# Innere Architektur

## Überlegungen zur allgemeinen Architektur

### Events

Die Architektur der Gestenerkennungssoftware legt den Schwerpunkt auf eventbasierte Programmierung. Dies hat den Vorteil, dass auf Schleifen und Threading verzichtet werden kann. Das verringert die Komplexität der Anwendung. Zur Statussynchronisation können Event-Argument-Objekte verwendet. Dies fördert die Wart- und Anpassbarkeit des Codes. Ein wichtiger Nachteil ist das Memory Management (s.u.).

### Berechnungen

Berechnungen an den 3D-Skeletten sind in eigene Klassen gekapselt, was Korrekturen vereinfacht und Duplicated Code verhindert.

### Hauptfunktion: Gestenerkennung

Gesten werden vom Gesture-Checker erkannt. Jener überpfüft, ob die Reihenfolge der zu erfüllenden Gestenteile stimmt. Falls die Erkennung erfolreich war, triggert er die vom API-Benutzer für die erfolgreiche Erkennung hinterlegte Funktion.

## Lösungsansätz für aufgetretene Probleme

### Zuweisungsalgorithmus für neue und bestehende Personen

Die Unterscheidung von mehreren Personen hat Schwierigkeiten bereitet. Die Kinect wechselt die Nummerierung der erkannten Skelette ohne erkennbares System. Deshalb muss die Zuweisung der erkannten Skelette an neue oder bestehende Personen per Software erfolgen. Dies erfordert ein Matching zwischen den bisherigen Skeletten der existierenden Personen und den Skeletten die wir jeweils neu von der Kinect bekommen.

Die Ähnlichkeit eines Skelettes zu einem anderen wird lediglich anhand des Skelettgliedes „Hüfte“ bewertet. Diese Bewertung ergibt eine 2D-Matrix mit den Abweichungen als Einträge. Das Problem war nun die Auswertung dieser Ähnlichkeiten. Wie kann man am besten auswerten, welches Skelett welcher Person zuzuweisen ist? Eine Idee war, jeweils das Minimum in der Matrix zu suchen, die Zuweisung zu machen und sowohl Skelett als auch Person aus der Match-Matrix zu löschen. Das ist aber eventuell im Durchschnitt nicht die beste Zuweisung.

Falls die Kinect ein Skelett in einem Frame nicht mehr erkennt, weil z.B. die Sicht durch eine andere Person, oder ein Objekt verdeckt ist, löschen wir die Person nicht direkt, sondern speichern sie in einem Speicher für 5 s. Falls sie wieder erkannt wird, wird sie reaktiviert, oder nach der Zeit gelöscht.

Wir entschieden uns vorerst für eine naive Lösung mit Listen, die jedoch gut zu funktionieren scheint:

Es werden drei Fälle unterschieden:

1. Es hat mehr Skelette als schon bestehende Personen, d.h. es kam eine Person ins Bild, sie muss aus dem Cache geladen werden. neu erstellt werden. Zudem müssen ihr die benötigten Events registriert werden.
2. Es hat mehr bestehende Personen als neue Skelette, d.h. es ging eine Person aus dem Bild. Sie muss gelöscht/vergessen werden (Sie bleibt in einem dafür vorgesehenen Cache).
3. Es hat gleich viele Personen und Skelette, d.h. Zuweisung muss neu gemacht werden, sonst nichts.

Die Zuweisung erfolgt nun einfach mittels zwei temporärer Listen, aus welchen die gematchten Elemente gelöscht werden. Was übrig bleibt muss nach den drei Fällen (s.o.) beurteilt werden.

### Zeitmessung in der .NET-Umgebung

Für die Zeitmessung in .NET gibt es verschiedene Möglichkeiten:

DateTime.Now.Milliseconds: Gibt die Millisekunden der aktuellen Sekunde aus, d.H. nur Werte zwischen 0 und 999, damit ist keine sinnvolle Zeitmessung möglich.

DateTime.Now.Ticks: Gemäss MSDN die Anzahl Millisekunden seit Systemstart \* 10. Sind aber alles andere als genau somit auch unbrauchbar.

Lösung: Zeitdifferenz in Millisekunden von DateTime.Now und dem 1.Jan.1970 in Millisekunden, für Timeouts. Durch das Skeduling des CPU taugt diese Methode aber nicht für die Geschwindigkeitsberechnung der Skelette. Für diese haben wir den von der Kinect mitgelieferten Timestamp in den Skeletten gespeichert.

### Event-Triggers aus Subklassen

Events, aus Subklassen können nicht direkt aufgerufen werden, sonden müssen in der Subklasse von einer protected Funktion gekapselt werden –in folgendem Stil: *fireSuperclassEvent()*.

Eine andere Möglichkeit besteht darin, die Events der Superklasse virtual zu deklarieren, und sie in der Subklasse zu überschreiben.

### GestureChecker-Statemachine: Unterscheidung zwischen triggered und success

Für Gesten, die eine laufende Bewegung erkennen, haben wir zwei Events. Das Erste (triggered) wird aktiviert, sobald die Erkennung erfolgreich war, Success wird aktiviert, sobald die minimale Distanz/Zeit, während der die Geste laufen muss, erreicht ist. Eine andere Möglichkeit ist, RelativeMovement im Checker mit duration zu verwenden.

### …

## Domain

Die folgende Domainanalyse ist stark vereinfacht und soll einen groben Überblick über die Gestenerkennungssoftware ermöglichen.

|  |  |
| --- | --- |
| Objekt | Beschreibung |
| Checker | Der Checker benutzt die von der *Person* abgespeicherten Skelettdaten um verschiedene zeitabhängige Berechnungen durchführen zu können, z.B:  absolute Geschwindigkeit, relative Geschwindigkeit, etc. |
| Condition | Eine *Condition* ist der eigentliche Gestenteil. Sie kann mit *check* auf Gültigkeit überprüft werden. Zudem beinhaltet sie die Events *OnSuccess* und *OnFail*. |
| Device | Das *Device* kapselt das physikalische Gerät. Es gibt *Personen* per Event zurück wenn sie aktiv werden und meldet neue *Skelette*. |
| GestureChecker | Der *GestureChecker* führt Buch über eine komplette Geste. Er speichert wie weit fortgeschritten die Erkennung einer Geste ist und feuert Events bei der erfolgreichen Beendigung oder beim Abbruch. |
| SkeletonMath | Klasse für Vektorberechnungen auf Skelettdaten. |
| SmothendSkeleton | Geglättete *Skelette* sind die Datenquelle für alle Berechnungen und damit für die Gestenerkennung. Sie werden in der *Person* gespeichert und vom *Checker* verarbeitet. |



## Beispielsequenz einer Geste

Der Ablauf einer Gestenerkennung lässt sich in drei Phasen aufteilen: Die grafische Darstellung ist auf der folgenden Seite.

1. Initialisierung:

In jedem Framezyklus werden alle *Personen* vom *Device* eindeutig identifiziert. Die identifizierten *Personen* bekommen vom *Device* ihr aktuelles *Skelett* zugewiesen. Jede *Person* erhält so mit der Zeit einen Skelettcache von 10 zeitlich geordneten *Skeletten*, welche für dynamische Berechnungen verwendet werden können.

1. Prüfen:

Nach der Zuweisung vom neuen *Skelett* wird von den der *Person* zugeordneten *GestureCheckern* geprüft ob das aktuelle *Skelett* einen gültigen Kontext für die registrierten *Conditions* bildet.

1. Feedback:

Der GestureChecker führt Buch über die Reihenfolge der Gestenteile (Conditions). Fall eine Geste komplett fertig durchlaufen wird, ist sie erfolgreich und wird mit einem von der User-API registrierten Event quittiert.



## Memory Management: Speicherzuweisung für Eventhandling

Das verwenden von Events hat den Nachteil, dass long-lived Publisher ihre Referenzen zu Event-Subscribern im Speicher behalten und so zu Speicherlecks führen können. Unsere Applikation wurde auf Lecks geprüft und für stabil empfunden.

### Code-Analyse der Zuweisungen

In folgender Tabelle sind alle EventHandler-Referenzen aufgeführt. Es ist ersichtlich, dass praktisch alle EventHandler-Referenzen vom GarbaggeCollector beim Aufräumen einer *Person*-Instanz indirekt im Speicher invalidiert werden. Der Event im Device ist auch unproblematisch, da sowohl die *KinectSensor*- als auch die *Device*-Instanz long-lived sind. Zur Sicherheit wird er jedoch beim Entfernen einer *Person* aus dem Expiration-Cache deregistriert.

Da eine *Person*-Instanz jeweils nur während der Bedienungszeit existiert, sind die verwendeten Events unproblematisch. Der Speicher wird nicht zu sehr beansprucht.

|  |  |  |  |
| --- | --- | --- | --- |
| Klasse | Publisher | Subscriber | Subscriber lebt länger oder gleich lang wie Publisher |
| Condition | Person.NewSkeleton | Condition.check | ✔ |
| Device | KinectSensor.SkeletonFrameReady | Device.NewSkeletons | ✔ |
| Device | Person.OnWave | Device.personWaved | **?** |
| GestureChecker | Condition.Succeeded | GestureChecker.ConditionComplete | ✔ |
| GestureChecker | Condition.Failed | GestureChecker.ConditionFailed | ✔ |
| ~~GestureChecker~~ | ~~Timer.Elapsed~~ | ~~GestureChecker.Timeout~~ | **✘** |
| Person | GestureChecker.\* | Person.\* | ✔ |

Bemerkung: Der Publisher hält jeweils eine Referenz auf den Subscriber, nicht aber umgekehrt.

Die einzige Ausnahme bildete die Verwendung der *Timer*-Klasse im *GestureChecker*. Der *Timer* registrierte sich bei System Events um den erfolglosen Ablauf einer Geste durch einen Timeout abzubrechen. Durch das Verwenden von vielen *GestureCheckers* entstand ein Speicherleck (siehe unten). Der Timer wurde durch eine einfachere Zeitmessung ersetzt und wird gar nicht mehr verwendet.

### C:\Users\scj\Desktop\timerLeak.pngOnline-Analyse mit Mocking

Zum Testen der Stabilität des Speichers wurde ein *Device (MockingDevice)* vorgetäuscht und das Erscheinen und Verschwinden von vielen Personen vorgetäuscht. Zudem wurden kritische Klassen wie *Condition*, *GestureChecker* und EventArgs mit grossen Byte-Arrays aufgepumpt. So wurde das einzige Speichleck schnell sichtbar.

Was jedoch zum Problem wurde war die Verwendung des C#-Timers. Jener wurde benutzt um auf einen Timeout in der GestureChecker-Statemachine zu reagieren. Dabei behielt ein System-Event des jeweils praktisch alle Objekte der Applikation. Durch die automatisierte Analyse mit SciTech .NET Memory Profiler konnten die aufgepumpten Objekte zurückverfolgt werden. Im dem Allocation-Tree ist gut sichtbar, dass praktisch alle Instanzen der Applikation vom Timer gehalten werden.

Da aus der Analyse der Aufrufshierarchie zusätzlich ersichtlich war dass der *Timer* sehr viel Rechenzeit verbrauchte, wurde komplett auf diese Klasse verzichtet. Stattdessen wird jetzt eine weniger komplexe Zeitmessung aufgrund der Systemzeit verwendet um den Timeout einer Geste zu messen.

# Äussere Architektur (API)

## Allgemeine Überlegungen

### Multi-Layer

Die User-API wird in verschiedenen Layern aufgebaut. Der Benutzer des Gestenerkennungsframework kann entscheiden, welchen Layer und damit auch welche Komplexität er benutzen will. Das hohe Layer bietet einen eingeschränkten Funktionsumfang, den man sehr einfach einbinden kann. Das tiefe Layer bietet Möglichkeit eigene Gesten zu definieren oder auf Low-Level Eigenschaften zuzugreifen. Beide Layer lassen sich kombiniert einsetzen. Beispielsweise muss sich der Benutzer der API nicht um die Aktivierung der Personen kümmern, kann aber dennoch eigene Gesten definieren – oder umgekehrt.

## Schnittstellendefinition – Hoher Layer

|  |  |
| --- | --- |
| Klasse | Beschreibung |
| Device | Von einer *Device*-Instanz bekommt man alle neuen *Personen* indem man sich beim *NewActivePerson*-Event registriert. Nach dem Aufruf von *Start()* initialisiert das *Device* die Kinect und beginnt mit der Erkennung von *Personen*. Sobald sich eine Person durch Winken anmeldet, wird dieser Event gefeuert. Im *ActivePersonEventArgs*-Parameter wird diese *Person* mitgegeben. |
| Person | Bei einer *Person*-Instanz können Gestenreaktionen registriert werden. Dies sind die folgenden:   * *OnZoom*: übermittelt den Zoomfaktor * *OnSwipe*: signalisiert eine Wisch-Geste * *OnWave*: signalisiert Winken |

Beispiel:

Device d = new Device(); // Erstellen des des Devices

d.PersonActive += NewPerson; // Registrieren auf neue Personen

d.Start(); // Starten der Kinect

void NewPerson(object src, ActivePersonEventArgs activePersonEventArgs )

{

activePersonEventArgs.Person.OnSwipe += // Auf Swipe der aktiven Person hören

delegate(object sender, GestureEventArgs e)

{

Console.Write(activePersonEventArgs.Person.Id // Anzeigen der ID der Person

+ " swiped " +

((SwipeGestureEventArgs)e) // Casten der EventArgs um weiter

Informationen zu bekommen

.Direction.ToString()); // Ausgeben der Richtung des Swipes

};

}

## Schnittstellendefinition – Tiefer Layer

|  |  |
| --- | --- |
| Klasse | Beschreibung |
| Device | Von einer *Device*-Instanz bekommt man alle neuen *Personen* indem man sich beim *NewPerson*-Event registriert. Nach dem Aufruf von *Start()* initialisiert das *Device* die Kinect und beginnt mit der Erkennung von *Personen*. Die bei *NewPerson* registrierten Funktionen werden jetzt mit dem *NewPersonEventArgs*-Parameter aufgerufen. Jener enthält jeweils eine neue Person.  Der *PersonActive*-Event wird gefeuert, wenn sich eine *Person* einloggt. Im *ActivePersonEventArgs*-Parameter wird diese *Person* mitgegeben. |
| Person | Bei einer *Person*-Instanz können zusätzlich zu den Standartgesten auch noch die Roh-Skelettdaten ausgelesen werden. |
| GestureChecker | Wenn man einen eigenen GestureChecker erstellt und mit eigenen Conditions befüllt, kann man sich auf seine eigenen Gesten registrieren. |
| Condition | Hiermit kann man eigene Bedingungen definieren, die chronologisch überprüft werden im Gesture Checker. Die Condition gehört jeweils nur zu einer Person und sie wird vom GestureChecker automatisch aktiviert und deaktiviert. |
| Checker | Damit kann man komplexe Berechnungen mit bestimmten Punkten machen, und man kriegt einfach Richtungen und Geschwindigkeiten zurück. |
| SkeletonMath | Diese Klasse bietet Vektoroperationen an, die speziell auf Skelettpunkte ausgelegt sind. |
| GestureEventArgs | Hiermit kann die Geste noch Argumente übergeben. |

Beispiel:

class Tester

{

private static JumpGestureChecker jgc;

static void Main(string[] args)

{

Device d = new Device(); // Erstellen des des Devices

d.NewPerson += NewPerson; // Registrieren auf neue Personen

d.Start(); // Starten der Kinect

}

static void NewPerson(object src, NewPersonEventArgs newPersonEventArgs )

{

jgc = new JumpGestureChecker(

newPersonEventArgs.Person); // Anlegen des Eigenen GestureCheckers

jgc.Successful += delegate

{ Console.WriteLine("Jump"); }; // Registrieren auf dessen Event}

}

class JumpGestureChecker : GestureChecker // Klasse implementiert GestureChecker

{

public JumpGestureChecker(Person p) // übergeben der zu überwachenden Person

: base(new List<Condition>

{

new JumpCondition(p) // Anlegen einens GestureCheckers mit einer

JumpCondition

}, 1000){} // timeout ist hier nicht von Belang

}

class JumpCondition : Condition // JumpCondition prüft, ob gesprungen wurde

{

private Checker c; // Checker für die Berechnungen

public JumpCondition(Person p) : base(p)

{

c = new Checker(p);

}

protected override void Check(

object src, NewSkeletonEventArgs e) // überprüfung bei jedem neuen Skelett

{

if (c.GetAbsoluteMovement(JointType.HipCenter) // Bewegung der Hüfte

.Contains(Direction.Upward)) // nach oben?

{

FireSucceeded(this, new JumpGestureEventArgs()); // Condition erfolgreich

}

else

{

FireFailed(this, new FailedGestureEventArgs

{Condition = this}); // nicht erfolgreich

}

}

}

class JumpGestureEventArgs : GestureEventArgs{} // Args für optionale Parameter