Übersicht: Ablauf von Bewegung und Fressen im Insekten-Simulator

5. April 1998

1 Allgemeines zum Insekten-Simulator

1.1 Aufbau

Der Simulator besteht im wesentlichen aus drei Schichten:

- *Die Simulationsschnittstelle* `Sim`*:* Diese Ebene stellt die ausschließliche Schnittstelle für Clients dar. Objekte von "außen" dürfen nur die in `Sim.idl` aufgeführten Methoden aufrufen, um mit dem Simulator-Kern in Verbindung zu treten.
- *Der Simulator-Kern* `Kernel`*:* Hier findet die eigentliche Verrechnung der über die Simulationsschnittstelle eingehenden Daten statt. So implementiert z.B. die Insekten-Klasse auf Kernel-Ebene `KerInsectImpl` das in `Sim.idl` bereitgestellte Interface `Insect`.
- *Die physikalische Verarbeitungsschicht* `Physical`*:* Hier findet die rein physikalische Verwaltung von Raum und Zeit statt. Insekten, Pflanzen und Steine werden auf Objekte (`Things`) reduziert. Auf dieser Ebene befindet sich der Zeitgeber (`Clock`) und die Verwaltung der räumlichen Anordnung der Objekte (`World`) sowie deren Sichtbereich.

1.2 Kommunikation der Schichten

Die Kommunikation der einzelnen Schichten soll natürlich möglichst geordnet vor sich gehen (z.B. zur Vermeidung von deadlocks). Außerdem möchte man die Schichten aus Effizienzgründen auch auf verschiedene Rechner verlegen können. Aus diesem Grund kommunizieren Objekte aus unterschiedlichen Schichten ebenfalls über den Corba-Mechanismus, obwohl sie zur Zeit lediglich in verschiedenen Prozessen auf demselben Rechner gestartet werden.

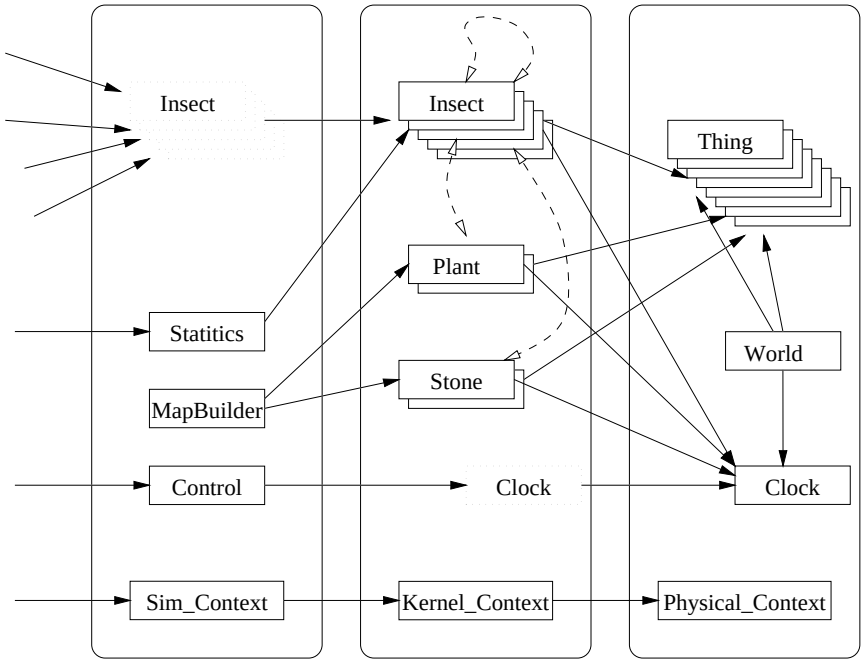

Abbildung 1: Aufbau des Simulators

1.3 Das Insekt im Simulator

Ein Insekt wird im Simulator in erster Linie auf der Kernel-Ebene dargestellt. Die Verbindung zur Simulations-Schicht wird dadurch hergestellt, daß das C++-Objekt `KerInsectImpl` die Corba-Schnittstelle `Sim::Insect` implementiert, insofern also streng genommen zu beiden Schichten gehört. Die Verbindung zur physikalischen Schicht geschieht durch eine eineindeutige Zuordnung zu einem physischen 'Ding' `Physical::Thing`.

2 Ablauf der Insektenbewegung auf dem Server

Betrachten wir nun, was passiert, wenn man von einem Insekten-Client aus eine Bewegung des zugehörigen Insekts im Kernel veranlassen möchte, also z.B. im InsectTool eine neue Position eingibt und diese per 'doMove'-Button abschickt:

Zunächst wird vom Client über das `Sim`-Interface die Funktion `doMove()` aufgerufen. Diese wird von der C++-Klasse `KerInsectImpl` auf Kernel-Ebene implementiert. Die Funktion `doMove()` errechnet nun aus den Zielkoordinaten und den gegenwärtigen Koordinaten die zu bewältigende Distanz und teilt dann unter Berücksichtigung der Geschwindigkeit diese Strecke in mehrere gleichgroße

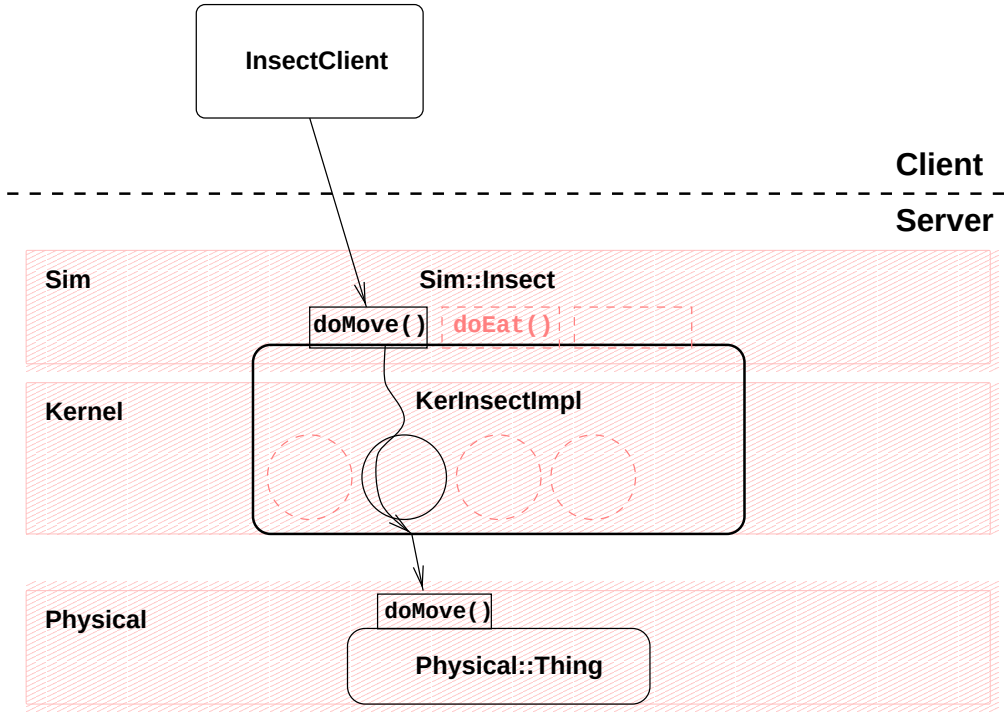

Abbildung 2: Ablauf der Bewegung eines Insekts im Simulator

Stücke auf.

Die Streckenpunkte werden daraufhin über eine `PositionList`, einer Liste aus x-y-Koordinatenpaaren an die physikalische Ebene übergeben. Dort läuft die eigentliche Bewegung des 'Dings' dann zunächst unabhängig an. Um die Bewegung jedoch weiter auf Kernel-Ebene zu überwachen, wird im Hilfsobjekt `Mover` die Anzahl der Schritte und die zu bewältigende Distanz gespeichert. Die Überwachung wird dabei in der von der Methode `start()` durchlaufenen Endlosschleife (quasi der 'Herzschlag' des Insekts) durch den Aufruf der Funktion `processMove()` vorgenommen. Diese Methode befragt ihrerseits den `Mover`, ob die gewünschte Bewegung mit den vorhandenen Energieressourcen vereinbar ist und inwieweit diese durch die Bewegung reduziert werden. Sind nicht mehr genügend Ressourcen vorhanden, wird der physikalischen Ebene ein Signal zum Stoppen der Bewegung übergeben, und das Ziel im `Mover` gelöscht.

Problematisch könnte bei dieser Form der Abarbeitung die direkte Verbindung zwischen `Sim::Insect::doMove()` und `Physical::Thing::doMove()` sein, die bewirkt, daß ein wildgewordener Insekten-Client, durch endloses Aufrufen von Bewegungsbefehlen in der Lage ist, den Ablauf auf der physikalischen Ebene auszubremsen.

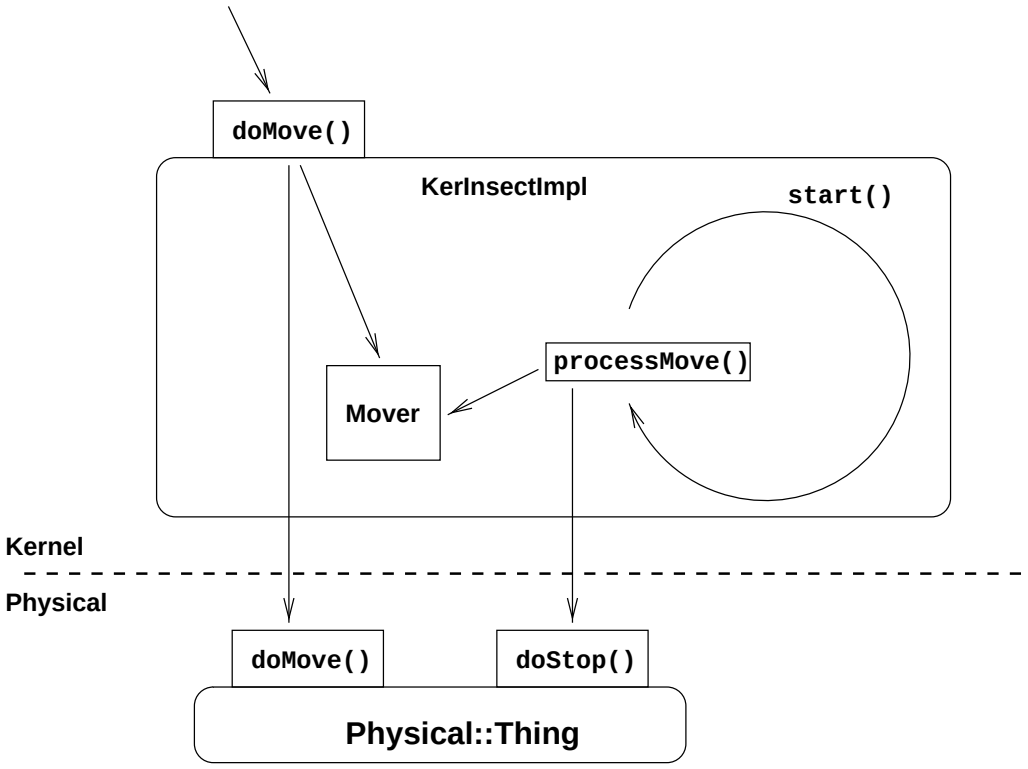

Abbildung 3: Ablauf der Bewegung im Kernel

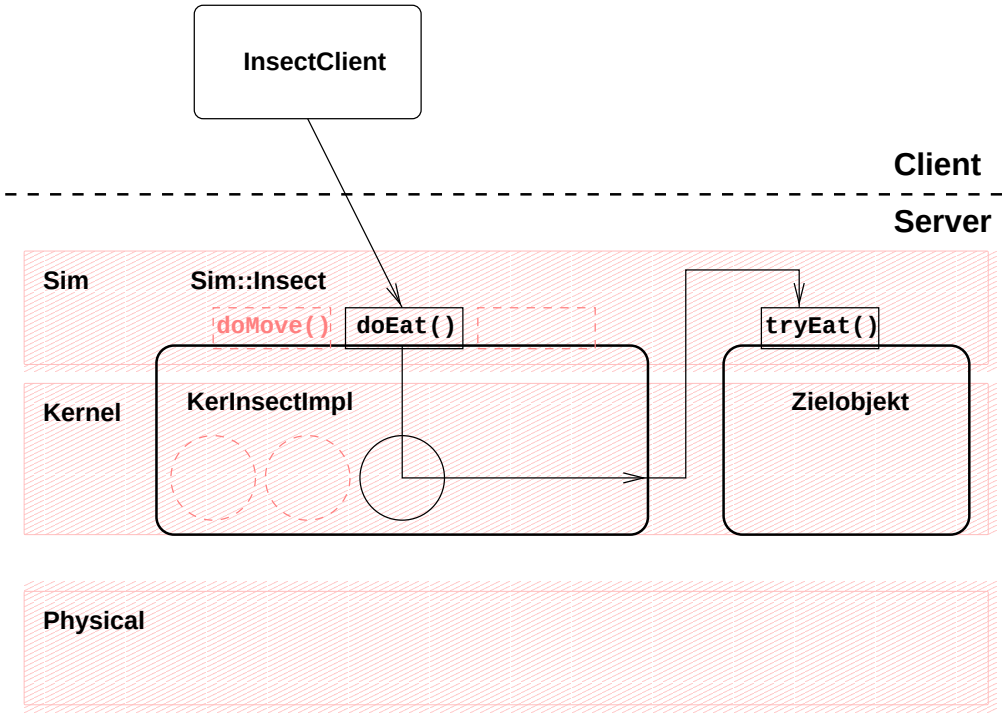

Abbildung 4: Ablauf des Fressens auf dem Server

3 Ablauf des Fressens auf dem Server

Ähnlich wie bei der Bewegung verhält es sich beim Fressen: Will ein Insekt ein 'Ding' (also ein anderes Insekt, eine Pflanze oder einen Stein) fressen, so beginnt die Ausführungskette mit der Angabe eines Zielobjekts (durch `Sim::ThingIndex` gekennzeichnet) und dem Aufruf von `Sim::Insect::doEat()`.Anders als bei der Bewegung hat das Fressen keinerlei Auswirkung auf die physikalische Ebene, in `doEat()` wird allein der `Eater` zur Aufnahme des Zielobjektes in eine Liste veranlaßt.

Das Fressen an sich wird vollständig von `processEat()` übernommen. Dort wird neben der Überprüfung der Distanz zum Zielobjekt auch die Funktion `Eater_.test()` aufgerufen, in der die prinzipielle Möglichkeit des Fressens überprüft wird. Bei positivem Ergebnis wird nun die Methode `tryEat()` im Zielobjekt aufgerufen, die für den Transfer der Nährstoffe sorgt. Die so erhaltenen Nährwerte werden dann an den Eater übergeben, der die resultierenden Angleichungen

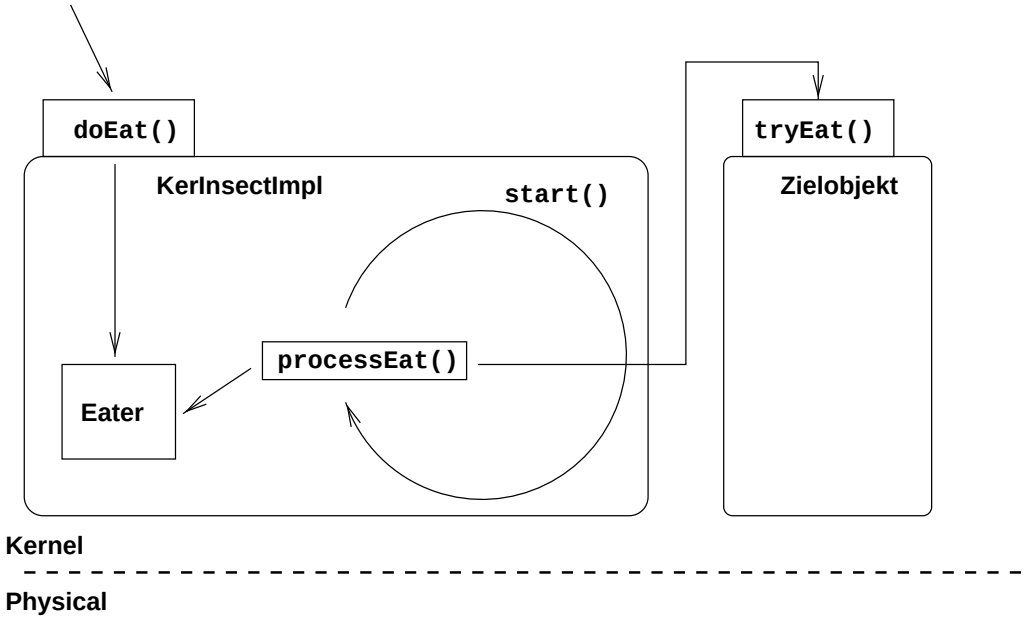

Abbildung 5: Ablauf des Fressens im Kernel

an den eigenen Attribute vornimmt.

4 Attribute

Die Attribute eines Insekts werden in der `Sim`-Schnittstelle deklariert und sind in `KerThingImpl`, der Superklasse von `KerInsectImpl` implementiert, grundsätzlich haben also auch Pflanzen und Steine Attribute. Der Zugriff auf die Attribute erfolgt dabei über die Methoden `getAttr()` und `setAttr()`. Will der `Mover` also Attribute einholen und verändern, so muß er dies mittels `Insect.getAttr()` und `Insect.setAttr()` tun, wobei `Insect` eine Referenz auf das zugehörige Insekt ist, das je nach Art der Methode als `const`-Objekt (also nur zum Lesen der Attribute) übergeben werden kann.

Es müssen nun die in `Sim.idl` definierten Attributstypen auf Ihre Bedeutsamkeit für Bewegung und Fressen hin untersucht werden. Hier ein Auszug aus der Datei:

```
//Sim.idl
...

  /** all external known attribute types of objects */
  enum AttrType {
    Kind, Size, Visibility , Armor, Attack, Reserve,
```

```
    Fitness, Sugar, Carbohydrate, Protein, Weight, Water
  };

  /** datatype for attribute values */
  typedef double AttrValue;

  /** datatype for attribute  */
  struct Attr {
    AttrType Type;
    AttrValue Value;
  };

  typedef sequence<Attr> AttrList;
...
```

Es wird nun zwischen festen und veränderlichen Attributen unterschieden: Attribute wie `Kind` und `Size` sind sicherlich fest, während z.B. die Nahrungswerte sich während des Lebenszyklus eines Insekts ständig ändern. Bei den veränderlichen Attributen wird noch einmal zwischen primären und abgeleiteten Attributen unterschieden, wobei die Attribute `Sugar`, `Carbohydrate`, `Protein` und `Water` primär sind und `Weight` und `Fitness` sich aus den anderen Attributen ableiten. `setAttrib`-Methoden sind damit nur für die 4 primären Nahrungswert-Attribute erlaubt, während die abgeleiteten Attribute lediglich fertig berechnete Vereinfachungen darstellen. Die jeweils für verschiedene Aspekte, z.B. Bewegung (`Mover`) oder Essen (`Eater`), notwendigen Modifikationen und Berechnungen der Attribute, die nicht mit der Kommunikation mit anderen Objekten verwoben sind, werden wo immer möglich in entsprechende Berechnungsobjekte ausgelagert. Diese sind im Folgenden:

Name	Funktion	zugehöriges process... kooperiert mit:
Eater	Essen	tryEat()
Fighter	Kämpfen	tryAttack()
Holder	halten und bewegen anderer Objekte	tryHold()
Looker	Beobachten anderer Objekte	getAttr()
Mover	eigene Bewegung	Physical::Thing
Observer	Benachrichtigung bei spezifizierten Attributsänderungen	self->getAttr()