VIRTUAL WORLD DEVELOPMENT

Course Description:

The course will address the conceptual architecture of virtual worlds, including software implementation, physiological and cognitive constraints, design of experience, and the mathematics and philosophy of inclusion. Topics include the development of software tools, editing and interaction techniques, the disposition of virtual world entities, the nature of space, situated knowledge, divergent models for multiple participants, experiential mathematics, cyberspace, and cultural, legal, moral and ethical issues.

During the Laboratory section, each student will design and construct a virtual world using CAD construction kits (AutoCAD, Wavefront, Alias, Swivel3D), dynamic simulation packages (Body Electric, VEOS), behavior transducers (position trackers, gloves, spaceballs, other interface devices), and computational resources (SGI 4D/320, DEC 5000/200, Sun 4).

SITUATED SYLLABUS

VR emphasizes the relationship between participant and environment. Theories which recognize that content and context are inextricably interrelated (intertwingled) are called *situated*. In AI, it's called situated automata; in industrial engineering, it's ecological psychology; in mathematics, it's general systems; in psychology, its Gestalt.

In traditional classes, what you are expected to learn is defined ahead of time. In this class, we will respond to events dynamically. This is called *situated learning*. The theory is that the dynamic context impacts what and how we learn. As a consequence, the syllabus will change dynamically over time, in response to class needs and evolving understandings.

In VR, you enter first into a void; you then load a constructed database that is the world. In physical reality, you are born into a world that is already full. The difference is that VR demands active participation, inside and out. VR emphasizes *constructivism*, the theory that mind and body coparticipate to construct our experience of reality. In education, constructivism says that students build their own understanding through experience. This means that students must actively engage in and work with the subject matter, and that will be the general rule for this class. It also means that each student will construct an essentially different understanding of VR. In VR software terms, we all live in divergent realities.

Finally, the essence of VR is direct interaction with information. Right now, universities use a *symbolic mediation* strategy almost exclusively. To learn something, we translate it into what it is not (symbols), study the symbols (words, formulae, programs), then reconstruct the something. In this class, we will attempt to engage in direct learning by constantly anchoring the symbols we use with the reality they refer to.

Virtual World Development is a graduate seminar. Class grading will be based on individual understanding, participation, and growth. You will be required to attend class and to develop or improve a skill related to the design and construction of virtual worlds. Each student will probably develop a different skill, at a different rate, with different criteria of success.

Students who are concerned about this approach to teaching should meet with the instructor. An option of individual performance contracts is available.

POSSIBLE CONTENT (no particular order, not necessarily complete):

VR software architectures and functionalities

VR varieties and taxonomies

Theory of Inclusion

Systems oriented programming
- parallelism
- modularity
- partitioning

Situated agents (entities)
- reactivity and responsiveness
- dramatic theory
- embedded narrative
- dispositions
- autonomy

Building virtual worlds
- software tools
  - CAD
  - dynamics
  - animation
  - scientific visualization
- techniques
  - enumeration
  - decomposition
  - CSG
  - boundary models
  - sweeps
- design
  - participation who/what/why
  - physiological constraints
  - tight coupling

Display
- rendering choices
- adaptive refinement
- viewpoint control, navigation

Abstraction
- varieties of space
- networks
- form abstraction
- application specific construction kits
- semantics

Computation
- pattern matching
- constraints and possibility spaces
- inference
- history and statistics
- resource management
- editors
- molecular programming

Inclusive tools
- cursors
- backdrops and foredrops
- virtual body
- wand
- inhabitation
- artificial life

Multiple participation
- inconsistency maintenance
- uniqueness
- negotiation

Experiential mathematics
- logic blocks
- boundary mathematics
- spatial algebra

Teleoperation and telepresence
- presence
- out-of-body experience
- physical and sensory extension

Physiological modeling
- sensory models
- physical constraints
- plasticity

Cognitive modeling
- information processing
- gestalt
- situated intelligence

PROJECTS

Each student is expected to contribute to our state of knowledge about VR. Projects document this contribution. Project work will be presented to the class, so that the class can review the work. This list is suggestive:

Virtual World Development

Design and/or build a VR tool
- wand
- physiological model
- virtual body
- virtual community
- editor for entities
- form abstraction
- logic blocks
- mapping tools
- projection tools
- navigation tools
- divergent worlds
- conversational programming
- music

Research and design a VR language
- behavioral modeling
- construction from inside VR
- algebraic specifications
- gesture languages
- virtual machines
- sound

Write and publish a VR article
- VEOS
- entities
- social implications

Develop VR operating system tools
- FERN
- Linda
- device drivers
- graphics drivers
- world building maintenance tools
- UM

Explore an important VR issue
- access
- cognitive plasticity
- ecstasy machines
- cultural bias
- ownership and legalities
- philosophy and metaphysics

TEXTS

No text is required for this class. Readings and references will be provided by the instructor as appropriate. Some good general (popular) books on VR include:

Aukstakalnis, S., & Blatner, D. (1992). Silicon Mirage: The Art & Science of Virtual Reality. Berkeley, CA: Peachpit Press.

Rheingold, H. (1992). Virtual Reality: The Revolutionary Technology of Computer-Generated Artificial Worlds - & How It Promises to Transform Society. New York, NY: Simon & Schuster Trade.

Ellis, S.R. (ed.). (1991).Pictorial Communication in Virtual and Real Environments. London: Taylor & Francis.

Benedikt, M.L.(ed.). (1991). Cyberspace: First Steps. Cambridge, MA: MIT Press.

Woolley, B. (1992). Virtual Worlds: A Journey in Hype and Hyperreality. Oxford, UK: Blackwell Publishers.

Presence: Teleoperators and Virtual Environments. Published quarterly by the MIT Press, Cambridge, MA 02142. Subscription requests should be addressed to MIT Press Journals, 55 Hayward Street, Cambridge, MA 02142. Telephone: (617) 253-2889. ISSN 1054-7460.

CLASS PROJECT

The class project is to build a collection of tools (disembodied machines) for a virtual environment. The tools will be expressed in the following language.

LANGUAGE

Logic:

```
constants =  { true   false }
operators =  { if-then-else   and   or   not   equivalent }
```

Rational numbers:

```
constants = { 0  1  2  ... }
operators = { +  -  *  /  ^  mod }
relations = { = > < }
```

Function theory:

```
base case   =  function(ground) = whatever
recur case  =  function(variable)
                  = whatever and function(smaller-variable)
```

Data structures:

```
Set  { a  b  ... }
List [ a  b  ... ]
Stream [ a, b, ... ]
```

Database theory:

```
Get(pattern)
Put(pattern)
Copy(pattern)
```

Control theory:

sequential unary apply

```
f({ a b ... })  =   { f(a) f({ b ... }) }
```

sequential binary apply

```
f({ a b c d ... })  =  f({ f(a b) f({ c d ... }) })
```

parallel unary apply

```
f({ a b ... })  =   { f(a) f(b) ... }
```

parallel binary apply

```
f({ a b c d ... })  =  f({ f(a b) f(c d) ... })
```

parallel nary apply

```
f({ a b ... })  =  f(a b ...)
```

Some Examples

Factorial:

```
fac(1) = 1
fac(n_) = (n * fac(n - 1))                              sequential
fac(n_) = *{ 1 .. n }                                   parallel
```

Move an entity:

```
Move(0) = entity-position
Move(1) = entity-position + 1
Move(n_) = Move(1) and Move(n - 1)                      sequential
Jack(place_) =   (entity-position = place-position)
```

JUST WHAT IS VIRTUAL REALITY ANYWAY?

What are the defining characteristics of virtual reality, of a virtual reality system? Suggest techniques for "measuring" each characteristic.

Develop a taxonomy (hierarchy, state space) of partial and complete VR systems.

- computer-generated, television or live image?
- inclusion or partial immersion or watching 3D?
- wearing a computer?
- how many dimensions? multisensory?
- input and output bandwidth?
- available system resources?
- responsiveness and timeliness?
- physical/virtual mixture?
- degree of presence and physical remoteness?
- occlusive, overlay or annotated?

Consider (some of) these issues:

- bandwidth
- sensory modality
- degree of coupling and feedback
- input and sensor types
- output and display types
- interactivity
- realtime responsiveness
- meaning
- human physiology
- presence
- telepresence
- dimensionality
- realism
- autonomy
- formality
- anthropomorphism

Virtual World Development

Classify (some of) the following systems in your taxonomy:

- Dataglove
- Heads-up display
- Inclusive display
- flight simulator
- computer animation
- stereo sound
- Nintendo games
- SIMNET
- computer aided design (CAD)
- Landsat database
- remote controlled robot
- voice recognition
- holographs
- email
- command line computer interfaces
- desktop metaphor (WIMP) computer interfaces
- electron microscope
- the physical world
- television
- telephone
- automobiles
- drawings
- sculpture
- thinking
- meditation
- dreaming
- books and reading
- photographs
- movies
- Disneyland
- this assignment
- [add your own]

VRCHITECTURE

Design an architecture for a virtual reality system. Identify essential components with boxes, and essential communication channels with lines connecting boxes.

The components of your architecture should all be of the same type. For example, a hardware architecture identifies all the hardware components, a software architecture identifies all the software modules, a functional architecture identifies the essential transformations.

Clearly identify the purpose of each component. Clearly identify what is being communicated over connections.}

DESIGN A SOFTWARE TOOLKIT

The class is to specify the software tools and interface techniques for a Virtual Reality suite.

Do not use English for the specification. Use command words, or a formal language, or functions, or predicate calculus, or diagrams, or animations, or a programming language, or any form that you can be explicit about what a tool does or how it works, but do not use English, use a specification language.

Follow this ***organizational structure***:

Form eleven groups of two members. Each group will be responsible for a particular tool in the suite. No group may duplicate another group's functionality.

Include at least these six tools:

The Wand

The Virtual Body, sub-components:
- visual sensors
- audio sensors
- position sensors

The Physiological Model, sub-components:
- body position model
- voice recognition

The remaining five groups may specify a tool of their choice.

Each group must coordinate its specification language and protocol with other groups (tools) that use them. For example, the Wand may depend on the position of the virtual body.

Super-observers must report the progress of class activity.

This is a two hour exercise, I will collect a two page design specification from each tool/group and a two page design integration from the class as a whole.

Grades for the class will be based on functionality, integration, and clarity of specification.

A SPECIFICATION LANGUAGE FOR VIRTUAL WORLDS

The basic facility:

creation of labels for elementary objects in the world
creation of terms expressing complex structures between relations
creation of equations which permit algebraic operations
on labels and terms

The set of unique labels defines the elements of the Domain. They map onto objects in the semantic model.

The set of permissible terms defines the structures of the Domain. Structures require operators for Construction and for Access/Deconstruction.

The set of specified equations defines the virtual world, and provides substitution as the primary computational mechanism.

Definitions

labels:	Domain-labels = {a b c d ...}
terms:	
pattern sets:	set-label = { members-of-set } or Pattern___
procedural logic:	doIF Term doTHEN Term = boolean-term
build-your-own:	function(set-argument) = definitional-term
	relation(matrix-argument) = boolean-term
equations:	Term1 = Term2
worlds:	world-label = { set-of-equations }

Pattern Language

name_	matches one item in a set
name__	matches one or more items
name___	matches one or more or no items

Database Manipulation

```
Get(pattern_ from set_)
Put(term into set)
Copy(pattern_ from set_)
```

Mechanism

Available process threads are assigned to equations.

Equations are expanded via match-and-substitute until no more matches.

Labels which are not recognized by the pattern matcher are "literal".

Input devices put values into an associated equation.

Display devices get values from an associated equation.

EXPANDIBLE VIRTUAL CUBE WORLD
DESIGN/DEVELOPMENT/SPECIFICATION ASSIGNMENT

Using the VR specification language, do as many of the following tasks as you can. Those working in groups should attempt more.

1. Specify the geometry of a cube.

2. Specify some properties of a cube. Choose properties that permit some specific cube functionality.

3. Specify some transformations on a cube.

4. Specify an environmental cube and a contained object cube.

5. Specify some form of interaction with a cube, using a defined device such as the glove, the wand, or the spaceball.

6. Add some more cubes and specify some ways in which they relate.

7. Specify some multisensory viewpoints on a cube.

8. Specify a disposition of a cube. Choose behaviors that permit some specific cube goals.

Combine the above specifications to build a world:

9. Block and Wand: A wand (or a spaceball, or ...) is used to manipulate a block.

10. Blocks World: pick up blocks and build structures with them.

11. Logic blocks: block structures map onto propositional calculus and prove theorems.

12. Block structure builder: name a particular configuration of blocks, the existing configuration will rearrange itself to form the target configuration.

13. Block structure builder + restructuring baby: Blocks will take steps to rearrange into a particular configuration while a baby dynamically changes the existing configuration.

14. Block obstacles: Move a virtual body through a maze of blocks.

15. Topple blocks: Remove blocks from a block structure until it falls down.

16. Architectural blocks: Configurations of blocks represent architectural spaces. Write design constraints for a building or a community.

17. Creative blocks: make up your own block world interactions.

THE STRUCTURE OF A CUBE

The *key idea* is that the structure (geometry) of an object is an intrinsic property. Structure should make no reference to external relations.

Note that translation, rotation, scale, and orientation are Relations between an object and an external coordinate system, and are thus not part of a cube's geometry.

Fortunately, there are established conceptual tools (Cartesian geometry, unit vectors) for describing "cube space".

EMBED THE CUBE IN A SPACE

Assume unit vectors i, j, and k. Associate each with an orthogonal side of the Cube.

Given rules for ijk: i*j = i*k = j*k = 0

Assume a local origin (0i 0j 0k).

i = (1i 0j 0k)
j = (0i 1j 0k)
k = (0i 0j 1k)

DIFFERENTIATE PARTS

Cubes have 27 parts: 8 vertices, 12 edges, 6 faces, 1 volume.

Notation: (ai bj ck) for all parts.

Let {a, b, c} take on three possible states: {0, _, 1},

where _ is any value 0 =< _ =< 1

Let d = {0, 1} (Knonecker delta, either 0 or 1)

Vertices:	{di dj dk}
Edges:	{di dj _k} or {di _j dk} or {_i dj dk}
Faces:	{di _j _k} or {_i dj _k} or {_i _j dk}
Solid:	{_i _j _k}

More notation:

Let i, j, and k be symmetrically equivalent, and thus unlabeled.

Vertices:	{d d d}	(all three states are Kronecker)
Edges:	{d d _}	(one state is not Kronecker)
Faces:	{d _ _}	(only one state is Kronecker)
Solid:	{_ _ _}	(no state is Kronecker)

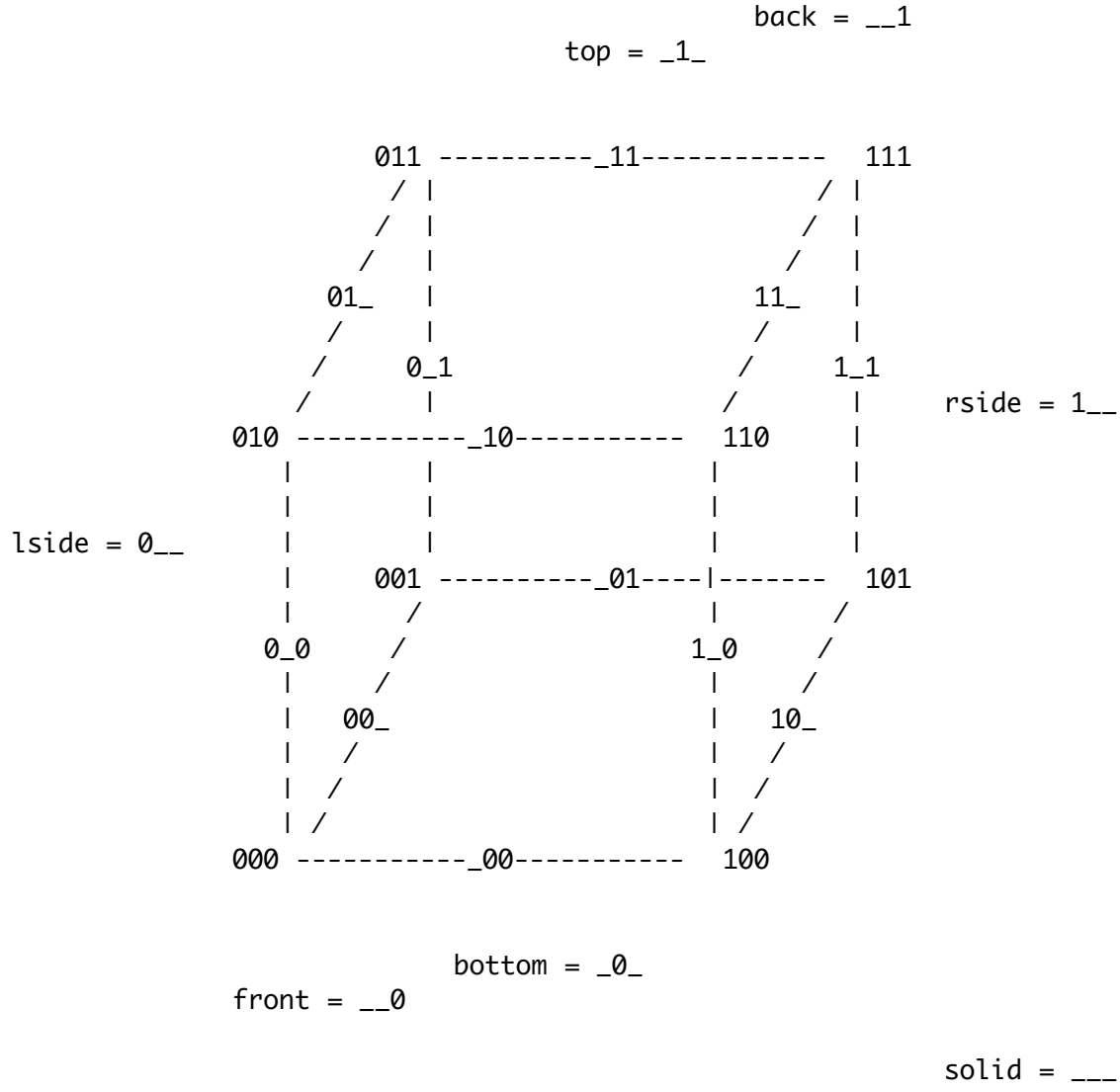
Let u stand for any of i, j, or k.

PROPERTIES

```
Parallel(e1_ e2_) =    e1{d d _} = e2{d d _}            _ in same location
Parallel(f1_ f2_) =    f1{d _ _} = f2{d _ _}            _ _ in same location


Perpendicular(e1_ e2_) =    not(Parallel(e1 e2))
Perpendicular(f1_ f2_) =    not(Parallel(f1 f2))


On(v_ e_) =    v{du} = e{du}                            values of d equal
On(v_ f_) =    v{du} = f{du}                            value of d equal
On(e_ f_) =    e{du} = f{du}                            value of d equal


Meets(e1_ e2_) =     e1{du} = e2{du}                    some d equal
Meets(f1_ f2_) =     not(Parallel(f1 f2))


Distance(v1_ v2_) =    number of different {du}
Distance(e1_ e2_) =    number of different {du}
```

PICTORIALLY

```
                    011  _11  111
                    0_1  __1  1_1
                    001  _01  101

               01_  _1_  11_
               0__  ___  1__
               00_  _0_  10_

          010  _10  110
          0_0  __0  1_0
          000  _00  100
```



MULTIPLICATION TABLES

To determine vertex of intersection of two edges (or edge of intersection of two faces, or more generally, lower dimensional element defined by two other elements), down-multiply representation:

*	0	_	1
0	0	0	_
_	0	_	1
1	_	1	1

To determine edge formed by two vertices (or general up element), up-multiply representations:

%	0	_	1
0	0	_	_
_	_	_	_
1	_	_	1

Note than non-intersecting vertices identify faces (or solids)

NOTES ON REPRESENTATION

By multiplying i, j, or k by a scalar, the cube generalizes to an arbitrary block.

ijk provides lots of established mathematical support.

{0 _ 1} provides unification of different parts of a cube and visual imagery.

Binary Kronecker delta provides easy implementation, but could be renamed (0 = low, 1 = high, _ = any) for understanding.

Properties are trivial calculations.

Generality of notation is difficult to express algebraically. In general, the more abstract, the more powerful and the harder to express.

ALGEBRAIC SPECIFICATION LANGUAGE

The algebraic specification language (ASL) is intended to provide the formal structure needed for modular programming and the flexibility needed for unencumbered design.

The main advantage of ASL is that it makes it simple to understand both what the machine is doing and what the designer is doing. Computation can be thought of as algebraic match-and-substitute. Specification can be thought of as construction of mathematical systems.

This language does not provide a visual metaphor or any intuitive user interface tools. These design tools can be built on top of the ASL, forming another layer which uses the functionality in the algebraic specification.

In Computer Science, this approach is called abstract data types. Specifying worlds can also be thought of as creating objects in object-oriented programming. In mathematics, using ASL is building a formal algebraic theory. In logic, it is Predicate Calculus with equality.

Reference: Bergstra, Heering and Klint (eds), Algebraic Specification, ACM Press: 1989.

Caution: This approach is under development, is likely to change, and can benefit from your critical analysis.

THE LANGUAGE OF ALGEBRAIC WORLDS

A ***world*** consists of a set of equations:

```
World  ::=  { equation ... }
```

An ***equation*** is an identity relation between two terms:

```
Equation  ::=  ( Term = Term )
```

A ***term*** is an expression which can be constructed from the labels and connectives in a language. A term is either a constant, a variable, a function-label, a relation-label, a set, or an equation:

```
Term ::= constant | variable | function | relation | set | equation
```

A ***constant*** is a label, labels are typographic strings:

```
Constant  ::=  label
```

A ***variable*** is a label with one or more underscores:

Variable ::= label_

A ***function-label*** is a labeled collection of variables:

Function ::= functionlabel[variables__]

A ***relation-label*** is also a labeled collection of variables:

Relation ::= relationlabel[variables__]

A ***set*** is a collection of terms bounded by curly brackets.

Set ::= {term ...}

Using this language assures that the designed world can be programmed. The template of what belongs in a world description follows.

THE STRUCTURE OF AN ALGEBRAIC SPECIFICATION

The name of the particular world is declared to be equivalent to the set of equations which define that world.

WorldName = { world-equations ... }

Include the following types of equations in every World:

sets	(collections of elementary elements or things)
functions	(transformations of things)
relations	(Boolean assertions about things)
rules	(equations which define the behavior of a world)

* Set names identify a collection of ground constants, the elementary objects in the world. Each label denotes a thing in the designed world.

(SETNAME = { label1 ... })

* Functions transform things and the characteristics of things. How a function changes a constant is specified by a base case equation.

(functionlabel[constant] = term-which-reduces-to-a-constant)

The general case of an function specifies how that function changes collections of things.

(functionlabel[variablepatterns__] = term)

Functions can be recursive. Constants are functions with no arguments.

* Relations establish associations between collections of things.

(relationlabel[variablepatterns__] = boolean-term)

* Equations express an equivalence between terms (structures in the world). Equations define the axioms, properties, or rules for the world; they specify how the world works.

(term-with-variables = term-without-variables)

SOME TOOLS

The ASL is easier to use when some tools are included.

Mathematical Operators:

Logic, arithmetic, and set theory can be assumed to be available. The logical connectives {not or and if} permit complex Boolean relations and permit control structure to be embedded in equations. Arithmetic {+ - * % ^ < >} provides counting and numerical calculation. Sets can be composed and decomposed using set operators { union intersection complement }.

Generators:

The ellipsis ... can be thought of as a unique label generator. Rather than naming all the blocks for example, BLOCK = { block1 block2 block3 }, you can name the pattern for block names, BLOCK = { block1 ...}. The ellipsis can be constrained; BLOCK = { block1 ... block10 } will generate ten block names.

Indices:

It is often useful to index items in a set. The dot notation indicates parts of a complex term:

element1.part3	identifies part3 of element1.

An index notation can be used to identify a specific arbitrary element:

element:i	identifies an arbitrary single element i selected from the set of elements, without replacement.

Indices can also identify one element selected from a set:

element:1	selects one element, with replacement.

Set functions:

The ASL permits functions to operate on sets. The pattern variable with a double underbar, label__, specifies one or more members of a set which match the pattern. A set calculus allows sets to be treated as variables: CAPITALX = {a ...} labels the set itself, smallx = {a ...} labels the collection of elements.

SET1 = SET2	declares set equality
set1 = {a b c}	associates the label with an arbitrary element

Operations on capital-letter sets can be assumed to apply over the entire set.

SET1 + 5	adds 5 to each element of set1
+[SET1]	adds the elements of SET1

Typing:

The SETLABEL can be thought of as the type of the elements in a set. The type of constants and variables can be indicated after the label:

x_BOOL	x belongs to the set {true false}
b.BLOCK	b labels a block

Natural notation:

The pattern language lets us determine the form of expressions:

```
( block1 Is On block2 ) = On[block1 block2]
( if (p = true) then (q = true) ) = (If[p, q ] = true)
```

Sequences:

The elements of a set are separated by spaces, and in the case of ambiguity, by parentheses. The elements of a sequence are separated by commas.

```
{a b c}              a set
{a, b, c}            a sequence
and[ a b c ]         a set of arguments
if[ a, b ]           a sequence of arguments
```

Delimiters:

```
parentheses     (a=b)       for clarity
brackets        fn[a b]     for arguments
curly braces      {a b}       for sets
```

Using other worlds:

The USE command takes world labels as arguments and automatically includes the equations which define that world. USE is declared equal to a set of labels which identify the used objects.

```
USE[ UnitVector ] = {i j}             uses two unit vectors
```

ASL EXAMPLES, LOGIC

```
BooleanLogic = {

      BOOL = { true false }                        ;domain
      not[false] = true                            ;constructors
      not[true] = false
      not[ not[ p_ ] ] = p                         ;reduction patterns
      or[ p_ ] = p
      or[ true p__ ] = true
      or[ false p__ ] = or[ p ]
      and[ p__ ] = not[ or[ Map[ not, p ] ] ]      ;definition
      ( if p_ then q_ else r_ ) = if[p, q, r]      ;change syntax
      if[ p_, q_, r_ ] = and[ or[ not[p] q ] or[ p r ] ]
      }


Equality = {

      EQ = { x_ = y_ }
      }


BoundaryLogic = {

      BL = { <> }                                  ;domain
      call[ a___ b___ ] =  a b                     ;constructors
      cross[ a___ ] = < a >
      <> a___  = <>                                ;reduction rules
      << a___ >> = a
      < a__ b___ > a__ = < b > a
      }


ParseBOOLtoBL = {

      USE[ BOOL ] = { p }
      USE[ BL ] = { <> }
      true = <>
      false = <<>>
      not[ p_ ] = <p>
      or[ p__ ] = p
      }
```

ALGEBRAIC SPECIFICATION EXAMPLES, CUBE

```
GenericCube = {

      ;the structure of the SPACE embodying cubeness

      USE[ UnitVector ] = { i j k }
      V = { 0 - 1 }
      D = { 0 1 }
      < vi_V, vj_V, vk_V > = [vi, vj, vk] * [i, j, k]T
      < di_D, dj_D, dk_D > = [di, dj, dk] * [i, j, k]T

      origin = < 0, 0, 0 >
      center = < .5, .5, .5 >

      ;the PARTS of a cube, the DOMAIN of elementary elements

      PART = { < vi_, vj_, vk_> }
      VIRTEX = { < di_, dj_, dk_ > }
      EDGE = { <di_, dj_, ->  <di_, -, dk_>  <-, dj_, dk_> }
      FACE = { <di_, -, ->  <-, dj_, ->  <-, -, dk_> }
      SELF = { <-, -, -> }

      ;the operator which yields properties of the cube

      (p1_PART ^* p2_PART) = < ^*[p1.i p2.i] ^*[p1.j p2.j] ^*[p1.k p2.k] >

      ^*[ 0 0 ] = 0
      ^*[ 0 - ] = 0
      ^*[ 0 1 ] = -
      ^*[ 1 - ] = 1
      ^*[ 1 1 ] = 1
      ^*[ - - ] = -

      ;properties

      parallel[ p1_PART p2_PART ] = {

            p1_EDGE ^* p2_EDGE = _FACE
            p1_EDGE ^* p2_EDGE = _SOLID
            p1_EDGE ^* p2_FACE = _EDGE
            p1_EDGE ^* p2_FACE = _FACE
            p1_FACE ^* p2_FACE = _SOLID
            }
```

```
      perpendicular[ p1_PART p2_PART] = {

            p1_EDGE ^* p2_EDGE = _VIRTEX
            p1_EDGE ^* p2_FACE = _VIRTEX
            p1_FACE ^* p2_FACE = _EDGE
            }


      skew[ p1_PART p2_PART ] =

            p1_EDGE ^* p2_EDGE = _EDGE


      on[ p1_PART, p2_PART ] = {

            p1_VIRTEX ^* p2_VIRTEX = _VIRTEX
            p1_VIRTEX ^* p2_EDGE = _VIRTEX
            p1_VIRTEX ^* p2_FACE = _VIRTEX
            }


      connectedby[ p1_PART, p2_PART ] = {

            p1_VIRTEX ^* p2_VIRTEX = _EDGE
            p1_VIRTEX ^* p2_VIRTEX = _FACE
            p1_VIRTEX ^* p2_VIRTEX = _SOLID
            p1_VIRTEX ^* p2_EDGE = _EDGE
            p1_VIRTEX ^* p2_EDGE = _FACE
            p1_VIRTEX ^* p2_FACE = _FACE
              }

GenericBlock = {

      USE[ GenericCube ] = { cube }
      BLOCK = { [a1_ARITH, a2_ARITH, a3_ARITH] * [cube.i, cube.j, cube.k]T }
      IsCube[ b_BLOCK ] = ( b.a1 = b.a2 = b.a3 )
      }
```

```
CubesInaCube = {

      USE[ GenericBlock ] = { world b1 ... }
      worldscale = [1000, 1000, 1000]
      bigworld = worldscale * world
      location[ b_ ] = < bi, bj, bk >
      InWorld[b_] =
        (( <0,0,0> <= location[ b ] >= worldscale * <1,1,1> ) = true )
      location[b1] = <0,0,0>
      location[b2] = <1,0,0>
      }


StackOfBlocks = {

      USE[ GenericBlock ] = { world b1 ... }
      STACK = { [[ b1_ ... ]] }
      CONFIGURATION = { [[ b1___ ]]__ }

      emptytable = [[ ]]
      [[ ]] [[ ]] = [[ ]]
      location[ emptytable ] = < _, 0, _ >

      PutBlockOnStack[ b1_, s_CONFIGURATION ] =
        ( [[ b1 ]] [[ s ]] = [[ b1, s ]] )

      TakeBlockOffStack[ b1_, s_CONFIGURATION ] =
        ( [[ b1, s ]] = [[ b1 ]] [[ s ]] )

      On[ b1_, b2_ ] =
        ( [[ ___, b1, b2, ___ ]] = true )

      Above[ b1_, b2_ ] =
        ( [[ ___, b1, ___, b2, ___ ]] = true )

      OnTable[ b_ ] = ( [[ ___, b ]] = true )

      OnTopOfStack[ b_ ] = ( [[ b, ___ ]] = true )
```

FINAL EXAMINATION

Your assignment for today (1 hour):

=== DESIGN A VR CURRICULUM ===

Make an outline of the topics to be covered in your vision of an ideal VR curriculum. The curriculum can cover any VR related topics (for example: hardware, software, human factors, world design). Include all the topics you consider relevant for a full course in VR; do not limit your topics to those covered in this class. Indicate the importance of each topic, and provide sufficient detail so that an instructor can create clear lesson plans for each topic.

You can create a curriculum for a single one quarter course, for a series of courses over a year, or for a degree program. Include suggestions for teaching style, types of educational experience, and methods of evaluation.

This is a realtime individual exercise. If you can't think of something today, it probably isn't important.

=== TASK 2 ===

Give yourself a grade for this current class, between 2.0 and 4.0 (grainsize of measurement is .1). Justify your grade (one page maximum).

=== TASK 3 ===

Please fill out the course evaluation forms.

=== FINALLY ===

Class projects (design and/or build a VR tool) should turned in today.