Life of occam-Pi

Peter Welch (**phw@kent.ac.uk**)

CPA 2013, Napier University, 28 August, 2013

A Long Long Time Ago ...

Computers were invented: programmable calculating machines that were *Turing complete* … meaning they could do anything any future computer could do (which was pretty cool).

> Babbage's *Analytical Engine* (1837). Sadly, too far ahead of its time, not enough funding and a bad choice (in retrospect) of number representation (decimal).

Working machines had to wait for a better understanding of theory:

> Russell's *Principia Mathematica* (1910-13) …
> Godell's *Incompleteness Theorem* (1931) …
> Church's *Lambda Calculus* (1936) …
> Turing's *On Computable Numbers* (1936) …

And seriously better funding:

> The *Second World War* (1939-45) …

A Long Long Time Ago …

Following the war, the pace picked up as the commercial potential became obvious … in the USA at least (*ENIAC*, 1946), but eventually everywhere.

Ideas for hardware and software developed and were put into practice at a rate unprecedented for any other technology.

From 1970 to this day, a new and unfortunate race started. It involves the phenomenon known as *Moore's Law*. Paraphrasing (and only slightly mis-quoting) *David May* (2005):

> *"Hardware capabilities double every 18 months. Unfortunately, software overheads respond by doubling every 17 months …"*

We experience this every day. Our laptops/tablets/phones are millions of times more powerful than the computers that landed the Apollo astronauts on the moon. But what are they doing most of the time … why, so often, won't they respond to us … why the *spinning wheel of death*?

A Long Long Time Ago …

Sometime in the late 60s (or early 70s), Dijkstra made a similar point:

> *"In the beginning when we had no computers, we had no problems. Then when we had small computers, we had small problems. Now that we have big computers, we have big problems …"*

Now (2012), we have tiny computers again – mass-produced and in most everyone's pockets.

Thanks to the ingenuity of our hardware engineers, their astonishing power is barely related to their physical size.

Thanks to the ingenuity of our software engineers, we have astonishing problems …

Bad News on the Doorstep …

Something went wrong.

The foundations of computing from the first half of the *20th. Century* show that *mathematics* is probably important. The pioneers were mathematicians or engineers (who relied on mathematical models of the materials they were engineering).

Such foundations seem mostly abandoned today:

> Walk into a class of (say) second-year CS undergraduates in any (maybe, almost any) university and ask them what a *loop invariant* is … or *recursion invariant* (if they are into functional programming) or *class invariant* (if they are into object orientation). Result?

I did this in my university (academic years 2011-12 and 2012-13). Mostly blank stares (not an uncommon reaction) … but gentle further questioning revealed that they really didn't know. Many, thank goodness, did express curiosity though.

Bad News on the Doorstep …

Nailing *invariance* is one of the most powerful forms of analysis – whether in mathematics or programming. It is necessary before construction.

How can (and why do) we teach *loops* or *recursion* or *classes* without teaching the concept of *invariance*? We must form invariants in our head whenever we program a loop (recursive function, class) *– otherwise, our code will just be guesswork*. So, why not declare those invariants explicitly in our code (where they could also play a crucial role in verification)? We don't need to be mathematicians to write invariants – just good engineers. And we need to be good engineers to develop software.

Perhaps invariance is taught in a *theory* course somewhere? Not good enough! Programming practice *cannot be taught* independently of theory – they must be together. Programming practice *cannot be engaged in* independently of theory – they must be together. Ignoring elementary theory means programming blind … and the eventual result is *chaos*.

Bad News on the Doorstep …

Missing *invariance*, however, is only one of our problems with software.

Dijkstra suggested that only well-trained mathematicians should be allowed to program computers.

That is not my opinion … but I believe that in failing to engineer certain crucial mathematics into the tools provided for programming, we have failed as engineers and should not be surprised when things go very wrong.

☹ ☹ ☹ ☹ ☹

Addressing such failure is long overdue …

A Generation Lost in Space …

Where does concurrency fit in?

Computer systems – to be of use in this world – need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system … it should be *simpler*.

Yet concurrency is thought to be an *advanced* topic, *harder* than serial computing (which therefore needs to be taught and mastered first).

A Generation Lost in Space …

Where does concurrency fit in?

This *serial-first* tradition makes no sense …

… which has *(radical)* implications on how we educate people for computer science …

… and on how we apply what we have learnt …

A Generation Lost in Space …

Where does concurrency fit in?

This *serial-first* tradition makes no sense …

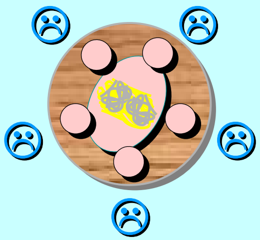
Concurrency is a powerful tool for *simplifying* the description of systems.

Performance spins out from the above, but is ***not*** the primary focus.

Of course, we need a model of concurrency that is *mathematically clean*, yields no engineering surprises and scales well with system complexity.

A Generation Lost in Space …

The story of The Dining Philosophers is due to Edsger Dijkstra – one of the founding fathers of Computer Science.



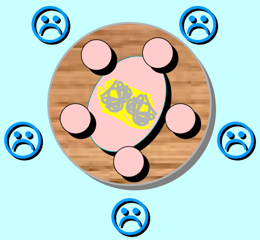
It illustrates a classic problem in concurrency: *how to share resources safely between competing consumers*.

http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

Historical document

A Generation Lost in Space ...

The story of The Dining Philosophers is due to Edsger Dijkstra – one of the founding fathers of Computer Science.



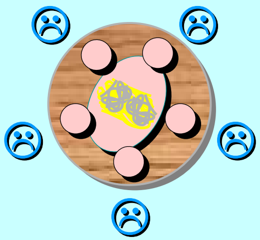
It illustrates a classic problem in concurrency: *how to share resources safely between competing consumers*.

In this example, the *resources* are the forks and the *consumers* are the philosophers.

Problems arise because of the limited nature of the resources (only 5 forks) and because each consumer (5 of them) needs 2 forks at a time.

A Generation Lost in Space …

The story of **The Dining Philosophers** is due to Edsger Dijkstra – one of the founding fathers of Computer Science.



The source of the story was a deadlock that would mysteriously arise from time to time in an early multiprocessing operating system.

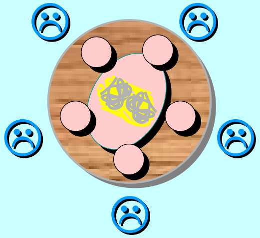
The philosophers are user processes that need file I/O.

To read or write a file, a process has to acquire a data buffer (to smooth data transfer and make it fast). If 2 files need to be open at the same time, 2 buffers are needed.

In those days, memory was scarce – so the number of buffers was limited. *The forks are the buffers*.

A Generation Lost in Space ...

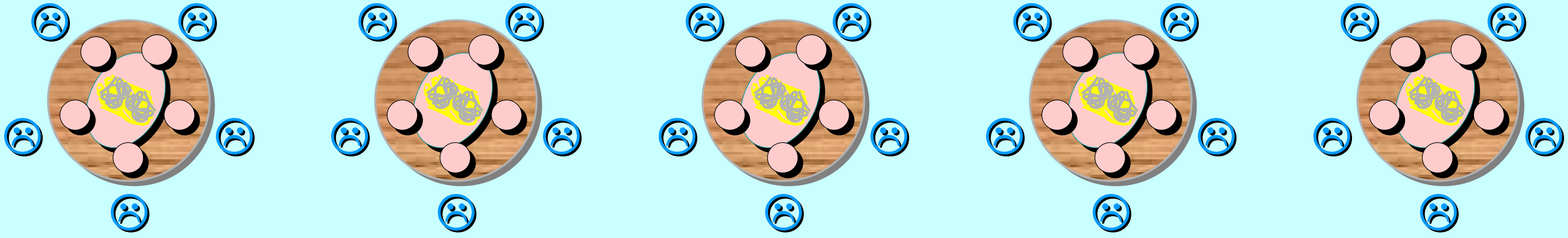
The story of **The Dining Philosophers** is due to Edsger Dijkstra – one of the founding fathers of Computer Science.



Today – some 37 years later – memory is not so scarce!

Yet, operating system (or specific application) deadlock is rampant. How often does your whole laptop/tablet/phone (or one of its apps) lock up on you?

We have been, and still are, making the same mistakes again and again and again ...



A Generation Lost in Space …

Where does concurrency fit in?

These lessons have not been learned …

The *serial-first* tradition makes no sense …

Concurrency is a powerful tool for ***simplifying*** the description of systems.

Performance spins out from the above, but is ***not*** the primary focus.

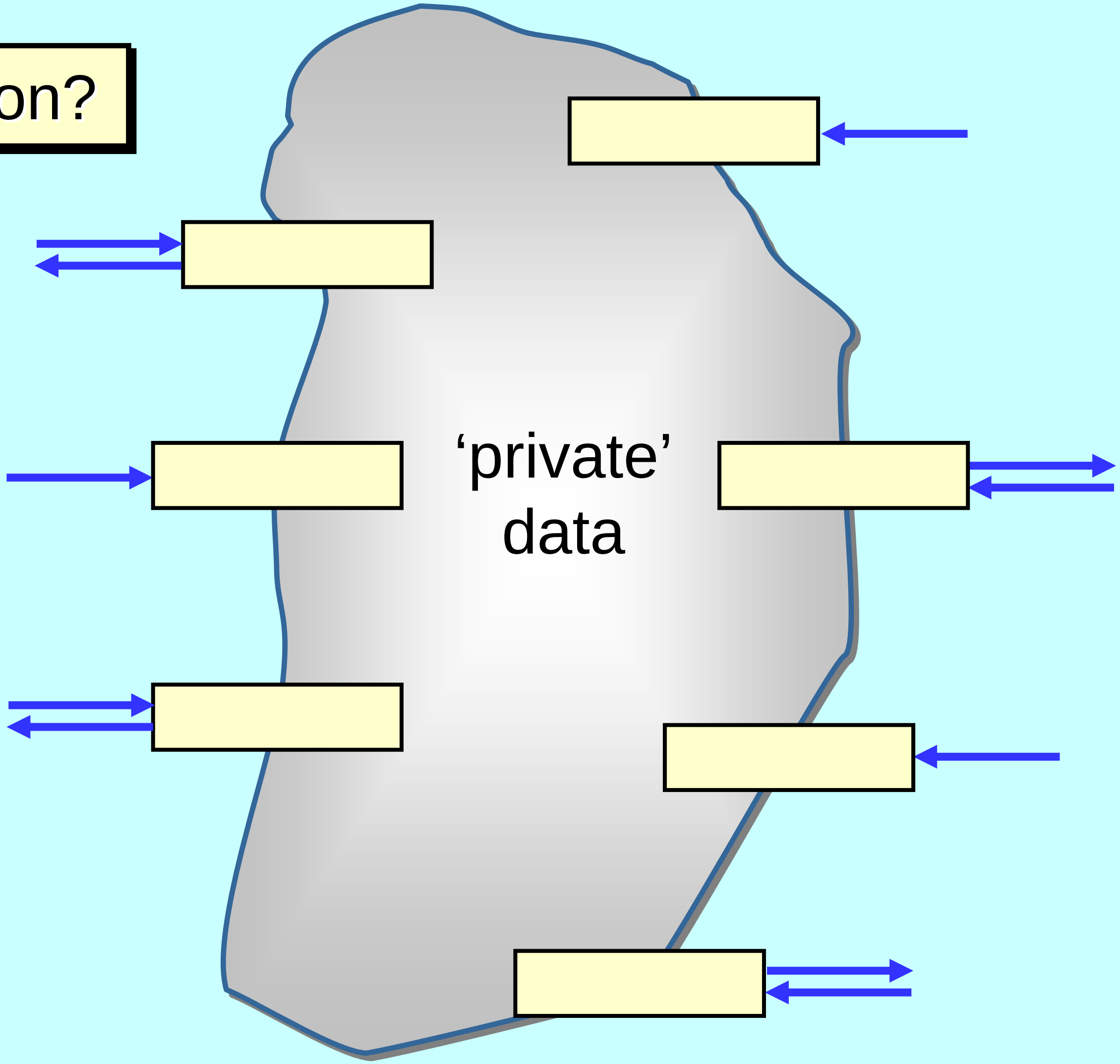
Of course, we need a model of concurrency that is ***mathematically clean***, yields no engineering surprises and scales well with system complexity.

A Generation Lost in Space ...

And object orientation?

What we tell our students:

"An opaque object interacting with a wider system of objects via its formal public interface."

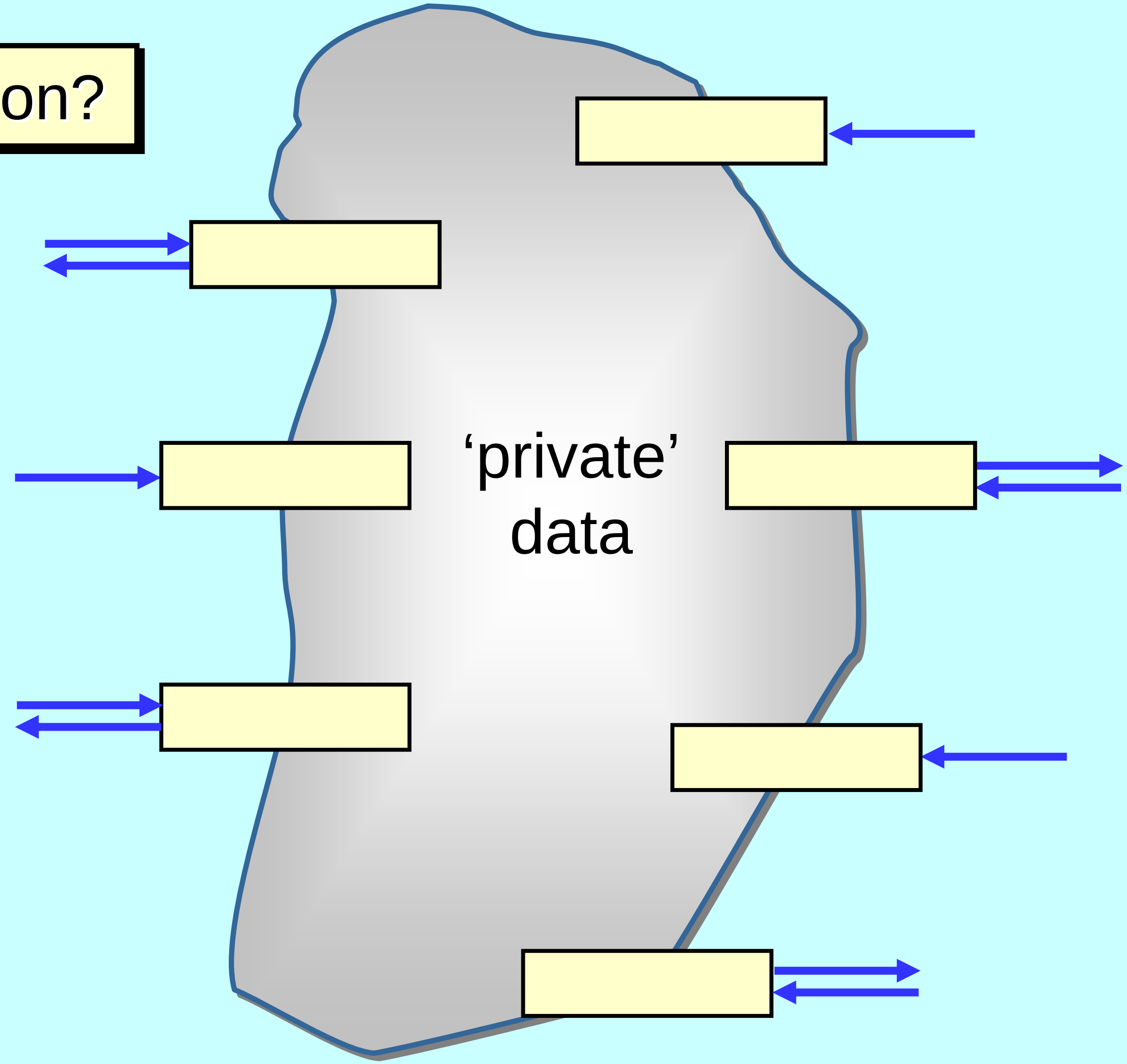


A Generation Lost in Space …

And object orientation?

Class invariants:

"Predicates on private data held within an object that are always true between method calls."

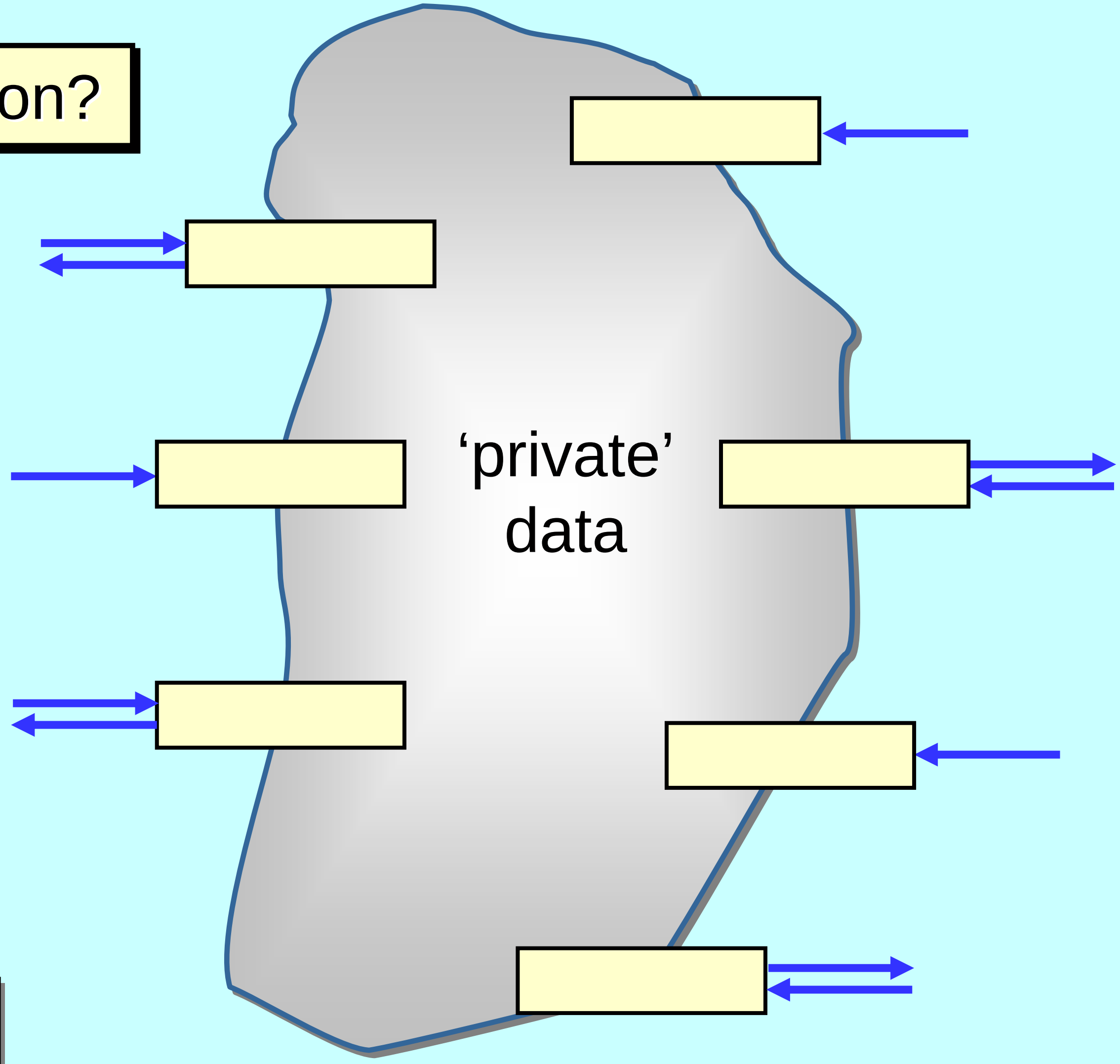


A Generation Lost in Space …

And object orientation?

Class invariants:

When writing method code, we may assume the invariants … and must re-establish them by the end.



Not good enough!

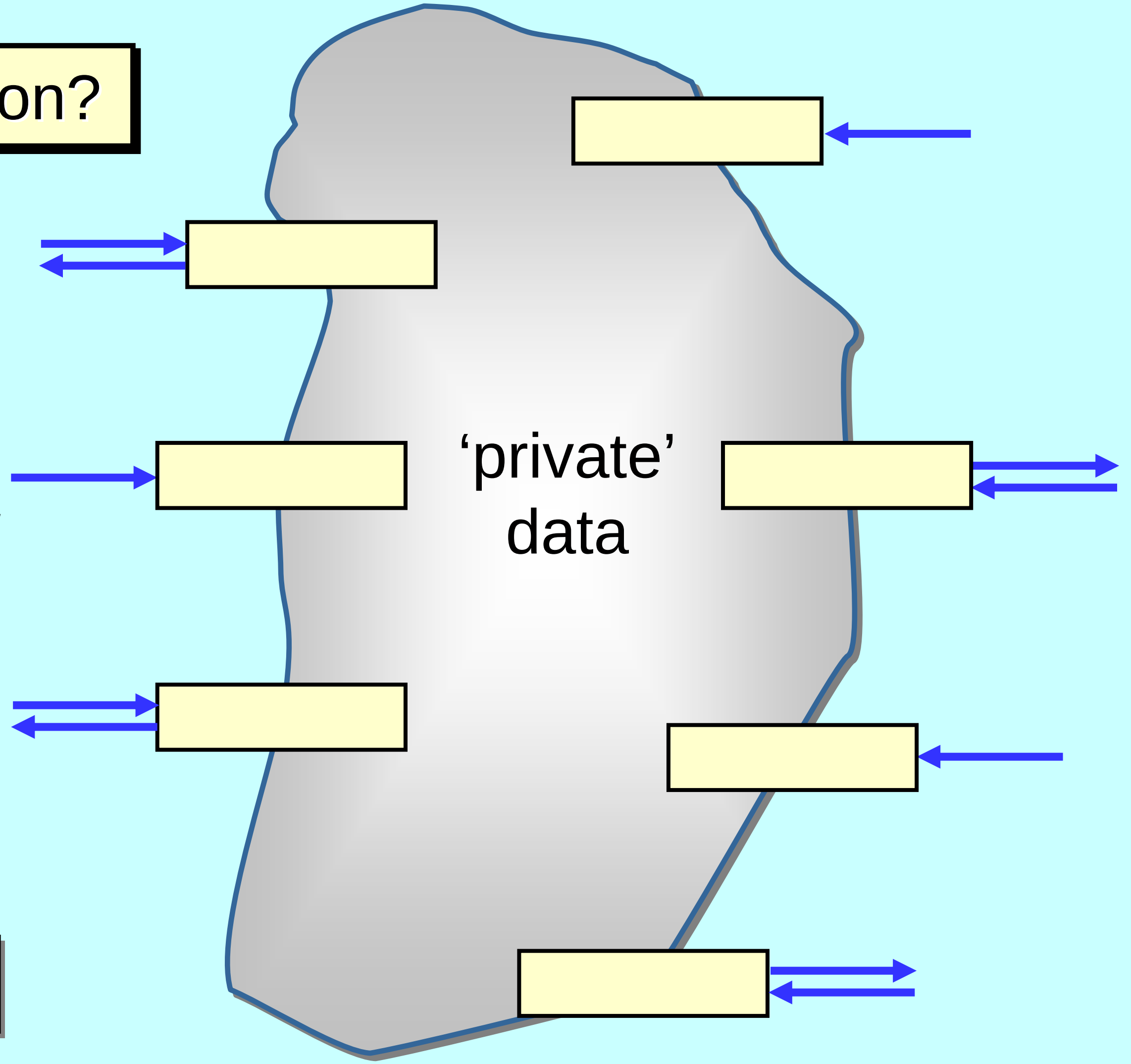
A Generation Lost in Space …

And object orientation?

Class invariants:

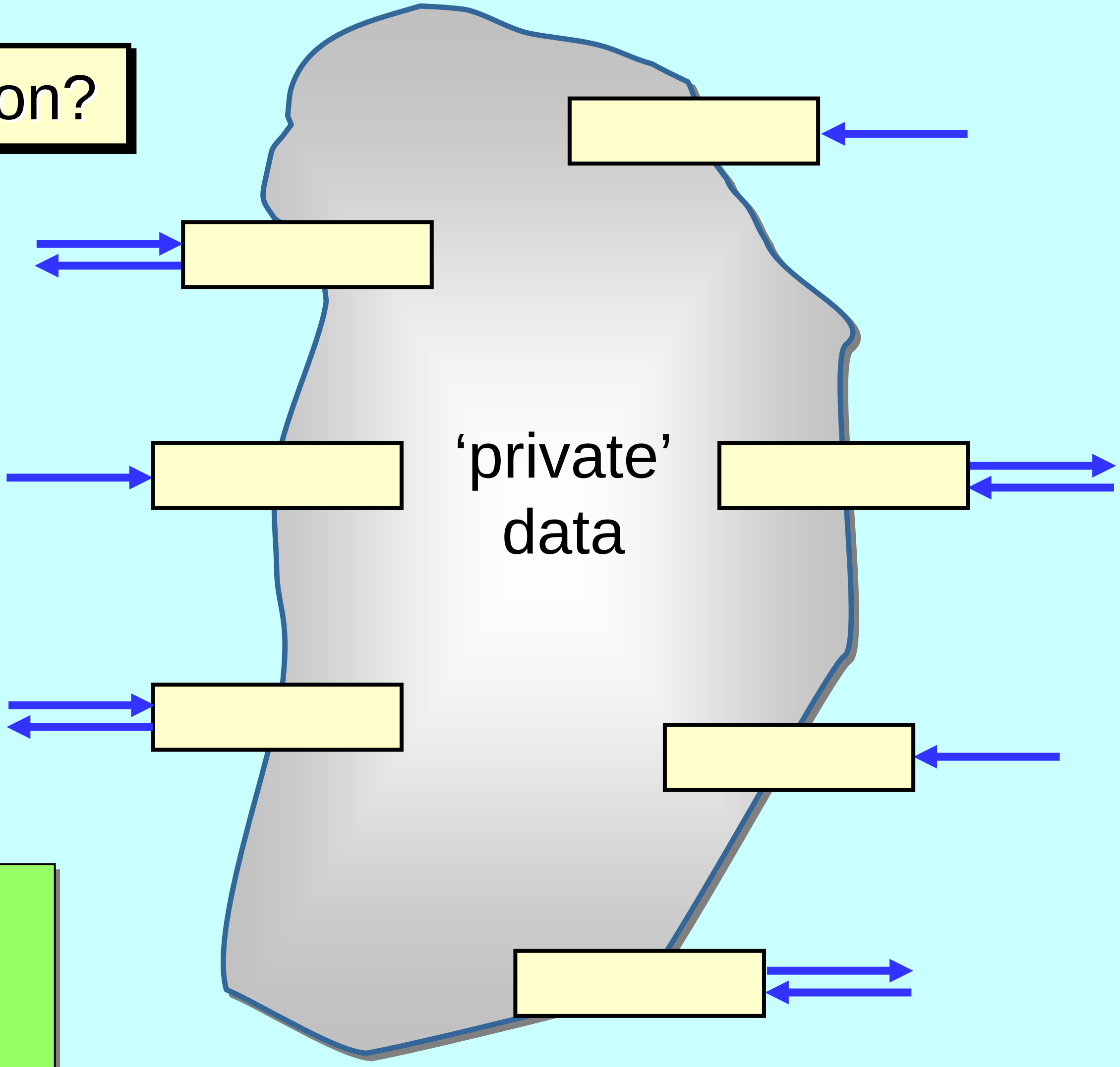
Invariants must also be established before any call-out … in case of call-back! ☹

Who does this?



A Generation Lost in Space …

And object orientation?



Class invariants:

Invariants must also be established before any call-out … in case of call-back! ☹

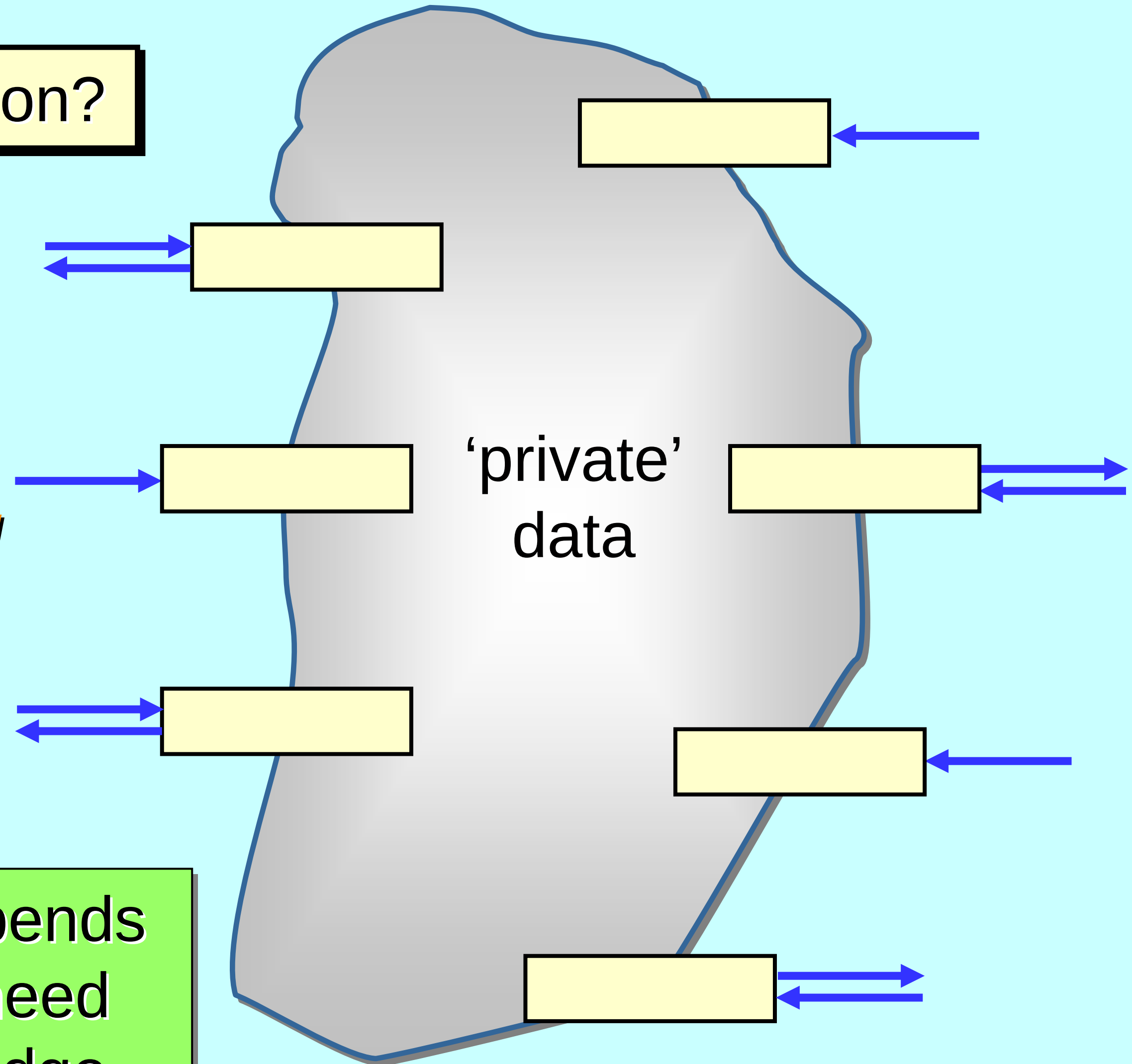
But not doing this is safe only ***if no call-backs happen*** …

A Generation Lost in Space …

And object orientation?

Class invariants:

Invariants must also be established before any call-out … in case of call-back! ☹



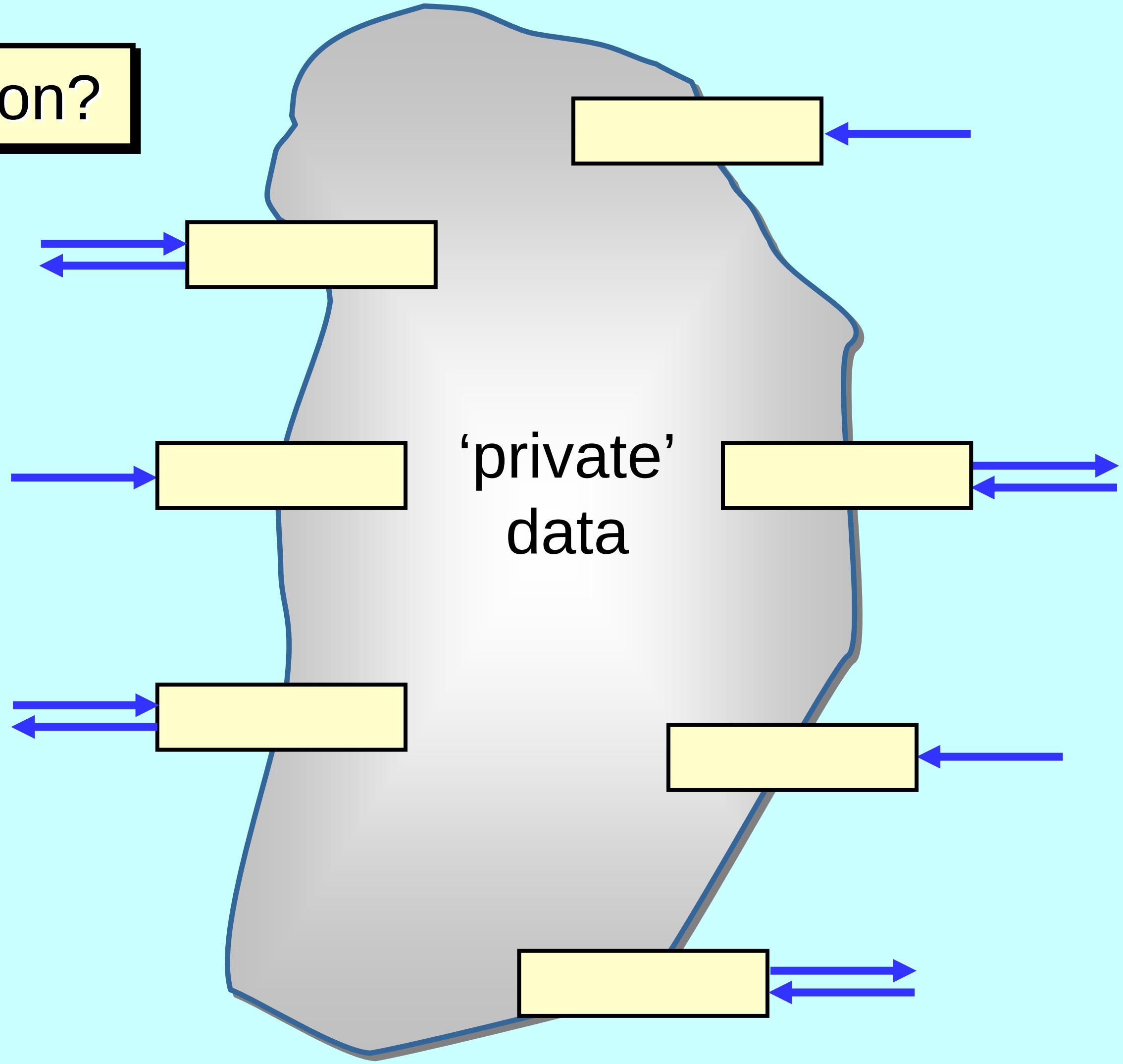
Object semantics depends on its context … we need *whole-system* knowledge.

A Generation Lost in Space ...

And object orientation?

What we tell our students:

> *"An opaque object interacting with a wider system of objects via its formal public interface."*



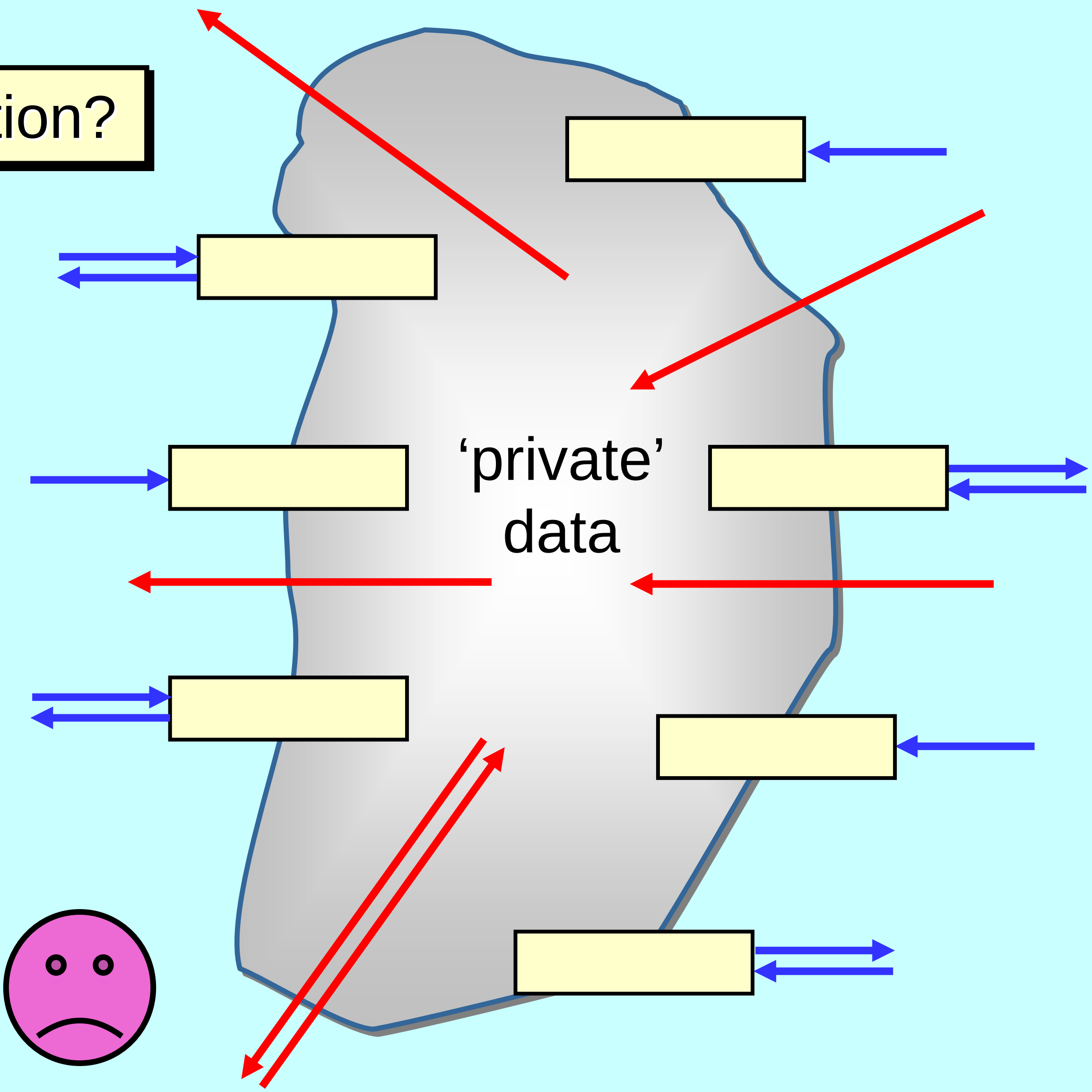
A Generation Lost in Space …

And object orientation?

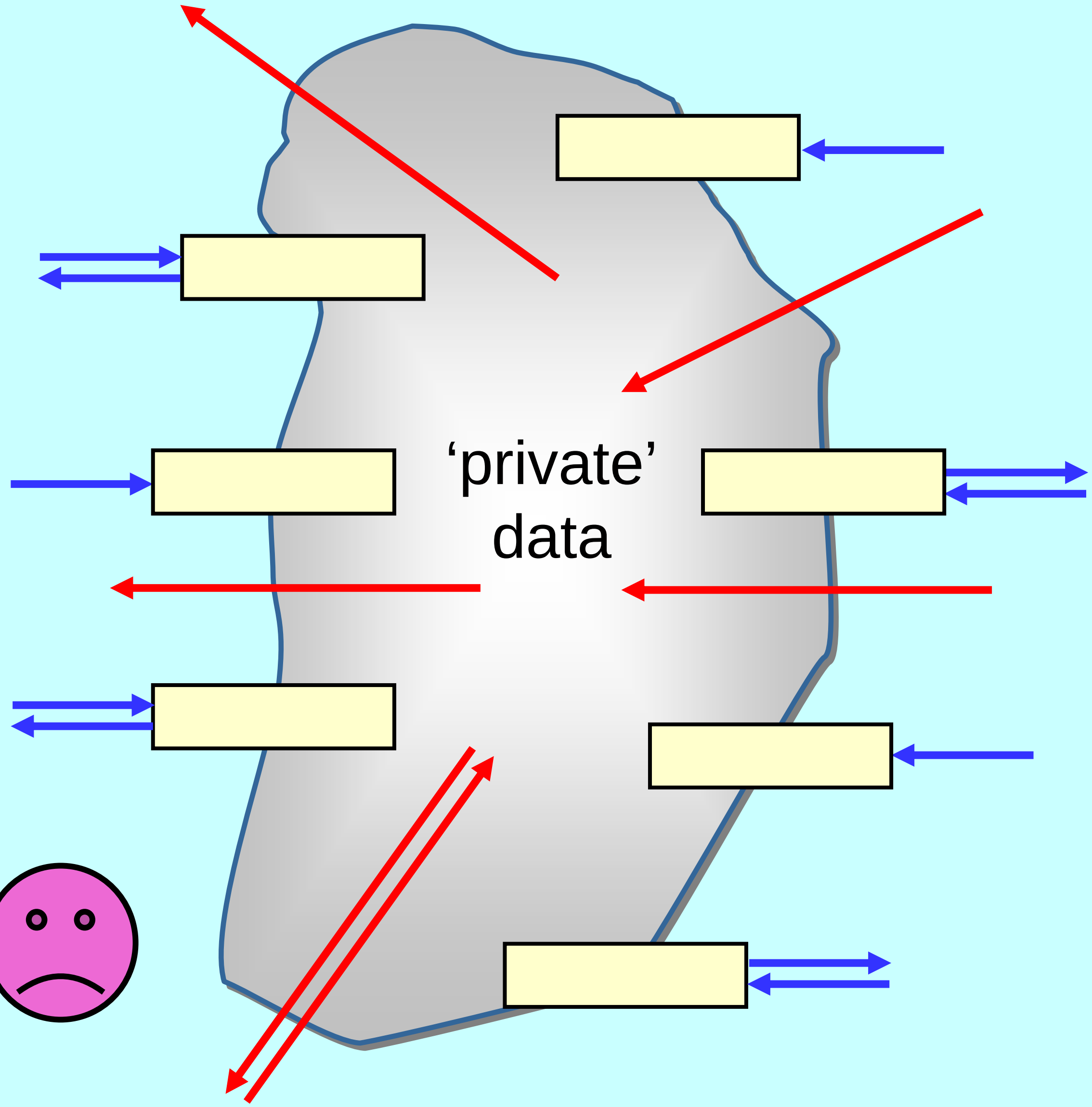
The truth:

"Undeclared interactions between the object and other parts of the system …"

Documentation = Source Code



A Generation Lost in Space …



It gets worse …

Lost in Space ...

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

A `Counter` object maintains an integer `count`, initially zero. It has reference to a `Logger` object.

Its `increment` method adds one to `count` and `log`s the result with `Logger`.

Its `getCount` method returns the current `count`.

Lost in Space …

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

So, if a `Counter` is holding a `count` value of 42 and its `increment` method is invoked, its `count` becomes 43 … ???

We can't say … !!!
☹ ☹ ☹

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

Lost in Space …

Local reasoning is not enough …

Can the value of `count` change?

Better take a look at **`Logger`** …

```
public class Logger {

  public void log (int count)
  {
    System.out.println ("Log " + System.currentTimeMillis () +
                        ": " + count);
  }

}
```

That looks safe enough …

But what if `Counter` were given a *sub-class* of `Logger` … ???

A *sub-class* could override the `log` method to do anything (like *call-back* to `Counter`) …

So knowing `Logger` doesn't help …

Lost in Space ...

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

Local reasoning is not enough …

Can the value of `count` change?

We need *whole-system* knowledge.

Lost in Space ...

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

Local reasoning is not enough …

*Can the value of **count** change?*

The short answer is … YES

☹ ☹ ☹

A Generation Lost in Space …

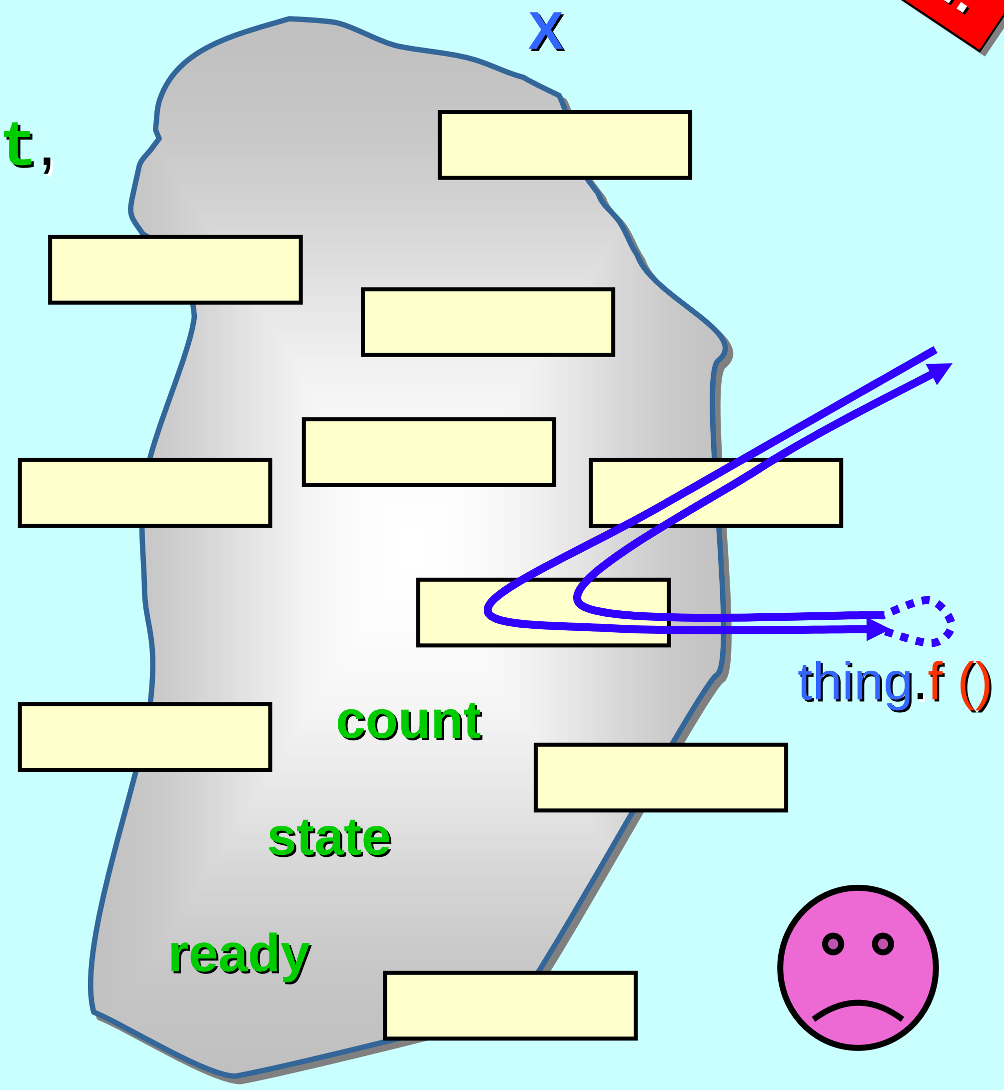
Here's how …

Suppose class **X** has a *private* integer field, **`count`**, and *private* methods that see and change it.

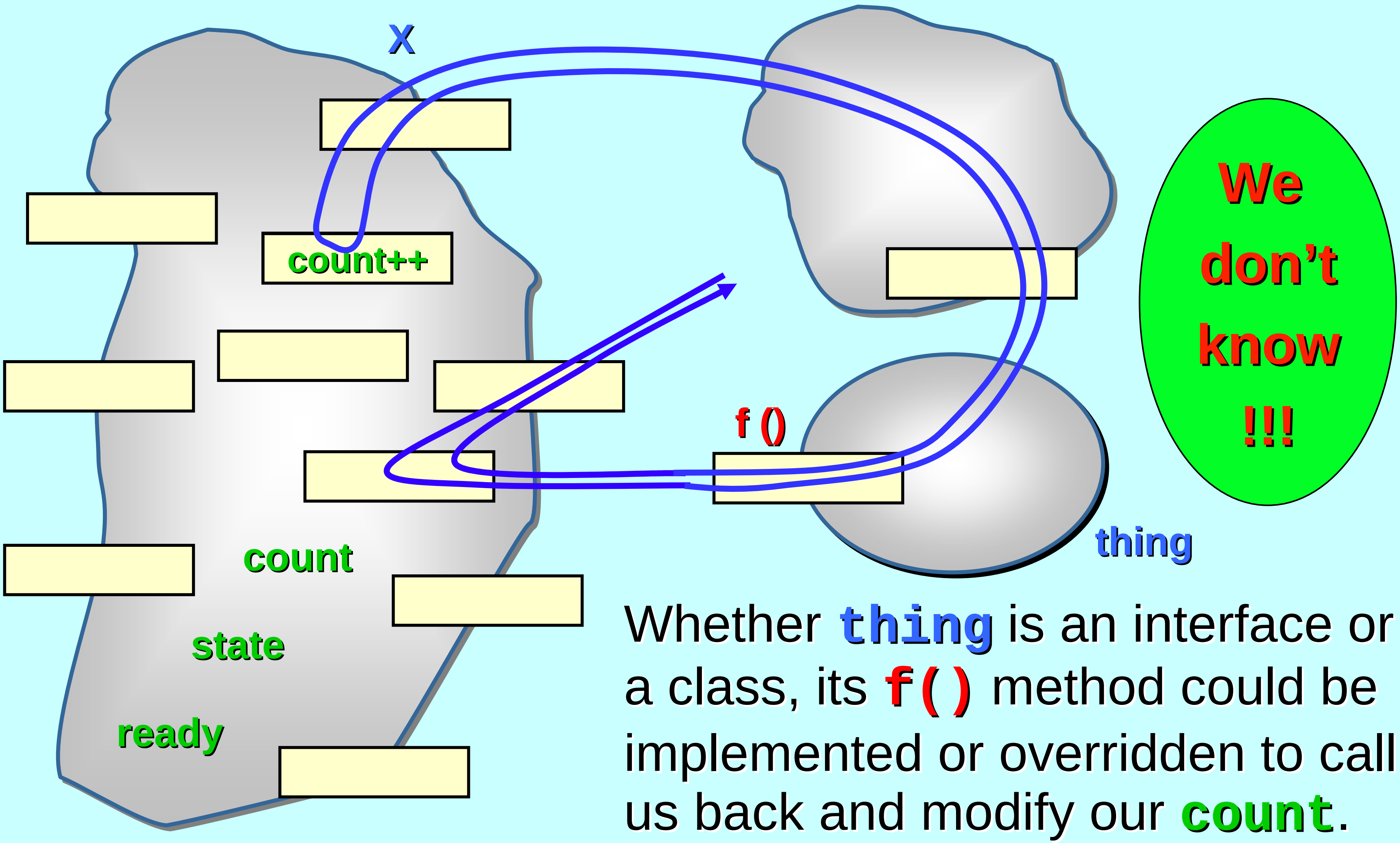
Suppose the following code occurs in one of those methods:

```
count = 42; thing.f();
```

What is the value of **`count`** after these two statements?

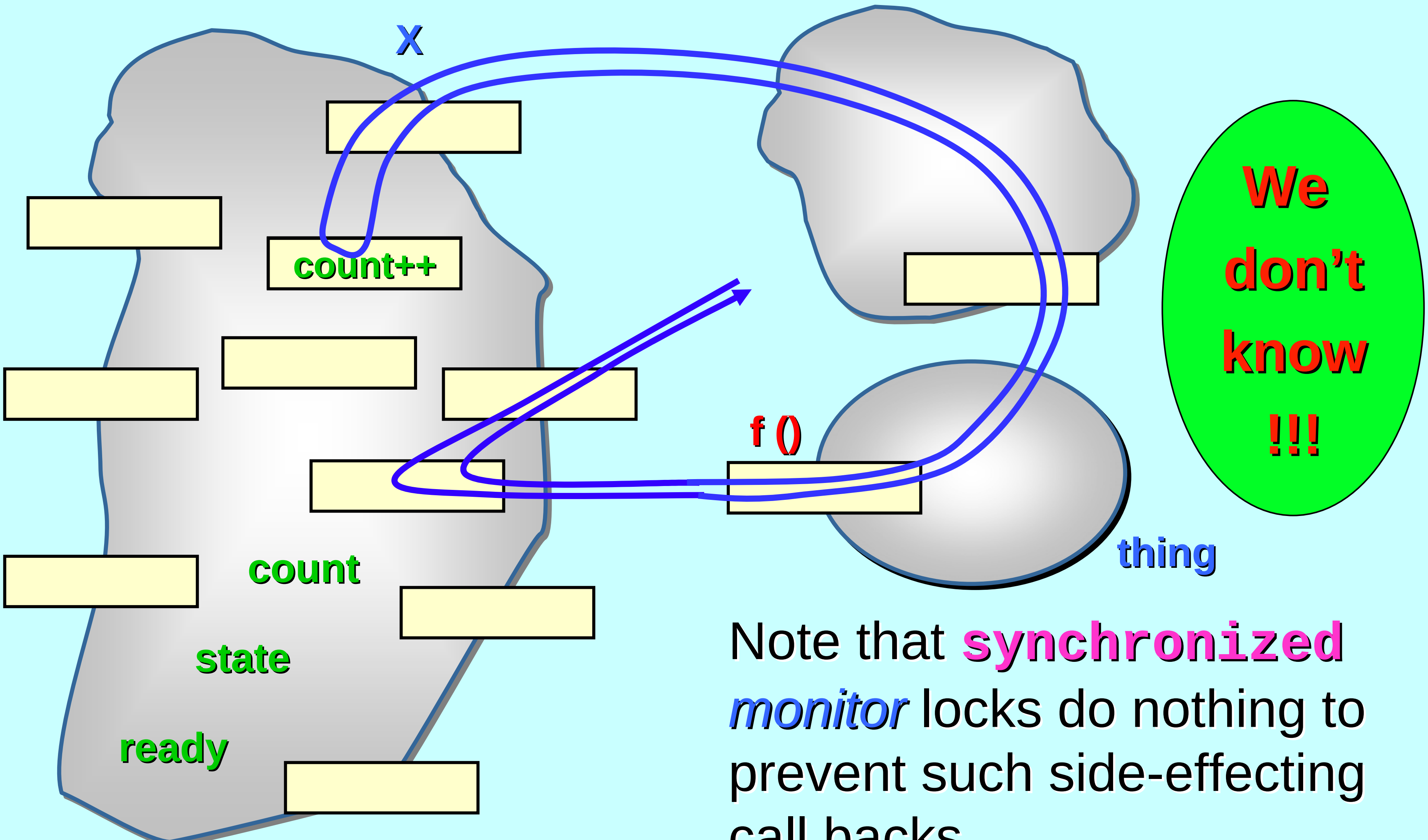


A Generation Lost in Space …



Whether **thing** is an interface or a class, its **f()** method could be implemented or overridden to call us back and modify our **count**.

A Generation Lost in Space …



Note that **synchronized** *monitor* locks do nothing to prevent such side-effecting call backs …

A Generation Lost in Space ...

We don’t know the value of **`count`** after the following two statements:

```
count = 42; thing.f();
```

This lack of ability to reason locally about local data is strangely familiar. In the bad old days, free use of global variables led us into exactly the same mess.

Structured programming led us out of that mire. Did *object orientation* just take us back in?

A Generation Found ...

What is the value of **`count`** after these two statements?

```
SEQ
  count := 42
  thing ! anything
```

This time we do know. *What-you-see-is-what-you-get*. The answer is **`42`**.

The only way **`count`** can be changed is if *this process* changes it - and it doesn’t! Local analysis is sufficient. We don’t need to worry about what lies beyond the **`thing`** channel. Our intuitive understanding about the sequence of instructions is always honoured.

Lost in Space ...

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger)
  {
    this.logger = logger;
  }

  public void increment ()
  {
    count++;
    logger.log (count);
  }

  public int getCount ()
  {
    return count;
  }

}
```

Local reasoning is not enough …

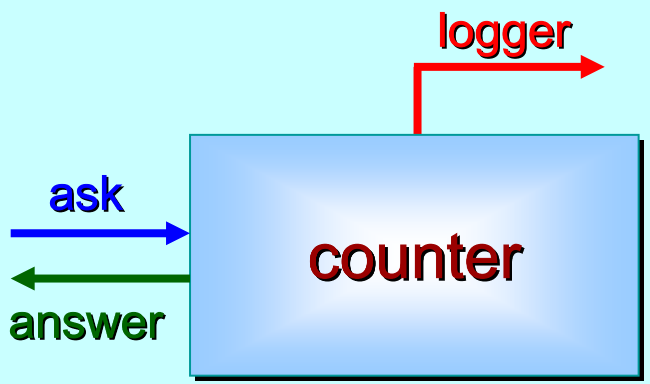
Can the value of count change?

The short answer is … YES ☹ ☹ ☹

```
PROTOCOL COUNTER.ASK
  CASE
    increment
    get.count
:

PROTOCOL COUNTER.ANSWER IS INT:

PROTOCOL LOGGER IS INT:
```

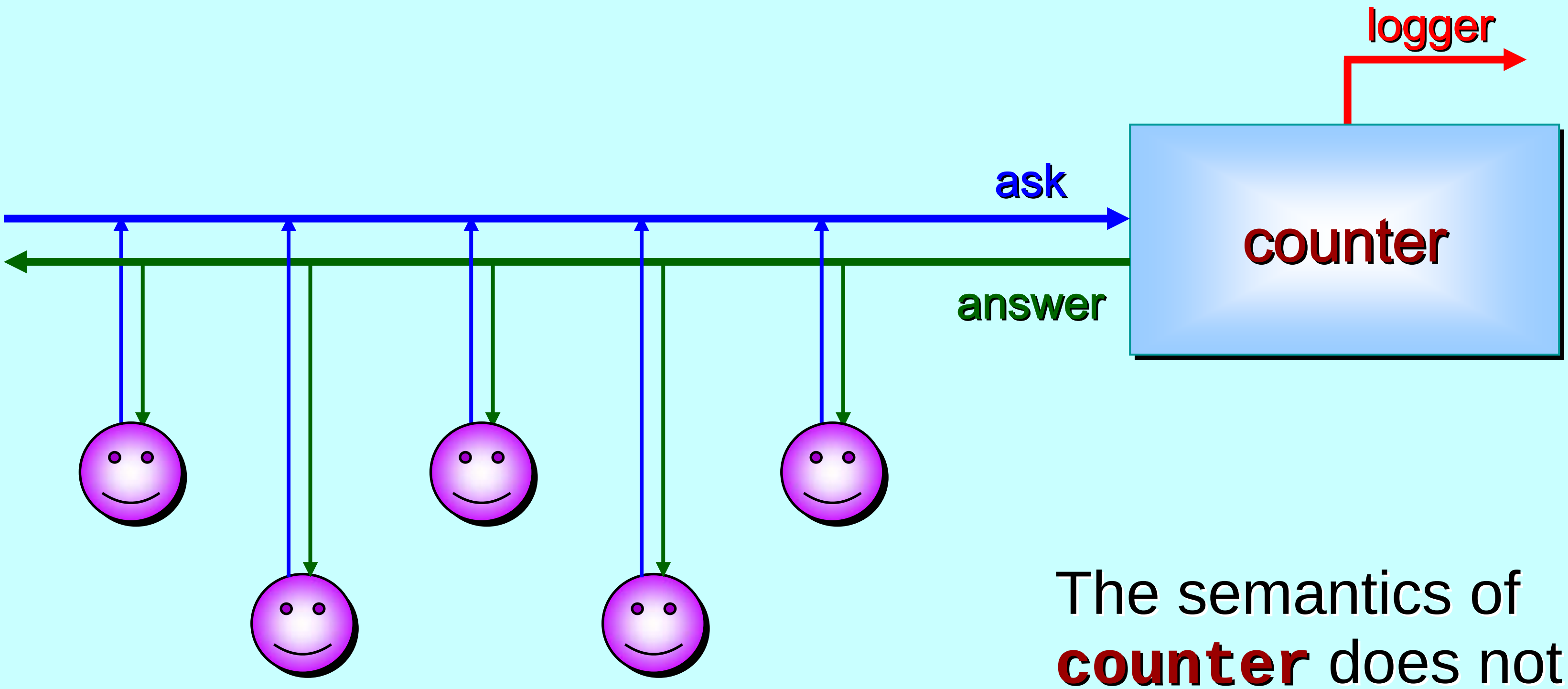



```
PROC counter (CHAN COUNTER.ASK ask?,
              CHAN COUNTER.ANSWER answer!,
              CHAN LOGGER logger)
  INITIAL INT count IS 0:
  WHILE TRUE
    ask ? CASE
      increment
        SEQ
          count := count + 1
          logger ! count
      get.count
        answer ! count
:
```

Local reasoning is enough … ***what you see is what you get***.

Can the value of count change?

NO ☺ ☺ ☺

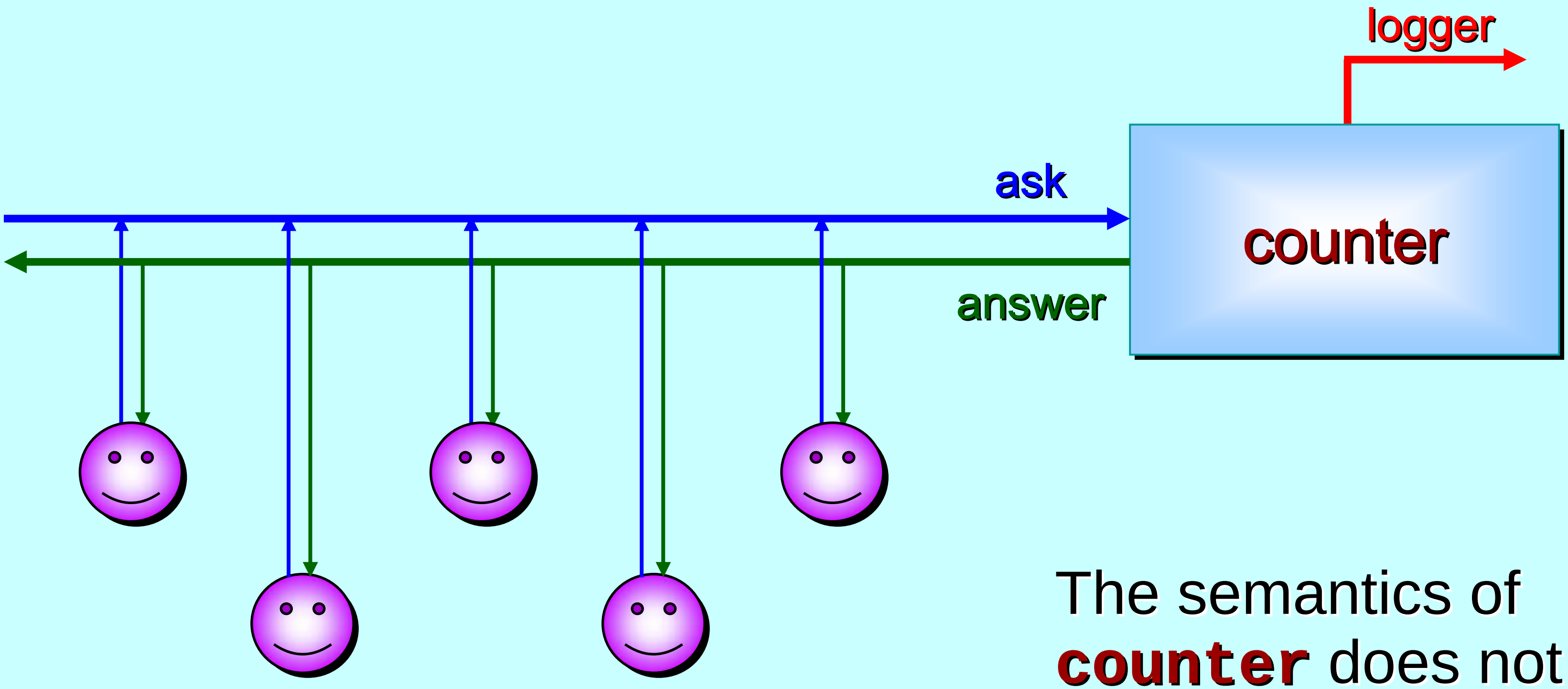



```
SHARED ! CHAN COUNTER.ASK ask:
SHARED ? CHAN COUNTER.ANSWER answer:
SHARED ! CHAN LOGGER logger:

PAR
  counter (ask?, answer!, logger!)
  PAR i = 0 FOR 5
    smiley (ask!, answer?)
```

The semantics of **`counter`** does not depend on context.

Even if the **`logger`** process is one of its clients, it makes no difference to **`counter`**.

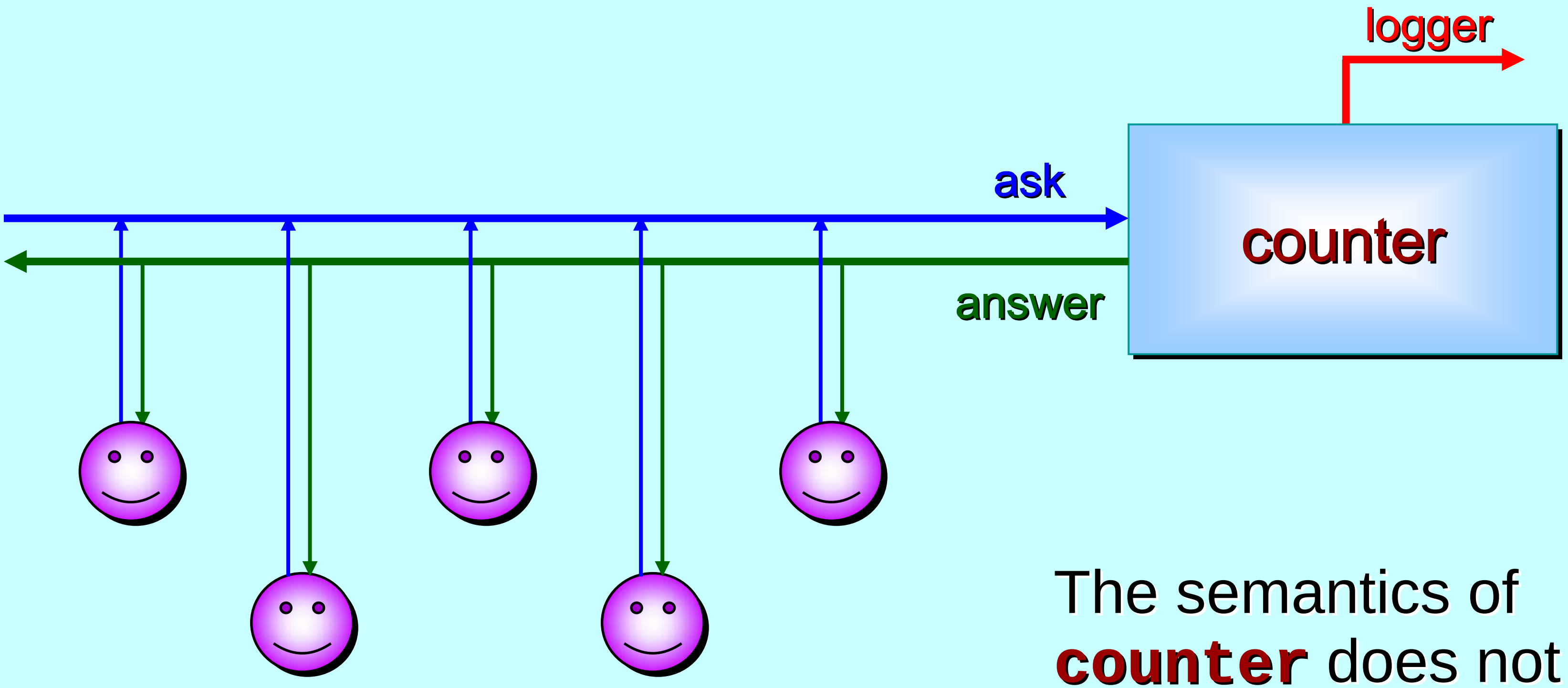



```
SHARED ! CHAN COUNTER.ASK ask:
SHARED ? CHAN COUNTER.ANSWER answer:
SHARED ! CHAN LOGGER logger:

PAR
  counter (ask?, answer!, logger!)
  PAR i = 0 FOR 5
    smiley (ask!, answer?)
```

The semantics of **`counter`** does not depend on context.

If the **`logger`** process calls back, its request will be queued until **`counter`** can take it.

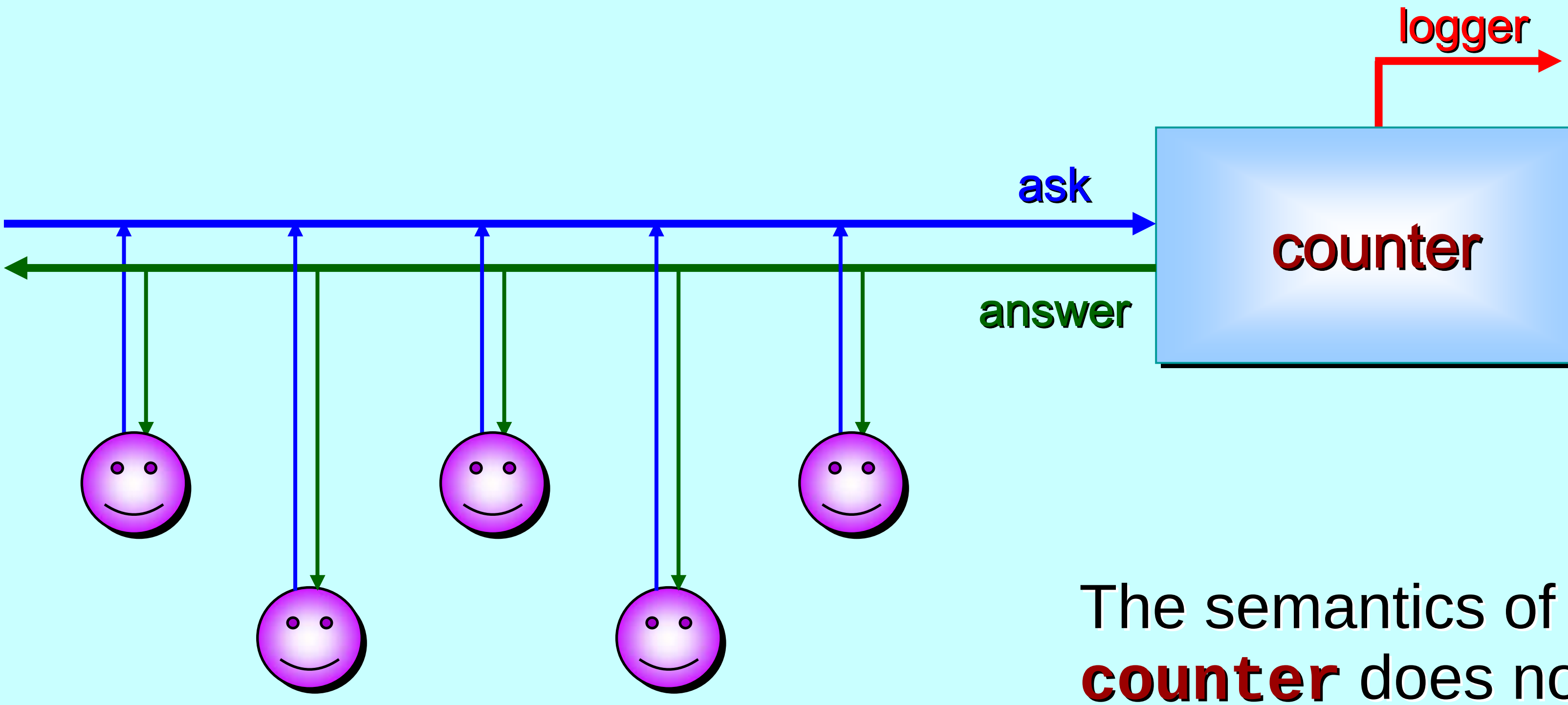



```
SHARED ! CHAN COUNTER.ASK ask:
SHARED ? CHAN COUNTER.ANSWER answer:
SHARED ! CHAN LOGGER logger:

PAR
  counter (ask?, answer!, logger!)
  PAR i = 0 FOR 5
    smiley (ask!, answer?)
```

The semantics of **`counter`** does not depend on context.

If the call-back is part of an *extended input*, there will be deadlock – but **`counter`** semantics has not changed.



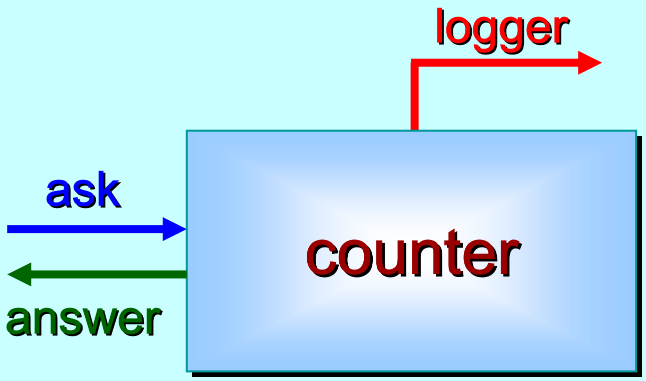
The semantics of **counter** does not depend on context.

If the call-back is part of an *extended input*, there will be deadlock – this is an application error (not a fault with **counter**).

```
SHARED ! CHAN COUNTER.ASK ask:
SHARED ? CHAN COUNTER.ANSWER answer:
SHARED ! CHAN LOGGER logger:

PAR
  counter (ask?, answer!, logger!)
  PAR i = 0 FOR 5
    smiley (ask!, answer?)
```

A Generation Found ...



Local reasoning is all that's needed.

```
PROC counter (CHAN COUNTER.ASK ask?,
              CHAN COUNTER.ANSWER answer!,
              CHAN LOGGER logger)
  INITIAL INT count IS 0:
  WHILE TRUE
    ask ? CASE
      increment
        SEQ
          count := count + 1
          logger ! count
      get.count
        answer ! count
:
```

WYSIWYG

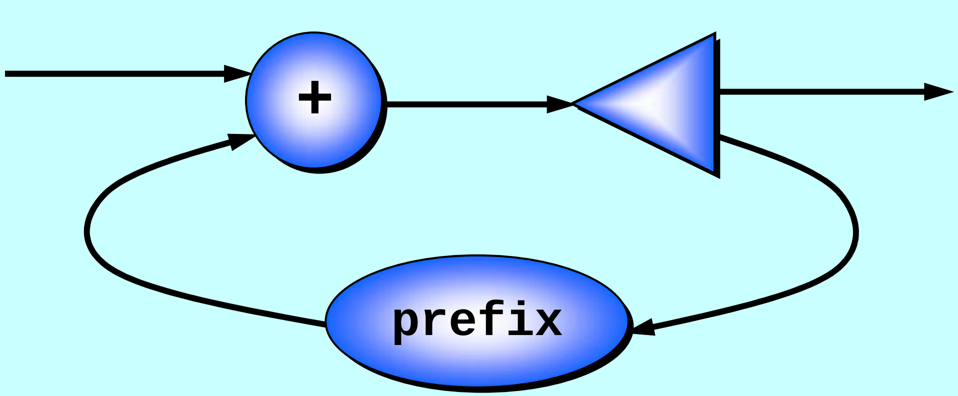
Can the value of count change?

NO ☺ ☺ ☺

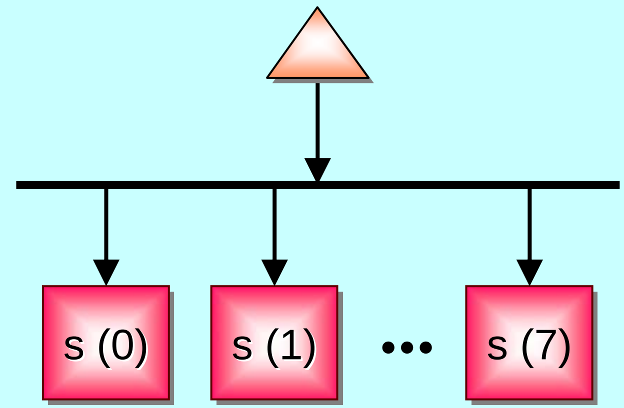
Process Oriented Design

A Diagram Language (in 4 pictures)

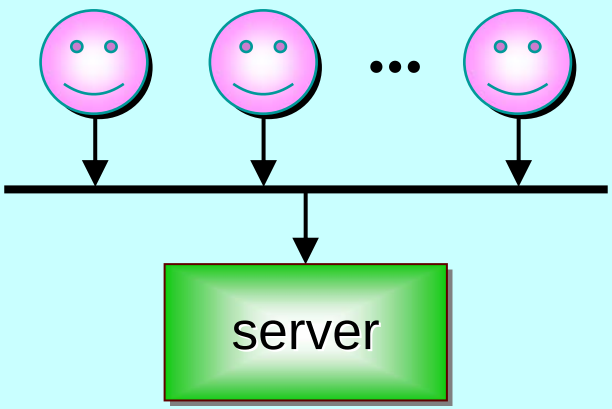
Process Oriented Design (in 4 diagrams)



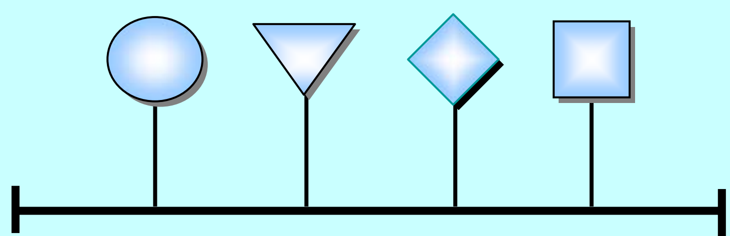
(a) a network of three processes, connected by three internal (hidden) and to two external channels.



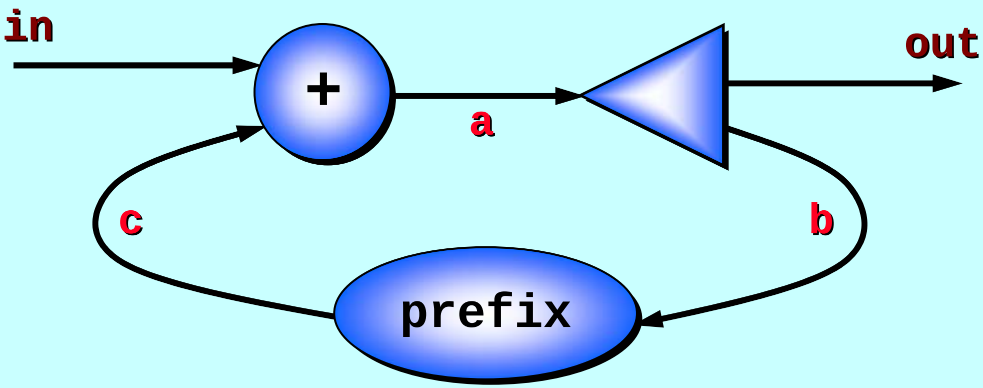
(b) a process writing to a bank of servers sharing the channel reading end – it must not care which one it gets.



(c) a number of client processes sharing the writing end of a channel to a server.

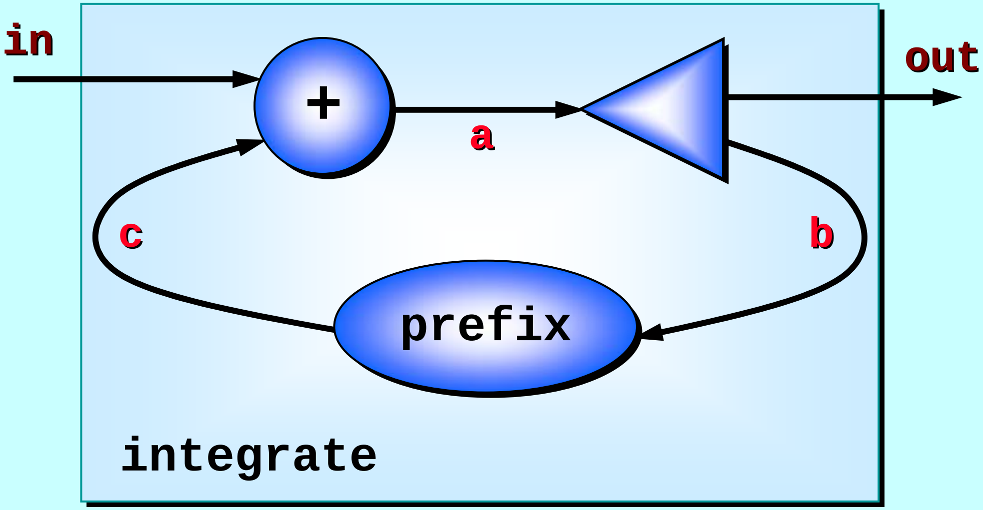


(d) some processes enrolled on a shared barrier – any process synchronising must wait for all to synchronise.



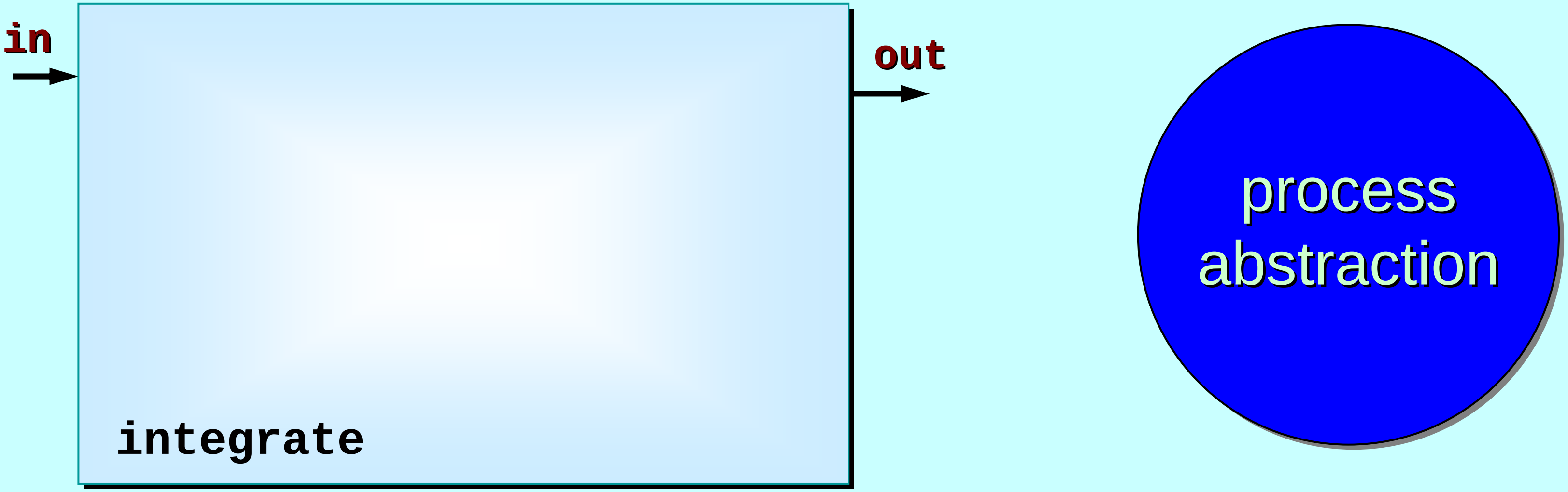
(a) a network of three processes, connected by three internal (hidden) and to two external channels.

```
CHAN REAL64 a, b, c:
PAR
  adder (in?, c?, a!)
  delta (a?, b!, out!)
  prefix (b?, c!)
```



process abstraction

```
PROC integrate (CHAN REAL64 in?, out!)
  CHAN REAL64 a, b, c:
  PAR
    adder (in?, c?, a!)
    delta (a?, b!, out!)
    prefix (b?, c!)
:
```

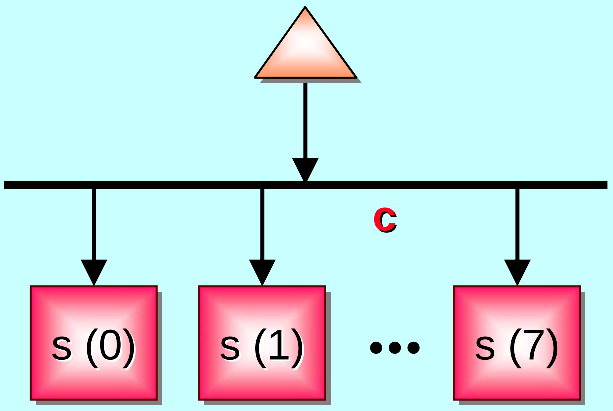



```
PROC integrate (CHAN REAL64 in?, out!)
  CHAN REAL64 a, b, c:
  PAR
    adder (in?, c?, a!)
    delta (a?, b!, out!)
    prefix (b?, c!)
:
```


```
PROC integrate (CHAN REAL64 in?, out!)
```

Like **adder**, **delta** and **prefix** previously, **integrate** is a process that can now be used as a component in another network.

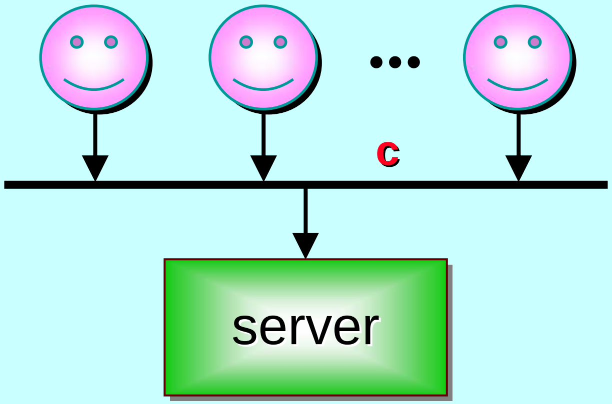
Concurrent systems have structure – networks within networks.
We must be able to express this! And we can … ☺ ☺ ☺



(b) a process writing to a bank of servers sharing the channel reading end – it must not care which one it gets.

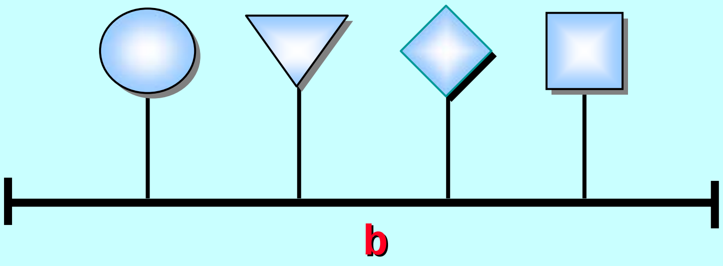
```
SHARED ? CHAN SOME.SERVICE c:
PAR
  triangle (c!)
  PAR i = 0 FOR 8
    s (i, c?)
```

```
SHARED ! CHAN ANOTHER.SERVICE c:
PAR
  PAR i = 0 FOR n.clients
    smiley (i, c!)
  server (c?)
```



(c) a number of client processes sharing the writing end of a channel to a server.

```
BARRIER b:
PAR ENROLL b
  circle (b)
  triangle2 (b)
  diamond (b)
  square (b)
```



(d) some processes enrolled on a shared barrier – any process synchronising must wait for all to synchronise.

CSP Semantics

Traces, failures, divergences and refinement (in 3 slides)

CSP Semantics – Traces (1/3)

An *event* (e.g. *channel communication*, *barrier sync*) happens when, and only when, all processes relevant to it choose to engage.

A process *trace* is a finite sequence of events in which a process *may* engage.

Safety (trace refinement)

P *trace-refines* **Q** means the traces of **P** are also traces of **Q** – anything **P** *may* do, so *may* **Q**. Turning this round, if there is something **Q** *cannot* do, **P** *cannot* do it either. Now, if **Q** is a specification, then **P** is *safe* in the sense that **P** *cannot* exhibit behaviour (presumably 'bad') disallowed by **Q**.

This is not enough – e.g. **STOP** trace-refines anything, since it does nothing; but it's not an acceptable implementation of anything!

CSP Semantics – Failures (2/3)

A process ***state*** is what a process has become after executing one of its traces. An event is ***external*** to a process if other processes may engage on it. A state is ***stable*** if there is no ***internal*** (i.e. hidden) event on which it may engage. A stable resolution of a state is a stable state reached by zero or more internal events only.

A process ***failure*** is a state paired with a set of external events on which a stable resolution of that state refuses to engage.

Liveness (failure refinement)

P ***failure-refines*** **Q** means (**P** trace-refines **Q**) and (the failures of **P** are also failures of **Q**). So, if a *<state, event-set>* is *not* a failure of **Q**, it is *not* a failure of **P** either. Now, if **Q** is a spec, then **P** fulfills its ***liveness*** conditions: if the spec (**Q**) says that in this state you *will react* to one of these events (i.e. there is no failure here), the implementation (**P**) *will react*. (and the reaction will be safe, because of trace refinement).

CSP Semantics – Divergences (3/3)

A process state is ***divergent*** if, from that state, the process *may* engage in an infinite sequence of ***internal*** events (i.e. it *may* forever refuse to engage with its environment). This is usually a bad thing.

Livelock-free (failure-divergence refinement)

P ***failure-divergence-refines*** **Q** means (**P** failure-refines* **Q**) and (the divergences of **P** are also divergences of **Q**). Now, if **Q** is a specification *with no divergences* (which would be usual), then the implementation (**P**) *also has no divergences*.

* a divergent state is unstable but *may* recover to a stable state. However, under failure-divergence semantics, *a divergent state is considered so dangerous* that further consideration of process behaviour is not worth pursuing – i.e. a process in a divergent state, and in all subsequent states, refuses everything.

Dynamic networks (and occam-π)

Emergent engineering:
a generic space-time modelling and
swarm architecture

(in 45 slides)

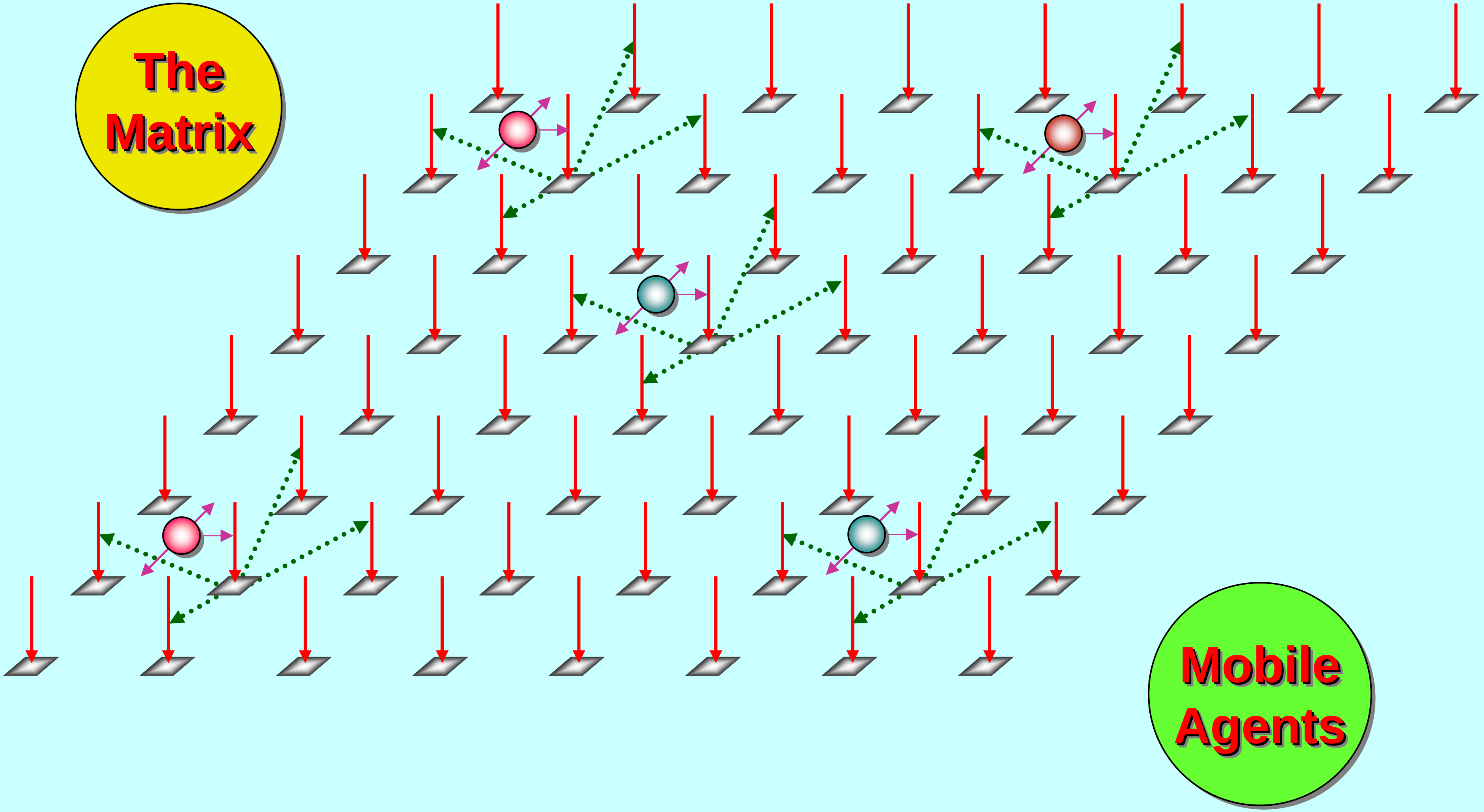
Modelling Bio-Mechanisms

- **In-vivo ⇔ In-silico**
  - One of the UK *'Grand Challenge'* areas.
  - Move *life-sciences* from *description* to *modelling / prediction*.
  - Example: the Nematode worm.
  - Development: from fertilised cell to adult (with virtual experiments).
  - Sensors and movement: reaction to stimuli.
  - Interaction between organisms and other pieces of environment.
- **Modelling technologies**
  - Communicating process networks – fundamentally good fit.
  - Cope with growth / decay, combine / split (evolving topologies).
  - Mobility and location / neighbour awareness.
  - Simplicity, dynamics, performance and safety.
- **occam-π (and JCSP)**
  - Robust and lightweight – good theoretical support.
  - *~10,000,000* processes with useful behaviour in useful time.
  - Enough to make a start …

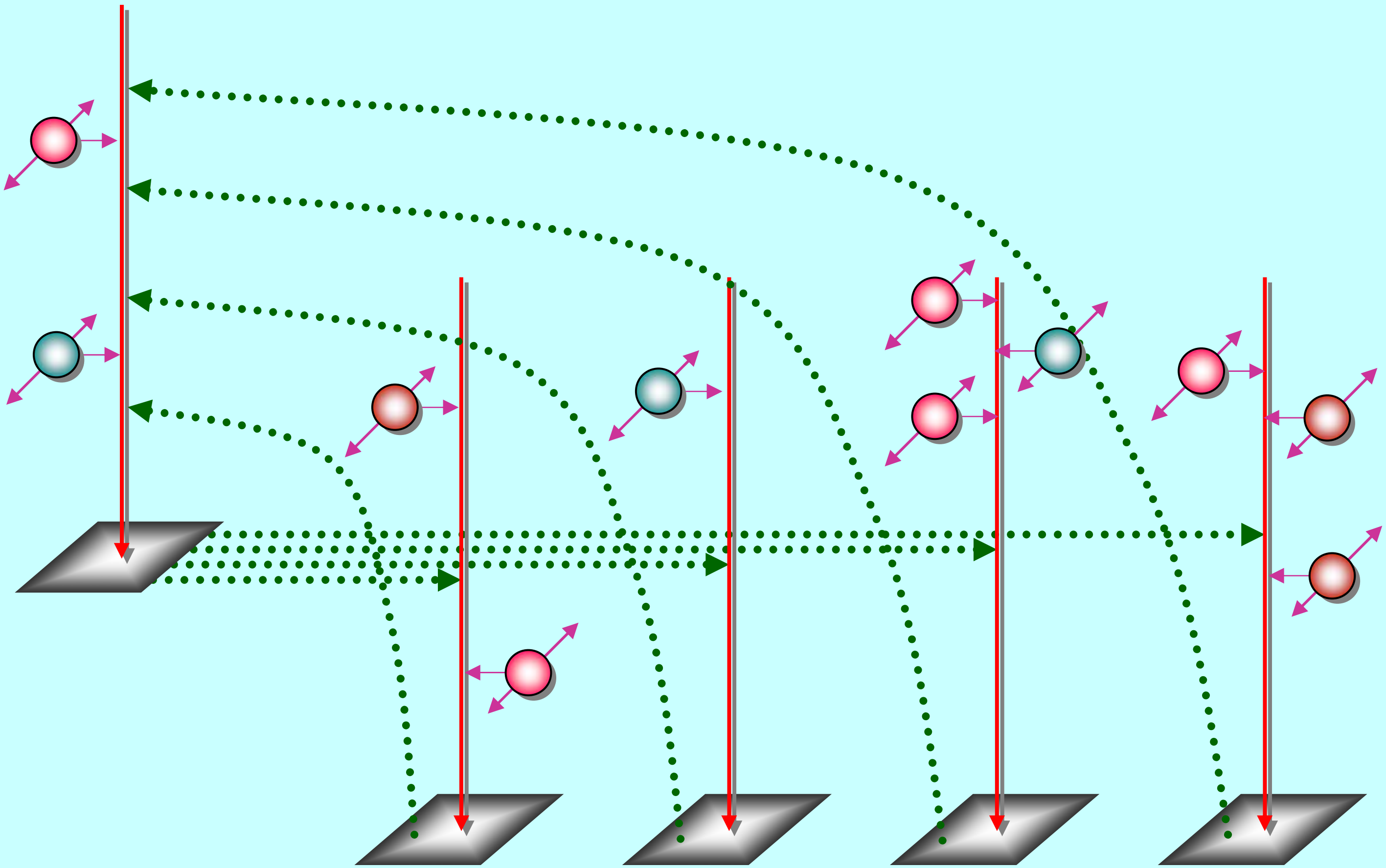
Mobility and Location Awareness

- **Classical communicating process applications**
  - *Static* network structures.
  - *Static* memory / silicon requirements (pre-allocated).
  - Great for hardware design and software for embedded controllers.
  - Consistent and rich underlying theory – *CSP*.
- **Dynamic communicating processes – some questions**
  - *Mutating topologies:* how to keep them safe?
  - *Mobile channel-ends and processes:* dual notions?
  - *Simple operational semantics:* low overhead implementation? ***Yes.***
  - *Process algebra:* combine the best of CSP and the π-*calculus*? ***Yes.***
  - *Refinement:* for manageable system verification … can we keep?
  - *Location awareness:* how can mobile processes know where they are, how can they find each other and link up?
  - *Programmability:* at what level – individual processes or clusters?
  - *Overall behaviour:* planned or emergent?

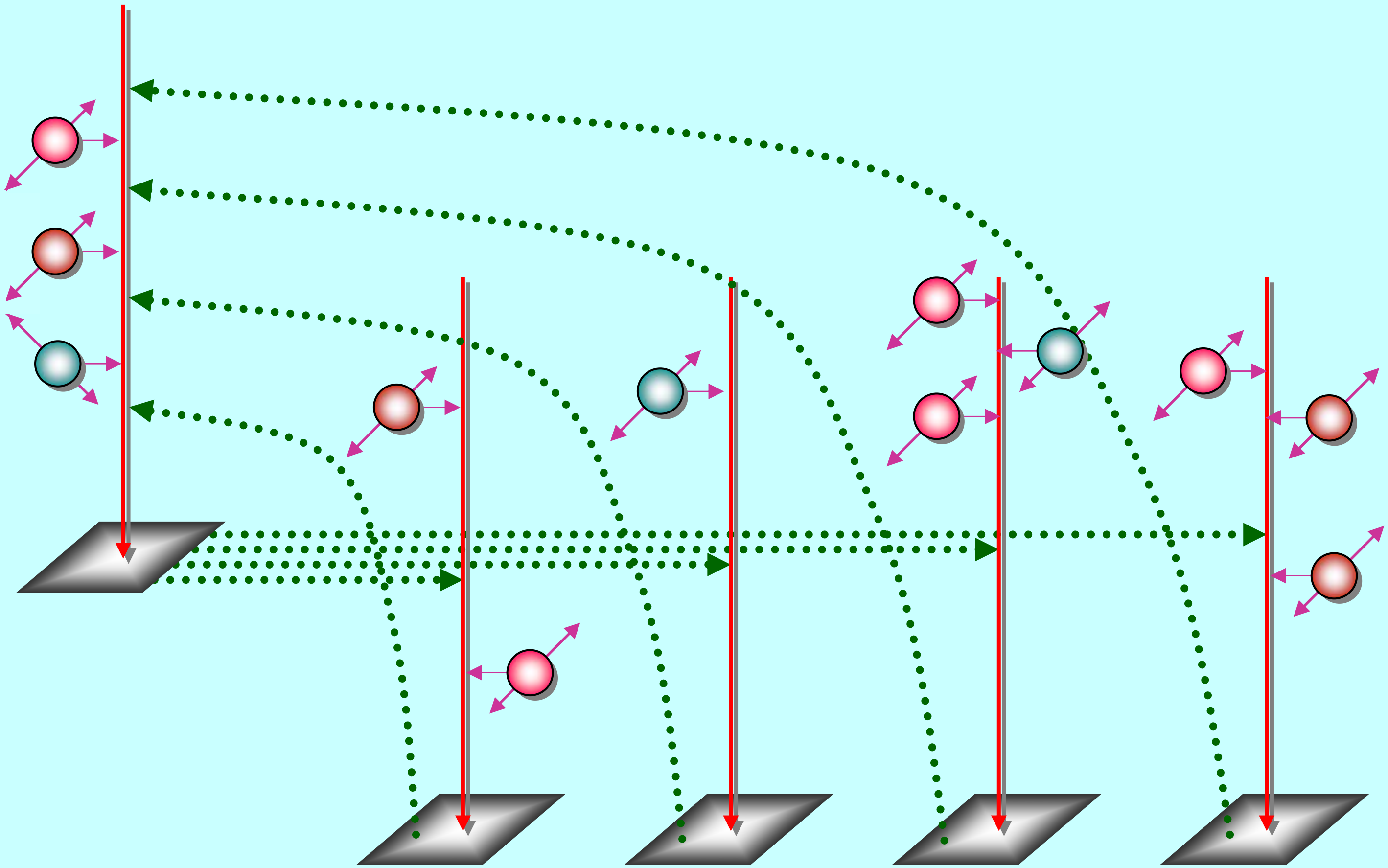
Location (Neighbourhood) Awareness



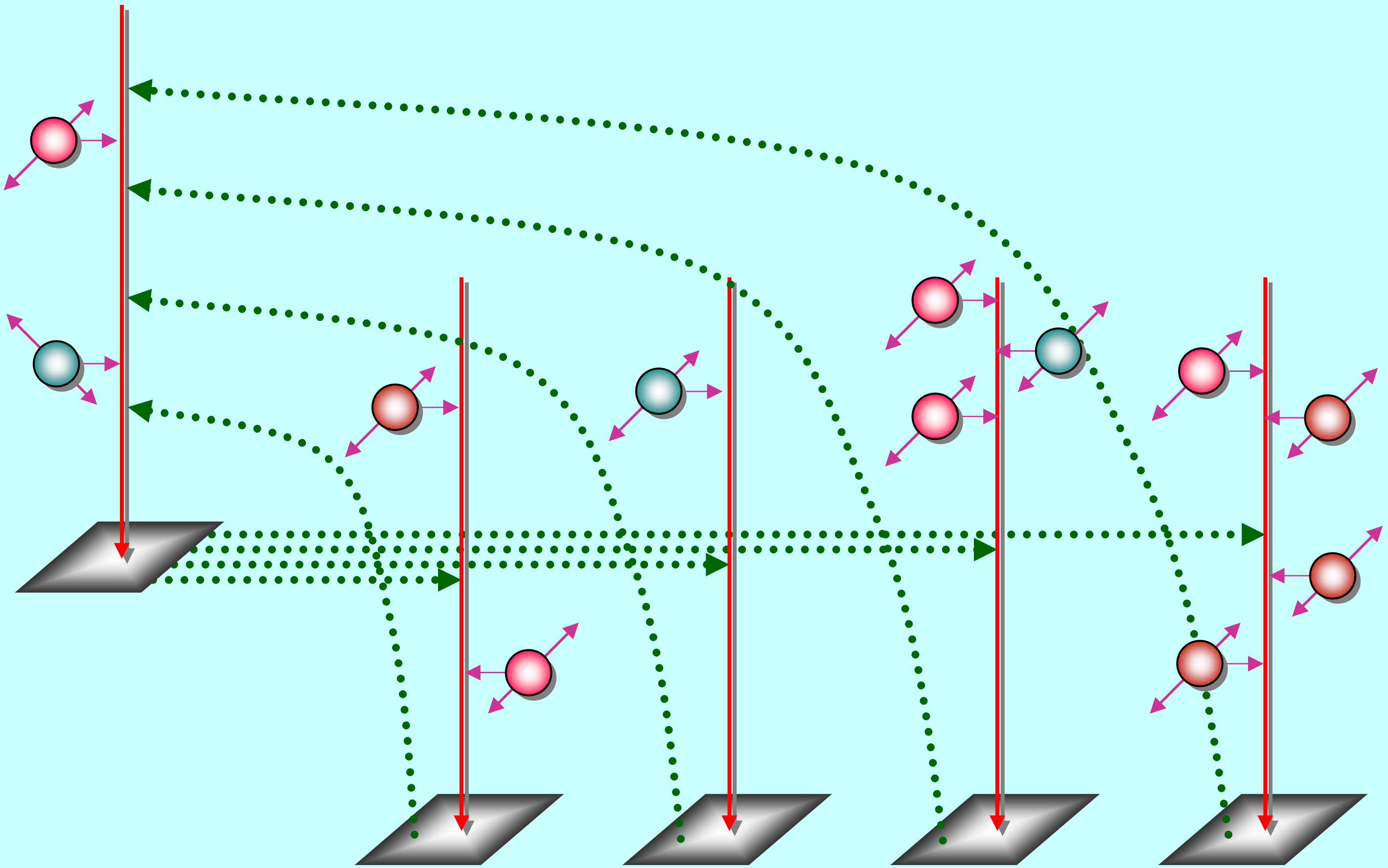
Location (Neighbourhood) Awareness



Location (Neighbourhood) Awareness



Location (Neighbourhood) Awareness



Mobility and Location Awareness

- **The Matrix**
  - A network of (mostly passive) server processes.
  - Responds to client requests from the mobile agents and, occasionally, from *neighbouring* server nodes.
  - Deadlock avoided (in the matrix) *either* by one-place buffered server channels *or* by pure-client slave processes (one per matrix node) that ask their server node for elements (e.g. mobile agents) and forward them to neighbouring nodes.
  - Server nodes only see neighbours, maintain registry of currently located agents (and, maybe, agents on the neighbouring nodes) and answer queries from local agents (including moving them).
- **The Agents**
  - Attached to one node of the Matrix at a time.
  - Sense presence of other agents – on local or neighbouring nodes.
  - Interact with other local agents – must use agent-specific protocol to avoid deadlock. May decide to reproduce, split or move.
  - Local (or global) *sync barriers* to maintain sense of time.

A Thesis and Hypothesis

- **Thesis**
  - Natural systems are concurrent at all levels of scale. Control is devolved. Central command cannot manage the complexity.
  - Natural systems are complex, robust, efficient, long-lived and continuously evolving. ***We should take the hint!***
  - Natural mechanisms should map on to simple engineering principles with low cost and high benefit. Concurrency is a natural mechanism.
  - We should look on ***concurrency*** as a ***core design mechanism*** – not as something difficult, used only to boost performance.
  - Computer science took a wrong turn once. Concurrency should not introduce the algorithmic distortions and hazards evident in current practice. It should ***simplify*** and ***hasten*** the construction, commisioning and maintenance of systems.

- **Hypothesis**
  - The wrong turn can be corrected and this correction is needed now.

Case Study: blood clotting

Haemostasis: we consider a greatly simplified model of the formation of blood clots in response to damage in blood vessels.

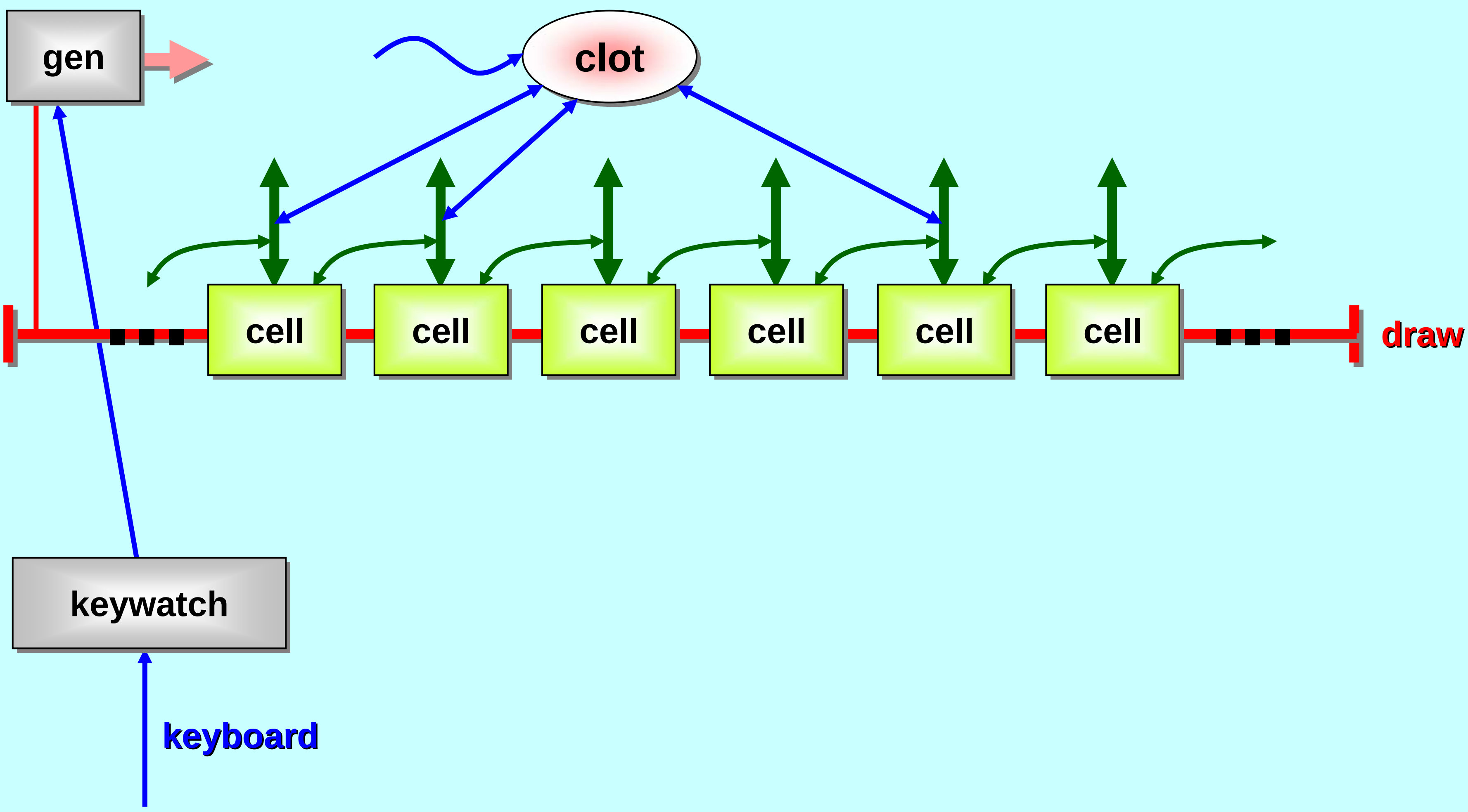
Platelets are passive quasi-cells carried in the bloodstream. They become ***activated*** when a balance between chemical suppressants and activators shift in favour of activation.

When activated, they become ***sticky*** *…*

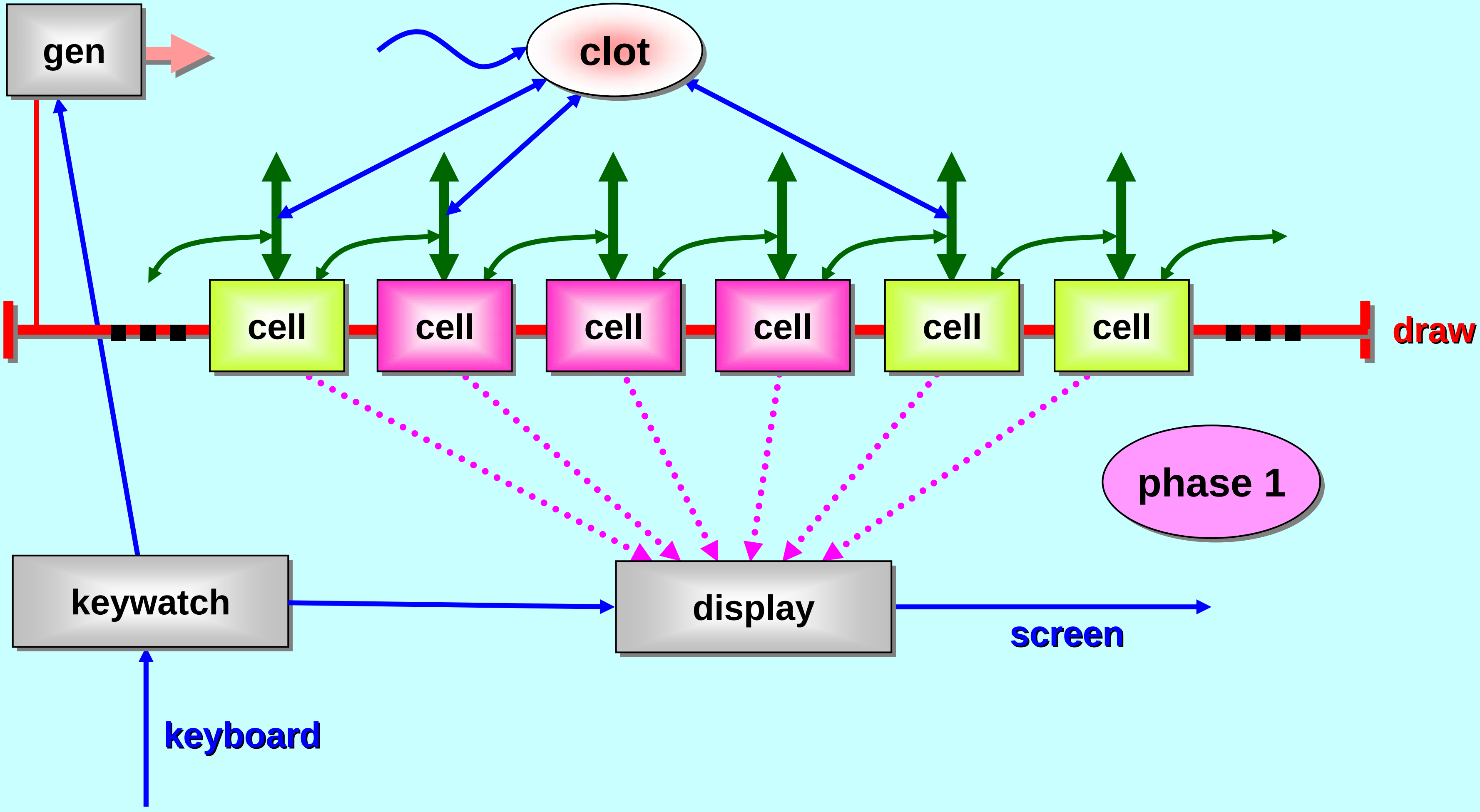
We are just going to model the clumping together of such sticky activated platelets to form ***clots***.

To learn and refine our modelling techniques, we shall start with a simple one-dimensional model of a bloodstream.

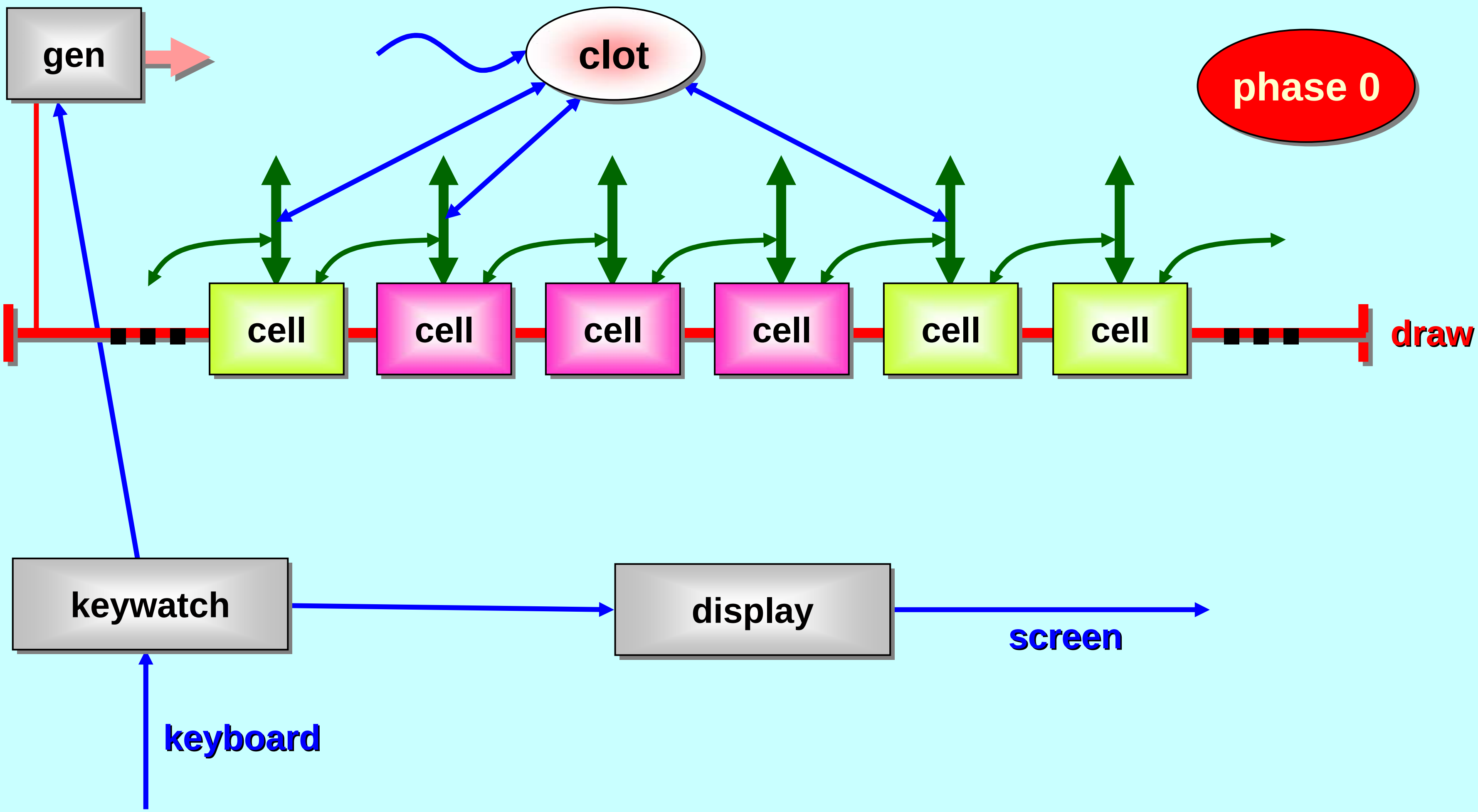
Platelet Model ('lazy' CA)



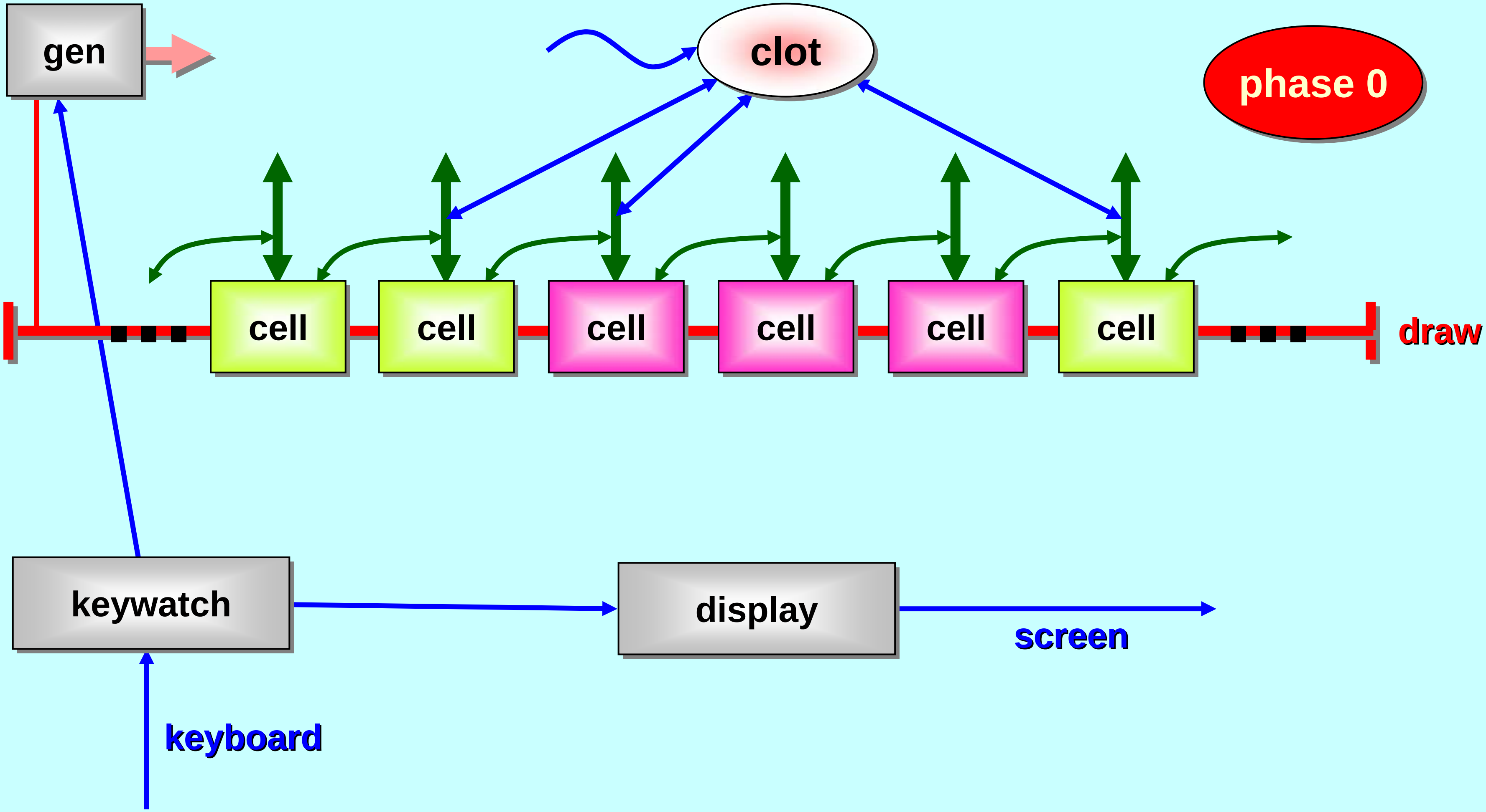
Platelet Model ('lazy' CA)



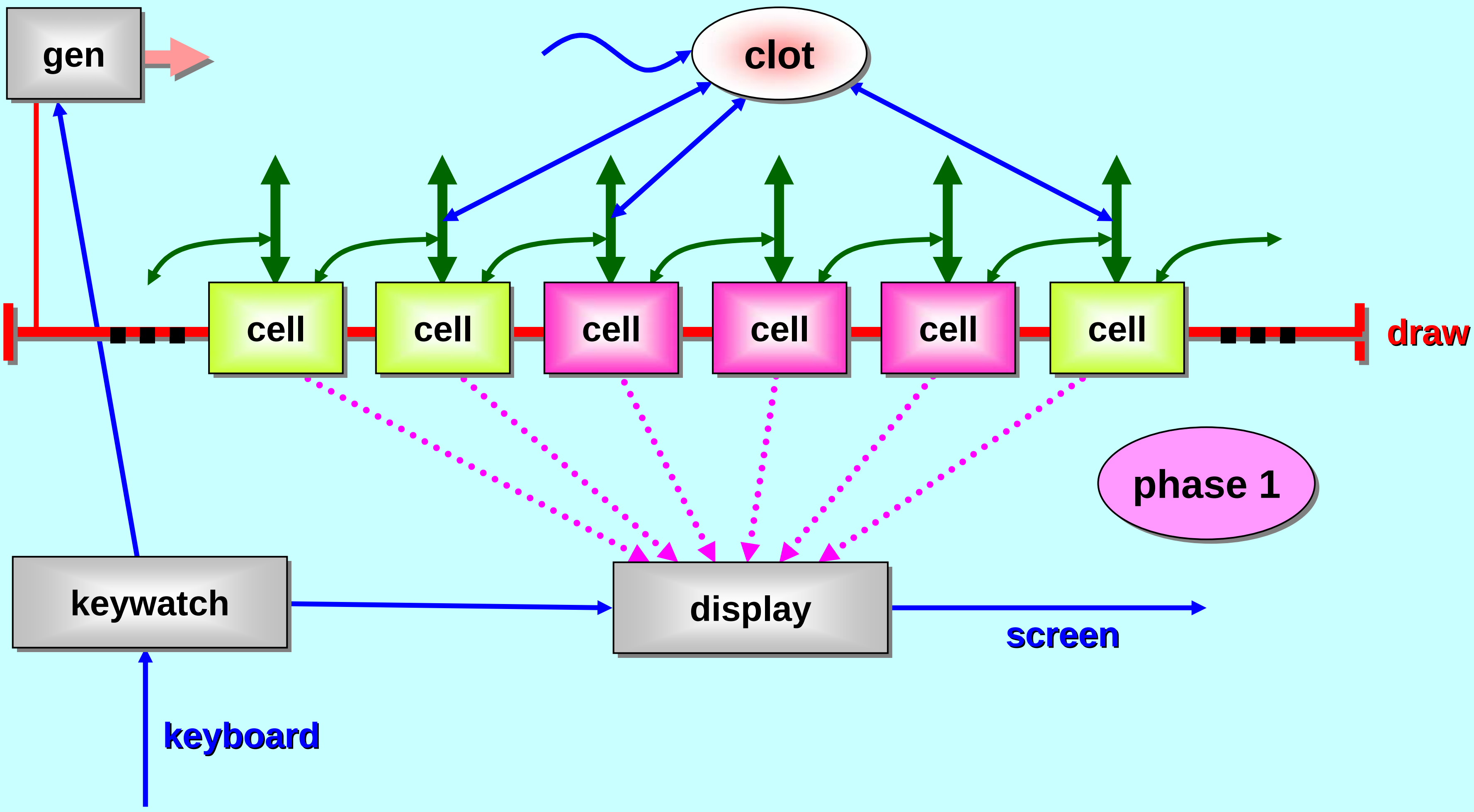
Platelet Model ('lazy' CA)



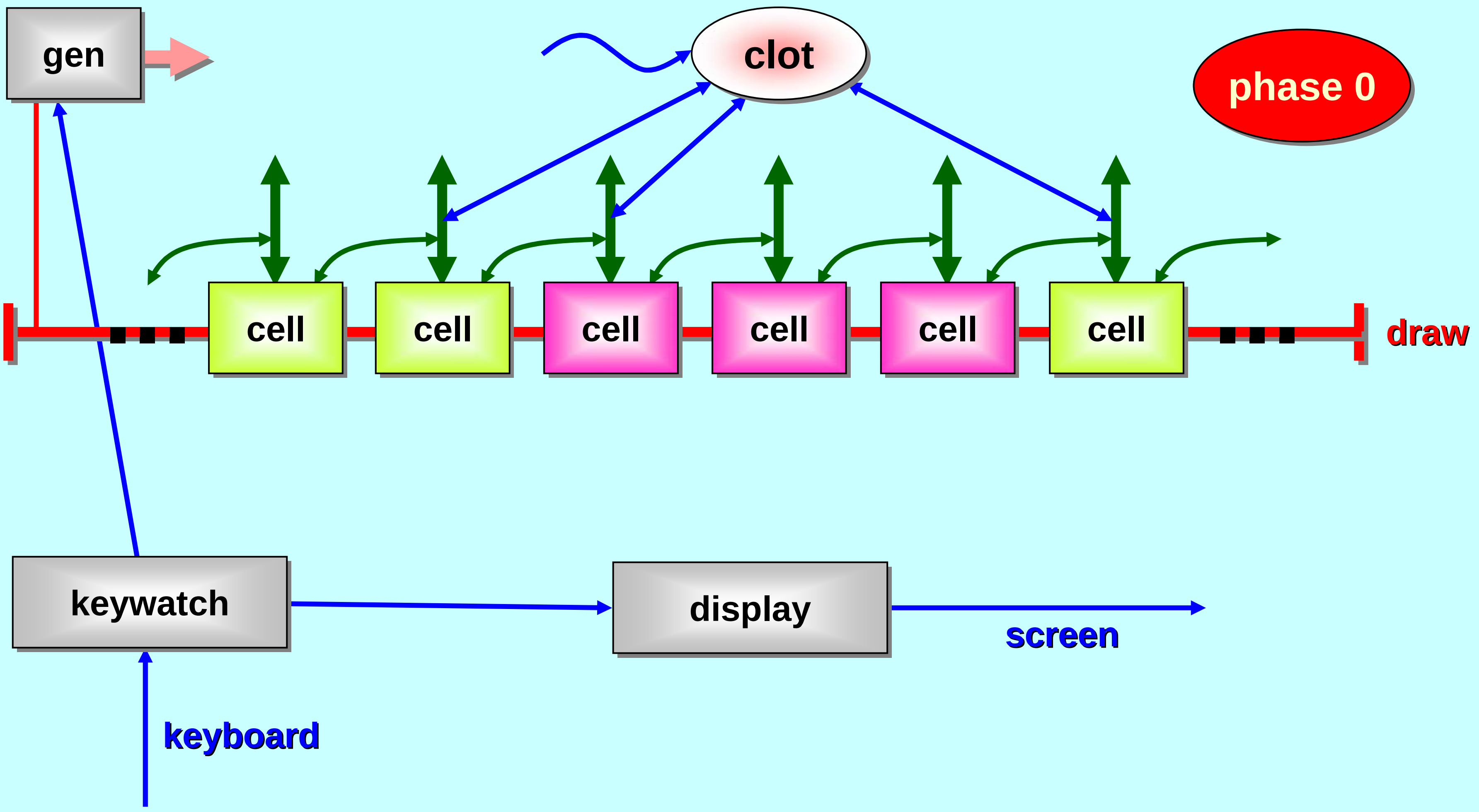
Platelet Model ('lazy' CA)



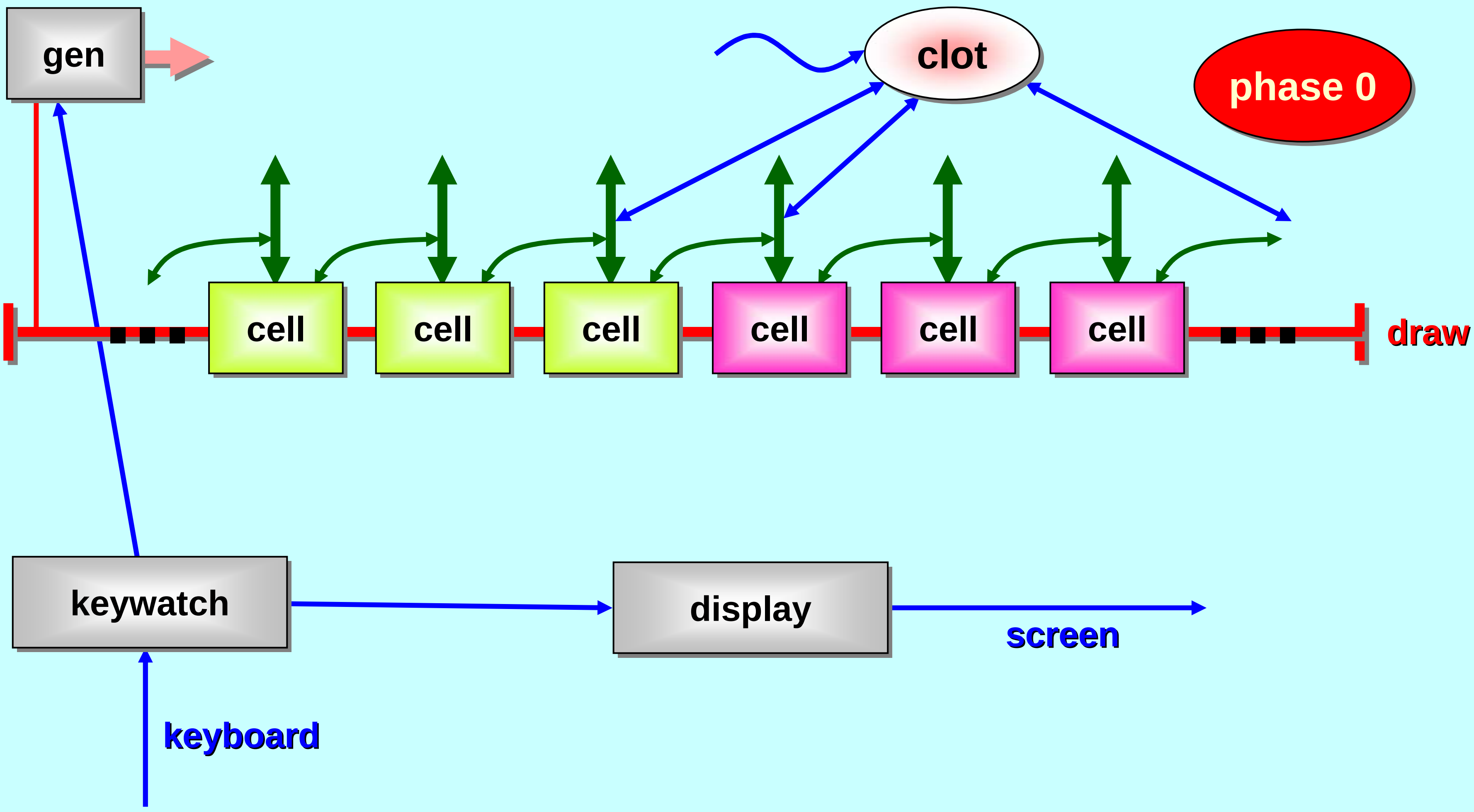
Platelet Model ('lazy' CA)



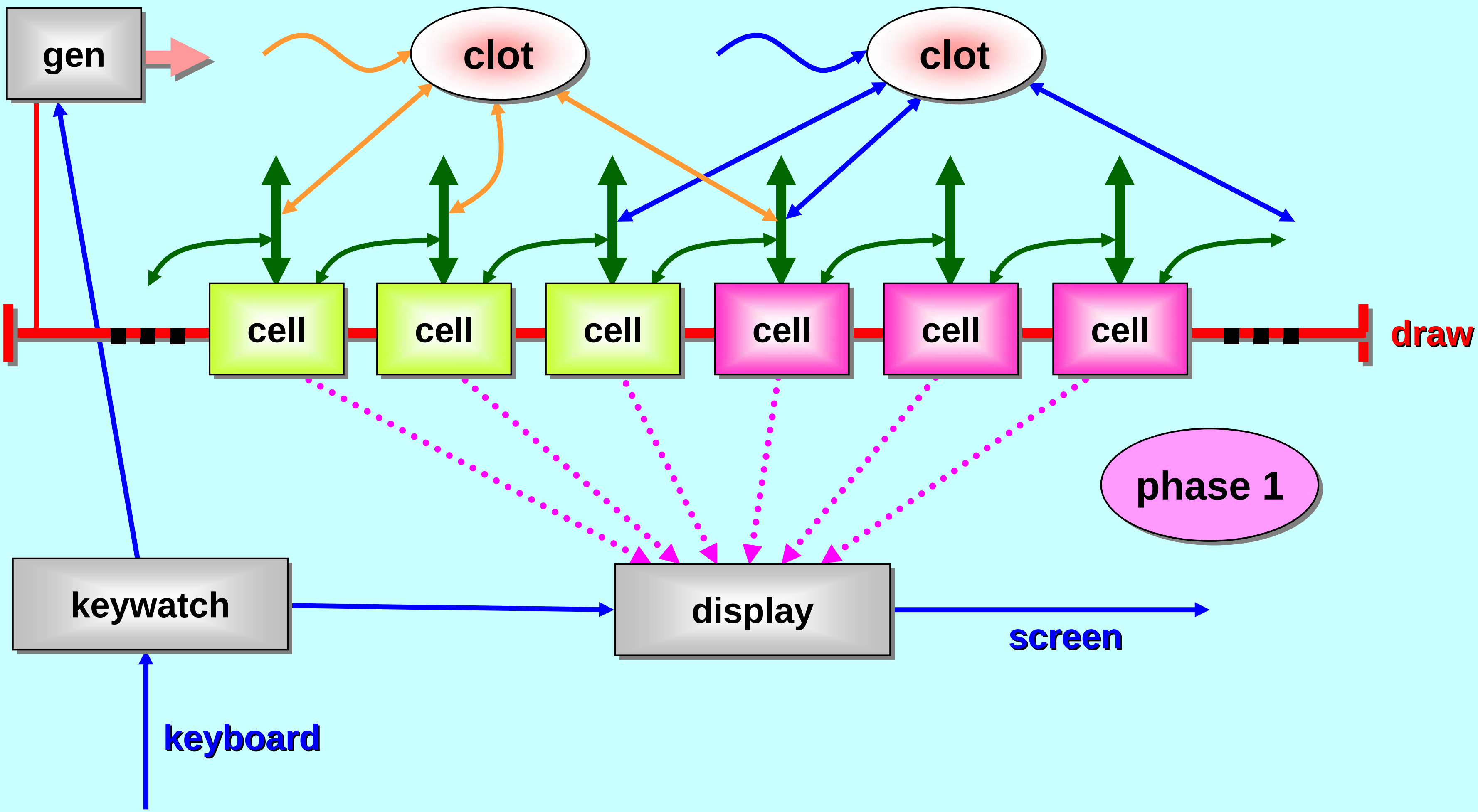
Platelet Model ('lazy' CA)



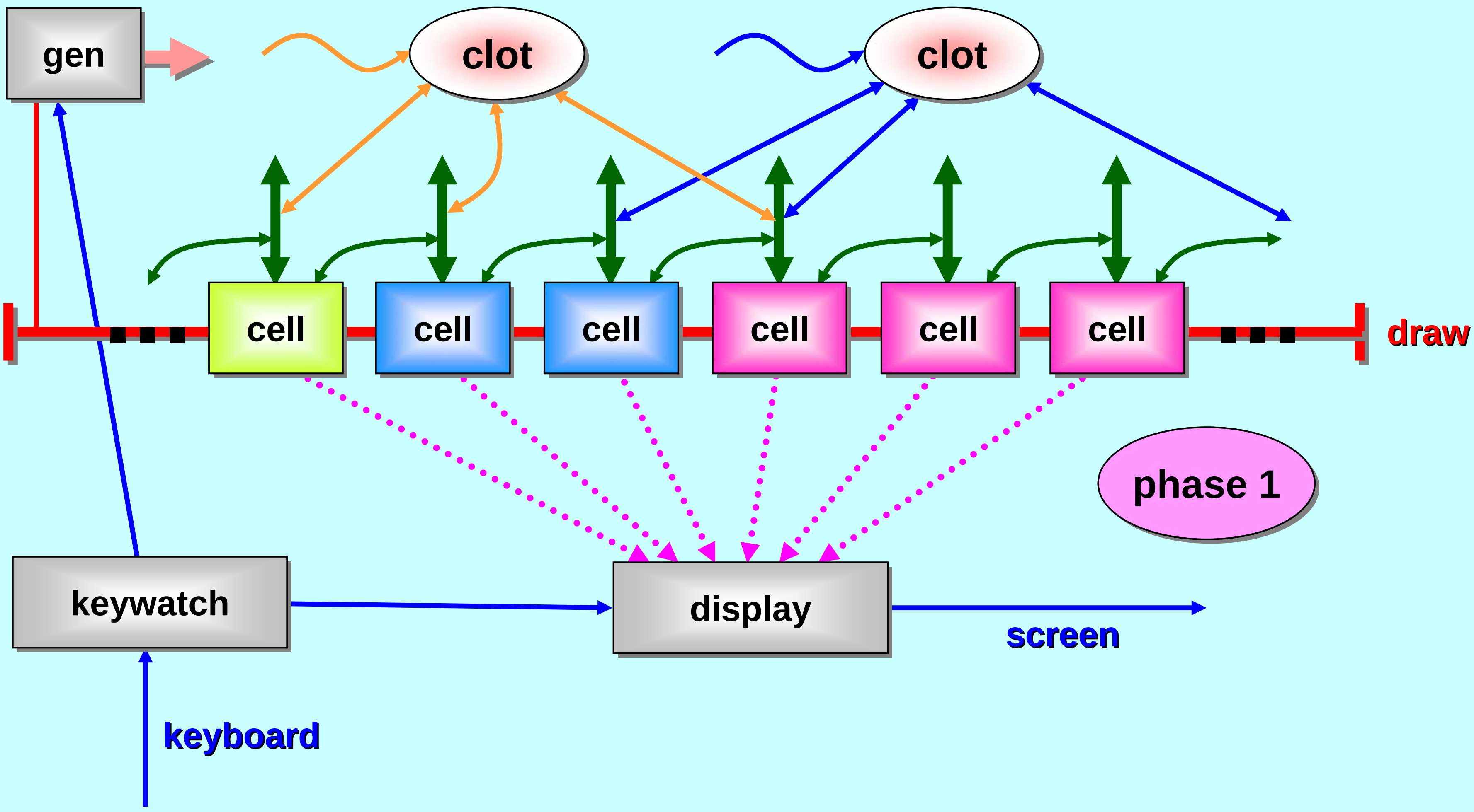
Platelet Model ('lazy' CA)



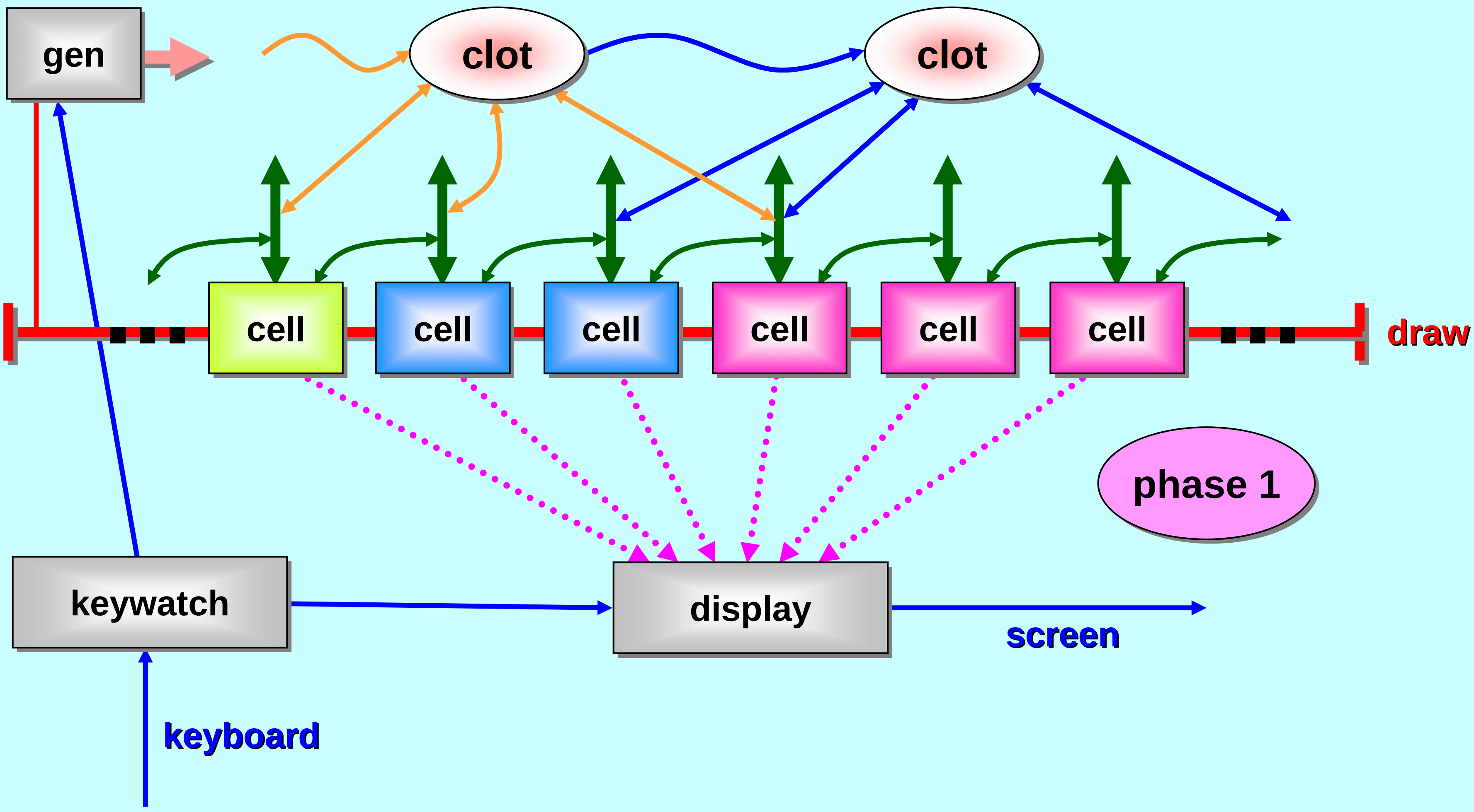
Platelet Model ('lazy' CA)



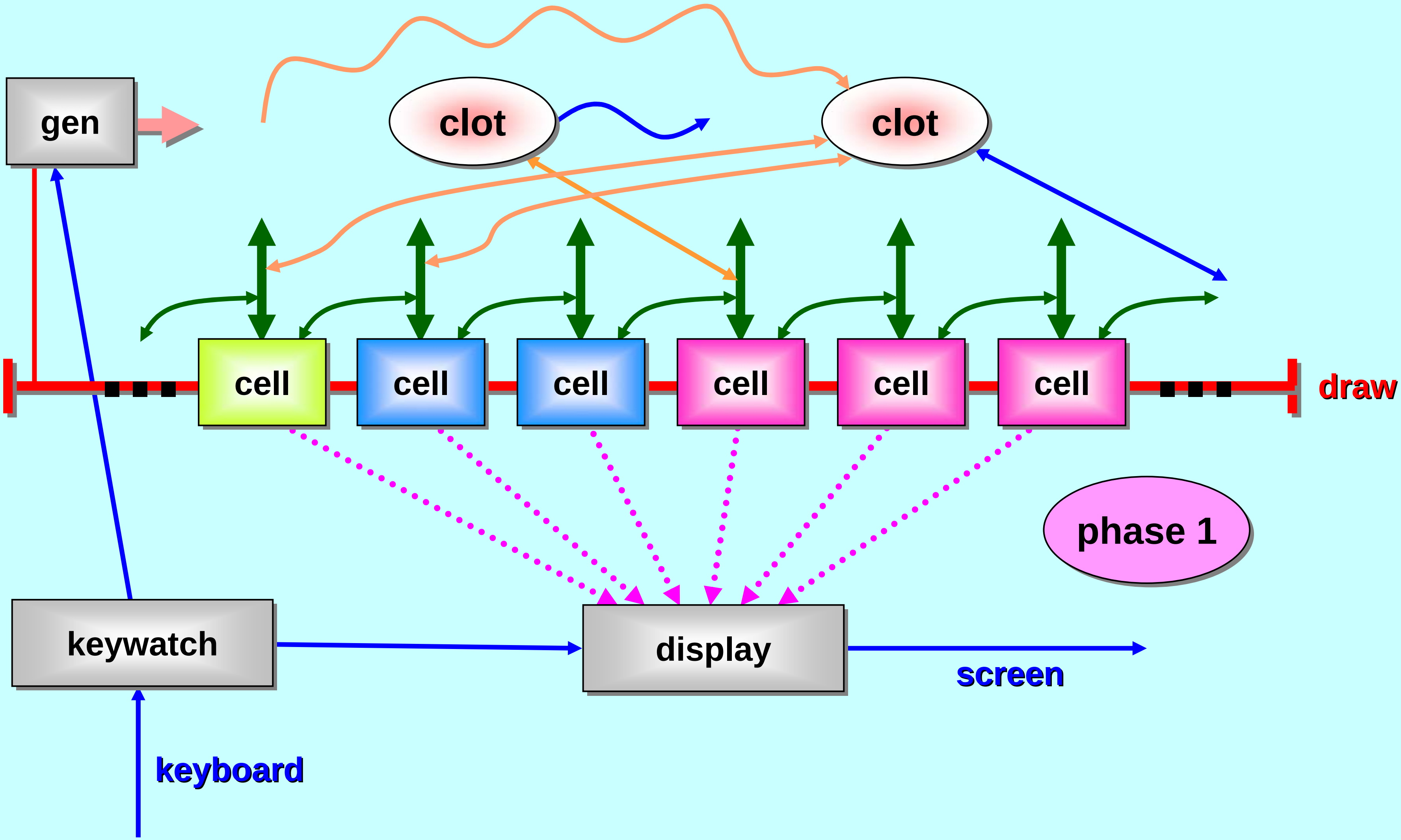
Platelet Model ('lazy' CA)



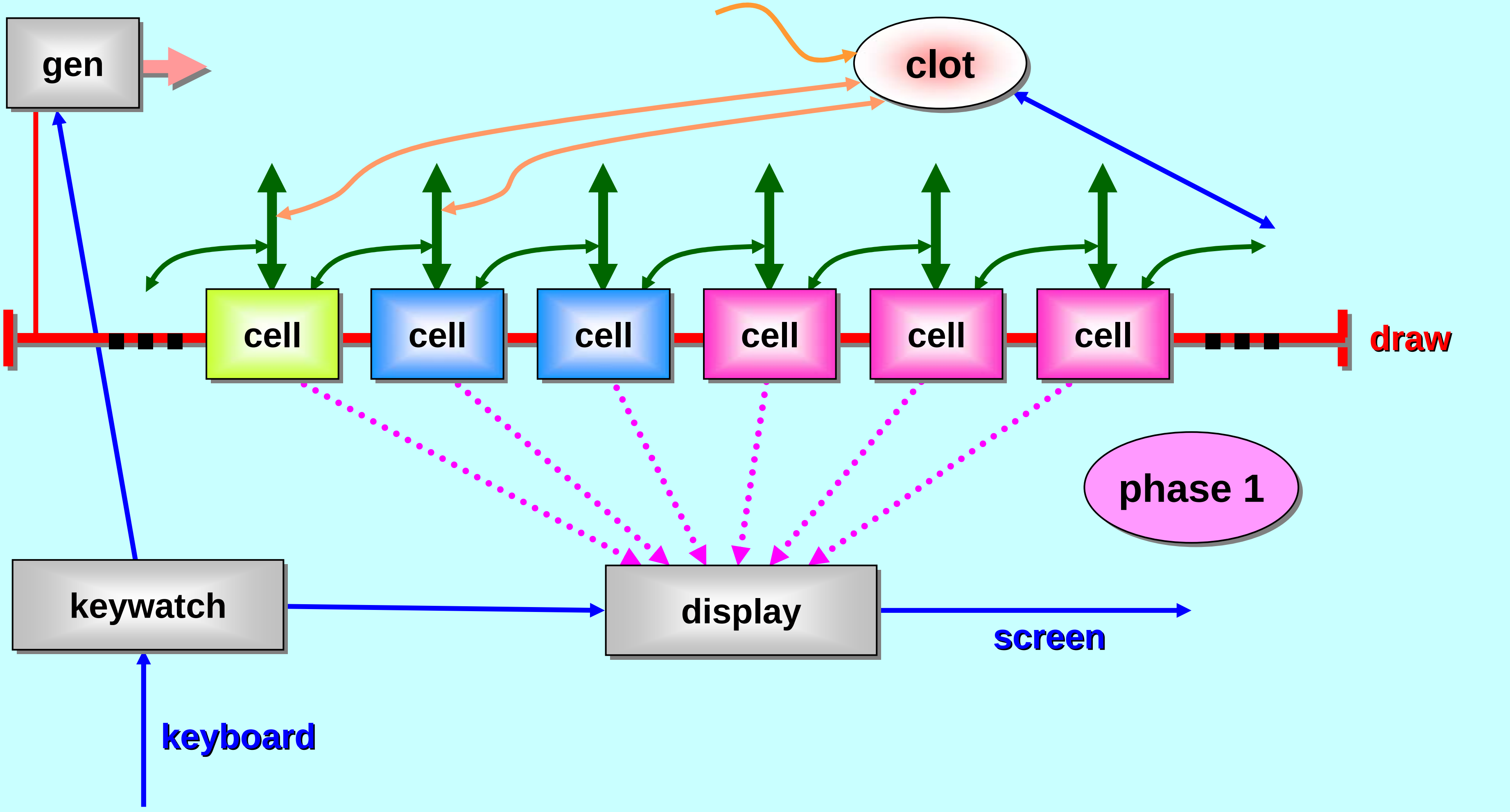
Platelet Model ('lazy' CA)



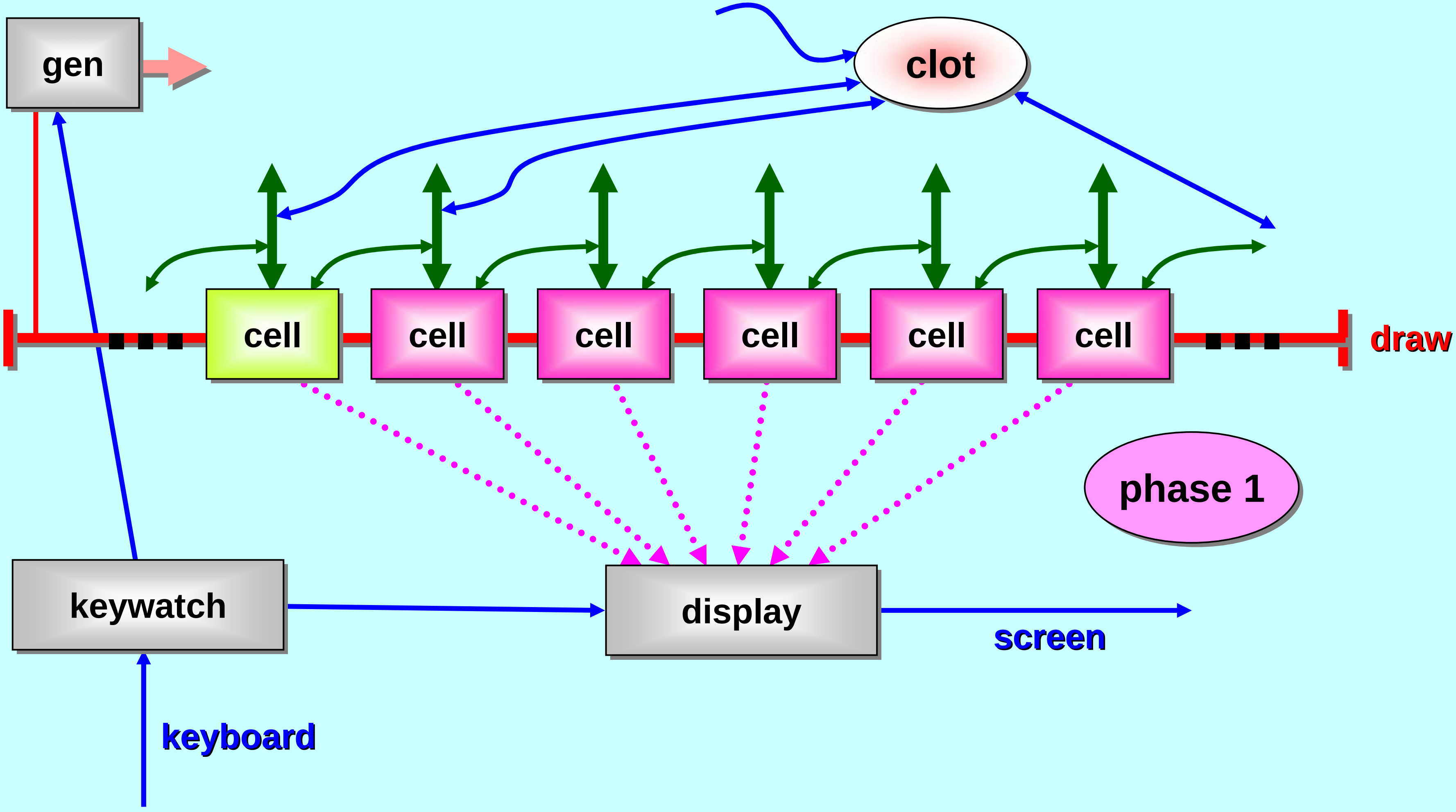
Platelet Model ('lazy' CA)



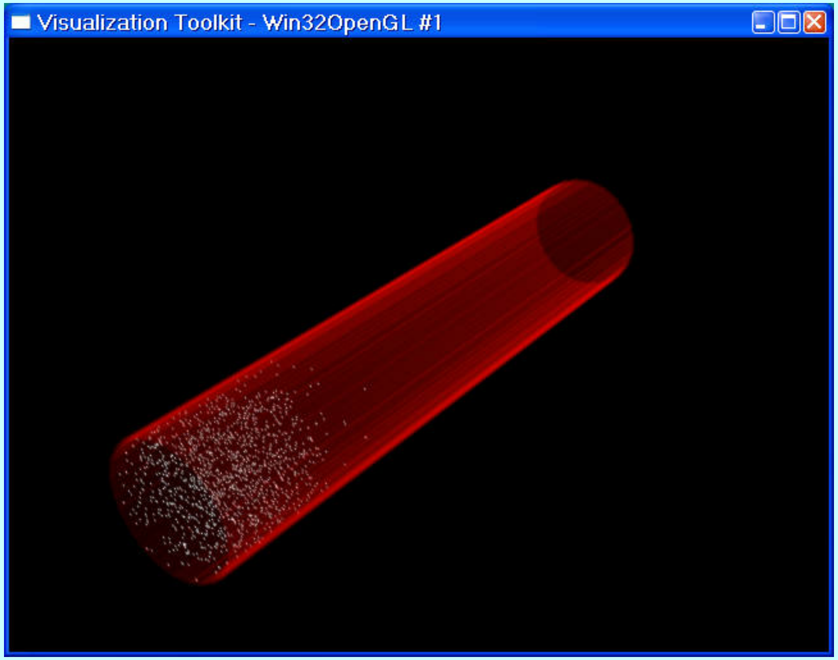
Platelet Model ('lazy' CA)



Platelet Model ('lazy' CA)

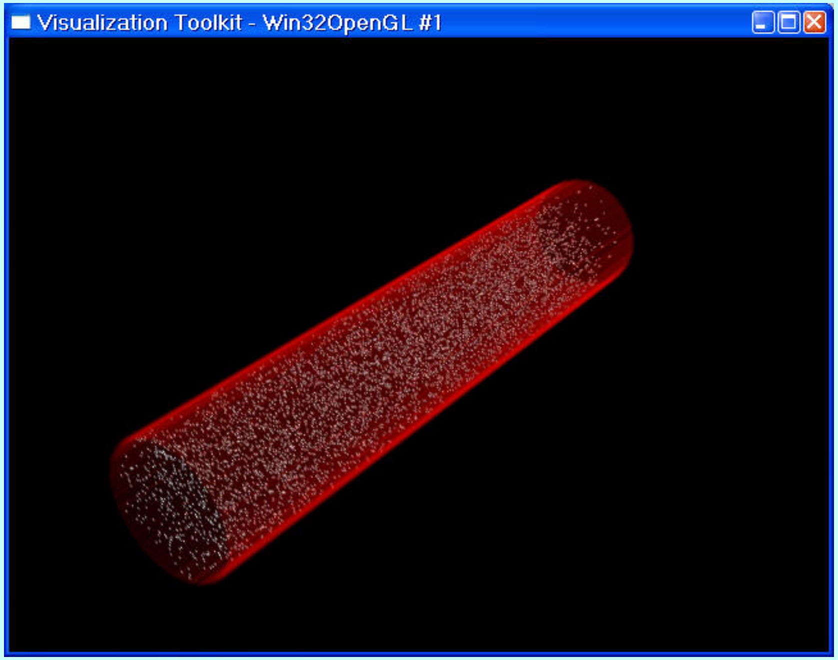


3-D Bloodstream



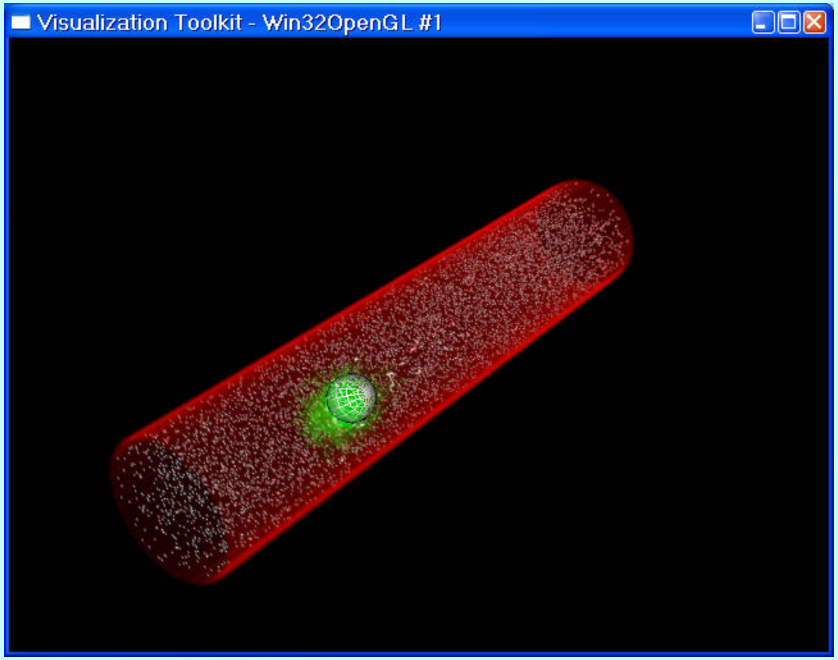
40 million processes and counting …

3-D Bloodstream



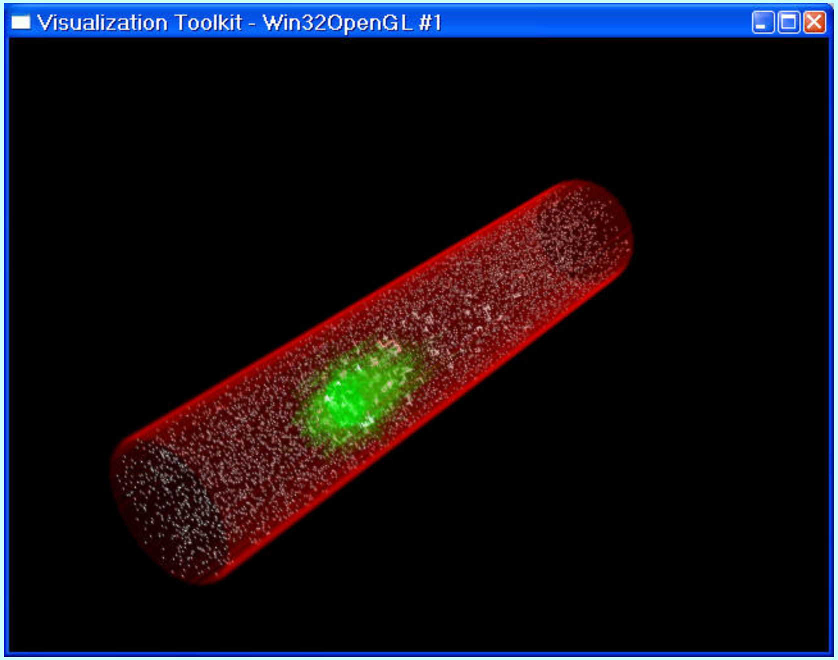
40 million processes and counting …

3-D Bloodstream



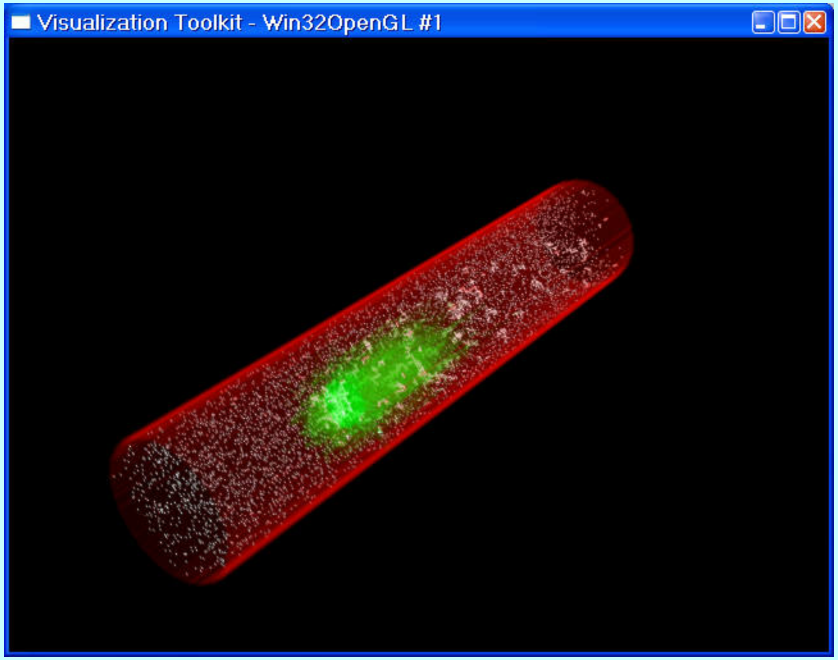
40 million processes and counting …

3-D Bloodstream



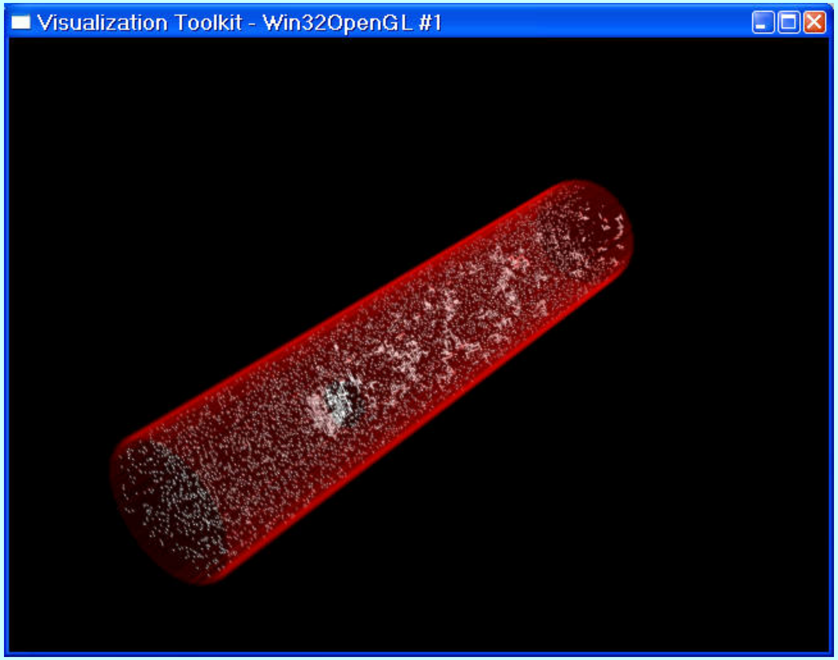
40 million processes and counting …

3-D Bloodstream



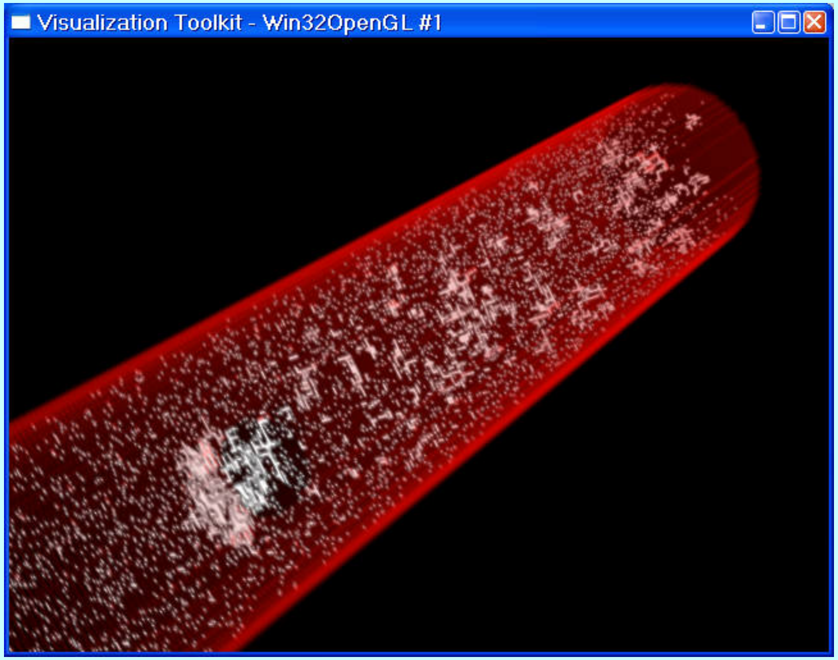
40 million processes and counting …

3-D Bloodstream



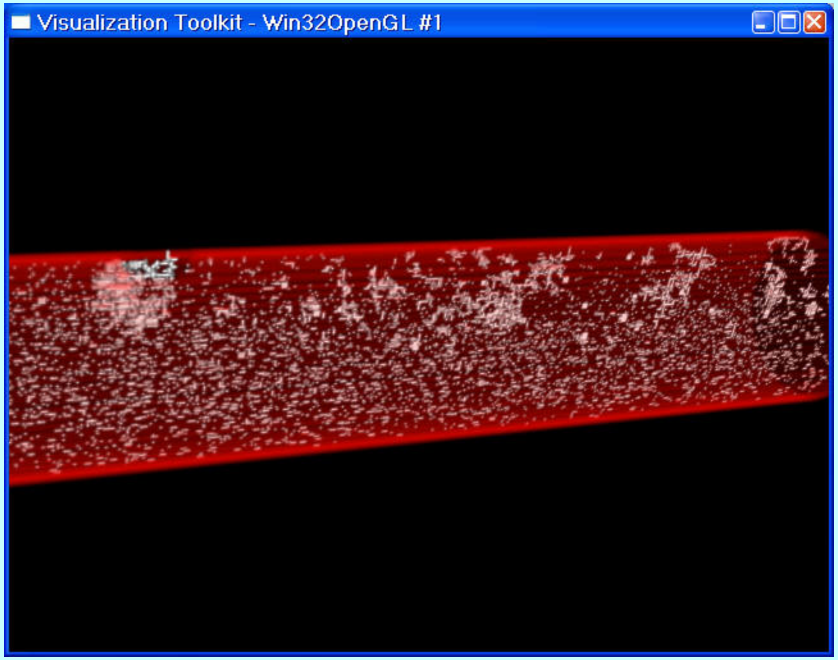
40 million processes and counting …

3-D Bloodstream



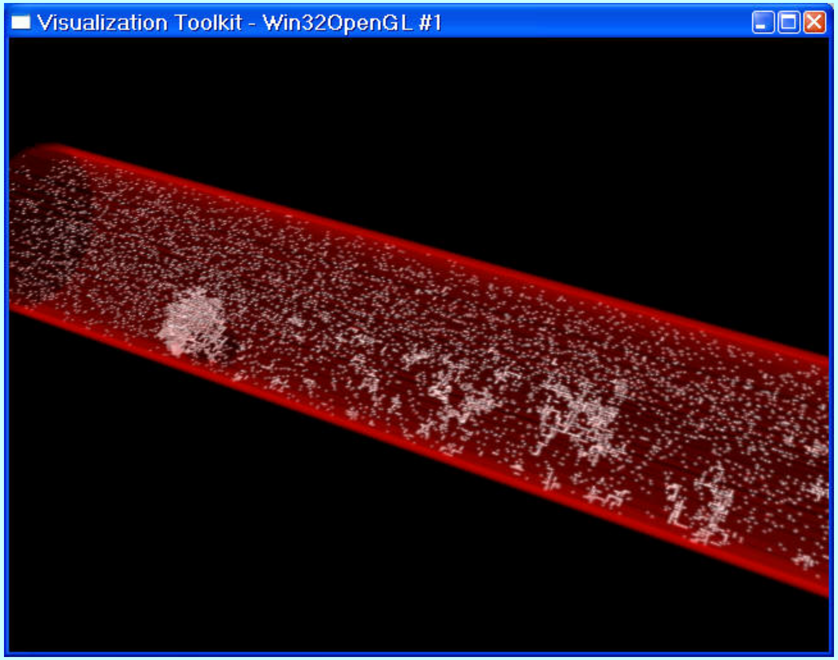
40 million processes and counting …

3-D Bloodstream



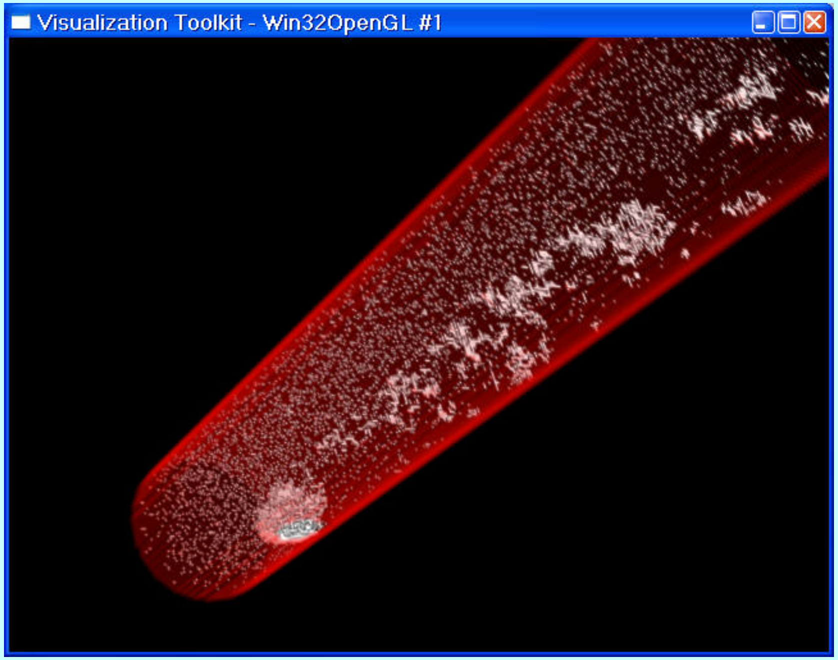
40 million processes and counting …

3-D Bloodstream



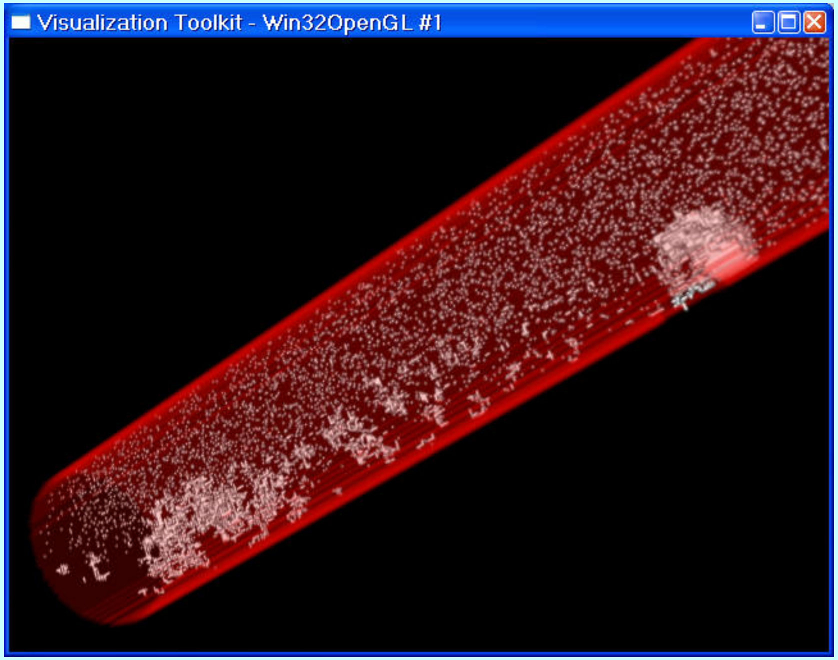
40 million processes and counting …

3-D Bloodstream



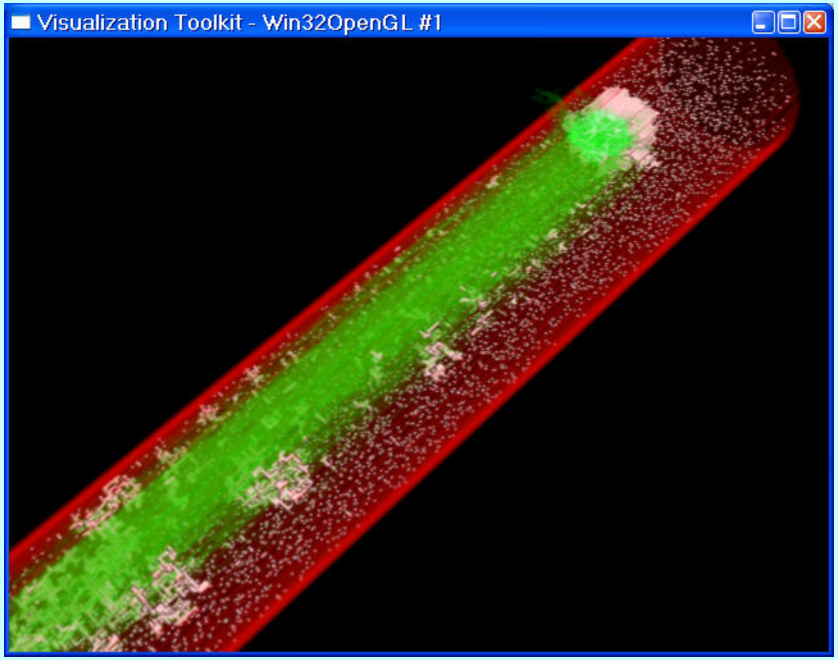
40 million processes and counting …

3-D Bloodstream



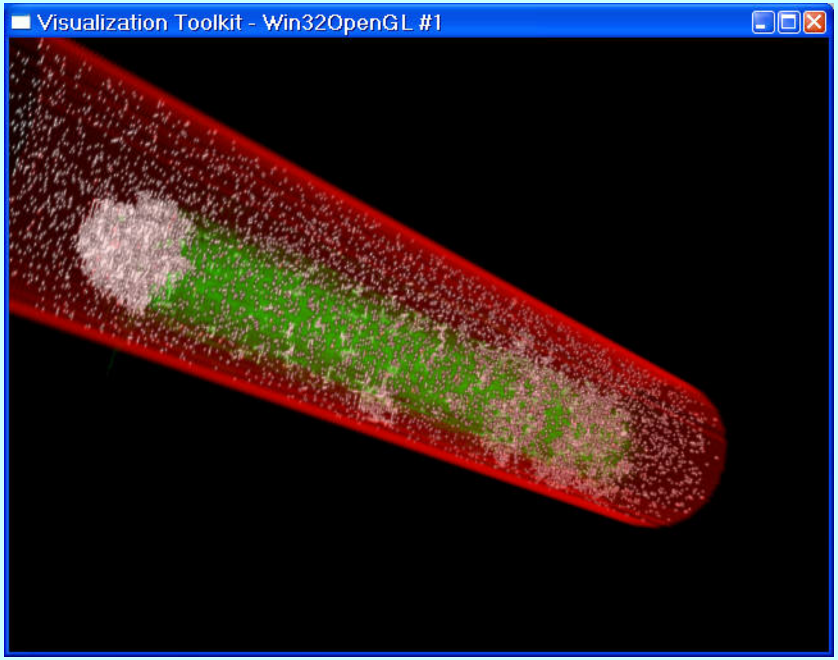
40 million processes and counting …

3-D Bloodstream



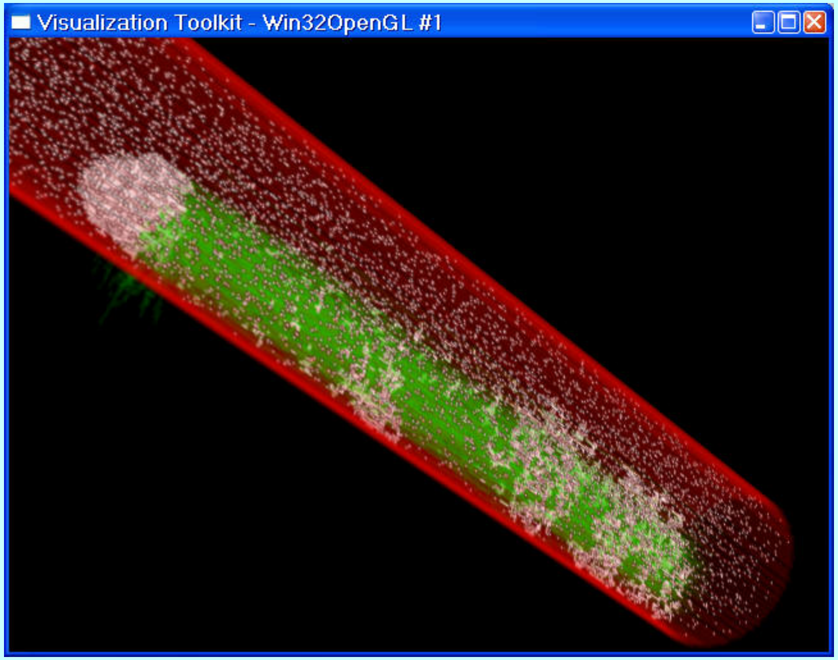
40 million processes and counting …

3-D Bloodstream



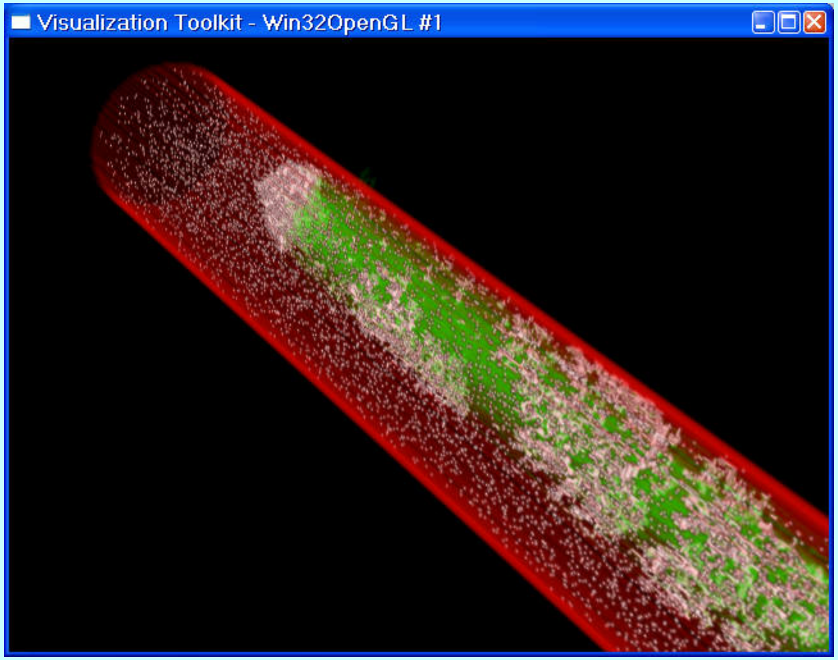
40 million processes and counting …

3-D Bloodstream



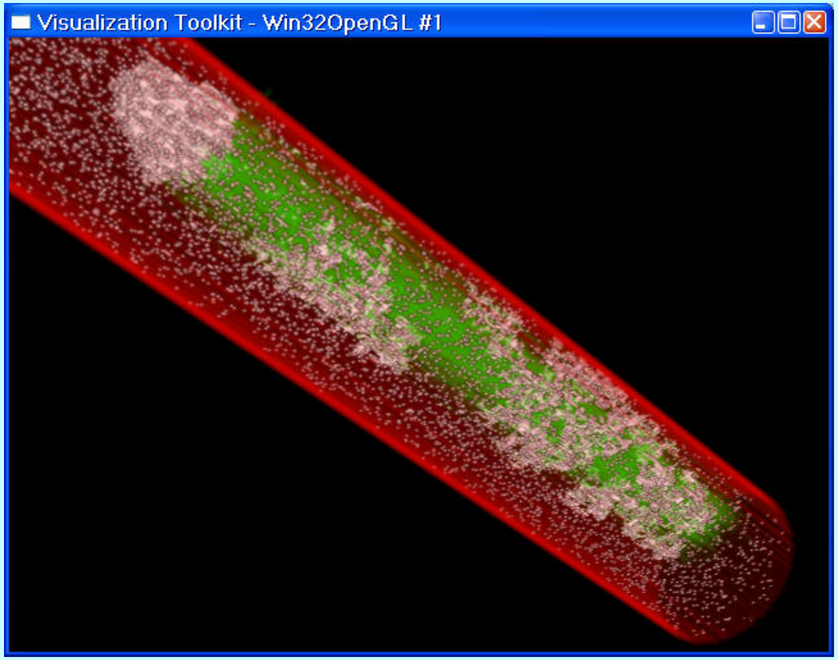
40 million processes and counting …

3-D Bloodstream



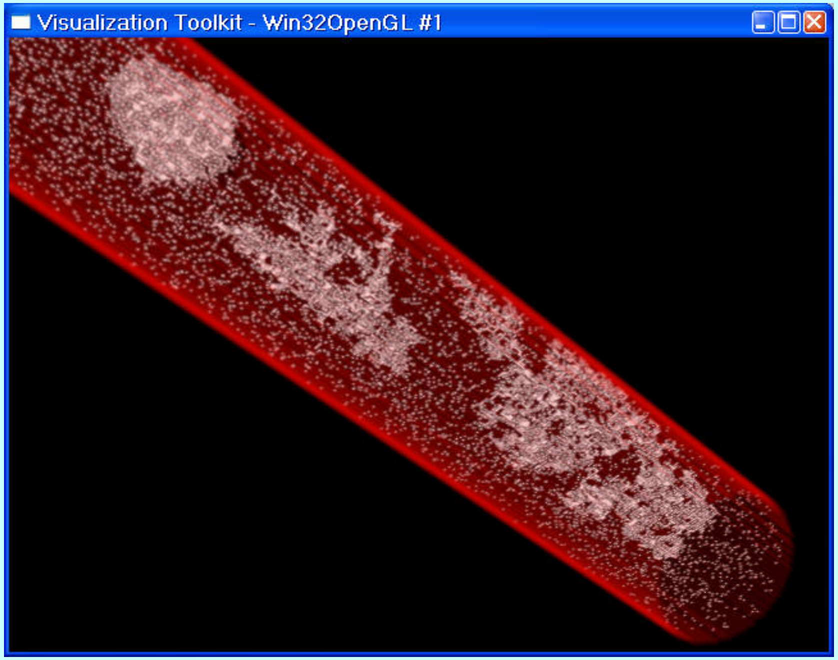
40 million processes and counting …

3-D Bloodstream



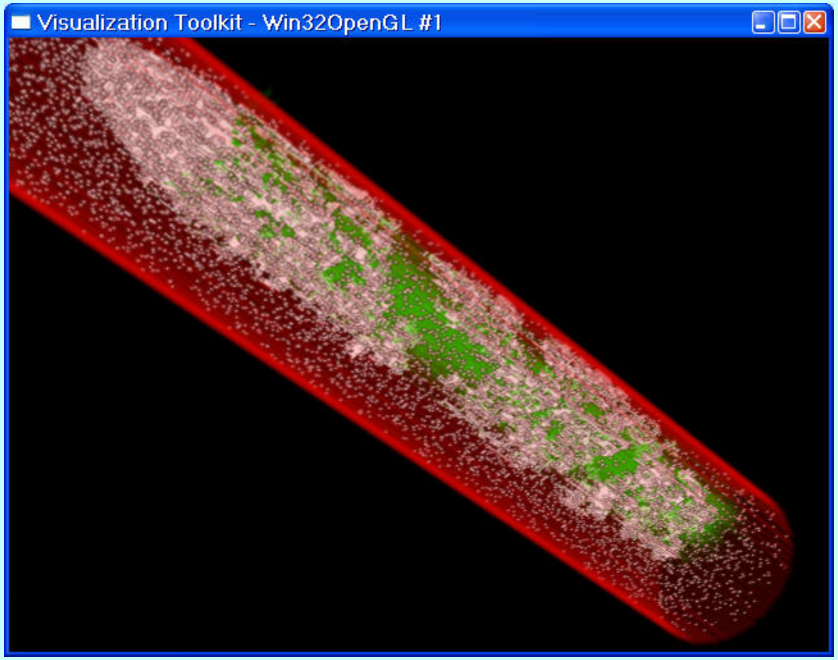
40 million processes and counting …

3-D Bloodstream



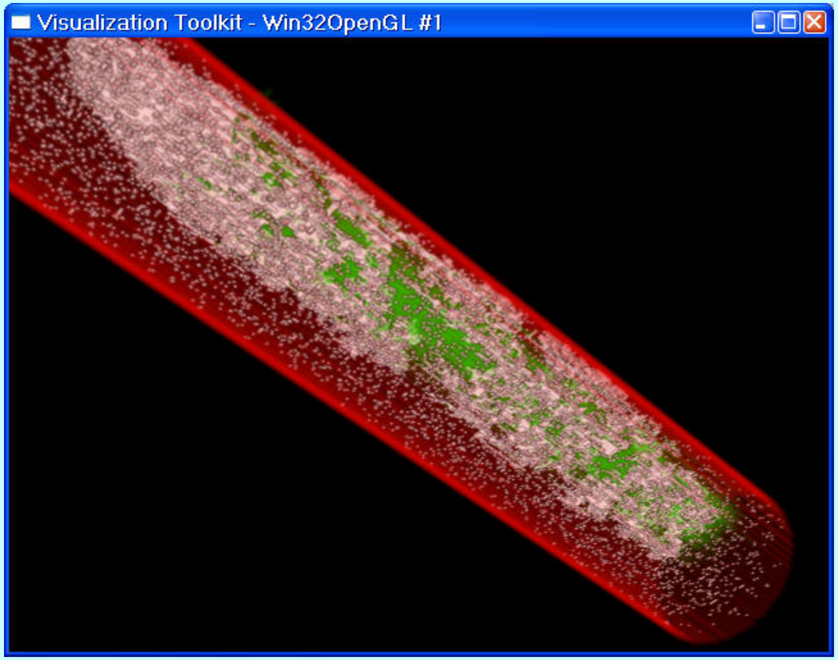
40 million processes and counting …

3-D Bloodstream



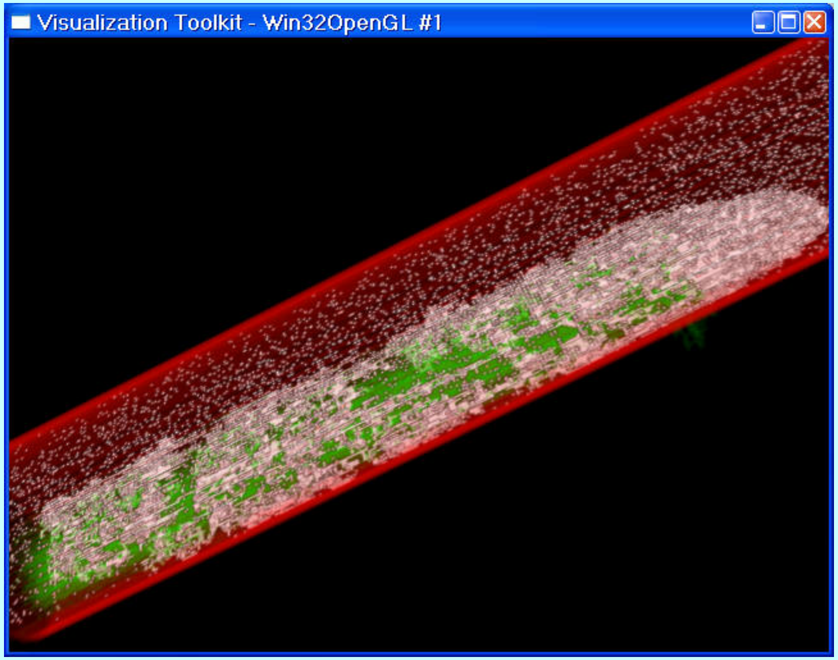
40 million processes and counting …

3-D Bloodstream



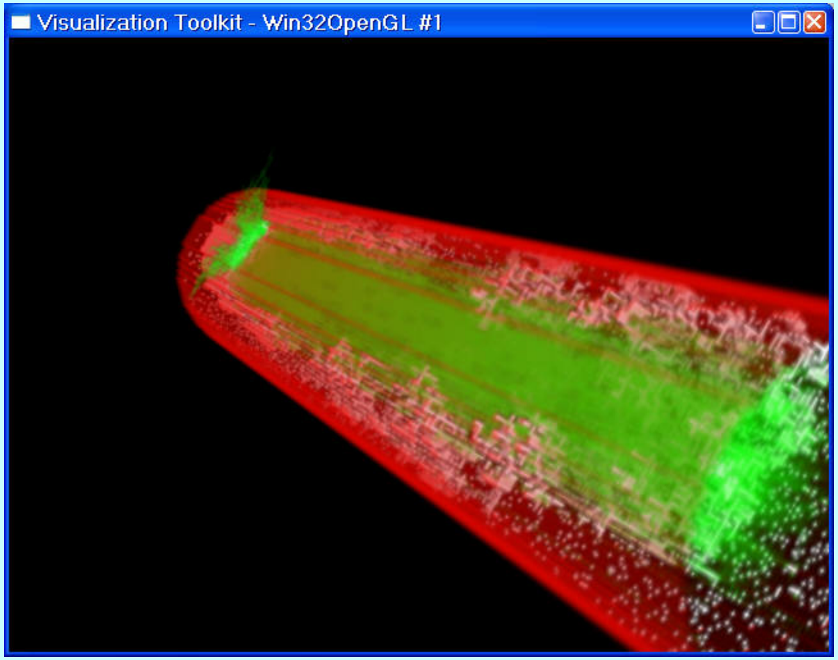
40 million processes and counting …

3-D Bloodstream



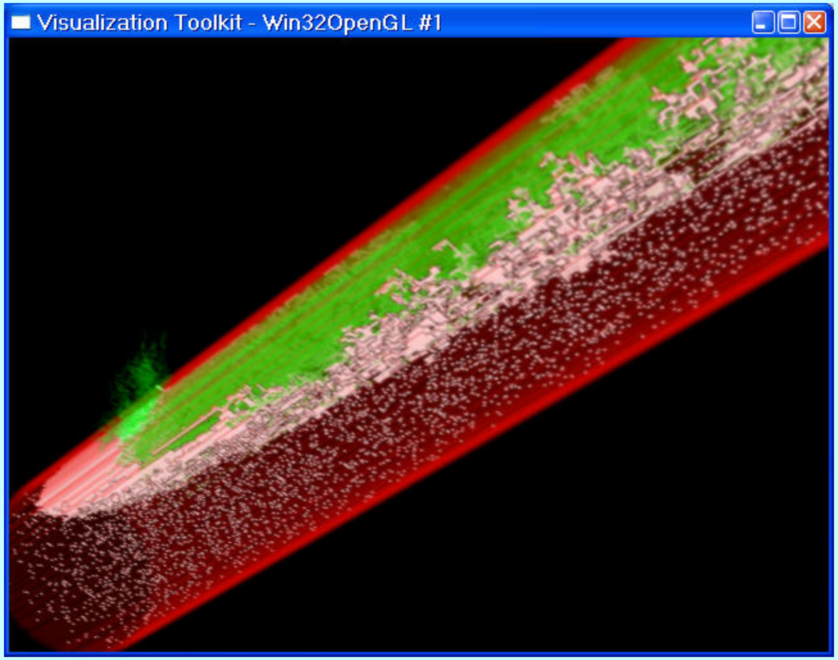
40 million processes and counting …

3-D Bloodstream



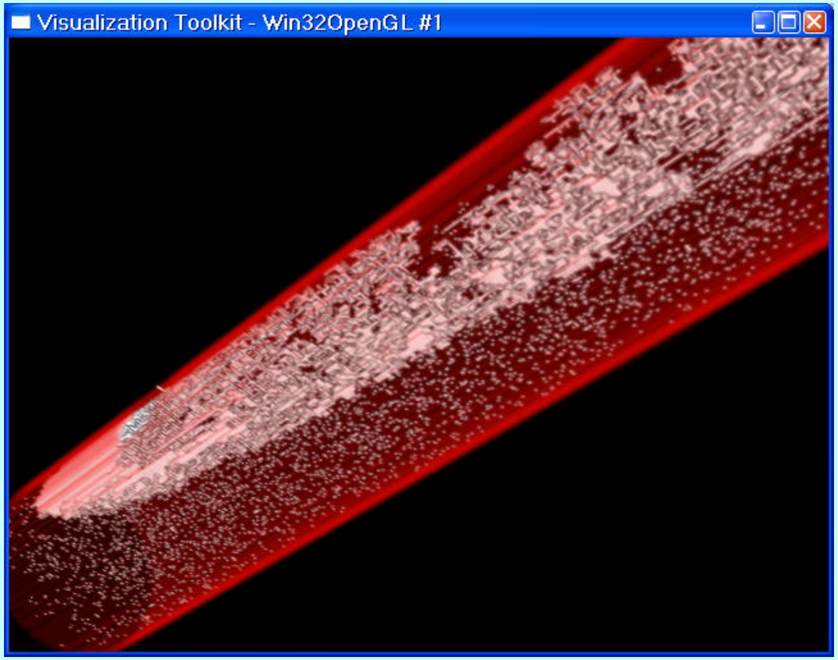
40 million processes and counting …

3-D Bloodstream



40 million processes and counting …

3-D Bloodstream



40 million processes and counting …

(occam-2π) Unfinished Buisiness

- recursive union types
- remove current **`PROTOCOLS`**
- introduce *session protocols*
- unify static/dynamic allocation
- self-verifying code
- **`BARRIER`** and output guards
- *what else … ???*

(in 19 slides)

(occam-2π) Recursive Union Types

There has been a proposal (OEP 156) for **UNION** types, since 2006:

```
DATA TYPE FOO
  CASE
    sugar, BOOL, REAL64, []BYTE
    salt, BYTE, BYTE
    pepper
:
```

```
DATA TYPE COLOUR
  CASE
    red
    green
    blue
:
```

Example literals:

```
[sugar, TRUE, 99.99, "Krakatoa"]
[salt, 42, 'A']
[pepper]
```

```
[red]
[green]
[blue]
```

(occam-2π) Recursive Union Types

```
DATA TYPE FOO
  CASE
    sugar, BOOL, REAL64, []BYTE
    salt, BYTE, BYTE
    pepper
:
```

Therefore, the kind of variant cannot be changed here.

Processing values of a union variable requires a **CASE** process to determine the variant:

```
CASE my.foo
  sugar, BOOL b, REAL64 r, []BYTE s
    ...  'b', 'r' and 's' abbreviate the component fields
    ...  optional (can change my.foo variant)
  salt, BYTE m, BYTE n
    ...  'm' and 'n' abbreviate the component fields
    ...  optional (can change my.foo variant)
  pepper
    ...  (can change my.foo variant)
```

(occam-2π) Recursive Union Types

Recursive union types are allowed:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

If only *tree* structures are allowed to be constructed, all elements of a recursive structure have only a single reference – i.e. *no aliasing* (in line with all occam elements). This enables *safe parallel processing* of all such structures.

However, this rule is (perhaps) too severe, ruling out too many useful data structures, such as cyclic or doubly-linked lists – see the paper by Pedersen and Smith (in this conference). *More thinking needed* …

(occam-2π) Recursive Union Types

Recursive union types are allowed:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

If only *tree* structures are allowed to be constructed, all elements of a recursive structure have only a single reference – i.e. *no aliasing* (in line with all occam elements). This enables *safe parallel processing* of all such structures.

However, garbage collection is trivial with the rule, with costs directly proportional to the size of the lost structure – no searching.

(occam-2π) Recursive Union Types

Recursive union types are allowed:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

In line with all occam elements, no "**`null pointer exceptions`**" can occur. *[Aside: the compiler may complain about the defined status of variables or fields. The programmer must deal with such complaints – either by correcting the code or, if the code is correct and the defined status depends on run-time events, inserting an explicit run-time check.]*

Recursive union types are an enabling data structure for writing compilers (and more), missing from occam for too long. Novel and simple parallel approaches are possible … *to be efficiently consumed by multicore*.

(occam-2π) Recursive Union Types

Example – inserting into a sorted tree:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

```
PROC insert (VAL FOO x, TREE t)
  CASE t
    node, TREE left, FOO y, INT count, TREE right
      IF
        x < y
          insert (x, left)
        x = y
          count := count + 1
        x > y
          insert (x, right)
    empty
      t := [node, [empty], x, 1, [empty]]
:
```

(occam-2π) Recursive Union Types

Example – counting the elements of a tree:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

```
PROC sum (VAL TREE t, RESULT INT total)
  CASE t
    node, TREE left, FOO x, INT count, TREE right
      INT a, b:
      SEQ
        PAR
          sum (left, a)
          sum (right, b)
        total := count + (a + b)
    empty
      total := 0
:
```

(occam-2π) Recursive Union Types

Example – counting the elements of a tree:

```
RECURSIVE DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE
    empty
:
```

```
INT FUNCTION sum (VAL TREE t)
  INT total:
  VALOF
    CASE t
      node, TREE left, FOO x, INT count, TREE right
        total := count + (sum (left) + sum (right))
      empty
        total := 0
    RESULT total
:
```

occam functions have no side-effects, so sub-expressions may be run in parallel automatically.

(occam-2π) Remove Current Protocols

Sequential Protocols

Mostly, these are simply replaced by **RECORD** data types.

The one *semantic* win for sequential protocols over **RECORD**s was taking advantage of the *sequence* in the protocol – for example:

```
in ? i; A[i]
```

where an early item of data is used to address the location of a later one.

However, this is won back with *session protocols*, with no loss of syntactic clarity or runtime efficiency. See later.

(occam-2π) Remove Current Protocols

Counted Array Protocols

These are replaced by dynamically sized arrays.

(occam-2π) Remove Current Protocols

Variant (**CASE**) Protocols

These are replaced by *union* data types.

The one *pragmatic* win for variant protocols over *union* types was when program logic meant that a *large* data variant did not need to be considered by the receiving process – so that space for that *large* variant did not need to be allocated .

In the new proposal (see later), this is won back (and more) through all *large* data items being on the heap and only references being (safely) moved.

The above paragraph assumes processes connected in the same memory space. But it's still true for processes in different memory spaces – the data is copied from heap to heap and the reference obtained by the receiving process will be valid for its memory.

(occam-2π) Session Protocols

Adam Sampson's *"Two-Way Protocols"* (CPA 2008, York)

These are communication protocols in the sense normally understood (i.e. *patterns* of communication).

They are associated with a single channel, which may have **SHARED** ends.

The channel is *directed* (in the same sense as a current *channel record* is directed), but may be used in both directions (possibly at the same time!).

(occam-2π) Session Protocols

Adam Sampson's *"Two-Way Protocols"* (CPA 2008, York)

The simplest session protocol is *one* data type, sent *one* way, *once*. This corresponds to a classical channel.

Structured sessions consist of separately typed messages flying in (nested) **`SEQ`**, **`ALT`** and/or **`PAR`**. This declared structure *is* the *session protocol* – syntactic details are not yet settled. The compiler checks that all code operating on the channel conforms, tracking use across all processes and procedures. Channel parameters carrying a session protocol will have to declare which (named) part of the protocol their **`PROC`** implements.

We *may* be able to drop *channel records* from the language. These are mainly used for two-way conversations and are more safely handled by a *session protocol* (and with far less syntactic clutter).

(occam-2π) Unify Static / Dynamic

Compiler-known *small* items (<= 8 or 16 bytes?) are pre-allocated on their process stack.

Everything else is dynamically allocated on the heap, with references on the stack.

The programmer is blind to the above. In particular, there is the same syntax for declaring sized arrays, regardless of whether the size is known to the compiler:

```
[n]THING t:          -- 'n' may be a run-time value
```

Array size is no longer part of the type. An array variable declared with one size may be assigned to an array with another size (same type, of course). An array variable may be declared without size, but must then be assigned (either by assignment or incoming communication) to an actual array value before being used.

(occam-2π) *Unify Static / Dynamic*

Assignment and communication are handled in the most efficient way.

Stack items (always *small*) are assigned/communicated by *copying*.

Heap items are assigned/communicated by *reference*:

Normally, this is the reference to the item held by the sender …

However, if compiler usage analysis of the sending process shows the assigned/communicated data is used later by that process, a reference to a (deeply) cloned copy is sent.

This is Neil Brown's algorithm (*"Auto-Mobiles: Optimised Message-Passing"*, CPA 2009, TU Eindhoven).

(occam-2π) Unify Static / Dynamic

Assignment and communication are handled in the most efficient way.

Data may optionally qualified as **`MOBILE`** (if the application semantics demands that only one copy may exist at all times):

Small **`MOBILE`** items (on the stack) are assigned/communicated by *copying* – as before.

Large **`MOBILE`** items (on the heap) are assigned/communicated by *reference* – as before. The reference will be to the item held by the sender …

However, if compiler usage analysis of the sending process shows the assigned/communicated data is used later by that process, this is a semantic error and the compilation fails (reporting the error).

This is the current algorithm for ***occam-π*** mobiles.

(occam-2π) Self-Verifying Code

See my *EndNote* paper (*"Adding Formal Verification to occam-π"*, CPA 2011, Limerick).

This is a proposal to make formal verification of **occam-π** programs manageable entirely within the language.

The language is extended with qualifiers on types and processes (to indicate relevance for verification and/or execution) and assertions about refinement (including deadlock, livelock and determinism).

The compiler abstracts a set of CSP equations and assertions, delegates their analysis to the FDR3 model checker and reports back in terms related to the **occam-π** source. The full (FDR3) range of CSP assertions is accessible, with no knowledge of CSP formalism required by the **occam-π** programmer.

Programs are proved just by writing and compiling programs.

(occam-2π) Barrier and Output Guards

We have them in JCSP … why not in occam-2π ?

```
ALT
  SYNC bar
    ...  over the barrier, carry on
  out ! n
    ...  message taken, continue
  in ? x
    ...  message arrived, process it
  tim ? AFTER timeout
    ...  response
```

So long as the additional costs on ALTs not using them can be made negligible …

(occam-2π) What Else … ???

Allow barriers and channels to be mixed with data in record fields?

Classically, synchronising elements and passive data have been kept separate. Operations on them have different syntax (e.g. sending on a channel is *not* a procedure call). The latter has clear semantic benefit and should remain. Can we relax on the former? What are the benefits?

What else … ???

occam3 INITIAL, FINAL, RESOURCE and SERVER declarations.

occam3 parametrised MODULE types and libraries.

Extended outputs (**! !** as well as **??**).

Abstract data types (not classes!) and generics.

Almost done …

Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms *(e.g. processes, communication, synchronisation, networks)* and higher level patterns *(e.g. client-server, phased barrier, I/O-PAR)* … ?

- they can use those primitives and patterns with the same fluency as they use serial computing primitives, *without tripping over dark hazards* … ?

- they can develop their own patterns when the standard ones don't apply … ?

- they can use formal methods to verify good behaviour *(e.g. freedom from deadlock and livelock, safety, liveness)*, without training in the underlying mathematics (*process algebra, denotational semantics)* … ?

- they can do this as *normal everyday practice*, without any sense of fear … ?

Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they quickly develop a correct and intuitive understa… the primitive mechanisms (*e.g. processes, communication, sy… n, networks*) and higher level patterns (*e.g. client-server, ph… , I/O-PAR*) … ?
- they can use those primitives and patter… same fluency as they use serial computing primitives, *without tr… dark hazards* … ?
- they can develop their own pat… the standard ones don't apply … ?
- they can use formal met… y good behaviour (*e.g. freedom from deadlock and livelock… ness*), without training in the underlying mathematics (*proc… , denotational semantics*) … ?
- they can do this… *al everyday practice*, without any sense of fear … ?

Yes, we can!

Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they qui… velop a correct and intuitive understa… the primitive mech… *processes, communication, sy… n, networks)* and high… s (e.g. *client-server, ph… , I/O-PAR)* … ?

- they can use those … patter… same fluency as they use serial computing primit… *dark hazards* … ?

- they can develop their own pat… tandard ones don't apply … ?

- they can use formal met… y goo… g. *freedom from deadlock and livelock… ness)*, withou… underlying mathematics (*proc… , denotational seman…*

- they can do this… *al everyday practice*, without any se… r … ?

And not only students …

Yes, … can!

So, which language has …

a *dynamic concurrency model* built into its core design … with full denotational semantics (based on the *CSP traces/failures/divergences model*) …

no *data race hazards* (eliminated by compiler aliasing analysis) …

deterministic concurrency by default. Non-determinism is introduced *only* by explicit use of special features (e.g. choice, shared channels) …

the *fastest and most effective* multicore scheduler on the planet (*probably*) …

program verification by programming (and a little thinking) …

ease of learning, ease of use (e.g. 90 min Lego Robots *'Fresher'* workshop) …

past major industrial use (20-25 years ago) …

demonstrated powers of expression and performance in a range of currently important application areas (e.g. large-scale modelling, emergence, embedded micro-systems) …

Any questions?

And not a word was spoken,

The transputers all were broken …

((almost) Don McLean, 1971)