR surgical planning using projector-Kinect system. In *Proceedings of the Fourth Symposium on Information and Communication Technology*. ACM, 164–171.

37. Andrew D Wilson. 2017. Fast Lossless Depth Image Compression. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces*. ACM, 100–105.

38. Andrew D Wilson and Hrvoje Benko. 2016. Projected Augmented Reality with the RoomAlive Toolkit. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*. ACM, 517–520.