Ausgangslage:

Für eine erfolgreiche Anwendung vom PresentationWriter sind folgende Geräte nötig:

* Beamer, der auf eine Leinwand projiziert, die keine Unebenheiten hat
* Webcam:
  + Die das ganze projizierte Bild erkennt.
  + Mindestens 24 fps(Frames pro Sekunde) liefert
  + Die Projektion auf mindestens 300 x 200 Pixeln abgebildet wird
* Eine CPU mit mindestens 2 Cores mit mindestens 2 GHz.

Diese Kriterien sind für eine optimale Funktionalität nötig. Falls diese nicht erfüllt sind, ist ein Funktionieren möglich, kann aber nicht gewährleistet werden.

Technologien

Wir haben uns für die Programmiersprache C# entschieden, da uns diese besser liegt. Zudem war es einfacher auf Systemfunktionen wie z.B. Direct Show und Mausemulation zuzugreifen. In Java wäre das nicht möglich gewesen. Mit C++ hätten wir bei der Bildbearbeitung etwas weniger Performance benötigt, dafür hätten wir mehr Einarbeitungszeit benötigt. Aus unserer Sicht haben wir uns richtig entschieden.

Beim grafischen Framework haben wir uns für AForge entschieden, da es nativ in C# geschrieben ist und gut dokumentiert ist. Im Gegensatz dazu hätten wir mit OpenCV etwas besser optimierte Algorithmen gehabt.

Da wir nur einen Prototypen produzieren, ist Performance für uns nicht besonders wichtig, was unsere Wahl ebenfalls beeinflusst hat.

Architektur

Grundüberlegungen:

Unser Hauptprojekt wird eine ausführbare Datei, die alle unsere Subprojekte zusammenbringt.

Grobablauf unseres Programms:

//SSD

1. Verbindung mit der Kamera
   1. Der Nutzer kriegt eine Auswahl aller gefundenen Kameras und entscheidet sich für eine.
   2. Anhand der Auswahl wird die Kamera angelegt und gestartet.
2. Das Programm sucht einen passenden Visualisierer und legt diesen an.
3. Ein DataParser wird angelegt.
4. Die Kalibration kann nun automatisch oder nach einer Interaktion des Benutzers gestartet werden.
5. Die Details über den Ablauf des Kalibrationsvorgang sind im Abschnitt Kalibration aufgeführt.
6. Kalibration wird erfolgreich abgeschlossen und das Pentracking gestartet.
7. Die Details über den Ablauf des Trackings sind im Abschnitt Pen-Tracking aufgeführt.

Visualisierung

Ein erstes Problem Bot sich uns, als wir versuchten, aus einer Library auf den Bildschirm zu zeichnen. Das wiederspricht eigentlich allen Architekturmodellen, jedoch ist es bei unserer Lösung nötig für die Kalibration.

Probleme:

* Wir wissen nicht, in welcher Umgebung (Anzahl Treads) unsere Library läuft. Gewisse Kameras erzeugen zudem verschiede Threads in denen die Frames übergeben werden.
* Jedes GUI muss in einem eigenen Thread laufen das ein spezielles ThreadSTA-Attribut hat.
* Der Garbage-Collector räumt den GUI-Thread ab, selbst wenn er in Zukunft noch verwendet werden müsste.
* Wenn ein GUI gestartet wird, öffnet es sich sofort. Das ist in unseren Fall nicht erwünscht. Ein Hide, das gerade nach dem Start aufgerufen wird, hat keine Wirkung. Es muss erst der komplette Start und die Darstellung des GUIs(ca. 10ms) abgewartet werden, bis es versteckt werden kann.

Umsetzungen:

WPF. Sämtliche Interaktionen müssen über den Dispatcher geschehen. Andernfalls wird sofort eine Exception geworfen. Dadurch kann die Concurrency vom aufrufenden Code wieder Sequenziell abgearbeitet werden, wie es für ein GUI nötig ist. Eine Lösung zu entwickeln, die komplett Concurrency-Safe ist ziemlich aufwändig und Zeitintensiv.

WinForms: Mit dem AllowUnsafeThreadExecution Attribut kann die ganze Thread-security abgeschaltet werden. Dafür müssen alle Grafikelemente vor dem bearbeiten gelockt werden und der aufrufende Code ist verantwortlich dafür, dass die Aufrufreihenfolge und der Ablauf stimmt. Durch Double-Buffering kann die Performance erhöht werden, wenn mehrere Objekte gezeichnet werden. Es ist uns klar, dass diese Umsetzung diverse Architekturprinzipien und Guidelines verletzt. In einem finalen Release darf diese Umsetzung nicht verwendet werden.

Kalibration

Überlegungen

Die Grundüberlegung jedes Ansatzes der Kalibration ist derselbe: Im ersten Schritt werden die Ecken des Bildschirms gefunden, in den weiteren Schritten werden zusätzliche Daten gesammelt um die Interpolation zu verbessern.

Erste Erkenntnisse

1. Als erster Schritt ist es nötig, dass ein Frame einzulesen, um die Auflösung der Kamera zu erhalten.
2. Das erhaltene Bild ist weder Flächen- noch Winkeltreu.
3. Die Belichtungskorrektur der Kamera ist sehr gefährlich, um ein sinnvolles Differenzbild zu erhalten, muss darauf geachtet werden, dass auf beiden Bildern die gleiche Helligkeitsverteilung herrscht. Dies Verunmöglicht, ein universelles Bild zu speichern, mit dem man den Hintergrund entfernen könnte.
4. Die Belichtungskorrektur benötigt ca. 1-2 Frames, bis sie reagiert. Aus Performance-Gründen lässt sich das leider nicht zu unserem Vorteil nutzen.
5. Von gewissen Webcams wird das Bild gespiegelt zurückgegeben, von anderen nicht.
6. Bis die Bilder der Kamera dem entsprechen, was man erwarten würde, muss man ca. 100ms warten. Diese Dauer ist zudem vom gewählten Visualisierer abhängig.
7. Die Linse der Kamera kann für eine zusätzliche Verzerrung sorgen.
8. Berechnungen mit C# Bitmaps sind extrem inperformant.
9. Eine sequentielle Bearbeitung der Daten führt dazu, dass man mit veralteten (und somit falschen) Daten arbeitet.
10. Eine parallele Bearbeitung der führt meistens zum unkontrollierten Spawnen von Tasks, die das System verlangsamen und schliesslich ein Ablaufen verunmöglichen. Falls das Auftritt, hat man ebenfalls Probleme mit alten Daten.
11. Wenn ein Teil der Anzeige durch ein anderes Fenster oder einem Objekt verdeckt wird, ist das unmöglich automatisch zu detektieren. Daher muss die Erkennung noch durch den Nutzer verifiziert werden.

Allgemeine Massnahmen:

* Der Ablauf wird von einer Semaphore geschützt. Wenn kein Lock akquiriert werden kann, wird das ankommende Bild verworfen und nicht behandelt.
* Nach jedem Zeichnen wird 100 ms gewartet bevor die Semaphore freigegeben wird, damit sichergegangen werden kann, dass das erwartete Bild von der Kamera ankommt.

Umsetzungsversuche

1. Selber programmierte Eckdetecktion mit Differenzbildern und zufällig verteilten Rechtecken.
   1. Ablauf:
      1. Der Bildschirm wird Schwarz gemacht. Das entsprechende Bild wird gespeichert.
      2. Der Bildschirm wird weiss gemacht und das Differenzbild gemacht. Mit 4 Scanlinien werden von jeder Ecke ausgehend unter 45° ein Punkt gesucht, dessen Nachbarn in Richtung der gegenüberliegenden Ecke Liegen eine bestimmte Deckung besitzen. Es wird ein Quadrat mit Seitenlänge 3-5 Pixel berücksichtigt.
      3. Es werden 3 Rechtecke mit zufälligen Eckpunkten gezeichnet und die Eckpunkte gespeichert, eins pro Farbkanal. Auf dem Differenzbild zum Ersten werden mit dem geleichen Verfahren zu jedem Farbkanal die Ecken gesucht.
   2. Erfolge: Ecken des Bildschirms wurden ungefähr erkannt, Von den Dargestellten Rechtecken auch meist.
   3. Nachteile:
      1. Ich habe nur wenig Verbesserungspotenzial bei dieser Lösung gesehen, und dieses wäre teuer zu erkaufen gewesen.
      2. Da grosse Teile des Bildschirms umgefärbt wurden, gab es Probleme mit der Belichtungskorrektur.
      3. Die Farbkanäle waren schwierig sauber zu trennen und haben häufig Probleme verursacht. Überschneidungen waren sehr Problematisch.
2. Verwendung des AForge-Blob-Detctors:

Diese wird in den kommenden Lösungen immer eingesetzt. Ich werde bei jeder Lösung nur auf geänderte und wichtige Aspekte erwähnen. Funktionierende Teile wurden jeweils übernommen. Die verschiedenen Kalibratoren die Im Code existieren sind keine 1 zu 1 Umsetzungen der hier beschriebenen Ansätze, sondern Kombinationen davon. Der Code von verworfenen Umsetzungen wurde in anderen Umsetzungen teilweise Übernommen und aktualisiert. Diese Aktualisierungen wurden nicht Nachgepflegt aus dem Grund, dass ich diese Lösungen nicht als sinnvoll für ein Weiterverfolgen erachte.

1. Einfacher Dizzerenzbild-Kalibrator
   1. Als erstes Bild wird der Bildschirm schwarz gefärbt. Dieses Bild dient als Grundbild für alle folgenden Differenzbilder
   2. Der Bildschirm wird komplett weiss gefärbt. Auf dem Differenzbild wird der grösste Blob gesucht. Das sind dann die Ecken des Bildschirms.
   3. Es werden Grüne und blaue Quadrate jeweils versetzt auf den Bildschirm projiziert, die bei jedem Frame leicht verschoben werden um mehr Referenzpunkte zu bekommen. Die Erkennung des roten Farbkanals ist am schlechtesten, deshalb verwenden wir blau und grün. Zwei Farben sind ausreichend, wenn sie mit einem Feld Abstand projiziert werden, dass es keine Nachbarschaftskonflikte gibt. *Bild* Diese werden auf dem Differenzbild gesucht. Deren Zuordnung wird in einem Folgenden Abschnitt auf Seite ??? behandelt.
   4. Probleme: Durch die Belichtungskorrektur der Kamera wird ein universeller Differenzbild Hintergrund verunmöglicht. Es gibt zu viele Fehlerkennungen auf dem Bild.
2. Histogramm-analyse
   1. Hierbei werden keine Differenzbilder gebildet, sondern Helligkeitswerte des Bildes analysiert und so können überdurchschnittlich helle Flächen (Projektionen) gefunden werden. Die restlichen Bildinformationen werden entfernt um eine Blob-Erkennung zu ermöglichen.
   2. Durch unterschiedliche Lichtverhältnisse in Bildbereichen ist dieser Ansatz anfällig auf Störungen. Die Lösung für dieses Problem ist ein aufteilen des Bildes. Dabei wird das Bild in verschiedene Regionen aufgeteilt, wo die Helligkeitswerte weniger stark verteilt sind. Anschliesend werden die Flächen wieder zu einem Bild zusammengesetzt.
   3. Ablauf:
      1. Es wird eine weisse Fläche projiziert. Das Bild wird in 2x2 Teile aufgeteilt und der grösste Blob gesucht. Das führt zu den Eckepunkten.
      2. Es werden die grünen und blauen Quadrate aus dem Vorherigen Verfahren projiziert. Das Bild wird in 4 x 4 Teile aufgeteilt und der blaue oder grüne Farbkanal separat analysiert. In den zwei resultierenden Bildern werden Blobs erkannt und analysiert.
   4. Erkenntnisse:
      1. Die Erkennung der Blobs klappt relativ gut, jedoch sind die Kanten relativ unscharf was für eine genaue Erkennung ein Nachteil ist.
      2. Die automatische Belichtungskorrektur ist kein Problem mehr.
      3. An den Kanten bei denen das Bild aufgeteilt wurde entstehen häufig Einschnitte oder sonstige Ungenauigkeiten. Diese sind keine grossen Probleme für den Blob-Detector, jedoch sind sie eine weitere Quelle für Ungenauigkeiten.
3. Schachbrett-Muster:
   1. Um Differenzbilder nutzen zu können, aber sich nicht durch die Belichtungskorrektur beeinflussen zu lassen, muss darauf geachtet werden, dass jedes projizierte Bild eine identische Helligkeitsverteilung zum Referenzbild besitzt. Das bedeutet, dass für jedes Bild, das man analysieren möchte, zwei Bilder Projiziert werden müssen. Da diese Art der Erkennung viel genauer ist, nehmen wir diesen Overhead in Kauf und verzichten allenfalls auf ein paar Referenzbilder. „Lieber wenig gute Daten, als viel schlechte“
   2. Ablauf:
      1. Es wird ein Schachbrett mit n x m Quadraten projiziert. Das Verhältnis von n zu m sollte dem Bildschirmverhältnis entsprechen und die Quadrate sollten noch gut auf dem Kamerabild sichtbar sein (15 – 30 Pixel Seitenlänge)
      2. Das Schachbrett wird erneut projiziert, diesmal mit invertierten Farben. Auf dem Differenzbild sollte jetzt die Leinwand klar zu erkennen sein. Da es an den Kanten der Quadrate Lücken haben wird(diejenigen Pixelreihen, die von der Kamera auf beiden Bildern als Beleuchtet erkannt hat), müssen wir diese z.B. mit einem Gaussfilter mit einer kleinen Kernel-Size eliminieren, da es sonst bei der Bloberkennung zu Fehlerkennungen kommt. Jetzt können die Ecken des Bildschirms sehr genau erkannt werden.
      3. Anschliessend werden jeweils zwei Bilder projiziert, deren Differenz genau das bekannte Blau-Grüne Muster ergibt. *Bild* So können die Recktecke auf dem Bild ganz scharf und klar erkannt werden und identifiziert werden. Durch das Bilden eines weiteren Differenzbildes mit den beiden erwünschten Farbkanälen kann das meiste Bildrauschen und allfällige Unsicherheiten entfernt werden.

Mapping der Quadrate:

Vom AForge BlobCounter erhalten wir genau alle Blobs die auf dem Kamerabild gefunden wurden. Diese Daten beinhalten: Schwerpunkt, Umfassungsrechteck, Fläche, Breite, Höhe. Eine Zuordnung zu machen, welcher Blob zu welchen pojizietem Quadrat gehört. Durch eine Filterung mit einer vordefinierten minimalen und maximalen Höhe/Breite können Da wir die Ecken des Bildschirms mit Sicherheit kennen, können wir anhand des konstruierten Vierecks bestimmen ob ein Blob innerhalb der Projektion liegt und andernfalls entfernen. Somit können wir jetzt 99% der Fehlerkennungen eliminieren.

Die Schwerpunkte der Blobs sind jedoch mit Koordinaten des Kamerabildes beschrieben, was es für uns schwierig macht, deren Position auf dem Bildschirm zu bestimmen, da diese relativ komplex voneinander abhängen. So kann z.B. die kleinste X-Koordinate zuunterst liegen und die grösste Y-Koordinate rechts oben. Je nach Verzerrung ist es so schon sehr schwierig nur schon die obere linke Ecke zu finden.

Mögliche Ansätze:

1. Rekursiv:
   1. Hierbei nehmen wir an, dass die Nachbarn eines Blobs eindeutig bestimmt werden können. Dies ist möglich anhand der Distanz.
   2. Ablauf:
      1. Der Algorithmus bestimmt auf jedem Farbkanal denjenigen Blob, der am nächsten bei der oberen linken Ecke liegt. Durch die Positionsangabe 0|0 für grün oder 1|1 ist bekannt, dass es sich um einen Eckpunkt handeln muss. Der Punkt wird markiert, dass er nicht nochmals bearbeitet werden muss.
      2. Es werden also die zwei nächsten Nachbarn gesucht. Anhand der Differenz der der Koordinaten kann bestimmt werden, ob der Punkt oberhalb (Y-Differenz ist grösser als die Differenz der X-Achse und negativ), rechts (X-Differenz ist grösser und positiv), unten oder links liegt.
      3. Die Funktion berechnet die Koordinaten die der gefundene Blob haben muss (zwei in die entsprechende Achse verschoben) und startet einen Rekursiven Aufruf auf jedem unmarkierten Nachbar, nachdem der Ausgangspunkt als erledigt markiert wurde.
      4. Der rekursive Aufruf prüft zudem, ob die übergebenen Koordinaten plausibel erscheinen und bestimmt anhand deren, wie viele Nachbarn zu erwarten sind.
   3. Dieser Algorithmus funktioniert perfekt unter synthetischen Umständen, d.h. mit generierten Bildern.
   4. Aufgrund des iterativen Vorgehens sollte dieser Algorithmus in der Lage sein, um Bereiche, in denen keine Blobs gefunden wurden, herumzugehen.
   5. Erkenntnisse:
      1. Wenn ein Quadrat nicht erkannt wurde, wird meist die Diagonale als nächster Nachbar gefunden, das für die folgende Erkennung schwerwiegende Folgen hat.
      2. Häufig werden nicht alle Blobs erreicht. Somit werden nicht alle Blobs zugeordnet und Informationen verloren.
      3. Falsch zugeordnete Blobs können unmöglich identifiziert werden. Eine solche hat jedoch für die nachfolgende Interpolation schwerwiegende Folgen indem sie in einem Bildbereich das mapping komplett zerstört.
   6. Verbesserungen:
      1. Es wird nicht nur eine erwartete Position gespeichert, sondern mehrere.
      2. Die Position wird nicht bei der Iteration gespeichert, sondern jedes Mal, wenn der Blob als Nachbar gefunden wurde.
      3. Es wird nicht nur in der linken oberen Ecke gestartet, sondern in allen vier Ecken.
      4. Die finale Position wird festgelegt durch Auswahl der Position, die am häufigsten bestimmt wurde.
      5. Man hat die Möglichkeit, die Position anhand eines folgenden ähnlichen Bildes zu verifizieren. Dabei werden zur Verifikation ausserdem die Daten desjenigen Blobs verwendet, der den ähnlichsten Schwerpunkt hat und die sich somit entsprechen sollten.
      6. Es werden nicht mehr die nächsten Nachbarn gesucht, sondern es wird analysiert, auf welcher Achse die Blobs liegen und gezielt dort der nächste Blob gesucht. So können Diagonal liegende Punkte nicht mehr die Erkennung stören.
   7. Weitere Erkenntnisse:
      1. Die Ergebnisse sind nur wenig besser geworden und genügen den Anforderungen noch nicht.
      2. Die Fehlerkennungen sind auf folgenden Bildern ähnlich, somit bringt die Verifikation praktisch nichts.
      3. Das Debugging, das schon vorher aufwendig und mühsam zurückzuverfolgen war, ist jetzt praktisch unmöglich. Man kann nicht mehr herausfinden woher gewisse Werte kommen oder den Ablauf rekonstruieren.
2. Nutzung der Interpolation
   1. Hierbei wird ausgenutzt, dass man anhand der Interpolation mit den Eckpunkten die Bildschirmkoordinaten ungefähr voraussagen kann. Es gibt dazu mehrere Möglichkeiten, die im Abschnitt ??? beschrieben werden.
   2. Ablauf:
      1. Man iteriert durch die Blobs und bestimmt zu jedem die Interpolation des Schwerpunkts.
      2. Es wird überprüft, in welchem Quadrat der Schwerpunkt zu liegen kommt. Falls das Quadrat die gleiche Farbe hat, wird die Position als gültig angenommen.
   3. Vorteile:
      1. Mit dieser Methode können Abweichungen von +- einem Quadrat erkannt werden.
      2. Es gibt praktisch keine Fehlerkennungen.
      3. Durch vergrössern der Quadrate kann die Erkennung verbessert werden.
      4. Dieses Verfahren ist viel performanter als andere.
   4. Nachteile:
      1. Es wird nur ein kleiner Teil der Blobs korrekt erkannt, die meisten bleiben ungültig.
      2. Durch Vergrösserung der Quadrate wird die Menge der nutzbaren Daten ebenfalls verkleinert.
3. Strahlensatz:
   1. Es wird ein physikalisches Modell mit einer Lochkamera erstellt. Durch Anwendung des Strahlensatzes kann ein mapping erstellt werden.
   2. Eine zusätzliche Voraussetzung dafür ist dafür ist, dass die Projektion ein rechtwinkliges Dreieck ist. Das heisst, dass die Trapezkorrektur des Beamers immer zwingend ausgeführt werden muss.
4. K-Means auf den Achsen
   1. Anhand der Eckpunkte des Bildschirms kann die obere Kante berechnet werden. Man sucht die n (Anzahl Quadrate auf der X-Achse) nächsten Blobs zu dieser Achse. Mittels Standartabweichung können Quadrate, die fälschlicherweise dazugekommen sind, wieder entfernt werden.
   2. Ablauf:
      1. Die erkannten Blobs werden auf der Achse, die anhand ihrer Schwerpunkte interpoliert wurde angeordnet und anschliessend die Position jedes Blobs bestimmt.
      2. Mit dem gleichen Verfahren werden nun die Blobs bestimmt, die am nächsten bei der eben bestimmten Achse der Schwerpunkte liegen.
   3. Dieses Verfahren kann in beiden Dimensionen angewandt werden, um die Daten zu verbessern.

Interpolation:

Hierbei geht es darum, wie man anhand der gefundenen Referenzpunkte die genauste Interpolation für ein mapping eines Kamerapixels auf eine Bildschirmkoordinate erhält. Diese Verfahren haben wir testweise auch auf die Ecken des erkannten Bildschirms angewandt, um die Genauigkeit zu überprüfen.

Lineare Interpolation:

*Skizze*

Hierbei wird ausgenutzt, dass die Verhältnisse der Strecken auf der Abbildung identisch bleiben. Dies ist in unserem Fall nicht ganz korrekt, da es die perspektivische Verzerrung nicht berücksichtigt, was sich nicht schwerwiegend auswirkt, jedoch zu einer gewissen Ungenauigkeit führt. Nachteile dieses Verfahrens sind zudem, dass in jedem Quadranten des gesuchten Punktes ein Punkt gefunden werden muss, die zusammen auf dem Bildschirm ein Rechteck bilden. Diese Voraussetzung ist für die Eckpunkte gegeben, jedoch ein erschwerender Faktor wenn eine unregelmässige Verteilung von Referenzpunkten gegeben ist. Ausserdem kann dieses Verfahren nur in eine Richtung angewandt werden, nämlich von Bildschirmkoordinaten zu Kamerapixeln. Dies ist für uns die falsche Richtung, sodass wir alle Werte vorberechnen und speichern müssen.

Formel:

*Formel*

Baryzentrische Koordinaten:

Funktionsweise:

Baryzentrische Koordinaten beschreiben in jedem n-Eck n Faktoren, die das Flächenverhältnis zwischen den Flächen zwischen den Kanten und einem Punkt beschreiben. Deren Summe ist immer 1. Anhand dieser Faktoren lässt sich sehr einfach ein Rebasing durchführen. Das heisst, wenn man Die Punkte in einem anderen System hat, die den Eckpunkten im Ursprungssystem entsprechen, kann man die Position des Punktes im Zielsystem analysieren. Dies ist in unserem Fall bei allen Referenzpunkten gegeben. Ein Punkt kann sogar bestimmt werden, wenn er nicht innerhalb des Dreiecks liegt. Wir werden hier dieses Thema nicht weiter behandeln, es gibt viele andere Papers, die dieses Thema sehr ausführlich behandeln.

Unsere Anwendung:

Byzantinische Koordinaten berücksichtigen leider keine perspektivischen Verzerrungen, wie sie bei uns vorkommen, was deren Funktionalität leider etwas einschränkt. Somit können wir sie nur verwenden, wenn ein Punkt innerhalb oder zumindest in der Nähe liegt. Andernfalls gibt es extreme Ungenauigkeiten, die ein Resultat unbrauchbar machen. Wir definieren die Nähe dadurch, dass alle Faktoren kleiner als 1.5 sein müssen. Wenn sie kleiner als 1 sind ist der Punkt innerhalb. Mit 1.5 darf er zu einer Kante eine Fläche gegen aussen aufspannen, die maximal der Hälfte der ganzen Fläche entspricht. Das entspricht einen maximalen Abstand der halben Dreieckshöhe zur Kante. Da die Nutzung von n-Ecken weder zusätzliche Genauigkeit noch sonstige Vorteile bringt, nutzen wir Baryzentrische Koordinaten nur in Dreiecken. Durch eine Mittelung der resultierenden Koordinaten anhand mehrerer Dreiecke können wir die Störungen der Verzerrung etwas minimieren. Wenn wir genügend Referenzpunkte haben, fällt das nicht mehr ins Gewicht.

Ergebnisse:

Wenn der Beamer und das Notebook nebeneinander stehen, funktioniert die Interpolation anhand der Eckepunkte ziemlich gut. Wir haben jedoch hauptsächlich unter erschwerenden Umständen getestet. Hier gab es bei der linearen Interpolation gewisse Verschiebungen in horizontaler Richtung. Es entspricht nicht unserer erwünschten Genauigkeit.

Mit den baryzentrischen Koordinaten anhand der Eckpunkte hatten wir anfangs sehr viele Ungenauigkeiten. Zudem waren Knicke in den Linien sichtbar über die Diagonale. Eine Mittelung der Ergebnisse aller vier Dreiecke hat jedoch geholfen, die Knicke zu eliminieren und die vertikalen Fehlerfaktoren zu eliminieren. Dafür haben wir damit Probleme an den Rändern, da dort unrealistische Werte entstehen. *Tabelle.* Die horizontale Verschiebung ist je nach Versuchsaufbau ähnlich zum linearen Ansatz. Mir mehr Referenzpunkten lässt sich dieses Problem aber sicher lösen.

Kompensation der Verzerrung:

Wenn nur drei oder vier Referenzpunkte für eine Interpolation verwendet werden, ist das Resultat in der aufgespannten Fläche korrekt und sollte keine Störungen beinhalten. Wenn mit diesem Verfahren jedoch eine Linie über mehrere Dreiecke gezogen wird, hat diese an jedem Übergang eine neue Richtung. Bild. Das wird von einem Nutzer dadurch wahrgenommen, dass seine gezogene Linie nicht gerade ist, sondern Knicke enthält. Das wird als störend empfunden.

Lösung:

Es wird nicht nur das Resultat eines Vierecks analysiert, sondern direkte Nachbarn werden ebenfalls berücksichtigt. Durch eine entsprechende Gewichtung sollte das Resultat nicht verfälscht werden, aber die Knicke soweit geglättet werden, dass sie den Nutzer nicht mehr auffallen. Da Baryzentrische Koordinaten leider keine Abstandsfunktion definieren, gewichten wir Resultate, die innerhalb liegen doppelt, solche in der Nähe(siehe Oben) einfach. Anschliessend werden die Werte linear gemittelt. So können wir dennoch eine leichte Gewichtung einfügen.

Input-Emulation

Gemäss Aufgabenstellung war in unserem Fall lediglich eine PowerPoint-Integration gefordert. Wir haben uns jedoch überlegt, dass wir mit einer Mausemulation weit mehr machen könnten. Ausserdem kann man auch PowerPoint mit einer emulierten Maus gut bedienen. Zudem haben wir schon Erfahrungen gesammelt, wie man mit C# Mausbefehle auf Driver-Level injizieren kann. Diesen Weg benutzten ebenfalls Devices anderer Hersteller, wie z.B. Logitec Presenter. Diese emulieren jedoch eine Tastatur, was für uns nicht sinnvoll, aber optional auch möglich ist.

Unter Windows 8 ist es zudem möglich, Touch-Events zu generieren. Damit müssten wir uns nicht mehr um die Analyse der Gesten kümmern, da uns das vom Betriebssystem abgenommen wird.

Da wir jedoch auch ältere Systeme unterstützen wollen, haben wir einen Teil der Windows 8 Touch-funktionalität nachgebildet, die wir mit Mausbefehlen nachbauen. Dies ist ein Rechtsklick durch warten an einem Ort.