# A Tutorial on Building Cognitive Models with the EPIC Architecture for Human Cognition and Performance

**Presenter:**
**David E. Kieras**
**University of Michigan**

**Co-Presenter:**
**Anthony Hornof**
**University of Oregon**

**Collaborator on EPIC Development:**
**David Meyer, University of Michigan**

**Sponsor of EPIC Development:**
**Office of Naval Research, Cognitive Sciences Program**
**Susan Chipman, Program Manager**

# Tutorial Overview

Tutorial Purpose

Tutorial Schedule

# Tutorial Purpose

**Provide an introduction to building and running models in EPIC.**

**Learn enough about EPIC to decide whether you want to use it.**

**Psychological theory underlying EPIC de-emphasized.**
Some overview here to provide basis, but available elsewhere.
If substantive issues come up, we will try to move them off-line.

**Hands-on try-it-out activity emphasized.**
Learn what EPIC does by trying it out directly.
Production rule programming only.

- Most of tutorial is about how to write and run models at the prodution rule level, with parameter modifications as needed.
- Programming a device model in C++ is required for full usage of EPIC.
  Will only overview that here.

**Exercises focus on distinctive aspects of EPIC:**
EPIC's visual system, and its role in visual search.
Executive processes in multiple-task situations.

## Tutorial Schedule

**Introductions, Overview of the Tutorial** (.25 hr.)

**Brief Survey of EPIC for the Tutorial** (.75 hr.)

**Exercise 1. Running and Observing an Existing Model** (1 hr.)

**Exercise 2. Modifying an Existing Model** (1 hr.)

**Modeling Multiple-Task Execution in EPIC** (.5 hr.)

**Exercise 3: Programming a Multi-task Model** (1 hr.)

**Overview of Device Processor Programming** (.5 hr.)

**Wrap-up Discussion** (.5 hr.)

# Description of the EPIC Architecture

Goals of EPIC Project

The EPIC Architecture

Diagram of the Current EPIC Architecture

Perceptual Processors

Motor Processors

Motor Processors (continued)

Cognitive Processor

Sample Rules - 1

Sample Rules - 2

Distinctive Features of EPIC Approach

Importance of Perceptual-Motor Constraints

Some Important Perceptual-Motor Constraints

Modeling Issues - Inputs and Outputs

# Goals of EPIC Project

**Develop a predictive and explanatory theory of human cognition and performance.**

Codify scientific knowledge.
Elucidate executive processes.
Explain multitask performance.

**Make it accurate and practical enough to use for simulated humans in system design methodology.**

Simulate the human-machine system; iterate machine design to achieve required system performance.
Similar to parallel-developed GOMS modeling system for HCI design.

# The EPIC Architecture

**Basic assumptions**

Production-rule cognitive processor.
Parallel perceptual and motor processors.

**Fixed architectural properties**

Components, pathways, and most time parameters

**Task-dependent properties**

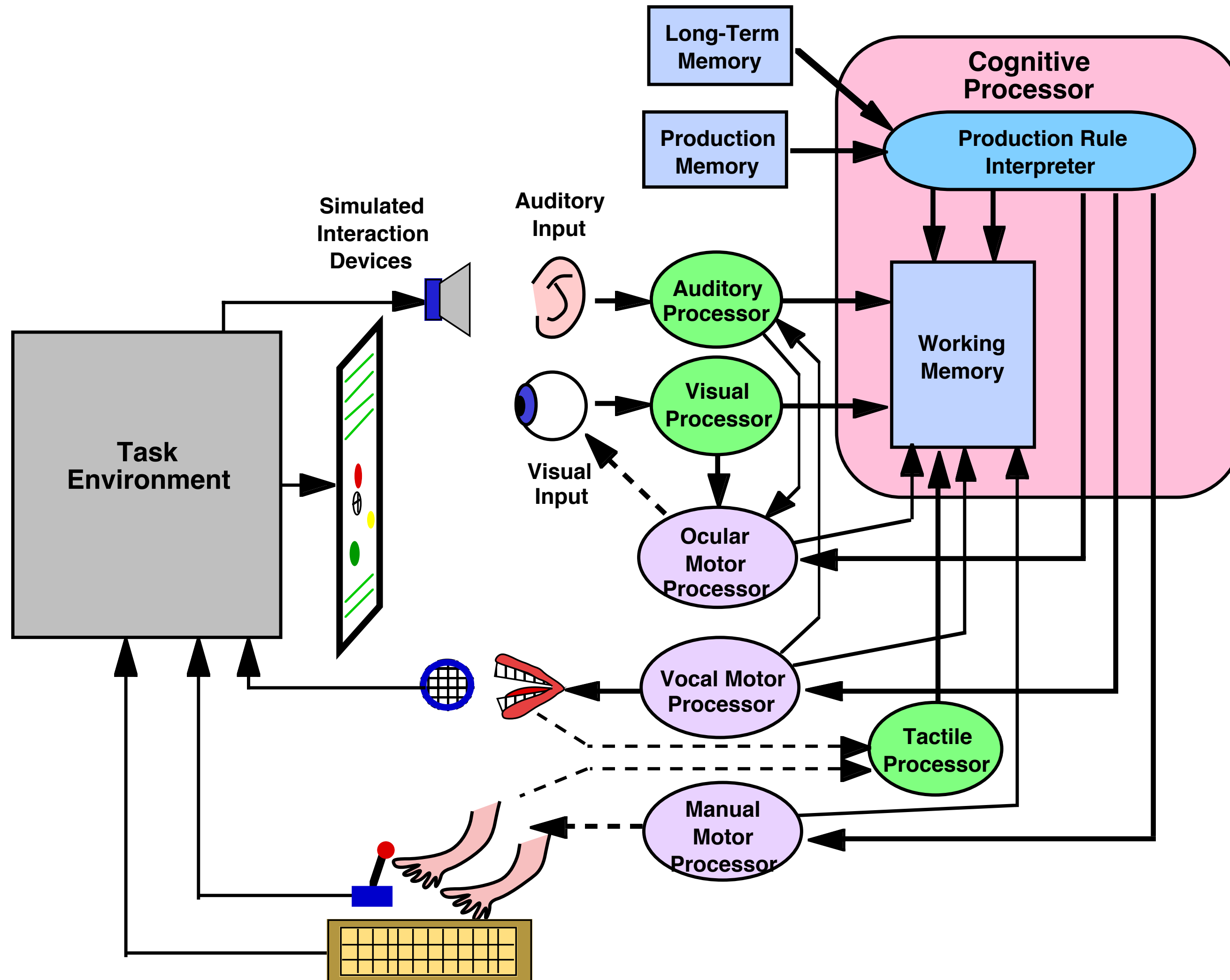
Cognitive processor production rules.
Perceptual recoding.
Response requirements and styles.

**Currently, a performance modeling system.**

Theory of human performance not finished - plenty of work still to be done!
But learning mechanisms being planned.

See *Epic Architecture Principles of Operation* for details.

# Diagram of the Current EPIC Architecture



# Perceptual Processors

**Inputs**

Symbolically-coded changes in sensory properties.

**Outputs**

Items in modality-specific partitions of Working Memory.

**Auditory**

- Not used in this tutorial - see Principles of Operation document.

**Visual**

- Eye model transduces visual properties depending on retinal zone.
  Fovea, Parafovea, Periphery.
  Other availability functions possible; subject of research.
- Visual properties take different times to transduce.
  Detection: Timing: 50 ms.
  Shape information: Timing: 100 ms, typical.
- Encodes additional perceptual properties in Visual Working Memory.
  Timing: Additional 100 ms, typical.
- Maintains internal representation of visual objects.
  Location information directly available to motor processors.
- Certain changes reported to the Ocular Motor Processor.
  Onsets, movement.

# Motor Processors

**Inputs**

Symbolic instructions from the cognitive processor.

**Outputs**

Symbolic movement specifications and times.

**Motor processing**

Movement instructions expanded into motor features.

- E.g., style, effector, direction, extent.

Motor movement features prepared.

- Features can be prepared in advance or re-used.
  Later execution is faster.

Movement is physically executed.

**Timing:**

50 ms/feature preparation.
50 ms movement initiation delay.
Movement-specific execution time (e.g. Fitts' Law).

**Cognitive processor informed of current state.**

# Motor Processors (continued)

**Ocular Motor Processors (voluntary & involuntary)**

Generates eye movements from commands or visual events.

- Long-loop cognitive control - voluntary processor.
  Saccades.
- Short-loop visual control - involuntary processor.
  Saccades and smooth movements.

**Manual Motor Processor**

Both hands are controlled by a single processor

- A fundamental limitation.

A variety of hand movement styles (more to be re-implemented)

- Pointing, button pushing, controlling.

**Vocal Motor Processor**

Not very elaborated at this time.

# Cognitive Processor

**Programmed with production rules:**

Rules represent the procedural knowledge required to perform the task.
Uses the Parsimonious Production System (PPS) interpreter - very simple.

**Interpreter updates working memory on each cycle, and fires all rules that match on each cycle.**

Timing: 50 ms/cycle

**Working Memory partitions:**

**Modal stores:**

- Visual
  Represents current visual situation.
  Slaved to visual input.
- Auditory
  Items disappear with time.
- Motor
  States of motor processors.

**Control store:**

- Goal, Step, Strategy, Status items for method control and sequencing.

**Tag store:**

- Associates a modal working memory item with a symbol designating a role in production rules - analogous to a variable and its binding.

**Amodal WM:**

- Additional information whose psychological status is not yet clear.

# Sample Rules - 1

```
(Top-see-fixation-point
        If
        (
                (Goal Do Visual_search)
                (Step Waitfor Fixation-present)
                (Visual ?object Shape Cross_Hairs)
                (Visual ?object Color Red)
        )
        Then
        (
                (Add (Tag ?object fixation-point))
                (Delete (Step Waitfor Fixation-present))
                (Add (Step Waitfor probe-present))
        ))
```

# Sample Rules - 2

```
(Top-make-response
    If
    (
        (Goal Do Visual_search)
        (Step Make Response)
        (Tag ?target target)
        (Tag ?cursor cursor)
        (Motor Manual Modality Free)
    )
    Then
    (
        (Send_to_motor Manual Perform Ply ?cursor ?target Right)
        (Delete (Step Make Response))
        (Add (Step Make Response2))
    ))
```

# Distinctive Features of EPIC Approach

**Emphasis on executive processes that coordinate multitask performance.**
Multitask performance stresses the architecture.
An important but underdeveloped area for theory.

**Take advantage of underexploited but powerful constraints:**
Perceptual-motor abilities and limitations.
Detailed and exact quantitative fits to human data.

**“Zero-based” theoretical budget:**
Question traditional assumptions.
Do not add a mechanism until it is needed to account for data.
Avoid egregious assumptions of cognitive limitations.
Prefer strategy limitations over architectural ones.

**Focus on major phenomena and mechanisms that are important determinants of performance, rather than minor “interesting” ones.**

**Compare multiple strategies for doing a task.**
Isolate strategy effects from architectural properties.

# Importance of Perceptual-Motor Constraints

**In many tasks, performance is primarily limited by peripheral perceptual-motor activities rather than central cognitive limitations.**

- Account for many key issues in a variety of tasks.
- Analogous to traditional bottlenecks in computing.

**Ignoring can result in absurd models.**

**Can ignore only in very heavily cognitive tasks.**

# Some Important Perceptual-Motor Constraints

Visual resolution depends on eye position, specifics of visual properties.

Different eye movement types and timing.

Different hand movement types and timing.

Cross-constraints for visually-aimed movements.

Hands bottlenecked through single processor, unless two-hand movement style has been learned.

Auditory and speech input recognition timing.

Speech output timing.

Verbal working memory uses auditory and vocal processors, thereby limited by decay and production rate properties.

Visual working memory appears to have large short-term capacity, small but reliable long-term capacity under cognitive control.

# Modeling Issues - Inputs and Outputs

**What you put into an EPIC model for a task:**

**A simulated device:**

Represents system under analysis or design.
Generates display events according to supplied scenarios.
Responds to inputs from simulated human.

**A simulated human specified with:**

A production-rule representation of a task strategy.
Values for task-specific time parameters.
Choices of movement styles not determined by the task.

**EPIC supplies the general cognitive architecture:**

Structure of interconnected processors.
Task-independent process timing and constraints. E.g.:

- Visual resolution constraints.
- Basic perceptual processing times.
- Eye movement times.
- Hand movement times depending on style, distance.

**Run EPIC model and get:**

- Predicted times and action sequences for all possible scenarios subsumed by the model.
- Generative property: A single rule set generates behavior for a large set of specific scenarios.

# Modeling Issues - Fixed and Free Parameters

**What is fixed**

- The connection and mechanisms of processors.
- Most time parameters.
- The feature structures and time parameters of motor processors.

**What is free to vary**

- Task-specific production rule programming
  - Constrained by the requirement of performing the task correctly and efficiently.
- Task-specific perceptual encoding type and times
  - Must be constant over similar stimuli
- The style of movements
  - Often not determined by the task

# Current State of EPIC  Software

**Previous version implemented in LISP:**

Software structure was chaotic, difficult to maintain, lacked clear correspondence to psychological structure.

LISP is powerful and flexible, but lacks good support for higher level organization of complex software.

**New version implemented in C++**

Software structure is clearly aligned with psychological structure, and is easy to modify and maintain.

C++ has good support for complex software, and is very efficient, but is relatively harder to work in than LISP (assuming equal knowledge).

**State of the new version compared to the old:**

Not everything has been re-implemented in the new version.

- But the old capabilities will be easy to add to the new version.

Some key new psychological ideas not yet incorporated.

- Need more theoretical work to define them in terms of the  architecture.

**Approach:**

An architecture is never completed - so instead of taking forever to complete it, let's start using it, and let the needs of modelers dictate what gets added next.

# Model for Exercises 1 and 2

Williams (1967) Visual Search Task

Sample Display - from Model

Basic Results:

More Detail on the Visual System - Structure

The EpicApp Views of the Visual System

Simple Visual Availability Model

"Starter" EPIC Model for Williams' Task

PPS Memory Contents and Run Messages (1 of 2)

PPS Memory Contents and Run Messages (2 of 2)

# Williams (1967) Visual Search Task

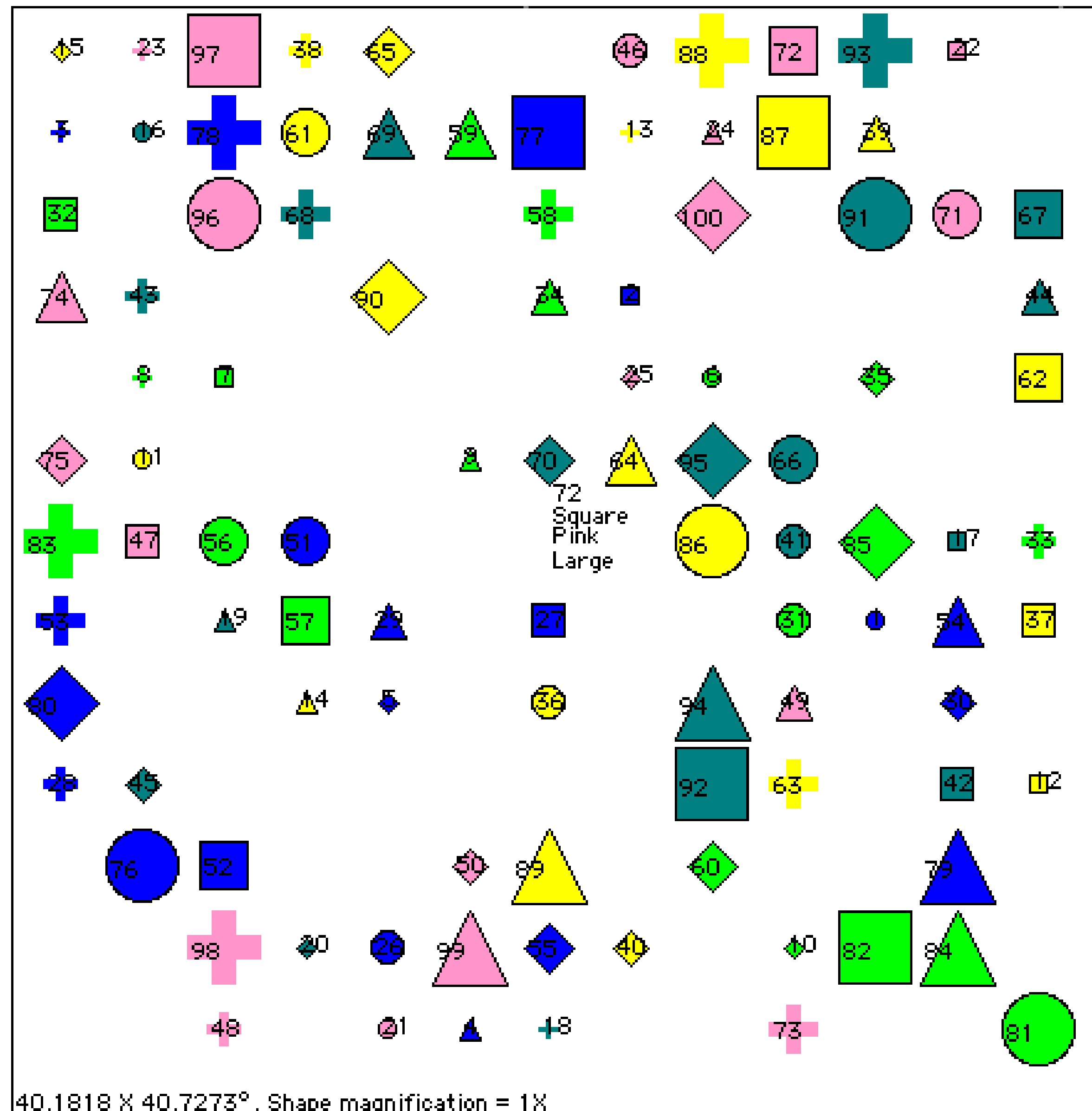
**A classic study of eye movements in visual search**

Williams, L.G. (1967). The effects of target specification on objects fixated during visual search. In A.F. Sanders(Ed.) *Attention and Performance*, North-Holland. 355-360.

**Summary**

A text specification of target, 100 objects on display: 4 sizes, 5 shapes, 5 colors. Each object has an ID number, specification always includes the ID number. Find the object with the specified ID number. Specification could also include color, size, shape of target (all eight combinations used).

# Sample Display - from Model



# Basic Results:

| Fixation Type | Prop. | RT(s) | Est. Fixations |
|---|---|---|---|
| ID only* | .20 | 22.8 | 68 |
| Color | .61 | 6.8 | 20 |
| Size w/o Color | .36 | 16.1 | 48 |
| Very Large | .54 | | |
| Oth. Size | .30 | | |
| Shape only | .25 | 20.7 | 62 |

**Results averaged over all conditions where the stimulus property was in the probe.**

E.g. all cases where Color was supplied were averaged together.

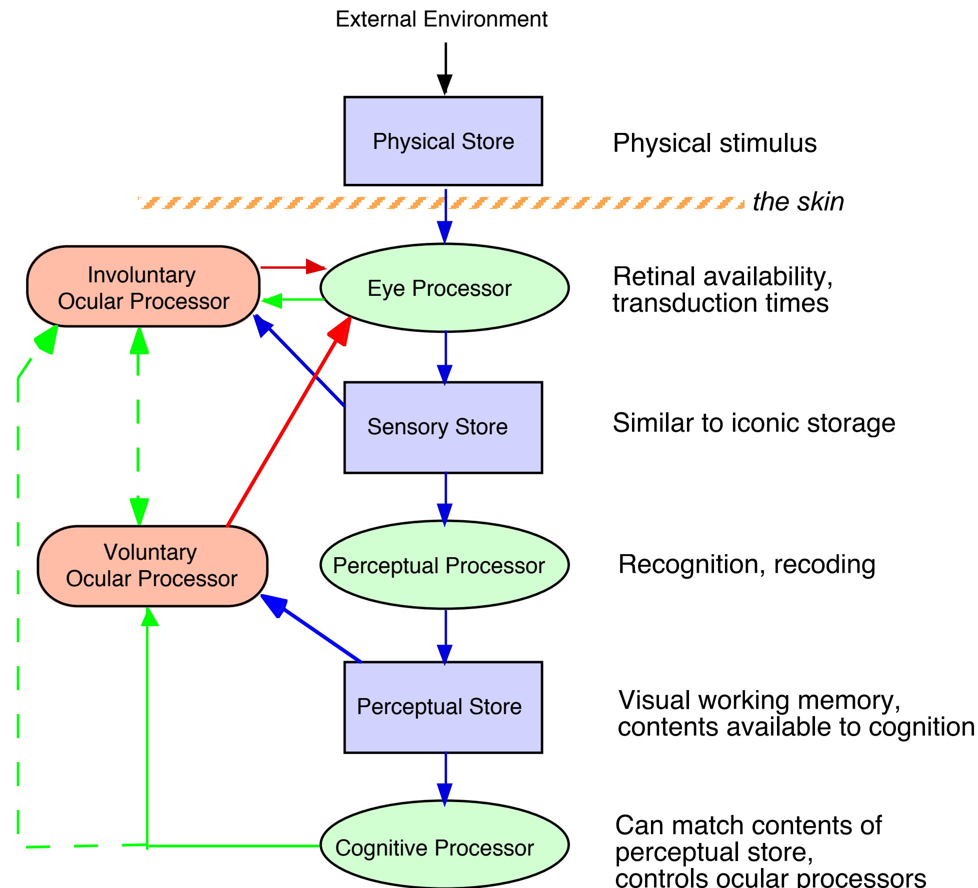
***When only the ID is supplied, fixations uniformly distributed across features; very slow.**

Proportion shown is for color and shape; size is .25

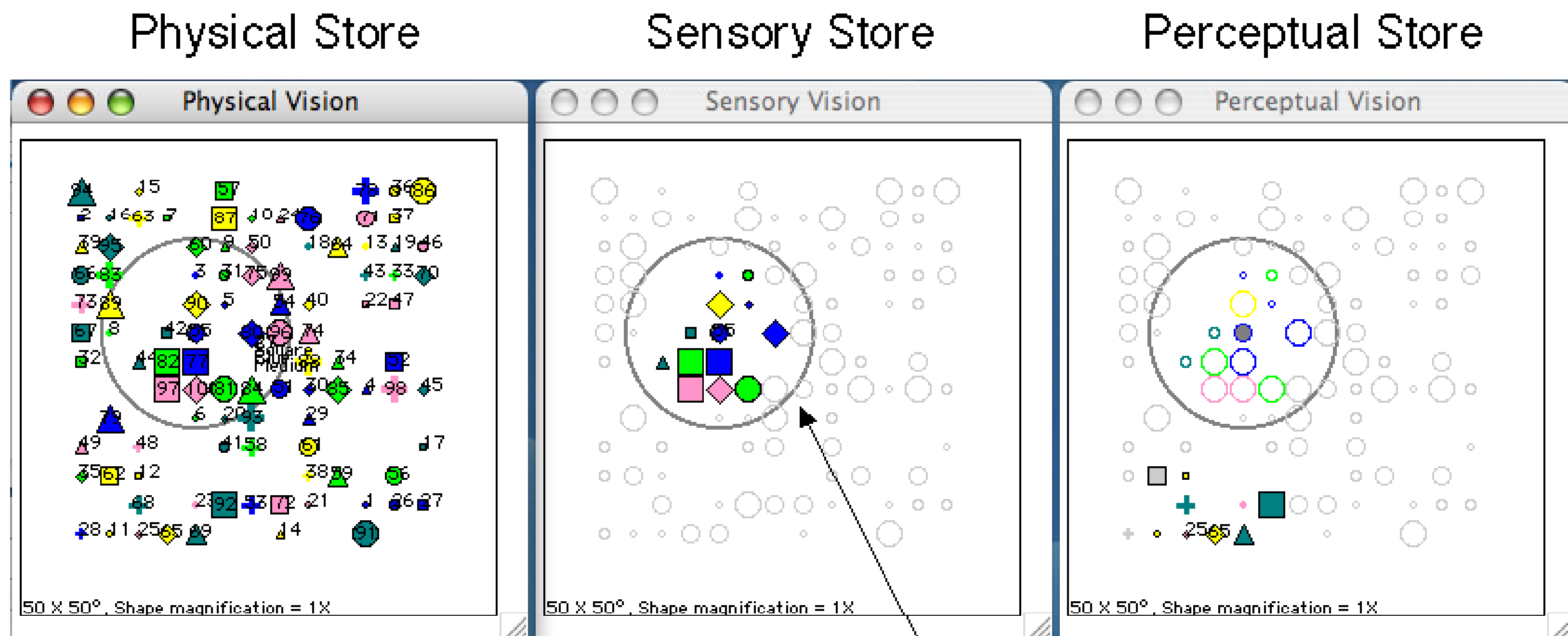
**Eye movements strongly guided by color, not by shape, by largest size only.**

**Number of fixations estimated from RT and reported rate of 3/sec.**

# More Detail on the Visual System - Structure



# The EpicApp Views of the Visual System




**You can watch object features gradually become available in each store from left to right, and gradually decay from the rightmost store.**
Text, only available in the fovea, is currently making its way across.

***Geometric* location and size are intrinsic to an object, but the *encoded* size requires processing.**

**Visual perceptual memory provides some lag, making properties available after eye has moved away or the object disappeared.**
Current guesstimate is about a second.

# Simple Visual Availability Model

**Where on the retina can different visual properties be detected?**

Eccentricity - angle on the retina between fovea and object.

Powerful effect of increasing eccentricity.

**Retinal zones**

The retina is divided into a set of zones, defined as concentric circles around the fovea.

Availability for each property defined in terms of which zones they are available in.

- E.g. text only in fovea, shape in parafovea.

Zone boundaries might fluctuate.

But different visual properties share the same zone boundaries.

**Actually, availability depends on more than just distance from the fovea, and differs depending on the visual property.**

This can be represented in EPIC, but using simpler model for now.

# "Starter" EPIC Model for Williams' Task

**Uses default availability settings.**

Spatial Availability:

- Color and Shape available within 7.5 degrees.
- Text (the ID number) and Size (encoded) available in fovea (1 degree).

Temporal Availability:

- Color and Size available with 50 ms delay.
- Text and Shape available with 100 ms delay.

**Production Rules: Random_ID_Search.prs**

Move the eyes to a randomly selected object that has not been checked.

- (Ignore shape, color, and size.)

If the object's ID matches the target ID, press the button to end the trial.

If the ID does not match, tag the object as "checked".

Loop until the target is found.

# PPS Memory Contents and Run Messages (1 of 2)

```
...
3350:Cycle 66:
Goal
  (Goal Do Visual_search)
Step
  (Step Look_at Random_Object)
Tag
  (Tag 61 Target_Label)
  ...
  (Tag Vpsychobj7 fixation_object)
Visual
  (Visual Vpsychobj1 Above Vpsychobj2)
  (Visual Vpsychobj1 Eccentricity Periphery)
  (Visual Vpsychobj1 Status Visible)
  (Visual Vpsychobj1 Text 61)
  ...
  (Visual Vpsychobj7 Color Yellow)
  (Visual Vpsychobj7 Eccentricity Fovea)
  (Visual Vpsychobj7 Shape Triangle)
  (Visual Vpsychobj7 Size Medium)
  (Visual Vpsychobj7 Status Visible)
  (Visual Vpsychobj7 Text 39)
```

# PPS Memory Contents and Run Messages (2 of 2)

```
Motor
  (Motor Manual Execution Free)
  (Motor Manual Modality Free)
  (Motor Manual Preparation Free)
  (Motor Manual Processor Free)
  ...
  (Motor Ocular Execution Free)
  ...
  (Motor Vocal Execution Free)
  ...
*** Rules fired:
Fire: Continue_random_search
  Bindings: ((?label:39)
  (?nominee:Vpsychobj5)
  (?object:Vpsychobj7)
  (?target_label:61))
  Add: (Tag Vpsychobj5 checked)
  Add: (Tag Vpsychobj5 fixation_object)
  Delete: (Tag Vpsychobj7 fixation_object)
  Motor command:
  (Ocular Perform Move Vpsychobj5)
3400:Cycle 67:
...
5300:Cycle 105: No rules fired
5350:Cycle 106: No rules fired
...
```

## Exercise 1: Running an Existing Model

**Follow the the Exercise 1 handout. The basic structure is:**

**Run the model and learn the basic EpicApp control and displays.**

**Study the flow of information in the memories and processors.**

**Discuss.**

# Working with PPS Production Rules

Production Rule Syntax - 1

Production Rule Syntax - 2

Production Rule Execution: Matching

Production Rule Execution: Firing

Contents of the Production Rule File

Processor Parameters

Basic Programming Concept

Programming Conventions

Programming Sequential Flow of Control

Hierarchical-Sequential Flow of Control

Basic Motor Commands

Basic Motor Processor States

Slow Movement Sequence

# Production Rule Syntax - 1

**Production system working memory (the “database”) contains *clauses***

Clause - a list of Symbols
Symbol - a whitespace-delimited character string or number.
Only constants are present in the working memory.

**First Symbol by convention indicates the “kind” of clause:**

**Psychological working memories**

Modality-specific working memory systems.

- Visual, Auditory, Tactile, Motor Manual/Vocal/Ocular

WM: Hypothetical amodal working memory.

**Control Store**

Goal: current goal being worked on; enables set of production rules constituting the method for the goal
Step: current sequential step in the method.
Strategy: selects which branches to take for different strategies within a method
Status: states of a method

**Tag store:**

Attach a symbol to an object in a modal working memory to “tag” its role in subsequent production rules.

- E.g. “visual object #23 is the current stimulus”
  (Tag Visobj23 Current_stimulus)

# Production Rule Syntax - 2

**(<rulename> If (<condition> ....) Then (<action> ...))**

**<rulename> : Symbol**

**<condition> :**

<conditionpattern> I (Not <conditionpattern>) I <predicate>
<conditionpattern> : (<conditionterm> ...)
<conditionterm> : <term> I <wildcard>
<term> : Symbol I <variable>
<variable> : ? + string
<wildcard> : ???
<predicate> :
(Randomly-choose-one) I
(Different <term> <term>) I
(Equal <term> <term>) I etc.

**<action> :**

(Add <actionpattern>) I
(Delete <actionpattern>) I
(Send_to_motor <modality> <command> <actionpattern>)

<actionpattern> : (<term> ...)
<modality> : Manual I Vocal I Ocular
<command> : Modality specific Symbol

# Production Rule Execution: Matching

**On each *cycle:***

**Update the data base from the previous cycle:**
Delete action clauses are deleted from the production system memory.
Add action clauses are added to the production system memory.

**Compute the set of rules whose conditions are currently matched - the "fired" rule list.**
All conditions in a rule must be matched to clauses in the memory, and all predicates must be true.

**Only constants in the conditions - no variables:**
If all conditions are matched, the rule fires.

**Conditions and predicates include variables:**

- If a variable appears in a condition, it is assigned a value that allows the condition to match a clause in the memory. It must have the same value in the other rule conditions.
- Each possible set of variable values that satisfies the rule conditions is computed - the *binding sets*.
- The binding sets are then "filtered" through the predicates.
- If at least one binding set remains, the rule fires.

# Production Rule Execution: Firing

**On each *cycle:***

**For each rule in the fired rule list:**

**For each binding set:**

Bind the variables to the values.

Execute the rule actions in order listed

- Each variable replaced with its value.
- Add and Delete clauses are saved for next cycle.
  Rules cannot affect each other within a cycle.
- Motor commands are sent to the processors.
  Arrive just before the beginning of the next cycle.

# Contents of the Production Rule File

**A simple text file, edit with your favorite plain-text editor.**
If using a programming IDE, use your source code file editor.

**Setting the initial contents of production system working memory:**
(Define Initial_memory_contents <clause> ...)

**Defining a "named location" to serve as a psuedo-object to use as a target for eye movements or aimed hand movements.**
(Define Named_location <location_name> <x > <y>)
The <x> and <y> values are floating-point values in degrees of visual angle, where (0.0, 0.0) corresponds to "straight ahead".

**(Define Parameters <parameter_specification> ....)**
Next slide

**Production rules**
Order and whitespace are ignored, but should be human-readable.

# Processor Parameters

**Each processor has a set of parameters that are normally set at standard or typical values.**

All parameters printed out at beginning of a simulation run.

These values can be modified in the production rule file.

Modified values will be shown in the parameter listing.

**(Define Parameters <parameter_specification> ....)**

<parameter_specification> : (<processor_name> <parameter_name> <specifications>)

The specifications depend on the processor and parameter.

- The possibilities and how to modify them can be inferred from the dump of all parameter values at the start of a run.

**Some specific examples:**

A simple Eye processor parameter whose value is either constant at 1.0 or randomized as a N(1, .1) random variable, sampled whenever the parameter value is sample.

- (Eye Eccentricity_fluctuation_factor Fixed 1.0)
- (Eye Eccentricity_fluctuation_factor Normal When_used 1.0 .1)

A perceptual processor encoding time for the Text property:

- (Visual_perceptual_processor Recoding_time Text 50)

# Basic Programming Concept

**The flow of control must be explicit in the rules, not implicit in the architecture.**

Sequential processing: Each rule disables itself and enables the next rule in sequence.

Goal-directed processing: Each rule governed by a goal condition.

A variety of flow controls are possible and useful.

**Procedures represented as GOMS-like methods.**

A set of rules controlled by the same Goal condition.

Sequence controlled by Step conditions.

**Demons: Free-standing rules.**

Can fire at any time.

Easily provide interruptability.

**Multi-threaded processing:**

Multiple goals, steps can be present.

Multiple rules can fire simultaneously and independently.

Chains of rules can be spawned, terminated at will.

# Programming Conventions

**For ease and consistency in rule writing, each clause consists of a certain number of Symbols:**

Goals: Goal, a verb, a noun

- Verb+Noun abbreviated in following summaries.

Steps: Step, a method name, a verb, a noun

- Verb+Noun abbreviated in following summaries.

Tags: Tag, a method name, a modal object name, a tagging Symbol

**Method names**

A Symbol based on the Goal for the method.

Prevents some name collision problems.

- Remember: memory is shared by all rules!

Enable easy matching of all Step and Tag clauses belonging to a method.

- E.g. for “cleanup" at the end of a method.

Abbreviated as a single lower-case letter in following summaries.

**No harm in leaving method name out of top-level rule clauses.**

Avoids some clutter.

# Programming Sequential Flow of Control

```
(Top1
        If ((Goal Top)(Step T1))
        Then ((Delete (Step T1))(Add (Step T2))))
(Top2
        If ((Goal Top)(Step T2))
        Then ((Delete (Step T2))(Add (Step T3))))
(Top3
        If ((Goal Top)(Step T3))
        Then ((Delete (Step T3))(Add (Step T4))))
```

**If memory initially contains (Goal Top) and (Step T1), the rules will fire in sequence.**

Each rule is triggered by the Goal and a Step.

Each rule's actions disable the rule, and enable the next rule.

# Hierarchical-Sequential Flow of Control

```
(Top1_call_submethod
        If ((Goal Top)(Step T1))
        Then ((Add (Goal Sub))(Delete (Step T1))(Add (Step T2))))
(Top2_wait_for_return
        If ((Goal Top)(Step T2)(Not(Goal Sub))
        Then ((Delete (Step T2))(Add (Step T3))))
... ... ... ... ... ...

(Sub_startup
        If ((Goal Sub) Not(Step s ???))
        Then ((Add (Step s S1))))
(Sub1
        If ((Goal Sub) (Step s S1))
        Then ((Delete (Step s S1))(Add (Step s S2))))
(Sub2_terminate
        If ((Goal Sub) (Step s S2))
        Then ((Delete (Step s S2))(Delete (Goal Sub))))
```

**Top method invokes a submethod:**

Add the submethod Goal; wait at the next Step for the Goal to disappear.

**The submethod structure:**

Startup rule triggered by Goal and lack of submethod steps.

- Enables first step, disables itself.

Termination rule Deletes the Goal and the last Step.

- Disappearance of subgoal signals completion to invoking method.

# Basic Motor Commands

**Rule programming conventions**

(rule_name
If ( <other conditions>
(<motor processor state>))
Then ((Send_to_motor <modality> <command> <specifications>)
<other actions>)

**Motor command syntax**

<modality> : Manual | Vocal | Ocular
<command> : Prepare | Perform
<specifications> :

- Ocular:
  Move <target_object_name>
- Manual:
  Punch <key_name> <hand> <finger>
  Specified hand and finger is assumed to be poised over the key whose name is sent to the device.
  Ply <cursor_object_name> <target_object_name>
  Assumes hand is on control device (e.g. Joystick)
  Point <target_object_name>
  Assumes a default cursor object named Cursor

# Basic Motor Processor States

**Modality Free**

Any previously commanded movement is complete, and no activity for a new movement is underway.

- Modality Busy means that some type of movement processing is underway.

**Processor Free**

Any previously commanded movement has already started execution. A new movement command will be accepted and prepared, and will start execution when any previously commanded movement execution is complete.

- Processor Busy means that a new command will either "jam" the processor or overwrite a previously prepared movement.

**Other states aren't as important.**

See Principles of Operation document.

# Slow Movement Sequence

```
(Slow_wait_to_start
If ((Goal Make Responses) (Step make first)
        (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Punch J Right Index)
        (Delete (Step make first))(Add (Step make second)))

(Slow_wait_for_first_done
If ((Goal Make Responses) (Step make second)
        (Motor Manual Modality Free))
Then ((Send_to_motor Manual Perform Punch K Right Middle)
        (Delete (Step make second))(Add (Step waiton second)))

(Slow_wait_for_second_done
If ((Goal Make Responses) (Step waiton second)
        (Motor Manual Modality Free))
Then ((Delete (Step waiton second))(Add (Step continue  working)))
```

**Wait for any previous activity to be done before starting.**

**Wait for each movement to be fully complete before starting the next.**

**Wait for last movement to be fully complete before continuing processing.**

# Fast Movement Sequence

```
(Fast_start_first
If ((Goal Make Responses) (Step make first)
        (Motor Manual Processor Free))
Then ((Send_to_motor Manual Perform Punch J Right Index)
        (Delete (Step make first))(Add (Step make second)))

(Fast_start_second_and_continue
If ((Goal Make Responses) (Step make second)
        (Motor Manual Processor Free))
Then ((Send_to_motor Manual Perform Punch K Right Middle)
        (Delete (Step make second))(Add (Step continue working)))
```

**Get the motor processor started on each movement as soon as it will accept and ensure execution of a movement.**

**Don’t hang around for last movement to be completed.**

# Preparing a Movement in Advance

```
(Prepare_the_response
If ((Goal Make Response) (Step get ready)
        (Motor Manual Processor Free))
Then ((Send_to_motor Manual Prepare Punch J Right Index)
        (Delete (Step get ready))(Add (Step waitfor stimulus)))

(Punch_when_red_appears
If ((Goal Make Response) (Step waitfor stimulus)
        (Visual ?stimulus Color Red)
        (Motor Manual Preparation Free))
Then ((Send_to_motor Manual Perform Punch K Right Middle)
        (Delete (Step waitfor stimulus))(Add (Step continue working)))
```

**Prepare command causes movement to be prepared, but execution is not started.**

**Subsequent Perform command will start immediate execution if same movement features involved.**

**If delay is long enough to allow preparation to complete, response will be much faster (e.g. 100 ms) than if no preparation done.**

# Exercise 2: Modifying an Existing Model

Follow the the Exercise 2 handout.

The basic idea is to make a substantive modification to the Williams 1967 model, re-run and compare the results.

Possible modifications:

- Use features besides just ID.
- Eliminate the memory for which objects have already been inspected.
- Other ideas?

Report and discuss results, questions, and issues.

# Multiple-task Processing in EPIC

Programming Parallel Flow of Control

Hierarchical-Overlapping

Multiple Subgoal Threads

Multiple Rules for a Step

Multiple Step Threads

Demons

Executive Processes in EPIC

The Primary Executive Function

Possible Executive Processes

A Parallel General Executive Process

What EPIC Enables

# Programming Parallel Flow of Control

**Procedures represented as GOMS-like methods.**
A set of rules controlled by the same Goal condition.
Sequence controlled by Step conditions.

**Demons: Free-standing rules.**
Can fire at any time.
Easily provide interruptability.

**Multi-threaded processing:**
Multiple goals, steps can be present.
Multiple rules can fire simultaneously, independently.
Chains of rules can be spawned, terminated at will.

# Hierarchical-Overlapping

```
(Top1_start_submethod
        If ((Goal Top)(Step T1))
        Then ((Add (Goal Sub))(Delete (Step T1))(Add (Step T2))))

(Top2_keep_processing
        If ((Goal Top)(Step T2))
        Then ((Delete (Step T2))(Add (Step T3))))
        ... ... ... ... ... ...
(Top8_wait_for_submethod_complete
        If ((Goal Top)(Step T8)(Not(Goal Sub))
        Then ((Delete (Step T8))(Add (Step T9))))
```

**The top method starts a submethod, which starts executing while the top level method continues to execute.**

**At step Top8, waits for the submethod to complete before continuing.**

# Multiple Subgoal Threads

```
(Top1_start_two_submethods
        If ((Goal Top) (Step T1))
        Then ((Add (Goal Sub1))(Add (Goal Sub2)
                (Delete (Step T1))(Add (Step T2))))

(Top2_wait_for_both_complete
        If ((Goal Top)(Step T2)
                (Not(Goal Sub1)(Not(Goal Sub2)))
        Then ((Delete (Step T2))(Add (Step T3))))
```

**The top method starts two submethods, which execute concurrently, and waits for them both to complete.**

# Multiple Rules for a Step

```
(Top1
        If ((Goal Top)(Step  T1))
        Then ((Delete (Step  T1))(Add (Step  T2))))

(Top2_branch_1
        If ((Goal Top)(Step  T2) /* some condition */)
        Then (/* some action */))

(Top2_branch_2
        If ((Goal Top)(Step  T2) /* some condition */)
        Then (/* some action */))

(Top2_continue
        If ((Goal Top)(Step  T2))
        Then ((Delete (Step  T2))(Add (Step T3))))
```

**The top method enables three rules to do the next step. The two branch rules may or may not fire.**

**The continue rule always fires, disabling the branch rules, and execution continues.**

# Multiple Step Threads

```
(Top1_spawn_threads
        If ((Goal Top)(Step  T1))
        Then ((Delete (Step  T1))(Add (Step T2))
                (Add (Step ThrA1))(Add (Step ThrB1))))

(TopThreadA
        If ((Goal Top)(Step ThrA1))
        Then ((Delete (Step ThrA1))(Add (Step ThrA2))))

(TopThreadB
        If ((Goal Top)(Step ThrB1))
        Then ((Delete (Step ThrB1))(Add (Step ThrB2))))
```

**The first step not only enables the usual next step, but also spawns two additional threads of rules.**

**The threads will continue on their own execution path independently of the main thread.**

# Demons

```
(Watchfor_red
        If ((Visual ?obj Color Red))
        Then (/* some action */))

(Watchfor_disappearing_blip
        If ((Goal Process Blip)
                (Tag ?blip Current_blip)(Visual ?blip Status Disappearing))
        Then ((Delete (Goal Process Blip))(Add (Goal Abort Processing))))
```

**The first rule fires unconditionally if a red object appears in the visual working memory.**

**The second rule fires if the current blip being processed begins to disappear.**

The blip-processing is shut down, and an abort method is started.

# Executive Processes in EPIC

**An executive process is simply a set of production rules that supervise other production rule sets.**

Manipulate goals, other items, in memory to exert executive control.
No special executive-function mechanism is needed.

**Because of cognitive parallelism:**

Executive rules can execute concurrently with task rules.
Task rules can execute concurrently with other task rules.

**Simplifies theorizing about executive processes:**

If parallel processing is needed, it is available.
Each task can be represented with a distinct and independently-executing set of rules - can potentially combine any pair of task rule sets as-is.
Can apply software engineering principles of abstraction, encapsulation, and code re-use.

# The Primary Executive Function

**Assume a dual task, e.g. "Wickens" task:**

A continuous visual tracking task using a joystick.
An intermittent visual choice RT task requiring button-press responses.
Two task stimuli might be in different parts of visual field.
Both tasks both require:

- Ocular motor processor for acquiring the stimulus.
- Manual motor processor for making the response.

Can perform each task with a separate set of production rules.

**Critical resources are the motor processors.**

Can't be used simultaneously to perform different actions.
Tasks must somehow share them to prevent conflict.
Efficiency of sharing determines overall dual-task performance.

**Executive process coordinates execution of the two tasks.**

Allow both tasks to run concurrently, but prevent motor conflicts.

**Alternative to executive: Learn a single rule set for the dual task.**

A dual task can be performed as if it were a single complex task.
Merge separate production rule sets for the tasks into a single rule set governed by a merged goal:

- (Goal Do Combined_Tracking_Choice_Task)

May in fact be the result of extensive practice, but not earlier in learning!

# Possible Executive Processes

**Single-task executive:**
Ignore one of the tasks in favor of good performance on the other.

**Time-slicing trivial executive:**
On each cycle, switch task goals - run each task on alternate cycles.
If a rule checks motor state before motor command, conflicts avoided.

**Sequential executive: Run one task, then the other, then repeat.**
No resource conflicts because only one task runs at a time!

**Parallel executive: Run both tasks at the same time.**
Intervene to prevent resource conflicts only when necessary.

**Specialized Executive: Contains task-specific coordination rules.**
Where and when will visual input for each task appear?
When can motor processors be used by each task?
Good performance requires "tuning" the specialized executive.

**General Executive: General rules for coordinating any set of subtasks.**
Task rules don't have to be modified depending on other tasks present.
Executive rules do not have to be changed to suit particular subtasks.

# A Parallel General Executive Process

**General resource management during parallel task execution:**

Both tasks are running constantly - both goals always present.
One task goal is marked as higher priority than other.
Each task asks for permission to use a motor processor.
Executive grants request if resource is free, or withholds permission if not.
Task releases resource when finished with it.
Executive reallocates a released resource as needed.
In case of conflict, higher-priority task preempts lower-priority.

**Protocol:**

Each task must ask for and wait for permission to use a resource.
Each task should release the resource when finished, or be willing to be lower priority and get preempted.

**Advantages:**

Tasks control resources only when they need to.
Task rules can be written independently of the other task rules.
Executive rules are general - don't depend on specific tasks.

**Disadvantages:**

Some overhead in ask/wait/release processing.

- Can minimize with broad-but-shallow executive rule set.

# What EPIC Enables

For many years, cognitive theory has discussed executive processes and resource allocation as fundamental concepts in human multiple-task performance.

Now we can directly model them in a realistic psychological framework.

EPIC's flexible flow-of-control mechanisms makes these concepts concrete and quantitatively testable.

Can easily implement a variety of executive strategies and explore their consequences.

# Task for Exercise 3

**The "Wickens" Dual-Task Experiment**

**Background task: Constantly Track in the Background**

**Choice Task Trial Starts**

**Make Choice Response**

**Choice Trial Ends**

# The "Wickens" Dual-Task Experiment

**Reference:**

Martin-Emerson, R., & Wickens, C. D. (1992). The vertical visual field and implications for the head-up display. *Proceedings of the 36th Annual Symposium of the Human Factors Society.* Santa Monica, CA: Human Factors Society.

**Display:**

A tracking target in a fixed position.
A tracking cursor controlled by the joystick.
A circle some distance away from the tracking target.
Choice RT stimuli appear in the circle.

- A left- or right-arrow.

**Task:**

Perform compensatory tracking, keeping the cursor on the target; use joystick in right hand.

- Tracking task either high or low in difficulty.

Respond to choice stimulus as rapidly as possible.

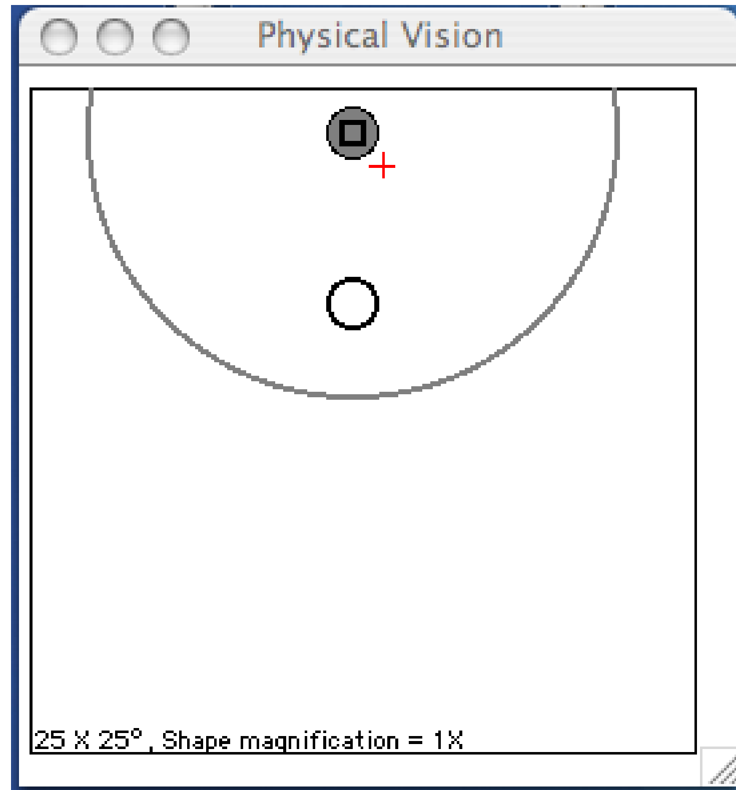
- Arrow points right or left; press corresponding button with left hand.

Choice stimulus appears at various distances from target.

**Illustrate sequence of events.**

# Background task: Constantly Track in the Background

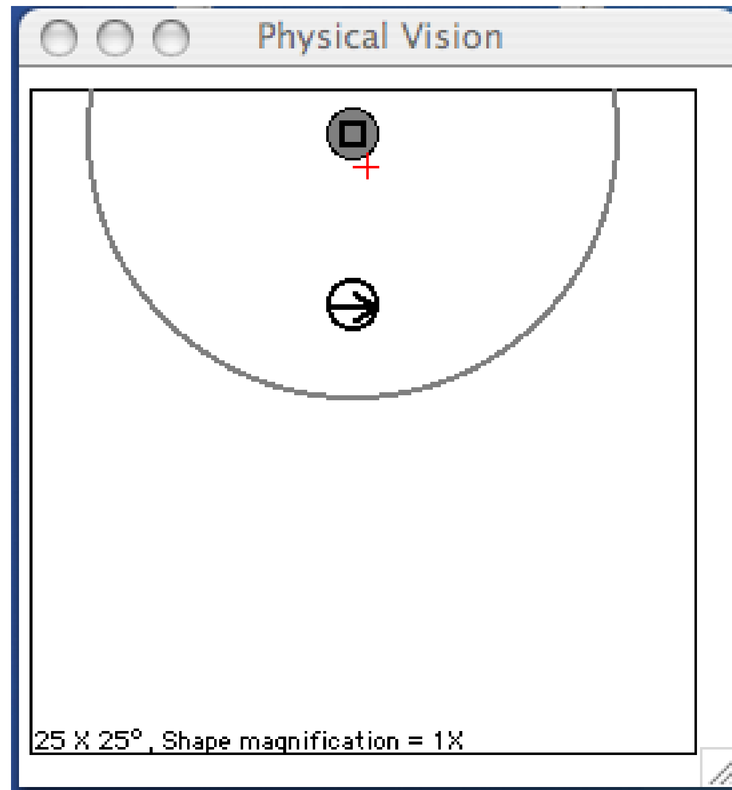
**Track by moving the joystick to keep the cross-hairs cursor on the square target.**



# Choice Task Trial Starts

Choice stimulus appears in the choice circle, some distance away from the tracking target.

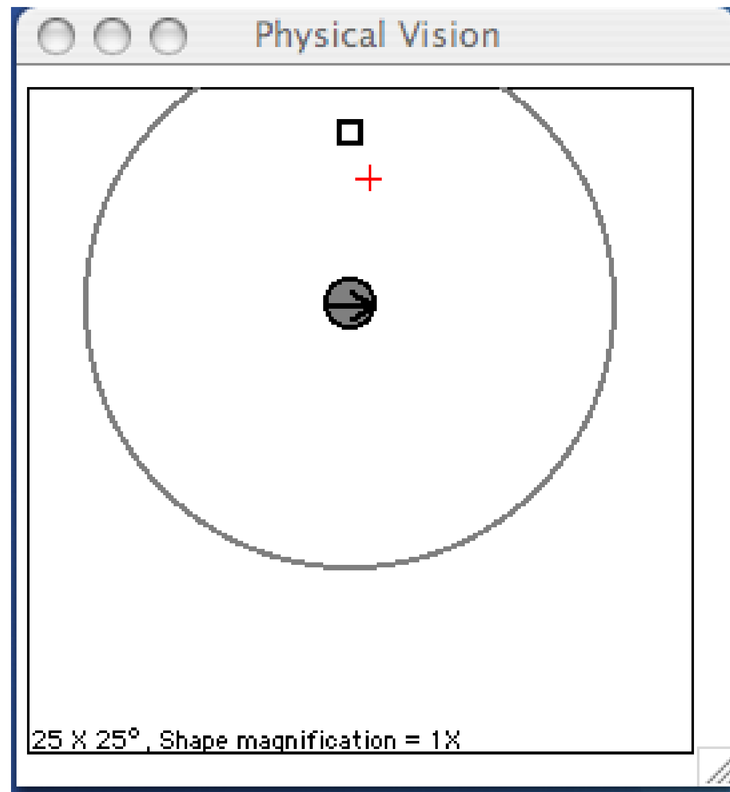
Tracking error (RMS) starts accumulating and continues for 2 seconds.



# Make Choice Response

**Look at the choice stimulus and press a button depending on whether the arrow points to the left or to the right.**

**Left-hand middle finger for left, index finger for right.**

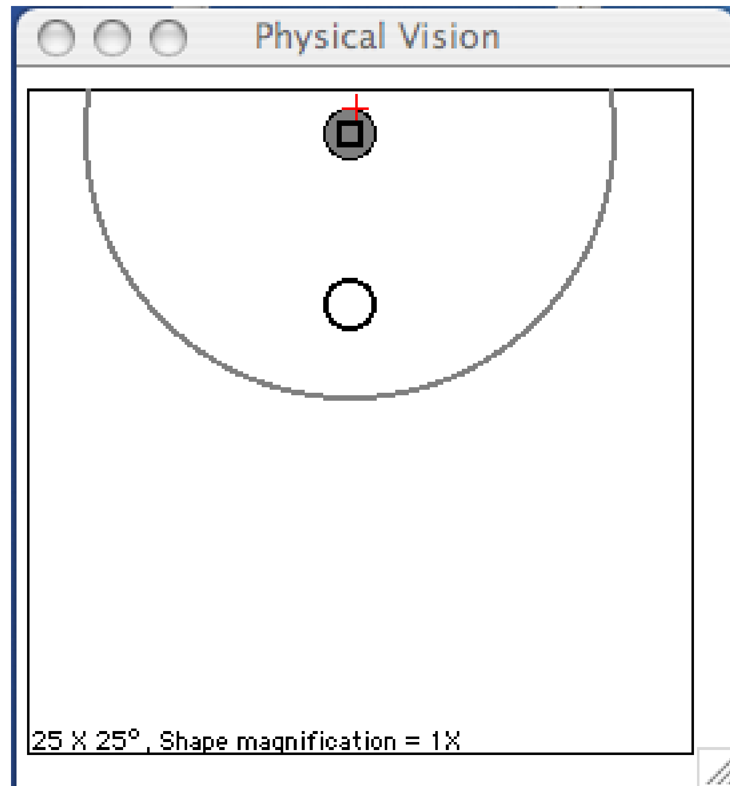


# Choice Trial Ends

**Choice stimulus disappears after response, RT recorded.**

**Stop accumulating RMS error after total of 2 seconds.**

**Tracking should continue.**



# Models for Exercise 3

**Model Components for Wickens Task**

**Fully Sequential Specialized Executive**

**Parallel General Executive**

# Model Components for Wickens Task

**A set of Startup rules for Goal: Identify Objects:**
Simply identifies and tags the different screen objects for later reference.

**A set of rules for tracking:**
Governed by (Goal Do Tracking)
One thread ensures that eye is kept on tracking target
Another thread does the tracking.
Task is inherently continuous; rules could be constantly enabled.

**A set of rules for choice reaction:**
Governed by (Goal Do Choice)
Move the eye to the choice stimulus location.
Select the response depending on shape property.
Task is intermittent, but is also highest priority.

**A set of executive rules for task coordination:**
In general form, contain no information about specific tasks.
In specialized form, depend on choice task stimulus onset as signal that choice task should be started.
Possibilities explored in the exercise:

- Single task execution.
- Fully Sequential Specialized Executive.
- Parallel General Executive for Resource Allocation.

# Fully Sequential Specialized Executive

**Run the Identify Objects startup rules.**

**1. Start the tracking task by inserting its governing goal.**
Tracking task starts: moves the eye and begins manual Ply actions.

**2. Wait for the onset of a new visual object, and tag as choice stimulus.**
Executive is specialized because it knows that a new object must be the choice stimulus.

**3. Stop the tracking task by removing its goal; start the choice task by inserting its goal.**
Choice task starts; moves the eye, selects manual Punch response, terminates by removing own goal.
Choice task rules all wait for movements to be complete:

- Before commanding a new movement.
- Before going to next step.

**4. When choice task is done, resuming tracking by going back to 1.**

# Parallel General Executive

**Run the Identify Objects startup rules.**

**Start both the Tracking task and the Choice task by inserting their governing goals; make Choice higher priority than Tracking.**

**Tracking requests use of Manual and Ocular processors, and starts running, checking for permissions to use them in each thread:**
If has Ocular, move eye to tracking target if needed.
If has Manual, make a Ply movement if needed.

**Choice task starts running:**

1. Wait for a new object to appear.
2. When new object appears, tag as stimulus, request use of Ocular.
   - Request preempts Tracking use of Ocular.
3. When Choice has Ocular, move eye to choice stimulus, release Ocular.
   - Ocular reallocated to Tracking.
4. When shape is available, request use of Manual.
   - Request pre-empts Tracking's use of Manual.
5. When choice has Manual, command response, release Manual.
   - Manual reallocated to Tracking.
6. Choice task returns to waiting for a new stimulus by going back to 1.

**Result: Task resource usage "automatically" interleaved!**

# Exercise 3: Exploring Multi-task Executives

**Follow the instructions on the handout. Steps:**

1. **Running the Single-Task  Models**
2. **Run the Fully Sequential Specialized Executive**
3. **Speeding up Sequential Model (Optional)**
4. **Try the Parallel General Executive**

**Discussion**

# Overview of Device Programming

Why Does the Device Need to be Programmed?

The Device and its Environment

Overview: How to Write a Device Model

Changing the Display - 1

Changing the Display - 2

Responding to Human Movements

Producing Time Delays

Taking Advantage of C++

A Tour of a Sample Device

# Why Does the Device Need to be Programmed?

**Need both a simulated human and a simulated environment.**

**Why not connect a simulated human to an existing piece of software?**

Connecting to existing software is technically very difficult, not general.

Very useful, but software may not exist for situation of interest.

- Considerable valuable data in the older literature!
- Often need to simulated a proposed rather than existing task situation.
  - E.g. use of cognitive modeling in system design in HCI.

In fact, simulated environment can be much simpler than the actual one.

- Simulated device can often be abstract and a subset.

**Why not a high-level specialized language for modeling experiments?**

Not everything of interest is a simple lab experiment!

Modeling wide range of tasks requires a powerful programming language.

- E.g. includes full range of trig functions.

A specialized powerful language is difficult to develop, maintain, and learn.

Why not just use an existing general-purpose well-established language?

**C++ is well-established, widely taught, suitable for complex systems, and can be written to be very fast and relatively easy to work with.**

But it is an "industrial strength" language, and so takes some effort.

Experience shows that given a "starter" device, people can get started without becoming C++ experts beforehand.

# The Device and its Environment

**Device (simulated environment) module participates in EPIC system:**
Device is programmed as another processor running in simulated parallel with the simulated human's processors.
So have to develop device in the context of the EPIC software.

**Need an appropriate C++ programming environment. Currently used:**
Macintosh OS X: CodeWarrior.
Windows: Microsoft Visual Studio C++.

**Overall software structure: Division between functional core and GUI .**
Macintosh and Windows versions differ only in the GUI code (almost).
Device programming is almost completely platform independent.
- Device is an extension of the functional core only.

**Current properties of Macintosh and Windows versions:**
Macintosh:
- Still in transition to full native Mac OS X facilities.
- Relatively easy environment to work in.

Windows:
- Hampered by complex MSVS environment and substandard C++.
- Has full "plug-in" architecture (using DLL's).

# Overview: How to Write a Device Model

**Install a copy of the EPIC Library suitable for your programming environment.**

Some details depend on the environment.

**Choose a similar existing device to use as a "starter."**

**Define a class that inherits from Device_processor.**

Provides the standard interface between the EPIC environment and a specific device.

**Override the virtual functions for handling the device input events you are interested in.**

**Handle the events by changing the display and recording information as required.**

**Overall strategy:**

The device only needs to produce behavior relevant to production rules.
Define the device as an event-driven state machine.

# Changing the Display - 1

**Call member functions for the Human_processor class to put visual objects and changes into the human's visual field.**

Specify locations, sizes in terms of visual angle.

Device specifies the properties that production rules will respond to .

**Symbol class used to specify names, properties, and values - holds one of:**

character string

single floating point number

two floating point numbers

**Device uses a unique physical name for each presented object.**

Using concatenate_to_Symbol function to “build” names.

Modify objects by referring to physical name.

**Use Geometry_Utilities classes for locations and sizes, all specified in terms of visual angle.**

Locations: use Point, an (x, y) pair where (0, 0) is center of visual field.

- e.g. (0, 0) is the center of the screen.

Sizes: use Size, a [horizontal size, vertical size] that corresponds to a rectangular bounding box for the object.

May have to compute visual angle values from screen layout pixel coordinates.

# Changing the Display - 2

**Make an object appear - specify location and size**

make_visual_object_appear(object_physical_name, object_center_location, object_size);

**Change an object's location or size:**

set_visual_object_location(object_physical_name, new_object_location);
set_visual_object_size(object_physical_name, new_object_size);

**Set or change other property of an object:**

set_visual_object_property(object_physical_name, property_name, property_value);

**Make an object disappear:**

make_visual_object_disappear(object_physical_name);

**Automatically generated displays:**

For a standard set of visual properties and their values, the EPICApp displays will automatically show the current state of the physical environment and the sensory and perceptual stores.

- E.g.: Shape Filled_Circle, Color Teal

Displays are relatively crude, but hard to argue with getting them for free.
Could be customized if necessary.

# Responding to Human Movements

**When simulated human makes a movement, the corresponding event will be sent to the device for handling.**

Prototypical event handling function:
void handle_event(Event_type event_ptr)

**Event arrives at the time the movement is made.**

The “time” field of the event contains the arrival time.
- Will always be the same as the current time.

Can also get the current time with get_time().

**Keystroke events - generated by Punch movement**

event_ptr->key_name: the name of key or button

**Cursor movement events - generated by Ply movement**

event_ptr->new_location: the new location of the cursor
event_ptr->cursor_name: the physical name of the cursor object
event_ptr->target_name: the physical name of the target object
- A series of movements is made, each to an intermediate position - allows cursor movement to be "animated."
- target_name is “nil” for intermediate cursor positions.
- Actual target name when the cursor arrives on the target.

# Producing Time Delays

**Use a delay function in Device_processor that creates a Device_delay_event.**

do_device_delay_event(milliseconds);
do_device_delay_event(delay, delay_type, delay_datum);
etc.

**In the event handler, branch on the current state or data in the event.**

handle_event(Smart_Pointer<const Device_Delay_event>)

# Taking Advantage of C++

**Define simple classes to organize code for stimulus objects, etc.**

Get some benefit of object-oriented programming paradigm.

**The C++ Standard Library includes containers and algorithms that can greatly simplify writing complex code.**

E.g. use *vector* class for resizeable arrays, *map* class for look-up jobs, *random_shuffle* algorithm to produce randomized stimuli, *sort* algorithm to put things in order.

**Since C++ compiled code is generally very efficient, can emphasize code clarity and organization instead of worrying about low-level efficiency.**

If you need to compute it - go ahead - it will probably be fast enough!

# A Tour of a Sample Device

**An excerpt of the Wickens task device.**

**Handout contains:**

Device_processor.h
Sample_device_code.h
Sample_device_code.cpp

## Wrap-up

General discussion of lessons learned and unresolved problems.

- Suggestions for future enhancements and improvements?

Future directions for EPIC

- Further perceptual-motor refinements.
- Issues in modeling of learning.

Future cooperation and code-sharing.