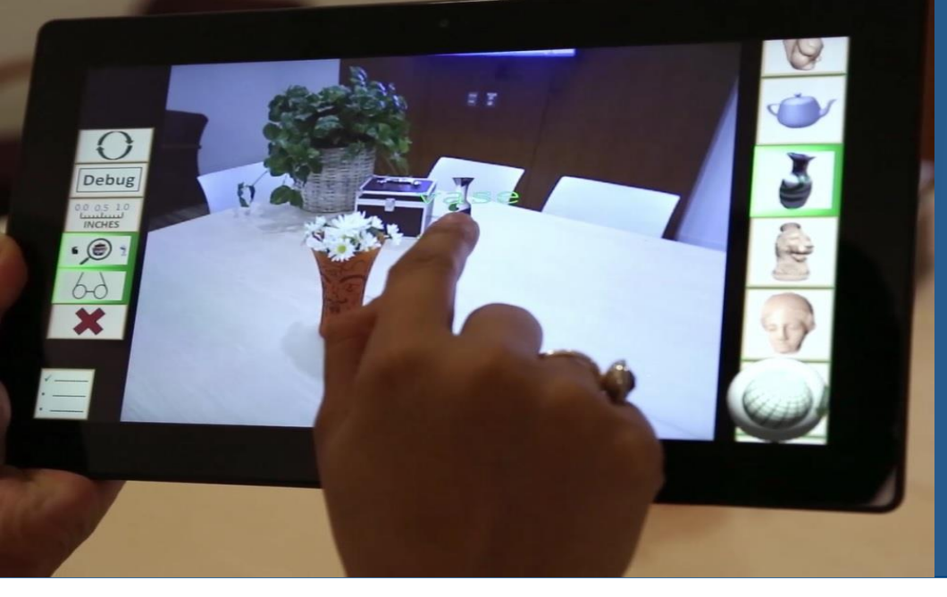


A Compute Model for Augmented Reality with Integrated-GPU Acceleration

Preeti Bindu Jeremy Bottleson Sungye Kim Jingyi Jin
Graphics Initiative Team, VPG, Intel Corporation

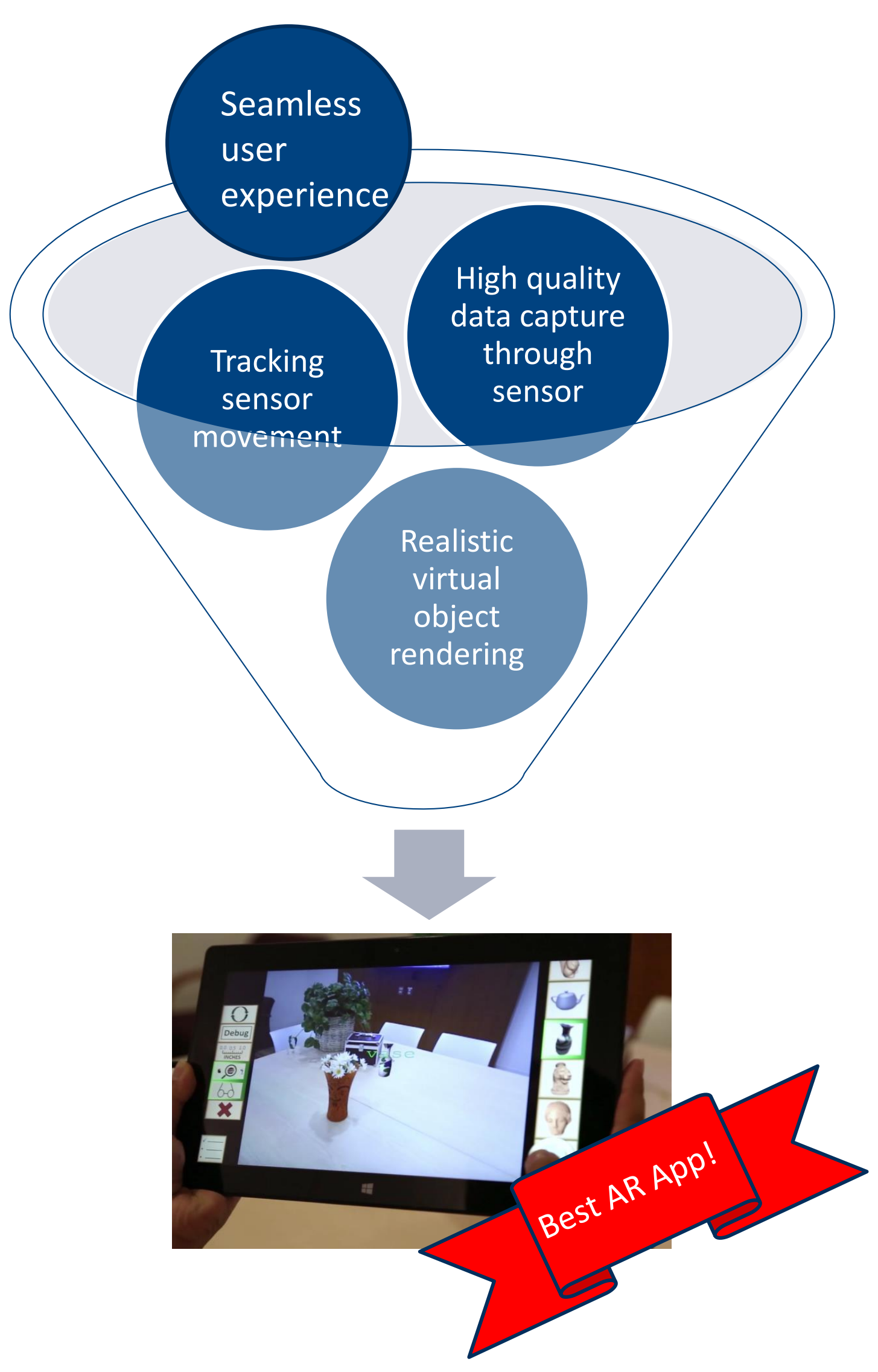
Motivation

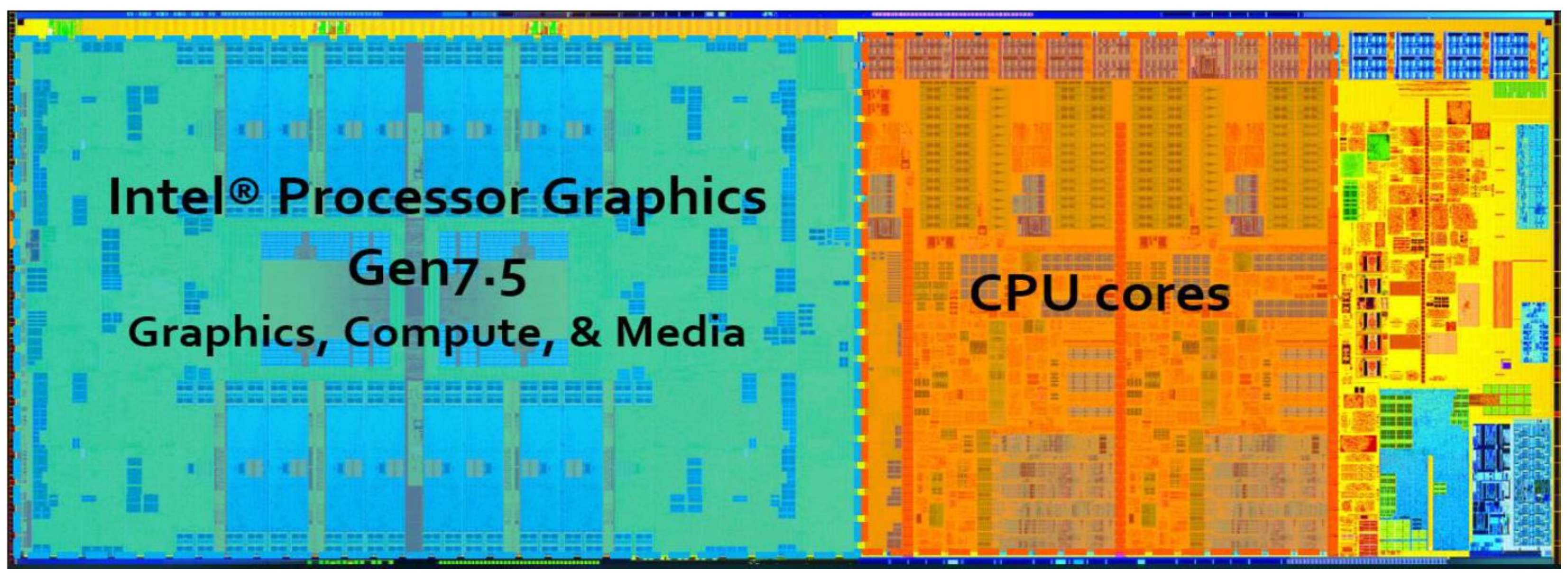
Augmented Reality (AR) is a live view of real-world sequences with enhanced digital information. To enable such enhancement, typical modules of an AR application can be very compute intensive, which is a factor that prevents users from having smooth user experience in low-end devices.

We present an AR application in which the performance is boosted by balancing heavy workload in both CPU and GPU:

1) To capture high-quality data and process such heavy data through color camera and depth sensors;
2) To localize or track sensor movement, which aligns real and virtual views when rendering two worlds;
3) To recover camera pose tracking even if that is lost momentarily;
4) To render virtual objects with photorealism to blend well into the real world;
5) To provide seamless user experience.

Components of a Typical AR App

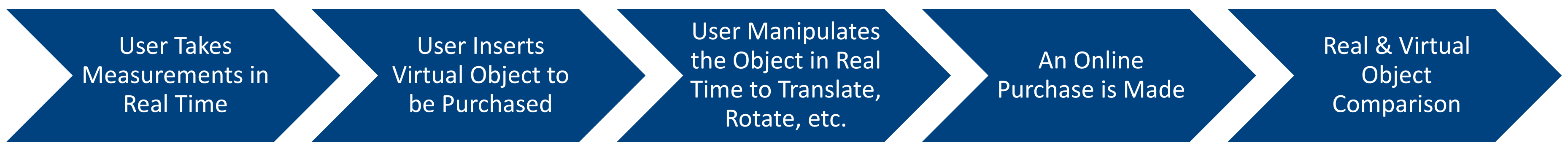




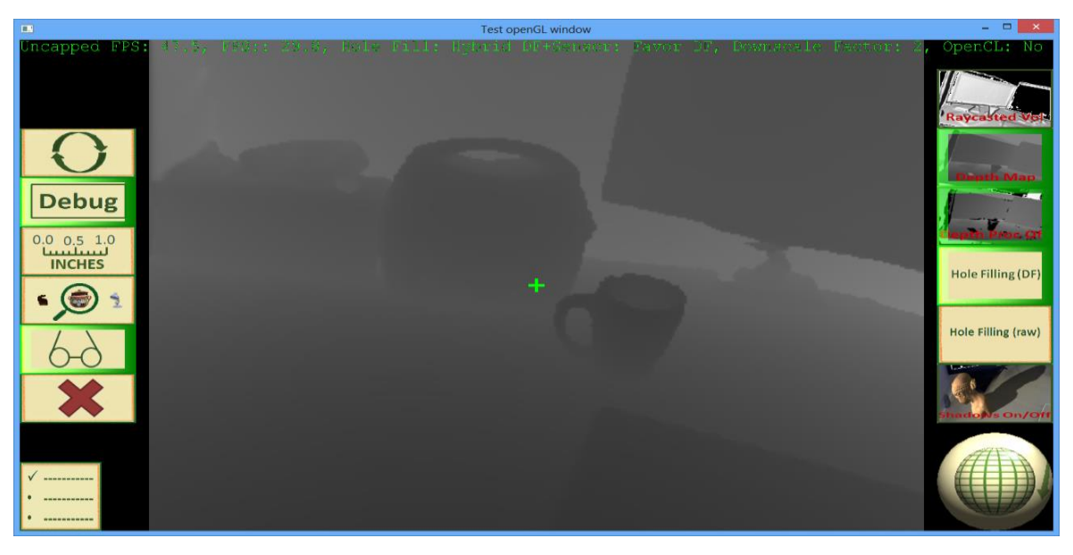
Silicon die layout for 4th Generation Intel® core processor, over half is dedicated to integrated GPU

Use Case

Visual Shopping Assistant (VSA) In Action



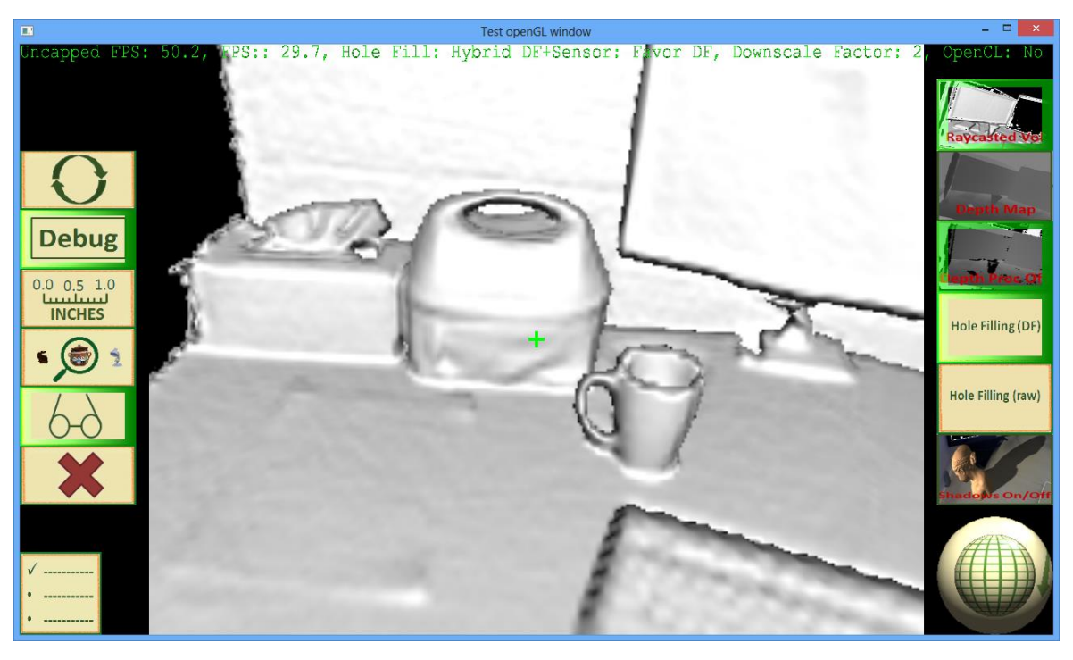
User Takes Measurements in Real Time → User Inserts Virtual Object to be Purchased → User Manipulates the Object in Real Time to Translate, Rotate, etc. → An Online Purchase is Made → Real & Virtual Object Comparison



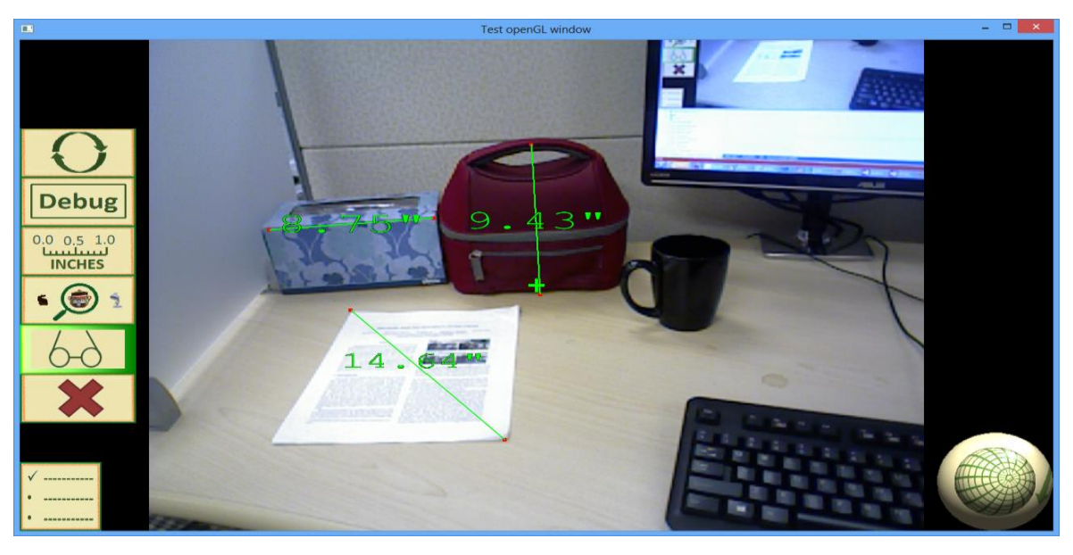
Depth Map

Self Shadow Rendering

VSA

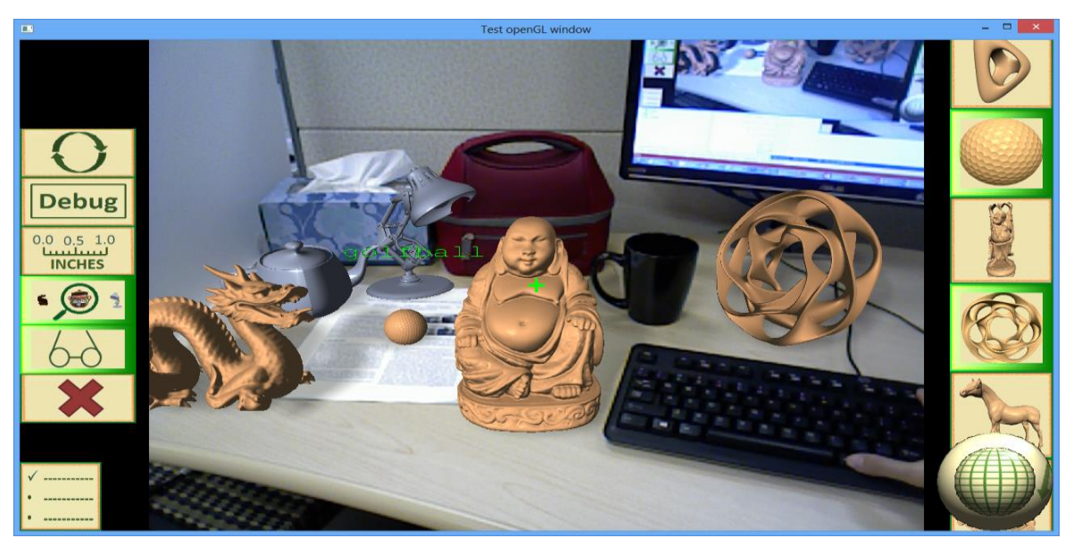


Ray casted Depth Map



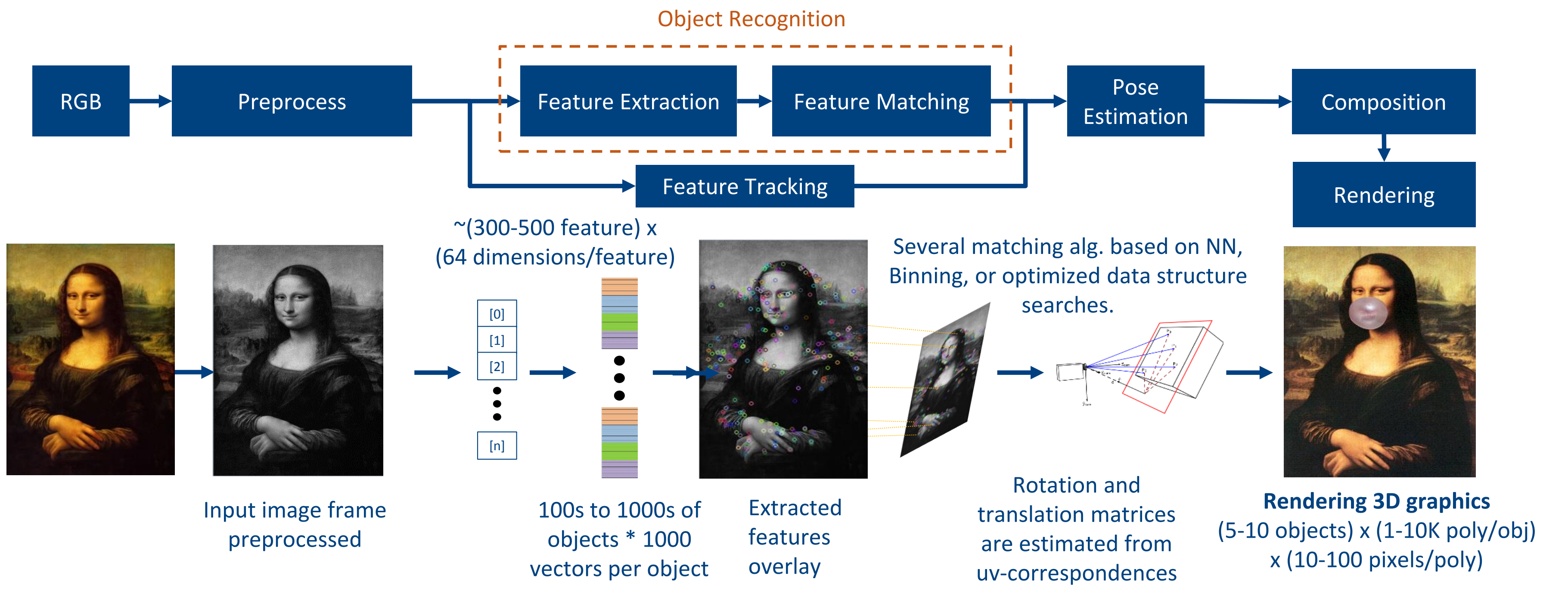
Measurement Tool

Occlusion



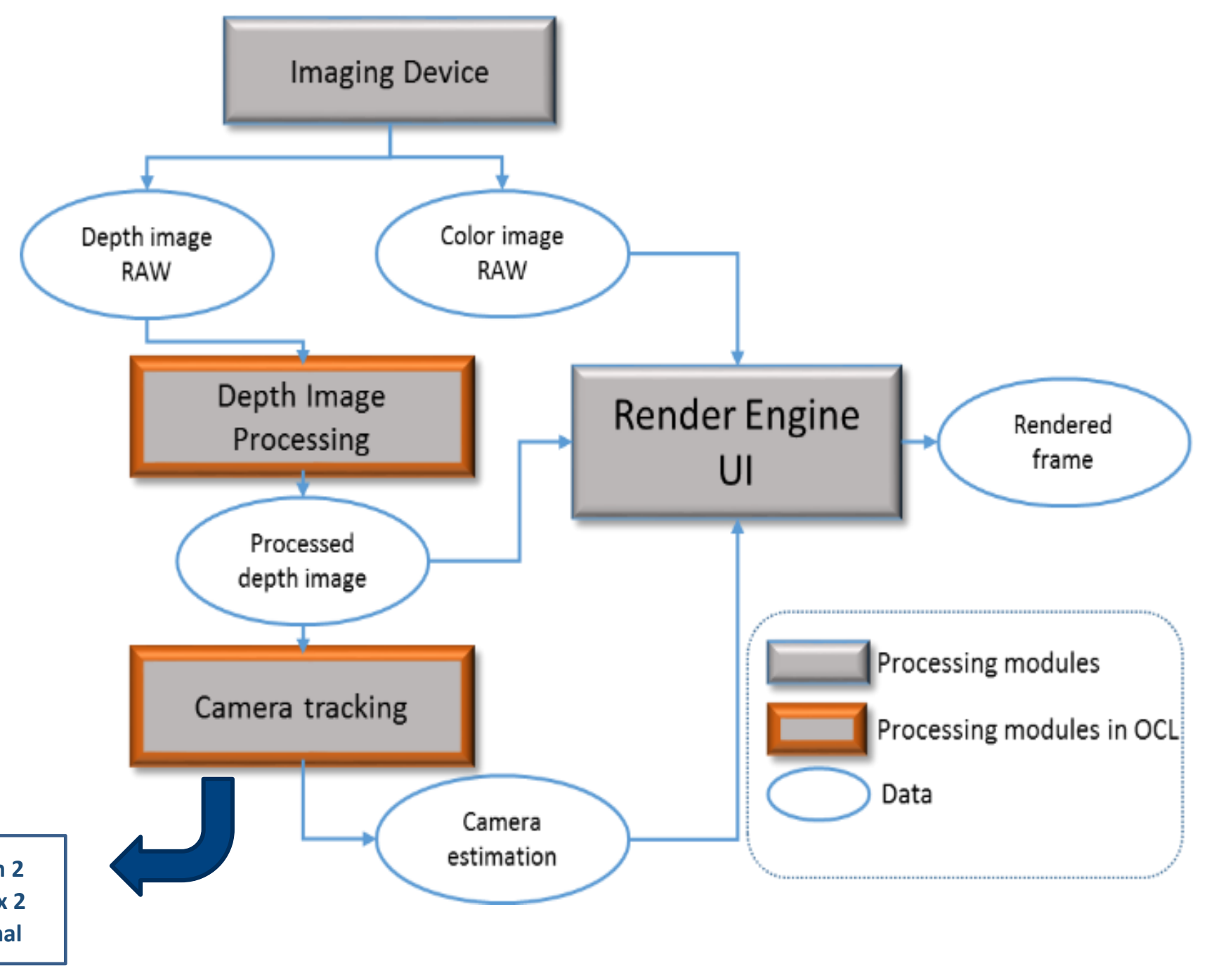
Shadow Rendering

AR Building Blocks

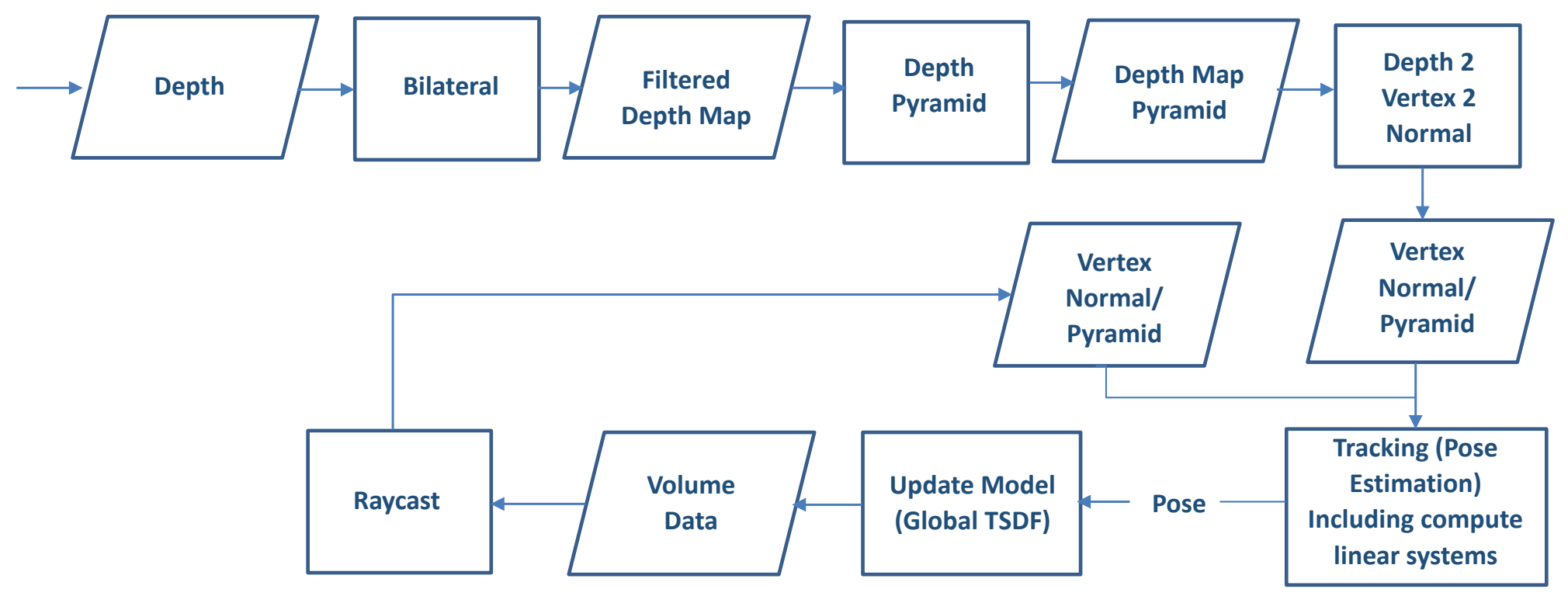


VSA Modules

1) Imaging Device: Real-time RGB & Depth camera sequence capture
2) Depth Image Processing: Fill up "holes" in the raw depth map
3) Camera Tracking: Compute camera pose based on depth image
4) Render Engine/UI: Use RGB and depth images along with camera pose to render real world with virtual 3D objects, Record/playback user interactions



VSA Block Diagram

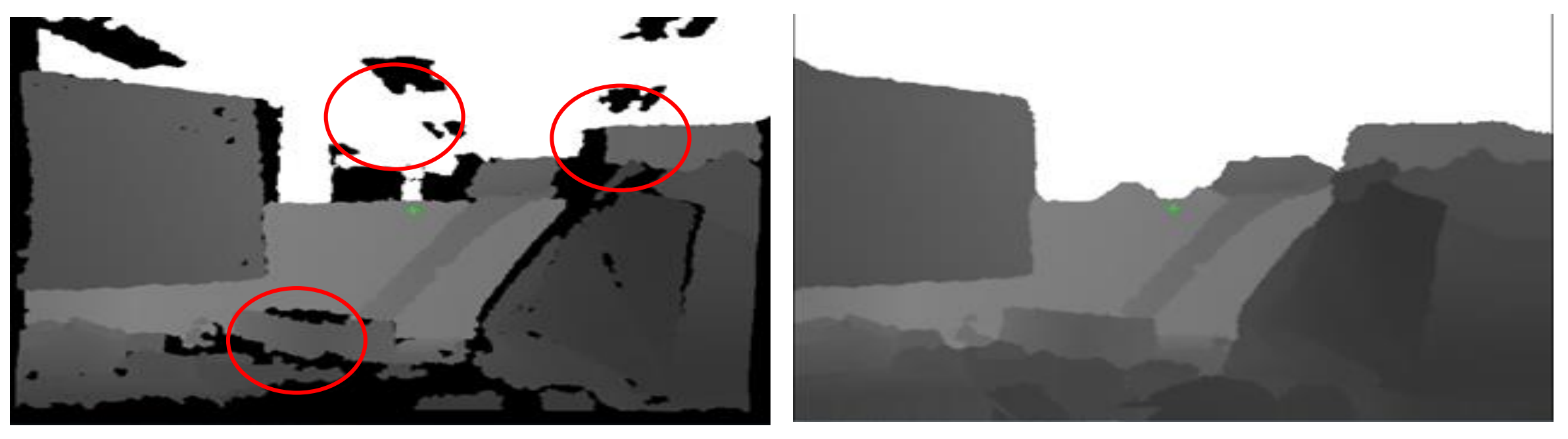


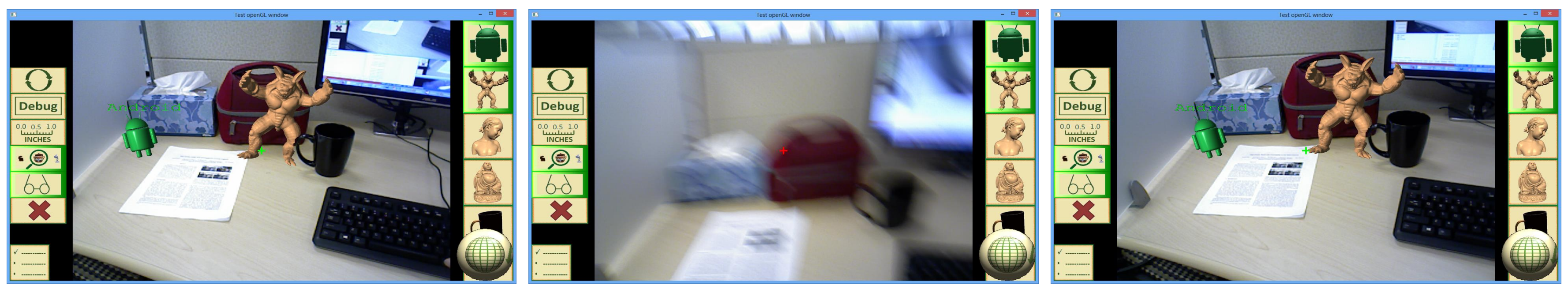
Camera Tracking Algorithm

OpenCL Optimization

Depth Map Processing

1) Fill up "holes" in depth maps
2) Canny edge detection to find delimiters
3) Nearest neighbor or interpolation to fill in the missing pixels.





Tracking and Recovery

1) Recover to get correct camera pose from lost tracking when the depth sensor shakes/loses current depth frame
2) Store keyframes consisting of feature descriptors extracted using BRISK or SURF run on an RGB image and the corresponding camera pose
3) Recover current camera pose by extracting features from the current frame and matching them to the previously generated keyframes.
4) Hamming distance computation for feature matching is implemented in OpenCL

Conclusion & Future Work

Platforms Running VSA
- Windows
- Android

HW Tested
- Intel Haswell on Surface Pro 2
- Intel Atom SoC on Bay Trail

Camera Technology
- Creative Gesture
- Prime Sense
- Kinect

- Incorporate OpenCL 2.0
- Implement more depth processing and tracking algorithms

International Workshop on OpenCL
12-13 May 2015