Motivation: Concurrency for All (Process Oriented Design)

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

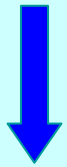
It was 20 years ago today …

Objects considered harmful …

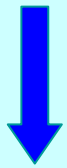
Modelling complex systems …

Blood clotting …

Nature is not organised as a single thread of control:

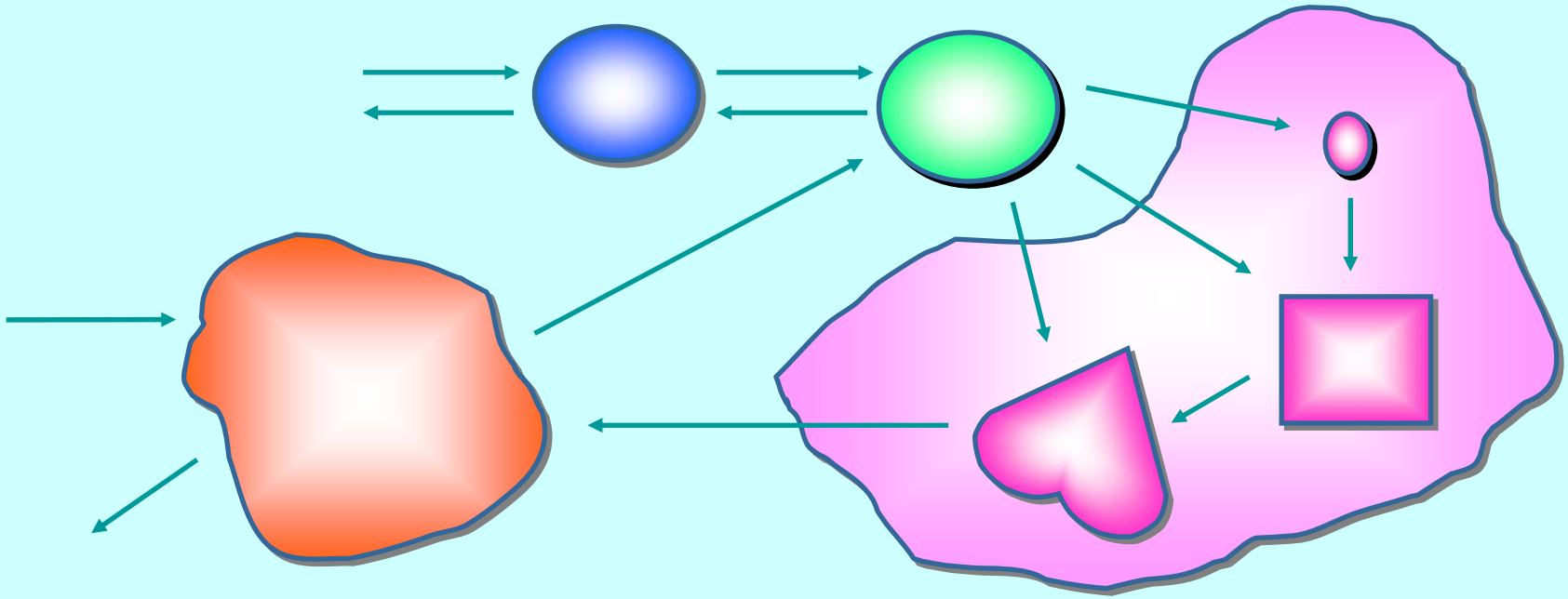


```
joe.eatBreakfast ();
sue.washUp ();
joe.driveToWork ();
sue.phone (sally);
US.government.sue (bill);
sun.zap (office);
```



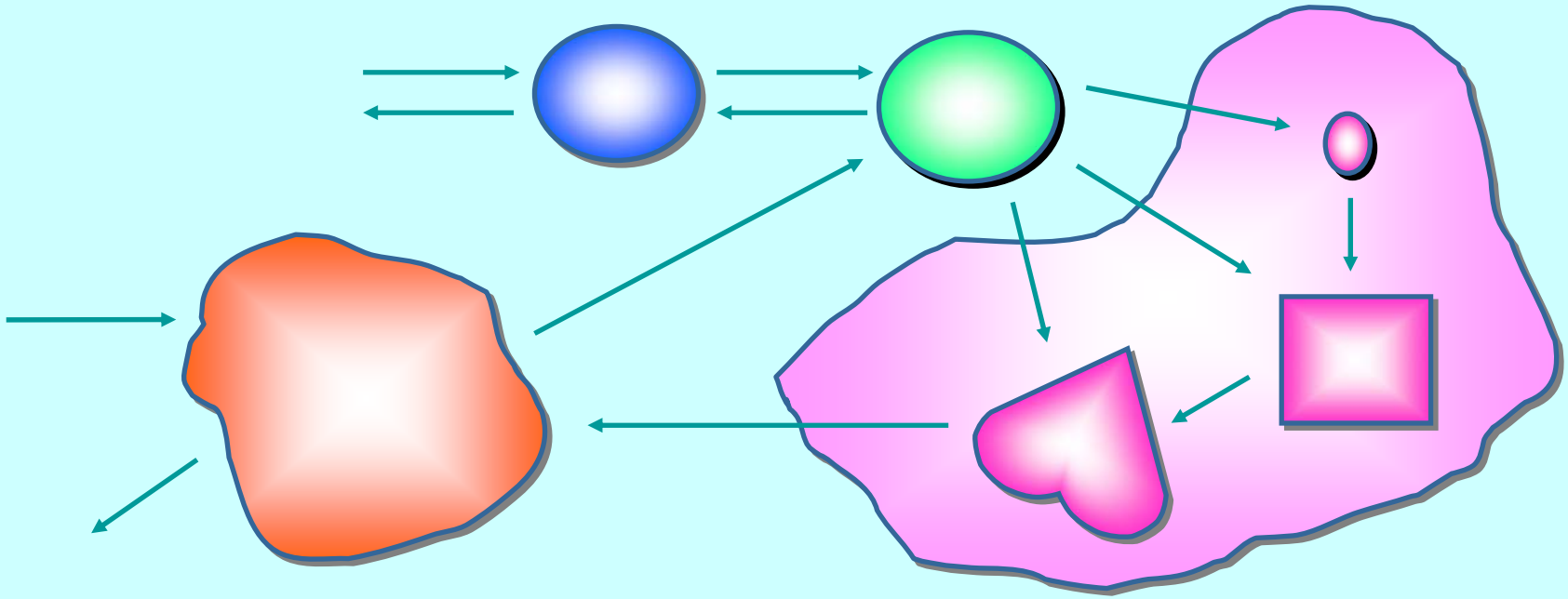
Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

… nanite … human … astronomic ...



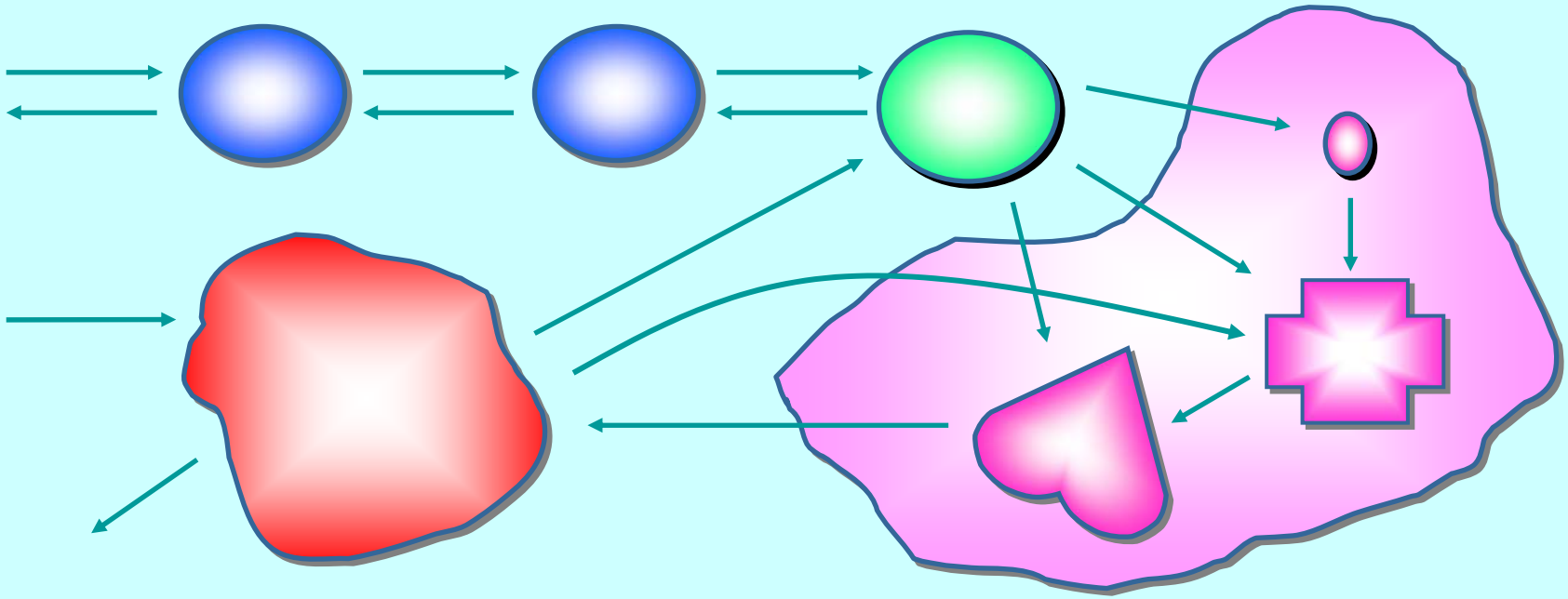
The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nanite … human … astronomic ...



The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nanite … human … astronomic ...



Motivation and Applications

- **Thesis**
  - Natural systems are robust, efficient, long-lived and continuously evolving. ***We should take the hint!***
  - Look on concurrency as a ***core design mechanism*** – not as something difficult, used only to boost performance.
- **Some applications**
  - Hardware design and modelling.
  - Static embedded systems and parallel supercomputing.
  - Field-programmable embedded systems and dynamic supercomputing (e.g. **SETI-at-home**).
  - Dynamic distributed systems, eCommerce, operating systems and games.
  - Biological system and ***nanite*** modelling.

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

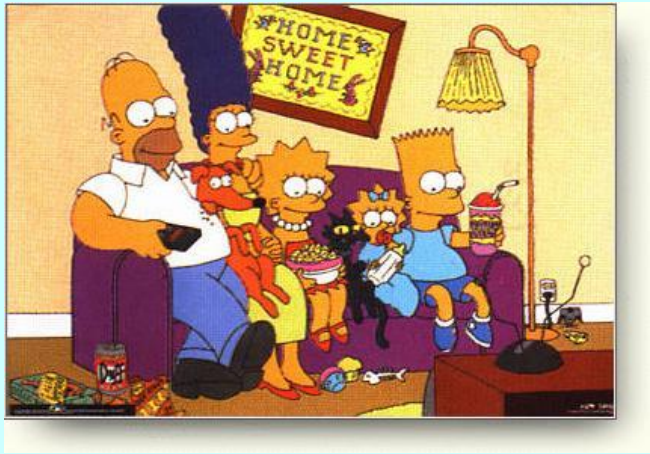
It was 20 years ago today …

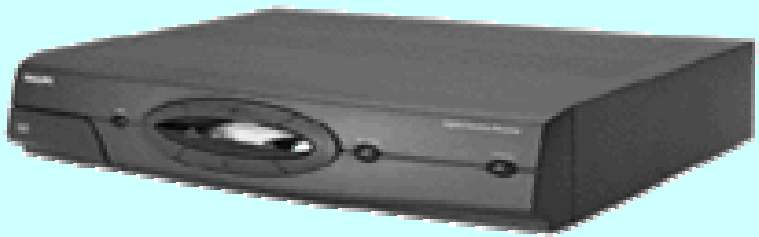
Objects considered harmful …

Modelling complex systems …

Blood clotting …

Components?





Components must be *composeable …*

… and they must compose *simply!*

Components?

Mind you, just because components compose …

… doesn't always mean that it makes sense …

Components?

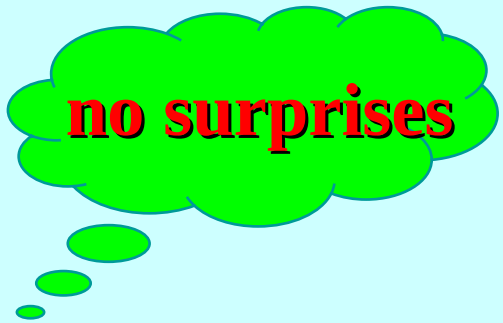
*

… to compose them …

*Image courtesy of Philips TASS `<http://www.tass.philips.com/>`

Components?

- If we understand A and B separately, we must be able to deduce *simply* their combined behaviour.



- Semantics [A + B] = Semantics [A] + Semantics [B]

- A and B must be *composeable …*

Composition?

- Complex systems are *composed* from *less complex* components …
- … which are *composed* from *simpler* components …
- … which are *composed* from *simpler* components …
- … etc …
- … which are *composed* from *simple* components.

Composition?

- Composition rules must be simple and yield no surprises.
- Whatever it is they encapsulate, ***components*** must have ***interfaces*** that are *clean*, *complete* and *explicit*.
- Hardware systems are forced (by physics/geometry) to be built like this.
- Software systems have no such constraints. We think we can do better than nature … and get into trouble.

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

It was 20 years ago today …

Objects considered harmful …

Modelling complex systems …

Blood clotting …

The Real World and Concurrency

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 mastered first).

This tradition is WRONG!

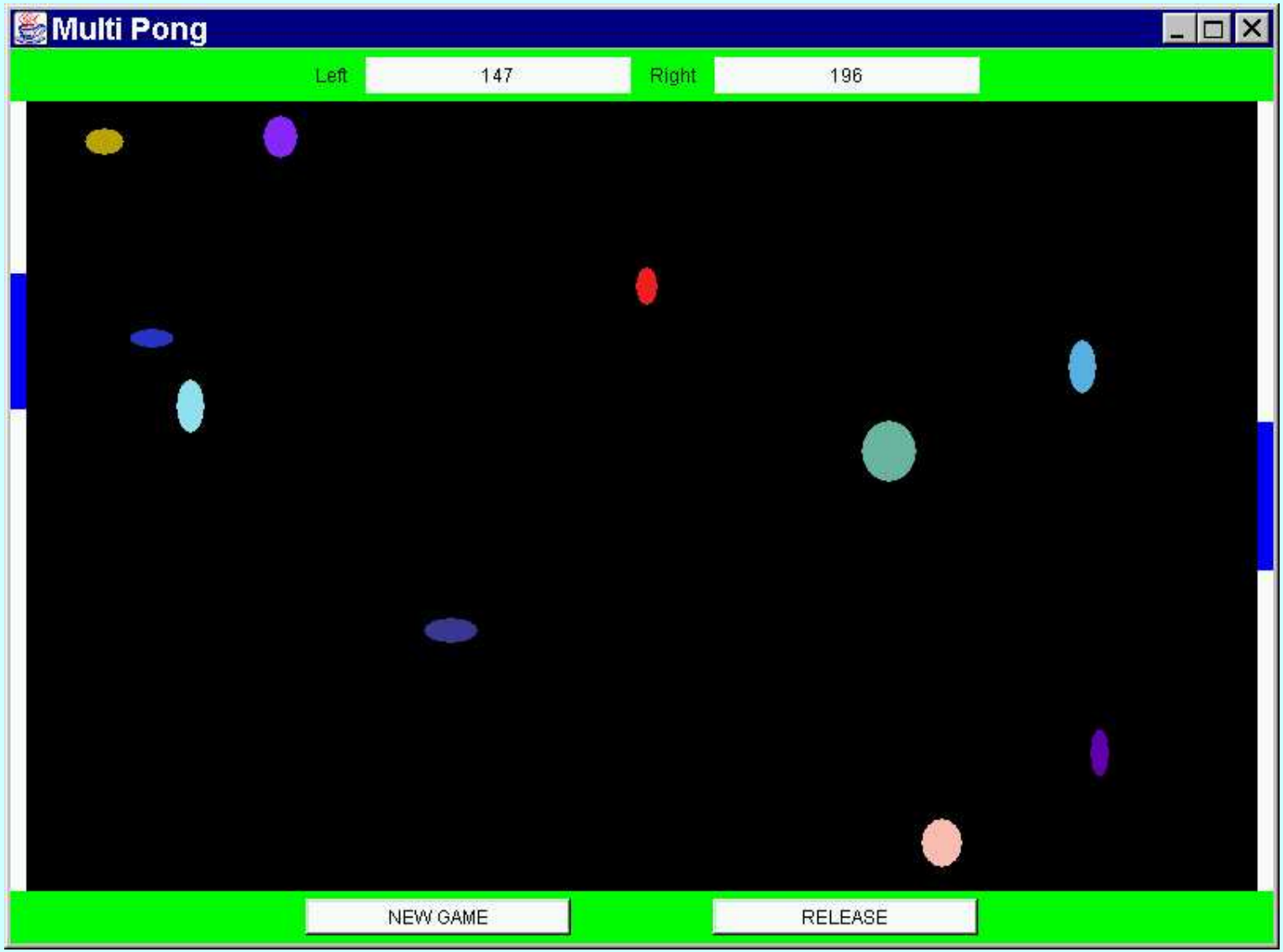
… which has (radical) implications on how we should educate people for computer science …

… and on how we apply what we have learnt …

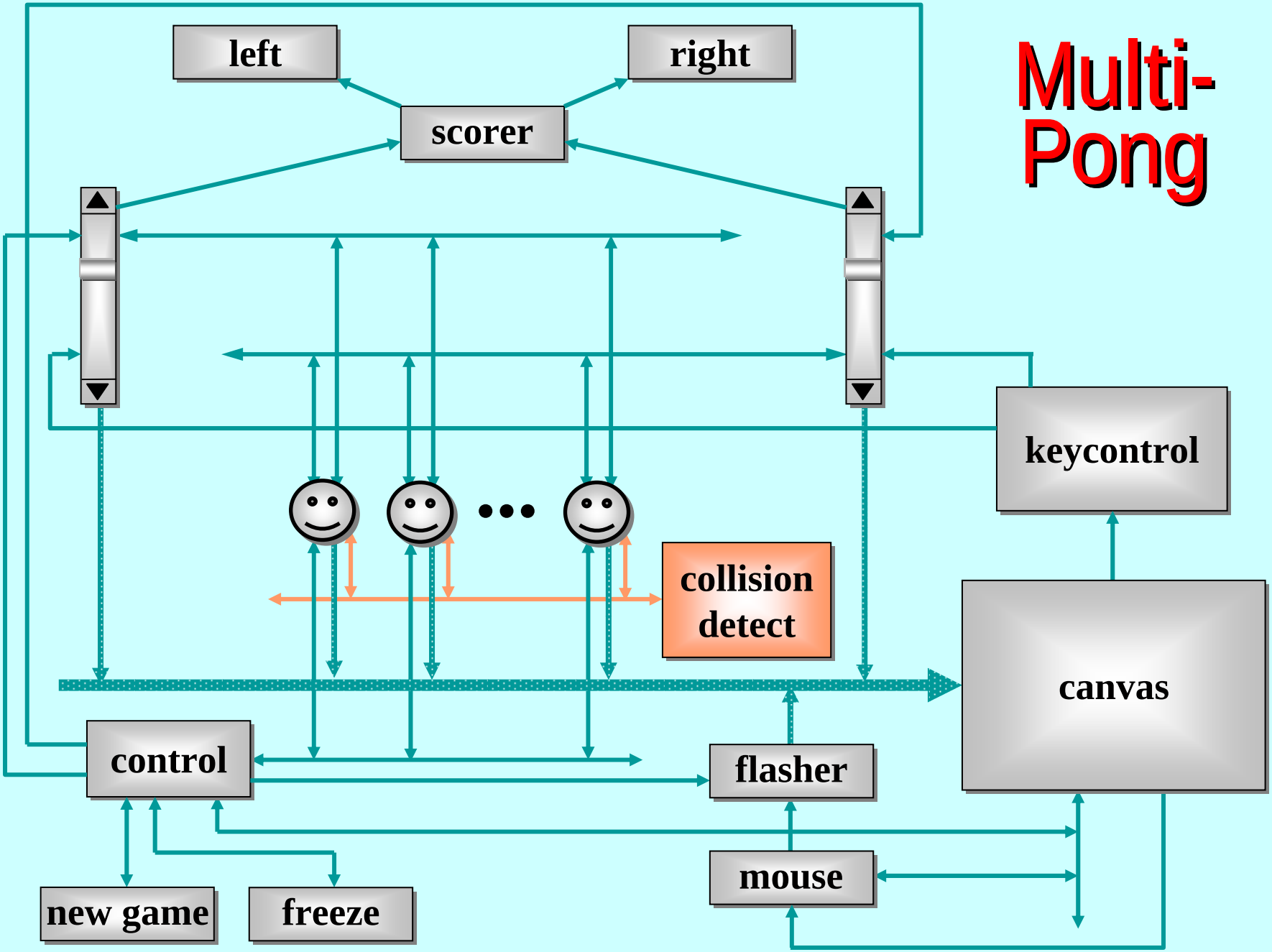
What we want from Parallelism

- A powerful tool for ***simplifying*** the description of systems.
- ***Performance*** that spins out from the above, but is ***not*** the primary focus.
- A model of concurrency that is ***mathematically clean***, yields no engineering surprises and scales well with system complexity.

Multi-Pong



Multi-Pong



Good News!

The good news is that we can worry about each process on its own. ***A process interacts with its environment through its channels. It does not interact directly with other processes.***

Some processes have *serial* implementations - ***these are just like traditional serial programs***.

Some processes have *parallel* implementations - ***networks of sub-processes (think hardware)***.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict.
This will scale!

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

It was 20 years ago today …

Objects considered harmful …

Modelling complex systems …

Blood clotting …

Twenty Years Ago …

"… improved understanding and architecture independence were the goals of the design by Inmos of the occam multiprocessing language and the Transputer. The goals were achieved by implementation of the abstract ideas of *process algebra* and with an efficiency that is today almost unimaginable and certainly unmatchable."

C.A.R.Hoare, March 2004.

2003 …

We have been extending the classical (CSP) occam language with ideas of mobility and dynamic network reconfiguration which are taken from Milner's π-calculus (occam-π).

We have found ways of implementing these extensions that still involve significantly less resource overhead than that imposed by the higher level – *but less structured, informal and non-compositional* – **concurrency primitives of existing languages (such as Java) or libraries (such as POSIX threads).**

2003 …

We have been extending the classical (CSP) occam language with ideas of mobility and dynamic network reconfiguration which are taken from Milner's π-calculus (occam-π).

As a result, we can run applications with the order of *millions* of concurrent processes on modestly powered PCs. We have plans to extend the system, without sacrifice of too much efficiency and none of logic, to simple clusters of workstations, wider networks such as the Grid and small embedded devices.

2003 …

In the interests of proveability, we have been careful to preserve the distinction between the original static point-to-point synchronised communication of *occam* and the dynamic asynchronous multiplexed communication of π-calculus; in this, we have been prepared to sacrifice the elegant sparsity of the π-calculus.

We conjecture that the extra complexity and discipline introduced will make the task of developing, proving and maintaining concurrent and distributed programs easier.

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

It was 20 years ago today …

Objects considered harmful …

Modelling complex systems …

Blood clotting …

Java Monitors – *Concerns*

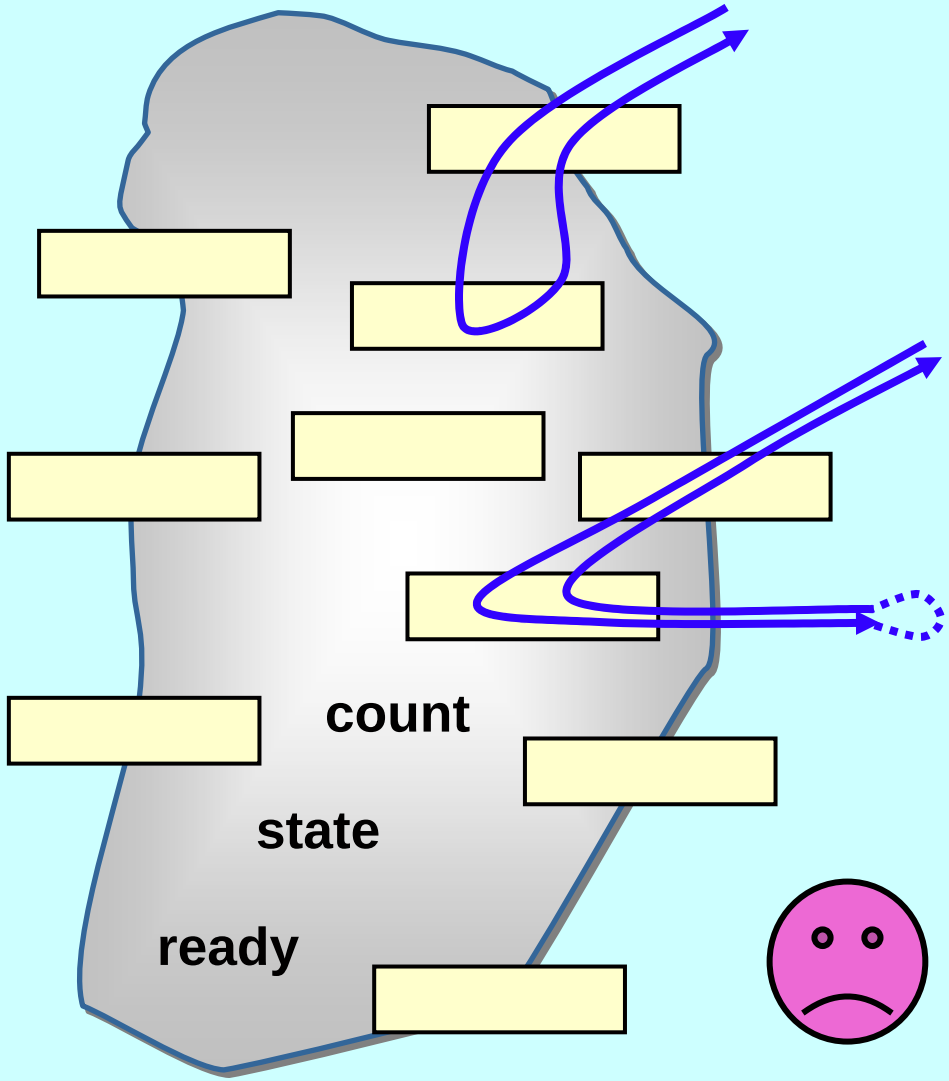
- ***Easy to learn*** - but ***very difficult to apply*** … ***safely*** …
- Monitor methods are ***tightly interdependent*** - their semantics compose in ***complex ways*** … the whole skill lies in setting up and staying in control of these complex interactions …
- Threads have no structure … there are no ***threads within threads*** …
- Big problems when it comes to ***scaling up complexity*** …

Objects Considered Harmful

Most objects are dead - they have no life of their own.

All methods have to be invoked by an external thread of control - they have to be *caller oriented* …

… a somewhat curious property of 'object oriented' design.

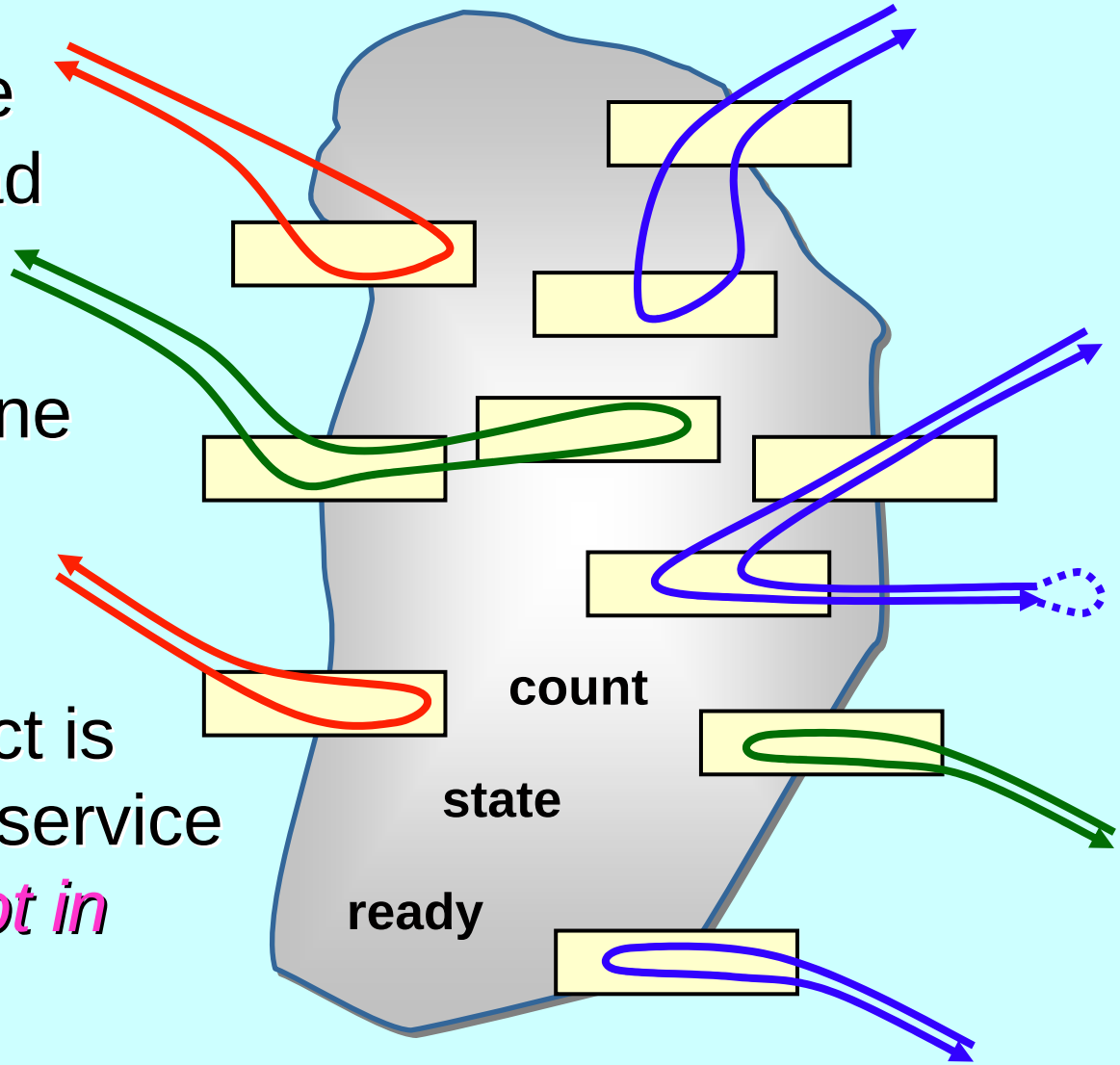


Objects Considered Harmful

The object is at the mercy of *any* thread that sees it.

Nothing can be done to prevent method invocation ...

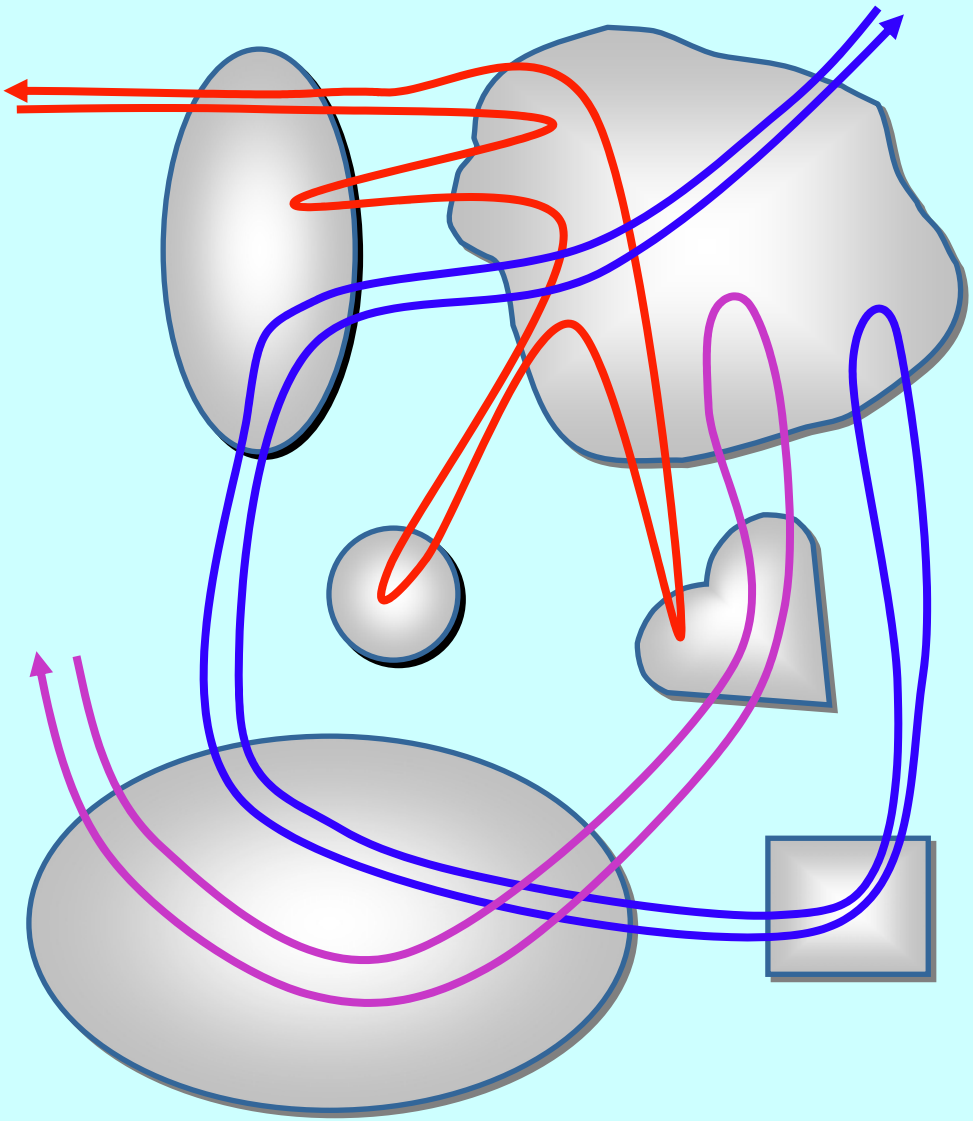
… even if the object is not in a fit state to service it. *The object is not in control of its life.*



Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

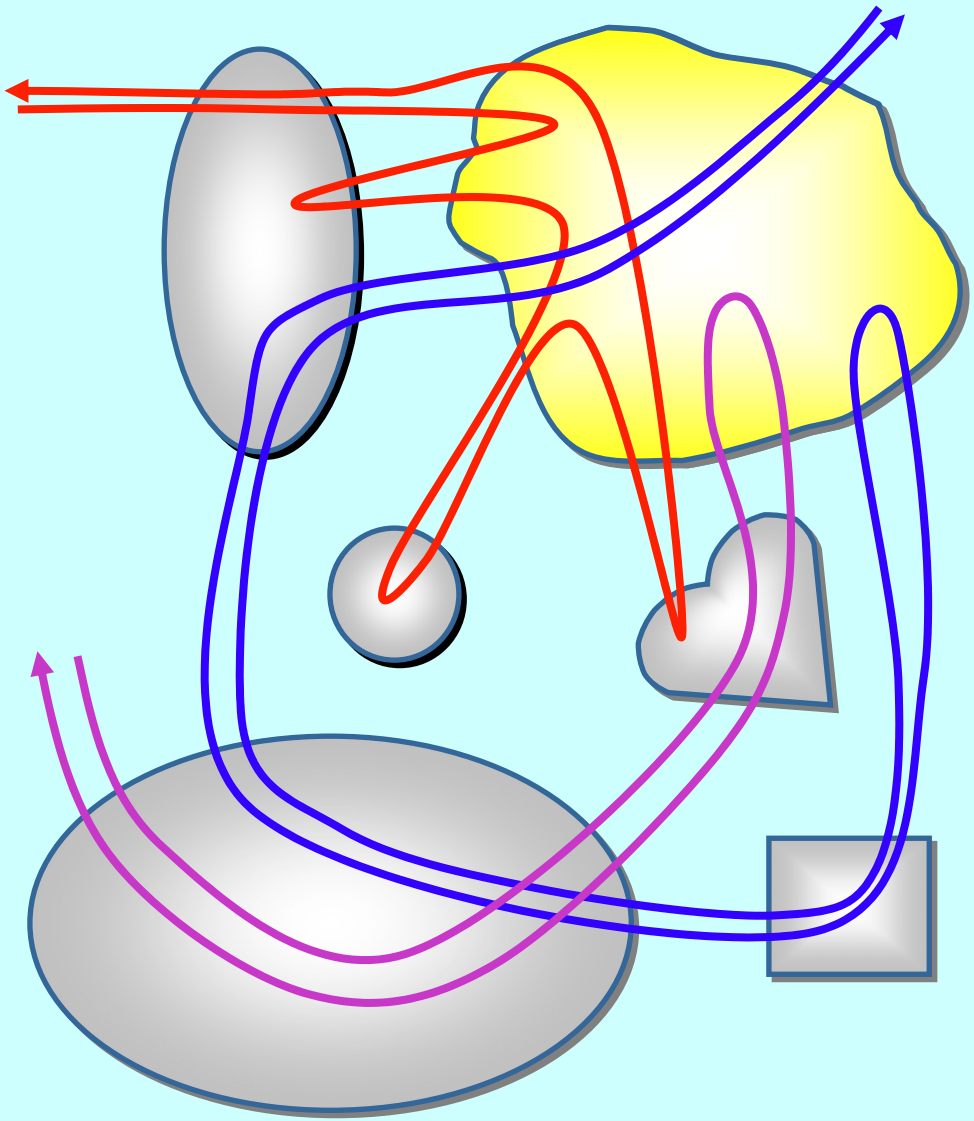
Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure*.



Threads-n-Locks Considered Harmful

Each object is at the mercy of ***any*** thread that sees it. Nothing can be done to ***prevent*** method invocation … even if the object is not in a fit state to service it. ***The object is not in control of its life***.

Big problems occur when multiple threads hit the same object.



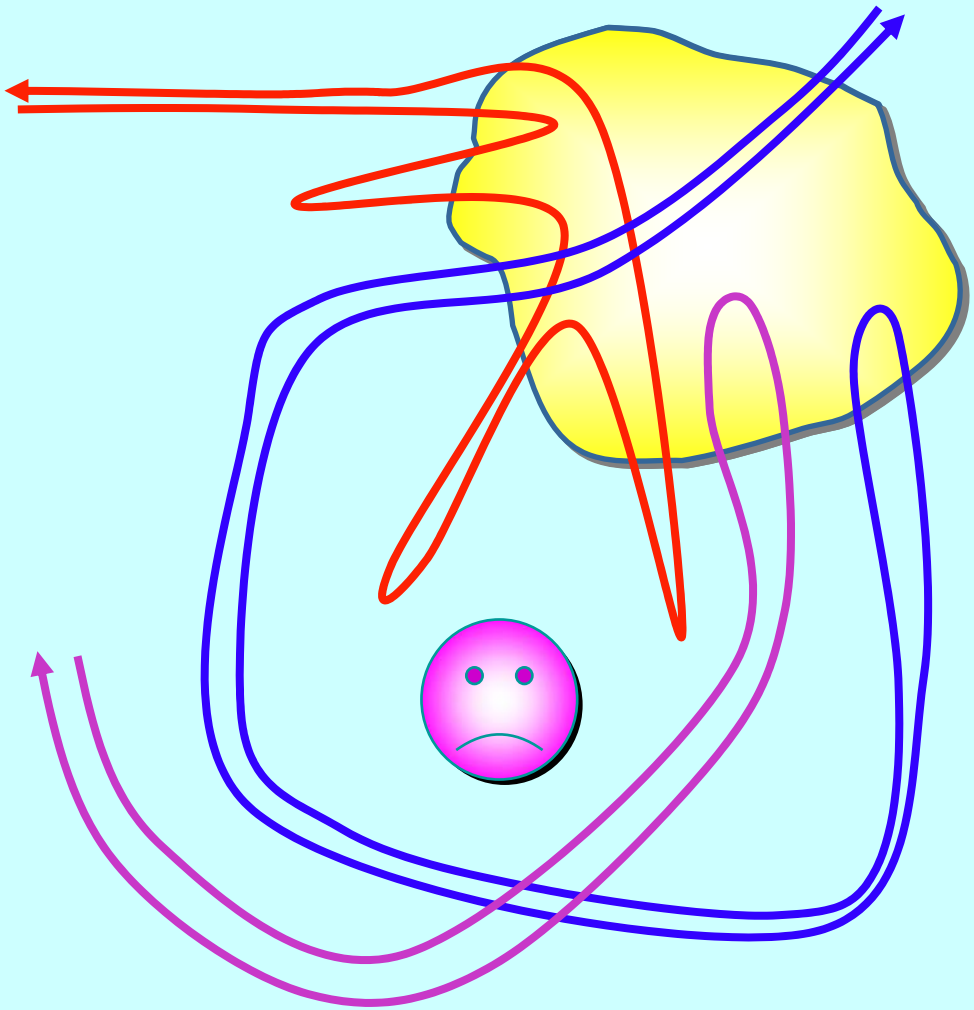
Threads-n-Locks Considered Harmful

Errors in claiming/releasing locks is probably the main reason our systems fail …

Too much locking and we get deadlock …

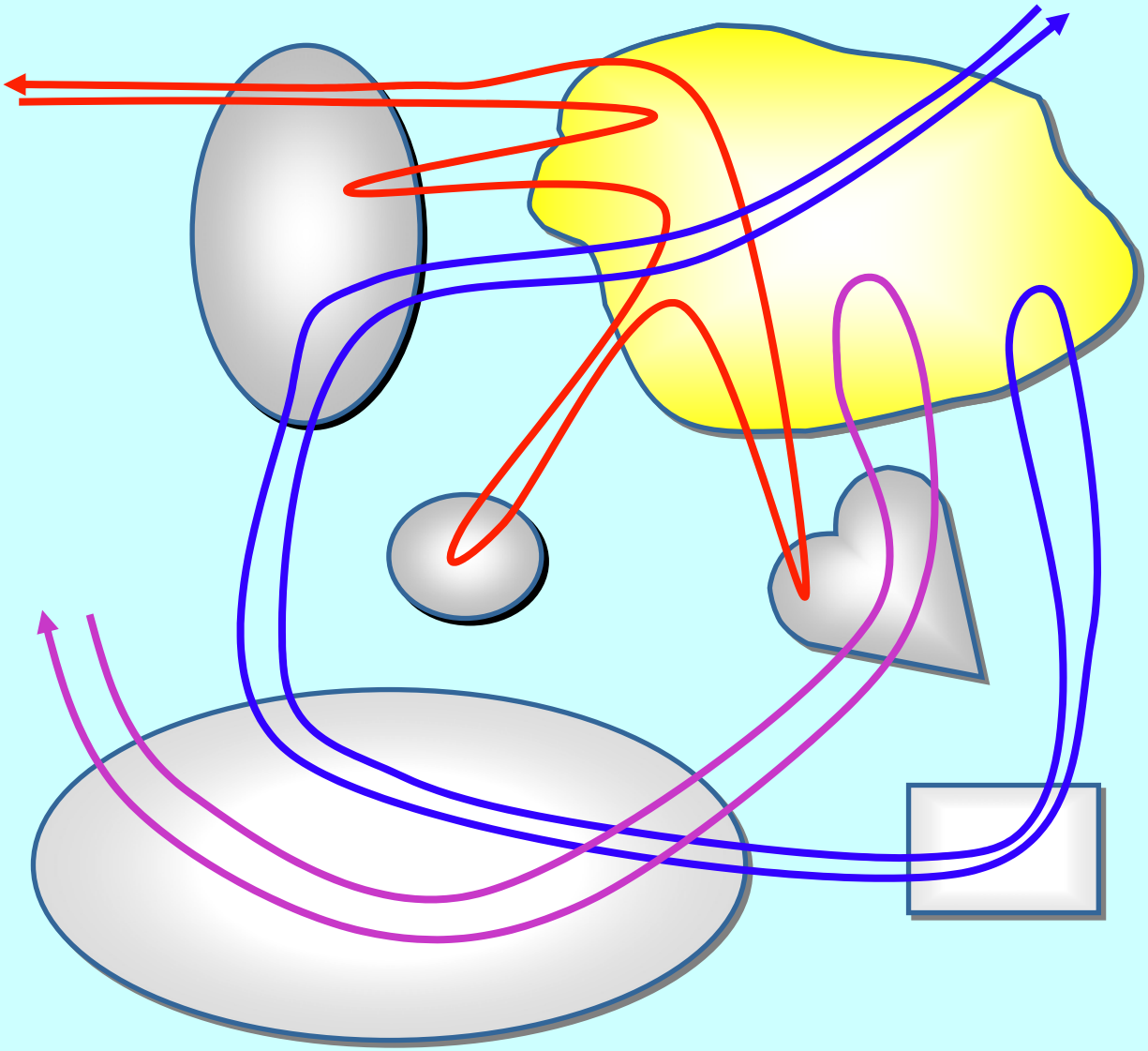
Too little locking and race hazards slowly corrupt …

Sorting this out requires controlling all possible interleavings … which is exponential in the number of threads …

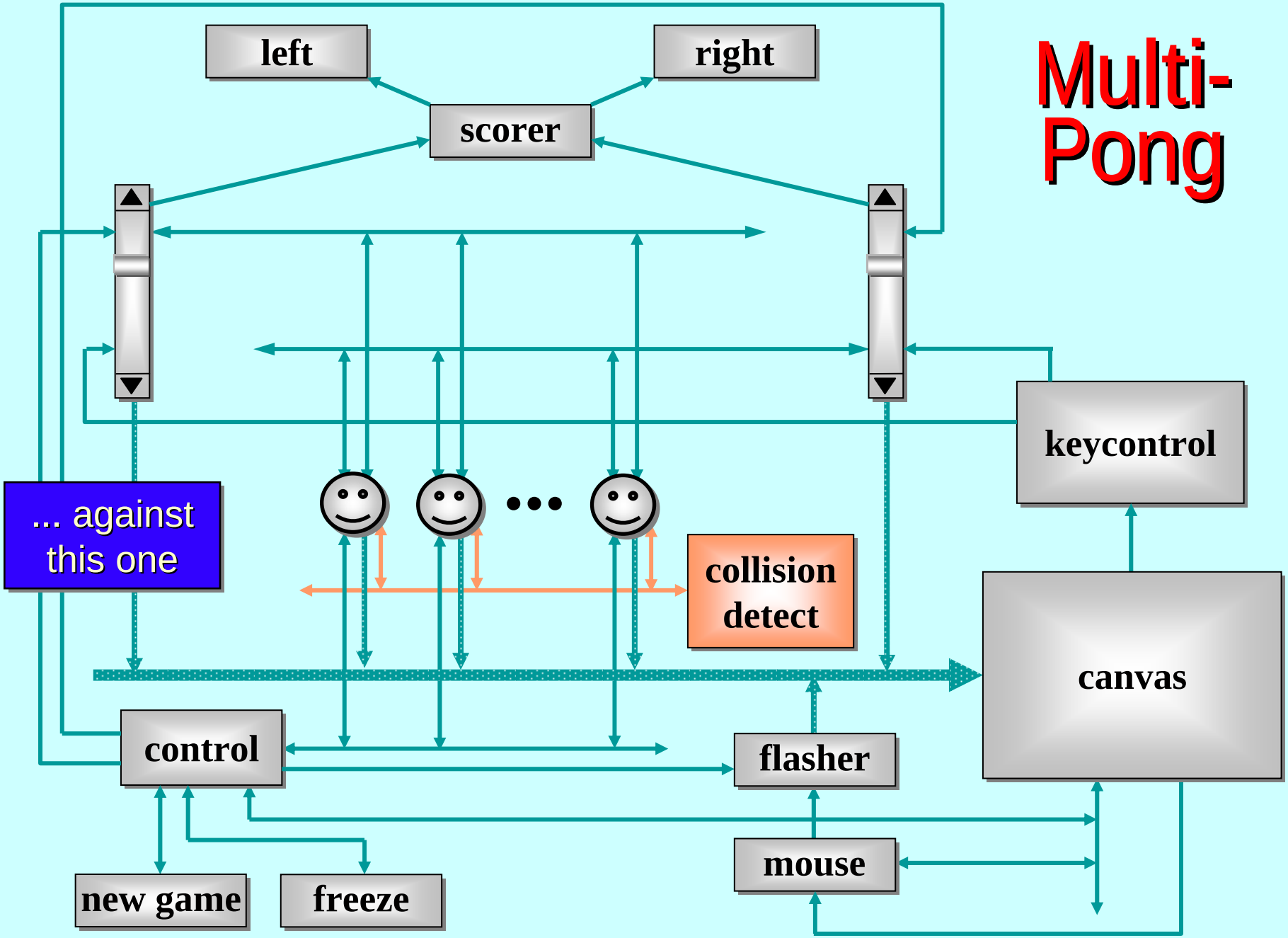


Threads-n-Locks Considered Harmful

Compare this design structure ...



Multi-Pong



... for example ...

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

It was 20 years ago today …

Objects considered harmful …

Modelling complex systems …

Blood clotting …

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 …

Modelling Nannite-Assemblies

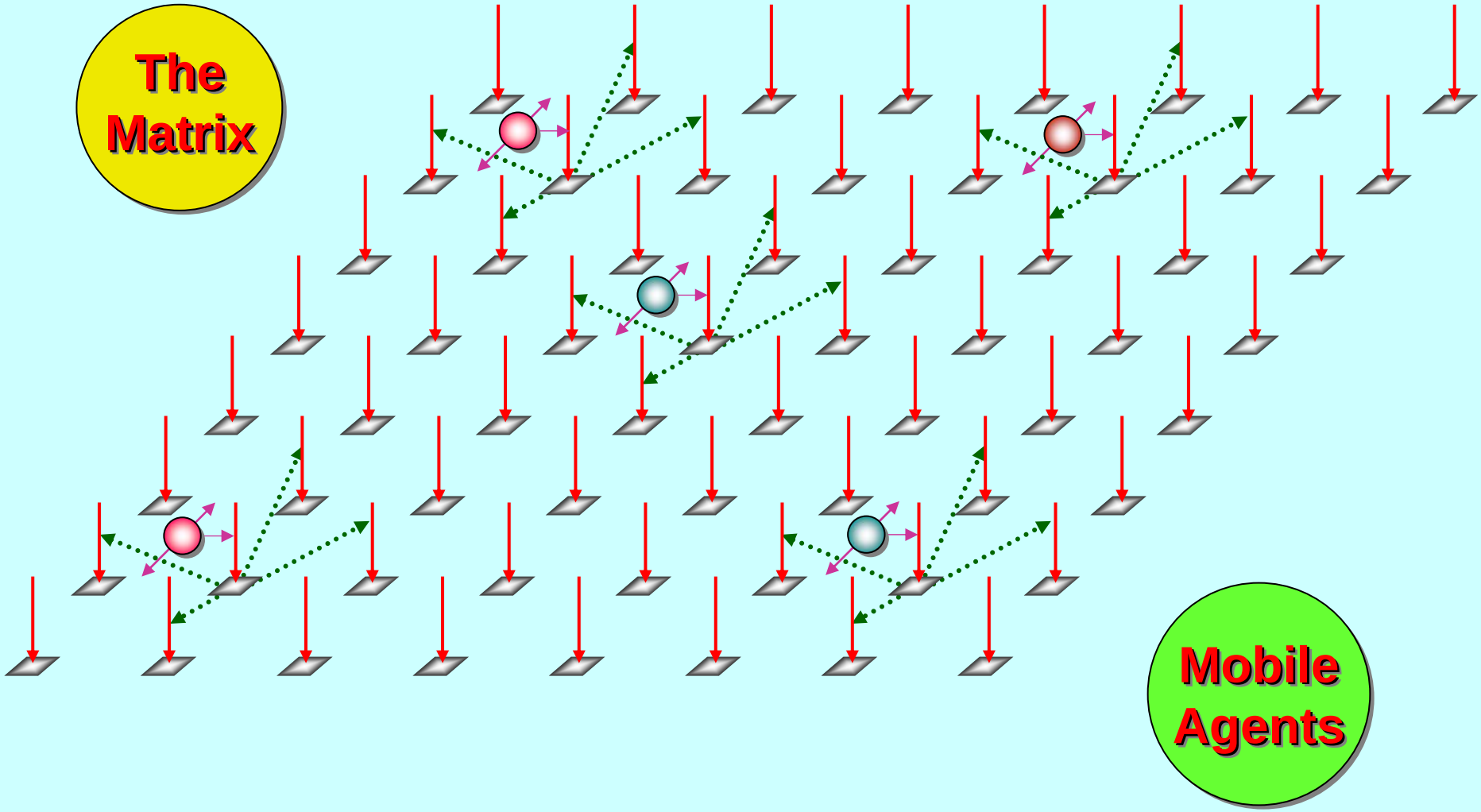
- **TUNA: Theory Underpinning Nanotech Assemblies**
  - Active ***nano-devices*** that manipulate the world at ***nano-scale*** to have ***macroscopic*** effects (e.g. through assembling artifacts).
  - Need vast numbers of them – but these can grow (exponentially).
  - Need capabilities to design, program and control complex and dynamic networks – build desired artifacts, not undesired ones.
  - Need a theory of dynamic networks and emergent properties.
- **Implementation 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 …

Funded ☺☺☺ … York, Surrey and Kent

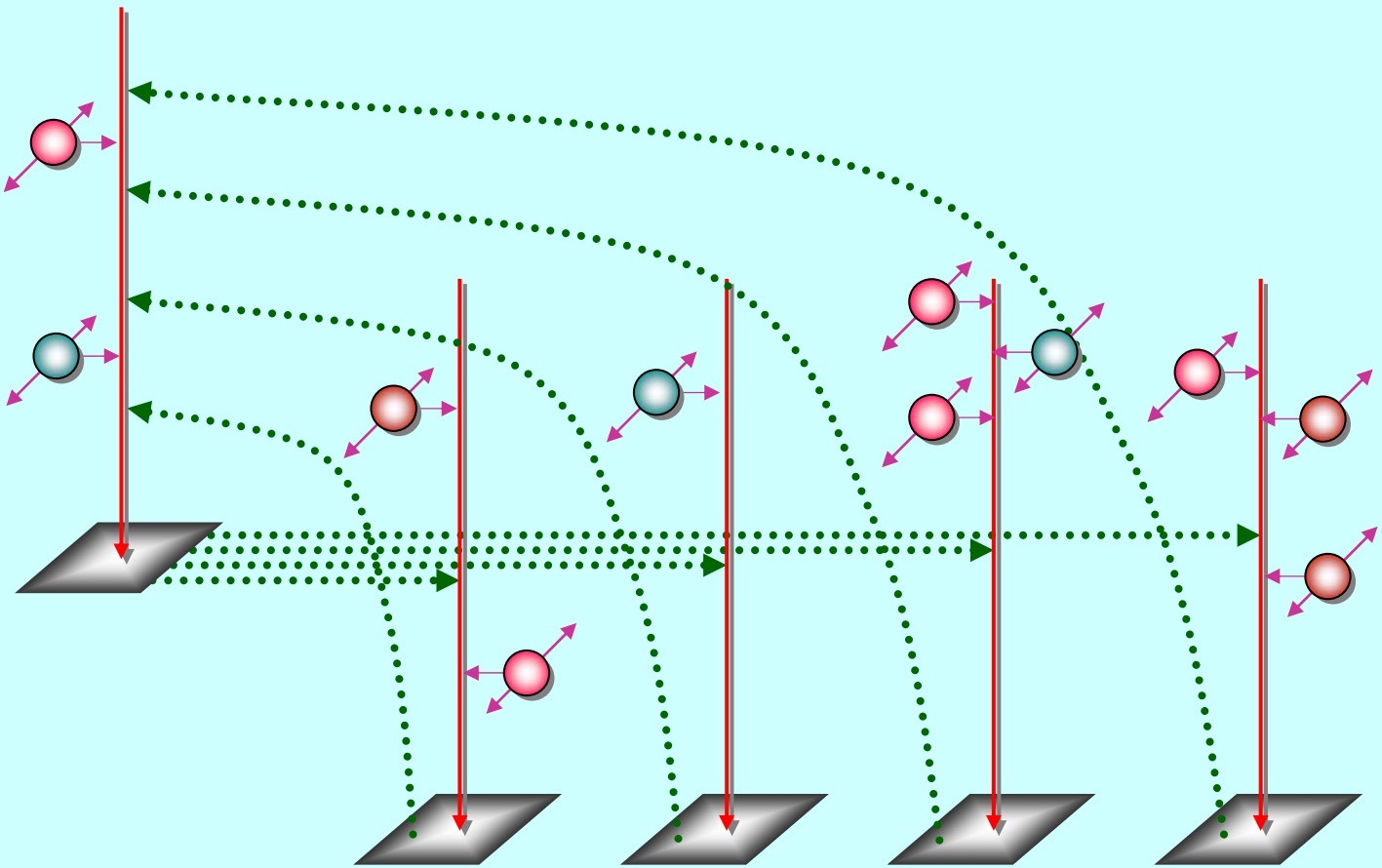
Mobility and Location Awareness

- **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?

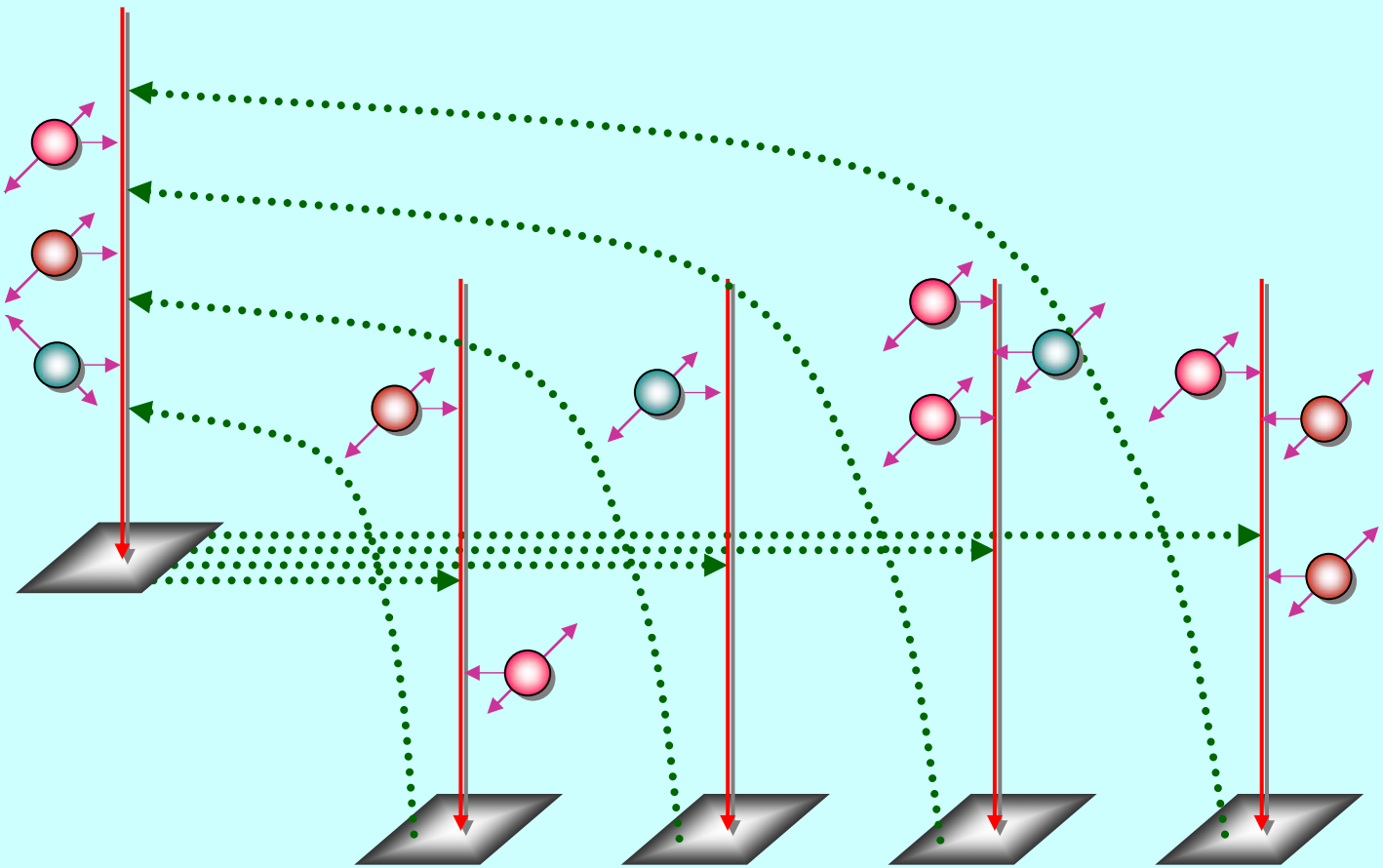
Location (Neighbourhood) Awareness



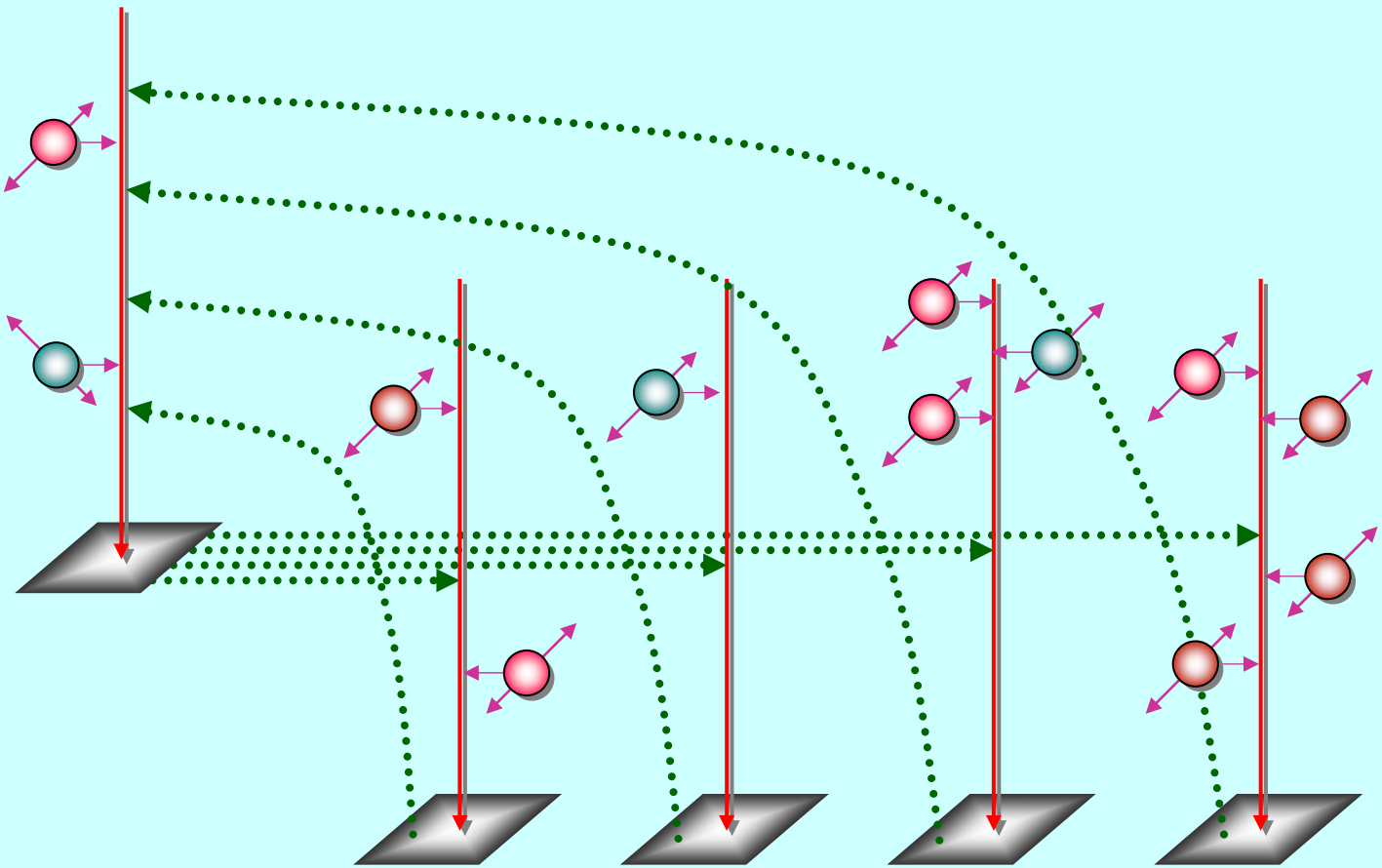
Location (Neighbourhood) Awareness



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. Central points of control do not remain stable for long.
  - Natural systems are 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 ***hasten*** the construction, commisioning and maintenance of systems.

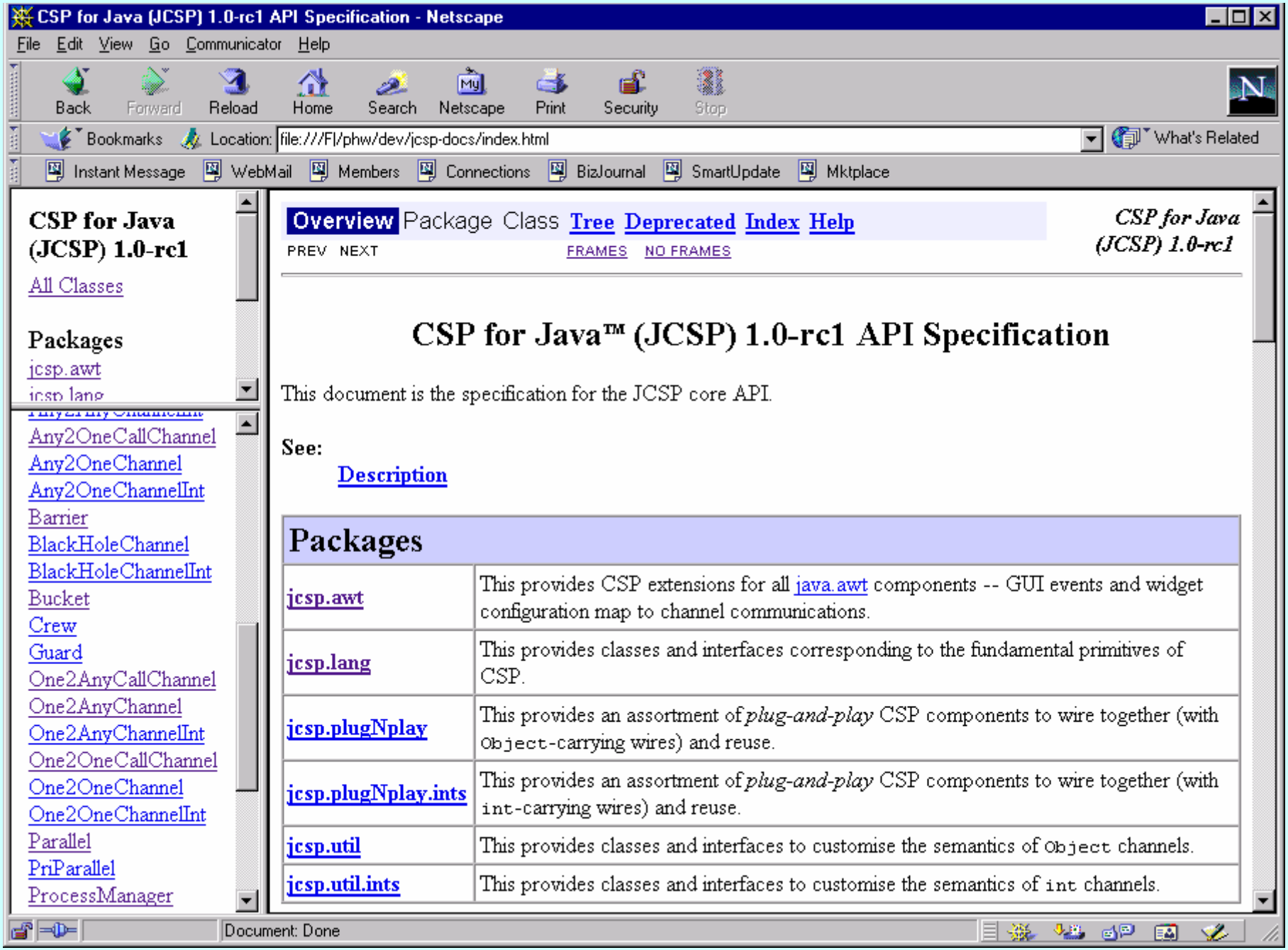
- **Hypothesis**
  - The wrong turn can be corrected and this correction is needed now.

Putting CSP into practice …

http://www.cs.ukc.ac.uk/projects/ofa/kroc/

Putting CSP into practice …

http://www.cs.ukc.ac.uk/projects/ofa/jcsp/


CSP for Java (JCSP) 1.0-rc1 API Specification - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///F|/phw/dev/jcsp-docs/index.html What's Related
Instant Message WebMail Members Connections BizJournal SmartUpdate Mktplace
CSP for Java (JCSP) 1.0-rc1
All Classes
Packages
jcsp.awt
jcsp.lang
Any2OneCallChannel
Any2OneChannel
Any2OneChannelInt
Barrier
BlackHoleChannel
BlackHoleChannelInt
Bucket
Crew
Guard
One2AnyCallChannel
One2AnyChannel
One2AnyChannelInt
One2OneCallChannel
One2OneChannel
One2OneChannelInt
Parallel
PriParallel
ProcessManager
Overview Package Class Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES
CSP for Java (JCSP) 1.0-rc1
CSP for Java™ (JCSP) 1.0-rc1 API Specification
This document is the specification for the JCSP core API.
See:
Description
Packages
jcsp.awt
This provides CSP extensions for all java.awt components -- GUI events and widget configuration map to channel communications.
jcsp.lang
This provides classes and interfaces corresponding to the fundamental primitives of CSP.
jcsp.plugNplay
This provides an assortment of plug-and-play CSP components to wire together (with Object-carrying wires) and reuse.
jcsp.plugNplay.ints
This provides an assortment of plug-and-play CSP components to wire together (with int-carrying wires) and reuse.
jcsp.util
This provides classes and interfaces to customise the semantics of Object channels.
jcsp.util.ints
This provides classes and interfaces to customise the semantics of int channels.
Document: Done

Motivation: Concurrency for All

Nature is not serial ...

Components must compose …

Nature is concurrent …

It was 20 years ago today …

Objects considered harmful …

Modelling complex systems …

Blood clotting …

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.

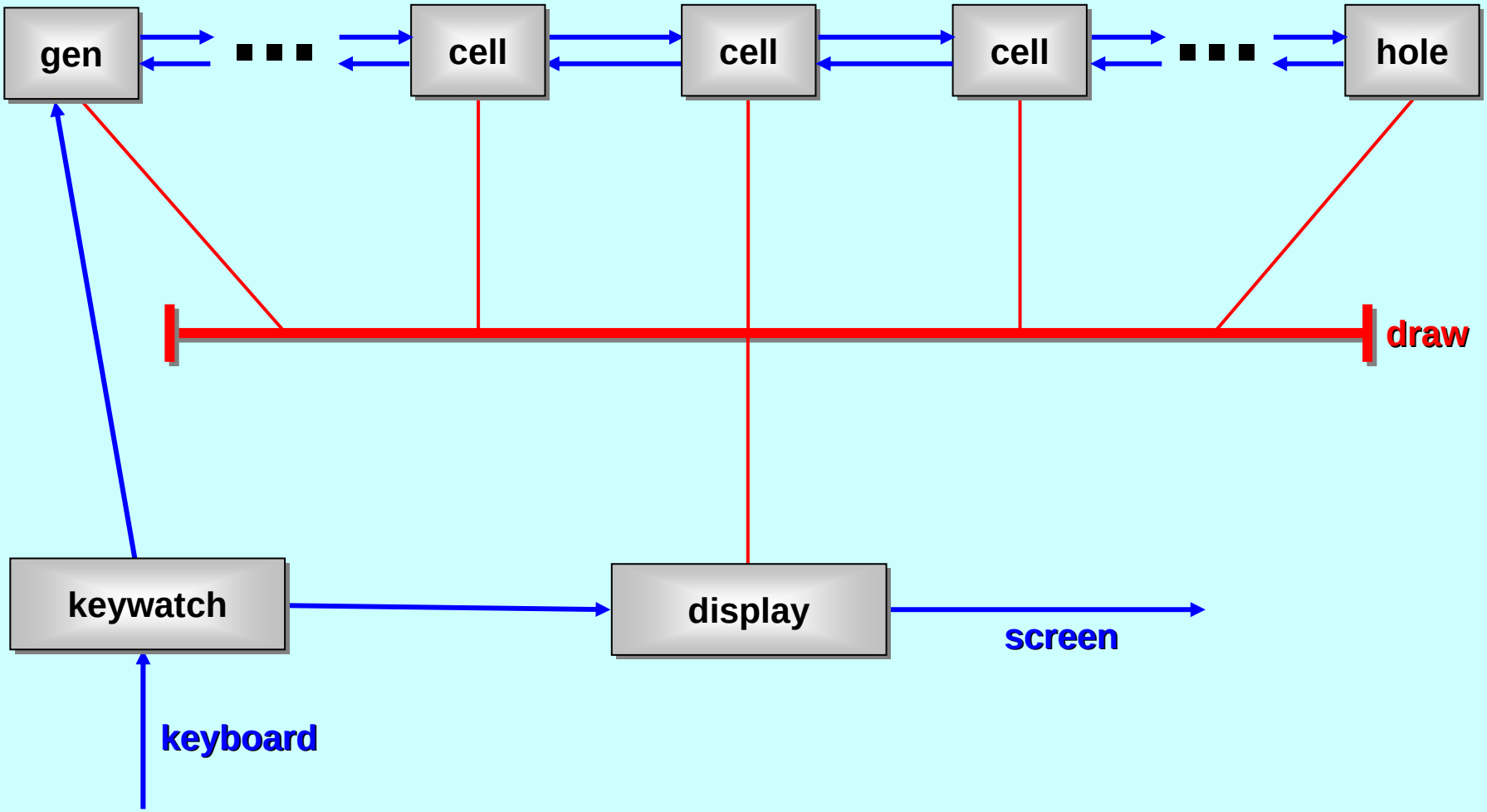
Platelet Model ('busy' CA)

Space is represented as a pipeline of ***cell*** processes. Activated (i.e. *sticky*) ***platelets*** are generated and injected into the pipeline at a user-determined randomised rate. They move through the ***cells*** at speeds inversely proportional to the size of the ***clot*** in which they become embedded – these speeds are randomised slightly. ***Clots*** that bump together stay together.

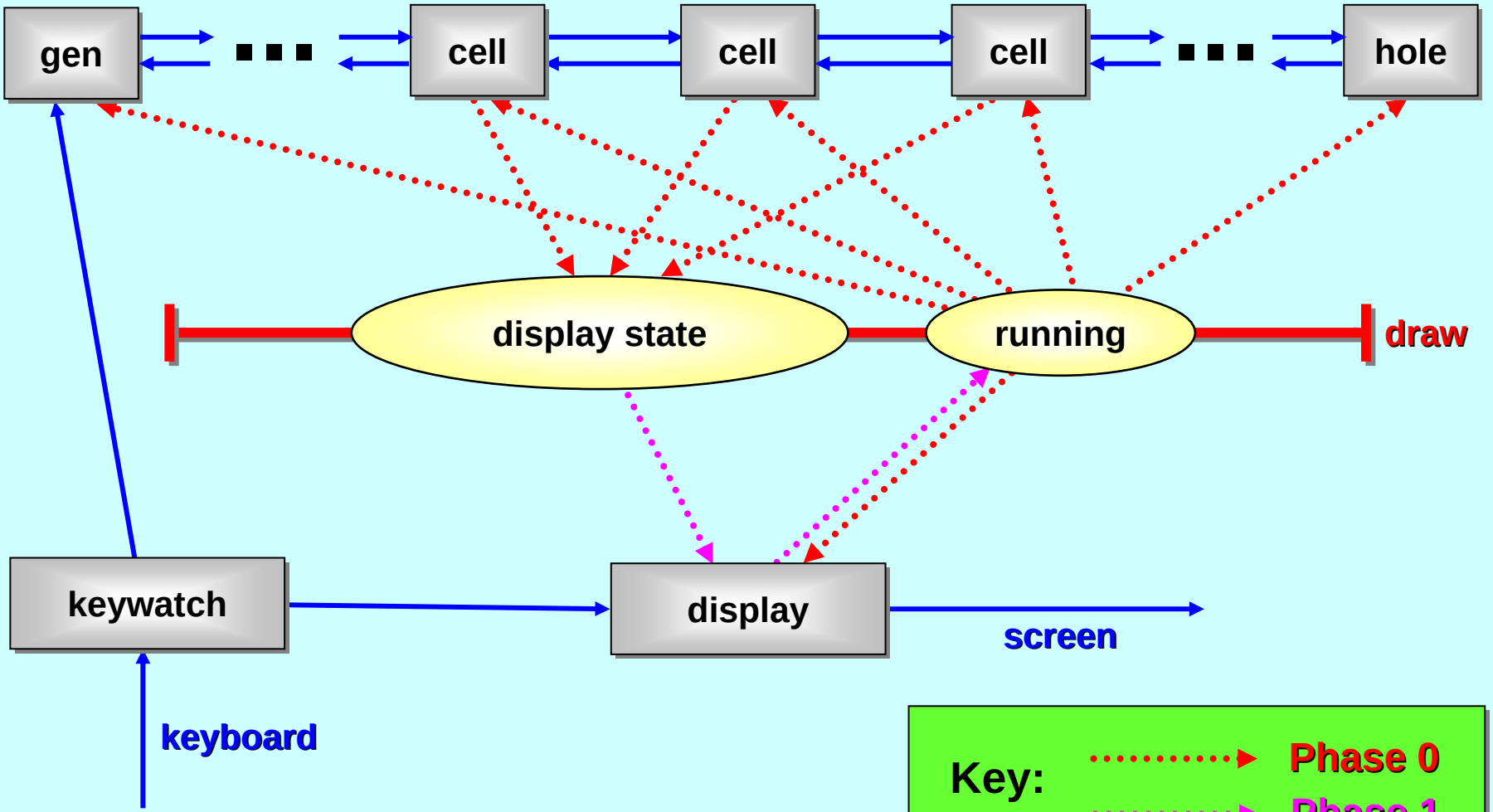
The ***cells*** do all the work and work all the time, even when empty. ***Platelets***/***clots*** pass through them – at which times, the ***cells*** compute part of their life-cycle.

Platelets/***clots*** are not directly modelled as processes.

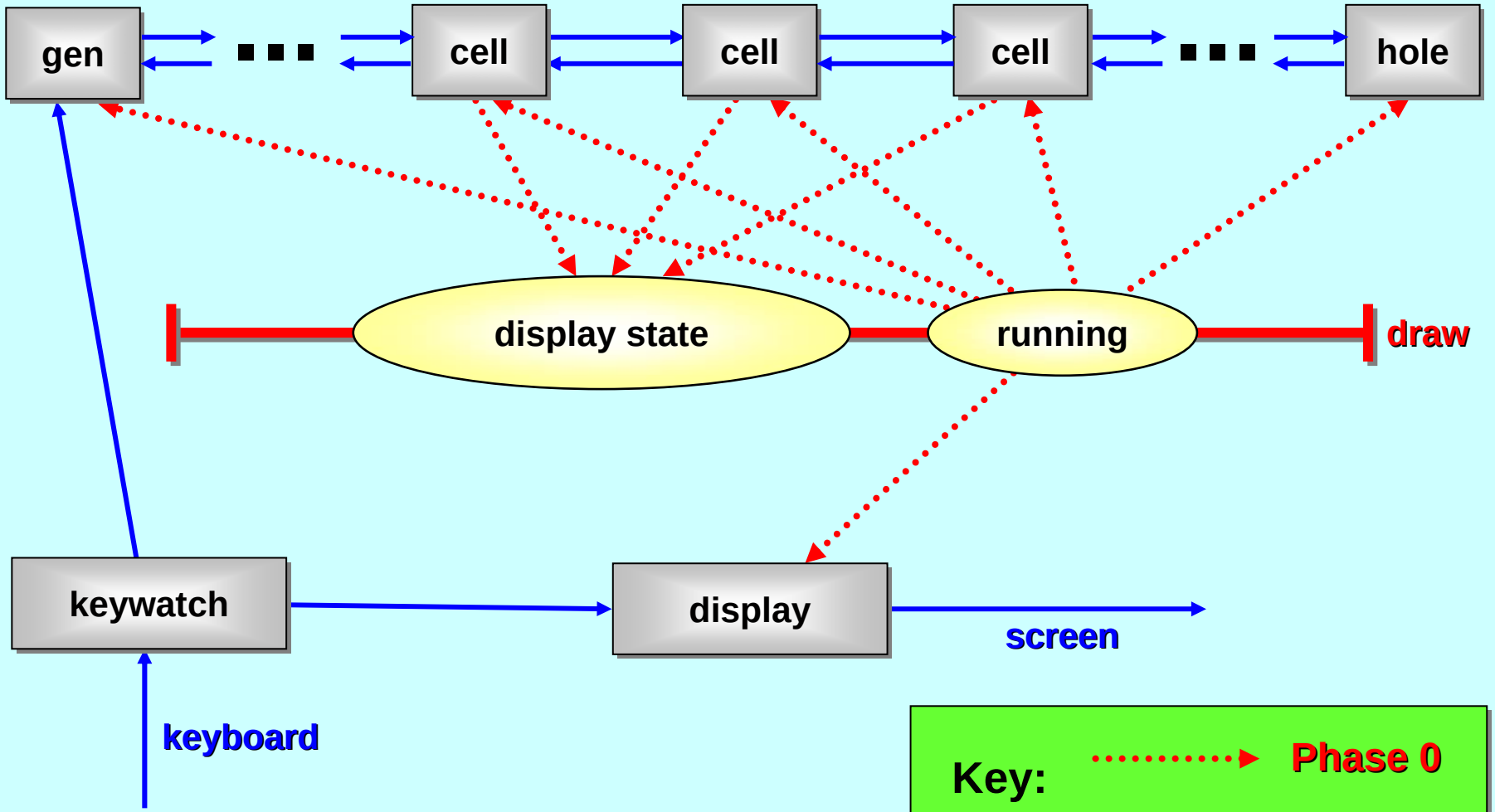
Platelet Model ('busy' CA)



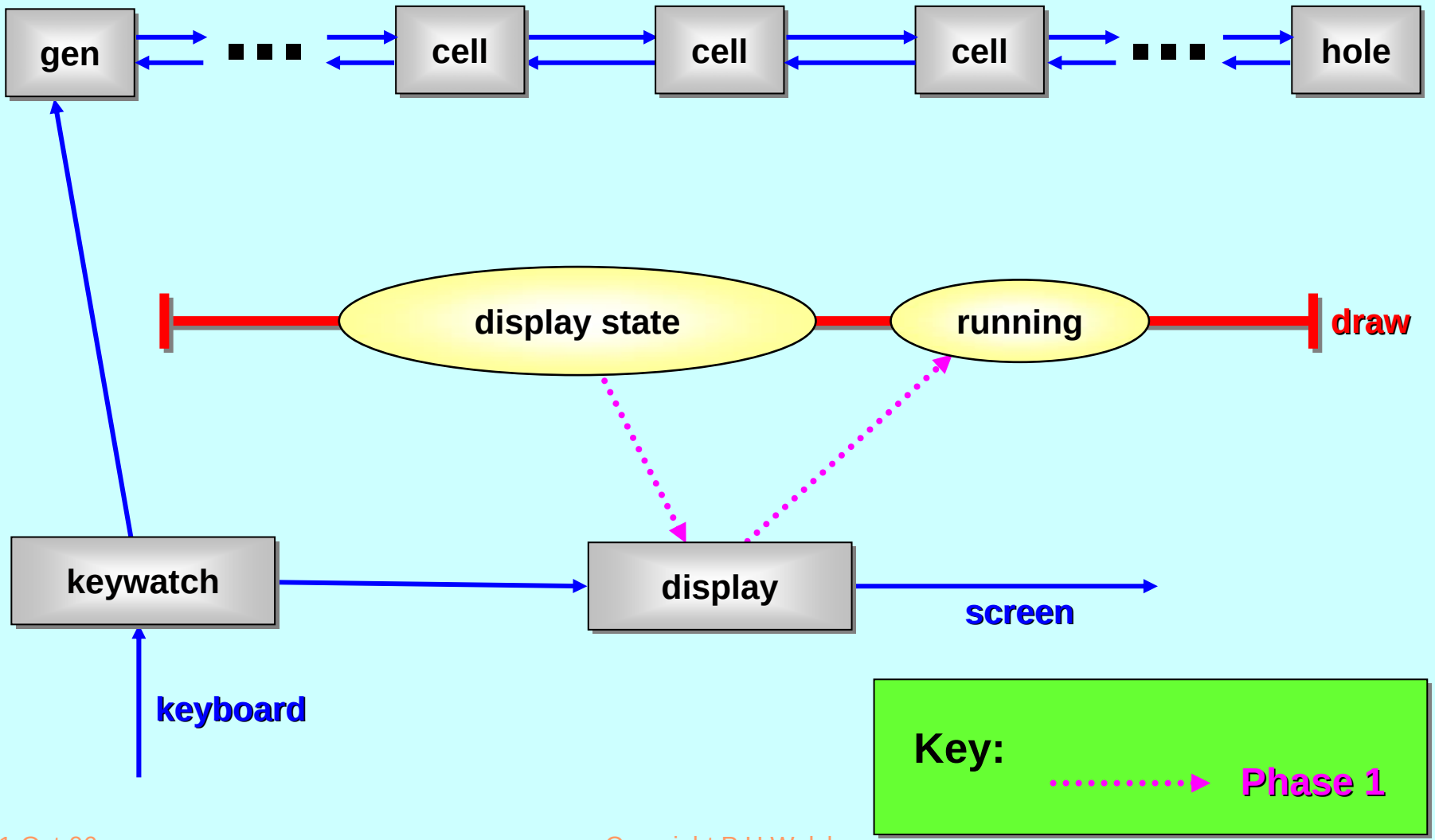
Platelet Model ('busy' CA)



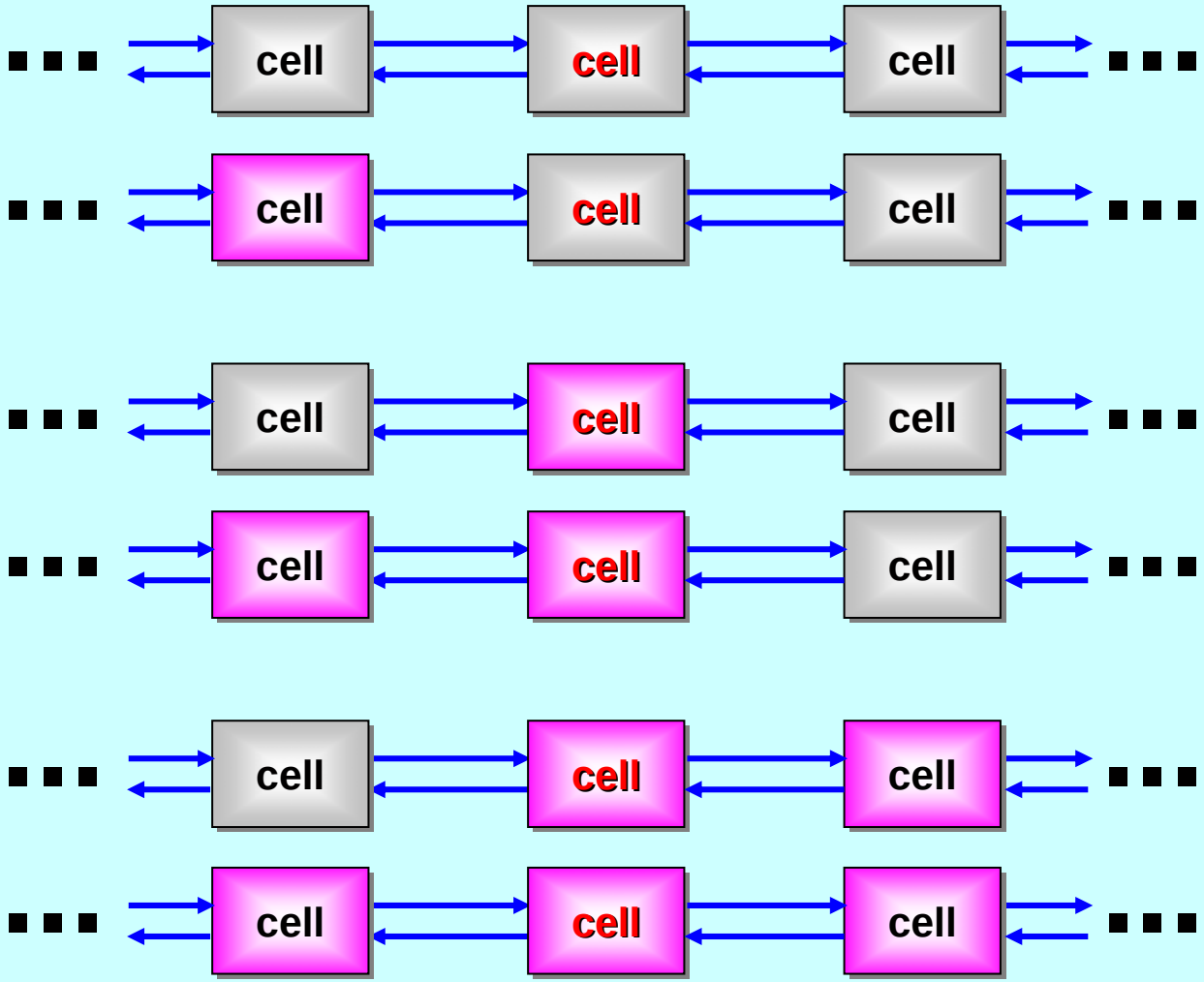
Platelet Model ('busy' CA)



Platelet Model ('busy' CA)



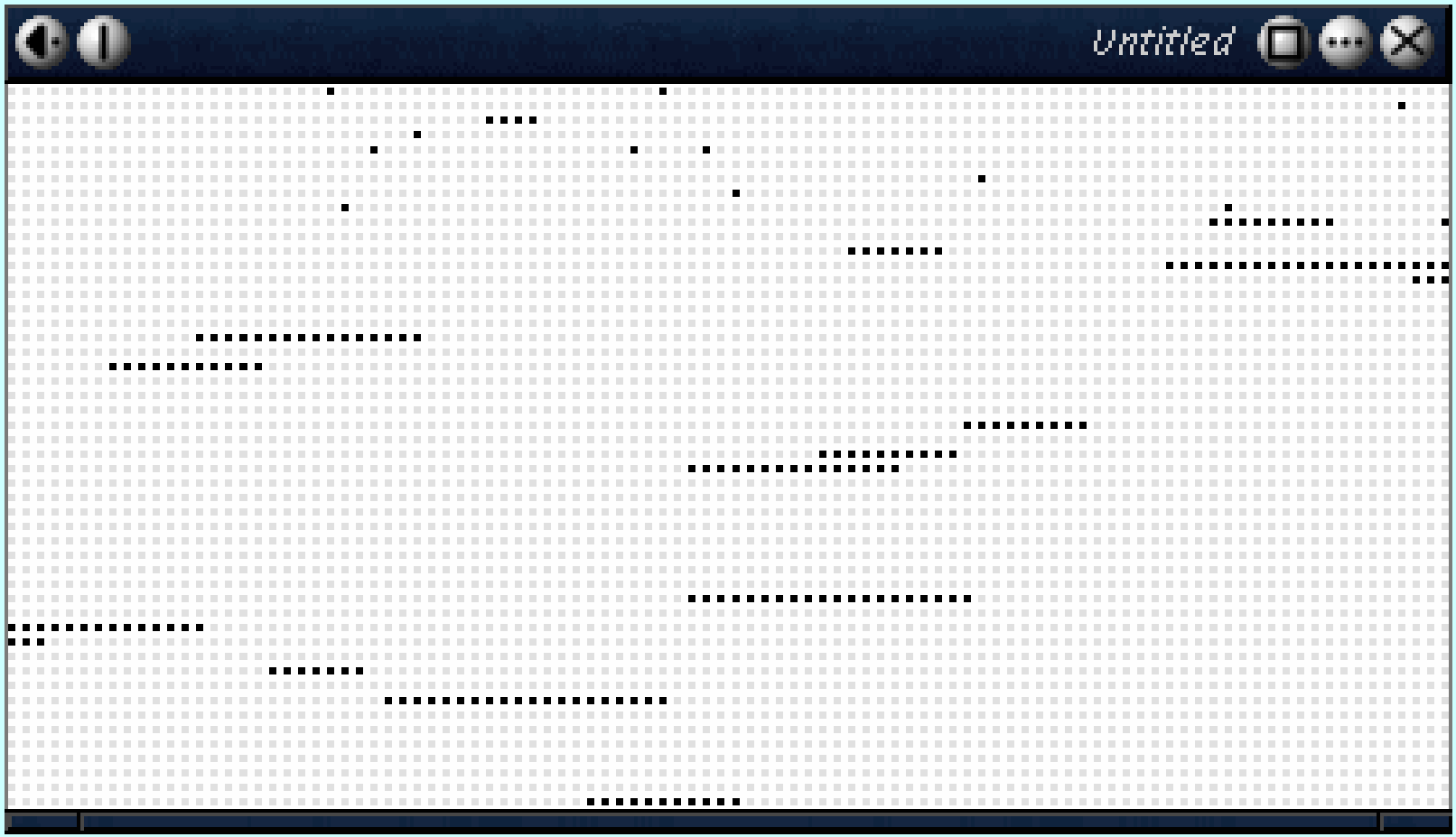
Platelet Model ('busy' CA)



Platelet Model ('busy' CA)

```
PROC cell (BYTE my.visible.state, BOOL running, BARRIER draw,
           CHAN CELL.CELL l.in?, l.out!, r.in?, r.out!)
  ...  local declarations / initialisation  (phase 0)
  WHILE running
    SEQ
      SYNC draw     -- phase 1
      ...  PAR-I/O exchange of full/empty state
      ...  if full,
      ...    discover clump size (pass count through)
      ...    if head,
      ...      decide on move (non-deterministic choice)
      ...      if move, tell empty cell ahead
      ...    else receive decision on move from cell ahead
      ...    if not tail, pass decision back
      ...    if tail and move, become empty
      ...  else if clump behind exists and moves, become full
      SYNC draw     -- phase 0
      ...  update my.visible.state
:
```

Platelet Model (Visualisation)



Platelet Model ('busy' CA)

Performance: each ***cell*** has to work harder if full (carrying a ***platelet***). Also, ***clot*** sizes are recomputed every cycle – so large clumps increase the cost. (2.4 GHz. P IV *'mobile'*).

Generate probability (n / 256)	Cell cycle time (ns)
0	650
1	660
2	670
4	680
8	700
16	740
32	1070 (total jam)

Platelet Model ('busy' → 'lazy' CA)

Scaling problem: every ***cell*** is active every cycle – regardless of whether it contains a platelet. This works well for systems with up to ~100K ***cells***.

For **TUNA**, we will need to be working in 3D (say, ~10M ***cells***), modelling many different types of agent with much richer rules of engagement.

These automata must become ***'lazy'***, whereby only processes with things to do remain in the computation.

Platelet Model ('busy' → 'lazy' CA)

Logical problem: the rules for the different stages in the life cycle of ***platelets***, or ***clots***, are coded into different cycles of the cells. Each ***cell*** sees lots of different ***platelets*** – sometimes bunched together as ***clots*** – and operates on them as they pass through.

No process directly models the development of a single ***clot***.

The following system addresses this. The ***cell*** processes are pure ***servers***, not enrolled on the time-synchronising ***barrier***. Their ***clients*** are ***clot*** processes, *generated dynamically*, that are enrolled on the ***barrier*** and use that ***barrier*** to synchronise access to the ***cell*** servers with their generator and the display.

The ***cell*** processes are only worked as ***clot boundaries*** pass over them.

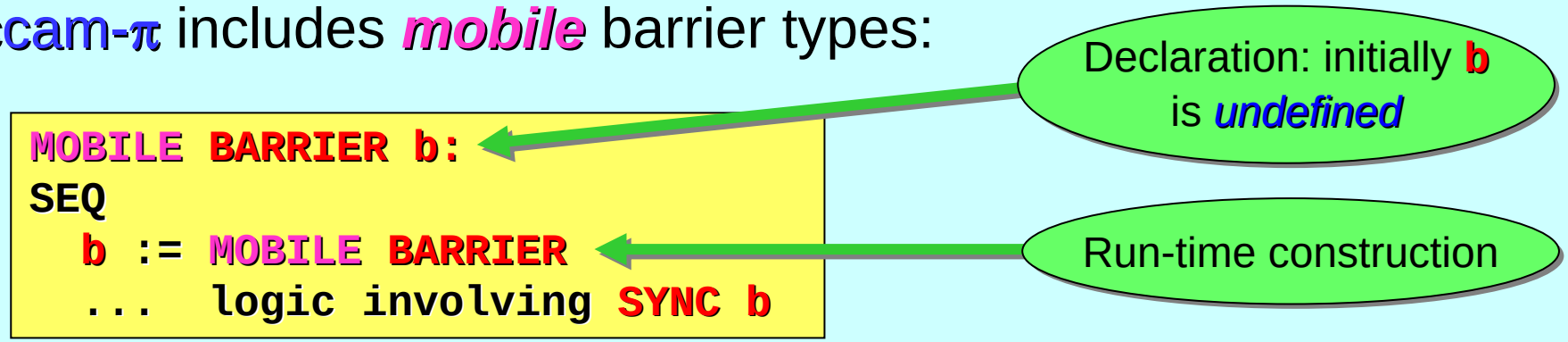
Platelet Model ('busy' → 'lazy' CA)

To manage this, we need to *move* *barriers* to FORKed processes. The general solution is given by making *barriers* MOBILE.

Their ***clien*** *...ally*, that are enrolled on ... synchronise access to the ***cell*** servers ... generator and the display.

Barriers (mobile)

occam-π includes *mobile* barrier types:



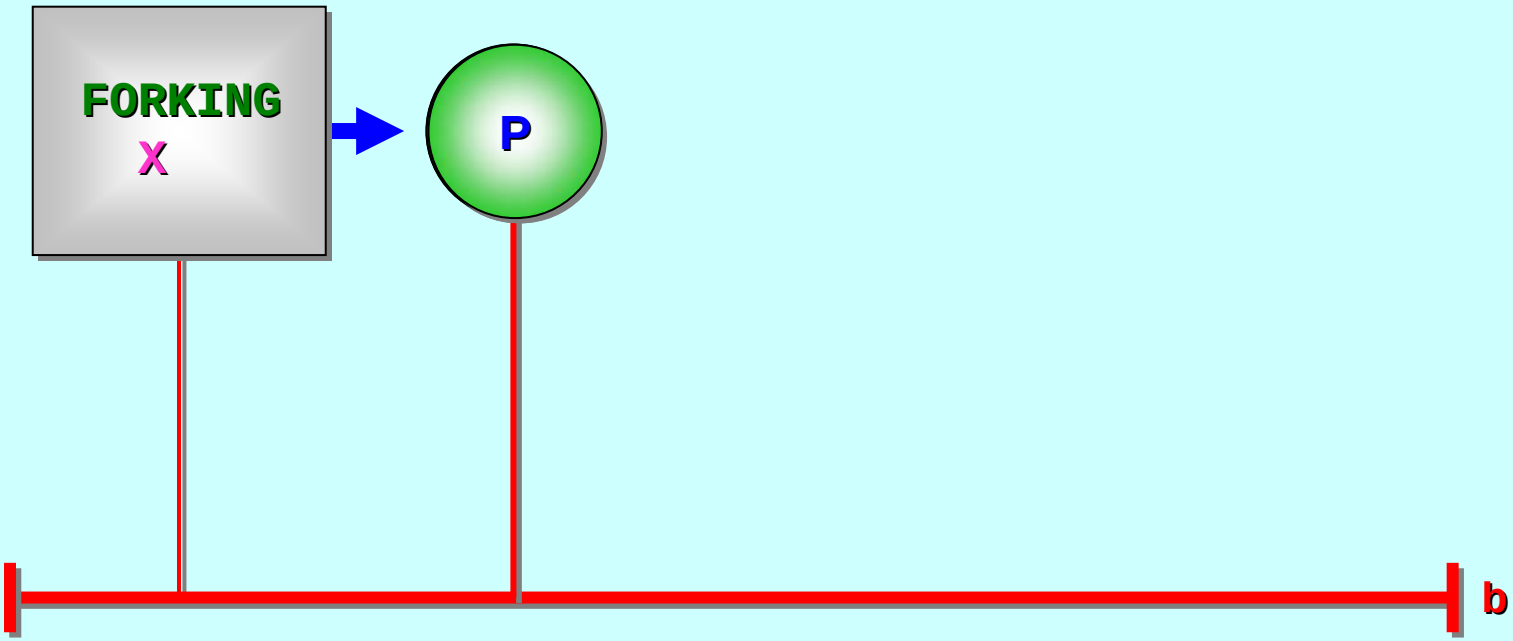
Whenever a barrier is constructed, the process doing the construction becomes *enrolled*.

Whenever a *defined* barrier variable is overwritten or goes out of scope, the process holding it *resigns*.

Channels may carry **MOBILE BARRIER**s as components of their messages (occam-π **PROTOCOL**s).

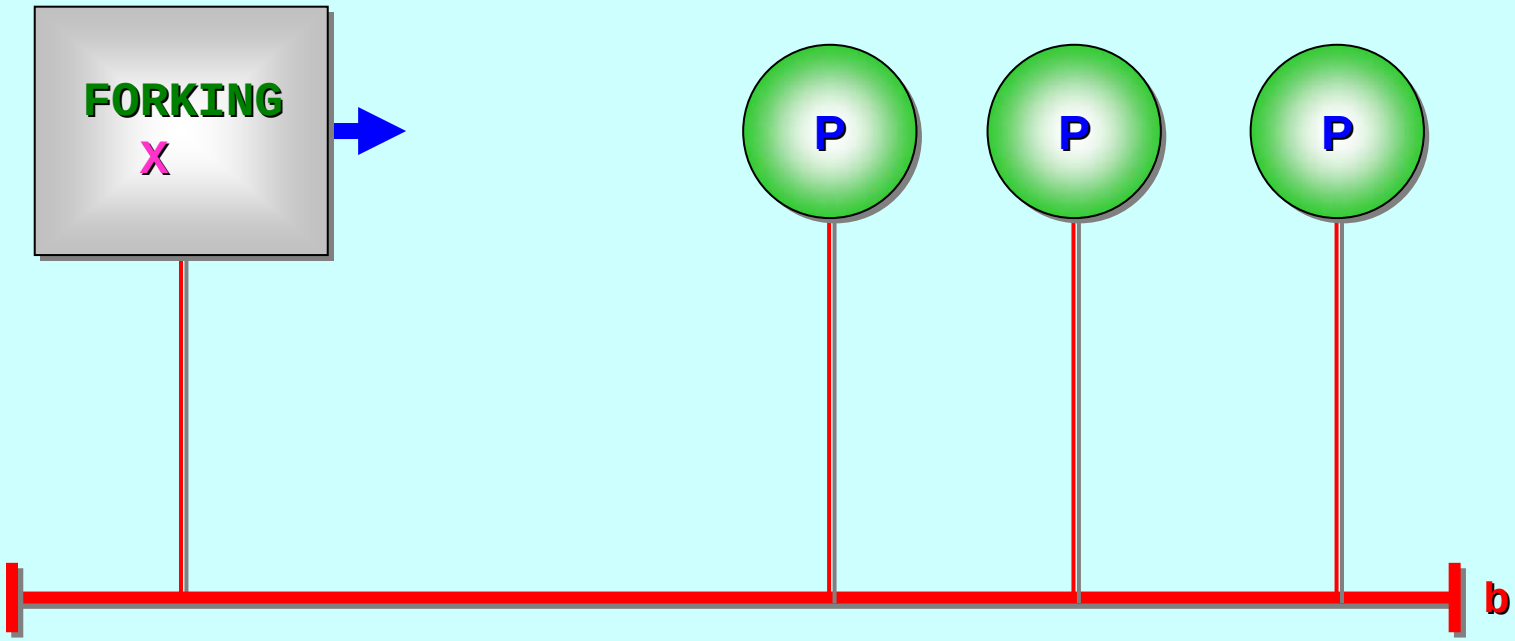
Whenever a barrier is communicated (e.g. to a *forked* process), the receiving process dynamically *enrolls* and the sending process *resigns* (unless a **CLONE** is sent).

Forking Processes with Barriers



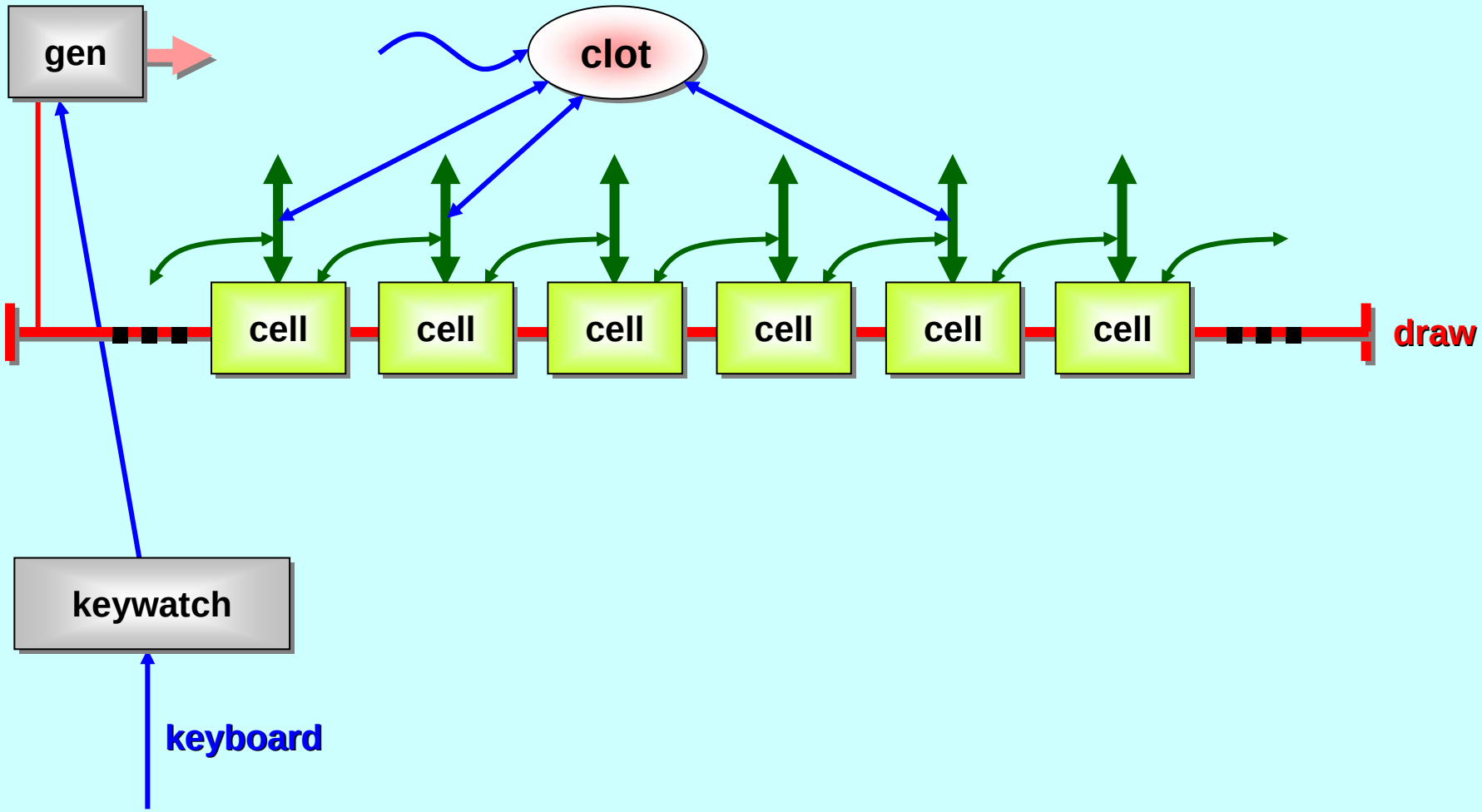
occam-π view

Forking Processes with Barriers

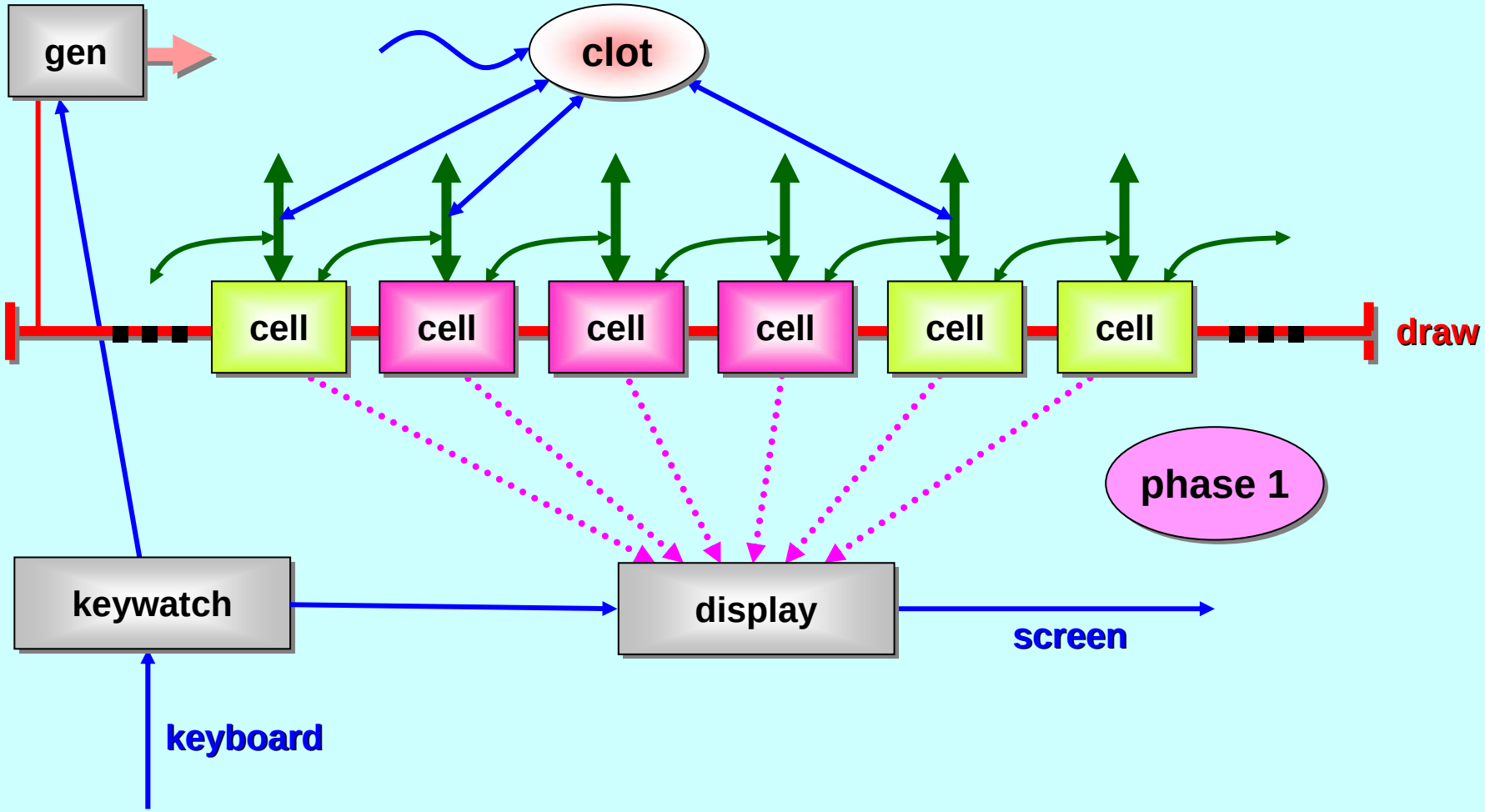


occam-π view

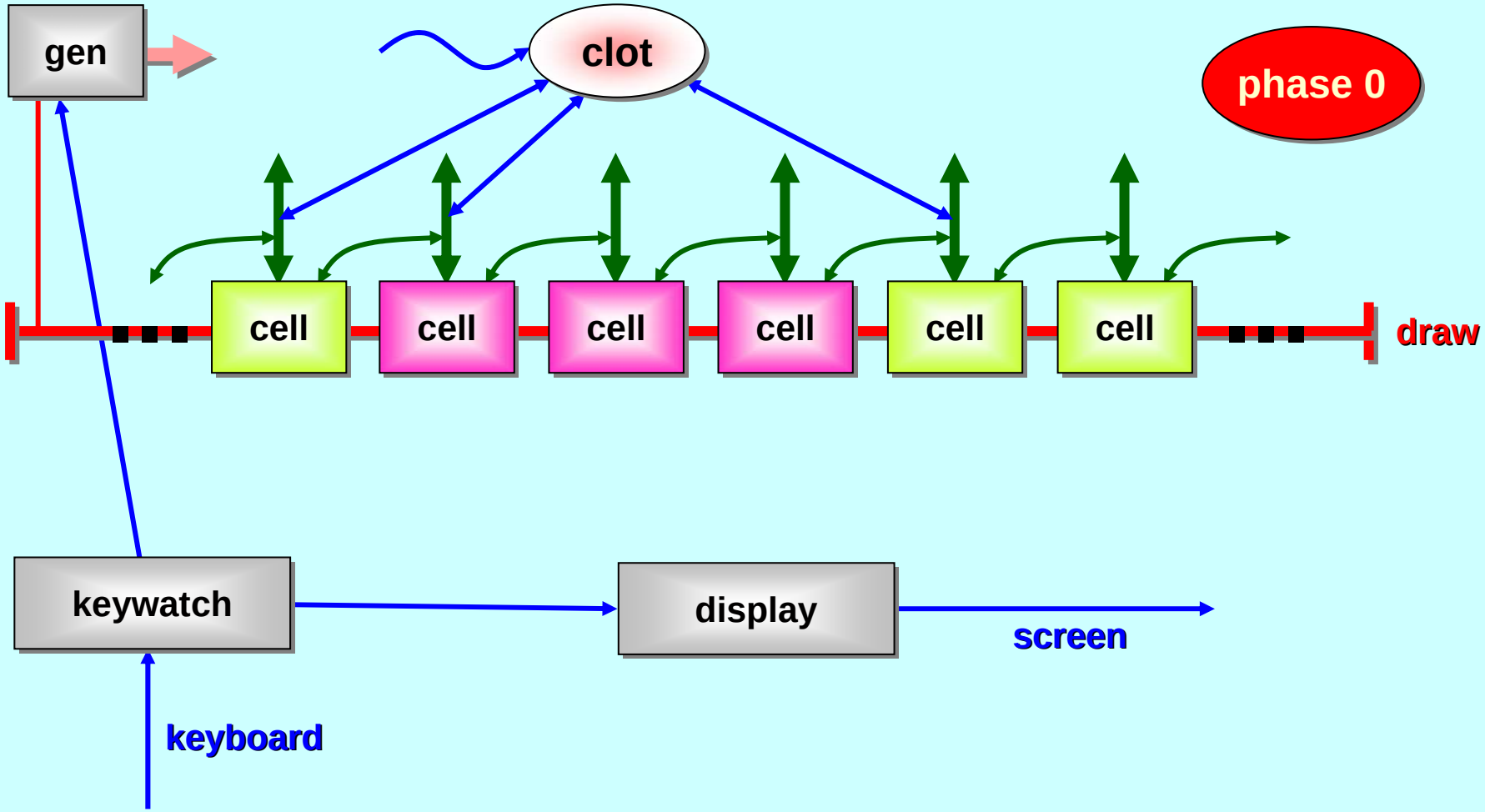
Platelet Model ('lazy' CA)



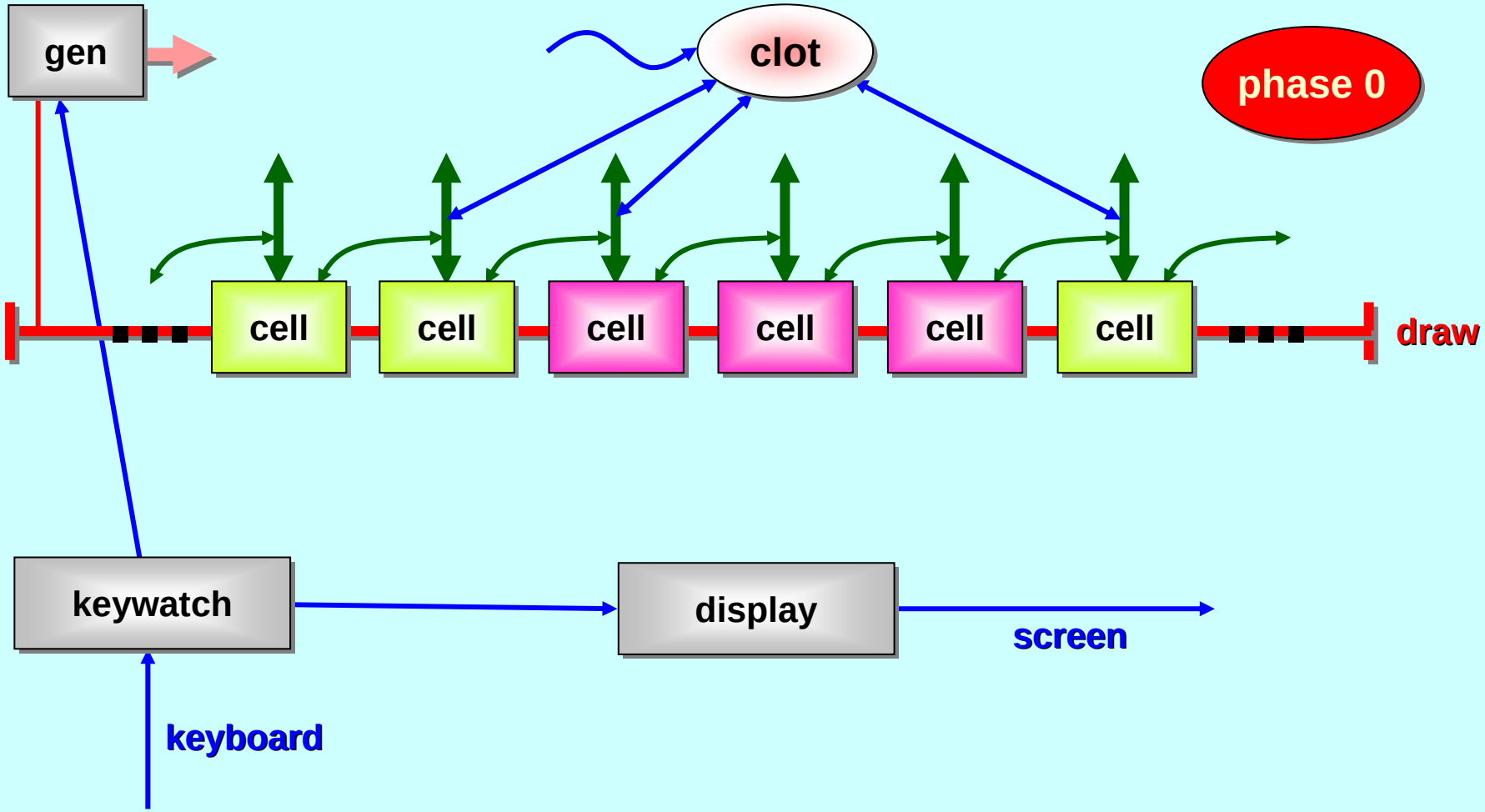
Platelet Model ('lazy' CA)



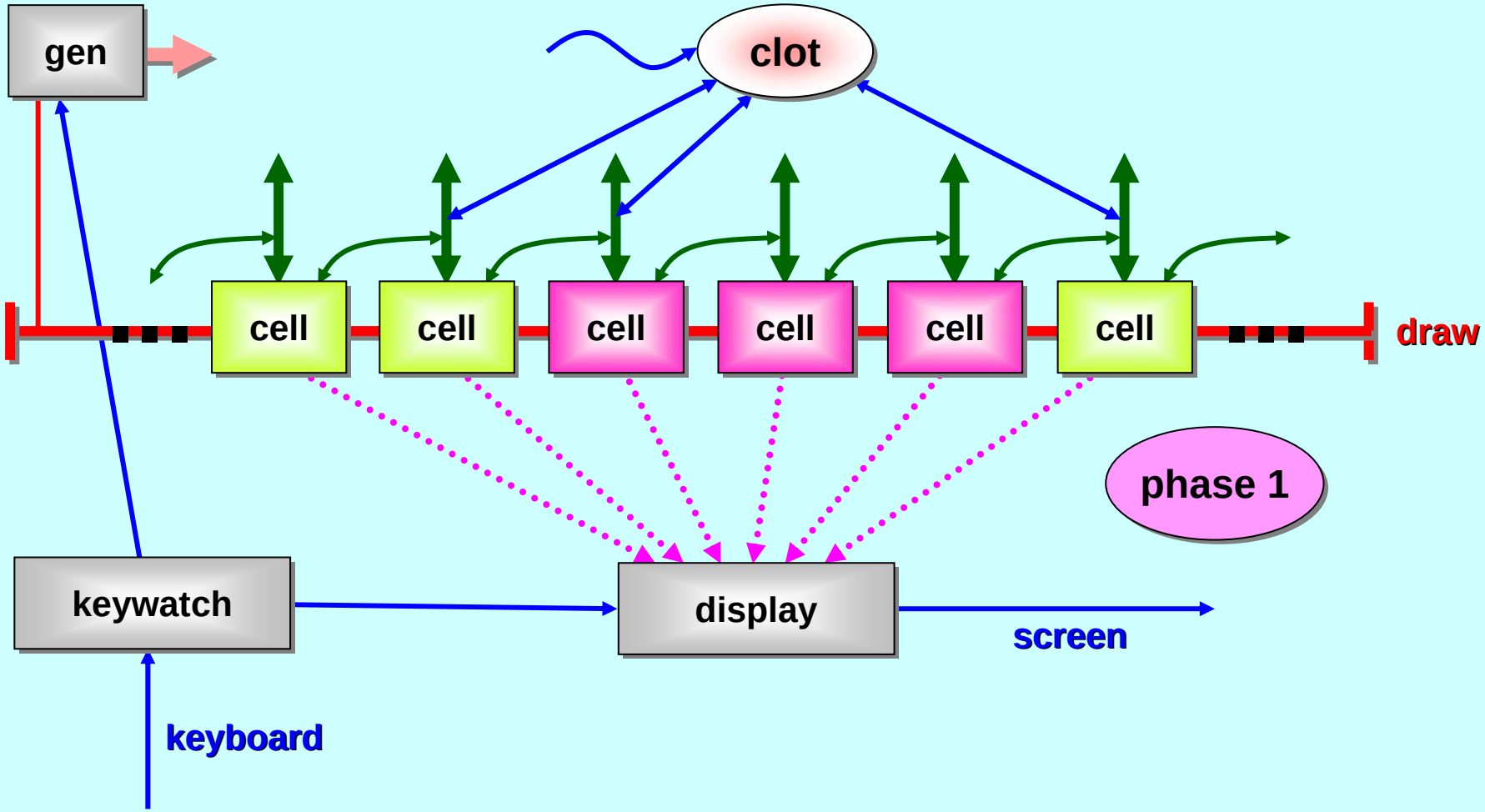
Platelet Model ('lazy' CA)



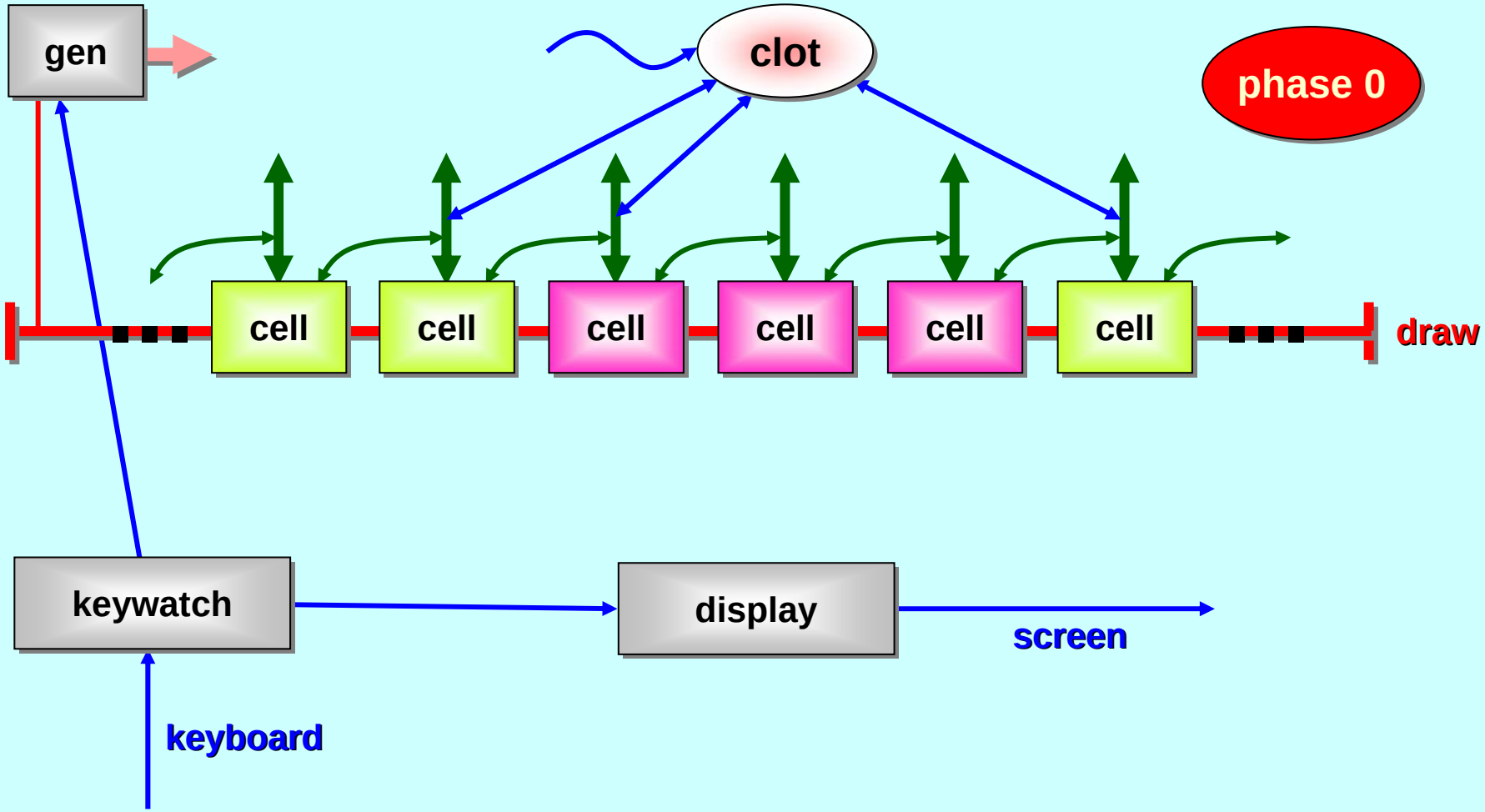
Platelet Model ('lazy' CA)



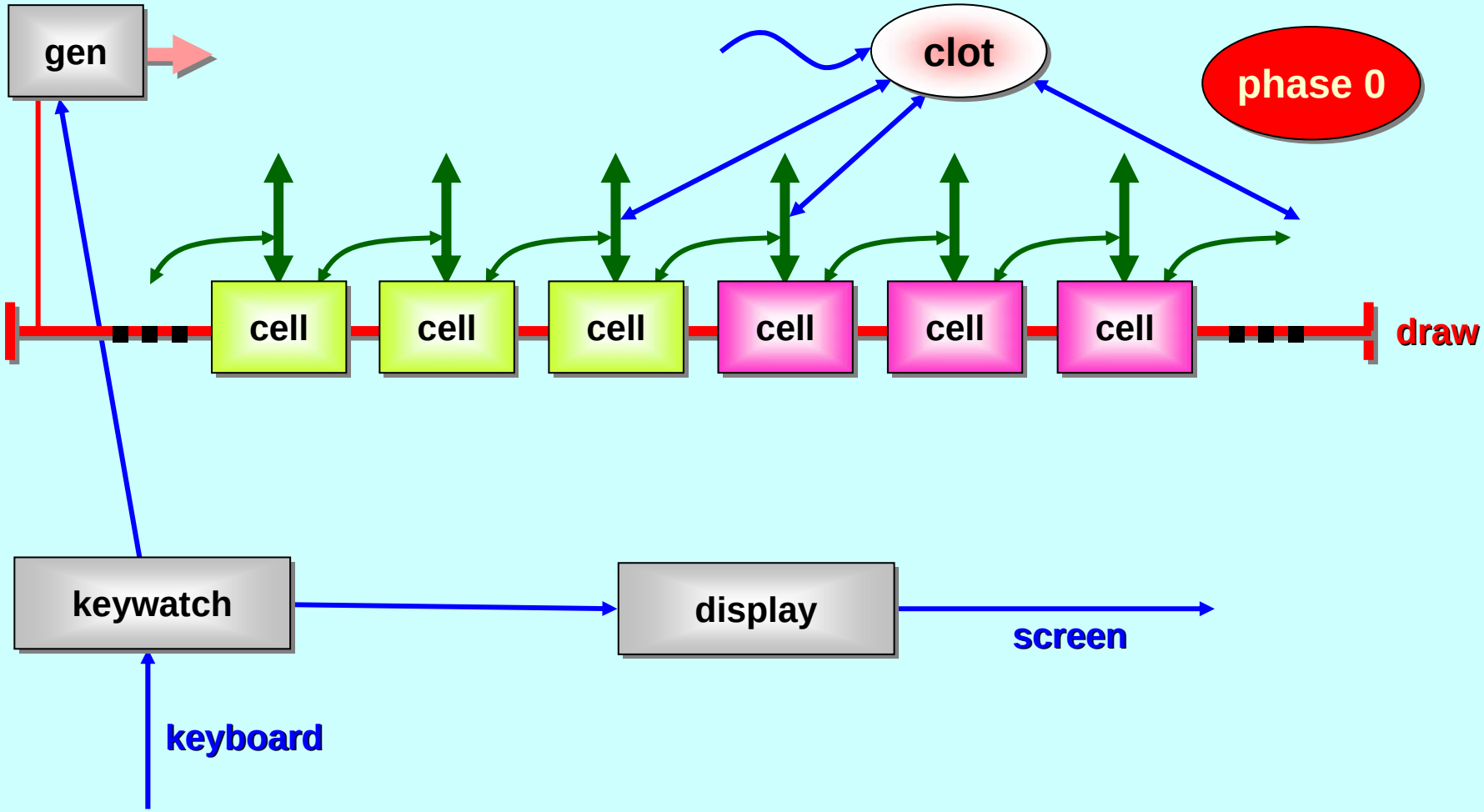
Platelet Model ('lazy' CA)



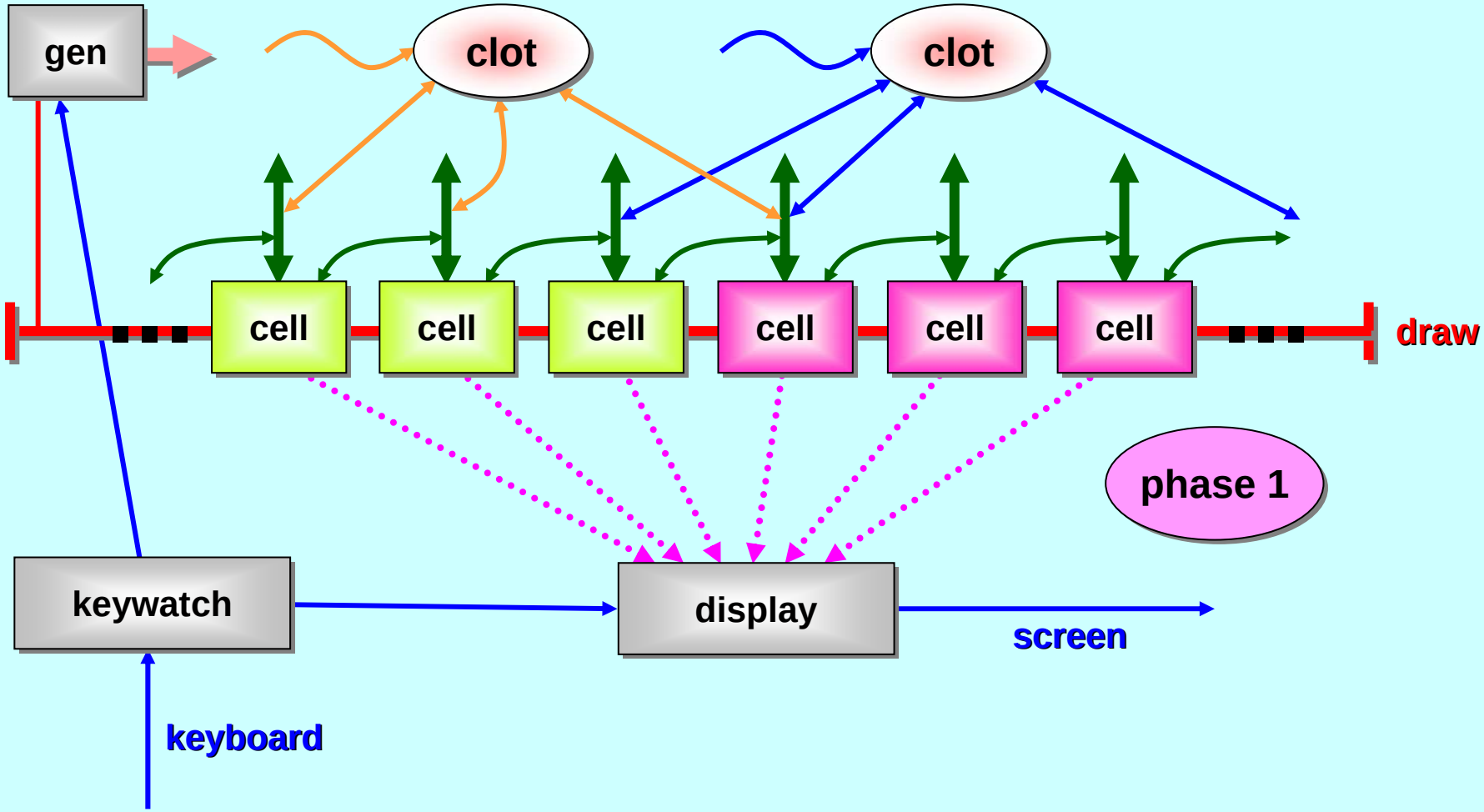
Platelet Model ('lazy' CA)



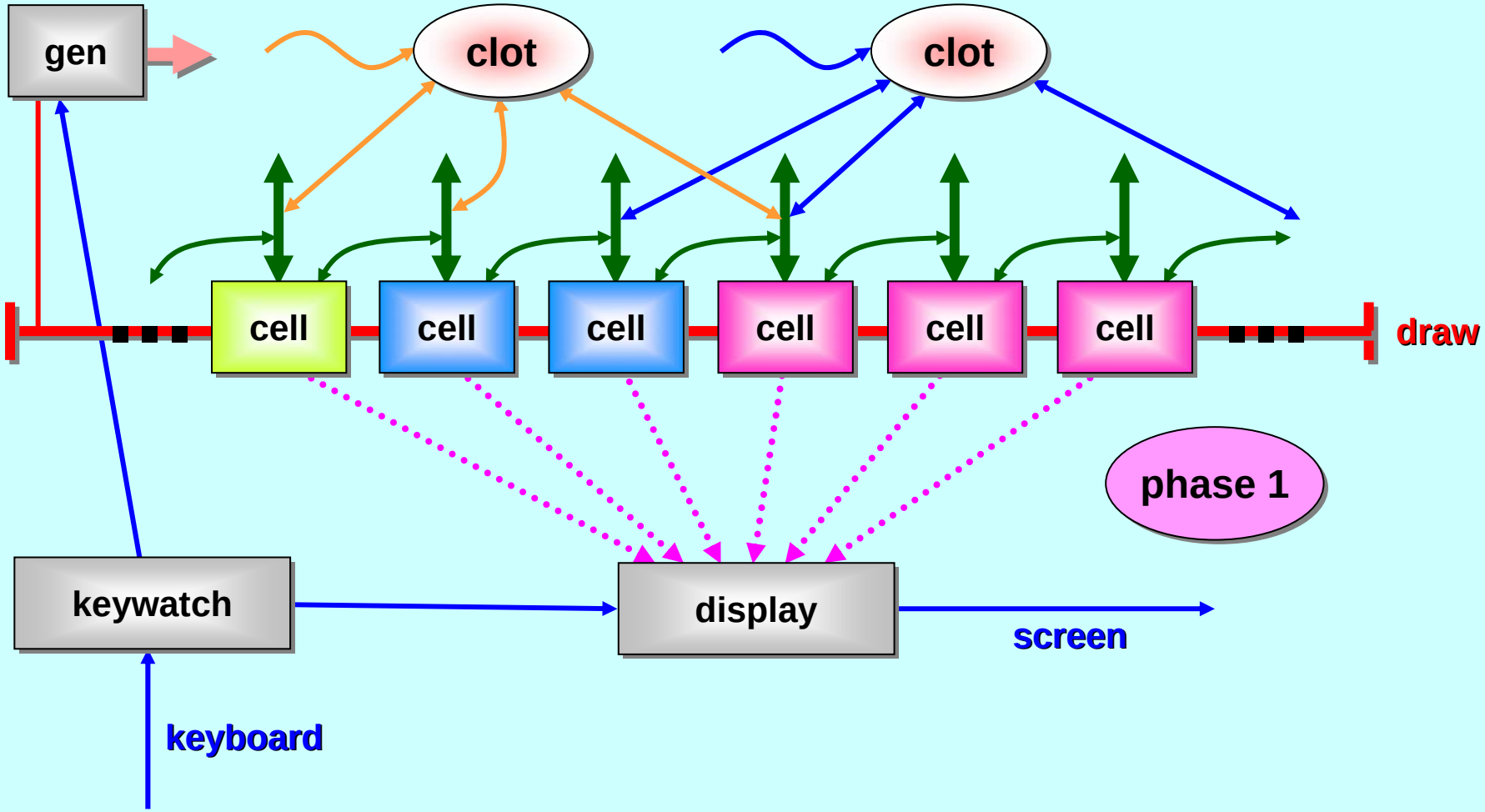
Platelet Model ('lazy' CA)



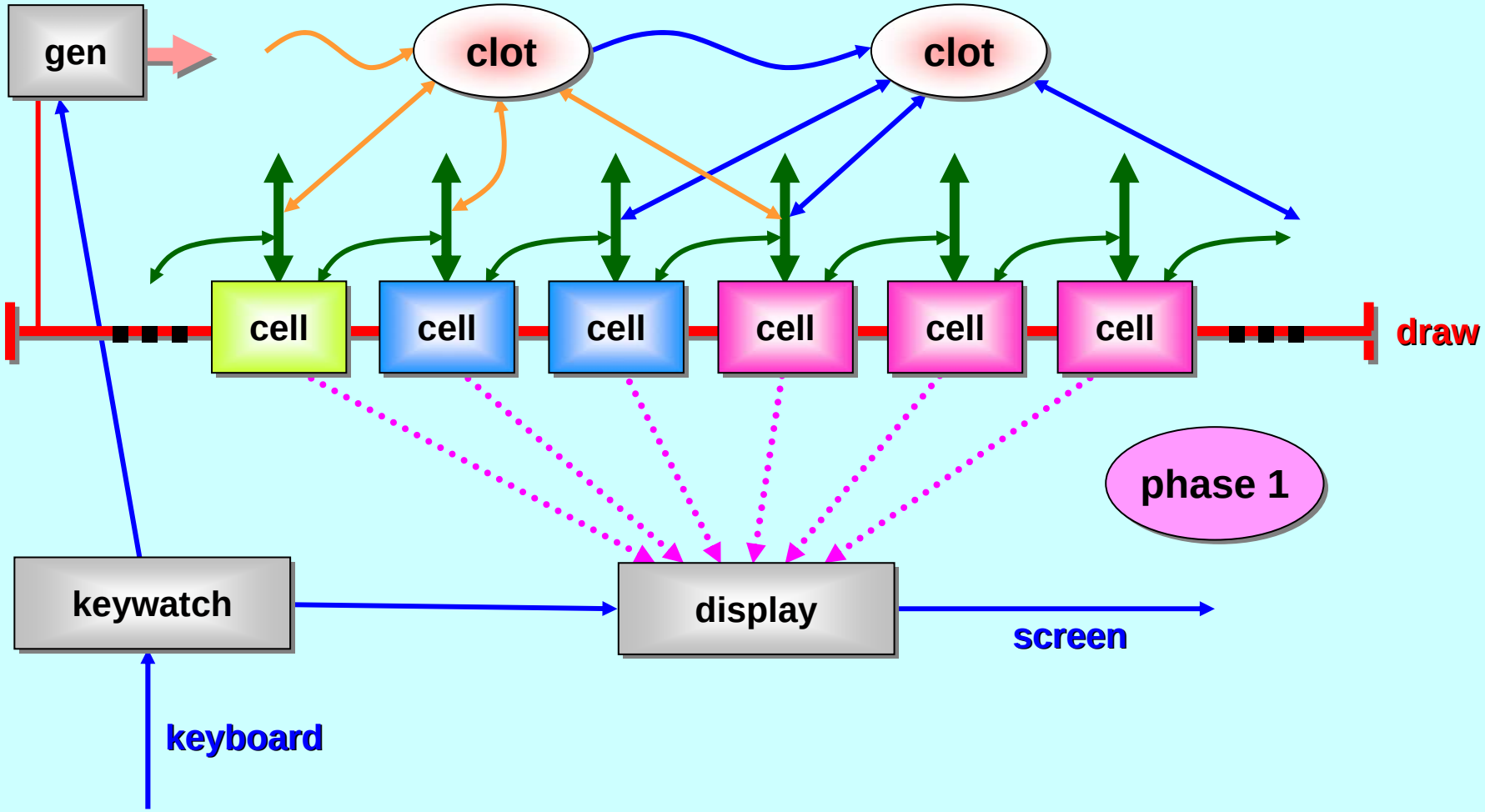
Platelet Model ('lazy' CA)



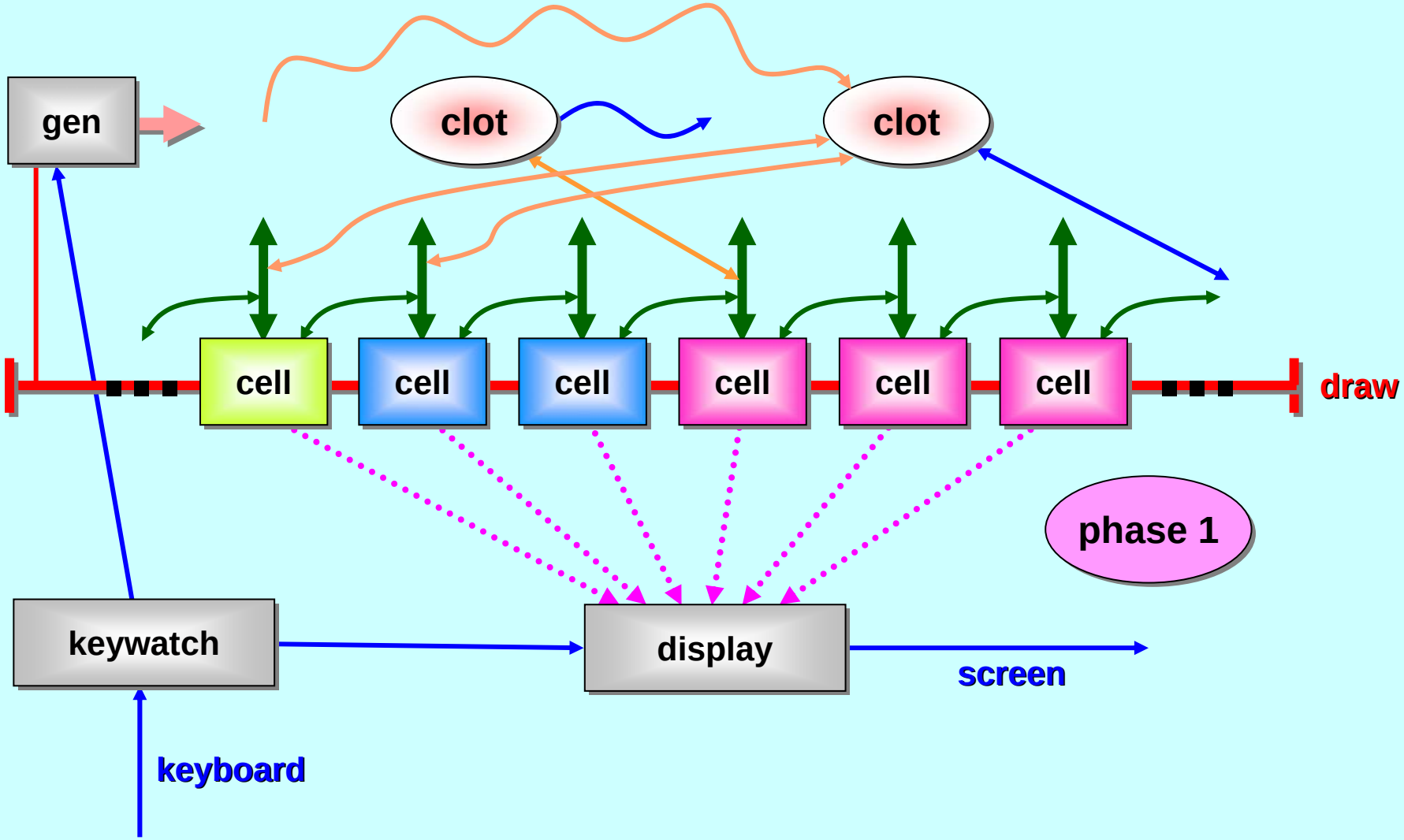
Platelet Model ('lazy' CA)



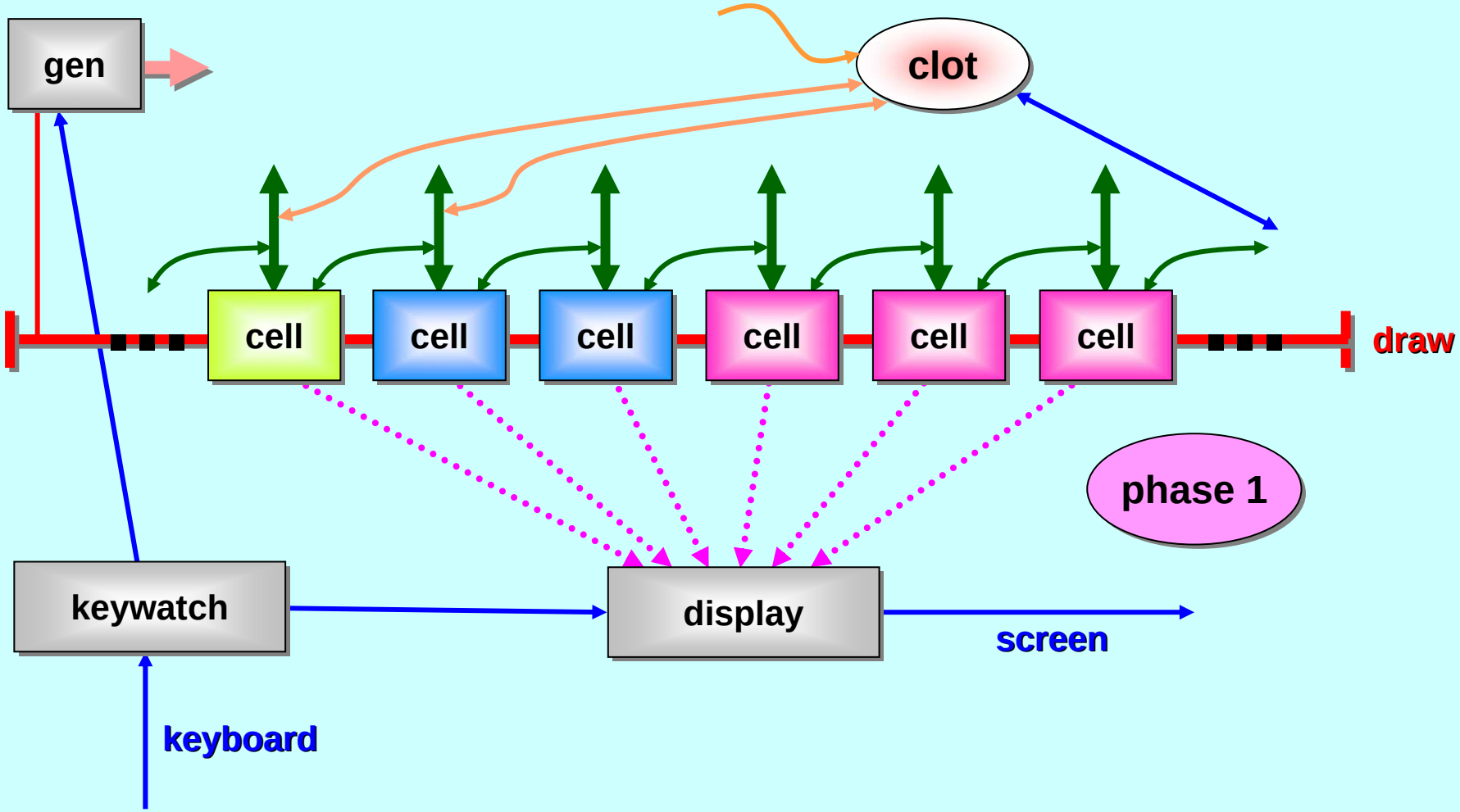
Platelet Model ('lazy' CA)



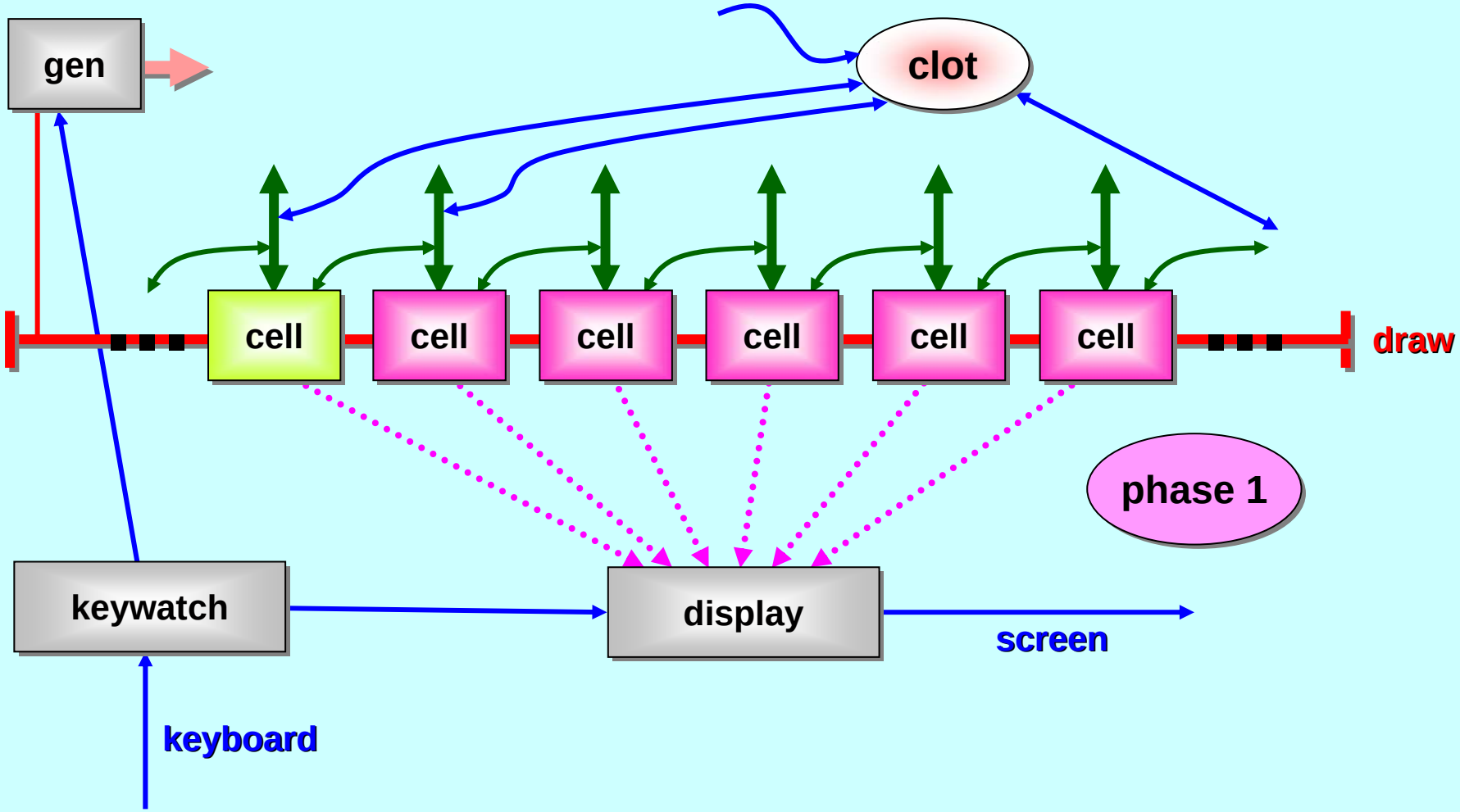
Platelet Model ('lazy' CA)



Platelet Model ('lazy' CA)



Platelet Model ('lazy' CA)

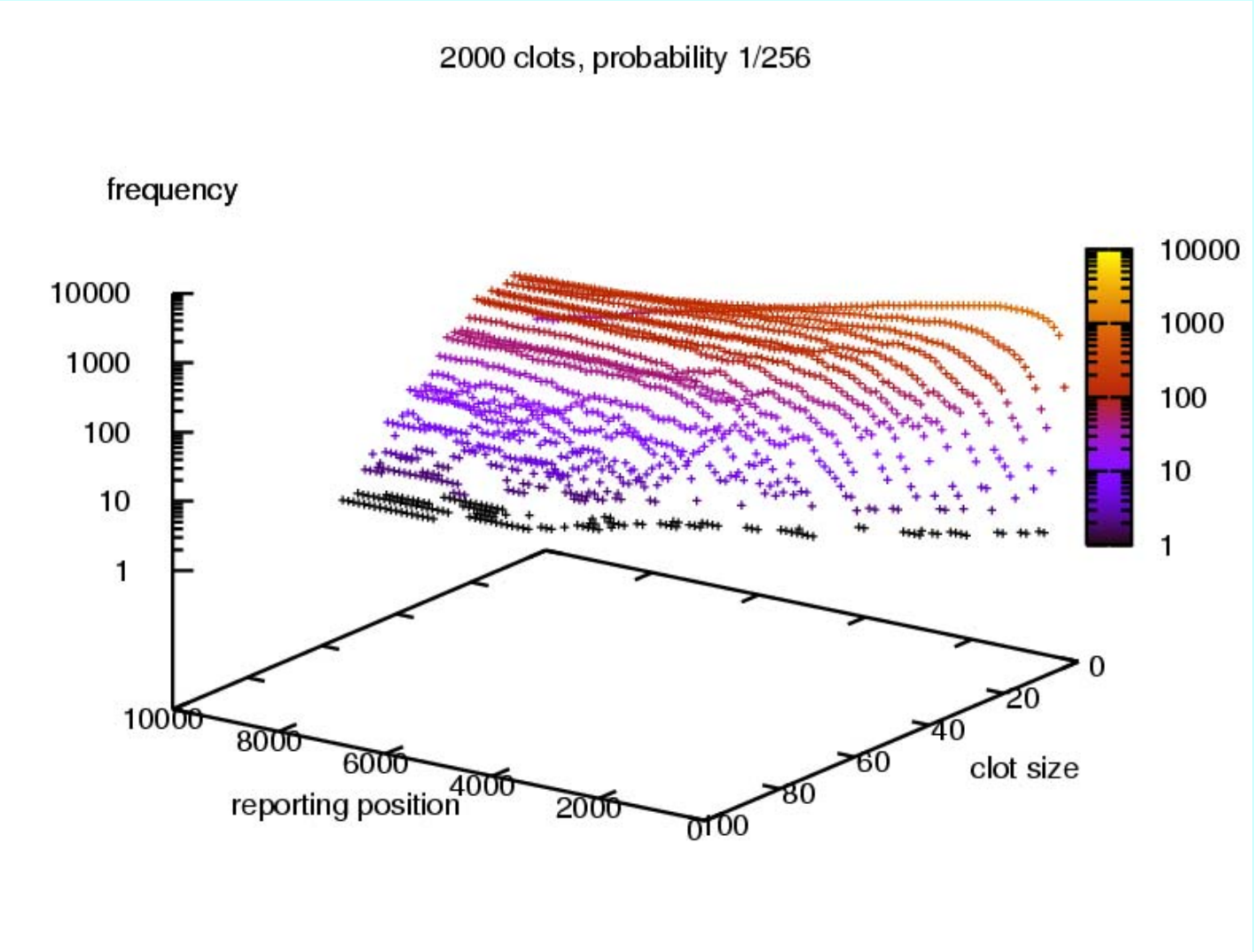


Platelet Model ('lazy' CA)

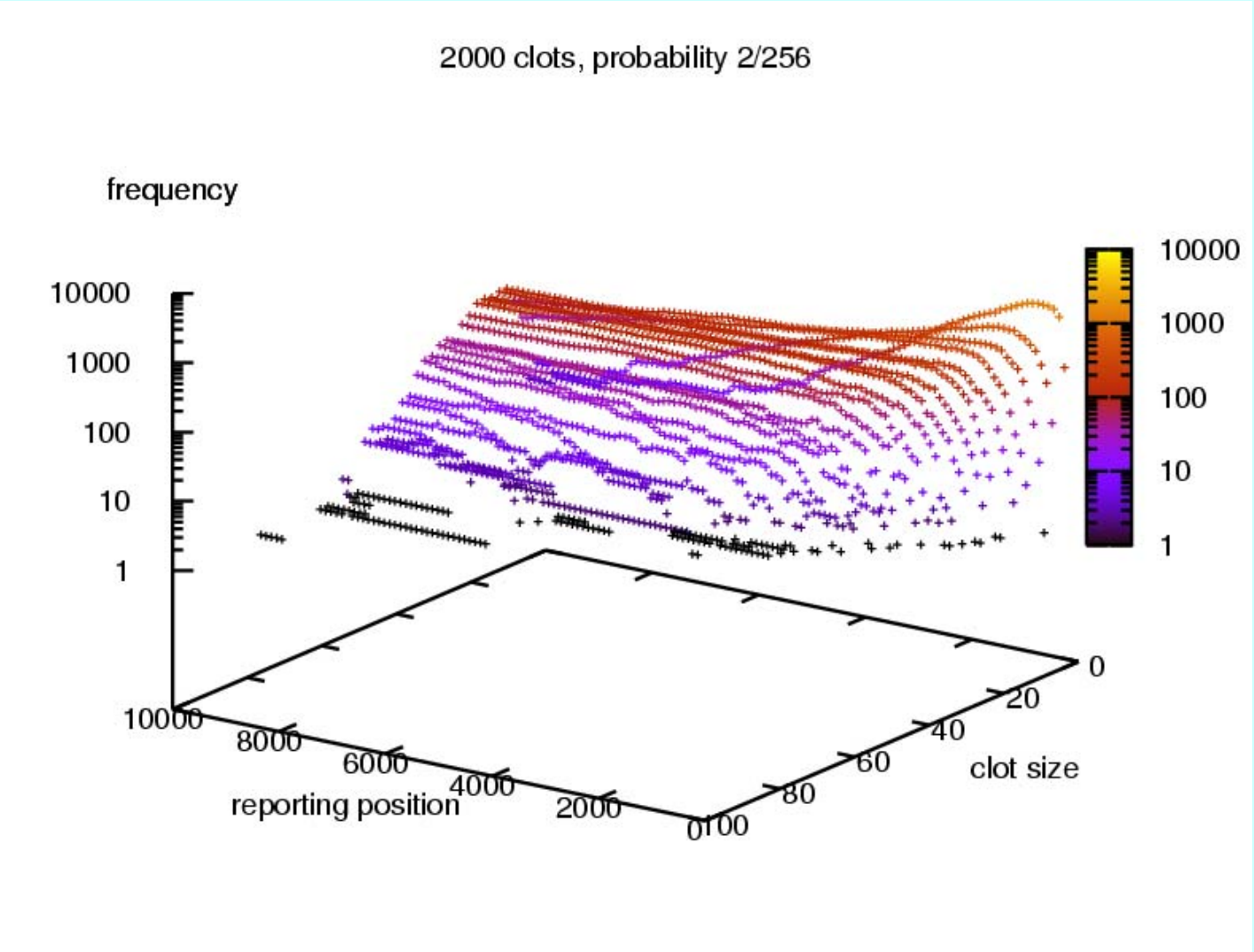
Performance: a ***cell*** only works when a ***clot*** boundary moves through. Run-time depends only on the number of ***clots***; the ***clot*** sizes are now irrelevant (2.4 GHz. P IV-M).

Generate probability (n / 256)	'Busy' (ns)	'Lazy' (ns)
0	650	0
1	660	8
2	670	12
4	680	14
8	700	16
16	740	18
32	1070 (total jam)	0 (total jam)

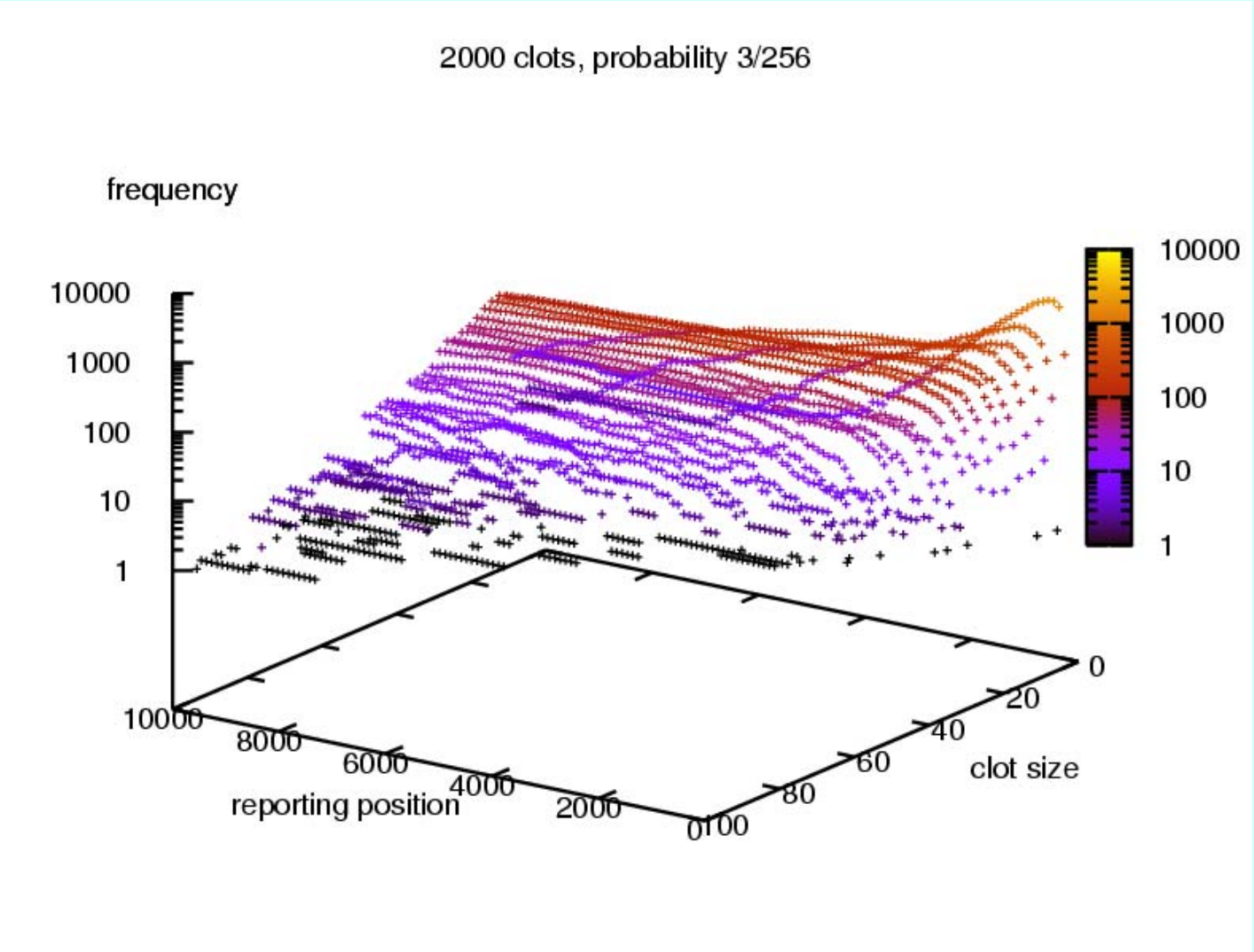
Clot Frequency by Position by Size



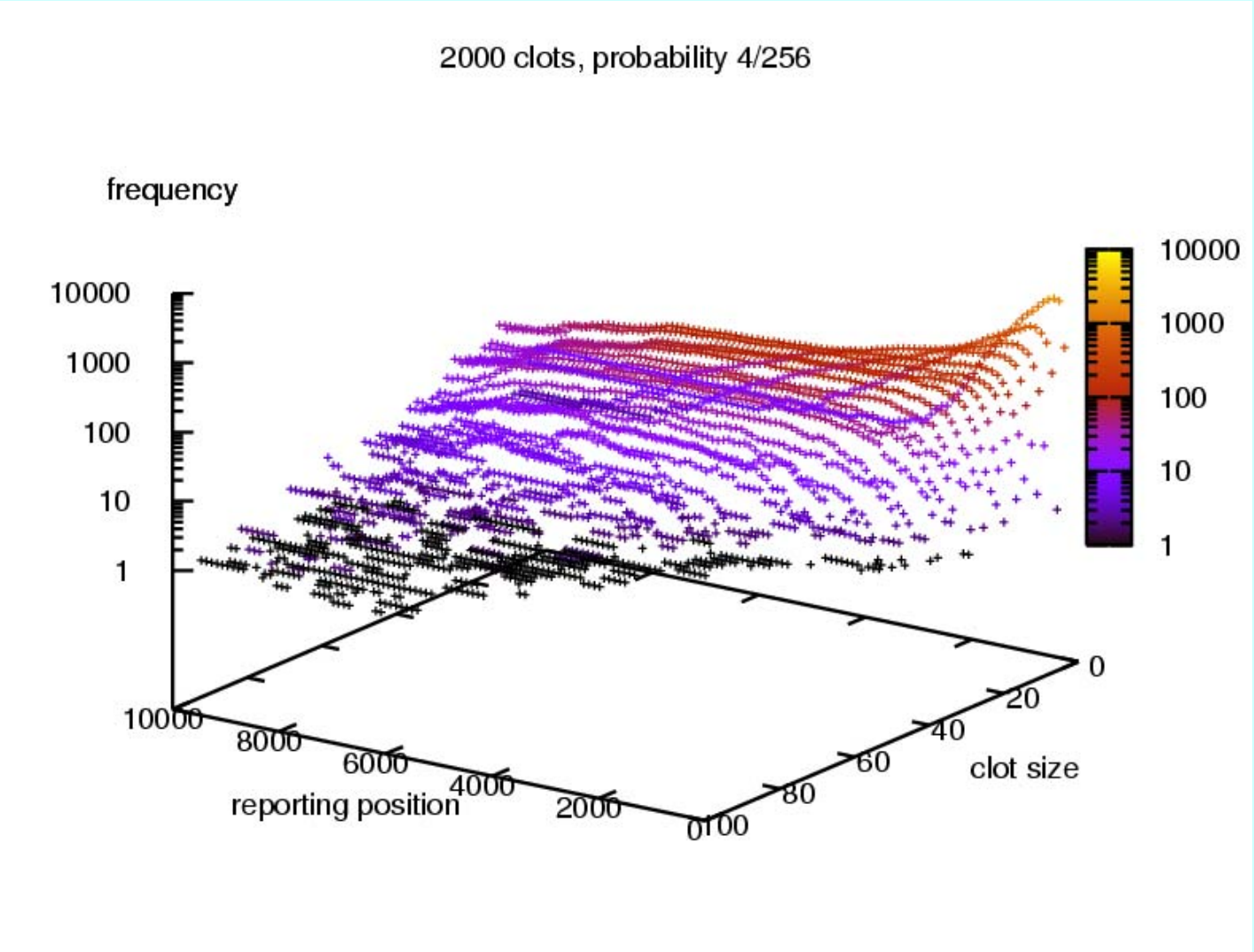
Clot Frequency by Position by Size



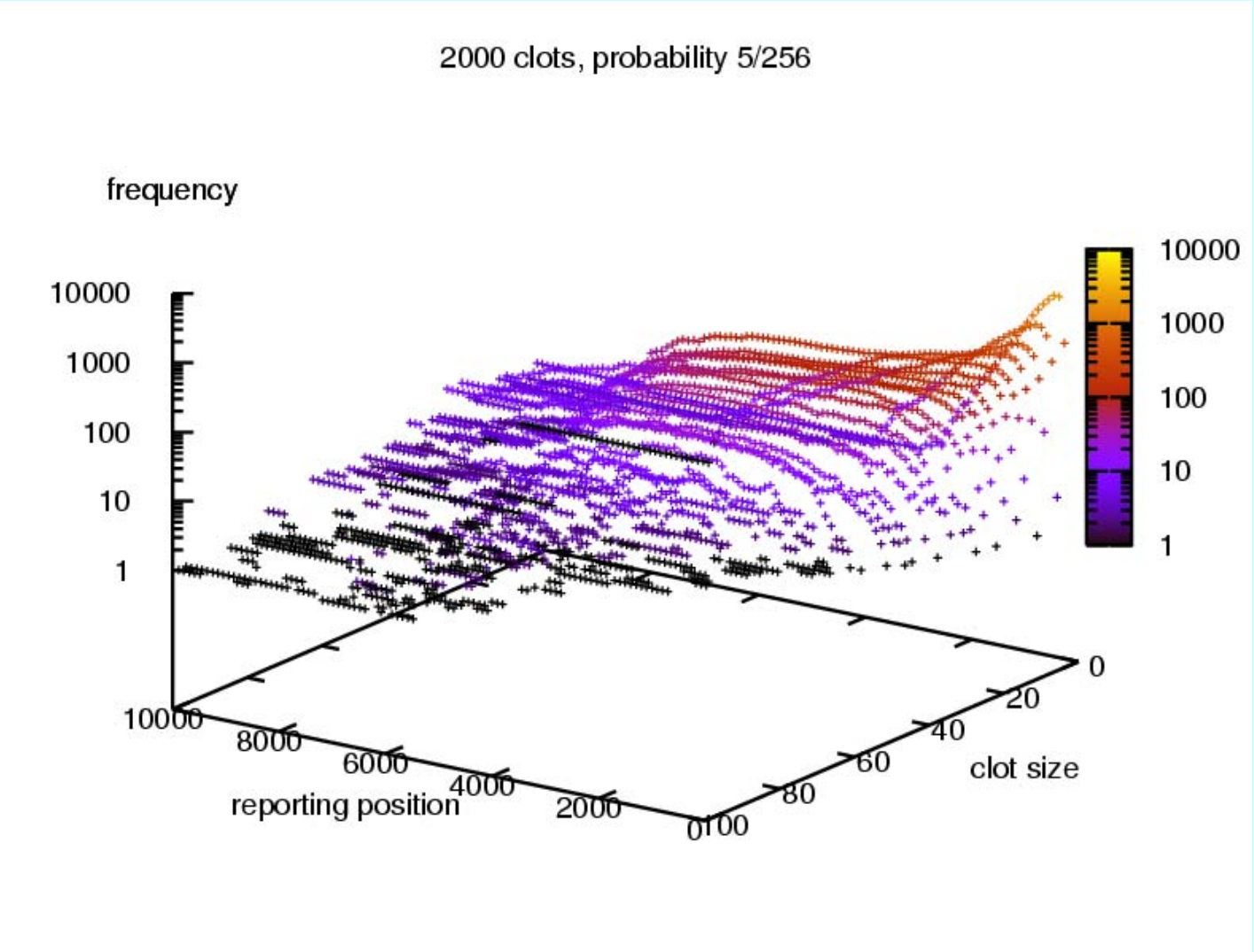
Clot Frequency by Position by Size



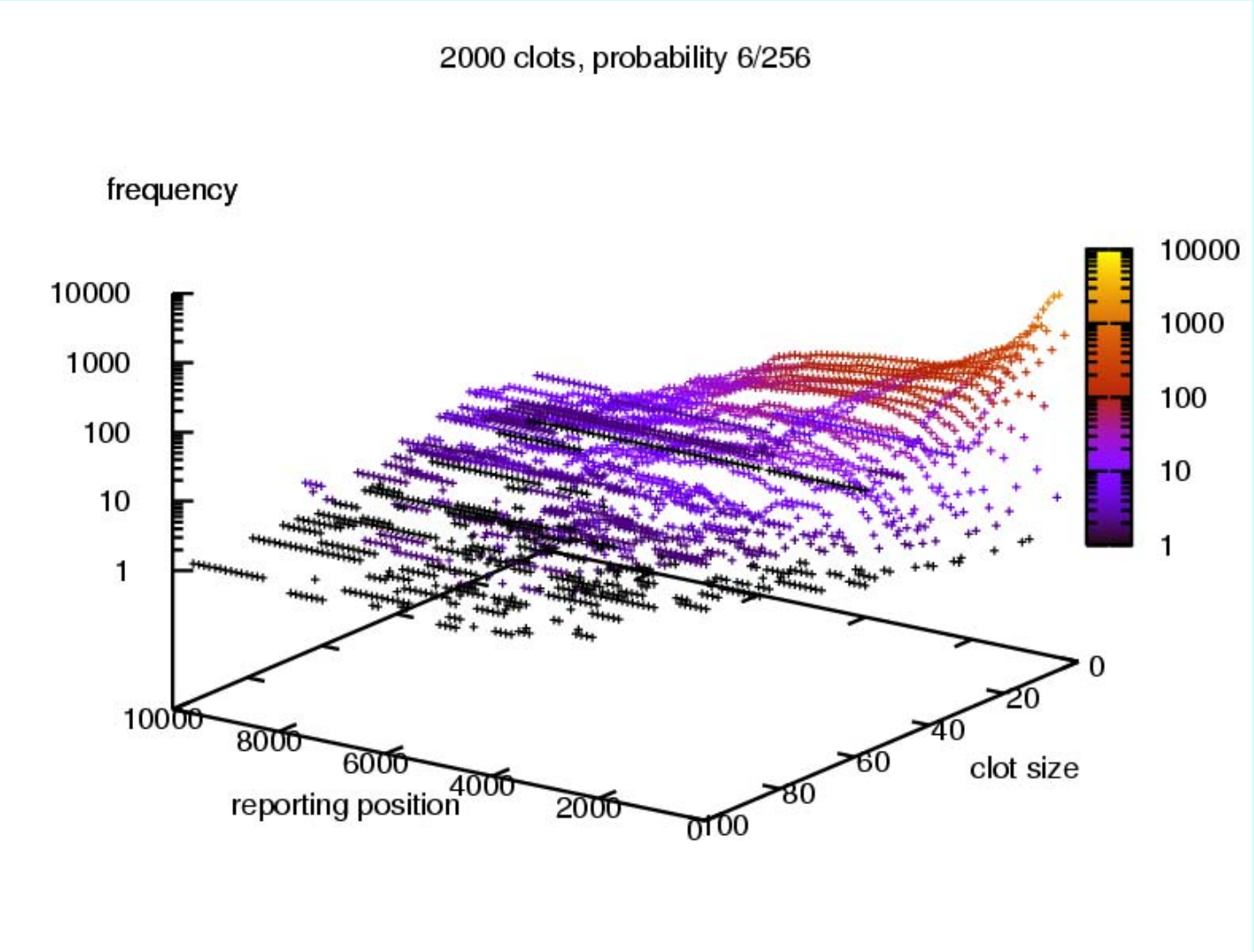
Clot Frequency by Position by Size



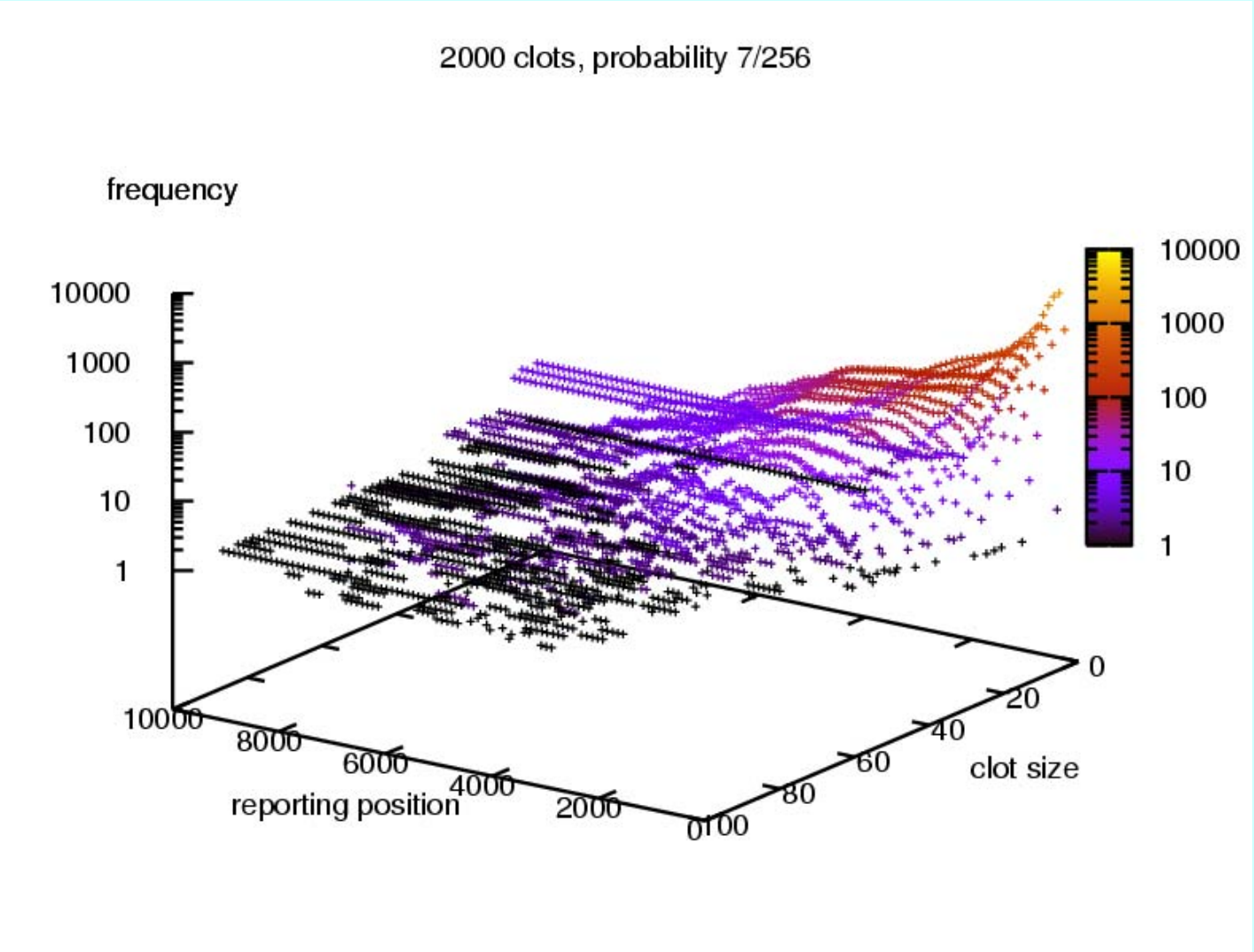
Clot Frequency by Position by Size



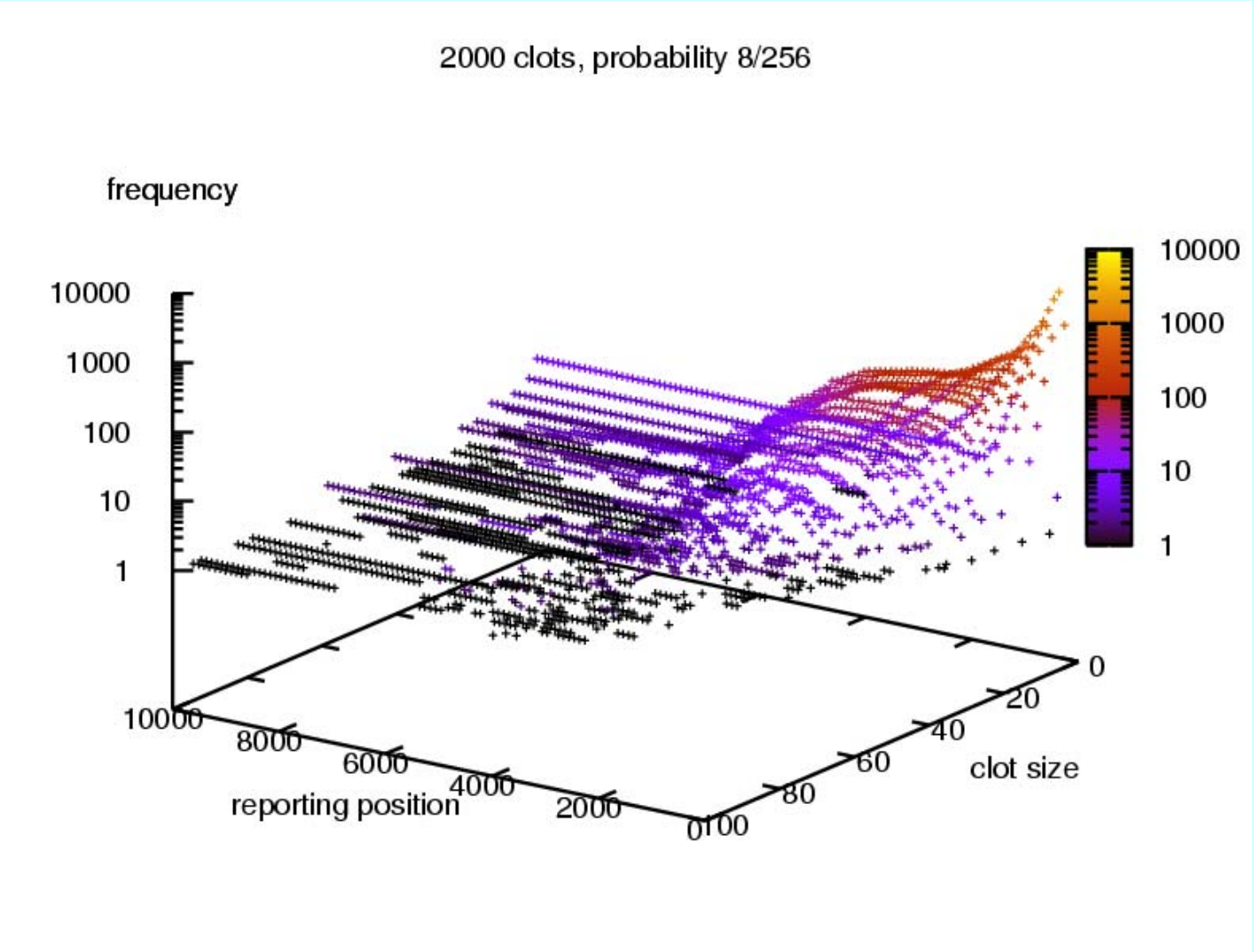
Clot Frequency by Position by Size



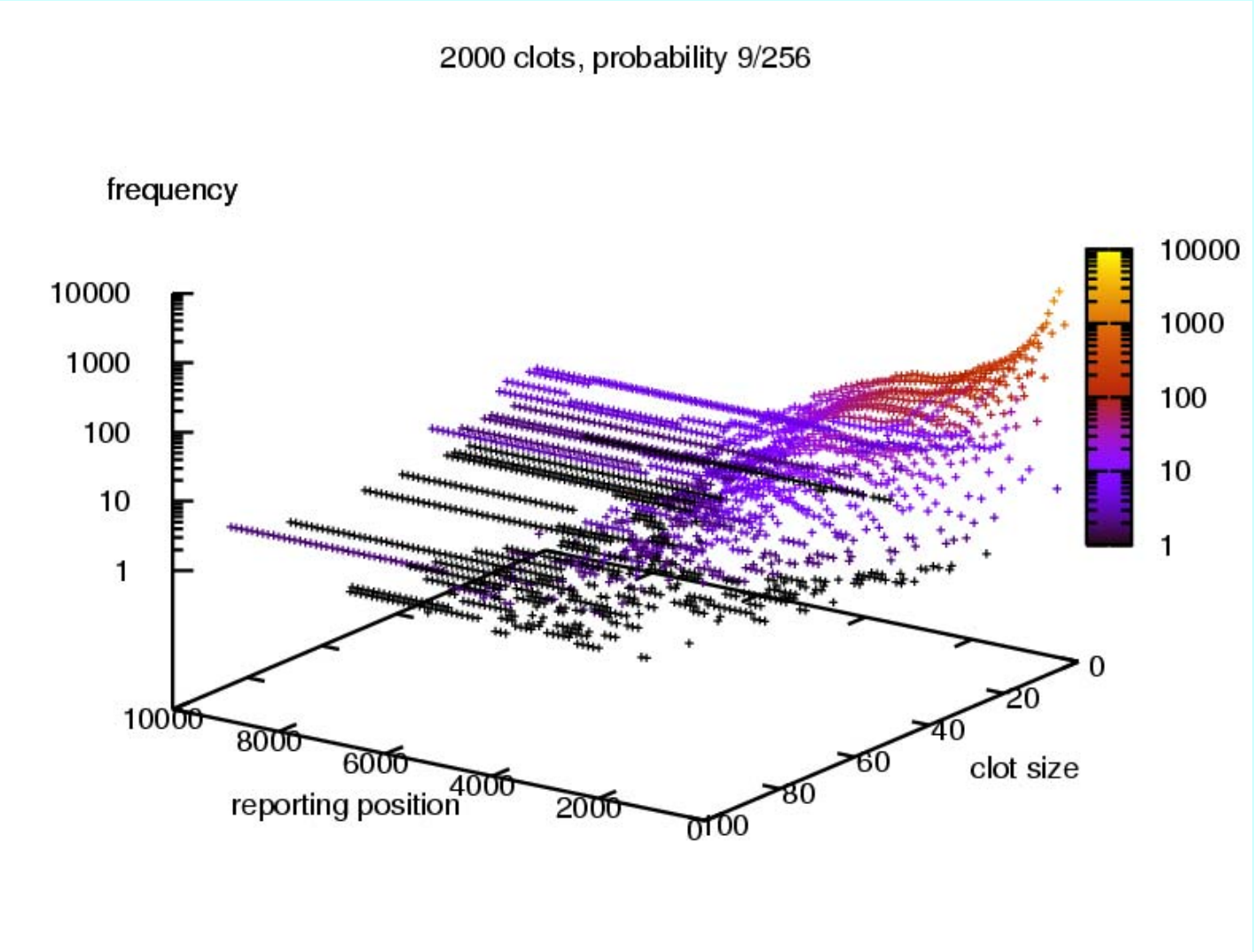
Clot Frequency by Position by Size



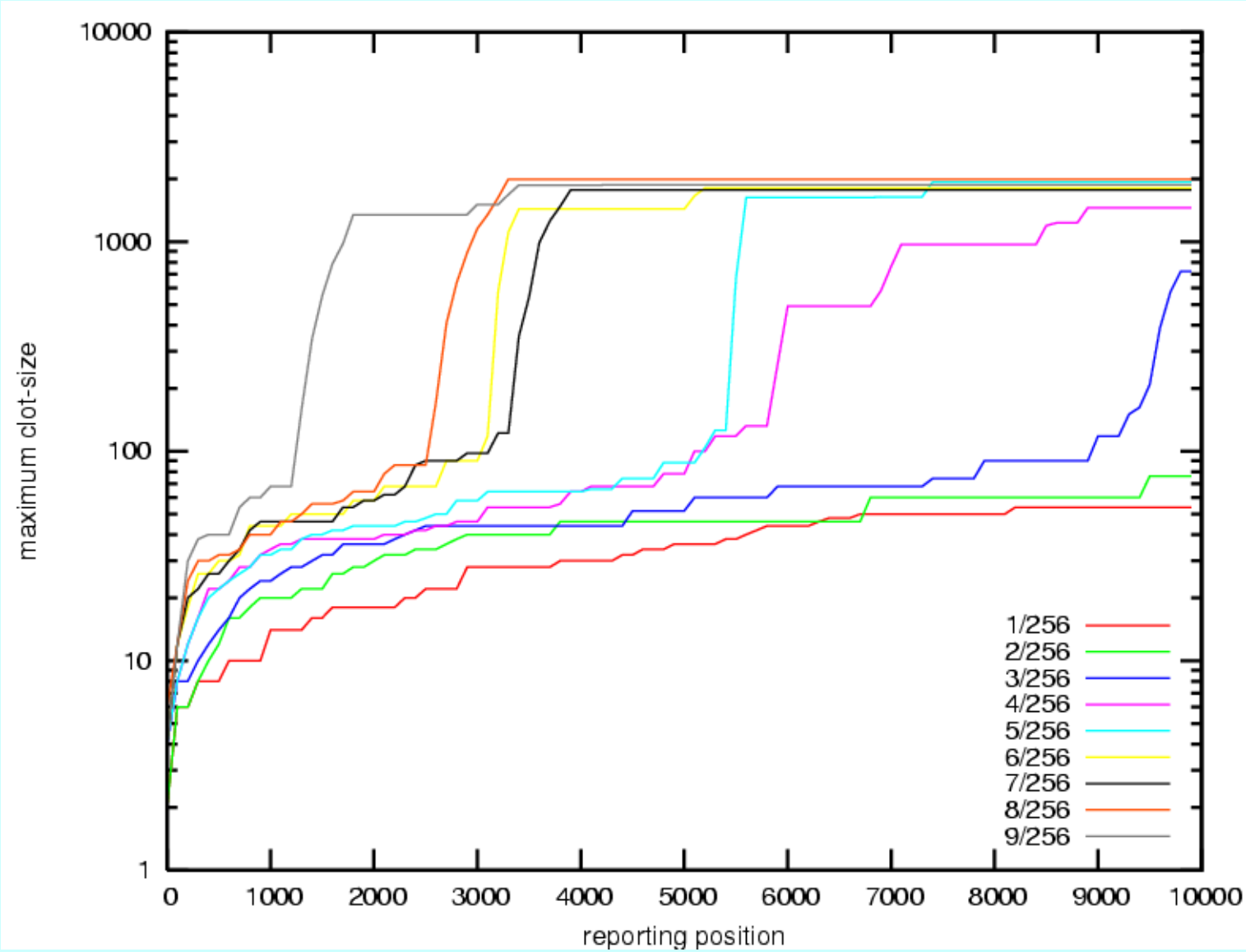
Clot Frequency by Position by Size



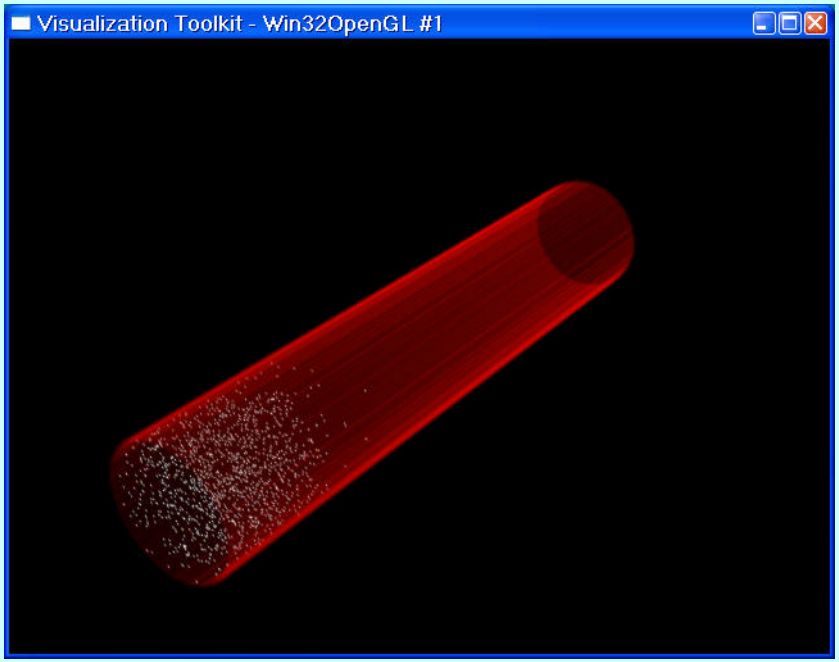
Clot Frequency by Position by Size



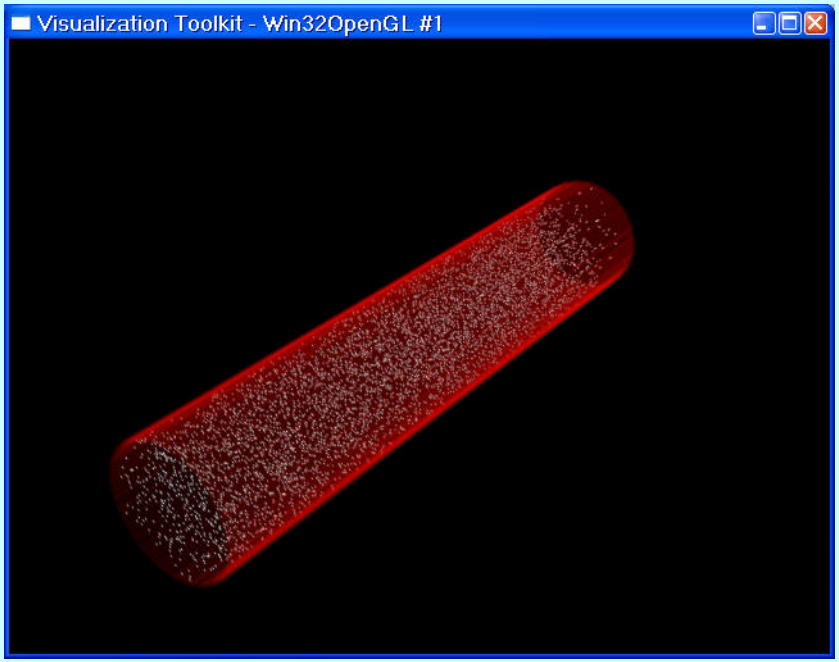
Maximum Clot Size by Position



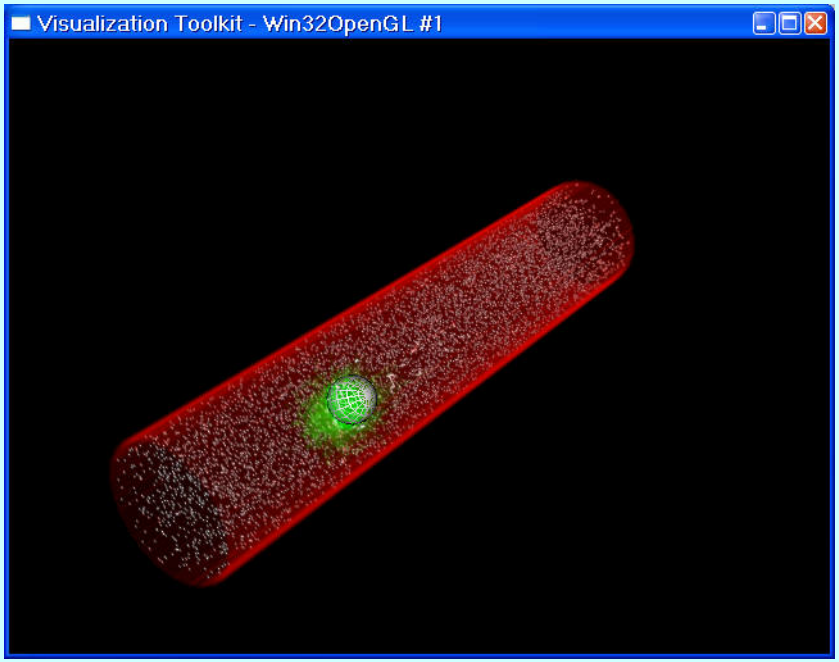
3-D Bloodstream



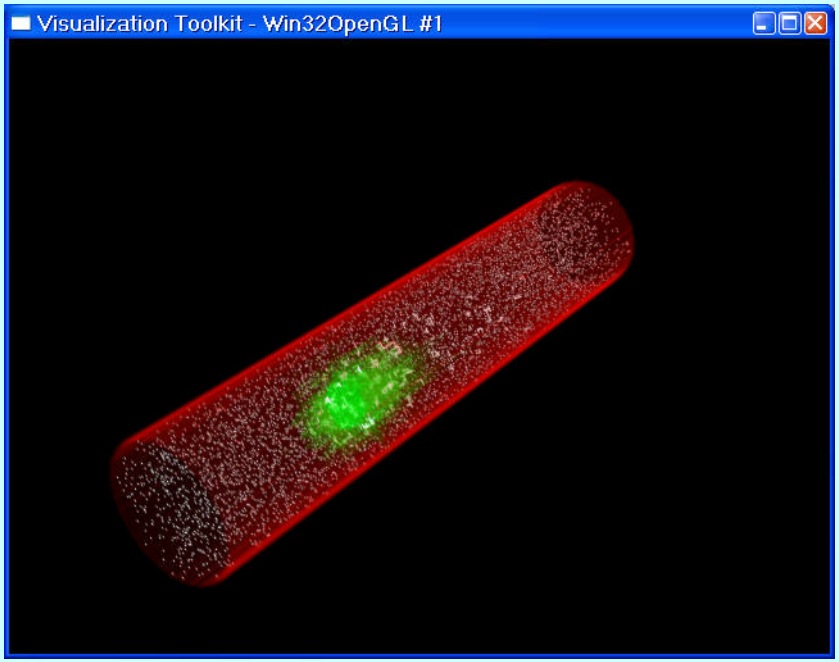
3-D Bloodstream



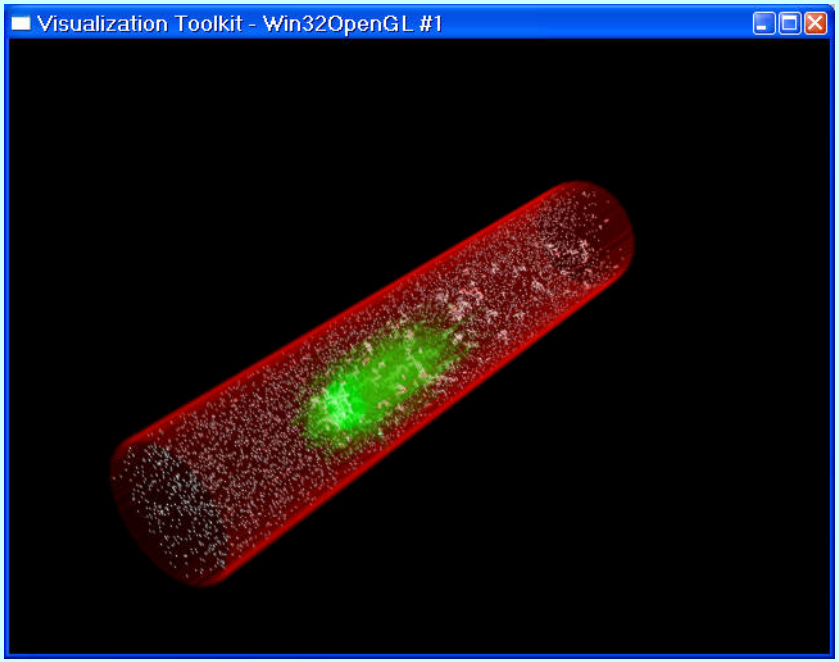
3-D Bloodstream



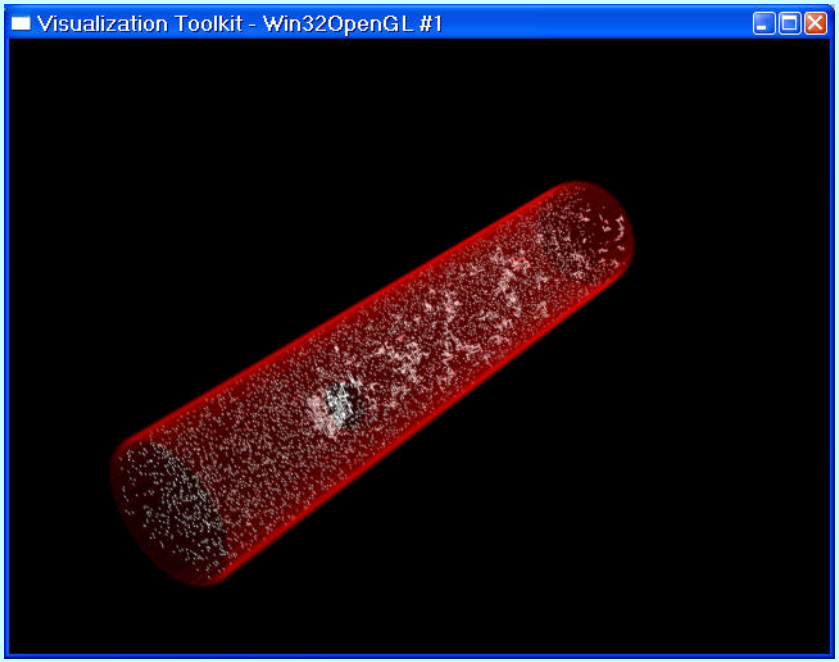
3-D Bloodstream



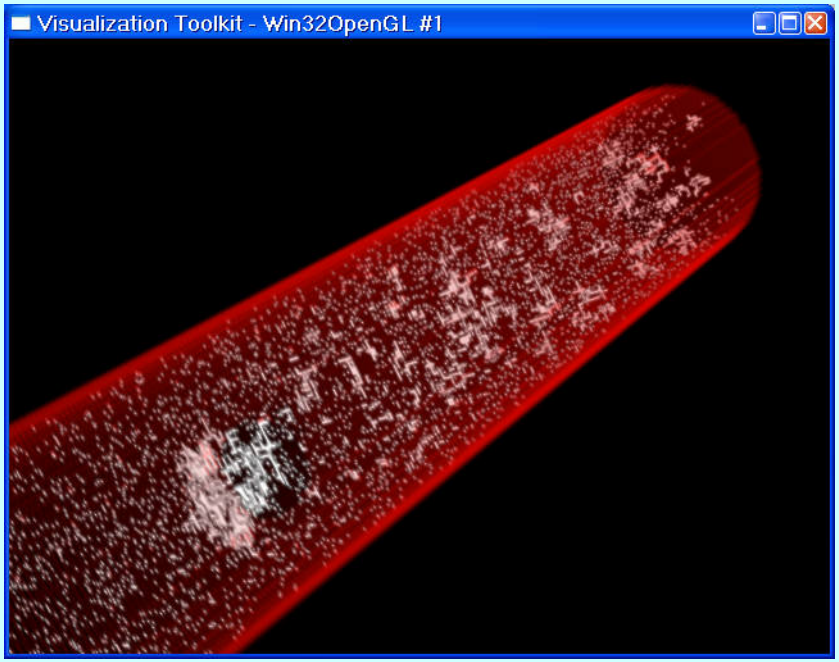
3-D Bloodstream



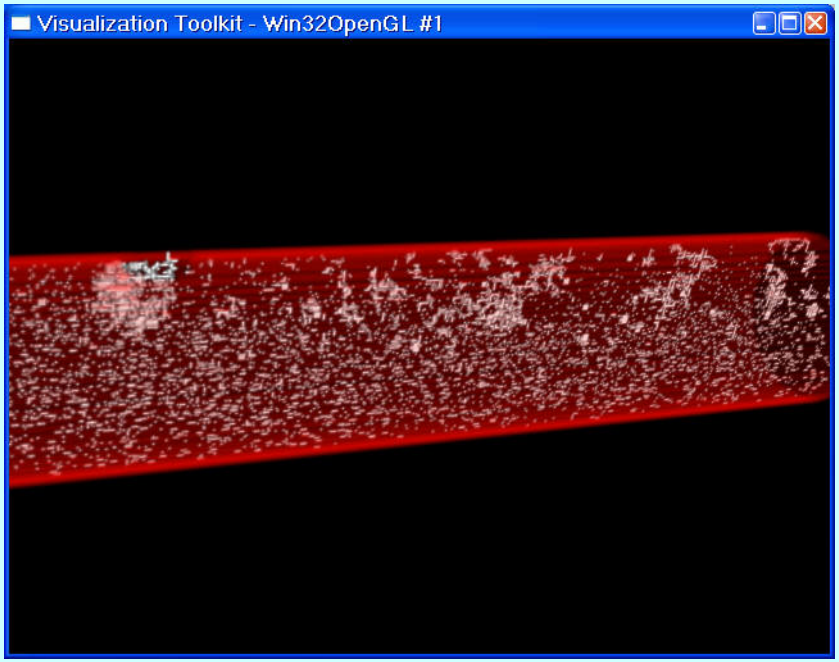
3-D Bloodstream



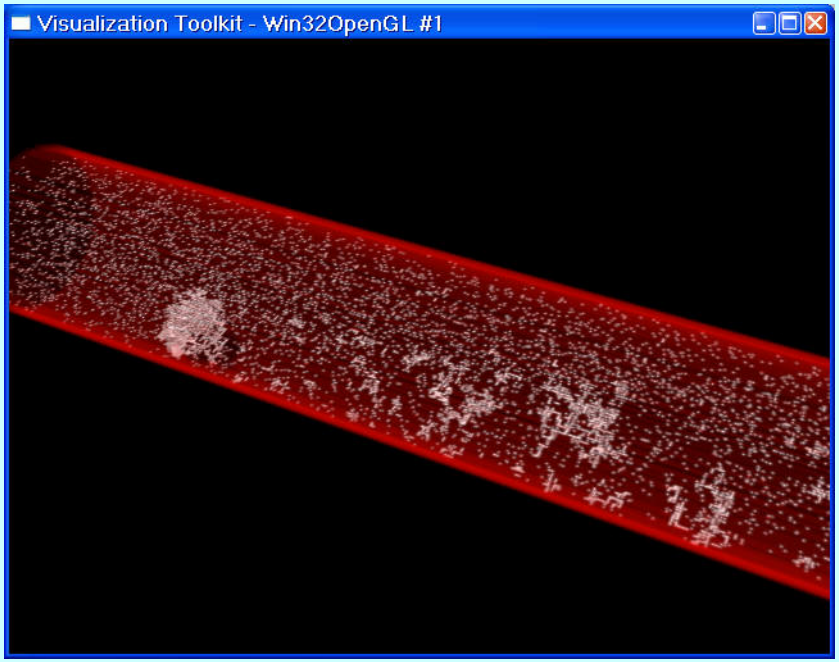
3-D Bloodstream



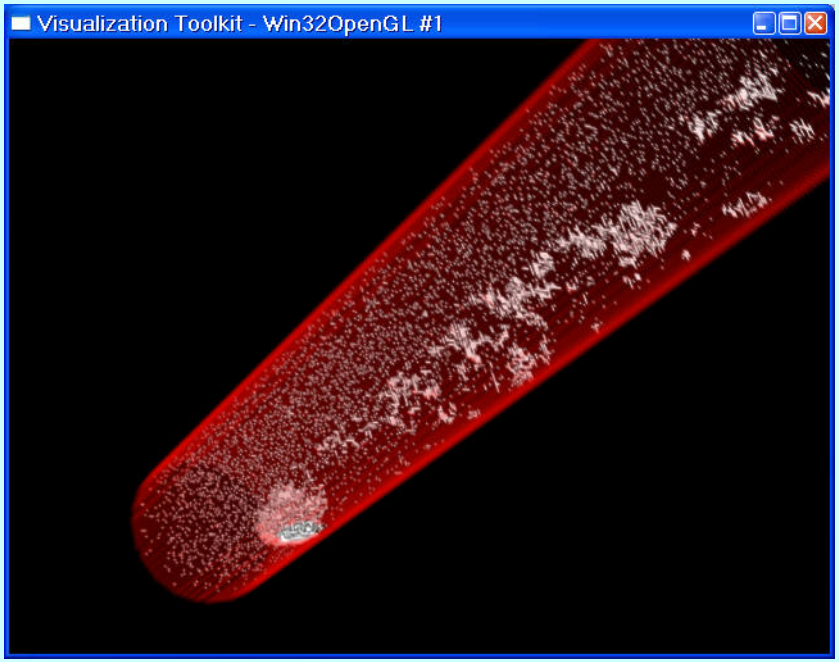
3-D Bloodstream



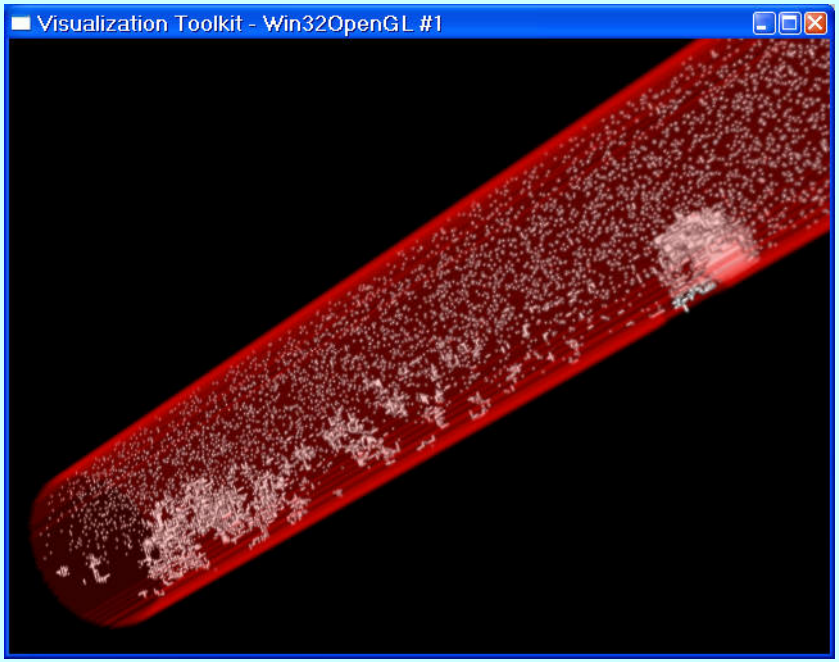
3-D Bloodstream



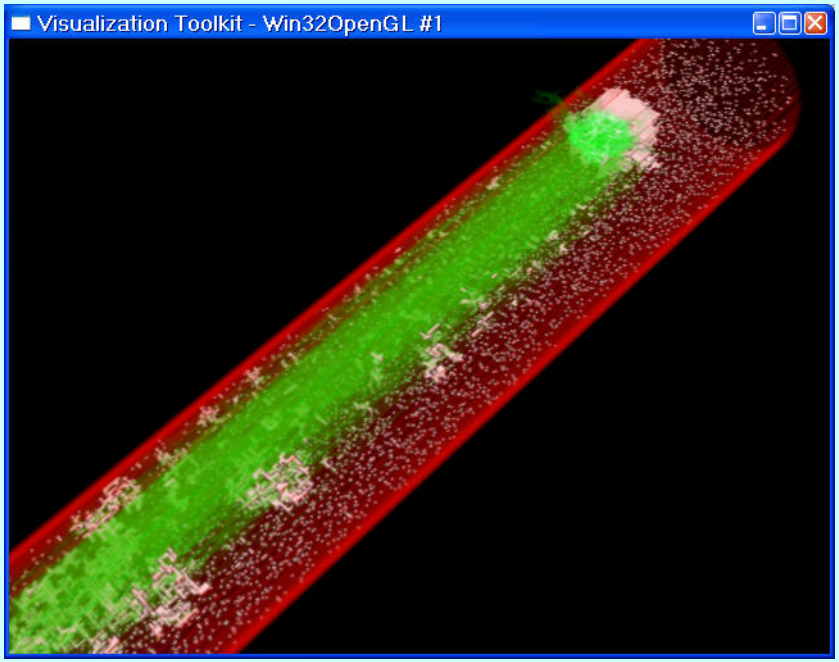
3-D Bloodstream



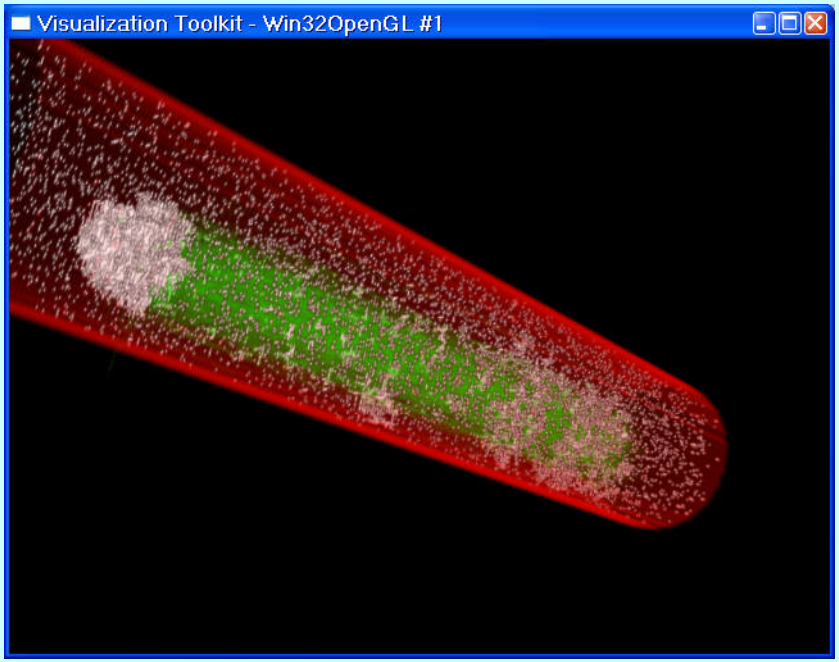
3-D Bloodstream



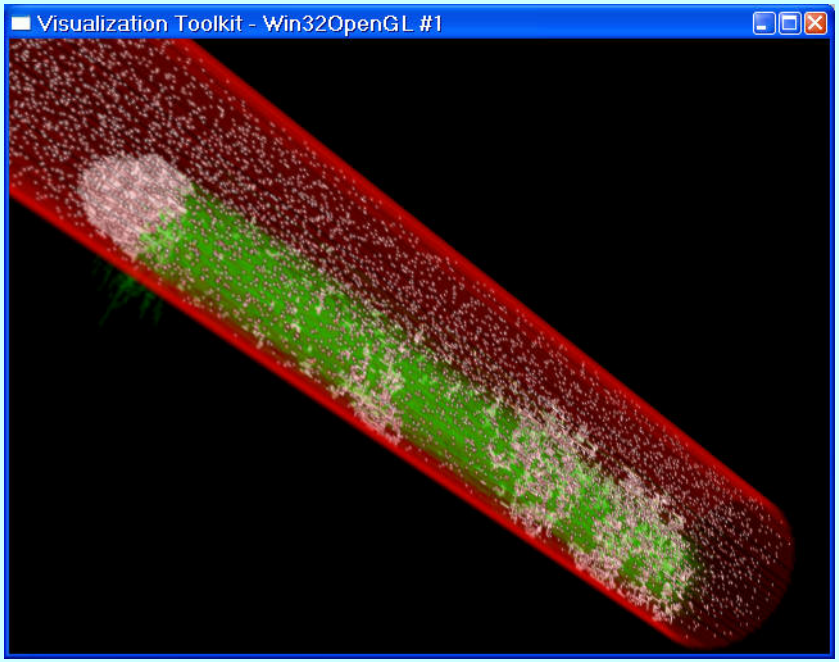
3-D Bloodstream



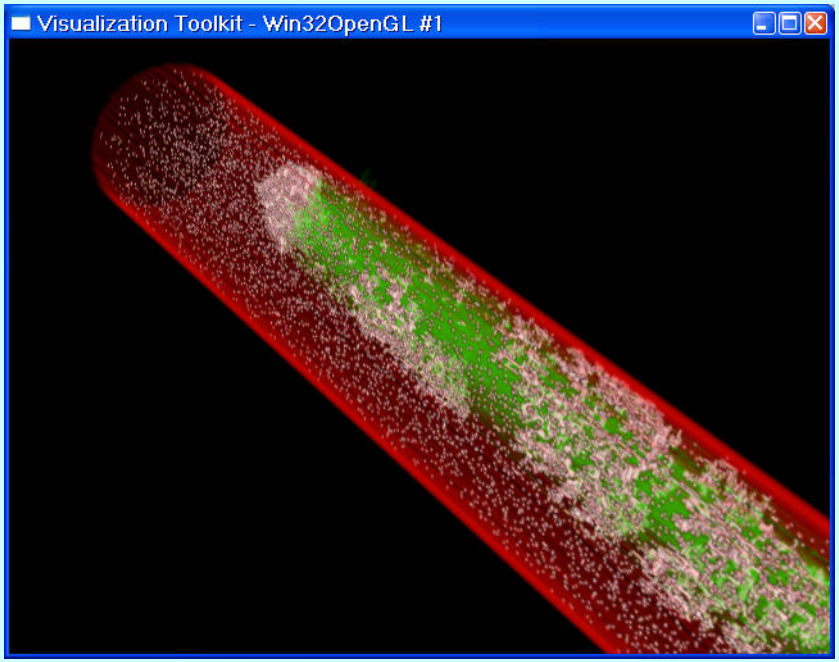
3-D Bloodstream



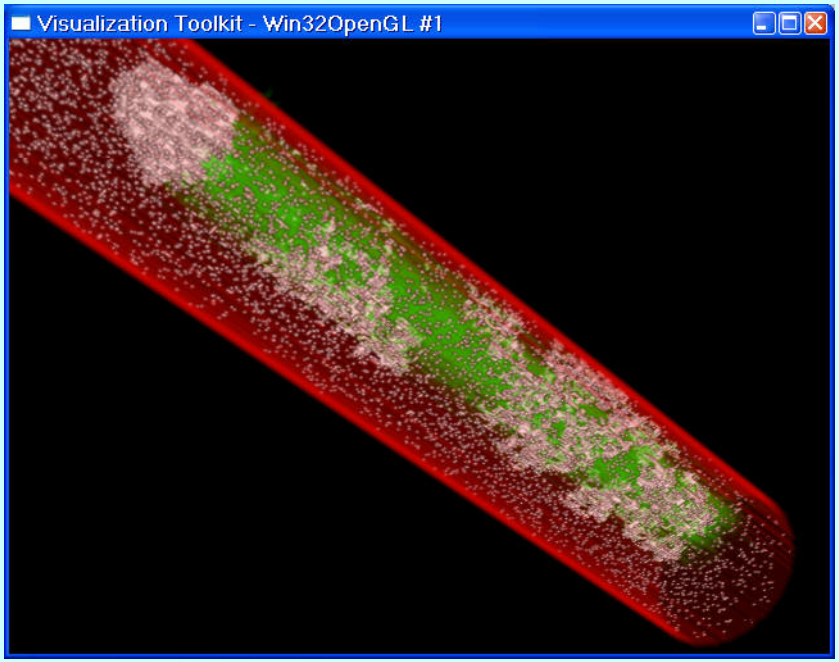
3-D Bloodstream



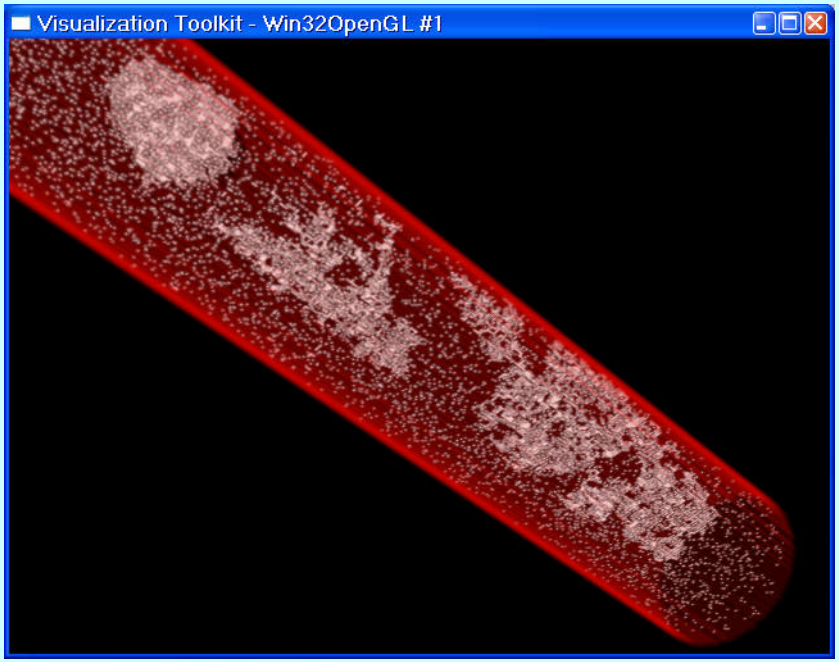
3-D Bloodstream



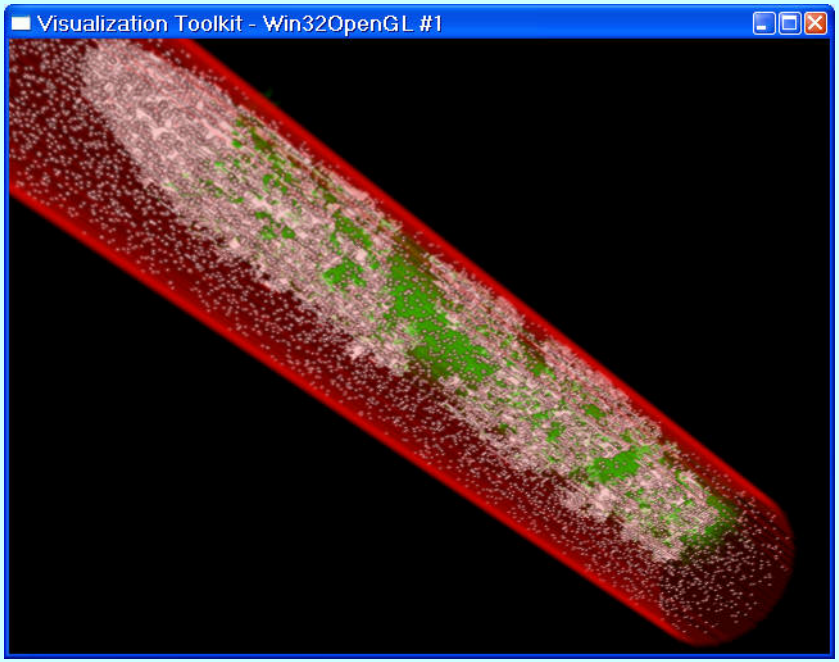
3-D Bloodstream



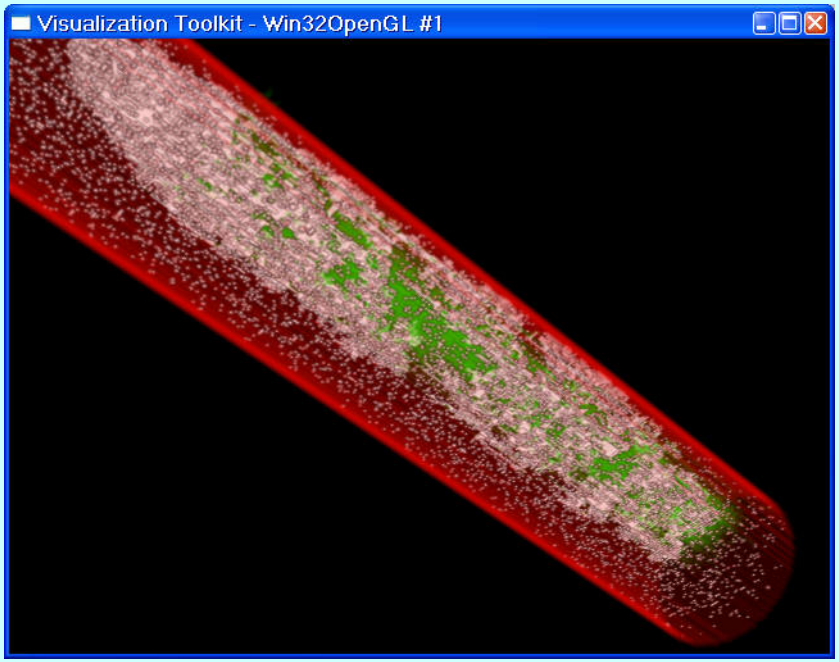
3-D Bloodstream



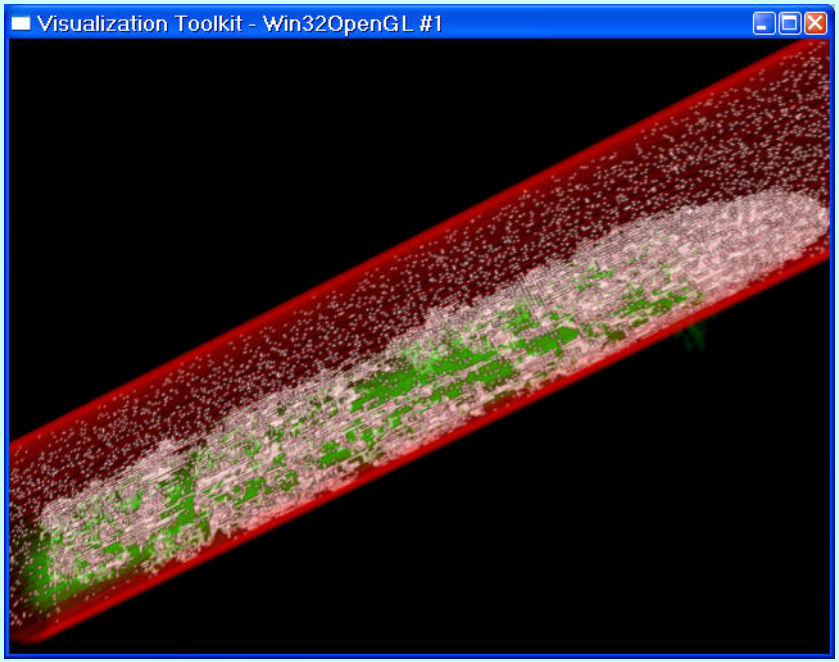
3-D Bloodstream



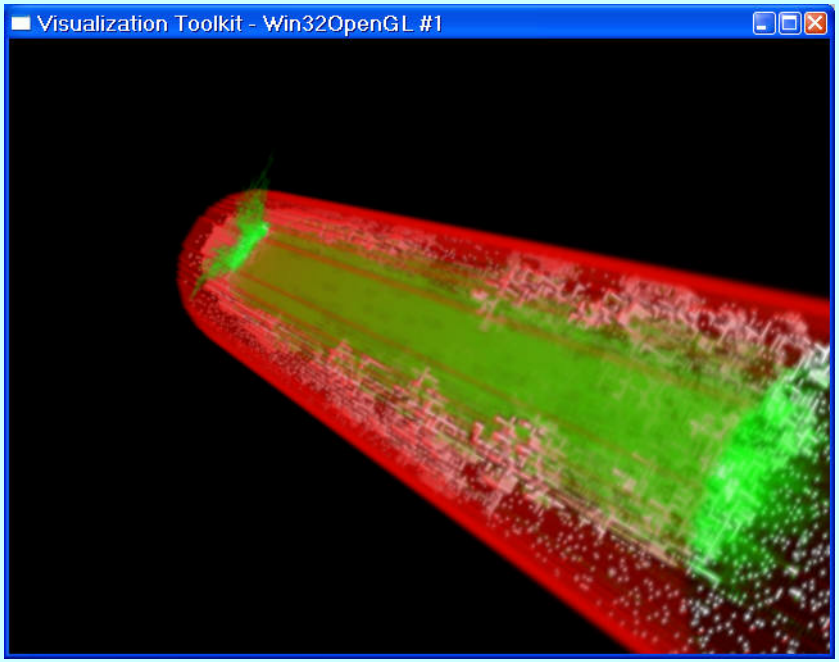
3-D Bloodstream



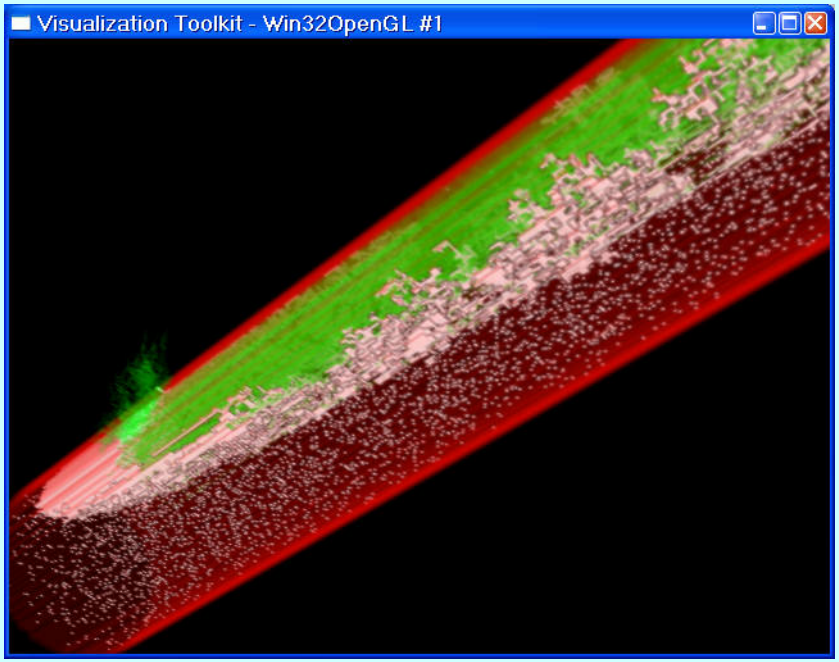
3-D Bloodstream



3-D Bloodstream



3-D Bloodstream



3-D Bloodstream