interactive agent generated architecture

by

Jeffrey Charles Stanley Krause
Bachelor of Architecture
University of Southern California
1994

Submitted to the Department of Architecture in partial fulfillment of the requirement for the degree Master of Science in Architecture Studies at the Massachusetts Institute of Technology
June 1996

© Jeffrey Krause 1996. All rights reserved.
The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

Signature of the Author — Jeffrey Krause, Department of Architecture, May 10, 1996

Certified by — William J. Mitchell, Dean, School of Architecture and Planning, Professor of Architecture and Media Arts and Sciences, Thesis Supervisor

Accepted by — Roy Strickland, Associate Professor of Architecture, Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 19 1996
LIBRARIES
Rotch

Thesis readers

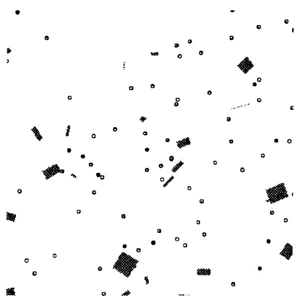
Takehiko Nagakura
Assistant Professor of School of Architecture and Planning

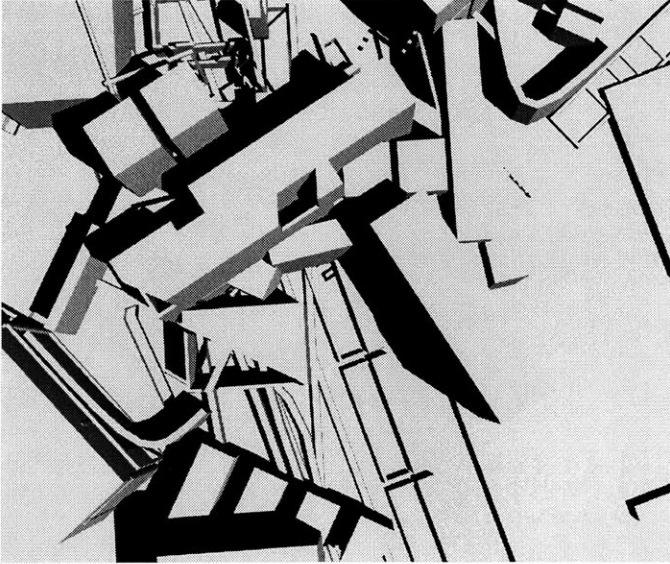
Support

Pattie Maes
Associate Professor of Media Arts and Sciences

William L. Porter
Norman B. and Muriel Professor of Architecture and Planning

John L. Williams
Associate Professor of Civil & Environmental Engineering





interactive agent generated architecture

by

Jeffrey Charles Stanley Krause
Bachelor of Architecture
University of Southern California
1994

Submitted to the Department of Architecture in partial fulfillment of the requirement for the degree Master of Science in Architecture Studies at the Massachusetts Institute of Technology
June 1996

abstract 00a

The thesis explores architectural form generation through two behavior based artificial intelligence approaches: the communication of agents in an unpredictable simulation system, and the codification of information within an evolutionary process. Both concepts stem from the evaluation of potentially definable mental constructions involving the process of translation and generation through base-level procedural methods. The experiments look towards the implementation of alternative computational processes regarding knowledge encapsulation, process recording, simulation environments, agent communication and interpretation from bottom-up design approaches. The experiment explores alternative approaches to design theory within the discipline of architectural computation.

Thesis supervisor: William J. Mitchell
Dean of the School of Architecture and Planning

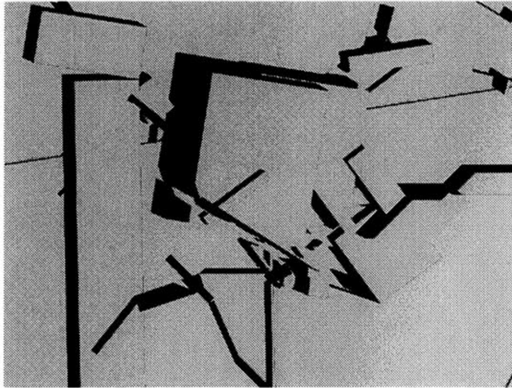


table of contents

abstract	00a
introduction	01a
kbai vs. bbai	01b
bbai & architecture	01c
on randomness and embedded knowledge	02a
generative paradox	02b
on emergence	02c
on modular component design	03a
on procedural thinking	03b
agent	04
agents in the environment	04a
movement	04b
behavior	04c
formal development	04d
communication	04e
structure of simulation	04f
simulation modules	04g
genetic	05a
on learning	05b
on from evolution	05c
on interpretation and conclusion	06a
definitions	06b
bibliography	06c
code	06d

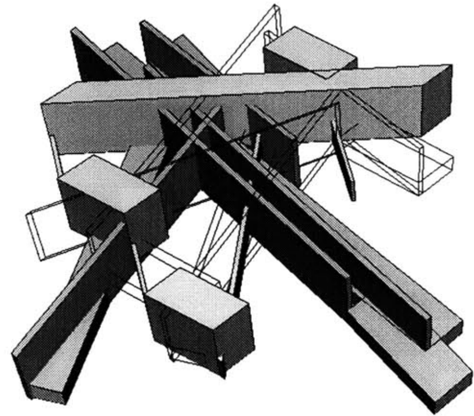
introduction 01[a]

The impetus of the thesis arises from the obscurity and the complexity surrounding the concepts of design and creativity. We have a difficult enough time defining these concepts let alone trying to explain our own idiosyncratic processes while inventing, designing, or creating. I attempt to take advantage of the ambiguity within this approach, to expound, explore, and manufacture through an implementation process without following conventional scientific investigation or research methodologies. Alternative to knowledge based examples the nature of the thesis is to operate in the immediacy: 'perpetuating the sketch', to meander through possibilities, exploring threads of thought which generate new ideas, and to work towards a better understanding of decentralized architectural processes which would be inconceivable through traditional computational methods.

Fundamentally, this work is a departure from traditional knowledge based artificial intelligence (KBAI) approaches which are structured in top down methods through knowledge based systems, heuristic programs, planning or reasoning systems. The starting point of the application is to think about emerging larger ideas through bottom up methods using very simple systems, agent behaviors, and communication in a large decentralized scale. In the agent model multiple agents communicate, transmitting information,

collaborating on the development of form and working autonomously to generate a collective object which is no longer the representation of the 'end-object', but an object which is a procedural scenario.

In the genetic model, information encapsulation in objects is mutable, transferable and inheritable. The genotypes which describe the formal makeup is judged, or in genetic terms tested for fitness. The most fit of the population in turn are mated for future generations.



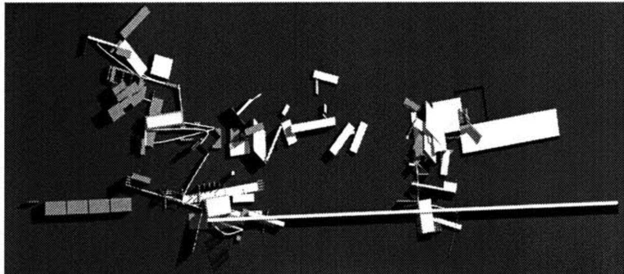
The thesis document is a compilation of theoretical and implementable concepts of bottom up approaches in computation. The rest of the document is committed to exploring theoretical background, a description of the experiments and the process of creating the applications.

what is behavior-based artificial intelligence and how does it differ from knowledge based artificial intelligence?
how can computational forms be synthesized and through what methods?

kbai vs. bbai 01b

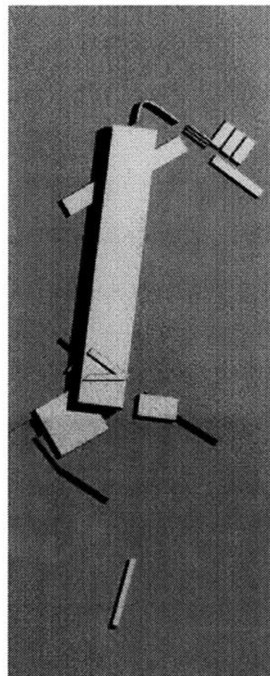
Knowledge based artificial intelligence(KBAI) most commonly refers to top down methods within problem approach and solving. KBAI systems usually model single competencies which deal with larger knowledge structures where

depth of investigation is more important than the breadth of the search. The systems tend to be closed and not autonomous. The emphasis of such methodologies is based upon embedding knowledge in to the system in a mostly static sense. The information within the system is then interpreted through reasoning and planning roles. There is no learning of the empirical knowledge within the system because this knowledge is built in from the start. KBAI concerns itself primarily with function oriented decomposition within domain independent modules. The internal models used within these methods are complete and need to be correct (relative) to the problems under investigation. The focus of the activities is problem solving through rational thought processes. For example trying to describe architectural components and their "fundamental" connections between one another: A building can be described as components, roof, body, and foundation. The body of the building is composed of rooms, walls, structural elements, and windows, etc. One problem with such a system is that when the understanding of the system transforms the model has to change. KBAI works through a process of knowing the problem and the solution and trying to decompose both to find the core procedural, or mappable method underlying the descriptive relationships between the two.



Behavior based artificial intelligence(BBAI) is a nouvelle approach for evolving multiple competencies at a very low level. The system is

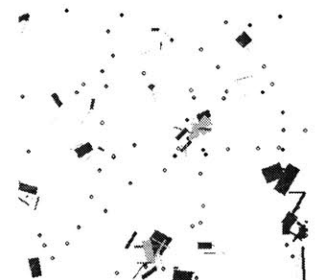
autonomous and often open. Agents within these types of systems find, evolve and derive their own goals where emphasis is placed less on the system and more on the behavior of the agents. The agent's interaction is active, involving more process like structures without general interpreters or reasoners. The agent's focus is on development, adaptation and interaction with the environment (space of occupation inclusive other agents). The environment which the agents inhabit is often unpredictable, complex or random. BBAI solutions usually include task-oriented decomposition with task dependent modules. The internal representations of problems, goals or actions within the agents are usually multiple, redundant, possibly inconsistent and nonobjective. BBAI, through bottom up approaches, tries to evolve the problem and solutions simultaneously. For a given situation multiple explorations, representations, and states exist even solutions which are potentially contradictory.

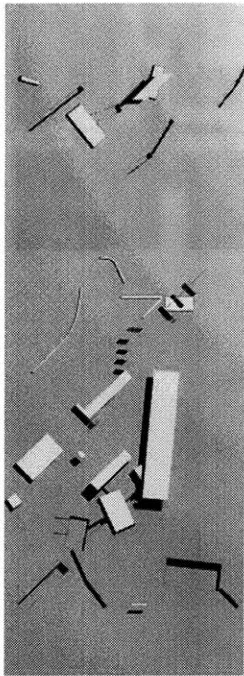


bbai & architecture 01c

How might architectural development be approached using BBAI systems?

Perhaps a key philosophical point to the argument would involve the interpretation and misinterpretation within the didactics of architecture discourse not only in relationship to design but

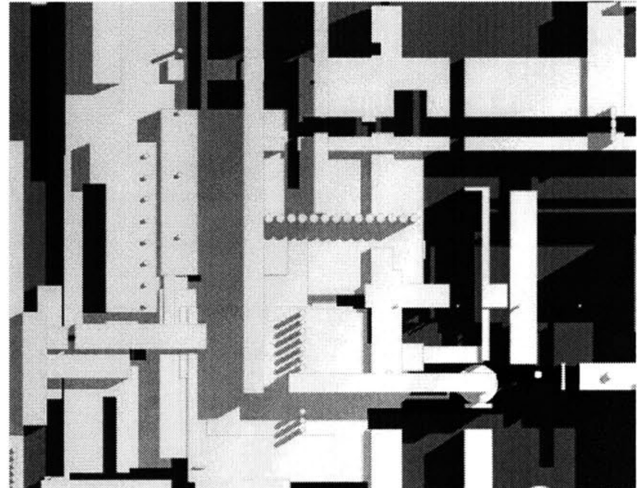




also judgment, intuition and experience. Within the question lies information regarding creative and flexible design processes and approaches which at any given moment are transformable, mutable and transient. One problem is with signifying systematic and objective definitions with formal consequences. The problem being the codification of these systems through static hierarchical formal models when they should be viewed as dynamic and changing systems. One focus of the investigation is to point towards imagining alternative forms or procedural generation and layout, one which at fundamental levels can change, allowing agents (as carriers of knowledge) to be responsive enough to an environment to adapt (with little effort) to situations and concerns without initiating monumental structural changes to a large program. Large hierarchical programs, built for solving single tasks, are often difficult to 'recode on the fly' and are (by definition) challenging to merge with other top-down or hierarchical models. The problem being flexibility in the base structures in regards to reactive situations to a changing or unpredictable environment. In KBAI systems if the environment changes then the program has to be recoded to respond to the changing parameters. Within emergent systems of BBAI we can view elements of the design process through these generative filters, focusing on emerging behavior rather than the codification of knowledge.

on randomness and embedded knowledge 02[a]

I pick up the pencil, rotate it slightly, draw a line lightly, short, a two second line. Lift the pencil, brush my hand over the paper. Push the pencil down, drawing, lightly, three seconds. Move the hand elsewhere. Draw multiple lines, quickly, lifting the pencil at each stroke.

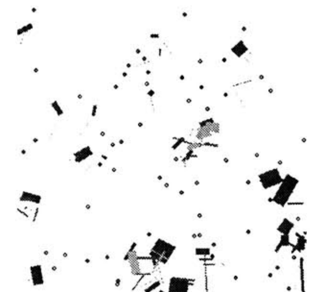


pline, copy, pline, offset, offset, trim, zoom, trim, regen, zoom, pline, pline, zoom.

agent 01 and 02 should be constructive. Agent 02 should be transformative. All agents should move randomly. Agents should emote behavior when their fields have been intersected.

All three examples describe procedures. They are algorithms which can be implemented and of course augmented and extended. They are drawing, or more importantly, generative techniques which have a considerable amount of information embedded intuitively behind the simple descriptive notion of the algorithm. The procedures of course are reusable, and practical to an extent. They are empty in regards to obvious interpretable solutions from the actions but extraordinarily wide open to the possibilities of formal derivations from the systems.

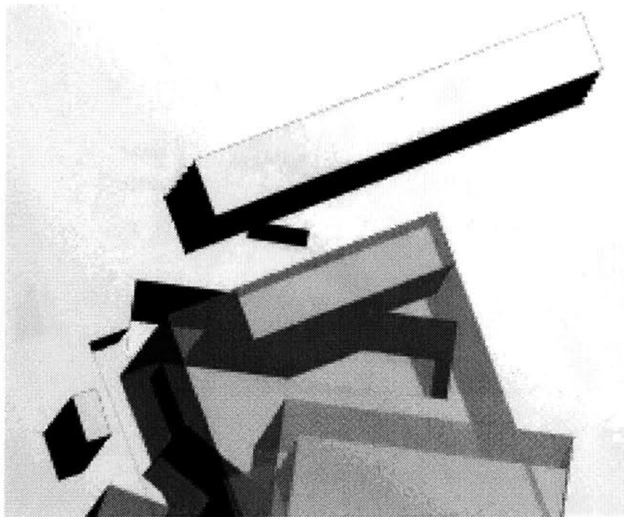
Each algorithm contains problematic structural problems which would need to be resolved if implemented computationally. One could focus



through the steps searching for missing information or expand the algorithm to encompass a wider and describe a larger system.

generative paradox 02[b]

what is the difference between a robust system which can generate everything, and a system of constraints which can generate a limited number of possibilities?
can a topic which represents the degradation and perpetuation of unpredictable structures augment a design process?

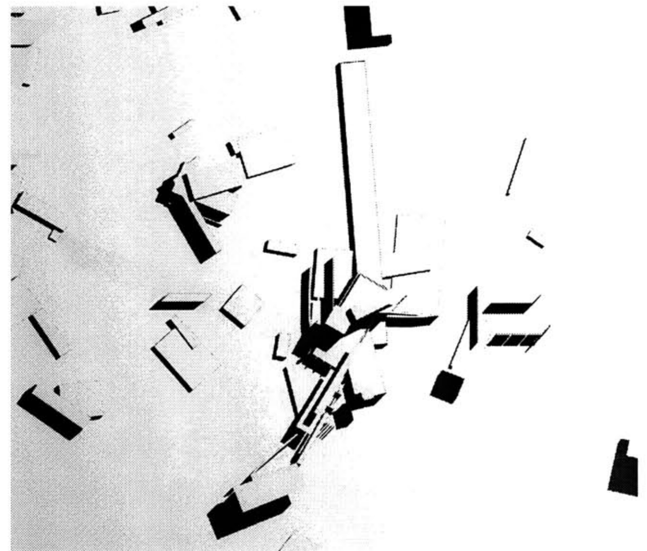


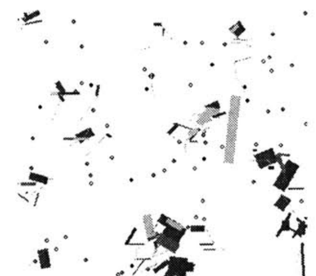
Randomness, within the context of design and thought processes, is usually dismissed as whimsical and nonsensical. The inherent trap of randomness is the idea of creating nothing from a set of nothing. The process of randomness becomes more compelling when applied to the genre of a procedurally described system. In this case the procedural system which defines the range of random interactions becomes the artifact to judge. This artifactual system can be described as narrow or inclusive as one wishes within a method of operations. The argument becomes multi-fold when applied to discipline analysis and interpretation, whereby robustness and consistency of the system can be viewed as 'tuning' randomness. One could also describe the operational field more systematically from the given set of parameters and actions, goals, sensors, or behaviors. These properties define a larger set of learning which takes place through

the description of the function: f->Sensors * Goals * Behaviors -> Action, which is updated over time to define a new function description: f_1-> A * G * B -> A_1.

how much information, meaning or sense does randomness allow?
how do we judge what information should be embedded?
how can we reflect on the actions generated from the system?

Claude Shannon, the founder of information theory wrestled with the problems of context, structure and meaning. Shannon was interested in the content and transmission, generation, and retrieval of information. Shannon looked to ideas of information degradation and generation through transformation processes, i.e. sending a television, or phone signal and having it transformed again through a perceivable medium. The paradox, known as Maxwell's demon derives through imagining an all inclusive generative system from a defined and describable system. Maxwell's demon can be applied to any discourse but is easily mappable to visualization, information storage and image manipulation. The representational example describes the generation of "all" images derived from a simple graphic imaging system. If a graphic image can represent a realistic image, it can represent all realistic images. To generate "all images" we implement a simple algorithm which toggles pixel colours within the image, either randomly or systematically. Within this example we will assume that the environment for the generation of "all





images" is 1280 pixels by 1024 pixels. This is a standard semi-high resolution image which could, for the most part, be an image (recognizable even) of anything. This image has 1,310,720 pixels within the frame. To make the problem easier consider only 1 million pixels. Each pixel within the field has to have a state, or mappable colour code, which describes its presence. We could use a larger amount of colours to describe the state but let us imagine that we only use black and white and eight shades of grey. Within this context the number of pictures which can be generated is $10^{1,000,000}$, or the states per pixel to the power of the number of pixels. 10^{10^6} is already an astronomically large number. 10^{100} is a googol, meaning 10 multiplied by itself 100 times. Our system is 10 multiplied by itself 1,000,000 times or 1 followed by a million zeroes. A googol is already a number larger enough to have no physical meaning, already trillions of times larger than the number of elementary particles in the observable universe. Conclusion: having a system to describe everything is not difficult, actualizing the potentials of the system is where complexity arises, filtering and judging become the difficult tasks. The problem with the above mentioned paradox and 'all-inclusive' systems is that they quickly transform in NP-hard problems. Of course from the example the majority of the generative images would be meaningless noise. The question arises on how to make the system generate a significant number of meaningful examples consistently without having to search through

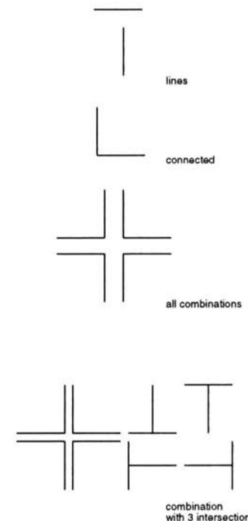
infinite sort space.

how can multiple agents collaborate
in robust and effective ways?
how can many simple agents
together do complex things?

on emergence 02c

The previous paradox raises more interesting issues in relationship to context and structure embedded within the potentials of random form or image generation. The interesting concern within a generative system becomes the complex relationships of the structure for generation and the potential outputs of a procedural system. The beauty of the system evolves through bounding the extents of predictability.

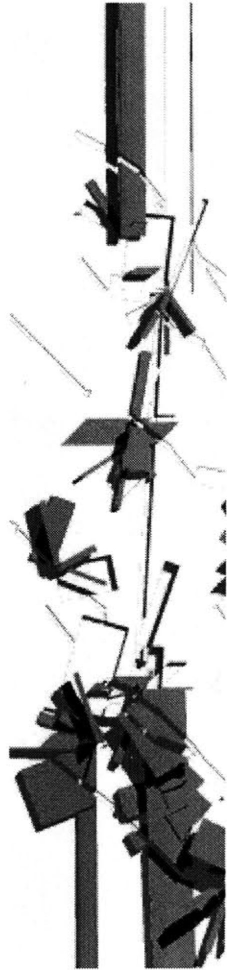
Let us look more closely at another system in which two perpendicular lines are being connected. Each line has two points, if they can only be connected at the end points the number of possibilities is severely restricted and can be visualized quite easily within four states or 2^2 possibilities. If the lines can be connected also at mid-points we have 2^3 possible connections of 3 points with another 3 points in space with two lines.

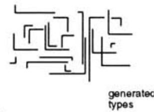


This is the structure and general argument threaded throughout the thesis, to manage simple connections and interdependencies to form larger emergent structures. The system can also be explored at another level which quickly elimi-

nates all predictability to one concern but limits the output to describable family classes.

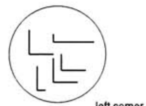
The same system of connections with random length legs will produce an infinite number of distinct objects. The emergent perceivable groupings of families becomes the evaluation factor for understanding the system at another level. The offspring (objects) may be infinite in describable appearance but the families are limited in description and grouping characteristics: large objects, small objects, horizontal objects, linear objects, similar objects, etc. The objects can only be grouped, compared or judged within predictable relationships to possible solutions which could be generated, i.e. the system is relative in which portions are predictable absolute.





generated types

small



left corner

similar length

linear

To describe consistency of output we need to judge the behavior structure and the relative family structures. For example, the possibility of generating objects of the same leg length would be unlikely (depending on the constraints of the variables). The point is not to assume, predict or work towards a larger ordering system, which describes standard accepted notions of proportion of 'conventional' beauty, but to celebrate the possibility of a larger more chaotic intervention within a simple structure which can emerge more didactic architectural relationships. These simple procedures offer greater insight to generating theories about emergent form and the creation of space. The system is very basic but the rich complexity which can evolve from the sys-

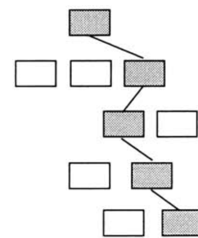
tem leads to more compelling concepts. Behavior based systems are built from fundamental components in which larger notions evolve.

how can localized interactions of objects lead to larger global and non-determined systems?
how can we effectively use modular programming on a larger scale to form more complex programs and representations within a design context?

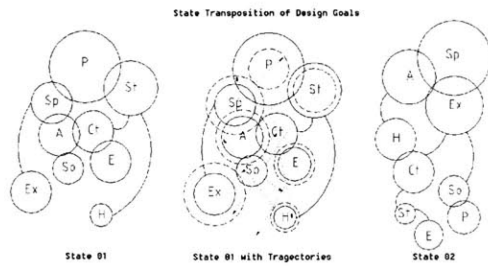
on modular component design 03[a]

The modular approach depends on the context for architectural problem solving and investigation giving weighted importance to the development of objects through scaleable manipulations within procedural methods. This theory departs from the traditional hierarchical approach on two fundamental stances.

01 Hierarchical tree models limit the importance of the entire process while working within the model. The state of the model is limited to the path one takes through the tree towards a given solution while not encompassing the solution search and judgments taken place within the process.

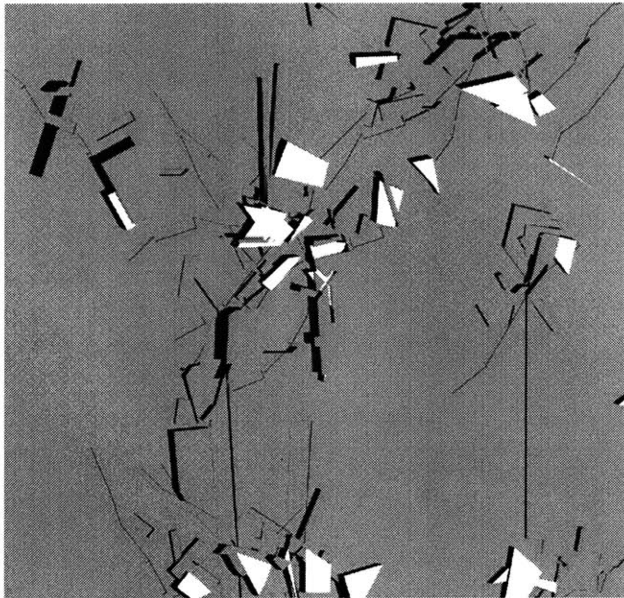


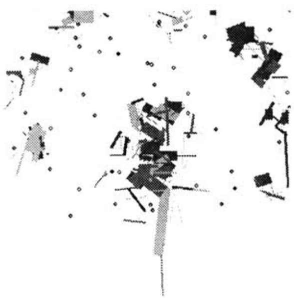
02 A more robust model to explore derivable dependencies with a larger emphasis on problem exploration than problem solving. One theory of



fluid thought models is described in the motion state diagram, in which concepts or procedures are dynamically linked moving through a theoretical space towards "design states". The design states can be explicitly defined, i.e. focus on spatial concerns, focus of affordability, or focus on circulation. The transformations between the states of the model is the generative process. The largest advantage of such a technique involves procedural information to evolve as the structure emerges.

The theory of how one could implement the motion state diagram has not been fully explored and would be valuable research further. Modular evaluative components are dynamically connected depending on the procedures defined within the context of the generated solution environment. What becomes particularly interesting is the concept of space which is integrally linked to smooth motion transformations between states within the model. The larger concept of the state motion model is that particular states can be describable but the transformations between the generating states are dynamic and change through the exploration of the translation process.

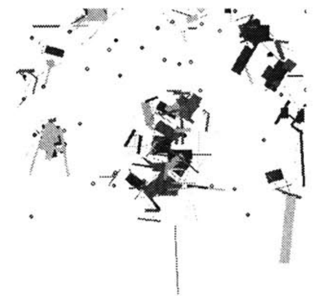




The theory of modular program design works in collaboration with procedural methods. The development of multiple procedural methods working in collaboration (or disjunction) can allow for emergence within the system at several levels. The concept is that a modular design can

be flexible enough to accommodate a variety of experiments by concatenating modules (as needed) and adjusting the properties of the modules. Modules within the framework of the application refer to the separate components. For example, each agent is a modifiable object module; there are movement, simulation, update, interaction, generation, report, export and plotting modules. The general concept behind modular design is to augment the modules to perform additional experiments. The module design is still at a very low level within the object oriented code which subsequently needs to be compiled for each experiment. For further development an interpretable system (higher-order programming) would prove useful for changing module design interactively in-between generations.

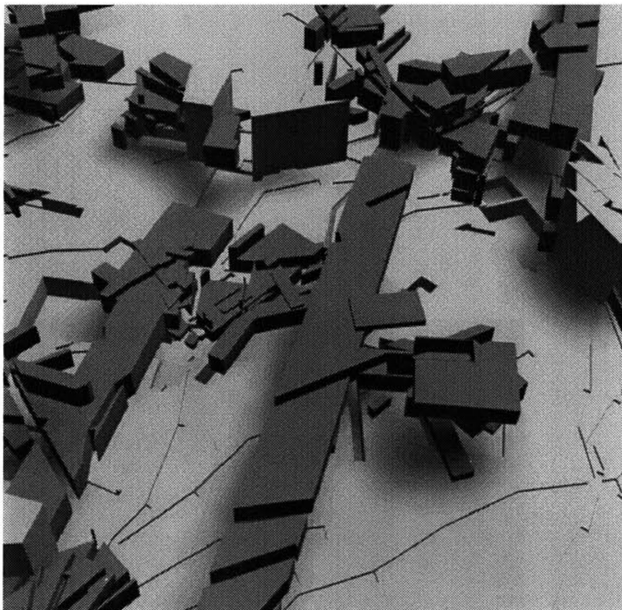
Modular design is implemented within the extents of this simulation application in two regards. First modularity explains the components of the system; for example the movement component moves the objects and then checks for interactions. Within this module information can be recorded, or updated depending on the will of the designer alternatively transpiring behaviors can be recorded to a file, recorded within objects, displayed or updated immediately or after the round. Interactions which occur can be queued or be computed immediately. And second, modular refers to the embedded comments within objects. For example, all objects have multiple display capabilities - the three-dimensional display module of an object can be



transformed so that the object fundamentally draws (displays) itself differently within the environment. For example, an object can test its environment, if it is adjacent and intersecting two solids on the walls of the object than it could transform itself into a void object.

on 03b procedural thinking

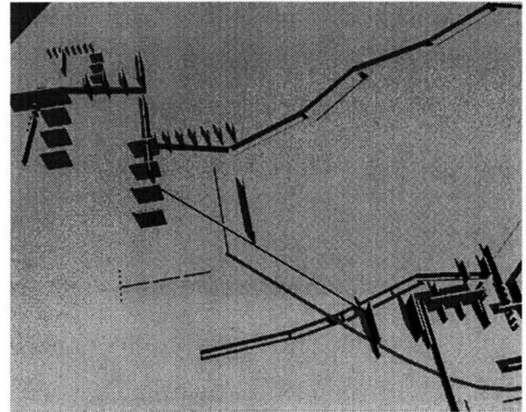
how to fit the project within the scope and context of an architectural process?

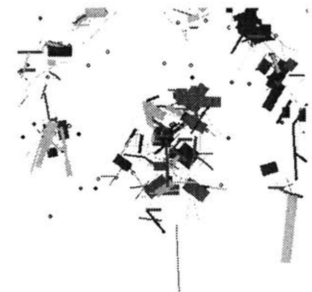


The procedures and methods involved tend to respond to early phases within a design process in which ambiguity and abstractness, in an inclusive search space, is more beneficial. But to describe the project solely as an attempt to work within early phases of design is a misconception. More accurate definitions and descriptions of a pedagogical nature are more appropriate. The system is highly internalized and inexperienced in regards to architectural knowledge aside from the formal architectural behavior inherently built into the system. This knowledge is limited to the context of the experiment. Situations and events are determined which allow for the transference of information, manipulation or the generation of information. One fundamental theory about the structure of the program is to determine the types of interactions between the objects and then place them in an environment which generates the simulation. For example: an object mov-

ing in a particular manner through the environment could learn that it is more beneficial to generate objects rather than destroying them. The possibility to learn is through implicit feedback mechanisms and communication with other agents, successful communication allows the agent to continue to develop.

A highly rational thinking process within a structured context can only describe the antithesis of the investigation. The project is approached in a free context exploring, playing and investigating through countless "coding sketches" and process experiments. This way of thinking and working is not as productive or efficient (based upon time constraints and undestandibility) as working with a conventional method with a viable hypothesis within a larger scientific, technological or architectural framework. The explorative, generate "on the fly" method allows for a more flexible approach which is less guided and less committed to the solving of a problem and more involved in opening up the questions. This is a new and unexplored way to computational, architectural, design and aesthetic approaches. The role of the designer as procedural inventor and orchestrator. The behavior based approach affords a more explorative stance because the system is inherently about exploring solution space in a nondeterminate and flexible fashion. The environment can be catastrophically different and the agents can explore and adapt evolving new methods over time.





agent model 04

agents in the environment 04[a]

Agents within the environment share, transfer, and communicate with one another. The following are descriptions of the object/ agents with their properties described. As part of a larger event (the simulation) the agents also record their history as traces, and their interactions, these are what will later be called 'design stories'. These stories are at a very preliminary level, alternative programs need to be coded to interpret decentralized data in relationship to formal outputs to see if any meaningful correlation could be determined.

points

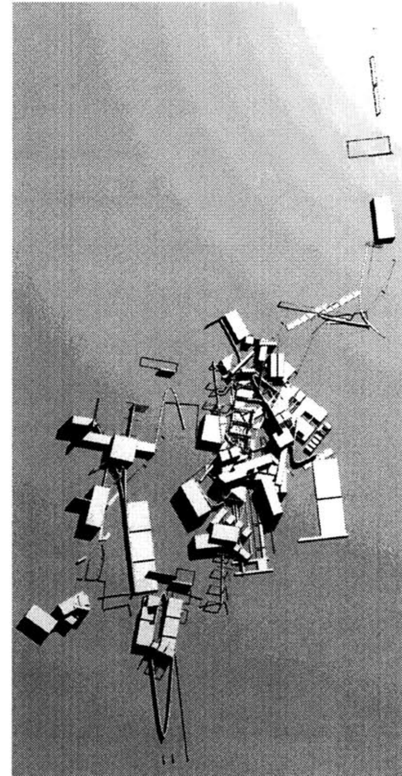
The point class is a traditional object-oriented base class with extra functionality. In a computational sense the base class is the fundamental building block from which other classes develop. The point class plays a significantly larger role within the structure of the program. All agents have inherited point class information. The following is the header file of the point class and the data structures located within the agent.

```
class FPoint
 {
   public:
 float x,y, move_factor, field, width, ob_slope, ob_x;
 int color, id, prop, ob_flag, ob_id;
 Tree *move;
 Tree *inter;

   FPoint          (FPoint&);
   FPoint          (float, float, float, int);
   FPoint          (float, float);
```

```
    FPoint          (int , int, int);
    FPoint          ();
    void move_object();
    void plot_zone();
    int property();
    void move_object(int);
    int intersect(FPoint, FPoint, FPoint);
    int ccw(FPoint, FPoint, FPoint);
    ~FPoint         ();
    float&  operator[] (int);
    FPoint  operator= (FPoint);

    friend ostream& operator<<(ostream& , FPoint&);
    void plot();
};
```

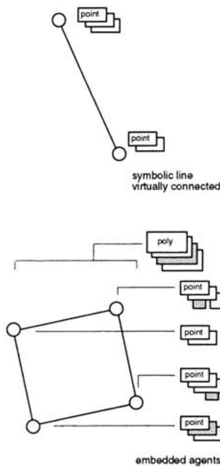


lines

Lines exist only for calculative purposes and do not exist as a class with methods. Properties of lines are hybrids of point characteristics, or as representable constituents of polygons. Although a point is not a specifically representable object, it can encapsulate conceptual formal information. A point moving in space on a straight line with a trajectory has implied meaning of linearity whereby offsets from the development of the intersection would be linear. Boundaries or traces represented as lines are inherently derived from concept points, these can be displayed (described) spatially (three, or two dimensionally) or as a line. The line class and its inclusivity was debated over extensively and a line class would be helpful fundamentally within a larger system especially in generating future concept agents, an agent which could have two dimensional influence through one dimensional existence.

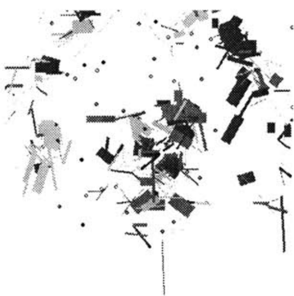
polygons

Polygons are higher level developments of points with implied line characteristics. A polygon is the other fundamental shape and form giver to the environment. For the most part, any development within the field takes place as a by-product of the information embedded within the polygon and its relative responsiveness to communicating with other agents. Polygons within this simulation all have 4 points but are not necessarily orthogonal. The points which make up the polygon agent can be randomly determined or 'consistent' polygons can be made describing regularized data within the object for specific tasking purposes. The points are the fundamental communicators within the agent passing messages to the polygon agent and the other point agents.



The diagram shows the makeup and breakdown of the polygon and the embedded points of the polygon. The line object between the points is for the most part nonexistent except for in geometric reasoning algorithms. Polygon can have a variety of movement characteristics. In most cases the Polygon class is placed on path trajectories towards a specific location in space.

```
class Polygon
{
 private:
  int n, id, ht, color, attraction;
  float area, area_ratio, long_side;
  Tree *move;
  Tree *inter;
  FPoint *p;
  void    calc_value();
  int point_influence[25];
  void clear_point_buf();
  float path_x;
  float path_y;
```

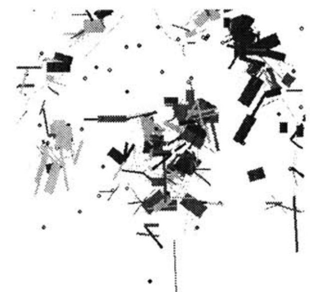


```
public:
 float length, width, value_ratio;
 int polygon_found;
 FPoint min_xy;
 Polygon() {n = 0; p = NULL; id = 0; length = 0.; width = 0.; poly-
gon_found = 0;}
 Polygon(int m) {n = m; p = new FPoint[m+1];}
 Polygon(int m, float* xpoint, float* ypoint, int pid, int col, int
height);
 Polygon(int m, float* xpoint, float* ypoint);
 Polygon(char a, int ident);
 Polygon(FPoint x1, FPoint y1, FPoint x2, FPoint y2);
 ~Polygon() {};
 void move_object(int);
 void action(int, int, float);

 void   set_vertex(int, FPoint);
 FPoint  get_vertex(int);
 int    get_nvert(){return n;}
 int    get_attraction(){return attraction;}
 void   set_id(int a){id = a;}
 int    get_id(){return id;}
 int    get_ht(){return ht;}
 int    get_color(){return color;}
 void   display();
 void   print3d();
 int   property();
 int   path(FPoint pnt1);
 void   calc_min_xy();
 int    inside(FPoint t);
 int    enclose(Polygon& a);
 int    overlap(Polygon& a);
 void   move_object(int id, float new_x, float new_y);
 void   rotate_object(int id, float about_x, float about_y, float
dtheta);
 friend  ofstream& operator<<(ofstream&, Polygon*);
 int intersect(FPoint, FPoint, FPoint, FPoint);
 int ccw(FPoint, FPoint, FPoint);
 friend  float angle(FPoint, FPoint);
 friend ostream& operator<<(ostream& , Polygon*);
 void   plot();
 void   plot3d();
 void   clear(){delete this;}
; };
```

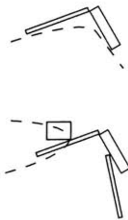
object

The Object class is a second order agent within the field. The Object is very similar to a polygon, but not used as a constructive generative agent. The Object class is a by-product of other agent interaction. The Object can be viewed as a computationally slimmer and more docile agent than the polygon which is much more robust. The Object class is used as the building block agents within the field. Simple manipulation of objects can lead to larger structures with unpre-



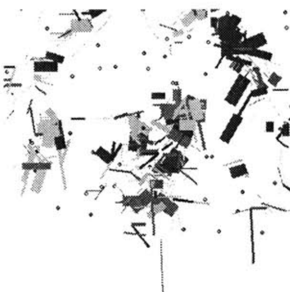
dictable consequences. The diagram below consists of Objects which were further manipulated by point agents which moved them by carrying them and placing them in other locations.

The rules to generate these agents are primarily built into the point class. The point class can carry around objects and deposit objects through the intersection/action with them. For the most part the path points fulfill this role for as they move linearly they pick up objects within the field. The placement of the Objects is less chaotic. When a point carrying an object intersects another Object the point can place the Object at either endpoint, the midpoint, or the intersection point of the Object..



third order objects

Third order objects which evolve through the structure of the system include splines and curves showing developing localized connections. As a patch within the environment grows significantly splines develop to connect elements within localized regions. This interaction is still at a very localized level but communication, as vibrations through the environment, can spread throughout larger distances. The splined objects are generated based upon weighted values in the points of the object class.

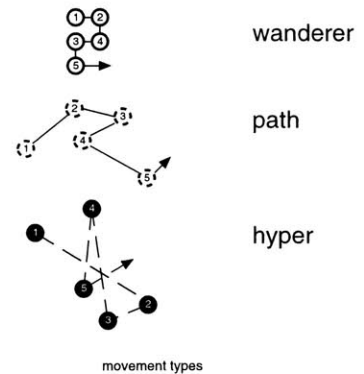


```
class Object
{
 private:
  int n, id, ht, color;
  FPoint *p;
  float plate;
  float z1, z2;
  int point, poly, act, ob_flag;

 public:
  Object();
```

```
  Object(FPoint a, FPoint b, int id_num, int pt, int pl);
  Object(float x1, float y1, float x2, float y2, int id_num, int pt, int pl,
float size);
  ~Object() {};
  FPoint get_vertex(int);
  float get_slope();
  float get_aslope();
  float get_dist();
  int get_flag(){return ob_flag;}
  void toggle_flag();
  int get_id(){return id;}
   void action(int, int, float);
  void update(FPoint t);
  void    plot();
  void    plot_transform();
  void    clear(){delete this;}
  void    print3d();
  void    print3d_ab();
};
```

movement of agents **04b**



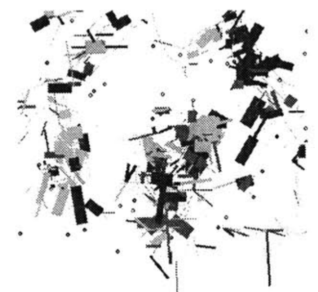
movement types

how can movement of objects within the environment evolve potential archetypal interactions?

The movement types embedded within the system are multi-fold and used specifically to derive characteristics and approaches to various emergent characteristics.

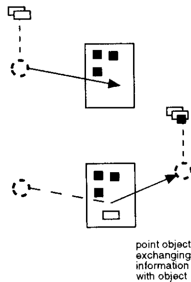
wanderer

Wandering agents are objects which statistically will remain in the same position over n-large generations. These objects will move orthogonally and locally within a region. The behavior which can be associated with such movement types is information spreading within a local region. For example these points can communicate with the agents within their region, this information can then be exchanged to other objects within a region to allow for a much larger understanding of developable regions in the model. The wanderer can also make multiple



changes onto the affected area of the local region, this can result in development, or deterioration explosion.

path



Objects moving along a path seem to have the largest influence across a wider spectrum within the environment agent model. These agents check for intersections as they move along incrementally through the environment. The agents can theoretically transmit and carry information across boundaries and regions from different sections within the environment.

hyper

Hyper agents move from location to location randomly within the model. These agents have the most successful interaction on a larger scale within the environment. Hyper agents can carry information from developing patches within the space, transmitting information across the environment.

higher order movement types

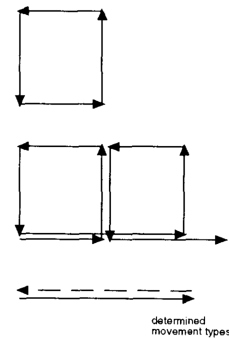
path-directed: path directed objects move through space but towards a local point, region or other object with a predetermined strength tendency. The strength tendency determines how long it will take for the object which is moving to converge on the determined point within a movement spectrum. The purpose of path

directed objects is to realize the potentials of an object moving in space along a path and to simulate what properties of development could take place along the path.

A more appropriate use of the movements types would be to match them with specific behavior types. The diagrams represent potential locations for specific development around a well-defined path within the system.

Coded movement sets patterns for the movements of objects. For determining specific and predictable behavior within the environment. Or through the analysis of regional behavior.

evolved movement types

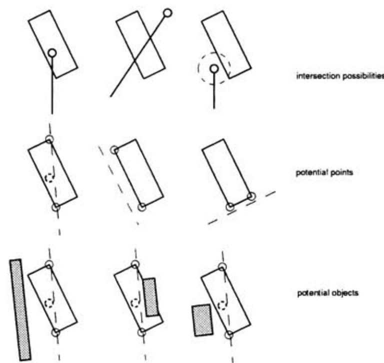


Future investigations will include agents which inherit or develop multiple movement behaviors. For example agents could exist having both hyper and wandering characteristics. As an agent 'teleports' through the environment, it could wander searching for interaction, if unsuccessful the agent could 'teleport' again. Other evolvable properties could include 'hyper-transporters', agents which move between developing patches within the environment transmitting information to agents.

Growth: The ability for objects to generate, augment or add a particular component to the field of generation of another object.

Decay: The subtraction of objects, properties or characteristics from agent within the environment. Decayed objects can also represent themselves as voids within the environment.

Transformation: Changes the existing state of objects or agents within the environment. Transformation behavior interaction are usually polar in nature or adaptive to the characteristics of the transforming object.



higher order formal development 04d

how is form evolved and generated within the diagram?

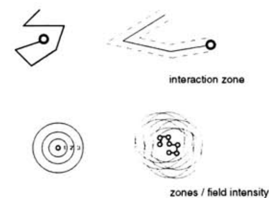
One form of development within the system is based upon direct object interaction. In this case the interaction is solely based on the objects intersecting in time and space. The diagram below shows a point and a polygon entering one anothers field. The field of the object is the local extents (or radius) of the objects influence and at what strength.

The diagram shows a possible interaction situation. If the object is generative it could produce a hybrid from the objects. In this case a point object is intersecting (communicating within the

determined field) with a polygon. Two 'embedded points' are chosen from the polygon by the point agent (intersecting) and then processed through the agents own mechanism to determine how to represent the formal by-product. The two points chosen are reasoned through the descriptive properties within the agent. For example a growth point intersects with a multi-property agent with 2 growth points and a decay and transformation point. The growth object could be attracted to the growth points and use the line-segment representing growth to develop another object/agent. A slope region is determined by the chosen points which in turn generates an object. The object developed then encapsulates strength properties from both agents and determines the localized placement adjacent to the other objects. Within this example a large number of possibilities could arise from the simple description of the data embedded within the object at any given time.

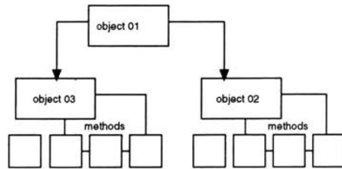
communication of agents **04e**

Agents, formal and conceptual, communicate to each other through interaction. This overarching principle evolves through the physical (direct) intersection of moving objects within the environment.

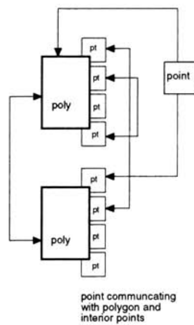


The system primarily operates on a decentralized level of interaction which gives rise to an alternative result in the process. At any moment the

objects within the simulation field are solely responsible to themselves and their direct (local) environment. This is to say that communication of elements only transpires through direct interaction.

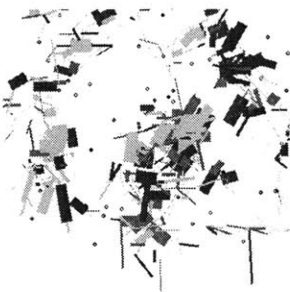


Traditional computational approaches represent communicative properties through the description of top down hierarchical methods. The diagram represents conventional information dissemination.



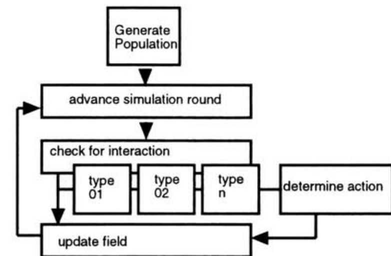
point communcating with polygon and interior points

In a behavior based system retrievers pass through separate object methods to reach necessary information between the agents. In the diagram information is passed directly through and to the objects which are communicating. This leads to interesting ideas of emerging ambiguity or contradictory agents. Whereby a polygon as an agent could have separate behavior than the constituent point agents which make up the polygon. It is inconclusive if this methodology is a useful way of viewing object communication. The method is computational faster, but takes longer to code and implement. The communication can also lead to unpredictable behavior of agents whereby the richness of an environment becomes more than the sum of each agent characteristic.



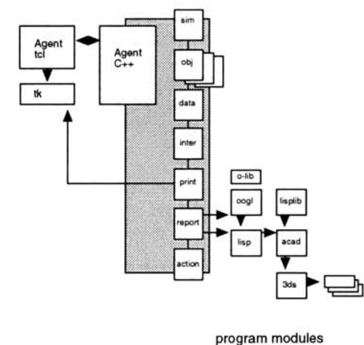
structure of the simulation 04f

The initial stage of the simulation is to generate a population of agents. The generation can be accomplished by population type, number, or behavior depending on the experiments under investigation. The properties of objects can be directed or generated. The number of point agents, and polygon agents, or movement types can be adjusted within the program.



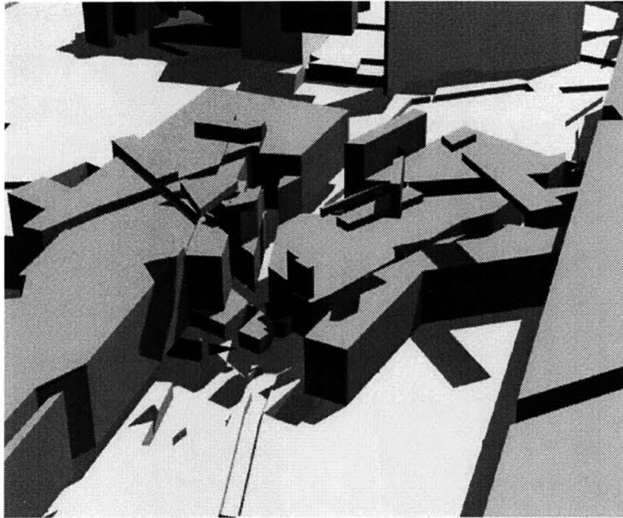
overall program structure

The entire program is operating on multiple levels simultaneously. The diagram shows how the application communicates and uses component modules to enact the simulation. The two largest sections of the application are the interpreter and the executable code. In order to get access and display information graphically Tcl/Tk is used as the visualization engine. Tcl is an interpreter which sends messages to the C++ code via a socket, and alternatively the C++ code sends information to the interpreter telling it what to do, what to update, and how to display information. The Tcl code sends messages to the Tk system which is bootstrapped on top of X11 windows. Inside of the C++ agent code are modules which can be configurable to work within the experiments needed at a given time. These modules can then communicate to other programs such as Autocad, 3dstudio, Geomview, Inventor, or other imaging programs. Depending



program modules

on the nature of the experiments (or invention) separate modules can be used.



The modules can be updated internally or patched with new information. For example if a the user wanted to run a simulation of all orthogonally based objects and movements, the move module and the pool generation module could be adjusted to accommodate the experiment.

simulation modules **048**

main

The agent_module is the main() controlling loop of the program. The agent_module initializes the population and its properties. After initializing the simulation the Tcl/Tk_module and interaction_module take over the control parameters. The agent_module also initiates the global variables of the system and the arrays, trees, or other data structures needed for storing or creating agent information.

agents

The agent_module contains the descriptive methods, behaviors, goals and data of the agents within the environment. The module also contains information of derivative forms of agents and communication properties.

round

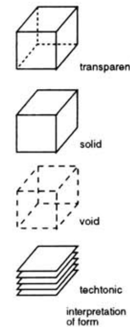
the round_module contains information regarding specific instructions or reasoning which should occur at each simulation round. This module determines whether agents should check for interaction or develop autonomously. Also the round_module determines the queuing of interactions within the environment.

data

The data_module describes what global data should be managed through the process. The module also manages data transmission and flow between modules, including information to be stored in files whether it is procedural or recorded.

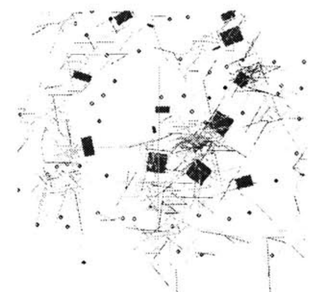
plot

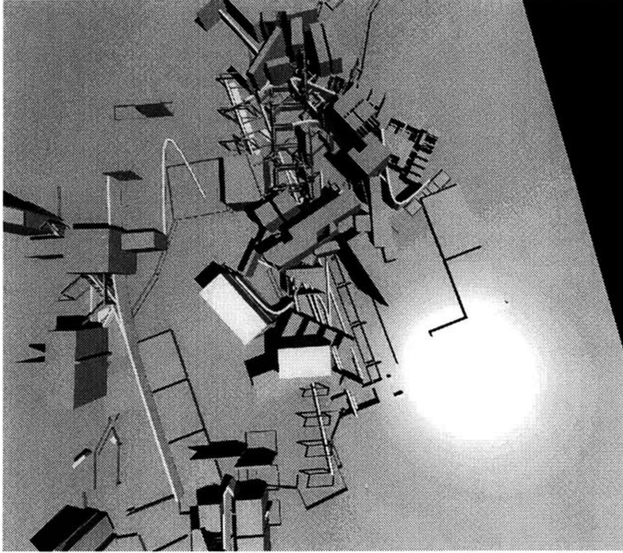
The plot module transforms the object to representable forms in other languages or file format, for example: to the Tcl/Tk graphics output, to a DXF file, as Autolisp code, or as OOGL code. Information embedded within the system conceptually relates to type information but not expressly in formal concerns.



interpretation of form

In order to understand how form can be derived from the system, or how additional information can be embedded within the structure so to allow another type of interpretable information transformation to take place, it is required to understand the properties and behaviors of the agents. Throughout these investigations the formal output was kept at a very low level with very little reasoning at higher orders of develop-





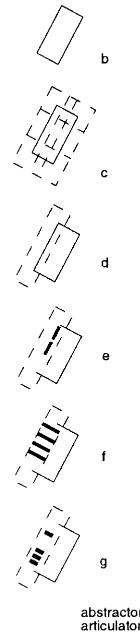
ment. To this end agents are mostly described as boxes with behaviors. Object development can be looked at a variety of levels depending on the focus of the investigation. Procedural information is built into and recorded within the system. This information can be interpreted or augmented has seen fit by the 'designer' within the process. In this case object behaviors (void, solid, transparent) are implemented into structures with these characteristics. These transformable behaviors determined by other interventions and communication of agents. For example an agent can embed height information of developable object within other agents, or alternatively normalize height value based upon past experience and communication.

Within the procedure other descriptive information about strength values, attraction behaviors, generative characteristics, movement sequences, height, size, form, position is simulation, etc. can be exchanged.

display

The display_module contains explicit methods for telling the interpreter how to draw, and if to draw within the environment. Also specified is information on when to draw and redraw depending on interaction or update of objects. It is not necessary for all information to be displayed graphically, for example object paths or strategic simulation points can be told by agents to show or hide themselves. This feature is beneficial while trying to visualize the complex infor-

mation transmissions of the system. As the system is generating, "weeding through data" can be quite cumbersome. Having the ability to display data in a intuitive manner helps for understanding and directing the development of the program.



move

The move_module contains descriptions of the characteristics of the moving patterns of agents. Movement patterns can be random, follow one of the predefined methods, or new patterns can be written.

construction conditions

Objects can be constructive constituents of each other, for example the polygon being made of points. But all points within the polygons have their own characteristics and behaviors. All agents can send messages to one another, which makes writing the code a bit more obtuse. It allows objects emerge and decay with greater flexibility.

higher order constructions

As agents develop and transform the environment, the can manipulate existing information (usually Object class agents) through a process called abstraction and articulation. Agents manipulate Objects into separately representable units but keep the perceivable characteristics of the forms. The diagram represents the transformation properties:

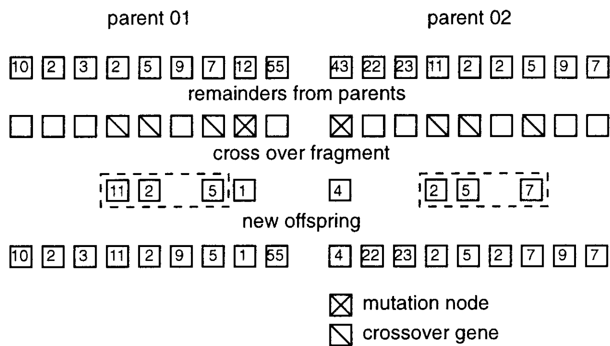
a: shows the initial line form.

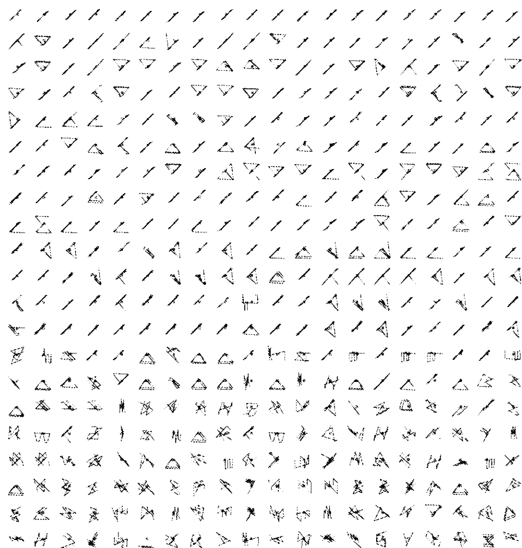
b: the extracted (generated) form from embedded information in the object.

c: shows the constructable areas of each bounded line zone of the object.

d: isolating one edge for further development

e-g: potential formal manipulations of the edge into articulated elements.



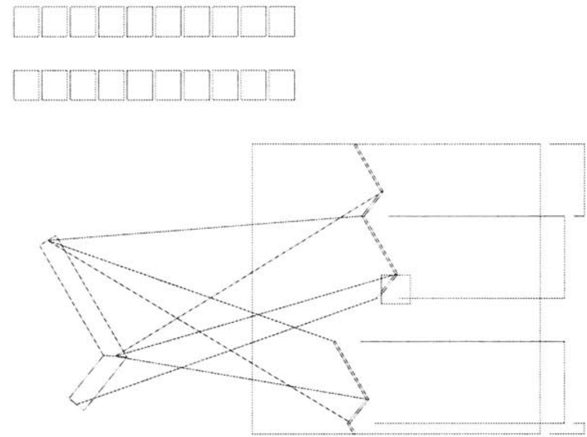


genetic experiment 05a

The point of the genetic experiment is to explore formal development through an evolutionary system in which characteristics evolve from embedded information within objects but which does not have to be explicitly coded. The notion of codifying entire bodies of architectural knowledge is admirable but at fundamental levels, especially when considering judgment values embedded within the system, a problematic endeavor. These judgments can imply and perpetuate preconceived aesthetic and formal values limiting the variability or robustness of the model and its output. In this example object information is embedded within the genome and then evolved through user interaction. The simulation is implemented within a larger genetic algorithm which manipulates the genotypes of the objects. The genotype of the objects contain procedures for representing and creating the object.

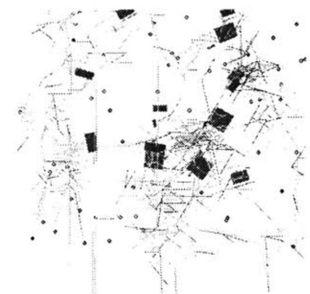
Evolution: The simulation begins by first creating

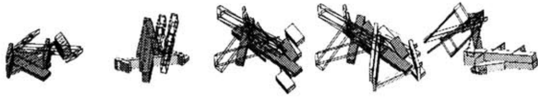
the population of genotypes. The seed most commonly used to create the population is random, alternatively existing genotype seeds can be used for the creation of the population. In any population a fitness or survivability has to be determined in order to process future generations. The fitness can be predetermined, meaning an implicit goal can be built into the system with which the objects can objectively be ranked. High percentile populations are mated and produce new generations. When a fitness is made explicit the simulation can converge autonomously towards a solution. The role of evolution becomes significantly different when fitness is determined by the user (interactively determined fitness). The first experiment measures itself against an objective fitness and in the second the fitness is determined by the user.



on learning 05b

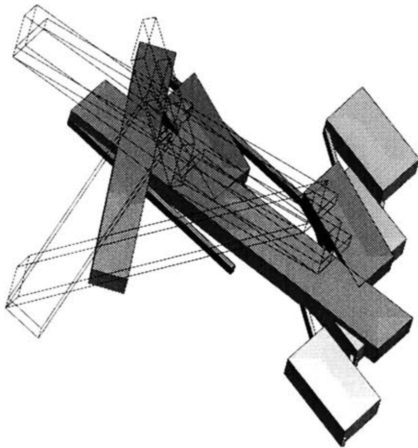
This first experiment was to initially test the robustness of a system which learns very basic components of drawing, and representation with a simple genotype description. The fitness is determined by how well the objects draw line through a box. A genetic object can draw lines within the environment and the number of intersection with the 'fitness box' are then calculated and the simulation evolves the population and continues. The genotype is a ten integer description of the objects movement parameters.





genotype representation:

position	characteristic
01	x position
02	y position
03	+/- movement
04	x offset
05	+/- movement
06	y offset
07	+/- movement
08	x offset
09	+/- movement
10	y offset

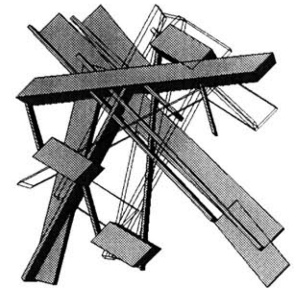


Each object is generated n number of times and the fitness is derived from the interaction with the box. The example shows randomly created objects each with a different genotype with the resulting output drawn.

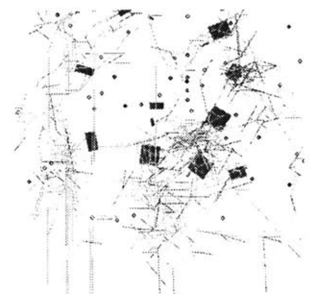
on form evolution 05c

This experiment is a derivative of the first example with a larger genotype describing three dimensional characteristics of the objects within spatial constraints but evolution is guided through the interaction of the viewer. For example, an initial pool of objects are created and then evaluated or judged by the viewer. This can be done in ranking order or by assigning some

value to the objects. After determination of fitness the generation continues but with the mating of the highest fit examples. The next generation has a larger probability of passing on characteristics of fit genotypes through the evolution of the code into other objects. Evaluation on objects can continue until an object with satisfactory characteristics have been evolved or until diversity within the system becomes significantly stymied. Potential development within the system are readily viewed even with a limited procedural genotype from this example at a base level tacit questions of emergence and understanding can be derived. As one looks at an object with idiosyncratic concerns it is always not possible to explicitly convey aesthetic, pragmatic, or functional descriptions. The way a designer or a perceiver views an object does not expressly have to be codified within the system, this is fundamental shift from KBAI. Within a KBAI example structured knowledge would be built into the system and reasoning would take place in response to the system, the system could then plan or generate from the interpretation of the responses. At this point little known about encoding complexities of design that it is more beneficial to react and develop alternative bottom up approaches and methodologies that can respond to larger systems without having the knowledge fundamentally built into the system.



If we were to evaluate the following example, how would we build in conditions and responses to such a highly complex way of reasoning on



coming to conclusive responses to the system. Idiosyncratic approaches can be embedded into the system throughout training or development over time, but initially should the designer of such a system succumb to dogmatic descriptions of composition or form descriptions? In the example below three walls are placed together. How can these figures be judged? What should the criteria be? In fact the objects could be evaluated in any number of concerns; spatial, compositional, in section or plan, space forming, in regards to circulation, experience, privacy, etc. Further, at any evaluative concern the questioning and disagreement or lack of conclusive understanding would continue to be divisive.

modular design of genetic structures

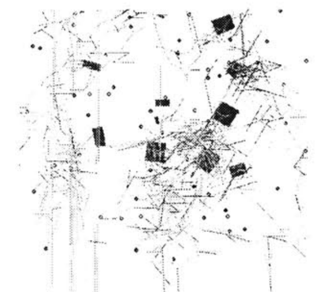
One would be quick to point out the raising complexity towards specific emergence of a genetically created object. As the data and procedure set become large the ability to derive objects becomes increasingly complex. The generation of initial populations becomes critical in order to generate a large enough pool within the spread creating diversity. The data could be further augmented into component structures each of which could significantly develop to pass characteristics through generations.

on interpretation and conclusion 06[a]

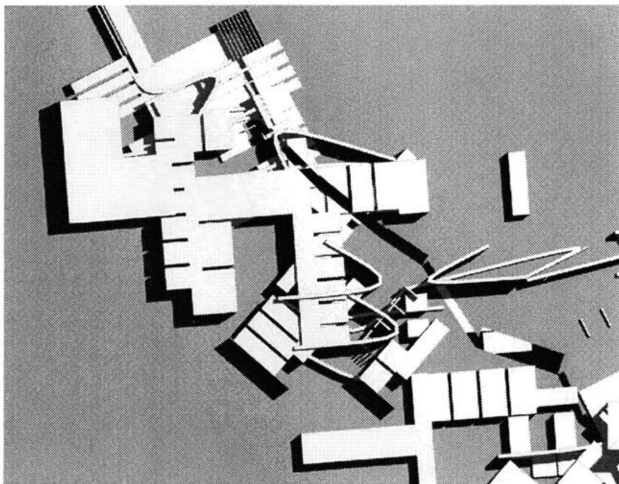
how should the figures or objects be viewed?
how can one place the research in a larger architectural framework?

The thesis examples should be viewed as one approach towards embedding "smart objects and agents" into architectural and computational thinking environments. One would argue that reducing architectural to formal concerns limits the description of the characteristics of architecture too myopically. The approach of such systems is to question how to embed information into agents, whether form, concepts, objects, or events in a manner which can take advantage of today's computational capabilities. It is also necessary to question the manner in which we traditionally design, draw and think both by hand and with the use of the computer.

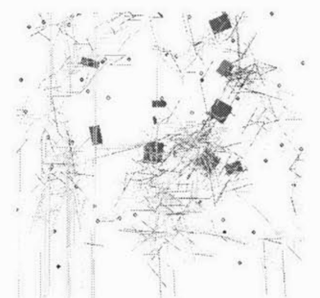
Larger questions need to be studies and explored within simulated environments. These questions are related to ideas and concepts of decentralized processes, emergent systems, judging creative computational expression, and managing highly complex notions of ambiguous design methodologies. The thesis represents a starting point for communicating generative ideas to a larger community for future development.



One component of the thesis was to explore notions of designing as a procedural way of thinking which can be related to both computational behavior based methods and generative processes. This is not to confirm that the mind and the computer are significantly mappable, but rather an attempt to move towards definable and codeable ways of thinking that can be experimentally tested. Ambiguity has been purposely implemented into nondeterminent algorithms within the applications and threaded throughout the thesis investigation. Fundamentally the thesis is about exploration of untested ideas, attempts at proving truth, science or predictable responses have been placed at a very low level within the investigation.



Behavior based investigations can lead to fundamentally new ways of approaching design or creative architectural and generative experiences. The thesis is a departure from traditional top down computational approaches in which evolving and expanding the problem space from bottom up levels becomes the focus. Questions of embedding knowledge into architectural agent objects looks towards the future of more intelligent CAAD software, and determining alternative approaches to generation which are much more process oriented and flexible rather than predetermined and static.



algorithm

<algorithm, programming> A detailed sequence of actions to perform to accomplish some task. Named after an Iranian mathematician, Al-Khawarizmi.

Technically, an algorithm must reach a result after a finite number of steps, thus ruling out brute force search methods for certain problems, though some might claim that brute force search was
also a valid (generic) algorithm. The term is also used loosely for any sequence of actions (which may or may not terminate).

Artificial Life

<algorithm, application> (a-life) The study of synthetic systems which behave like natural living systems in some way. Artificial Life complements the traditional biological sciences concerned with the analysis of living organisms by attempting to create lifelike behaviors within computers and other artificial media. Artificial Life can contribute to theoretical biology by modeling forms of life other than those which exist in nature. It has applications in environmental and financial modeling
and network communications.

There are some interesting implementations of artificial life using strangely shaped blocks. A video,
probably by the company Artificial Creatures who build insect-like robots in Cambridge, MA (USA), has several mechanical implementations of artificial life forms.

complexity

<algorithm> The level in difficulty in solving mathematically posed problems as measured by the
time, number of steps or arithmetic operations, or memory space required (called time complexi-

ty,
computational complexity, and space complexity, respectively).

The interesting aspect is usually how complexity scales with the size of the input (the "salability"), where the size of the input is described by some number N. Thus an algorithm may have computational complexity O(N^2) (of the order of the square of the size of the input), in which case if the input doubles in size, the computation will take four times as many steps. The ideal is a constant time algorithm (O(1)) or failing that, O(N).

See also NP-complete.

creativity

Various definitions are used to reach a better understanding and interpretation of the word. The methods involved by each group or category in hypothesizing aspects of creativity and interpretation have lead to general theories about the brain. If we examine the main definitions, and classify them, they can be organized into six major groups or classes. Of course many definitions can fit into one or more class.

Class A Gestalt or Perception: This class places a major emphasis upon the recombination of ideas or the restructuring of a "Gestalt."

Class B End product or Innovation: "Creativity is that process which results in a novel work that is accepted as tenable or useful or satisfying by a group at some point in time." Harmon refers to creativity as "any process by which something new is produced—an idea or an object, including a new form or arrangement of old elements."

Class C Aesthetic or Expressive: This category tends to more personally oriented, with a major emphasis on self- expression. It usually includes the role of the 'starving artist' who creates art for himself or herself.

Class D Psychoanalytic or Dynamic: This group is primarily defined by certain interactional

strength proportions between the id, ego, and superego.

Class E Solution thinking: In this category more emphasis is placed upon the thinking process itself rather than the actual solution of the problem. Guilford defines creativity in terms of a very large number of intellectual factors. The most important of these factors are the discovery factors and the divergent-thinking factors. The discovery factors are defined as the "ability to develop information out of what is given by stimulation." The divergent factors relate to one's ability to go off in different directions when faced with a problem.

Class F Alternative or other: In this category one could find the definition of creativity as "man's subjective relationship with his environment". Or according to Rand the "addition to the existing stored knowledge of mankind."

Boden outlines two different types of creativity, one type is the psychological or P-creative, the other historical or H-creative. Both types deal in novel ideas of creation, but novel in regards to different interpretations. P-creative thoughts, concepts or idea are novel to the individual mind which conjured the idea. P-creativity is a thought which is personally generated that has never been thought before, a idea in which the thinker previously had no understanding or connection. P-creative thoughts do not have to be novel in regards to societies interpretation. An idea is H-creative if the idea is truly novel—never been thought of, developed, or constructed before. Most people tend view the H-creative individual to be more creative, in that the idea is completely novel and never thought of before.

genetic algorithm

(GA) An evolutionary algorithm which generates each individual from some encoded form known as a "chromosome" or "genome".
Chromosomes are combined or mutated to breed new
individuals. "Crossover", the kind of recombination of chromosomes found in sexual reproduc-

tion
in nature, is often also used in GAs. Here, an off-spring's chromosome is created by joining segments chosen alternately from each of two parents' chromosomes which are of fixed length.

GAs are useful for multidimensional optimization problems in which the chromosome can encode the values for the different variables being optimized.

genetic programming

<programming> (GP) A programming technique which extends the genetic algorithm to the domain of whole computer programs. In GP, populations of programs are genetically bred to solve
problems. Genetic programming can solve problems of system identification, classification, control, robotics, optimization, game playing, and pattern recognition.

Starting with a primordial ooze of hundreds or thousands of randomly created programs composed
of functions and terminals appropriate to the problem, the population is progressively evolved over
a series of generations by applying the operations of Darwinian fitness proportionate reproduction
and crossover (sexual recombination).

heuristic

A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee solutions.

nondeterminism

<algorithm> A property of a computation which may have more than one result.

One way to implement a nondeterministic algorithm is using backtracking, another is to explore

(all) possible solutions in parallel.

NP-complete

<complexity> (Nondeterministic Polynomial time) A set or property of computational decision problems which is a subset of NP (i.e. can be solved by a nondeterministic Turing Machine in polynomial time), with the additional property that it is also NP-hard. Thus a solution for one NP-complete problem would solve all problems in NP. Many (but not all) naturally arising problems in class NP are in fact NP-complete.

NP-hard

<complexity> A set or property of computational search problems. A problem is NP-hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time.

Some NP-hard problems are also in NP (these are called "NP-complete"), some are not. If you could reduce an NP problem to an NP-hard problem and then solve it in polynomial time, you
could solve all NP problems.

Bibliography

Akin, Omer, Psychology of Architectural Design, Pion Limited, London, 1986.

Boden, Margaret A., The Creative Mind: Myths and Mechanisms, Weidenfeld and Nicholson, London, 1990.

Boden, Margaret A., Dimensions of Creativity, MIT Press, Cambridge, MA, 1994.

Dawkins, Richard, "The Evolution of Evolvability", *Artificial Life, SFI Studies in the Sciences of Complexity*, ed. C. Langton, Addison-Wesley Publishing Company, 1988.

Dawkins, Richad, The Blind Watchmaker, Harlow Longman, 1986.

Freud, Sigmund, On Creativity and the Unconscious, Harpor and Row, New York, 1958.

Goldberg, D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

Hausman, Carl R., and Rothenberg, Albert, The Creativity Question, Nuke University Press, Durham, NC, 1976.

Hofstadter, Douglas R., Godel, Escher, Bach, An Eternal Golden Braid, Vintage Books, New York, 1979.

Kipnis, Jeffrey, 'Architecture Unbound', pg 12-23, AA, London,1985.

Koza, J., Genetic Programming: on the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

Maes, Pattie, "Modeling Adaptive Autonmous Agents", *Artifial Life Journal*, edited C. Langton, Vol.1 , No. 1 & 2., MIT Press, 1994.

Mitchell, William, The Logic of Architecture, MIT Press, Cambridge, 1990.

Morrison, Foster, The Art of Modeling Dynamic Systems, Forecasting for Chaos, Randomness, and Determinism, John Wiley & Sons, Inc., New York, 1991.

Poundstone, William, The Recursive Universe, Cosmic Complexity and the Limits of Scientific Knowledge, Contemporary Press, Chicago, 1985.

Sartre, Jean-Paul, The Psychology of Imagination, Citadel Press Book, New York, 1991.

Sims, Karl, "Evolving 3d Morphology and Behavior by Competition", Thinking Machines Corporation, internal document.

Sims, Karl, "Interactive Evolution of Dynamical Systems", *Toward a Practice of Autonmous Systems: Proceedings of the Firt European Conference on Artifical Life*, ed. by Varela, Francisco, & Bourgine, MIT Press, 1992, pp. 171-178.

Sternberg, Robert, The Nature of Creativity, Cambridge University Press, 1988.

Virilio, Paul, The Lost Dimension, Semiotext(e), Columbia University, New York, 1991.

Woolley, Benjamim, Virtual Worlds, Blackwell Publishers, Oxford, 1992.

Dynamic Links
http://www.mit.edu:8001/people/jcsk/home.html
http://alife.santafe.edu/alife/papers.html
http://www.sunlabs.com/research/tcl/index.html
http://www.cogsci.indiana.edu/
http://mitpress.mit.edu/jrnls-catalog/adaptive.html
http://papa.informatik.tu-muenchen.de:80/~weissg/LNAI-1042/
http://www.aic.nrl.navy.mil/galist/
http://agents.www.media.mit.edu/groups/agents/

```
::::::::::::::
add_object.h
::::::::::::::
#ifndef _ADD_OBJECT_H
#define _ADD_OBJECT_H
void add_object(int);
#endif
::::::::::::::
fpoint.h
::::::::::::::
/*******************************************************************
*                                                                  *
*                    FPOINT.H                                      *
*                                                                  *
*******************************************************************/
#ifndef FPOINT_H
#define FPOINT_H

#include <iostream.h>
#include <math.h>

#define X 0            // make code more readable for humans
#define Y 1
#define min(A,B)  ((A)>(B) ? (B) : (A))
#define sqr(x)  ((x)*(x))


class FPoint
  {
    public:
  float x,y, move_factor, field, width, ob_slope, ob_x;
  int color, id, prop, ob_flag, ob_id;

    /*    constuctors */

    FPoint              (FPoint&);
    FPoint              (float, float, float, int);
    FPoint              (float, float);
    FPoint              (int , int, int);
    FPoint              ();
    void move_object();
    void plot_zone();
    int property();
    void move_object(int);
  int intersect(FPoint, FPoint, FPoint);
  int ccw(FPoint, FPoint, FPoint);
    /*
     *    destructor
     */
    ~FPoint             ();
    float&   operator[]  (int);
    FPoint   operator= (FPoint);

    friend ostream& operator<<(ostream& , FPoint&);
    void plot();
  };

#endif /* FPOINT_H */




::::::::::::::
move_object.h
::::::::::::::
#ifndef _MOVE_OBJECT_H
#define _MOVE_OBJECT_H
void move_object(int);
#endif
::::::::::::::
object.h
::::::::::::::
#ifndef _H_OBJECT
#define _H_OBJECT

#include <fstream.h>

#include <limits.h>
#include <fpoint.h>
#include <polygon.h>
class Object
{
 private:
  int n, id, ht, color;
  FPoint *p;
  float plate;
  float z1, z2;
  int point, poly, act, ob_flag;

 public:
  Object();
  Object(FPoint a, FPoint b, int id_num, int pt, int pl);
  Object(float x1, float y1, float x2, float y2, int id_num, int pt, int pl,
float size);
  ~Object() {};
  FPoint get_vertex(int);
  float get_slope();
  float get_aslope();
  float get_dist();
  int get_flag(){ return ob_flag;}
  void toggle_flag();
  int get_id(){ return id;}
   void action(int, int, float);
  void update(FPoint t);
  void    plot();

  void    clear(){ delete this;}
  void    print3d();
  void    print3d_ab();
};



#endif /* OBJECT_H */
::::::::::::::
polygon.h
::::::::::::::
#ifndef _H_POLYGON
#define _H_POLYGON

#include <fstream.h>

#include <limits.h>
#include <fpoint.h>
#include "object.h"

class Polygon
{
 private:
  int n, id, ht, color, attraction;
  float area, area_ratio, long_side;
  FPoint *p;
  void    calc_value();
  //int point_influence[25];
  //void clear_point_buf();
  float path_x;
  float path_y;
 public:
  float length, width, value_ratio;
  int polygon_found;
  FPoint min_xy;
  Polygon() {n = 0; p = NULL; id = 0; length = 0.; width = 0.; polygon_found =
0;}
  Polygon(int m) {n = m; p = new FPoint[m+1];}
  Polygon(int m, float* xpoint, float* ypoint, int pid, int col, int height);
  Polygon(int m, float* xpoint, float* ypoint);
  Polygon(char a, int ident);
  Polygon(FPoint x1, FPoint y1, FPoint x2, FPoint y2);
  ~Polygon() {};
  void move_object(int); // hmmmm
    void action(int, int, float);

  void    set_vertex(int, FPoint);
  FPoint  get_vertex(int);
  int     get_nvert(){ return n;}
  int     get_attraction(){ return attraction;}
  void    set_id(int a){ id = a;}
  int     get_id(){ return id;}
  int     get_ht(){ return ht;}
  int     get_color(){ return color;}
  void    display();
  void    print3d();
  int   property();
  int   path(FPoint pnt1);
  void    calc_min_xy();
  int     inside(FPoint t);                           /*determines if t is
inside the polygon*/
    int     enclose(Polygon& a);                        /*determines if a is
entirely inside the ply*/
  int     overlap(Polygon& a);                        /*determines if two
polygons overlap*/
  void    move_object(int id, float new_x, float new_y);    /*moves the upper
left hand corner
                                                        of the bounding box to a
new x and y*/
  void    rotate_object(int id, float about_x, float about_y, float dtheta);
                                                      /*rotates a polygon about
x and y thru theta*/

  friend  ofstream& operator<<(ofstream&, Polygon*);  /*outputs polygon to
file*/
  int intersect(FPoint, FPoint, FPoint, FPoint);
  int ccw(FPoint, FPoint, FPoint);
  friend  float angle(FPoint, FPoint);                /*calculates the angle ccw
from the axis
                                                        pointing down*/
  friend ostream& operator<<(ostream& , Polygon*);

  void    plot();                                     /*calls tcl command to
plot the polygon*/
  void    plot3d();
//  void         print() {};
  void    clear(){ delete this;}
//  int          match(Polygon* ){ cout << "polygon::match() NADA" << endl; };

};

#endif




::::::::::::::
random.h
::::::::::::::
/*******************************************************************
*                                                                  *
*                    RANDOM.H                                      *
*                                                                  *
*******************************************************************/
#ifndef RANDOM_H
#define RANDOM_H
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

class Random
{
 public:
  long idum;
  Random(long num)
    { idum = num;}
float ran1()
{
        int j;
        long k;
        static long iy=0;
        static long iv[NTAB];
        float temp;

        if (idum <= 0 || !iy) {
                if (-(idum) < 1) idum=1;
                else idum = -(idum);
                for (j=NTAB+7;j>=0;j--) {
                        k=(idum)/IQ;
                        idum=IA*(idum-k*IQ)-IR*k;
                        if (idum < 0) idum += IM;
                        if (j < NTAB) iv[j] = idum;
                }
                iy=iv[0];
```

```
        }
        k=(idum)/IQ;
        idum=IA*(idum-k*IQ)-IR*k;
        if (idum < 0) idum += IM;
        j=iy/NDIV;
        iy=iv[j];
        iv[j] = idum;
        if ((temp=AM*iy) > RNMX) return RNMX;
        else return temp;
      }
};

#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef NTAB
#undef NDIV
#undef EPS
#undef RNMX

#endif /* RANDOM_H */
::::::::::::::
register_C_with_tcl.h
::::::::::::::
#ifndef _H_CCOM
#define _H_CCOM

void register_C_with_tcl(void);

#endif /* _H_CCOM */
::::::::::::::
report.h
::::::::::::::
#ifndef _REPORT_H
#define _REPORT_H
void report();
#endif
::::::::::::::
say_hello.h
::::::::::::::
#ifndef _SAY_HELLO_H
#define _SAY_HELLO_H
void say_hello();
#endif
::::::::::::::
tcl_to_C.h
::::::::::::::
#ifndef TCL_TO_C_H_
#define  TCL_TO_C_H_
#include <tk.h>
#include <tcltkui.h>

//void register_C_with_tcl();
//void setup_tcltk(UIParams_t *,int , char **);
extern "C" int say_hello_ui (ClientData, Tcl_Interp *, int, char**);
extern "C" int move_object_ui (ClientData, Tcl_Interp *, int, char**);
extern "C" int report_ui (ClientData, Tcl_Interp *, int, char**);
extern "C" int zone_ui (ClientData, Tcl_Interp *, int, char**);
extern "C" int add_object_ui (ClientData, Tcl_Interp *, int, char**);
extern "C" int object_ui (ClientData, Tcl_Interp *, int, char**);

int add_polygon_ui (ClientData, Tcl_Interp *, int, char**);
int move_polygon_ui (ClientData, Tcl_Interp *, int, char**);
int rotate_polygon_ui (ClientData, Tcl_Interp *, int, char**);
int plot_polygon_ui (ClientData, Tcl_Interp *, int, char**);
int plot_circle_ui (ClientData, Tcl_Interp *, int, char**);
int add_point_ui (ClientData, Tcl_Interp *, int, char**);
int add_line_ui (ClientData, Tcl_Interp *, int, char**);
int move_object_number_ui (ClientData, Tcl_Interp *, int, char**);
//int report_ui(ClientData, Tcl_Interp *, int, char**);
//int say_hello_ui (ClientData, Tcl_Interp *, char**);
//extern "C" int say_hello_ui (ClientData, Tcl_Interp *, char**);
//int get_clock          (ClientData, Tcl_Interp *, int, char**);
#endif
::::::::::::::
tcltkui.h
::::::::::::::
#ifndef _H_TCLUI
#define _H_TCLUI
#include <tk.h>
#include <tcltkui.h>

typedef struct uiparams
{
  Tk_Window      w;
  Tk_Window      cwin;
  Pixmap         pixmap;
  Pixmap         buff;
  Tcl_Interp *   interp;
  int            x, y;
  double         blahl;
  Tcl_DString    buffer;
  int            tty;
  char           iname[32];
} UIParams_t, * UIParams_pt;

extern "C"
{

void setup_tcltk  ( UIParams_pt, int, char **);
void px_shell     ( ClientData,  int);
void px_xnotify   ( ClientData, XEvent *);
void wait2map     ( ClientData);
}

#endif /* _H_TCLUI */
::::::::::::::
add_object.C
::::::::::::::
#include <stdio.h>
#include <stdlib.h>
#include <tcltkui.h>

#include "random.h"
#include "fpoint.h"
#include "polygon.h"
#include "move_object.h"

extern FPoint* Point[];
extern Polygon* Poly[];
extern Random RAND_x;
extern int Max_points;
extern int Point_num;
extern int Poly_num;
extern Object_num;
void add_object(int num)
{
  if (num == 0) {
    cout << "adding poly" << endl;
    Poly[Poly_num] = new Polygon('a', Object_num);
    Poly[Poly_num]->rotate_object(1, Poly[Poly_num]->get_vertex(1).x,
Poly[Poly_num]->get_vertex(1).y, (int)(RAND_x.ran1()*500)%360);
    Poly[Poly_num]->plot();
    Poly_num++;
    Object_num++;
  }
  if (num == 1) {
    Point[Point_num] = new
FPoint(RAND_x.ran1()*500,RAND_x.ran1()*500,RAND_x.ran1()*100, Object_num);
    Point[Point_num]->plot();
    Point_num++;
    Object_num++;
  }
}


::::::::::::::
agent.C
::::::::::::::
#include <tcl.h>
#include <tk.h>
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <string.h>
#include <time.h>
#include <signal.h>

#include <tcltkui.h>
#include <tcl_to_C.h>
#include "register_C_with_tcl.h"

#include "fpoint.h"
#include "polygon.h"
#include "object.h"
#include "random.h"


// Global Variables

Random RAND_x(-4);

int Circle_offset = 5;
int Max_points;
int Point_num = 0;
int Poly_num = 0;
int Object_num = 1;
int Object_line_num = 0;
int Simulation_round = 0;

int s, q, r;

UIParams_t hello_ui;
FPoint *Point[500];
Polygon *Poly[500];
Object *Object_line[1000];
FPoint *Glob_center_point[20];
int Glob_center_point_num;

int main(int argc, char ** argv)
{

  extern FPoint *Point[500];
  extern Polygon *Poly[500];
  extern Object *Object_line[1000];
  extern int Max_points;
  extern int Point_num;
  extern int Poly_num;
  extern int Object_num;
  extern int Object_line_num;
  extern FPoint *Glob_center_point[20];
  extern int Glob_center_point_num;
  char tclcommand[200], points[50];
  extern Simulation_round;
  float x1;
  char start[50];
  s = r = q = 0;

  setup_tcltk(&hello_ui, argc, argv);
  register_C_with_tcl();

  char name[20];
  int rv, Max_points_points, Max_points_polys;
  clock_t begin_time, end_time;
  clock();
  sprintf(start,"rm temp.lsp");
       system(start);
  rv = Tcl_Eval(hello_ui.interp, "source  dtrt.tcl");
  if (rv != TCL_OK) {
    fprintf(stdout, "Exiting... cannot find tcl startup file: dtrt.tcl\n");
    exit(1);
   }
  rv = Tcl_Eval(hello_ui.interp, "source geometry.tcl");
  rv = Tcl_Eval(hello_ui.interp, "make_menu");
  int i, N, mv;
  int j;
  Tcl_LinkVar(hello_ui.interp, "numb", (char *) &Object_num, TCL_LINK_INT);
  cout << "Enter a number for generated points (<500): ";
  cin  >> Max_points_points;
  cout << endl;
  cout << "Enter a number for generated polys (<500): ";
  cin  >> Max_points_polys;
  cout << endl;
  cout << "Enter a number for centers (< 20): ";
  cin  >> Glob_center_point_num;
  cout << endl;
  end_time = clock();
  cout << "Number of points#  " << Max_points_points << endl;
  cout  << "Number of polys#  " << Max_points_polys << endl;
  cout << "seed#" << end_time<< endl;
  for (int rand_time=0; rand_time <=end_time;rand_time++){
    RAND_x.ran1();
  }
  Max_points = Max_points_polys+Max_points_points;
  for (i = 0; i < Glob_center_point_num; i++)
    Glob_center_point[i] = new FPoint();
  for (i = 0; i < Max_points_points; i++){
//    Point[Point_num] = new
FPoint(RAND_x.ran1()*500,RAND_x.ran1()*500,RAND_x.ran1()*100, Object_num);
 Point[Point_num] = new
FPoint(RAND_x.ran1()*50,RAND_x.ran1()*50,RAND_x.ran1()*100, Object_num);
    Point[Point_num]->plot();
       cout << "Pt #" << *(Point[Point_num]);
    Point_num++;
```

```
    Object_num++;
  }
  for (i = 0; i < Max_points_polys; i++){
    Poly[ Poly_num] = new Polygon('a', Object_num);
    Poly[ Poly_num] ->rotate_object(1, Poly[ Poly_num] ->get_vertex(1).x,
Poly[ Poly_num] ->get_vertex(1).y, (int)(RAND_x.ran1()*500)%360);
    Poly[ Poly_num] ->plot();
    Poly_num++;
    Object_num++;
  }

  int counter = 0;

  int inner=0;
  end_time = clock();

  //cout << "Time taken in secs " << clock()/CLOCKS_PER_SEC << endl;
  //cout << "Time taken in microsecs " << end_time << endl;

  Tk_MainLoop();

  cout << "Simulation rounds# " << Simulation_round << endl;
//  cout << "Number of points:  " << Max_points_points << endl;
//  cout  << "Number of polys:  " << Max_points_polys << endl;
  for (i = 0; i< Glob_center_point_num; i++)
    cout << "Central # " << *Glob_center_point[ i] << endl;
  cout << "After Tk_Main Loop" << endl;
  Tcl_DeleteInterp(hello_ui.interp);
  cout << "After Tcl_DeleterI" << endl;
  Tcl_DStringFree(&(hello_ui.buffer));
  cout << "After Tcl_DString" << endl;
  exit(0);
}
::::::::::::::
fpoint.C
::::::::::::::
/******************************************************************
*                                                                 *
*                 FPOINT.C                                        *
*                                                                 *
******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <tcltkui.h>

#include "fpoint.h"
#include "random.h"
#include "object.h"

extern Random RAND_x;
//extern Random RAND_y;
extern UIParams_t hello_ui;
extern int Circle_offset;
extern int Max_points;
extern int Object_num;
extern Object* Object_line[ ];
extern Object_line_num;
extern int Simulation_round;
FPoint::FPoint(FPoint& a)
{
    x = a[ X] ;
    y = a[ Y] ;
    move_factor = a.move_factor;
    if (move_factor > 200) move_factor = 10;
    if (move_factor < 0)   move_factor = 10;
    field = a.field;
    //cout << move_factor <<endl;
    prop = a.prop;
    width = a.width;
    ob_flag = 0;
    ob_id = 0;
}
FPoint::FPoint(float x1, float y1)
{

  x = x1;
  y = y1;
  move_factor = RAND_x.ran1()*100;
  field = RAND_x.ran1()*50;

  if (move_factor > 75)
    color = 1;
  else if (move_factor > 50)
    color = 4;
  else if (move_factor > 25)
    color = 2;
  else if (move_factor > 0)
    color = 3;
  prop = ((int)(RAND_x.ran1()*100))%3;
  width = RAND_x.ran1()*30;
  ob_flag = 0;
  ob_id = 0;
}
FPoint::FPoint(float x1, float y1, float m, int i)
{
  x = x1;
  y = y1;
  move_factor = m;
  field = RAND_x.ran1()*50;
   if (move_factor > 50)
    color = 1;
  else if (move_factor > 25)
    color = 2;
  else if (move_factor > 0)
    color = 3;
  id = i;
  prop = ((int)(RAND_x.ran1()*100))%3;
  width = RAND_x.ran1()*30;
  ob_flag = 0;
  ob_id = 0;
}

FPoint::FPoint(int x1, int y1, int m )
{
  x = x1;
  y = y1;
  move_factor = m;
    field = RAND_x.ran1()*50;
  prop = ((int)(RAND_x.ran1()*100))%3;
  width = RAND_x.ran1()*30;
  ob_flag = 0;
  ob_id = 0;
}

FPoint::FPoint()
{
  x =RAND_x.ran1()*500;
  y =RAND_x.ran1()*500;
  move_factor = RAND_x.ran1()*100;
  field = RAND_x.ran1()*50;
  if (move_factor > 50)
    color = 1;
  else if (move_factor > 25)
    color = 2;
  else if (move_factor > 0)
    color = 3;
  id = Object_num;
  //cout << "info id in creation :" << id << endl;
  prop = ((int)(RAND_x.ran1()*100))%3;
  width = RAND_x.ran1()*30;
  ob_flag = 0;
  ob_id = 0;
}

FPoint::~FPoint()
{
}

void FPoint::move_object()
{
x=RAND_x.ran1()*500;
y=RAND_x.ran1()*500;
}

void FPoint::move_object(int im)
{
  char tclcommand[ 200] , points[ 100] ;
  FILE *file;char lisp2[ 500] ;
  int simpht;
  if (Simulation_round > 100) color = 2;
  if (color == 3) {
  //  cout << "move_type: " << color << endl;
  float xx=0, yy=0;
  if (im == 0){
    y = y+Circle_offset/2;
    yy = 2;
  }
  if (im == 1){
    x = x+Circle_offset/2;
    xx = 2;
  }
  if (im == 2){
     y=y-Circle_offset/2;
     yy = -2;
   }
  if (im == 3){
    x=x-Circle_offset/2;
    xx = -2;
  }
  sprintf(tclcommand, "$c move %d %f %f", id, xx, yy);
    //cout << tclcommand << endl;
    int rv = Tcl_Eval(hello_ui.interp,tclcommand);
    if (rv != TCL_OK)
      {
       printf("error in fpoint.C  move type 1\n");
       exit(-1);
      }
   // plot();
  }
  if (color == 2) {
    //cout << "move_type: " << color << endl;
    float x1, y1, tempx, tempy;
    int k, rv;
    int count = 0;
    int j= ((int)(RAND_x.ran1()*100))%4;
    tempx = x;
    tempy = y;
    float xx, yy;
    if (j == 0){
      x = x+(RAND_x.ran1()*100)/2;
      y = y+(RAND_x.ran1()*100)/2;
      xx = x - tempx;
      yy = y - tempy;
    }
    if (j == 1){
      x = x+( RAND_x.ran1()*100)/2;
      y = y-( RAND_x.ran1()*100)/2;
      xx = x - tempx;
      yy = -(tempy - y);
    }
    if (j == 2){
      x = x-( RAND_x.ran1()*100)/2;
      y = y-( RAND_x.ran1()*100)/2;
      xx = -(tempx - x);
      yy = -(tempy - y);
    }
    if (j == 3) {
      x = x-( RAND_x.ran1()*100)/2;
      y = y+( RAND_x.ran1()*100)/2;
      xx = -(tempx - x);
      yy = y - tempy;
    }
    FPoint *ptemp = new FPoint(tempx, tempy);
    for (int test=0; test<Object_line_num; test++) {
       //cout << "anything here:: " << Object_line_num << endl;
       if (intersect (*ptemp, Object_line[ test] ->get_vertex(1),
Object_line[ test] ->get_vertex(2))) count++;
      if ((count > 0) && (ob_flag == 0)){
       //cout << "doing the dirty" << endl;
       //ob_slope = Object_line[ test] ->get_slope();
       //ob_x = Object_line[ test] ->get_dist();
       ob_id = test;
       Object_line[ test] ->toggle_flag();
       //sprintf(tclcommand, "$c delete ob%d", test);
       //rv = Tcl_Eval(hello_ui.interp,tclcommand);
       ob_flag = 1;
      }
      else if (count > 0){
       int test1 = (int)(RAND_x.ran1()*100)%3;
       simpht = ((int)(RAND_x.ran1()*100))%10;
       if (test1 == 0){
         cout << "V1 ";
         Object_line[ ob_id] ->update(Object_line[ test] ->get_vertex(1));
         Object_line[ ob_id] ->print3d_ab();
         sprintf(tclcommand, "add_line_special_ui { %f %f %f %f %f %f 3 2
curve}", \
                 (Object_line[ ob_id] ->get_vertex(2))[ X] , (Object_line[ ob_id] -
>get_vertex(2))[ Y] , \
                 (Object_line[ ob_id] ->get_vertex(1))[ X] , (Object_line[ ob_id] -
>get_vertex(1))[ Y] , \
                 (Object_line[ test] ->get_vertex(2))[ X] , (Object_line[ test] -
>get_vertex(2))[ Y] );
         cout << tclcommand << endl;
         rv = Tcl_Eval(hello_ui.interp,tclcommand);
```

```
          if (Object_line[ ob_id] ->get_slope() != Object_line[ test] -
>get_slope()){
            cout << " !=noslope ";

            sprintf(lisp2, " (command \"layer\" \"s\" \"curve\"\"\") (command
\"pline\" '(%f %f 0) '(%f %f 0) '(%f %f 0) \"\") (completer %d)\n",
 (Object_line[ ob_id] ->get_vertex(2))[ X] , \
                        (Object_line[ ob_id] ->get_vertex(2))[ Y] , \
                        (Object_line[ ob_id] ->get_vertex(1))[ X] , (Object_line[ ob_id] -
>get_vertex(1))[ Y] , \
                        (Object_line[ test] ->get_vertex(2))[ X] , (Object_line[ test] -
>get_vertex(2))[ Y] , simpht);

            /*  else {
               sprintf(lisp2, " (command \"pline\" '(%f %f 0) '(%f %f 0) '(%f %f
0)\"\") (completer %d)\n", \
                   (Object_line[ ob_id] ->get_vertex(2))[ X] , \
                   (Object_line[ ob_id] ->get_vertex(2))[ Y] , \
                   (Object_line[ ob_id] ->get_vertex(1))[ X] , (Object_line[ ob_id] -
>get_vertex(1))[ Y] , \
                   (Object_line[ test] ->get_vertex(2))[ X] , (Object_line[ test] -
>get_vertex(2))[ Y] , simpht);
               }*/
            file = fopen("temp.lsp", "a");
            fprintf (file, "%s", lisp2);
            fclose(file);
          }
        }
        if (test1 == 1){
          cout << "V2 ";
          Object_line[ ob_id] ->update(Object_line[ test] ->get_vertex(2));
          Object_line[ ob_id] ->print3d_ab();
          sprintf(tclcommand, "add_line_special_ui { %f %f %f %f %f %f 3 2
curve}", \
                 (Object_line[ ob_id] ->get_vertex(1))[ X] , (Object_line[ ob_id] -
>get_vertex(1))[ Y] , \
                 (Object_line[ ob_id] ->get_vertex(2))[ X] , (Object_line[ ob_id] -
>get_vertex(2))[ Y] , \
                 (Object_line[ test] ->get_vertex(1))[ X] , (Object_line[ test] -
>get_vertex(1))[ Y] );
          cout << tclcommand << endl;
          rv = Tcl_Eval(hello_ui.interp,tclcommand);
            if (Object_line[ ob_id] ->get_slope() != Object_line[ test] -
>get_slope()){
              cout << " !=noslope ";

              sprintf(lisp2, " (command \"layer\" \"s\" \"curve\"\"\") (command
\"pline\" '(%f %f 0)  '(%f %f 0) '(%f %f 0) \"\") (completer %d)\n",
 (Object_line[ ob_id] ->get_vertex(1))[ X] , \
                          (Object_line[ ob_id] ->get_vertex(1))[ Y] , \
                          (Object_line[ ob_id] ->get_vertex(2))[ X] ,
 (Object_line[ ob_id] ->get_vertex(2))[ Y] , \
                          (Object_line[ test] ->get_vertex(1))[ X] , (Object_line[ test] -
>get_vertex(1))[ Y] , simpht);

              file = fopen("temp.lsp", "a");
              fprintf (file, "%s", lisp2);
              fclose(file);
            }

        }
        if (test1 == 2){
          cout << "VM ";
          FPoint *ptemp1 = new FPoint( \
               ((Object_line[ test] ->get_vertex(1))[ X] +(Object_line[ test] -
>get_vertex(2))[ X] )/2, \
               ((Object_line[ test] ->get_vertex(1))[ Y] +(Object_line[ test] -
>get_vertex(2))[ Y] )/2);
          Object_line[ ob_id] ->update(*ptemp1);
        }
        ob_flag = 0;
        Object_line[ ob_id] ->toggle_flag();
        }
       count = 0;
      }
     sprintf(tclcommand, "add_line_ui {");
     sprintf(points, "%f %f %f %f 4 1 fp%d}", x, y, tempx, tempy, id);
     strcat(tclcommand, points);
//     int test = ((int)(RAND_x.ranl()*100))%2;
//     if (test == 1) {
//       rv = Tcl_Eval(hello_ui.interp,tclcommand);
//     }
//     if (rv != TCL_OK)
//       {
//      printf("error in fpoint.C  movetype 2 add line\n");
//      cout << tclcommand << endl;
        //exit(-1);
//       }

    sprintf(tclcommand, "$c move %d %f %f", id, xx, yy);

     //cout << tclcommand << endl;
     rv = Tcl_Eval(hello_ui.interp,tclcommand);
     if (rv != TCL_OK)
       {
        printf("error in fpoint.C  move type 2\n");
        exit(-1);
       }
//  plot();
  }
  if (color == 1) {
     //cout << "move_type: " << color << endl;
     float xx, yy;
     float tempx = RAND_x.ranl()*500; float tempy = RAND_x.ranl()*500;
     //cout << "x,y:" << x << "," << y << endl;
     if (tempx < x) xx = tempx - x;
     if (tempx > x) xx = tempx - x;
     if (tempy < y) yy = tempy - y;
     if (tempy > y) yy = tempy - y;
     x = tempx;
     y = tempy;
     char tclcommand[ 200] , points[ 50] ;
     sprintf(tclcommand, "$c move %d %f %f", id, xx, yy);
   //cout << tclcommand << endl;
     int rv = Tcl_Eval(hello_ui.interp,tclcommand);
     if (rv != TCL_OK)
       {
        printf("error in fpoint.C movetype 1 \n");
        exit(-1);
       }
     //cout << "new:" << tempx << "," << tempy << endl;
 }
  //cout << "Id, x,y:" << id << ":" << x << "," << y << endl;
        if (x > 500) x = x -500;
        if (x < 0) x = x +500;
        if (y > 500) y =y - 500;
        if (y < 0)  y = y + 500;
     //cout << tclcommand << endl;
     //cout << tclcommand << endl;
}

float& FPoint::operator[] (int i)
{
  if(i == 0)
    return x;
  if(i == 1)
    return y;
  else
    cout << "error in array bound" << i << "out of range"<<endl;
}

FPoint FPoint::operator=(FPoint i)
{
  this->x = i.x;
  this->y = i.y;
  this->field = i.field;
  this->move_factor = i.move_factor;
  this->prop = i.prop;
}

void FPoint::plot()
{
  char tclcommand[ 200] , points[ 100] ;
  float x1,y1;
  sprintf(tclcommand, "add_point_ui {");
  sprintf(points, "%f %f %f %f %d %d %f}", x, y, x+Circle_offset,
y+Circle_offset, color, id, field);
  strcat(tclcommand, points);
 // cout << tclcommand << endl;
  int rv = Tcl_Eval(hello_ui.interp,tclcommand);
  if (rv != TCL_OK)
    {
      printf("error in fpoint.C plot() \n");
      exit(-1);
    }
}

void FPoint::plot_zone()
{
 char tclcommand[ 200] , points[ 100] ;
float x1,y1;
  sprintf(tclcommand, "plot_zone {");
 sprintf(points, "%f %f %f %f %d %d %f}", x, y, x+Circle_offset,
y+Circle_offset, color, id, field);
  strcat(tclcommand, points);
  int rv = Tcl_Eval(hello_ui.interp,tclcommand);
  if (rv != TCL_OK)
    {
      printf("error in fpoint.C plot_zone() \n");
      exit(-1);
    }
}

int FPoint::property() {
 // if (prop == 0) cout << "grow" << endl;
 // if (prop == 1) cout << "decay" << endl;
 // if (prop == 2) cout << "transform" << endl;
 return prop;
}

ostream& operator<<(ostream& s, FPoint& a)
{
  return s << "    pt " << a.id << " " << a[ X] << " " << a[ Y] << " " <<
a.move_factor << " " << a.color << " " << a.prop << " " << a.field << " " <<
a.width << endl;
}


int FPoint::intersect(FPoint p1, FPoint p2, FPoint p3)
{
  float temp_x = x;
  float temp_y = y;
    FPoint *p0 = new FPoint(temp_x, temp_y);
    int j1, j2, k1, k2;

    j1 = ccw(*p0, p1, p2);
    j2 = ccw(*p0, p1, p3);
    k1 = ccw(p2, p3, *p0);
    k2 = ccw(p2, p3, p1);
    delete p0;
    return (j1*j2 <= 0) && (k1*k2 <= 0);

}


int FPoint::ccw(FPoint p0, FPoint p1, FPoint p2)
{
  float dx1, dx2, dy1, dy2;

  dx1 = p1[ X]  - p0[ X] ;
  dy1 = p1[ Y]  - p0[ Y] ;
  dx2 = p2[ X]  - p0[ X] ;
  dy2 = p2[ Y]  - p0[ Y] ;
  if (dx1*dy2 > dx2*dy1) return +1;
  if (dx1*dy2 < dx2*dy1) return -1;
  if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
  if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2)) return +1;
  return 0;
}
::::::::::::::
move_object.C
::::::::::::::
#include <stdio.h>
#include <stdlib.h>
#include <tcltkui.h>

#include "random.h"
#include "fpoint.h"
#include "object.h"
#include "polygon.h"
#include "move_object.h"

extern FPoint* Point[] ;
extern Polygon* Poly[] ;
extern Object* Object_line[] ;
extern Random RAND_x;
extern int Max_points;
extern int Point_num;
extern int Poly_num;
extern Object_num;
extern Object_line_num;
extern int Simulation_round;
extern FPoint *Glob_center_point[] ;
//extern int Glob_center_point_num;
void move_object(int num)
{
  int i, j, k, kk, ii, jj ,counter, inner, test, path_val;
  float testprop;
```

```
  for (i = 1; i <= num; i++){
    Simulation_round++;
    //cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< num : " << i << endl;
    for (k = 0; k < Point_num; k++) {
      j = ((int)(RAND_x.ran1()*100))%4;
      Point[k]->move_object(j);
    }
    for (kk = 0; kk < Poly_num; kk++) {
    //  cout << "poly moving toward: " << Poly[kk]->get_attraction() << endl;
      path_val = Poly[kk]->path(*Glob_center_point[Poly[kk]->get_attraction()]);
      //cout << "Path Val=" << path_val << ";";
      Poly[kk]->move_object(path_val);
    }
    inner=0;
    if (Simulation_round < 200)
      for (ii = 0; ii < Poly_num; ii++) {
       for (jj = 0; jj < Point_num; jj++) {
         test = Poly[ii]->inside(*Point[jj]);
         if (test == 1){
           inner++;
           //testprop = (float)Poly[ii]->property() + (float)Point[jj]-
>property()/3;
           //cout << testprop << " testy "<< endl;

           Poly[ii]->action(jj, ii, (float)Point[jj]->property());
           //cout << "Line #:"<< Object_line_num <<endl;

         }
       }
      }
    /*
      for (ii = 0; ii < Poly_num; ii++) {
      for (jj = 0; jj < Point_num; jj++) {
      test = Poly[ii]->intersect
      */
  }
}


:::::::::::::
object.C
:::::::::::::
#include "object.h"
#include "polygon.h"
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <tcltkui.h>
#include "fpoint.h"
#include "random.h"
#include "fstream.h"

extern UIParams_t hello_ui;
extern Random RAND_x;
extern int Object_num;

 Object::Object(){
   n = 2;
   p = new FPoint[n+1];
   act = (int)(RAND_x.ran1()*100)%3;
   int temp = (int)(RAND_x.ran1()*100)%3;
   if (temp == 0)
     plate = (int)(RAND_x.ran1()*100)%20+1;
   else
     plate = (int)(RAND_x.ran1()*100)%5+1;
       ob_flag = 0;
   ht = (((int)(RAND_x.ran1()*100))%25)+1;
 }

 Object::Object(FPoint a, FPoint b, int id_num, int pt, int pl){
   n = 2;
   p = new FPoint[n+1];
   p[1] = a;
   p[2] = b;
   id = id_num;
   point = pt;
   poly = pl;
   ht = (((int)(RAND_x.ran1()*100))%25)+1;
   act = (int)(RAND_x.ran1()*100)%3;
   ob_flag = 0;

 }
 Object::Object(float x1, float y1, float x2, float y2, int id_num, int pt, int
pl, float size){
   n = 2;
 //cout << "maker:" << x1 << "," << y1 << " " << x2 << "," <<  y2 << endl;
   p = new FPoint[n+1];
   p[1][X] = x1;
   p[1][Y] = y1;
   p[2][X] = x2;
   p[2][Y] = y2;
   id = id_num;
   point = pt;
   poly = pl;
   int temp = (int)(RAND_x.ran1()*100)%3;
   if (temp == 0)
     plate = (int)(RAND_x.ran1()*100)%20+1;
   else
     plate = (int)(RAND_x.ran1()*100)%5+1;

   if (plate == 0)
     plate = 1;
   ht = (((int)(RAND_x.ran1()*100))%25)+1;
   act = (int)size;
   ob_flag = 0;
   //cout << "baker:" <<     p[1][X] << "," << p[1][Y] << " " << p[2][X] << ","
<<  p[2][Y] << endl;
 }

FPoint Object::get_vertex(int i)
{
  if (i >= 0 && i <= n)
    return p[i];
  else
  {
    cout<< "Cannot get polygon vertex " << i << " out of range\n";
    return p[0] ;
   }
}

float Object::get_slope()
{
  float test;
  if (p[1][X] == p[2][X])
    test = 0;
  else
    test = (p[2][Y] - p[1][Y]) / (p[2][X] - p[1][X]);
  return test;
}
float Object::get_aslope()
{
  float test, test1;
  test = get_slope();
  if (test == 0) return 1;
  else
  return (-(1/test));
}

float Object::get_dist()
{
  return (p[1][X] - p[2][X]);
}
void Object::toggle_flag()
{
  if (ob_flag == 0) ob_flag = 1;
  else {
    ob_flag = 0;
  }

}

void Object::update(FPoint t)
{
  float new_x  = t[X] - p[1][X];
  float new_y =  t[Y] - p[1][Y];
  p[1][X] = p[1][X] + new_x;
  p[1][Y] = p[1][Y] + new_y;
  p[2][X] = p[2][X] + new_x;
  p[2][Y] = p[2][Y] + new_y;
  plot();

}
void Object::plot()
{
  char tclcommand[200], points[100];
  sprintf(tclcommand, "add_line_object_ui {");
   sprintf(points, "%f %f %f %f %d %d ob}", p[1][X], p[1][Y], p[2][X],
p[2][Y], act, (int)plate);
  strcat(tclcommand, points);
 // cout << tclcommand << endl;
  int rv = Tcl_Eval(hello_ui.interp,tclcommand);
  if (rv != TCL_OK)
    {
      printf("error in object.C plot \n");
      exit(-1);
    }
}


void Object::print3d()
{
  FILE *file;
  char lispcommand[500];
  char lisp2[500];
  char lisp3[200];
  char lisp1[200];
  float newx, newx1, newy, newy1, slope, slope1;
  int play = ((int)(RAND_x.ran1()*100))%20;
  int taper =  ((int)(RAND_x.ran1()*100))%20;
  int willtaper = ((int)(RAND_x.ran1()*100))%20;
  int moveup = ((int)(RAND_x.ran1()*100))%20;
  int moveupdist = ((int)(RAND_x.ran1()*100))%20;
    if (p[1][X] != p[2][X]){
      slope = ((p[1][Y] -p[2][Y])/ (p[1][X] - p[2][X]));
      slope1 = -(1/slope);
    }
    else{
      slope = 0;
      slope1 = 1;
    }
    if (slope == 0)
      slope1 = 1;

    newx = (p[2][X] + plate);
    newy = p[2][Y] - (slope1 * (p[2][X] - newx));
    newx1 = p[1][X] + plate;
    newy1 = p[1][Y] - (slope1 * (p[1][X] - newx1));

    /* bAse */
    sprintf(lisp2, "(command \"pline\" '(%f %f 0) '(%f %f 0) '(%f %f 0) '(%f %f
0) \"c\")", \
           p[1][X], p[1][Y],\
           p[2][X],p[2][Y],\
           newx, newy, \
           newx1, newy1);
  cout << "act:" << act << " ";
  if (act == 0) {
    if (willtaper > 5)
    sprintf(lispcommand, "(command \"extrude\" \"l\" \"\" %d \"\")", ht);
    else
         sprintf(lispcommand, "(command \"extrude\" \"l\" \"\" %d %d)", ht,
taper);
  }
  if (act == 1) {
      if (willtaper > 5)
    sprintf(lispcommand, "(command \"layer\" \"S\" \"sub\" \"\")(command
\"extrude\" \"l\" \"\" %d \"\") (command \"layer\" \"S\" \"0\" \"\")", -ht);
      else
         sprintf(lispcommand, "(command \"layer\" \"S\" \"sub\" \"\")(command
\"extrude\" \"l\" \"\" %d %d) (command \"layer\" \"S\" \"0\" \"\")", -ht,
taper);
      if (moveup < 5){
       cout << "MV'NG OBJ->" << endl;
       sprintf(lisp3, "(command \"move\" \"l\" \"\" '(0 0 0) '(0 0 %d))",
moveupdist);
        strcat(lispcommand, lisp3);
      }
    }
    if (act == 2) {
       sprintf(lispcommand, "(command \"extrude\" \"l\" \"\" %d \"\") (command
\"move\" \"l\" \"\"  '(0 0 0) '(0 0 %d))", ht, play);
    }
  file = fopen("temp.lsp", "a");


  //fprintf(file, "%s %s %s %s\n", lisp3, lisp2, lisp1, lispcommand);
    fprintf (file, "%s %s\n", lisp2, lispcommand);
  fclose(file);

}


void Object::print3d_ab()
{
  FILE *file;
  char lispcommand[500];
  char lisp2[500];
```

```
  char lisp3[200];
  char lisp1[200];
  float newx, newx1, newy, newy1, slope, slope1;
  int play = ((int)(RAND_x.ran1()*100))%20;
  int taper =  ((int)(RAND_x.ran1()*100))%20;
  int willtaper = ((int)(RAND_x.ran1()*100))%20;
  int moveup = ((int)(RAND_x.ran1()*100))%20;
  int moveupdist = ((int)(RAND_x.ran1()*100))%20;
  float thick = (float)(((int)(RAND_x.ran1()*100))%20);

  if (p[1][X]  != p[2][X]){
    slope = ((p[1][Y] -p[2][Y])/ (p[1][X] - p[2][X]));
    slope1 = -(1/slope);
  }
  else{
      slope = 0;
      slope1 = 1;
    }
  if (slope == 0)
    slope1 = 1;



  newx = (p[2][X] + plate);
  newy = p[2][Y] - (slope1 * (p[2][X] - newx));
  newx1 = p[1][X] + plate;
  newy1 = p[1][Y] - (slope1 * (p[1][X] - newx1));

  file = fopen("temp.lsp", "a");

  int orient =  ((int)(RAND_x.ran1()*100))%2;
  int numm =  ((int)(RAND_x.ran1()*100))%5;
  int solid = ((int)(RAND_x.ran1()*100))%2;
  if (thick < 5) thick = 0.5;

  sprintf (lisp3," (abstractor '(%f %f 0) '(%f %f 0) %d %d %d %f %d)", p[1][X],
p[1][Y], p[2][X],p[2][Y], orient, numm, ht, thick, solid);
  orient =  ((int)(RAND_x.ran1()*100))%2;
  numm =  ((int)(RAND_x.ran1()*100))%10;
  solid = ((int)(RAND_x.ran1()*100))%2;

  sprintf (lisp2," (abstractor '(%f %f 0) '(%f %f 0) %d %d %d %f %d)",
p[2][X],p[2][Y], newx, newy, orient, numm, ht, thick, solid);
  orient =  ((int)(RAND_x.ran1()*100))%2;
  numm =  ((int)(RAND_x.ran1()*100))%5;
  solid = ((int)(RAND_x.ran1()*100))%2;

  sprintf (lispcommand," (abstractor '(%f %f 0) '(%f %f 0) %d %d %d %f %d)",
newx, newy, newx1, newy1, orient, numm, ht, thick, solid);
  orient =  ((int)(RAND_x.ran1()*100))%2;
  numm =  ((int)(RAND_x.ran1()*100))%5;
  solid = ((int)(RAND_x.ran1()*100))%2;

  sprintf (lisp1," (abstractor '(%f %f 0) '(%f %f 0) %d %d %d %f %d)", newx1,
newy1, p[1][X], p[1][Y], p[2][X]  , orient, numm, ht, thick, solid);
  fprintf(file, "(command \"layer\" \"s\" \"ab\" \"\") %s %s %s %s (command
\"layer\" \"s\" \"0\" \"\")\n", lisp3, lisp2, lisp1, lispcommand);
  //  fprintf (file, "%s %s\n", lisp2, lispcommand);
  fclose(file);

}
::::::::::::::
polygon.C
::::::::::::::
#include "polygon.h"
#include "object.h"
#include <math.h>
#include <string.h>
#include <tcltkui.h>
#include "fpoint.h"
#include "random.h"

extern UIParams_t hello_ui;
extern Random RAND_x;
extern int Object_num;
extern  Object* Object_line[];
extern int Object_line_num;
extern int Poly_num;
extern int Simulation_round;
extern int Glob_center_point_num;

Polygon::Polygon(int num, float* xpoint, float* ypoint, int pid, int col, int
height)
{
  n = num;
  p = new FPoint[n+1];

  for(int i = 0; i < n; i++)
    p[i] = FPoint(xpoint[i], ypoint[i]);

  id = pid;
  ht = height;
  color = col;
  polygon_found = 0;
  //  calc_value();
//      clear_point_buf();
  attraction = (((int)(RAND_x.ran1()*100))%Glob_center_point_num);
       display();
}

 Polygon::Polygon(FPoint x1, FPoint y1, FPoint x2, FPoint y2)
{
  n = 4;
  p = new FPoint[n+1];

 // p[0] = y2;
  p[1] = x1;
  p[2] = y1;
  p[3] = x2;
  p[4] = y2;
 // p[5] = x1;
  id = 0;
  color = 4;
  //clear_point_buf();
  ht = (((int)(RAND_x.ran1()*100))%10)+3;
  attraction = (((int)(RAND_x.ran1()*100))%Glob_center_point_num);
display();
}
/* CODE FOR RANDOM SHAPED POLY's
 Polygon::Polygon(char a, int ident)
{
  n = 4;
  p = new FPoint[6];

  float x_offset = RAND_x.ran1()*30;
  float y_offset = RAND_x.ran1()*30;
       float x_temp = RAND_x.ran1()*30;
       float y_temp = RAND_x.ran1()*30;
  int i = 0;
```

```
  //  for(int i = 0; i < n; i++);
  p[1] = FPoint();
  p[2] = FPoint(p[1][X]+ x_offset, p[1][Y], p[1].move_factor, 0);
  p[3] = FPoint(p[1][X]+ x_temp, p[1][Y]+y_offset, p[1].move_factor, 0);
  p[4] =  FPoint(p[1][X], p[1][Y]+y_temp, p[1].move_factor, 0);
  //sentinals---->
  p[0] = FPoint(p[1][X], p[1][Y]+y_offset, p[1].move_factor, 0);
  p[5] = FPoint(p[1][X], p[1][Y], p[1].move_factor, 0);
  id = ident;
       //cout << "ID of POLY = " << id << endl;
  color = 2;
  //clear_point_buf();
  ht = (((int)(RAND_x.ran1()*100))%10)+3;
  attraction = (((int)(RAND_x.ran1()*100))%Glob_center_point_num);
  path_x = p[1][X];
  path_y = p[1][Y];
  display();
}
*/

/* CODE FOR ORTHO POLY'S */

 Polygon::Polygon(char a, int ident)
{
  n = 4;
  p = new FPoint[6];

  float x_offset = RAND_x.ran1()*30;
  float y_offset = RAND_x.ran1()*30;
  int i = 0;
  //  for(int i = 0; i < n; i++);
  p[1] = FPoint();
  p[2] = FPoint(p[1][X]+ x_offset, p[1][Y], p[1].move_factor, 0);
  p[3] = FPoint(p[1][X]+ x_offset, p[1][Y]+y_offset, p[1].move_factor, 0);
  p[4] =  FPoint(p[1][X], p[1][Y]+y_offset, p[1].move_factor, 0);
  //sentinals---->
  p[0] = FPoint(p[1][X], p[1][Y]+y_offset, p[1].move_factor, 0);
  p[5] = FPoint(p[1][X], p[1][Y], p[1].move_factor, 0);
  id = ident;
       //cout << "ID of POLY = " << id << endl;
  color = 2;
  //clear_point_buf();
  ht = (((int)(RAND_x.ran1()*100))%10)+3;
  attraction = (((int)(RAND_x.ran1()*100))%Glob_center_point_num);
  path_x = p[1][X];
  path_y = p[1][Y];
  display();
}



void Polygon::calc_min_xy()
{
  float min_x = 10000, min_y = 10000;

  for (int i = 0; i < n; i++) {
    if (min_x > p[i][X]) min_x = p[i][X];
    if (min_y > p[i][Y]) min_y = p[i][Y];
  }
  FPoint d(min_x,min_y);
  min_xy = d;
}


void Polygon::set_vertex(int i, FPoint a)
{
  if (i >= 0 && i <= n)
    p[i] = a;
  else
    cout <<"Cannot set polygon vertex" << i <<" out of range\n" ;
}


FPoint Polygon::get_vertex(int i)
{
  if (i >= 0 && i <= n)
    return p[i];
  else
  {
    cout<< "Cannot get polygon vertex " << i << " out of range\n";
    return p[0]  ;
   }
}


int Polygon::inside(FPoint t)
{
  int i, count = 0, j = 0;

//  FPoint *point, *p0, *p1, *p2, *p3;
//  p0 = new FPoint();
//  p1 = new FPoint();
//  p2 = new FPoint();
//  p3 = new FPoint();
  FPoint *point = new FPoint(0.0,0.0);
// /afs/athena.mit.edu/course/1/1.124/www/Lectures/L13/geops/node5.html
//  *p2 = t;
//  p3 = point;

 /*
    for (i = 1; i <= n; i++)
    {
       j = i + 1
       *p0 = p[i]; cout << *p0 << *p1;
       *p1 = p[j];
     //  if (!intersect(*p0, *p1, *p2, *p3))
     //    {
     //        *p1 = p[j];
           //  j = i;
             if (intersect(*p0, *p1, *p2, *p3)) count++;

      //    }
     }*/
        if (intersect(p[1], p[2], t, *point)) count++;
        if (intersect(p[2], p[3], t, *point)) count++;
        if (intersect(p[3], p[4], t, *point)) count++;
        if (intersect(p[4], p[1], t, *point)) count++;
     /*
    for (i = 1; i <= n; i++) {
       *p0 = p[i];
       *p1 =   p[i+1];
       if (intersect(*p0, *p1, *p2, *p3)) count++;
    }  */
  delete point;
 // delete p0;
 // delete p1;
  //delete p2;
  //delete p3;
//cout << "counter of lines hit = " << count << endl;
```

```
  return count & 1;  /* Return 0 if even (=outside), 1 if odd (=inside) */
}

int Polygon::enclose(Polygon& a)
{
  int i, j = 0;
  for (i = 0; i < a.get_nvert(); i++)
    {
      if (!inside(a.get_vertex(i))) j = 1;
    }
  if (j == 0)  return 1;
  else return 0;  /* return 0 if not enclosed, 1 if enclosed */
       /*note:  a polygon that exactly touches another is not enclosed */
}


int Polygon::overlap(Polygon& a)
{
  int i, j;

  for (i = 0, j = 1; !i && j <= a.n; j++)
    i += inside(a.get_vertex(j));

  for (j = 1; !i && j <= n; j++)
    i += a.inside(p[j]);

  return i;
}


void Polygon::move_object(int id, float new_x, float new_y)
{
  float dx = min_xy[X] - new_x;
  float dy = min_xy[Y] - new_y;
  for (int i = 0; i < n; i++) {
    p[i][X] -= dx;
    p[i][Y] -= dy;
  }
  min_xy[X] = new_x;
  min_xy[Y] = new_y;
}

void Polygon::move_object(int im) // cycle through points
{
  char tclcommand[200], points[100];
 int Circle_offset = 5;
  int i;
  float xx=0, yy=0, tempx, tempy;
  tempx = p[1][X];
    tempy = p[1][Y];
  if (im == 5)
    im = ((int)(RAND_x.ran1()*100))%4;
  if (im == 0){
   // y = y+Circle_offset/2;
    yy = 2;
    for (i = 1 ; i <= n; i++) {
      p[i][Y] = p[i][Y] +Circle_offset/2;
    }
  }
  if (im == 1){
   // x = x+Circle_offset/2;
    xx = 2;
      for (i = 1 ; i <= n; i++) {
      p[i][X] = p[i][X] + Circle_offset/2;
    }
  }
  if (im == 2){
    //  y=y-Circle_offset/2;
      yy = -2;
        for (i = 1 ; i <= n; i++) {
      p[i][Y] = p[i][Y] - Circle_offset/2;
    }
    }
  if (im == 3){
   // x=x-Circle_offset/2;
    xx = -2;
      for (i = 1 ; i <= n; i++) {
      p[i][X] = p[i][X] - Circle_offset/2;
    }
  }
  sprintf(tclcommand, "$c move %d %f %f", id, xx, yy);
    //cout << tclcommand << endl;
    int rv = Tcl_Eval(hello_ui.interp,tclcommand);
    if (rv != TCL_OK)
      {
       printf("error in fpoint.C  move type 1\n");
       exit(-1);
      }


  if ((Simulation_round%20) == 0) {


    char lisp2[500];  FILE *file;
    float newx, newx1, newy, newy1, slope, slope1;

    sprintf(tclcommand, "add_line_ui {");
    sprintf(points, "%f %f %f %f 0 1 path%d}", path_x, path_y, p[1][X], p[1][Y],
id);
    strcat(tclcommand, points);

    rv = Tcl_Eval(hello_ui.interp,tclcommand);
    if (rv != TCL_OK)
      {
       printf("error in poylgon.C  movetype add line\n");
       //      cout << tclcommand << endl;
       //exit(-1);
      }


    if (path_x != p[1][X]){
      slope = ((path_y - p[1][Y])/ (path_x - p[1][X]));
      slope1 = -(1/slope);
    }
    else{
      slope = 0;
      slope1 = 1;
    }
    if (slope == 0)
      slope1 = 1;

    newx = (p[1][X] + 2);
    newy = p[1][Y] - (slope1 * (p[1][X] - newx));
    newx1 = path_x + 2;
    newy1 = path_y - (slope1 * (path_x - newx1));

    /* bAse */
    sprintf(lisp2, "(command \"layer\" \"s\" \"path\" \"\")(command \"pline\"
'(%f %f 0) '(%f %f 0) '(%f %f 0) '(%f %f 0) \"c\")(command \"extrude\" \"l\"
\"\" 100 \"\")(command \"layer\" \"s\" \"0\" \"\"\n)", \
              path_x, path_y,\
              p[1][X],p[1][Y],\
              newx, newy, \
              newx1, newy1);


    file = fopen("temp.lsp", "a");
    fprintf (file, "%s", lisp2);
    fclose(file);

    path_x = p[1][X];
    path_y = p[1][Y];

  }
}

int Polygon::path(FPoint pnt1){
  int j= ((int)(RAND_x.ran1()*100))%4;
  int r_value;
  int r_seed = ((int)(RAND_x.ran1()*100))%100;
  //cout << "seed:" << r_seed << endl;
  j++;
  if (r_seed > 80)
    return 5;
  if ((p[j][X] < pnt1[X]) && (r_seed < 40))
    return 1;
  if ((p[j][X] > pnt1[X]) && (r_seed < 40))
   return 3;
  if (p[j][Y] < pnt1[Y])
    return 0;
  if (p[j][Y] > pnt1[Y])
    return 2;
  return 5;
}

void Polygon::action(int point, int poly, float size) {

  char tclcommand[200], points[100];

  int j= ((int)(RAND_x.ran1()*100))%4;
  int k= ((int)(RAND_x.ran1()*100))%4;
  int offx= ((int)(RAND_x.ran1()*100))%10;
  int offy= ((int)(RAND_x.ran1()*100))%10;
  int testx= ((int)(RAND_x.ran1()*100))%10;

  float x1= p[j+1][X]; float y1 = p[j+1][Y];
  float x2= p[k+1][X]; float y2 = p[k+1][Y];
  while (x2 == x1) {
    k= ((int)(RAND_x.ran1()*100))%4;
    x2= p[k+1][X]; float y2 = p[k+1][Y];
  }
   float m = ((y2 - y1) / (x2 - x1));
  int sign =  ((int)(RAND_x.ran1()*100))%2;
  int sign1 =  ((int)(RAND_x.ran1()*100))%2;
  float newx, newy, newx1, newy1;
   if (sign == 1) {
    newx = x1 + offx;
    newy = y1 + offy;
  }
  if (sign == 0) {
    newx = x1 - offx;
    newy = y1 - offy;
  }
  if (sign == 1)
    newx1 = x2 + testx;
  if (sign == 0)
    newx1 = x2 - testx;
  newy1 = (((newx1 - newx)*m) + newy);
  //cout << "object: " << newx << "," << newy << " & " << newx1 << "," << newy1
<< endl;
  Object_line[Object_line_num] = new Object(newx, newy, newx1, newy1,
Object_line_num, point, poly, size);
  Object_line[Object_line_num++]->plot();

}

void Polygon::display() {
cout << "Poly #" << id << " " << ht << " " << color << " " << n << attraction <<
endl;
for (int i=1; i <= n; i++)
  cout << p[i];

}

int Polygon::property() {
int s = 0;
for (int i=1; i <= n; i++)
  s = s+p[i].property();
return s;
}
void Polygon::rotate_object(int id, float about_x, float about_y, float dtheta)
{
  float l, theta;
  FPoint r(about_x, about_y);
  for (int i = 1; i <= n; i++) {
    l = sqrt((p[i][X]-about_x)*(p[i][X]-about_x) + (p[i][Y]-about_y)*(p[i][Y]-
about_y));
    if (!(l == 0)) {
      theta = angle(r, p[i]);
      p[i][X] = about_x + l*cos(dtheta + theta);
      p[i][Y] = about_y + l*sin(dtheta + theta);
    }
  }
  calc_min_xy();
}


void Polygon::calc_value()
{
  int long_side_vert;
  float length1, long_side_length, dtheta, min_x, max_x, min_y, max_y, max_area;
  FPoint p0;

  /**** rotate long side up ****/
  p[n] = p[0];
  long_side_length = sqrt((p[0][X] - p[1][X])*(p[0][X] - p[1][X])+
                   (p[0][Y] - p[1][Y])*(p[0][Y] - p[1][Y]));
  long_side_vert = 0;
  for (int i = 1; i < n; i++)
    {
      length1 = sqrt((p[i][X] - p[i+1][X])*(p[i][X] - p[i+1][X])+
                   (p[i][Y] - p[i+1][Y])*(p[i][Y] - p[i+1][Y]));
      if (length1>long_side_length) {
       long_side_length = length1;
       long_side_vert = i;
      }
```

```
  }
  dtheta = angle(p[long_side_vert], p[long_side_vert+1]);
  rotate_object(id, p[long_side_vert][X], p[long_side_vert][Y], (-1*dtheta));

  /**** calculate areas and value ratio ****/
  area = 0;
  min_x = 10000; max_x = -10000; min_y = 10000; max_y = -10000;
  p0 = p[0];
  for (i = 0; i < n; i++) {
    if (min_x > p[i][X]) min_x = p[i][X];
    if (max_x < p[i][X]) max_x = p[i][X];
    if (min_y > p[i][Y]) min_y = p[i][Y];
    if (max_y < p[i][Y]) max_y = p[i][Y];
    area = area + .5*fabs((p[i][X]-p0[X])*(p[i+1][Y]-p0[Y])-(p[i][Y]-
p0[Y])*(p[i+1][X]-p0[X]));
  }

  length = (max_x - min_x);
  width = (max_y - min_y);
  max_area = length*width;

  if (max_area == 0 ) value_ratio = 0;
  else value_ratio = area*color/max_area;

  calc_min_xy();
}


void Polygon::plot()
{
  char tclcommand[200], points[100];
  sprintf(tclcommand, "add_polygon_ui {%d ", n);
  for (int i = 1; i <= n; i++){
    sprintf(points, "%f %f ", p[i][X], p[i][Y]);
    strcat(tclcommand, points);
  }

  sprintf(points, "%d %d %d}", id, color, ht);
  strcat(tclcommand, points);

  int rv = Tcl_Eval(hello_ui.interp,tclcommand);
  if (rv != TCL_OK)
    {
      printf("error in polygon.C plot \n");
  //    exit(-1);
    }
}


float angle(FPoint p0, FPoint p1)
{
  float dx = p1[X]-p0[X];
  float dy = p1[Y]-p0[Y];

  if (dx > 0)
       return (atan(dy/dx));
  else
  {
    if( dx < 0 )
      return (M_PI + atan(dy/dx));
    else
      {
        if( dy >= 0 )
          return( M_PI/2 );
        else
          return( -M_PI/2 );
      }
  }

}

int Polygon::intersect(FPoint p0, FPoint p1, FPoint p2, FPoint p3)
{

    int j1, j2, k1, k2;

    j1 = ccw(p0, p1, p2);
    j2 = ccw(p0, p1, p3);
    k1 = ccw(p2, p3, p0);
    k2 = ccw(p2, p3, p1);
//cout << j1 << "," << j2 << "," << k1 << "," << k2 << endl;
    return (j1*j2 <= 0) && (k1*k2 <= 0);
    /*
    return ((ccw(l1p1, l1p2, l2p1)
    *ccw(l1p1, l1p2, l2p2)) <=0)
    && ((ccw(l2p1, l2p2, l1p1)
    *ccw(l2p1, l2p2, l1p2)) <=0);*/

/*
  return ((ccw(p0, p1, p2)
          *ccw(p0, p1, p3)) <=0)
    && ((ccw(p2, p3, p0)
        *ccw(p2, p3, p1)) <=0);*/
}


int Polygon::ccw(FPoint p0, FPoint p1, FPoint p2)
{
  float dx1, dx2, dy1, dy2;

  dx1 = p1[X] - p0[X];
  dy1 = p1[Y] - p0[Y];
  dx2 = p2[X] - p0[X];
  dy2 = p2[Y] - p0[Y];
//      cout << dx1 <<"," << dy1 << "," << dx2 << "," << dy2<< endl;
  if (dx1*dy2 > dx2*dy1) return +1;
  if (dx1*dy2 < dx2*dy1) return -1;
  if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
  if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2)) return +1;
  return 0;
}


ofstream& operator<<(ofstream& s, Polygon* a)
{
if (a != NULL) {
  s << "add_polygon_ui " << a->n << " ";
  for (int i = 0; i < a->n; i++) {
    s << a->p[i][X] << " " << a->p[i][Y] << " ";
  }
  s << a->id << " " << a->color << " " << a->ht << endl;
  return s;
}
}
ostream& operator<<(ostream& s, Polygon* a)
{
  s << "add_polygon_ui " << a->n << " ";
  for (int i = 0; i < a->n; i++) {
    s << a->p[i][X] << " " << a->p[i][Y] << " ";
  }
  s << a->id << " " << a->color << " " << a->ht << endl;
  return s;
}
/*
void Polygon::clear_point_buf() {
        for (int i=0; i<25; i++){
        point_influence[i] = 0;
//      cout << point_influence[i];
}
}
*/


void Polygon::print3d()
{
  FILE *file;
  char lispcommand[500];
  char lisp2[500];
  char points[500];
  char lispend[100];
  sprintf(lispcommand, "(command \"layer\" \"s\" \"poly\" \"\")");
  sprintf(lispend, "(command \"layer\" \"s\" \"0\" \"\")");
  sprintf(lisp2, "(command \"pline\"");
    for (int i = 1; i <= n; i++){
    sprintf(points, "'(%f %f 0.0)", p[i][X], p[i][Y]);
    strcat(lisp2, points);
  }
  sprintf(points, "\"C\")");
  strcat(lisp2, points);
  sprintf(points, "(command \"extrude\" \"l\" \"\" %f \"\")",
(((float)(ht+Simulation_round))/10));
 // cout << lispcommand << endl;
 // cout << lisp2 << endl;
 // cout << points << endl;
  file = fopen("temp.lsp", "a");
  fprintf (file, "%s %s %s %s", lispcommand, lisp2, points, lispend);
  fclose(file);
}
::::::::::::::
register_C_with_tcl.C
::::::::::::::
#include <tcl.h>
#include <tk.h>

#include "tcltkui.h"
#include "tcl_to_C.h"
#include "register_C_with_tcl.h"

extern UIParams_t hello_ui;

void register_C_with_tcl (void)
{

  Tcl_CreateCommand(hello_ui.interp,
                  "say_hello_ui",
                  (Tcl_CmdProc *)   say_hello_ui,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);

  Tcl_CreateCommand(hello_ui.interp,
                  "move_object_ui",
                  (Tcl_CmdProc *) move_object_ui ,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);

  Tcl_CreateCommand(hello_ui.interp,
                  "report_ui",
                  (Tcl_CmdProc *) report_ui ,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);

  Tcl_CreateCommand(hello_ui.interp,
                  "zone_ui",
                  (Tcl_CmdProc *) zone_ui ,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);

  Tcl_CreateCommand(hello_ui.interp,
                  "add_object_ui",
                  (Tcl_CmdProc *) add_object_ui ,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);

  Tcl_CreateCommand(hello_ui.interp,
                  "object_ui",
                  (Tcl_CmdProc *) object_ui ,
                  (ClientData)  hello_ui.w,
                  (Tcl_CmdDeleteProc*) NULL);
}

::::::::::::::
report.C
::::::::::::::
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>

#include "random.h"
#include "fpoint.h"
#include "report.h"
#include "polygon.h"
#include "object.h"
extern FPoint* Point[];
extern Polygon* Poly[];
extern Object* Object_line[];
extern Random RAND_x;
extern int Max_points;
extern int Point_num;
extern int Poly_num;
extern Object_line_num;

void report()
  {
    int i, j, k;
    printf("Report... Iteration:");
       //cout << "Point count = " << Point_num << endl;
   // for (i = 0; i < Point_num; i++)
      //cout << "agent num:"<< i << " " << *(Point[i]);
    for (i = 0; i < Poly_num; i++){
      //cout << "agent num:"<< i << endl;
       //Poly[i]->display();
       Poly[i]->print3d();
    }
       //ofstream temp_lisp ("temp.lsp");
       for (i=0; i < Object_line_num; i++) {
       Object_line[i]->print3d();
       }
       //temp_lisp.close();
```

```
  }


::::::::::::::
say_hello.C
::::::::::::::
#include <stdio.h>
#include "say_hello.h"
void say_hello()
  {
  printf("Hello \n");
  }
::::::::::::::
tcl_to_C.C
::::::::::::::
/*****************************************************************
*                                                               *
*                     modes_commands                            *
*                                                               *
*****************************************************************/
/*        tcl commands that modes parser must understand
*****************************************************************/
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include <tk.h>
#include <tcltkui.h>

#include "tcl_to_C.h"
#include <fpoint.h>
#include "say_hello.h"
#include "move_object.h"
#include "add_object.h"
#include "report.h"
#include "object.h"

#include <time.h>

extern FPoint* Point[];
extern int Point_num;
extern Object* Object_line[];
extern int Object_line_num;
extern UIParams_t hello_ui;
//extern FPoint Point[];

/*****************************************************************
*                                                               *
*                     say_hello_ui                              *
*                                                               *
*****************************************************************/

int say_hello_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
    if (argc != 1) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " say_hello \"", (char *) NULL);
      return TCL_ERROR;
      }
    say_hello();

    return TCL_OK;
    }



/*****************************************************************
*                                                               *
*                     move_object_ui                            *
*                                                               *
*****************************************************************/

int move_object_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
    int value, inc;
    char *string, *varValue, newValue[20];
    if (argc != 2) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " move_object \"", (char *) NULL);
      return TCL_ERROR;
      }

    string = Tcl_GetVar(interp, argv[1], TCL_LEAVE_ERR_MSG);
    if (string == NULL) {
      return TCL_ERROR;
    }
    if (Tcl_GetInt (interp, string, &value) != TCL_OK) {
      return TCL_ERROR;
    }
    sprintf(newValue, "VALUE !!! = %d", value);
   // cout << "THE BIG GUY.." << value << endl;
    move_object(value);

    return TCL_OK;
    }


/*****************************************************************
*                                                               *
*                     add_object_ui                             *
*                                                               *
*****************************************************************/

int add_object_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
    int value, inc;
    char *string, *varValue, newValue[20];
    if (argc != 2) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " move_object \"", (char *) NULL);
      return TCL_ERROR;
      }

    string = Tcl_GetVar(interp, argv[1], TCL_LEAVE_ERR_MSG);
    if (string == NULL) {
      return TCL_ERROR;
    }
    if (Tcl_GetInt (interp, string, &value) != TCL_OK) {
      return TCL_ERROR;
    }
    sprintf(newValue, "VALUE !!! = %d", value);
    add_object(value);

    return TCL_OK;
    }

/*****************************************************************
*                                                               *
*                     report_ui                                 *
*                                                               *
*****************************************************************/

int report_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
    if (argc != 1) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " say_hello \"", (char *) NULL);
      return TCL_ERROR;
      }


    report();

    return TCL_OK;
    }
/*****************************************************************
*                                                               *
*                     zone_ui                                   *
*                                                               *
*****************************************************************/

int zone_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
       int i;
    if (argc != 1) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " say_hello \"", (char *) NULL);
      return TCL_ERROR;
      }
     //printf("zone:");
          for (i = 0; i < Point_num; i++){
      Point[i]->plot_zone();
       cout << "carry:" << Point[i]->ob_flag << endl;
      }

    return TCL_OK;
    }

/*****************************************************************
*                                                               *
*                     object_ui                                 *
*                                                               *
*****************************************************************/

int object_ui( ClientData, Tcl_Interp * interp,
                         int argc, char ** argv)
  {
       int i;
    if (argc != 1) {
      Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                     " object_ui \"", (char *) NULL);
      return TCL_ERROR;
      }
     //printf("zone:");
          for (i = 0; i < Object_line_num; i++){
      Object_line[i]->plot();

      }

    return TCL_OK;
    }


/*****************************************************************
*                                                               *
*                     add_circle_ui                             *
*                                                               *
*****************************************************************/
/*        adds a circle
*****************************************************************/
int add_point_ui(ClientData, Tcl_Interp *interp, int argc, char **argv)
{
  cout << "tcl_to_C.C add_point_ui:" << endl;

/*  if (argc != 2) {
    Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                   " id\"", (char *) NULL);
    return TCL_ERROR;
  }
  int id   = (int)strtod(argv[1], (char**)NULL);
  char tclcommand[200];
  Polygon *a;
  for (int i = 0; i < 3; i++) {
    a = world[i]->get_element_id(id);
    if (a) {
      cout << "a = " << a << endl;
      a->plot();
    }
  }*/
  return TCL_OK;
}

/*****************************************************************
*                                                               *
*                     move_object_number                        *
*                                                               *
*****************************************************************/
/*        moves an object with id
*****************************************************************/
int move_object_number(ClientData, Tcl_Interp *interp, int argc, char **argv)
{
  cout << "tcl_to_C.C move_object_number:" << endl;

  if (argc != 2) {
    Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                   " id\"", (char *) NULL);
    return TCL_ERROR;
  }
  int id   = (int)strtod(argv[1], (char**)NULL);
    cout << "ID = " << id << " : " << Point[id] << endl;
/*
  char tclcommand[200];
  Polygon *a;
  for (int i = 0; i < 3; i++) {
    a = world[i]->get_element_id(id);
    if (a) {
      cout << "a = " << a << endl;
      a->plot();
    }
  }*/
  return TCL_OK;
}
```

```
/****************************************************************
*                                                               *
*                    add_line_ui                              *
*                                                               *
*****************************************************************/
/*           adds a line
*****************************************************************/
int add_line_ui(ClientData, Tcl_Interp *interp, int argc, char **argv)
{
  cout << "tcl_to_C.C add_line_ui:" << endl;

  return TCL_OK;
}



::::::::::::::
geometry.tcl
::::::::::::::

proc dpos m {
    wm geometry $m +100+100
}

wm withdraw .


# mkItems w
#
# Create a top-level window containing a canvas that displays the
# various item types and allows them to be selected and moved.  This
# demo can be used to test out the point-hit and rectangle-hit code
# for items.
#
# Arguments:
#    w -      Name to use for new top-level window.

proc make_menu {{w .citems}} {
    global c tk_library
    global zone_toggle
    global poly
    global point
    global i
    set i 100
    set zone_toggle 0
    set poly 0
    set point 1
    catch {destroy $w}
    toplevel $w
    dpos $w
    wm title $w "GenWorld - 8"
    wm iconname $w "Genworld"
    wm minsize $w 500 500
    set c $w.frame2.c
    frame $w.buttons
#   tk_setPalette gray60
  pack  $w.buttons -side right -expand y -fill both -pady 2m
  button $w.buttons.1 -text object -command "object_ui"
  button $w.buttons.2 -text B -command "add_point_ui"
  button $w.buttons.3 -text Move  -command "move_object_ui i"
  button $w.buttons.4 -text zone -command "zone_ui"
  button $w.buttons.5 -text report -command "report_ui"
  button $w.buttons.6 -text poly -command "add_object_ui poly"
  button $w.buttons.7 -text point -command "add_object_ui point"
  button $w.buttons.8 -text H -command "say_hello_ui"
  button $w.buttons.9 -text object_ui -command "$c delete object"
  button $w.buttons.10 -text zone_off -command "$c delete zone"
  entry $w.buttons.entry -width 3 -relief sunken -bg white -textvariable numb
  label $w.buttons.label -textvariable numb
  label $w.buttons.move -textvariable iterate
  button $w.buttons.code -text "See Code" -command "showCode $w"
  button $w.buttons.dismiss -text Quit -command "destroy ."
  pack  $w.buttons.1 $w.buttons.2 $w.buttons.3 $w.buttons.4 $w.buttons.5
$w.buttons.6 $w.buttons.7 $w.buttons.8 $w.buttons.9 $w.buttons.10
$w.buttons.label $w.buttons.entry $w.buttons.move $w.buttons.code
$w.buttons.dismiss -side top -expand 1 -fill both
#pack $w.buttons.1 $w.buttons.2  -expand 1p


    frame $w.frame1 -relief raised -bd 2
    frame $w.frame2 -relief raised -bd 2
    pack append $w $w.frame1 {top fill} $w.frame2 {top fill expand}

    canvas $c -scrollregion {0c 0c 30c 24c} -width 15c -height 12c -bg Gray70
    scrollbar $w.frame2.vscroll  -relief sunken -command "$c yview"
    scrollbar $w.frame2.hscroll -orient horiz -relief sunken -command "$c xview"
    pack append $w.frame2 $w.frame2.hscroll {bottom fillx} \
          $w.frame2.vscroll {right filly} $c {expand fill}
    $c config -xscroll "$w.frame2.hscroll set" -yscroll "$w.frame2.vscroll set"

    set font1 -Adobe-Helvetica-Medium-R-Normal-*-120-*
    set font2 -Adobe-Helvetica-Bold-R-Normal-*-240-*
    if {[winfo screendepth $c] > 4} {
       set blue DeepSkyBlue3
       set red red
       set blue blue
       set green SeaGreen3
    } else {
       set blue black
       set red black
       set blue blue
       set green black
    }
 set iterate 0

}

proc move_something i {
global iterate
set iterate [expr $iterate + $i]
#[ move_object_ui i]
}

proc add_point_ui {list} {
global c
set x [lindex $list 0]
set y [lindex $list 1]
set x1 [lindex $list 2]
set y1 [lindex $list 3]
set colorid [lindex $list 4]
set id [lindex $list 5]
set field [expr [lindex $list 6] /2]
set colorlist {red magenta yellow blue green}
set off [expr ($x1 - $x) /2]
set cen_x [expr $x + $off]
set cen_y [expr $y + $off]

if {$colorid == 5} {
  set color [lindex $colorlist 0]
}
if {$colorid == 4} {
   set color [lindex $colorlist 1]
}
if {$colorid == 3} {
   set color [lindex $colorlist 2]
}
if {$colorid ==  2} {
   set color [lindex $colorlist 3]
}
if {$colorid == 1} {
   set color [lindex $colorlist 4]
}
set xx [expr $cen_x+$field]
set yy [expr $cen_y+$field]
#eval [concat  $c create polygon [expr $cen_x - $field] [expr $cen_y - $field]
$xx [expr $cen_y - $field]  $xx $yy [expr $cen_x - $field]  $yy -stipple
@/afs/sipb/project/tcl/lib/tk/demos/bitmaps/grey.25 -tags zone]
eval [concat $c create oval $x $y $x1 $y1 -fill $color -tags $id]
}

proc plot_zone {list} {
global c
global zone_toggle
puts $zone_toggle
#if {$zone_toggle == 0} {
set x [lindex $list 0]
set y [lindex $list 1]
set field [expr [lindex $list 6]  /2]
set x1 [lindex $list 2]
set y1 [lindex $list 3]
set off [expr ($x1 - $x) /2]
set cen_x [expr $x + $off]
set cen_y [expr $y + $off]
set xx [expr $cen_x+$field]
set yy [expr $cen_y+$field]
puts $cen_x
eval [concat  $c create polygon [expr $cen_x - $field] [expr $cen_y - $field]
$xx [expr $cen_y - $field]  $xx $yy [expr $cen_x - $field]  $yy -stipple
@/afs/sipb/project/tcl/lib/tk/demos/bitmaps/grey.25 -tags zone]
set zone_toggle 1
#}
#elseif {$zone_toggle == 1} {
#$c delete zone
#set zone_toggle 0
#}
}


proc add_line_ui {list} {
global c
set x [lindex $list 0]
set y [lindex $list 1]
set x1 [lindex $list 2]
set y1 [lindex $list 3]
set colorid [lindex $list 4]
set type [lindex $list 6]
set colorlist {khaki3 darkorchid3 orange3 yellow grey80 blue green}
set wd [lindex $list 5]
if {$colorid == 0} {
  set color [lindex $colorlist 0]
}
if {$colorid == 1} {
   set color [lindex $colorlist 1]
}
if {$colorid == 2} {
   set color [lindex $colorlist 2]
}
if {$colorid ==  3} {
   set color [lindex $colorlist 3]
}
if {$colorid == 4} {
   set color [lindex $colorlist 4]
}
eval [concat $c create line $x $y $x1 $y1 -fill $color -width $wd -tags type]
}

proc add_line_object_ui {list} {
global c
set x [lindex $list 0]
set y [lindex $list 1]
set x1 [lindex $list 2]
set y1 [lindex $list 3]
set colorid [lindex $list 4]
set type [lindex $list 6]
set colorlist {khaki3 darkorchid3 orange3 yellow grey80 blue green}
set wd [lindex $list 5]
if {$colorid == 0} {
  set color [lindex $colorlist 0]
}
if {$colorid == 1} {
   set color [lindex $colorlist 1]
}
if {$colorid == 2} {
   set color [lindex $colorlist 2]
}
if {$colorid ==  3} {
   set color [lindex $colorlist 3]
}
if {$colorid == 4} {
   set color [lindex $colorlist 4]
}
eval [concat $c create line $x $y $x1 $y1 -fill $color -width $wd -tags object]
}

proc add_line_special_ui {list} {
global c
set x [lindex $list 0]
set y [lindex $list 1]
set x1 [lindex $list 2]
set y1 [lindex $list 3]
set x2 [lindex $list 4]
set y2 [lindex $list 5]
set colorid [lindex $list 6]


set colorlist {khaki3 magenta orange3 yellow blue green}
set wd [lindex $list 7]
if {$colorid == 0} {
  set color [lindex $colorlist 0]
}
if {$colorid == 1} {
   set color [lindex $colorlist 1]
}
if {$colorid == 2} {
   set color [lindex $colorlist 2]
}
if {$colorid ==  3} {
```

```
   set color [ lindex $colorlist 3]
}
if { $colorid == 4} {
   set color [ lindex $colorlist 4]
}
eval [ concat $c create line $x $y $x1 $y1 $x2 $y2 -fill $color -width $wd -
smooth on -tags smothey]
}


proc add_polygon_ui { list} {
global c
#  pick off different parts of the list
set n [ lindex $list 0]
set nnodes [ expr 2* $n]
set newlist [ lrange $list 1 $nnodes]
#puts stdout "$newlist $nnodes"
set idpos [ expr $nnodes+1]
set id [ lindex $list $idpos]
set colorpos [ expr $idpos+1]
set colorid [ lindex $list $colorpos]
#puts stdout "$newlist $id $colorid"

# First find corresponding colors to objects

set colorlist { red magenta yellow royalblue3 green}

if { $colorid == 5} {
  set color [ lindex $colorlist 0]
}
if { $colorid == 4} {
   set color [ lindex $colorlist 1]
}
if { $colorid == 3} {
   set color [ lindex $colorlist 2]
}
if { $colorid ==  2} {
   set color [ lindex $colorlist 3]
}
if { $colorid == 1} {
   set color [ lindex $colorlist 4]
}
# Now draw object on canvas
eval [ concat $c create polygon $newlist -fill $color -tags $id ]
puts $newlist
#  Now pass it over to C++ for inclusion in world list
#eval [ concat add_polygon_ui $list]
}


proc object_replot {
$c delete ob
object_ui
}
;* DXF takes an integer dxf code and an entity data list.
;* It returns the data element of the association pair.
;*
(defun dxf(code elist)
  (cdr (assoc code elist))   ;finds the association pair, strips 1st element
);defun
;*
(defun completer(ht)

  (if (< ht 1)
      (setq ht 1)
   )
  (command "pedit" "l" "s" "")
  (command "change" "l" "" "p" "la" "temp" "")

  (command "explode" "l")
  (setq templist (ssget "x" '((8 . "temp"))))

  (command "pedit" (ssname templist 0) "Y" "j" templist "" "X")

  (command "change" "l" "" "p" "la" "curve" "")
  (setq farpoint '(1000 1000 0))
  (setq ent1 (entlast))
  (command "offset" 1 ent1 farpoint "")
  (setq ent2 (entlast))

  (setq pt1 (cdr (assoc 10 (entget (entnext ent1)))))
  (setq temp ent1)
  (while (not (equal "SEQEND" (dxf 0 (entget (setq temp (entnext temp))))))
    (setq hold temp)
   )
  (setq pt2 (cdr (assoc 10 (entget hold))))
  (setq pt3 (cdr (assoc 10 (entget (entnext ent2)))))
  (setq temp ent2)
  (while (not (equal "SEQEND" (dxf 0 (entget (setq temp (entnext temp))))))
    (setq hold temp)
   )
  (setq pt4 (cdr (assoc 10 (entget hold))))
  (command "pline" pt1 pt3 "")
  (setq ent3 (entlast))
  (command "pline" pt2 pt4 "")
  (setq ent4 (entlast))
  (command "pedit" ent1 "j" ent2 ent3 ent4 "" "")
  (command "extrude" "l" "" ht "")

)
(defun completers(ht)
  (if (< ht 1)
      (setq ht 1)
   )

  (setq farpoint '(1000 1000 0))
  (setq ent1 (entlast))
  (command "offset" 1 ent1 farpoint "")
  (setq ent2 (entlast))

  (setq pt1 (cdr (assoc 10 (entget (entnext ent1)))))
  (setq temp ent1)
  (setq listy pt1)
  (while (not (equal "SEQEND" (dxf 0 (entget (setq temp (entnext temp))))))
    (setq hold temp)
    (print (cdr (assoc 10 (entget hold))))
    (setq listy (cons listy (cdr (assoc 10 (entget hold)))))
   )
  (setq pt2 (cdr (assoc 10 (entget hold))))
  (setq pt3 (cdr (assoc 10 (entget (entnext ent2)))))
  (setq temp ent2)
  (while (not (equal "SEQEND" (dxf 0 (entget (setq temp (entnext temp))))))
    (setq hold temp)
   )
  (setq pt4 (cdr (assoc 10 (entget hold))))
  (command "pline" pt1 pt3 "")
  (setq ent3 (entlast))
  (command "pline" pt2 pt4 "")
  (setq ent4 (entlast))
  (command "pedit" ent1 "j" ent2 ent3 ent4 "" "")
  (command "extrude" "l" "" ht "")
)

(defun test(plist)
  (command "pline"
          (while (car plist)
            (car plist)
            (setq plist (cdr plist))
            )
          )
  )

(defun ext(ptlist)
 (print (car ptlist))
  (if (cadr ptlist)
      (ext (cdr ptlist))
    )

  )

(defun abstractor (pt1 pt2 orient num ht wd sold)
  (if (= orient 1)
      (
       progn
;       (command "layer" "s" "temp_sol" "")
       (setq d (command "dist" pt1 pt2))
       (setq d (getvar "distance"))
       (setq offs (/ d (+ num 1)))
       (ucs_change pt1 pt2)
       (setq other (list offs wd 0))

       (command "box" "" other ht)
       (setq ent (entlast))
       (if (= sold 1)
           (progn
        (command "change" ent "" "P" "la" "SOL" "")
        (setq ent (entlast))
         )
         )
       (if (= sold 0)
          ( progn
                                        ;for interior volumes test...
         (setq intvol (list  0.5  0.5 0.5))
         (if (= ht 1) (setq ht 2))
         (setq intother (list (- (car other) 0.5) (- (cadr other) 0.5) (- ht
0.5)))
                                        ;       (command "layer" "s" "temp_sub" "")
         (command "box" intvol intother)
                                        ; end test
         (setq ent1 (entlast))
         (command "subtract" ent "" ent1 "")
         (setq ent (entlast))
         )
         )
       (if (> num 1)
           (command "array" ent "" "r" "" num (+ offs 1))
         )
       (command "ucs" "w")
       )
    )
  (if (= orient 0)
      (
       progn

       (setq d (command "dist" pt1 pt2))
       (setq d (getvar "distance"))
       (setq offs (/ d (+ num 1)))
       (ucs_change pt1 pt2)
       (setq other (list (/ d (+ num 1.0)) wd ht))
       (command "box" "" other)
       (setq ent (entlast))
        (if (= sold 1)
           (progn
           (command "change" ent "" "P" "la" "SOL" "")
            (setq ent (entlast))
         )
         )
       (if (= sold 0)
           (progn
             ;for interior volumes test...
            (setq intvol (list  0.5  0.5 0.5))
            (if (= ht 1) (setq ht 2))
            (setq intother (list (- (car other) 0.5) (- (cadr other) 0.5) (-
ht 0.5)))
            (command "box" intvol intother)
            (setq ent1 (entlast))
            (command "subtract" ent "" ent1 "")
            (setq ent (entlast))
            )
         )
       (command "rotate" ent "" '(0 0 0) 90)
       (if (> num 1)
           (command "array" ent "" "r" "" num (+ offs 1))
         )
       (command "ucs" "w")
       )
   )
  )

(defun ucs_change (pt1 pt2)
  (if (< (car pt1) (car pt2))
      (setq pt3 '(0 1000 0))
    )
  (if (> (car pt1) (car pt2))
      (setq pt3 '(0 -1000 0))
    )
  (if (= (car pt1) (car pt2))
      (if (> (cadr pt1) (cadr pt2))
         (setq pt3 '(1000 0 0))
       )
     (if (< (cadr pt1) (cadr pt2))
        (setq pt3 '(-1000 0 0))
      )
    )
  (command "ucs" 3 pt1 pt2 pt3)
)
```