Augmented Reality Full Theory

0_Overview

Table of Contents

Introduction to Augmented Reality

Definition

Azuma 98

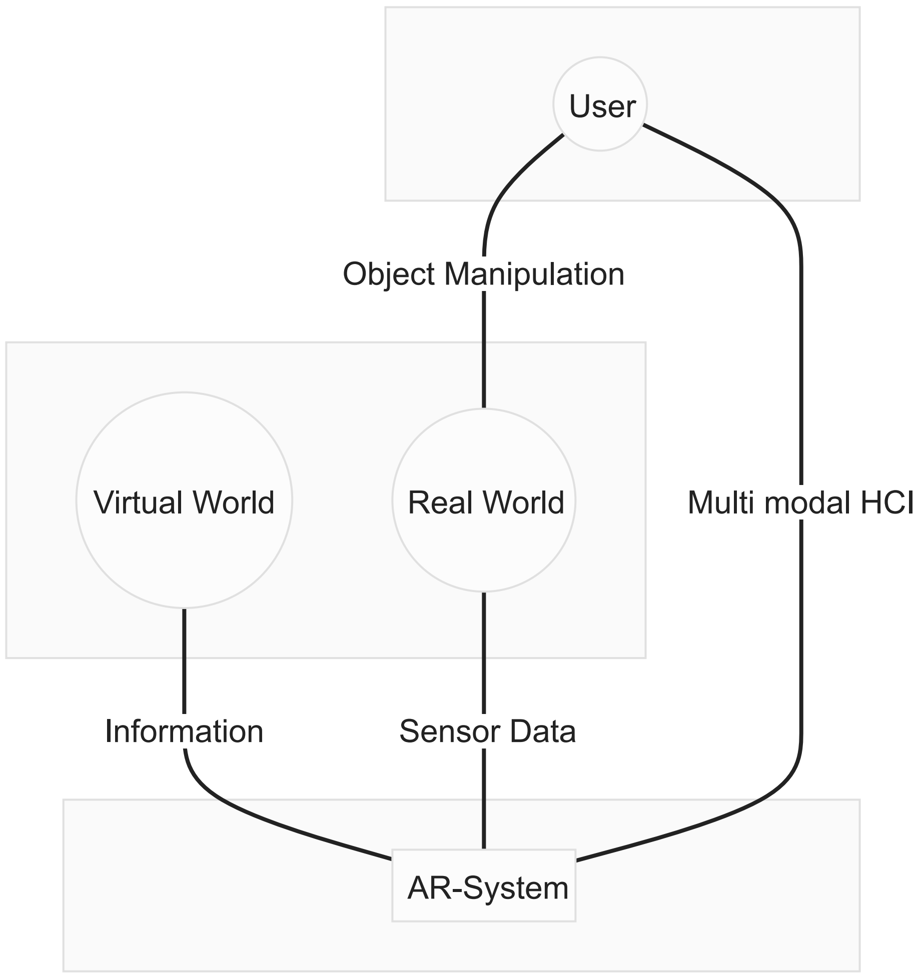
- Combination of Real + Virtual Space
- Interactive in real-time
- Registered in 3 Dimensions

Mixed Reality Continuum (Milgram and Kishino 94)

Mixed Reality describes the field in which we interact with a given combination of Enviroments (virtual and real):

$$Real\ Enviroment \leftarrow \overbrace{Augmented\ Reality\ (AR) \leftrightarrow Augmented\ Virtuality\ (AV)}^{Mixed\ Reality\ (MR)} \rightarrow Virtual\ Envirome$$

Multimedia Combination of Reality and "Virtuality"



Approaches

1. Head-based (Head-mounted Displays HMDs): Typically Goggles or VR-Glasses
2. Hand-based: Hand-held via Smartphone or Game-Console
3. Desktop-based: with fixed camera setup interacting with a "virtual desk"
4. Real Surfaces: Interacting with object recognition in camera
5. Multi-touch: Displays that allow multi-touch interaction
6. Hybrid and Tangible: Combination of Approaches

Sensors

1. Optical Sensors
2. Intertial Sensing
3. Radio-based Sensors ?
4. Acoustic Sensors
5. Mechanical Sensors
6. Outside-in vs. Inside-out
7. Sensor limitations vs. Human Limitations

Basic Computer Vision

#detection #bottom-up #top-down

Top-Down vs. Bottom-Up Computer Vision

Top-Down: virtual model → image
Bottom-Up: image → virtualized scene description

Feature Detection

Region-based Algorithms

- search for areas of similar pixels
- regions can be used to analyse features/ objects/ scene descriptions

Edge-based Detection

- define "cut-off point" (Sobel-Filter) and find large differences between neighbouring pixels (edgels)
- simple edge detectors can be described as masks

Feature Detection/Identification

- either top-down or bottom-up: using a scene model or a heuristic assumption
- find specific pixel blobs and calculate shapes from these blobs → compare with heuristic assumption of shape
- predict motion from computation of current local motion → estimate next feature detection

Coordinate Systems

#SRG #Scene-Graph #homogenous-matrix-transformations #DoF

Transformations

See in the Practical Chapter:

- Vectors
- Matrices
- Homogenous Matrix Operations

Scene Graphs

Motivation

World consists of:

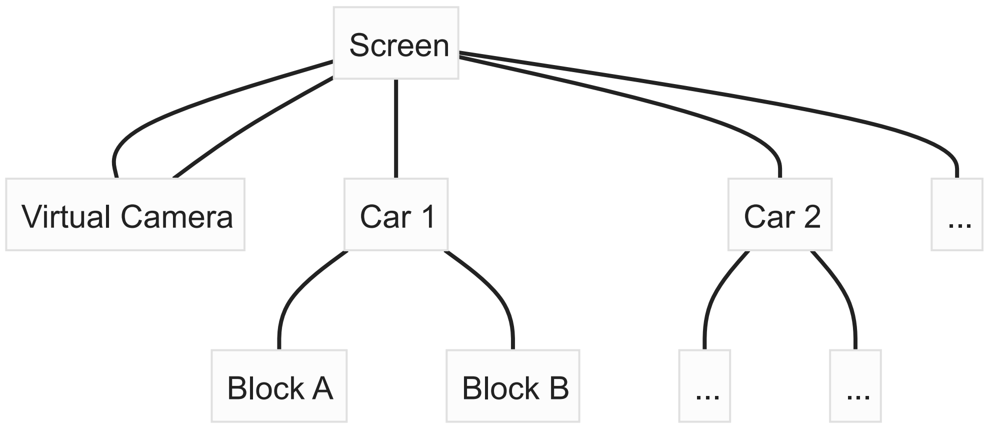
- **People (users):** hands, eyes, ...
- **Objects:** Subparts, tracked Targets, ...
- **Trackers**
  We want to:
- describe objects individually
- replicate objects without having to redescribe geometric details
- determine current pose of an object with respect to its relations

Components

1. Nodes (Coordinate Systems):
   - Camera (eye)
   - Scene
   - Groups of Objects
   - Objects Parts
2. Directed Edges (geometric transformations):

- Changes in Pos, Rot, Scale, Perspective → relative to predecessor
- In Graphics: typically a tree
- in AR: a true Graph (Spacial Relationship Graph (SRG))

Example for a Scene Graph:



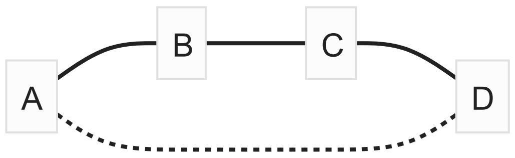
Spatial Relationship Graph (SRG)

Original Paper on SRGs in AR

3D World	SRG
Objects	Nodes
Things (physical or digital)	Coordinate systems ("pose")
Users	
Sensors	
Spatial Relationships	Edges
	Transformations
Time-dependent relationships	Properties of edges
static	registration
	calibration
dynamic	tracking
Operations	Properties of paths / cycles
Sensor Fusion	Graph traversal

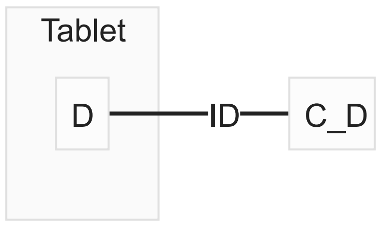
Edge Attributes

- Edges are **directed**
- Translation with Degrees of Freedom (6 DoF, 3 DoF, ...)
- Transformation parameters
- Static Edges are solid while dynamic edges are dashed
- Estimation Method (direct → solid, derived → dashed)
  → Transformations of coordinate System A to D is not known
  → but there is a concatination of direct transformations
  → we can derive the transformation and signify this with a dotted line

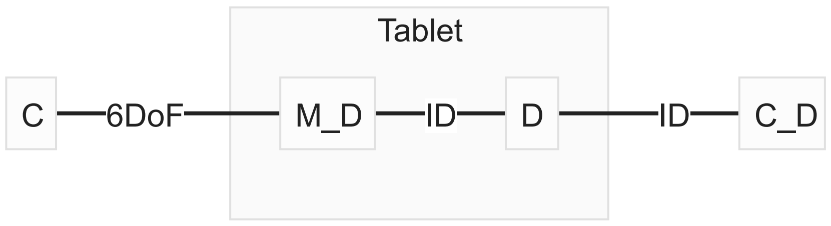


Example for SRG in Augmented Reality Scene

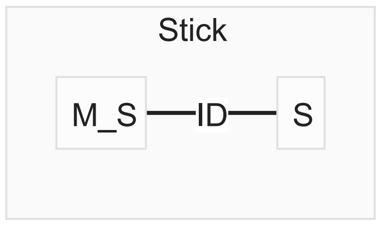
1. Tablet with Tracked Display D_T connected to a camera C_D which shows our final AR Scene of tracking in objects in a seperate room:



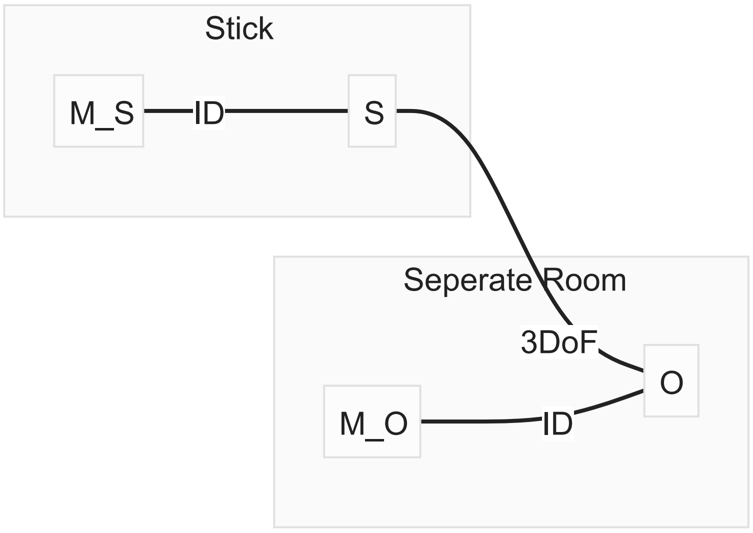
2. Tablet has a Marker M_D that is tracked by an outside-in camera tracking system C:



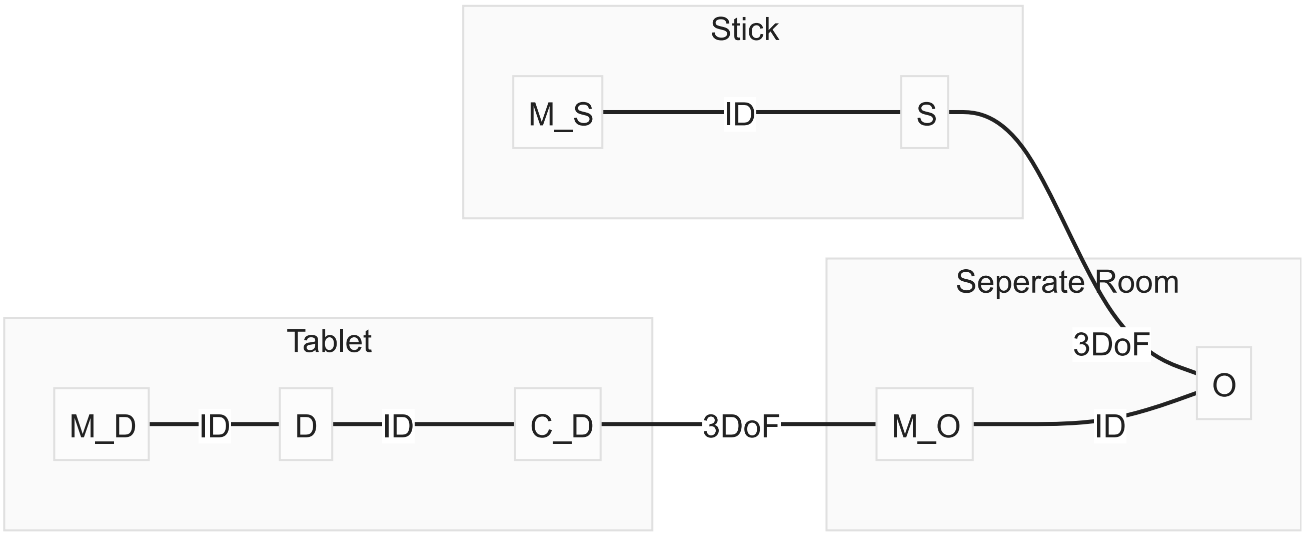
3. We have one Control Stick S (move and push: 3DoF) that is identified with a Marker M_S and can move through our scene



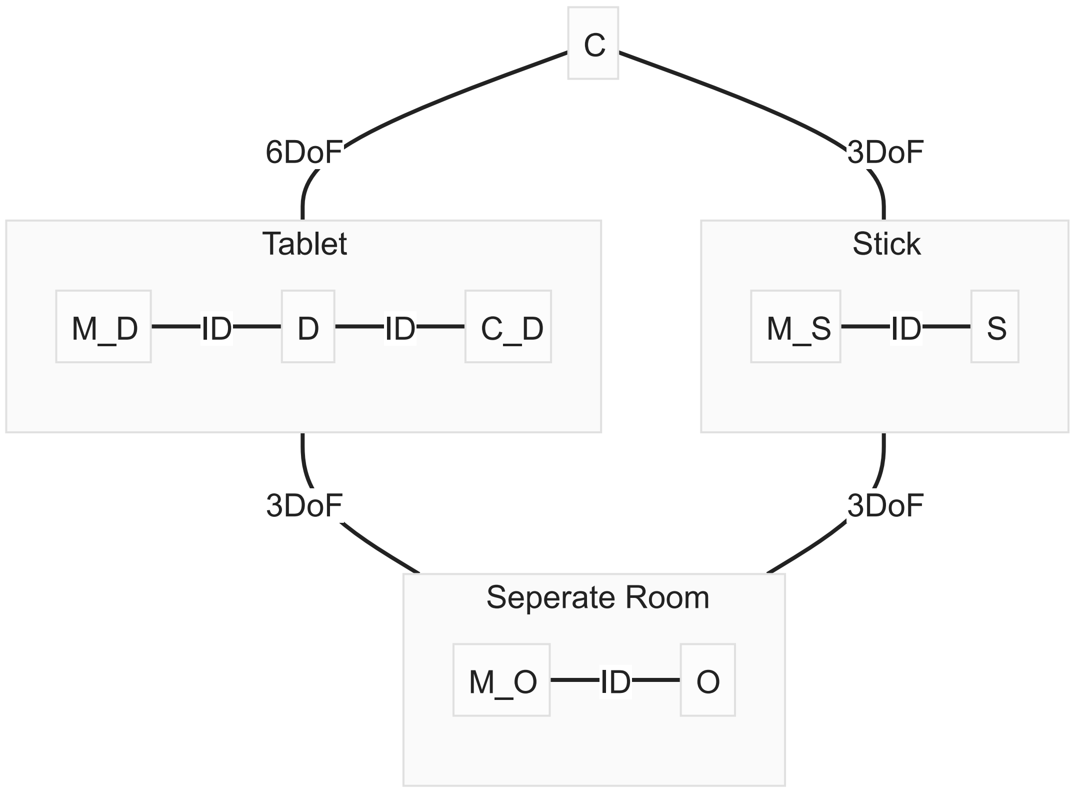
4. We have a object O in a seperate room which is moved by S and tracked via M_O



5. As stated in (1) our camera system tracks the objects in a seperate room



6. Lastly we add back the Outside-In camera C Tracking System defined in (2) again tracking our Tablet Marker M_D as well as our Stick Marker M_S (cameras track the markers directly and Stick should be directly connected to Object but for readability they are pointing at the whole subgraphs)



→ So we Track our Tablet M_D and our Stick M_S with C within the Room we are standing in so we can add augmentation
→ the Tablet shows the Object in the next room tracked via C_D that can be moved using the Stick S

Camera Calibration

#homogenous-matrix-transformations #calibration #perspective-n-point

Tsai's Algorithm

Image Formation (rotate followed by translation)

1. Map World Coordinates to Camera Coordinates
2. Project 3D-to-2D → From World Space to Screen Space
3. Account for lens distortion

Camera Calibration

1. Get Distorted Image
2. Calculate Focal Length, Distortion, ...

OpenCV based Marker-tracking

1. Projection from 3D Object to 2D Image Plane

$$\begin{pmatrix} x_{image} \\ y_{image} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_x \\ r_{20} & r_{21} & r_{22} & t_x \end{pmatrix} \cdot \begin{pmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \\ 1 \end{pmatrix}$$

2. Pose Estimation (Perspective-n-point pose estimation (PnP))
3. Detection

Markerless Optical Tracking and Feature Detection

#inside-out #outside-in #sensor

Why?

- Mobile User (Local inside-out tracker ie.: camera in users hand)
- this results in fast Movements
- generally unprepared Enviroment (no Markers, no Scene Description)

SLAM - Simultaneous Localization and Map-Building

Q: how to solve localization and map-building (3D Scene Reconstruction) simultaneously?
A: Build a map incrementally from motion and map this motion to the camera movement

PTAM Approach - Parallel Tracking and Mapping

Overview

Requirements

- Fast
- Accurate
- Robus

Difficulties

- tracking hand-held is more difficult than robots (movement speed and prediction)
- potential association errors (wrong matches)

Assumption

- mostly static scene
- small area

Main Concepts

- Seperate **Tracking** and **Mapping**
- Mapping based on key-frames
- reproject image patches to define search region

Application

1. Stereo Initialization: User sets to manual start-up images as a stereo initialization for tracker
2. Keyframe Insertion: New Keyframes are added after X Frames and Y Difference to previous Keyframes
3. Bundle Adjustment: Adjust X Keyframes (current + 4 neighbors)
4. Data Refinment: "luxury routine" (when there is time) make new measurements in old keyframes

→ Improvements possible with additional Sensors (modern IPhone Lidar Scanner for more direct Geometry Measurements)

Trackers

#tracker #sensor #outside-in #inside-out #HMD #detection #marker

Placement Strategies for Trackers

Sensors

Mobile → Wide Range and Dynamic Motion
Stationary → Precise but limited Range

Targets

Mobile → Many, Cheap and Natural
Stationary → must be set up (effort)

Devices for Time/Frequency Tracking

1. Ultrasound
   Pro: Small & Lightweight and independent of line-of sight
   Con: high variation due to enviromental factors and tethered
2. GPS
   Pro: Available world-wide, mobile
   Con: slow update rate, depends on line-of-sight with satellite and imprecise

Optical Sensors

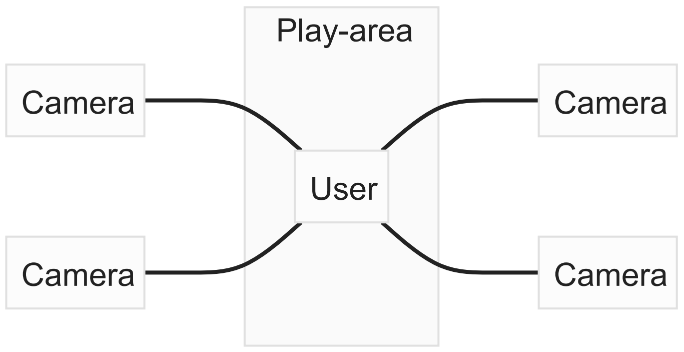
Outside-in vs. Inside-out

Placement Strategies

Sensors / Targets	Mobile	Stationary
Mobile	Inside-In	Inside-Out
Stationary	Outside-In	Outside-Out

Outside-In

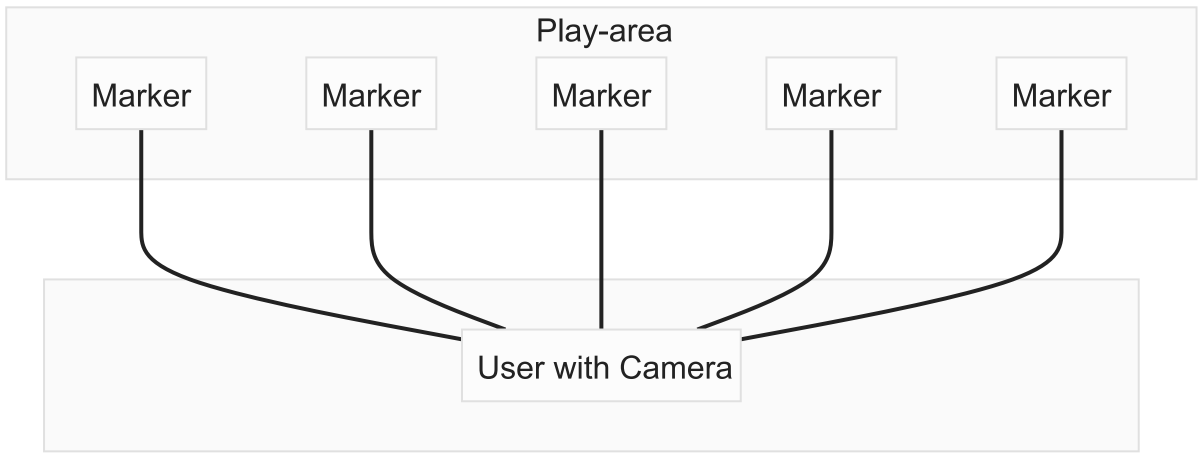
- Example: Stationary Sensor and Mobile Target (VR-HMD and fixed tracking Setup in Room)



→ Pro: Precise, Fast and Robust
→ Con: Small Range/ limited Area, Expensive, requires setup and line of sight

Inside-Out

- Example: Mobile Sensor and Stationary Target (Handheld Phone and AR-Marker)



	Marker-based	**"Marker-less" - (nat. Features)**	**SLAM / PTAM**
Pro	Cheap	Unobtrusive	Very Flexible
	Fast	Unnoticable	Unnotiable
	Precise		Relaxed Line of Sight
Con	Ugly markers	**Still** requires special markers	Untextured areas
	Line of sight !	Line of sight !	Slow
			Might drift

Inertial Sensing

Accelerometers

- measure acceleration through movement
- uses the inertia (resistance of matter to momentum) for measurement

Gyroscopes

- object-stabilized axis of rotation
- can measure rotational forces relative to rotation axis
- but is disturbed by gravity or vibration

→ mobile
→ low latency
→ minimal setup
→ low precision

Mechanical Trackers

- measure movement with mechanical arm
- mechanical arm has limited rotations
- movement can be measured at joints

→ high precision
→ limited range
→ restricts user movement

Electro-Magnetic Field Sensing

1. create electro magnetic field in play area
2. measure changes in field from metals on your body

→ requires no user calibration
→ high precision
→ fast tracking
→ but limited area and restrictions on user (no metal)

Hybrid Systems

Hybrid Systems in mobile applications should automatically connect

Classification

Approaches in these Systems should use sensors with a high **difference**, specifically:

1. Complementary (different abilities → better data from more different tracking)
2. Competitive (strong approach → take the best of different options)
3. Cooperative (abilities complement each other → no system can work alone)

Example - Nintendo Wii

- Accelerometers
- IR Camera + IR Light bar (inside-out)
- Regular Controls
  → Complementary, Competitive and Cooperative !

Sensor Fusion

#sensor #calibration #HMD

Kalman Filter

Definition

Mathematical Approach to iteratively predict the next Step within a defined System
→ Dynamic Process Modelling, for next step prediction from current recorded system

Problem

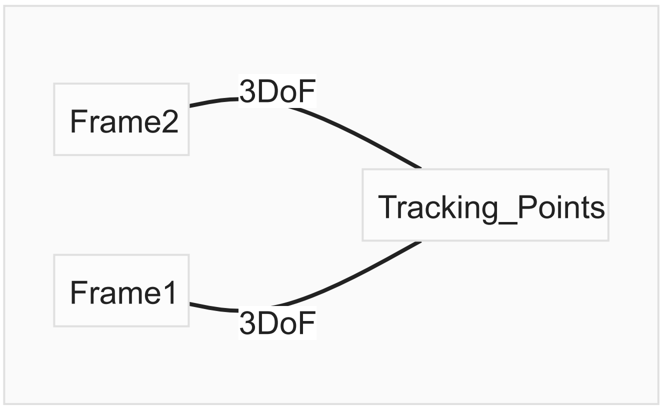
→ 6 DoF Tracking of an object (HMD - VR Headset)

1. Define Representations for Pose of Object ("Which parameters should we estimate?")
   1. Orientation
   2. Position
2. Define Motion Model (→ combined time pose update function)
3. Update Measurements in Model
4. Predict and Repeat

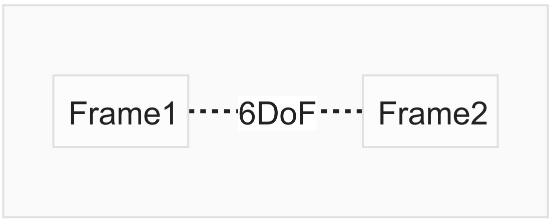
Calibration and Registration

Absolute Orientation

1. Set Up two systems that track an Object in a 2D Space (3DoF) from different perspectives
   → through the combination of data you can derive the relationship between both coordinate frames (Frame could be a fixed Camera, so 2D Image of 3D Scene)



2. We can now derive the relation between Frame1 and Frame2



Applications

- this idea can be used for 3D-3D-Pose Estimation
- have an ART Tracker that tracks a HMD as well as a model with tracking points
- we can now estimate the poses and overlay a virtual scene fittingly as we know the relation between our scene points and HMD
  → Overlay a 3D Model onto a defined tracking space

Hand-Eye Calibration

1. Have an Object with ie.: Mechanical Movement so we can **measure** its exact orientation
2. Have a Camera attached to this object that tracks an AR Marker
3. We can now derive the spacial relationship between robot and marker from different positions

→ Idea is basically to move the robot, we know the mechanical movements relation to the robot as well as the relation from the camera to the marker
→ we can derive the relations of the robot to the marker as well as the actual location in 3d space by moving the robot

Displays

#HMD

Head-Mounted Displays (HMDs)

Overview

- Ian Sutherland in 1986 creates first HMD
- since then a number of different approaches have been created
- mainly VR goggles and AR glasses

Technological Issues

- System Latency
- Discrepancy between Human FOV and small HMD FOV
- Resolution vs FOV
- High cost

Human Factors

1. **User Acceptance and Safety:** Will anyone wear this for extended Periods? Lack of Trust from Video Recording, Not seeing faces of other people, ...
2. **Depth Perception:** Possible Discrepancies between perceived location and actual location because of stereo imaging
3. **Adaptation:** Recovery after HMD use? Duration?
4. **Depth of Field:** smaller Depth of Field can lead to increased dizziness → accomodations

Other Displays for AR

Limitations of Current HMDs

1. Resolution
2. Field of View
3. Contrast
4. Weight
5. Cables
6. User Comfort
7. Shared Viewing

Options

1. Portable Displays (Tablets, Phones)

2. Stationary Monitors (Intelligent Enviroments, Wall-mounted Displays)
3. On real Objects (Virtual Showcase / Augmenting Experience with Visuals)

Display Calibration

#inside-out #outside-in #HMD #perspective-n-point

Single Point Active Alighment Method (SPAAM)

estimating P directly:

1. Given: 3D Points in a tracker target
2. Given: Corresponding 2D Points on Screen
3. Task: User Translates/Rotates Display so crosshair aligns with a known 3D Point

→ Calibration when HMD is moved
→ Calibration with new user
→ easy to implement

Display Relative Calibration (DRC)

1. Find Markers displayed inside the Display
2. Project 2D Points into 3D using known Camera Calibration

→ more advances but no better results than SPAAM
→ more complicated and elaborate setup

INDICA (Interaction-free Display Calibration)

1. Detect Eye Position automatically (using Inside-Camera)

→ no user-interaction required (no user-error)
→ continuous recalibration possible

Practical

0_Overview

Table of Contents

Matrix Vector Calculations

#homogenous-matrix-transformations

Vector Calculations

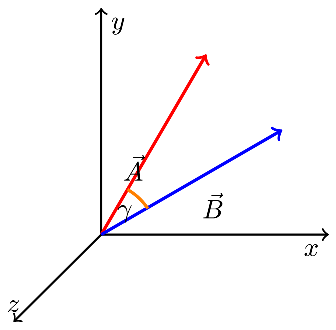
$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$$

Scalar Product

Description: calculates *sum* of dimensional products of vector

$$\begin{aligned} \vec{a} \cdot \vec{b} &= a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 \\ &= 1 \cdot 2 + 2 \cdot 1 + (-1 \cdot -1) \\ &= 5 \end{aligned}$$

Angle γ between vectors $\vec{a}$ and $\vec{b}$ can be calculated via scalar product:



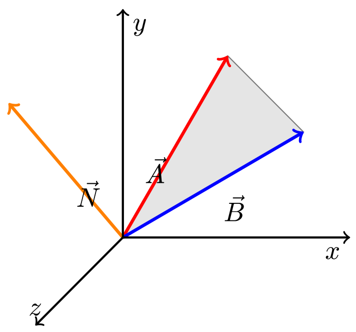
$$\begin{aligned} \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|} &= \frac{1 \cdot 2 + 2 \cdot 1 + (-1 \cdot -1)}{\sqrt{1^2 + 2^2 + -1^2} \cdot \sqrt{2^2 + 1^2 + -1^2}} \\ &= \frac{5}{\sqrt{6} \cdot \sqrt{6}} \\ &= \frac{5}{6} \\ \gamma &= \cos^{-1}(\frac{5}{6}) \approx 33.56 \end{aligned}$$

Cross-Product

$$\vec{a} = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 5 \\ 6 \\ 7 \end{pmatrix}$$

Calculate the vector $\vec{n}$ that is orthogonal to the plane defined by vectors $\vec{a}$ and $\vec{b}$

orthogonal: to be at a right angle



$$\vec{a} \times \vec{b} = \begin{pmatrix} a_2 \times b_3 - a_3 \times b_2 \\ a_3 \times b_1 - a_1 \times b_3 \\ a_1 \times b_2 - a_2 \times b_1 \end{pmatrix}$$

$$= \begin{pmatrix} 3 \times 7 - 4 \times 6 \\ 4 \times 5 - 2 \times 7 \\ 2 \times 6 - 3 \times 5 \end{pmatrix}$$

$$= \begin{pmatrix} -3 \\ 6 \\ -3 \end{pmatrix} = \vec{n}$$

normalize vector $\vec{n}$ by dividing with its length

$$\frac{\vec{n}}{\|\vec{n}\|} = \frac{\vec{n}}{3\sqrt{6}}$$

$$= \begin{pmatrix} -1\sqrt{6} \\ \sqrt{\frac{2}{3}} \\ -1\sqrt{6} \end{pmatrix}$$

Homogenous Matrices

homogenous transformations are operations that can be applied to matrices in a consistent format suitable for computation

Extending by dimension w

- for homogenous operations we want to extend our matrices and vectors by an additional dimension w that has a length of $1 = |w|$
- we do this so we can guarantee a consistent result, if w changes we know how to scale our variable back to its original size

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \implies A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vector-Matrix Multiplication

- imagine "laying" the vector on-top of your matrix and applying the first dimension of the vector to the first column of the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

$$= \begin{pmatrix} 1 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot w \\ 0 \cdot x + 2 \cdot y + 0 \cdot z + 2 \cdot w \\ 0 \cdot x + 0 \cdot y + 3 \cdot z + 3 \cdot w \\ 0 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot w \end{pmatrix}$$

$$= \begin{pmatrix} x + w \\ 2y + 2w \\ 3z + 3w \\ w \end{pmatrix}$$

Matrix-Matrix Multiplication

- simply put, cell 1 of the resulting matrix $C = A \times B$ will be the Scalar of the first **row** of A with the first **column** of B
- matrices need to match dimensions

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{pmatrix}$$

$$C = A \times B = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{11}b_{1m} + \cdots + a_{1n}b_{nm} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1m} + \cdots + a_{2n}b_{nm} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}b_{11} + \cdots + a_{nm}b_{n1} & a_{n1}b_{12} + \cdots + a_{nm}b_{n2} & \cdots & a_{n1}b_{1m} + \cdots + a_{nm}b_{nm} \end{pmatrix}$$

Translation

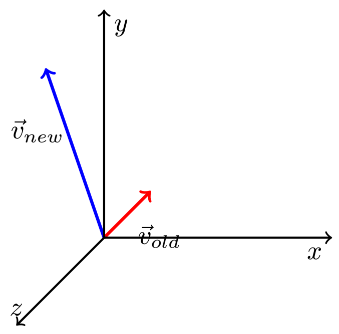
$$Translation = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying a translation T to a 3-Dimensional vector $\vec{a}$ (moving the point)

$$\vec{a} \cdot T = \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} a_x + t_x \\ a_y + t_y \\ a_z + t_z \\ 1 \end{pmatrix}$$

For a unit-vector $v_{old} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ that we translate by $\begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix}$ the result would be $v_{new} \begin{pmatrix} 0 \\ 3 \\ 2 \end{pmatrix}$:



Rotation

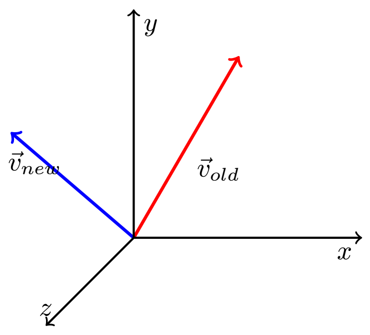
- There are different matrices for rotations around the x, y and z axis

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

$$R_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$

$$R_x(\alpha) = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For a unit-vector $v_{old} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}$ that we rotate with $\begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ the result would be $v_{new} \begin{pmatrix} -2 \\ 1 \\ -1 \end{pmatrix}$:



Scale

- Scaling operations can also just be used as stretch operations

$$Scale = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying a translation T to a 3-Dimensional vector $\vec{a}$:

$$\vec{a} \cdot T = \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} a_x \cdot S_x \\ a_y \cdot S_y \\ a_z \cdot S_z \\ 1 \end{pmatrix}$$

Combining Matrix Operations

- homogenous matrix operations can be combined to apply all of your operations at once (scale + translation + rotation)
- **order of operations** matters (!)
- the same two operations will have a different result, when applied in a different order

$$Operation = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$Operation$ can be represented by a translation and rotation (5 can be found by imagining the result of the Scalar of a second row (1,0,0,0) and last column (5,0,0,1)):

$$Operation = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

but (!) it could have also been the result of completly different operations

Point Mapping with homogenous Matrix Operations

Task: define a matrix h that maps the vector $\vec{v_{old}} = \begin{pmatrix} 5 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ to the new coordinate $\vec{v_{new}} = \begin{pmatrix} 0 \\ 10 \\ 0 \\ 1 \end{pmatrix}$

$$v_{new} = \begin{pmatrix}5\\0\\0\\1\end{pmatrix} \cdot \begin{pmatrix}0 & -1 & 0 & 0\\1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix} \times \begin{pmatrix}1 & 0 & 0 & 5\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix}$$

$$= \begin{pmatrix}5\\0\\0\\1\end{pmatrix} \cdot \begin{pmatrix}0 & -1 & 0 & 0\\1 & 0 & 0 & 5\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix}$$

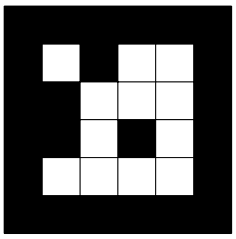
$$= \begin{pmatrix}0\\10\\0\\1\end{pmatrix}$$

Augmented Reality Markers

#detection #marker

Marker Identity

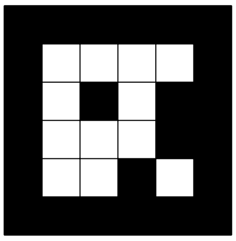
Given the following AR-Marker, how can we calculate its ID?



$$\begin{bmatrix} 0\cdot 8+1\cdot 4+0\cdot 2+0\cdot 1 \\ 1\cdot 8+0\cdot 4+0\cdot 2+0\cdot 1 \\ 1\cdot 8+0\cdot 4+1\cdot 2+0\cdot 1 \\ 0\cdot 8+0\cdot 4+0\cdot 2+0\cdot 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \\ a \\ 0 \end{bmatrix}$$

so the ID to our Marker is $0x48a0$

Consider the following rotation, that is still the same marker but would result in a different ID:



$$\begin{bmatrix} 0\cdot 8+0\cdot 4+0\cdot 2+0\cdot 1 \\ 0\cdot 8+1\cdot 4+0\cdot 2+1\cdot 1 \\ 0\cdot 8+0\cdot 4+0\cdot 2+1\cdot 1 \\ 0\cdot 8+0\cdot 4+1\cdot 2+0\cdot 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 5 \\ 1 \\ 2 \end{bmatrix}$$

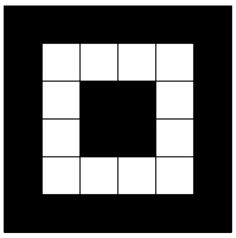
so the smallest ID for our Marker is actually $0x512$

Lessons

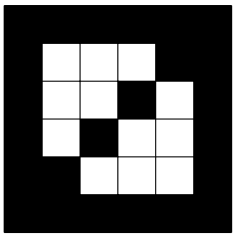
1. Markers Rotation should be unique because of **perspective transformation**
2. Sometimes Markers need to be rotated to get the smallest (actual) ID
3. smallest ID is always the smalles number you calculate when considering all rotations

Examples

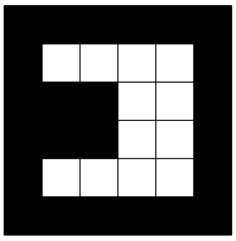
1. ❌ All Marker Rotations are the same → Issues with perspective transformation



2. ✗ Some Marker Rotations are unique, **but** not all → Issues with perspective transformation



3. ✓ All Marker Rotations are unique → (but currently not rotated for smallest ID)



Padding and Filters

Padding

- padding is applied to images so certain algorithms do not fail at corner/edge cases
- **Zero-Padding** is simply adding rows and columns of 0 around our image

$$Image = \begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix}$$

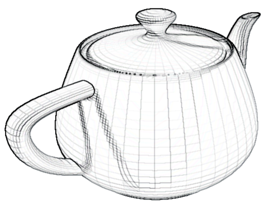
$$Image + Padding = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 \\ 0 & 10 & 10 & 10 & 0 \\ 0 & 10 & 10 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Sobel-Filter

- filter is applied to images for edge detection
- it finds the "most abrupt change" of contrast within an image
- this happens via a Kernel that is applied to regions of the image

Test Image:

With applied *Sobel-Filter*:



Sobel Calculation

- Sobel Filter is applied to an image by applying a vertical or horizontal kernel to regions of the image

$$Image = \begin{pmatrix} 0 & 0 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{pmatrix}$$

$$Kernel = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

- Kernel is applied and we calcualte a new value for our center pixel x_5:

$$Application = \left\| \begin{pmatrix} 1 \cdot 0 & 0 \cdot 0 & -1 \cdot 0 \\ 2 \cdot 10 & 0 \cdot 10 & -2 \cdot 0 \\ 1 \cdot 10 & 0 \cdot 10 & -1 \cdot 0 \end{pmatrix} \right\|$$

$$\begin{aligned} &= 1 \cdot 0 + 0 \cdot 0 + \ldots - 1 \cdot 0 \\ x_5 &= 30 \end{aligned}$$

$$Image_{new} = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & 30 & x_6 \\ x_7 & x_8 & x_8 \end{pmatrix}$$

OpenCV

#perspective-n-point #detection

Image Color Conversion

- first we always want to convert and image to grayscale before applying OpenCV / Contrasting functions for easier calculations

```
Mat OutputImage;
InputImage = frame.clone();
cvtColor(InputImage, OutputImage, COLOR_BGR2GRAY);
```

Threshholding

- defines a cutoff value and converts all pixels below/above that threshhold to black/white respectively

Threshhold

- uses a global value as a cutoff for all pixels

Advantages	Disadvantages
Simple	Errors with light variations
Fast	Noisy Backgrounds can be an issue
Easy to calculate	
→ Suitable for Images with consistent lighting	→ Unsuitable for videos or incosistent lighting

Adaptive Threshhold

- calculates the threshhold for each pixel based on a small region around it

Advantages	Disadvantages
works with non-uniform illumination/shadows	more expensive to compute
more flexibility in threshhold	can still fail in extreme noise / lighting variations
different threshholding methods	threshholding results can depend on neighbourhood size
→ Suitable for most Images and Video	→ more complex variations / options

```
using namespace cv;

while (cap.read(frame)) {
        // --- Process Frame ---
        Mat grayScale;
        imgFiltered = frame.clone();
        cvtColor(imgFiltered, grayScale, COLOR_BGR2GRAY);

        // Threshold to reduce the noise
        if (slider_value == 0) {
                adaptiveThreshold(grayScale, grayScale, 255, ADAPTIVE_THRESH_MEAN_C,
THRESH_BINARY, 33, 5);
        }
        else {
                threshold(grayScale, grayScale, slider_value, 255, THRESH_BINARY);
        }
}
```

Contours

findContours

- returns a list of vectors that are the detected contours in the image

approxPolyDP

- simplifies a curve of many points

```
contour_vector_t contours;

// RETR_LIST is a list of all found contour, SIMPLE is to just save the begin and
ending of each edge which belongs to the contour
findContours(grayScale, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);

//drawContours(imgFiltered, contours, -1, Scalar(0, 255, 0), 4);

// size is always positive, so unsigned int -> size_t; if you have not initialized the
vector it is -1, hence crash
for (size_t k = 0; k < contours.size(); k++) {
        // --- Process Contour ---
        contour_t approx_contour;

        // Simplifying of the contour with the Ramer-Douglas-Peuker Algorithm
        // true -> Only closed contours
        // Approximation of old curve, the difference (epsilon) should not be bigger
than: perimeter(->arcLength)*0.02
        approxPolyDP(contours[k], approx_contour, arcLength(contours[k], true) * 0.02,
true);
}
```

Rectangles from Contours

- every contour that can be simplified into 4 lines must be a rectangle → we are looking for rectangular markers

```
// 4 Corners -> We Color them as we have a rectangle here
if (approx_contour.size() == 4) {
        colour = QUADRILATERAL_COLOR;
}
else {
        continue;
}

// 1 -> 1 contour, we have a closed contour, true -> closed, 4 -> thickness
polylines(imgFiltered, approx_contour, true, colour, THICKNESS_VALUE);
```

Edge Location using Sobel

- previous results are not yet accurate enough
- apply Sobel-Filter around the edges for more accurate results

Line Fitting

- currently marker edges are represented in 6 points
- we want to fit a line through these points to get exact corner coordinates for the detected rectangles

```
// We now have the array of exact edge centers stored in "points", every row has two
values / 2 channels!

Mat highIntensityPoints(Size(1, 6), CV_32FC2, edgePointCenters);

// fitLine stores the calculated line in lineParams per column in the following way:
// vec.x, vec.y, point.x, point.y
// Norm 2, 0 and 0.01 -> Optimal parameters

// i -> Edge points
fitLine(highIntensityPoints, lineParamsMat.col(i), DIST_L2, 0, 0.01, 0.01);

// We have to jump through the 4x4 matrix, meaning the next value for the wanted line
is in the next row -> +4
// d = -50 is the scalar -> Length of the line, g: Point + d*Vector

// p1<----Middle---->p2
//   <-----100----->

// We need two points to draw the line
Point p1;

p1.x = (int)lineParams[8 + i] - (int)(50.0 * lineParams[i]);
```

```
p1.y = (int)lineParams[12 + i] - (int)(50.0 * lineParams[4 + i]);

Point p2;

p2.x = (int)lineParams[8 + i] + (int)(50.0 * lineParams[i]);
p2.y = (int)lineParams[12 + i] + (int)(50.0 * lineParams[4 + i]);

// Draw line
line(imgFiltered, p1, p2, CV_RGB(0, 255, 255), 1, 8, 0);
```

Perspective Transform of Rectangles

getPerspectiveTransform

- gets two sets of four 2D points and returns a homogenous Matrix to warp the second polygon into the shape of the first

warpPerspective

- is used to warp one marker image fittingly by the result of getPerspectiveTransform

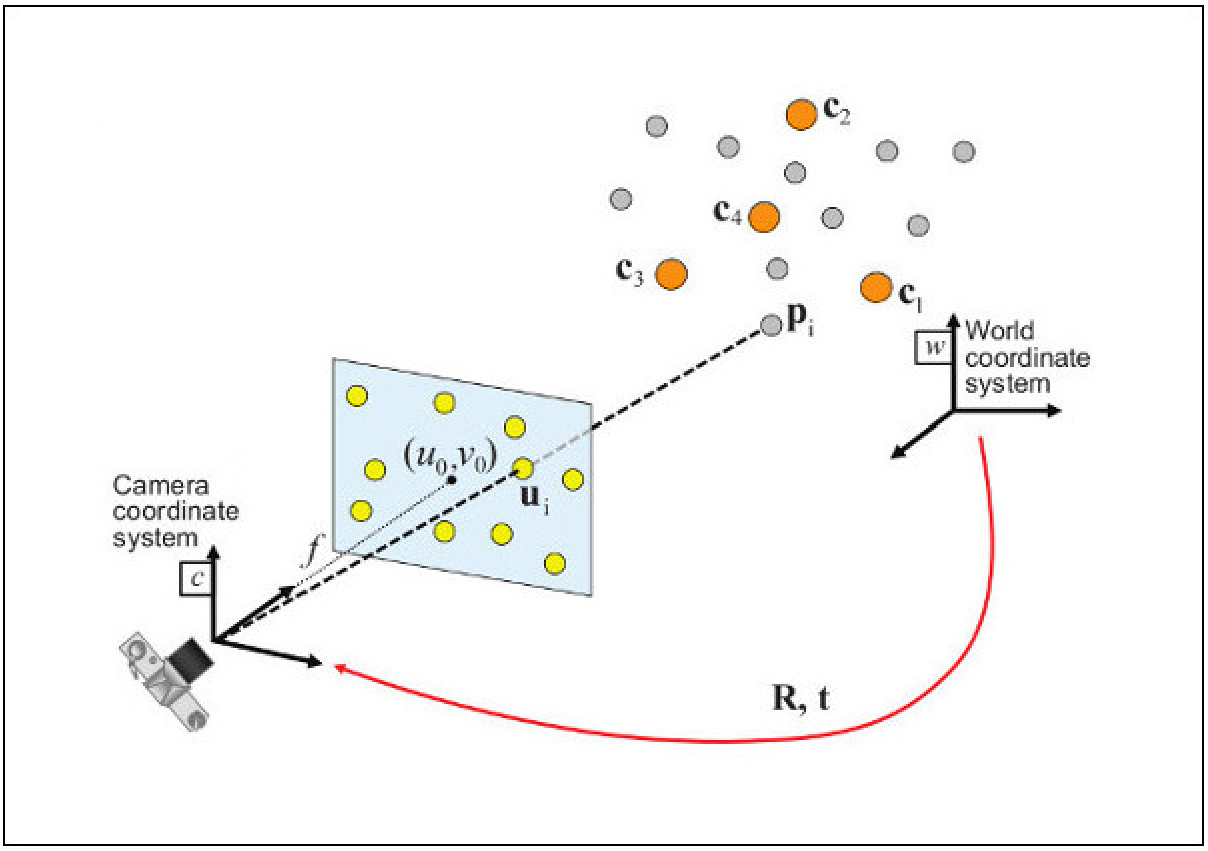
→ we use (−0.5, −0.5)(−0.5, 5.5)(5.5, 5.5)(5.5, −0.5) as 2D points for the first rectangle to get exactly a 6x6 marker rectangle
→ we only consider markers with a complete black border, to filter out potential false positives

```
// Coordinates on the original marker images to go to the actual center of the first
pixel -> 6x6
Point2f targetCorners[4];
targetCorners[0].x = -0.5; targetCorners[0].y = -0.5;
targetCorners[1].x = 5.5; targetCorners[1].y = -0.5;
targetCorners[2].x = 5.5; targetCorners[2].y = 5.5;
targetCorners[3].x = -0.5; targetCorners[3].y = 5.5;

// Create and calculate the matrix of perspective transform -> non affin -> parallel
stays not parallel
// Homography is a matrix to describe the transformation from an image region to the
2D projected image
Mat homographyMatrix(Size(3, 3), CV_32FC1);

// Corner which we calculated and our target Mat, find the transformation
homographyMatrix = getPerspectiveTransform(corners, targetCorners);

// Create image for the marker
Mat imageMarker(Size(6, 6), CV_8UC1);

// Change the perspective in the marker image using the previously calculated
Homography Matrix
// In the Homography Matrix there is also the position in the image saved
warpPerspective(grayScale, imageMarker, homographyMatrix, Size(6, 6));
```

Perspective-n-point pose computation (PnP)

Problem Points (ie.: Markers) from the real world are projected onto an image plane using the camera

Question How do we estimate their real-world positions?
Answer Find/Estimate a transformation that maps these 3D Points c_n onto our image plane u_i (Source OpenCV docs)



→ there are different algorithmic approaches to solve PnP Problems in OpenCV

```
Mat raux, taux;

//modelMarkerCorners represents the 3D coordinates of the corners of the markers
//imgMarkerCorners represents the projected 2D coordinates of these markers in our
image plane

//_Kmat is the intrinsic camera matrix, representing the internal parameters

//distCoeffs is the distortion coefficient of the camera

//raux is the output parameter and represents the rotation vector of the marker pose
//taux is the output parameter and represents the translation vector of the marker

//false says we only use raux and taux as ouput and not as additional estimation
parameters

solvePnP(modelMarkerCorners, imgMarkerCorners, _Kmat, distCoeffs, raux, taux, false);
```

Aruco Marker Detection

- How would you build a simply Marker Tracker Application? use the Pipeline above or use Aruco functions which ship with OpenCV

1. Transform Image to Gray
2. Use Aruco generated Markers for detection (they are already rotation safe !)
3. aruco.detectMarkers returns an array of marker corners and IDs from a frame
4. aruco.drawDetectedMarkers can directly draw these into the frame
5. evaluate your markers further with custom functions and logic

```
def evaluate_frame(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_250)
    parameters =  aruco.DetectorParameters_create()

    corners, ids, rejectedImgPoints = aruco.detectMarkers(gray, aruco_dict,
parameters=parameters)

    frame_markers = aruco.drawDetectedMarkers(frame.copy(), corners, ids)

    matches = find_matching_rectangles(rectangles=corners, identifier=ids,
radius_multiplier=1.5)

    return matches, corners, ids, frame_markers
```

OpenGL

#Lighting #Phong

General

- OpenGL is a state Machine, you set something *once* and it *stays* until you change it
  → glColor4f() until you set a new Color
  → glTranslatef() until you change the current matrix
- OpenGL works with a stack, you always have a current matrix you are working with and a stack which you can push/pop
  → OpenGL actually has 3 Stacks (there can be an additional Color Stack but it is not default)
- Operations are applied to the current Matrix

Stack / Matrix Modes

```
void glMatrixMode(GLenum mode);

GLenum mode {
    GL_MODELVIEW, //Applies subsequent matrix operations to the modelview matrix
stack.
        GL_PROJECTION, //Applies subsequent matrix operations to the projection matrix
stack.
        GL_TEXTURE, //Applies subsequent matrix operations to the texture matrix
stack.
        GL_COLOR //Applies subsequent matrix operations to the color matrix stack.
};
```

- Operations are applied to the **top Matrix** of the current stack
- Initially stack is empty and current matrix is the identity matrix
- Applying transformations update the current Matrix consecutively

```
glPushMatrix(); //Duplicates current Matrix and pushes stack down by 1
glPopMatrix(); //Pops current Matrix Stack, replacing current with one below

glLoadIdentity(); //replaces current matrix with identity Matrix
glLoadMatrix(const GLfloat * m); //replaces current matrix with the specified matrix
```

Functions

```
glTranslatef(float x, float y, float z);
        // specify the x y z component of a translation vector

glRotatef(float angle, float x, float y, float z);
        // specifies a rotation of angle degrees around the vector xyz
```

```
glScalef(float x, float y, float z);
        // specify scale factors along the x, y, and z axes, respectively.

glColor4f(float r, float g, float b, float alpha)
        //specify new red, green, blue and alpha values for the current color
```

Scope of OpenGL Calls

- Draw calls use the **current** matrix for their drawing → this means any transformations have to be specified beforehand
- Draw calls do **not** reset the current matrix (!)
- glColor is special and stays until you specify a new color
- Transformations are relevant until the current Matrix or Matrix mode is changed

Lighting / Phong-Rendering

- OpenGL considers lighting to be divided into four independent components: emissive, ambient, diffuse and specular
  → All four components are computed independently and then added together to render a scene with the **Phong-Lighting** Model (Model = Ambient + Diffuse + Specular)

Emissive

- this is the special case of OpenGL lighting, if a material has an *emissive* color it simulates light originating from this object
- this is unaffected by any lightsources
- **but** it also does not add light to a scene

Ambient

- this is the general scattered light level of a scene, it typically results in an image without any shading, just flat colours

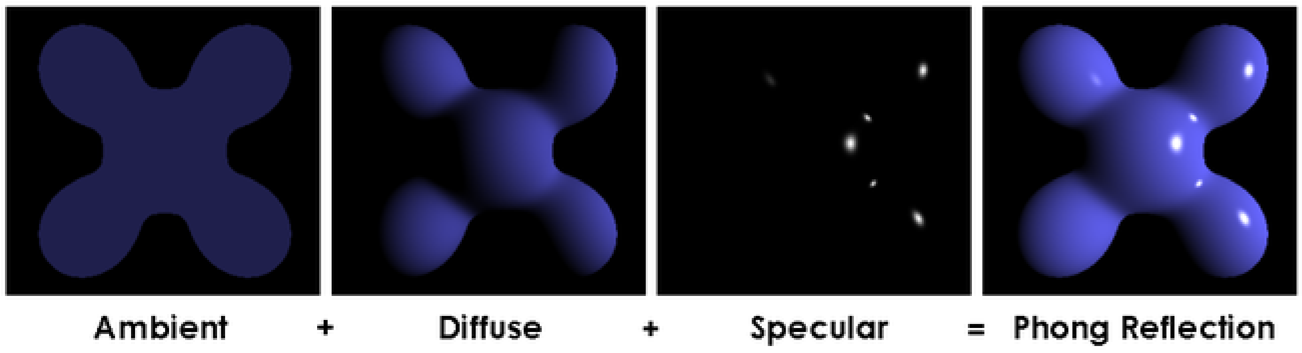
Diffuse

- the effect of directed light sources that scatter through the creases and crevices of our object
  → this is typically what you would see as adding shadows to your scene

Specular

- The Highlights of shiny material, where the roughness is low and bounces of the surface
  → highlights are the shiny parts c:

→ combining our lighting model results in a wonderfully lit object (Source Wikimedia)


Ambient
+
Diffuse
+
Specular
=
Phong Reflection