Class Hierarchy Design for Space Time Problems

by

Sanjay Chopra

A project report submitted to the Faculty of Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Approved:

[illegible]

Dr. Dennis Kafura
Committee Chairman

Osman Balci

Dr. Osman Balci
Committee Member

Sallie Henry

Dr. Sallie Henry
Committee Member

July 1995
Blacksburg, Virginia

Key words: Class Hierarchy, Object Oriented, Space time, Simulation

c.2

LD
5655
V851
1995
.467
c.2

Class Hierarchy Design for Space Time Problems

by

Sanjay Chopra

Dr. Dennis Kafura, Chairman

Computer Science

Abstract

The purpose of the project is to design a class hierarchy that will aid in the development of simulations for certain space time problems. The class hierarchy and the problem domain to which it applies are illustrated by considering simulations of three representative problems: a pool game; a collision detection system for robot arms; an automated highway system. The emphasis in the simulations is on the class hierarchy. The class hierarchy contains base classes to model objects, space, time and interactions among objects. These classes could be applied to other similar problems in the problem domain. For example the class objects help to model various objects like cars, pool balls, robots, trains, birds etc. Class space allows the user to subdivide the problem space into smaller dynamic sub-spaces. The user can define rules to decompose the space into 'n' smaller spaces when there are more than 'x' objects in the space.

Acknowledgments

I wish to express my sincere appreciation and thanks to Dr. Dennis Kafura for his invaluable guidance and instruction through the project. His cheerful and friendly support made the work very enjoyable.

I would also like to thank Dr. Osman Balci and Dr. Sallie Henry for being on my defense committee, and for their excellent teaching.

My thanks to every one at the Reactive Systems Laboratory, VPI&SU for maintaining all systems in great condition, particularly Actor.

Finally, I would like to acknowledge with deep gratitude the many opportunities provided to me by my parents; the love and support of my fiancée and brother, that have made this project possible.

List of Figures

Contents

1.0 Introduction

1.1 Problem Domain

Space time problems are those problems where mobile objects move and interact with one another in space with respect to time. The key elements of these problems are space, objects in the space, and the interactions among objects. In different problems different characteristics may be associated with the notion of "object", "space" and "interactions". For example, in the pool game simulation space constitutes of the pool table, while in a galaxy simulation, space is the entire galaxy. Objects in the pool game would be balls, walls, pockets and the cue, and the interactions in this case would be balls colliding with one another, the walls, the cue or the pockets. Objects in the simulation of a galaxy would be the planets, various stars, constellations, etc. and the interactions would be the gravitational forces between them.

Among all space time problems, those included in the problem domain have the following characteristics:

1. The emphasis is on individual objects and the interactions among them, rather than on patterns or aggregate behaviors.
2. There is a distinct space within which the objects interact.
3. The problem can be modeled using a discrete conservative simulation.

Some problems in the problem domain are : First, simulation of an automated highway system, which constitutes of a collision detection system to warn against imminent collisions involving cars moving on the highway. In this problems cars, curb, lanes etc. are the objects. All the roads and highways constitute the space. Second, modeling galaxies, planets, solar systems, comets and stars. This model has distinct objects in space interacting with other objects. Third, simulations to see the effects of: a falling meteor. Fourth, reactive systems that drive or are driven by events in the physical world and for which time and space are scarce resources. Fifth, systems for the command and control of real-world entities, such as the routing of air or railway traffic. In all such systems each object's interactions are considered within the defined problem space.

Problems outside the problem domain are illustrated by the following examples. First, simulating the depletion of the ozone layer caused by its interaction with other chemicals present in the atmosphere. In this problem though there are objects that interact in a given space with respect to time, the intent is to look at aggregate behaviors and not at individual objects. Second, predicting the pollution content of a river in a given period of time, due to the various pollutants being introduced at various stages down the stream. This problem also involves modeling aggregate behaviors and not at individual objects. Third, studying the traffic patterns to determine optimal timing of traffic lights (to facilitate smooth traffic) during the course of the day. Here, the emphasis in again on aggregate behavior (how many cars move in a certain direction in a given period of time at a certain time during the day), and not on individual cars crossing the traffic light.

The purpose of this project is to develop a generic class hierarchy for space time problems within the problem domain. This class hierarchy is not intended to be a simple encompassing solution for all space time problems (even for those within the problem domain). The class hierarchy is intended to be a comprehensive infrastructure to extend as per the problem requirements. The design of the class hierarchy has the flexibility to cater for subtle but important variations in problem characteristics. For instance, the class hierarchy can be extended to model objects of all shapes and sizes, to model various physical characteristics of objects, and to change the characteristics of space as per problem needs.

The following three space time problems have been used to design the class hierarchy. The pool game simulation covers problems where there is actual physical interaction among objects. The simulation needs to detect these interactions and resolve them by providing new paths to the objects. The automated highway problem is more reactive in nature where the motion of automobiles makes the system issue a warning if any two automobiles are about to collide. In this problem, cars are independent entities that cross space boundaries at will. In the robot arm collision detection system the idea is to issue a warning if the arm is about to collide with other objects around it. In this problem there is one mobile object and the space changes according to its movement. A complete solution to the pool game simulation and detailed designs for both the automated highway problem and the robot arm collisions detection system are presented.

1.1.1 Pool game simulation:

This problem simulates pool balls as they move and collide with one another on a pool table that constitutes the entire problem space. The "universe of discourse" of this problem is moving balls, stationary holes, cue, walls and the pool table.

The following are the reasons for choosing this problem:

1. This is a relatively simple two dimensional problem, allowing the design to be focused on the class hierarchy rather than the geometrical representation of the objects and the physics.

2. Objects (specially the balls) in this problem are very dynamic, requiring proper modeling of the space and the physics.

3. Objects interact with one another; balls interact with other balls and as a result acquire new directions and velocities, balls collide with walls and are deflected at an angle equal to the incident angle, balls interact with holes and are removed from the problem space

The implementation of the pool game problem employs simple kinematics and dynamic principles such as conservation of momentum and energy. Further, no table friction is assumed in this problem.

1.1.2 Robot arm collision detection system:

In this problem a real-time safety system warns of imminent unintentional collisions of a robot arm with nearby stationary objects [Shaf90]. During its operations the robot arm should not collide with itself, other robot arms and other objects in the environment. If an unintentional collision is about to occur the software should stop the robot arm. The objects in this problem are various components of the robot arm, other arms and objects. A robot arm has seven degrees of freedom, with each section of the arm being nearly cylindrical in shape. The operating environment is not static, that is, the system accommodates movement of the arms as well as other objects. The operating paradigm is one of receiving an indication of movement by certain objects, updating the representation of the environment to reflect that movement, and reporting any imminent collision.

This is a complex three dimensional problem, where movement of objects is input to the system and it must decide in real-time if that movement would result in a collision. The space is the environment around the robot arm and is subdivided into smaller sub-spaces or the sub-divisions reunite based on the movement of the robot arm. Space in this problem is relatively static as the robot arm has restricted degrees of freedom.

1.1.3 Automated highway problem:

In this problem "smart" cars sense the presence of other cars and obstacles and notify the driver of impending danger. The objects in this problem are cars, lanes, curb, etc. The space in this problem is the highway. This space is divided into smaller sub spaces called cells. All cars within a cell look out for other cars in the same cell and check for collision with them and with other objects within the cell. The implementation of this problem employs simple kinematics and dynamic principles.

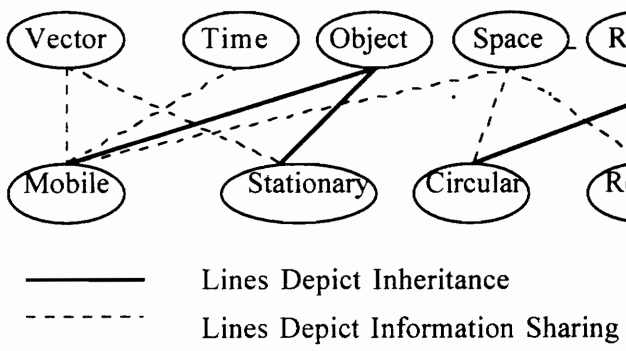
This is a more complex two dimensional problem but the objects (cars) do not cross space boundaries as frequently as in the pool game simulation, and the objective in this problem is to avoid interaction among cars rather than resolve the interaction.

1.2 Class Hierarchy

A generic class hierarchy [Bing93] is presented to solve problems in the problem domain defined above. This class hierarchy was developed to answer questions like: How effective are object oriented concepts [Kors90] in modeling space time problems? What attributes should objects possess? What is the structure of the space class? How does the space class help model and efficiently solve space time problems? How are various shapes and physical properties of various objects represented?

This class hierarchy [Wass91a] [Wass91b] is intended to be useful for any space time problem within the problem domain. It provides a basic infrastructure comprising of base classes and some problem specific classes that can be used as examples to solve other space time problems within the problem domain.

The class hierarchy constitutes of the following classes: vector, shape, linear, rectangular, circular, object, stationary, mobile, space and rule as shown in Fig. 1. Embedded within the space class are object list and llist classes.



Class Hierarchy
Fig. 1

1.2.1 Class Vector

Class vector helps in modeling the physics of a problem. This class is totally independent of the individual problems and can be used in any application where vector manipulations are needed.

Vectors are directed lines, that is, they have a direction and a length. Numerically, vectors are represented by two components, one for the x direction and one for the y direction. Each component is the length of the projection of the vector on the corresponding coordinate axis. By convention, a component is positive if the projection of the vector on the corresponding coordinate axis points in the same direction as the coordinate axis, and the component is negative if the projection points in the direction opposite to the coordinate axis.

Mathematically, vectors do not reside at any particular place on the plane, rather they can be moved around to wherever they are needed. There may be confusion over the fact that the coordinates of points (x, y) and the components of vectors are formally the same, that is, that both points and vectors are represented by pairs of numbers. The resolution of the confusion lies in the fact that associated with each point is a vector pointing from the origin of coordinates to the point, and the components of that vector are numerically equal to the coordinates of the point.

The following methods have been defined for class vector.

```
class vector {
 public:
        double vx;                          // x value of the vector
        double vy;                          // y value of the vector
        vector();                           // Constructor
        ~vector();                          // Destructor
        vector(double x, double y);         // A vector defined by the x and y values
        vector(point p1, point p2);         // A vector can be defined between two points
        inline point OtherEndPoint(const point p); // Returns the end point of a vector
        inline double Mag();                // Returns the magnitude of a vector
        inline double MagSqr();             // Returns the square of the magnitude
        inline vector operator*(const scalar s);  // Overloaded to cal. vector times scalar
        inline vector Reverse();            // Returns the reverse of the vector
        inline vector Unit();               // Calculates the unit vector
        inline vector ClockWiseNormal();    // A normal vector in the clockwise direction
        inline vector AntiClockWiseNormal();  // An anti-clockwise normal vector
        inline vector operator+(const vector v);  // To add vectors
        inline vector operator-(const vector v);  // To subtract vectors
        inline double operator&(const vector v); // To calculate the dot product
        inline double operator*(const vector v); // To calculate the cross product
        inline vector Display();            // To get the x and y components of the
vector
};
```

1.2.2 Class Shape

Class shape is a base class used to represent various objects in a problem. For instance, lines, rectangles, circles are derived from this base class to represent various two-dimensional objects and cubes, spheres, cubiods, cones, etc. to represent three dimensional objects. This class helps model the possible onscreen appearance of various objects. It also helps define events associated with various shapes, e.g., how to detect collisions between circles and how to resolve these collisions, as illustrated in the pool game simulation. All shapes are derived from this base class or one of derived classes of class shape.

```
class shape {   public:
        shape();                        // Constructor
        ~shape();                       // Destructor
        virtual void draw() = 0;        // All pure virtual functions
        virtual void move() = 0; ....; };
```

1.2.3 Linear

Class linear derives from class shape. A line segment is represented by one end point, E, a unit normal vector (N), and a vector (L), pointing from one end point to the other end point. Class Linear has the following declaration:

```
class linear : public shape {
public:
        point         E;                // A point : The origin of the line segment
        vector  L;                      // A vector pointing from the origin to the end
        vector  N;                      // A unit normal vector to the vector L.
        linear(){ }                     // Constructor
        linear(point p, vector v)       // Constructor with arguments
          {
           vector  Tv;
                Pl = p;
                Vl = v;
                Tv  = v.ClockWiseNormal();
                Nl  = Tv.Unit();
          }
        ~linear (){ }                   // Destructor
};
```

The unit perpendicular, N, strictly speaking, is redundant information. The line is actually completely described by its endpoint E and its length vector L. As N is required in many calculations it is being stored along with E and L for efficiency. To create a line segment, the user needs to specify a point and a vector. The constructor method for class Linear will automatically compute the value of N. Class Linear is used to model the walls of the pool table and space edges in the pool game simulation and lanes and curbs in the automated highway problem.

1.2.4 Rectangular

Class rectangular derives from the shape class. This class has the following declaration:

```
class rectangular : public shape {
protected:
        point   LeftTop;                    // Top left corner of the rectangle
        point   RightBottom;                // Bottom right corner of the rectangle
        linear* side[MAX_R_SIDES];          // Linear representing the sides for collision
                                            // detection.
                                            // Constructor
        rectangular(double x1, double y1, double x2, double y2);
        rectangular(){}
        ~rectangular () {}                  // Destructor
        void SetPoints(point p1, point p2)  // Set the points of a rectangle
        {LeftTop = p1;RightBottom = p2;}
        int contain(point p);               // To check if the point is inside the rectangle
        rectangular Getpoints();            // Get the two points of the rectangle
        linear GetSide(int num)             // Get the side of a rectangle
        {return *side[num];}
        void    ShowSides();                // To print the sides of a rectangle
        // Draw and move functions ...
};
```

This class may be used to model actual objects in the simulation or abstract objects. For example, in pool game simulation class rectangular is used to model space. Class rectangular is used as a template class to the space class. Whenever a space is constructed it constructs its respective shape object. Rectangular has a contain method that allows the space to find what objects lie within its domain. Further, the sides of the rectangle are used to check for collision of objects within the space with the space boundary. It is this class that provides a definite boundary to the abstract space. In case, of a dynamic space, the space can create smaller or larger instances of its respective shape. The subdivision of rectangular spaces is explained in detail in the space class (Section 2.6.1).

1.2.5 Circular

Class circular derives from shape class. This class has the following declaration:

```
class circle : public shape {
protected:
        double  Radius;                     // Radius of the circle
        point   Center;                     // Center of the circle
public:
        circle();                           // Constructors for circle
        circle(double rad, int xc, int yc);
        ~circle();                          // Destructor
        inline virtual void SetCircle(double r, point c);
                                            // Set center and radius
        inline virtual circle GetCircle();  // Get center and radius
        inline int contain(point p);        // Check if a circle contains a given point
        // Draw and move functions ...
};
```

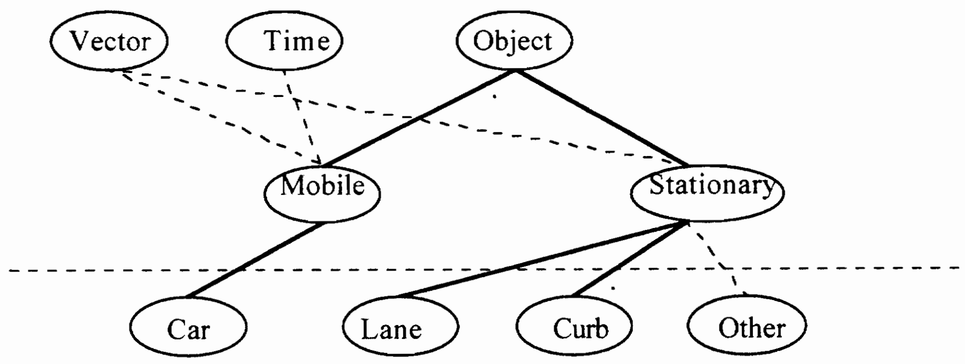
Like the rectangular class this class may be used to model actual objects or abstract objects.

1.2.6 Class Object

Objects in all simulations are uniquely identifiable, have a mass and are either mobile or stationary. For example, in the automated highway problem each automobile has a unique identification number, and a mass to get the momentum which is useful in calculating the braking force needed to stop the automobile in time. Stationary objects do not have any perception of time, where as, mobile objects have time associated with them. Objects are the entities around which all simulations revolve.

```
class object {
protected:
        int     Num;                        // Every object has a unique number.
        double  Mass;                       // Every object has some mass
        char    Type;                       // Type of object
public:
        object(){}                          // Constructors
        object(int num, char otype) {Num= num; Type = otype;}
        object(int num, double mass, char otype)
                        {Num= num; Mass = mass; Type = otype;}
        ~object () {}                       // Destructor
        // These functions may be redefined in the specific classes
        inline virtual int GetNumber() {return Num; }
        inline virtual double GetMass() {return Mass; }
        inline virtual char GetType() {return Type; }
};
```

Objects in any space time simulation will be derived from either the stationary or mobile instance of the object class. Fig. 2 illustrates for the automated highway problem how cars, lanes, curbs derive from one of the above mentioned sub classes of class object.



Various objects in Automated Highway Problem deriving from Object Class

Fig. 2

1.2.7 Stationary Objects

Class stationary derives from class object and has the following declaration:

```
class stationary : public object {
protected:
        vector  Svec;                   // Stationary vector
                                        // A pair of x and y components
public:
        stationary();                   // Constructor
        ~stationary ();                 // Destructor
};
```

1.2.8 Mobile Objects

Class mobile derives from class object and has the following declaration:

```
class mobile : public object {
protected:
        vector          Vec;            // A vector : Denoting the relative velocities in the
                                        // x and y direction.
                                        // The position of an object is derived from the shape
                                        // class.
        Abs_time        Time;           // This time is local to the object and is used to project
                                        // the time in future when  this mobile object could
                                        // collide with other objects.
        Abs_time        Atime;          // This is the absolute time that has elapsed since the
                                        // beginning of the simulation.  See section 6
public:
        mobile();
        ~mobile ();
        inline virtual void SetInitTime();
        inline virtual void SetTime(double t);
        inline virtual void SetInitVel();
        inline virtual void SetVec(double x, double y);
        inline virtual vector GetVec();
        inline double GTime();
};
```

1.2.9 Class Space

Class space is an abstract base class whose purpose is to efficiently help solve the problem using the basic strategy of divide and conquer. Class space helps in dividing the entire problem space into smaller sub-spaces, so instead of dealing with all interactions among all objects the system deals with interactions among objects within a particular sub-space. Subdivision of the space is application dependent and may also be computation dependent, as in the case of parallel computing.

For the robot arm collision detection system and the pool game simulation the space class could dynamically subdivide into smaller spaces if there are more than "n" objects within the space and then reunite if there are less than "n" objects in the newly formed sub-spaces.

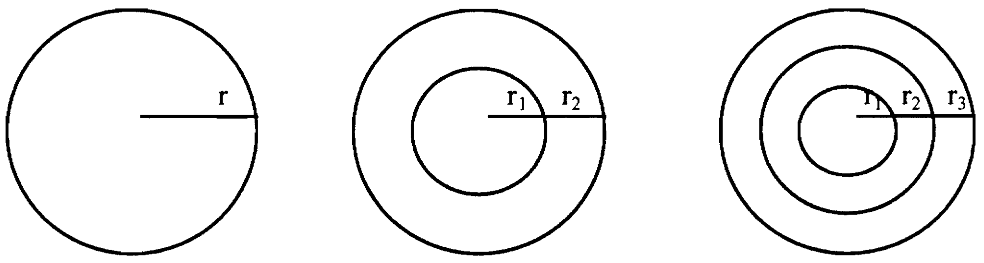
The following attributes and methods have been defined for the space class:

```
template <class SpaceShape>
class space : public stationary
{
        //      An efficient and easy way to manipulate objects
        LList<int, obj_lst>* SpaceLst;          // list of sub spaces
        int             ParentNum;              // parent node number
        int             obj_ctr;                // Object key counter
        int             spc_ctr;                // Space key counter
public:
        LList<int, obj_lst>* ObjectLst;         // list of objects
        Rule*           space_rule;             // Space rule(s)
        SpaceShape      sp_shape;               // Space shape type
        int             Subdivided;             // Subdivided flag
        int             ToBSubdivided;          // To be Subdivided flag
                                                // Constructor
        space(int num, int df, int sf, int nd, int pnum)
        {
                Num = num; obj_ctr = 0; spc_ctr = 0;
                Subdivided = FALSE; ToBSubdivided = FALSE;
                ParentNum = pnum;
                ObjectLst = new LList<int, obj_lst>(0, obj_lst(0, 'n'));
                SpaceLst = new LList<int, obj_lst>(0, obj_lst(0, 'n'));
                space_rule = new Rule(df, sf, nd); // if sf is FALSE other two do not count

        }
        space() { }
        ~space() { }                            // Destructor
        void define_shape(SpaceShape& m) {sp_shape = m;}
                                                // Assign any shape to the space
        void sp_init_rule(){space_rule->init();}// Initializes the rule as per the space
        int GetParent(){return ParentNum; }     // Return the parent node number
        void sp_show_rule() {//print}           // View the rule
        void ShowObjects()                      // Procedure used to test space lists
        void sp_clear_space_lst()               // Clear the space list
        void sp_clear_objects()                 // Clear the object list
        void subdivide(int num_div)             // To subdivide a space into n parts
                                                // New subspaces will have the same
                                                // shape and set of rules.
        void TransferObject(obj_lst tx_obj, double otime)
                                                // To transfer objects to the correct space on
                                                // colliding with the space that contains them.
                                                // The space goes through its siblings.
        void AssignObjects(int num_div)         // Method to assign objects to the correct
                                                // subspaces after a parent space has divided.  It
                                                // goes through the parents object  list and
                                                // assigns these objects to the corresponding
                                                // subspaces.
        void reunite(int num_div)               // This method reunites a dynamic space.  It
```

```
                                       // concatenates the subspaces objects lists to
                                       // form the parent spaces object list.  It further
                                       // clears the sub-spaces and deletes them.
        void wrap_up()                 // Remove a particular space from the list.
        void InsertObject(obj_lst new_obj)   // Insert object into the object list
        void RemoveObject(obj_lst old_obj)   // Remove objects from the object list
        int NumSpaces()                // Return number of spaces in the space list
        int NumObjects()               // Return the number of objects in a spaces
                                       // object list
};
```

The user has the flexibility to define the shape of the space by using this shape class as a template to the space class. Shape classes have the contain procedure which lets the space know what objects lie within its domain. Within the space class the user can either subdivide a particular shape or reunite it. Fig. 3 demonstrates a way of subdividing and reuniting a circularly shaped space.



r = Radius of Circular Space | $r = r_1 + r_2$ | $r = r_1 + r_2 + r_3$

Original Space | Subdivided into two subspaces | Subdivided into three subspaces

Subdivision of circular space

Fig. 3

1.2.10 Class Object List

This class helps maintain a list of objects to be used by any other class which needs to maintain information about a group of objects. For example in automated highway problem, space needs to keep a track of all cars within its boundary.

Class object list has the following attributes and methods:

```
class obj_lst {
public:
      int obj_num;              // Object Number
      char obj_type;            // Object Type
                                // Constructors
      obj_lst(int num, char otype) { obj_num = num; obj_type = otype;}
      obj_lst() {}
     ~obj_lst() {}              // Destructor
      };
```

1.2.11 Class Rule

Class Rule helps define the set of rules or conditions according to which space class would behave. It provides encapsulation and abstraction for these governing rules and allows rules to be accessed by another class. For example, rules govern how space decomposes itself into smaller subspaces or spaces reunite to form a bigger space. The following attributes and rules have been defined for this class:

```
// The rule class to store the rules that govern the nature and actions of the space

class Rule {
        int     Decomposition_Factor;      // Subdivide after these many objects
        int     Subdivision_Flg;           // Flag to check if space subdivides
        int     Num_Divisions;             // Number of subdivisions
public :
        Rule() {}                          // Constructors
        Rule(int d, int s, int n)  {
                Decomposition_Factor = d;
                Subdivision_Flg = s;
                Num_Divisions = n;
        }
        ~Rule() { }                              // Destructors
                                                 // Utility Get methods
        inline int GetDF () {return Decomposition_Factor;}
        inline int GetSF () {return Subdivision_Flg;}
        inline int GetND () {return Num_Divisions;}
        inline void init ()                      // Initialize
        {Decomposition_Factor = 0; Subdivision_Flg = 0;Num_Divisions = 0}
};
```

1.2.12 Class LList

This class helps define a doubly linked list of user definable nodes. Each node has a key and a value. This class is used to maintain a list of various objects and can be used by any class. For example, the space class in the pool game simulation maintains a list of all balls within its boundary, each space cell in the automated highway problem uses this list to track all automobiles within its boundary.

```
template<class K, class V> class LList;
template<class K, class V> class Link {
                                    // Friendship ensures that Links can be created,
                                    // manipulated, and destroyed only by appropriate
                                    // Llist class.
        friend class LList<K,V>;
private:
        K key;                      // Each link holds a key value pair
        V value;
        Link* pre;
        Link* suc;
public:
        Link (const K& k, const V& v) : key(k), value(v) { }
        ~Link () {delete suc; }     // recursively deletes all links
};
// Each list has a head, tail with default values. Current points to the currently accessed
values.
template<class K, class V> class LList {
        Link<K, V>* head;           // pointer to the head
        Link<K, V>* current;        // pointer to current
        Link<K, V>* tail;           // pointer to the tail
        V       def_val;            // Default value value
        K       def_key;            // Default key value
        int      sz;                       // Size of the list
                                           // Initialize
        void init()      { sz = 0; head = 0; current = 0; tail = 0; }
        public :
        LList() {init ();}                 // Constructor
        LList(const K& k, const V& d)// Constructor with args
                : def_key(k), def_val(d) { init(); }
        ~LList() {delete head; }           // delete all links recursively
        V& find(const K& k);               // find V corresponding to K
        void insert(const K ky, V val); // insert an element
        int size() {return sz; }           // return the size of the list
        void clear() {delete head; init (); }
                                           // clear the size
        void show()                        // print values to standard out
        void remove(const K& k);           // Remove element with key k
        LList<K,V>* concatenate(LList<K,V>* new_lst, LList<K,V>* add_lst);
                                           // Concatenate two lists
        int FillUp(const K& k);            // To fill up gaps created by removes
};
```

The above mentioned classes are intended to model space-time problems in the defined problem domain. These classes can be used as base classes with little or no modification, and problem specific classes can inherit from these classes. Let us now apply this class hierarchy to the three problems; pool game simulation, robot arm collision detection system, and automated highway problem.

2.0 An Object Oriented Simulation of the Pool Game (OOSPG)

OOSPG is a discrete event driven, conservative object oriented simulation [Hont89] of the pool game. In the simulation each “interaction” is considered an event and these interactions govern the simulation. A interaction could be a ball colliding with one another ball, a ball colliding with a wall, a hole, the cue, or any space boundary.

In OOSPG at the end of every simulation step collision times are calculated between various interacting objects within a space. Each space thus has a minimum collision time for itself. The least of these collision times for all spaces becomes the time to which each space advances on the next simulation step. Within each space, all mobile objects in the simulation are projected forward in time by this minimum collision time. At this point again collisions times are calculated for all the spaces for the various objects within their boundaries. The collision that will occur first among all the spaces decides the next simulation time step and so on.

The "universe of discourse" of OOSPG contains moving balls, stationary holes, vectors, and line segments(walls). Dynamically, the balls behave like perfectly elastic objects of possibly varying radius and mass. Dynamically, points and lines behave like stationary obstacles with infinite mass, that is, balls rebound elastically from them without disturbing them. The implementation employs simple kinematic and dynamic principles such as conservation of momentum and energy. There is total conservation of energy and no table friction is assumed.

2.1 The Simulation

At the beginning of the simulation all objects are at rest and the absolute time is zero. All objects are displayed on the screen and the user can impart the initial force to the cue ball using the cue. This force is in the direction of the cue and the magnitude of the force is proportional to the length of the cue as drawn by the user. At this point the cue ball is the only moving ball and its collision times are calculated with all balls, walls, holes and space(if applicable). The minimum of these times becomes the simulation time step. All objects that will not participate in a collision are moved forward in time by this time step. Collisions are resolved for all objects that participated in a collision and these objects have new vectors associated with them describing their new velocities and direction. These objects are also projected forward in time. All objects are then redrawn on the screen to give the user an illustration depicting things as they would happen in real time. There are two versions of the simulation:

- **Version I** : The entire table is considered one space. Space here is purely abstract. Collision is checked for all moving objects against all objects in the simulation. This version serves as the benchmark for version II, which uses a dynamic space. Section 5 "Evaluation" compares the two versions.

- **Version II** : The table is divided into dynamic spaces depending on the rules the user decides. For instance, space decomposes itself into smaller subspaces depending upon the traffic/congestion within it. If there are more than *'n'* objects within a space, then the space subdivides itself into '*m*' divisions. The decomposition factor 'n' and the number of divisions 'm' depend on the nature of the problem [Shaf90]. Both 'n' and 'm' can be specified by the user.

The current dynamic version running of OOSPG has two dimensional rectangular spaces. These spaces subdivide into four equal sub-spaces. This enables the space to be maintained by a quad tree structure [Chie89] [Shaf89] with each node representing a space instance. The user can specify the decomposition factor, that is, when the space contains more objects than specified by the decomposition factor, subdivide the space into four smaller subspaces.

In version I, where there is one space, each object has to check for collision against all other objects. Say if there are 'n' objects on the pool table each object checks for collision with 'n-1' objects. This is a O(n^2) process [Nico90]. Now, by subdividing the space into smaller sub spaces depending on a certain set of rules the following is achieved.

Each space provides to all mobile objects within it a list of other potential objects with which they can collide. A object now checks for collision with utmost 'm' objects, where 'm' is the decomposition factor[1]. Assume that there is an equal distribution of objects among the 's'

[1] If there are more than 'm' objects in a space, the space subdivides itself.

sub spaces and within each space there are ‘m’ objects leading to m^2 collisions checks. Since, there are ‘s’ spaces the total number of comparisons is $s* m^2$. As ‘n’ objects are equally divided into ‘m’ parts in ‘s’ spaces so n=m*s. The order of the algorithm would thus be $O(s * m^2)$, or $O(n/m* m^2)$, that is, O(n*m), where ‘m’ as defined earlier is the decomposition factor. Ideally, for m = 1, that is, one object per space in the simulation we would have an O(n) algorithm. It seems that the finer the granularity of objects within a space the better performance we would get, but that is not quiet the case as explained in the “Evaluation” section (Section 5).

The details of how the space nodes are maintained, how space maintains its object list, and the algorithms for subdivision and reunification of the space are in the space class details in section 2.6.1.

2.1.1 Termination of the simulation step

Each simulation step terminates either naturally or is terminated unusually. Remember the system has no table friction.

- *Rest state termination*

If all moving balls in the system collide with holes.

- *Time termination*

Time Termination : The simulation step is stopped after a specified amount of time, that is, at that point in time all moving objects stop at their current positions.

After every simulation step the user can impart force to the cue ball again and continue.

2.1.2 Graphical User Interface

OOSPG uses X-windows XT Intrinsics Release 4.0 for its graphical user interface [Tarl92] [Bart86]. Upon execution the system places balls on the table in user pre-defined locations[2] .

[2] See Future Work : *Better User Interface*

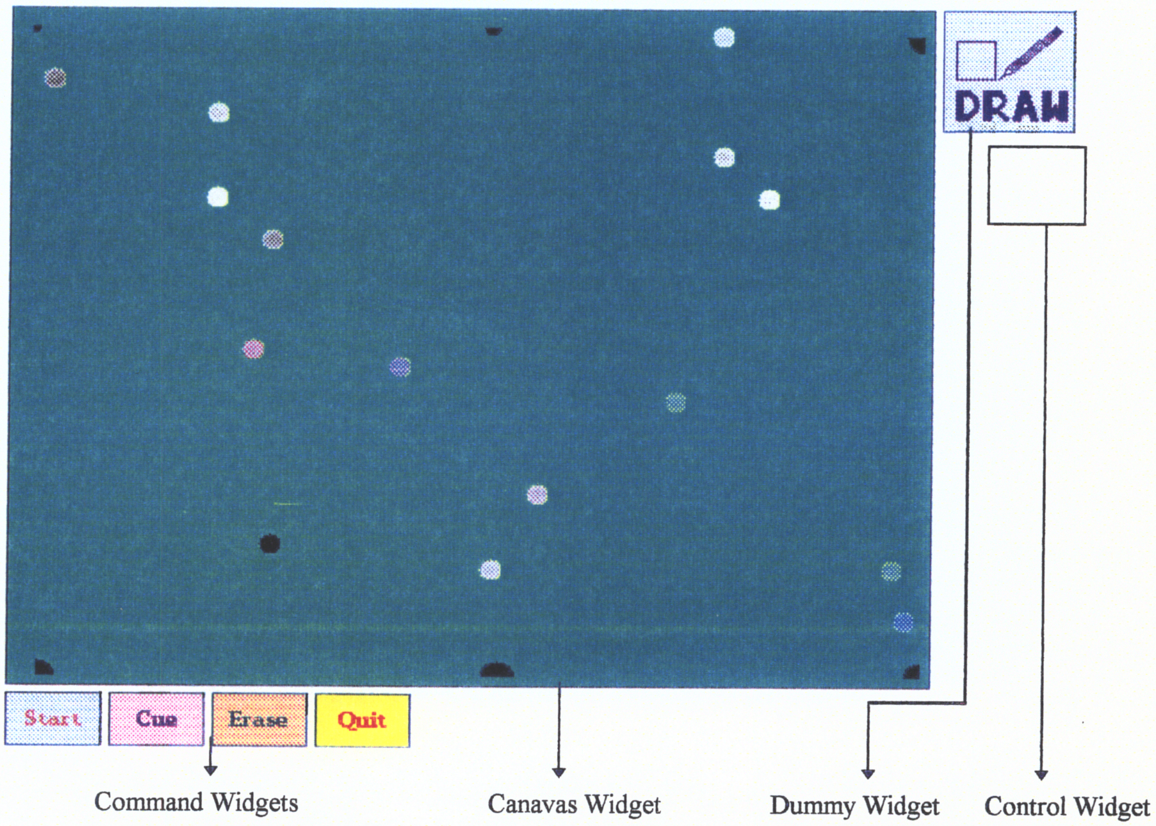
OOSPG then draws the space boundaries on the screen (if applicable). The user can now impart any amount of force to the cue ball in any direction by means of the cue. To define a cue the user uses the third (rightmost) mouse button. The user presses the third mouse button at the starting location, and drags the mouse (while the third button is pressed) to the desired ending location. Once the user lets go of the third mouse button the definition of the cue is complete. The amount of force imparted to the cue ball in X and Y direction is proportional to the cue vector. The direction given to the cue ball is same as the slope of the cue line[3] . Both force and direction are bundled into the vector of the ball which it receives from the cue on click of the "Done" button in the control widget box (on the upper right hand corner of the screen) as shown in Fig. 4. After the user has clicked on the "done" button the entire simulation is displayed on screen as it would occur in real time. The user can view the balls are moving, colliding with each other, with walls, holes, and space boundaries (if applicable).

The user has the capability to stop after any simulation step and start over from the initial configuration. He can further stop the simulation at any time and exit the system.

OOSPG provides the basic display infrastructure and widgets, which can be modified/upgraded to fit most simulation. Fig. 4 shows the widget classes available.

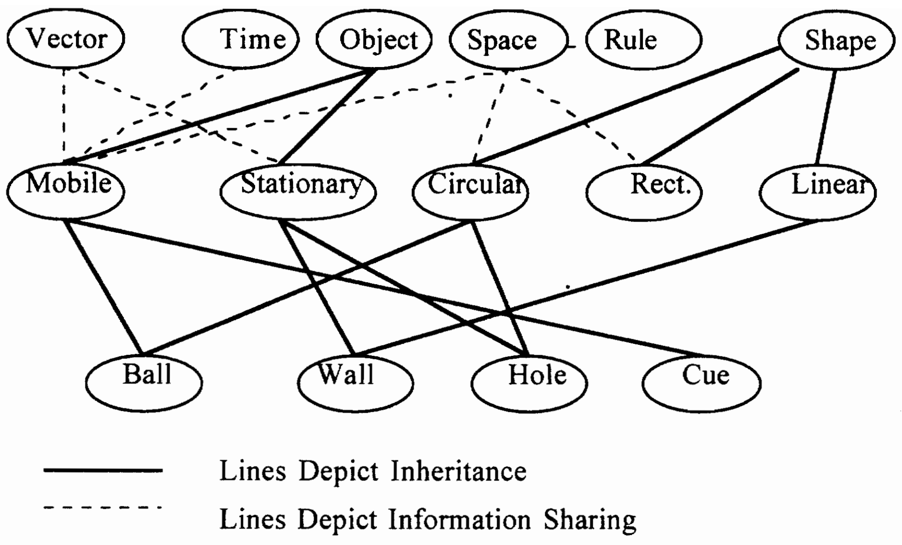
The following problem specific classes are used in OOSPG. All of these classes derive from one or more generic classes discussed in section 1.2. The complete class structure/hierarchy used in the OOSPG is shown in Fig. 5. As seen in Fig. 5 class ball inherits from classes mobile and circular, class wall from classes stationary and linear, class hole from classes stationary and circular, and class cue from class mobile.

[3] See Cue Class for details



OOSPG with its objects as it looks on the screen. All the widgets used are listed.

Fig. 4



Class Hierarchy for OOSPG

Fig. 5

2.2 Ball

In the pool game simulation balls are considered perfectly elastic circles and inherit from class mobile and class circle (multiple inheritance). Class ball inherits mass, and number from class object, vector from the class mobile, and center and radius from the class circle. Since balls are mobile objects that participate in collisions, they contain collision detection and resolution mechanisms. Class ball has the following declaration:

```
enum pattern {solid, stripped};                         // Possible patterns of the balls
                                                        // Solid is 0, stripped is 1
enum color {white,  yellow, purple, red, blue, orange, green, maroon, black};
                                                        // Possible colors of the balls
class ball : public mobile, public circle {
        color   col;                                    // Color of the Ball.
        pattern pat;                                    // Pattern of the Ball.
        int     moving;                                 // To check if ball is moving.
public:
        inline ball();                                  // Constructor
        inline ball(pattern p, color c, int num);       // Constructor with arguments
        inline ~ball();                                 // Destructor
        inline color GetColor();                        // Returns the color of the ball
        inline pattern GetPattern();                    // Returns the pattern of the ball
        inline void GetGoing (ball *bg);                // Moves the ball
        inline void StopBall (ball *b);                 // Stops the ball
        inline int IsMoving();                          // To check if ball is moving
        inline void SetPosition(int x, int y);          // To place a ball
        inline point GetPosition();                     // To get the center of the ball
        inline void SetMassNRadius(double x, int i);    // To set mass and radius
        inline void SetOType(char o_type);              // To set ball type
        inline void Assign(ball *t);                    // To copy a ball
        inline vector GetMomentum();                    // Returns momentum of the ball
        inline double GetVel() { return Vec.Mag();}     // Returns velocity of the ball
        inline double GetEnergy();                      // Returns energy of the ball
        inline point WhereBallAtT(const Abs_time t);// Where would the ball be at
                                                        // absolute time t
        inline point WhereBallAtDeltaT(const Abs_time t); // Where would the ball be at
                                                        // delta time t
        inline void MoveBall(const Abs_time t);         // Move ball to time t
        inline void MoveBallDeltaT(const Abs_time t);// Move ball to time delta t
                                                        // Collision Detection
        collision CollD_BallCenter_Line(linear l);      // Ball center and line l
        collision CollD_BallEdge_Line(linear l);        // Ball edge and line l
        collision CollD_BallEdge_Point(point p);        // Ball edge and point p
        collision CollD_Ball_Ball(ball *b1, ball *t);   // Ball and ball
        collision CollD_Ball_LineDepart(linear l, int d);// Ball and departing line
        collision CollD_Ball_PointIntersect(point p); // Ball and point intersection
        collision CollD_Ball_LineIntersect(linear l);   // Ball and line intersection
                                                        // Collision Resolution
        collision CollR_Ball_LineSeg(linear l);         // Ball and line l
        collision CollR_Ball_Ball(ball *b, ball *t);    // Ball and ball
        collision CollR_Ball_Point(point p);            // Ball and point p

};
```

Why do the collision detection(CD) and collision resolution(CR) methods reside in the ball?
It was observed during the design and development of objects and space classes that if CD and CR methods are within the space, the space needs to know the position, velocity, number, type of the objects contained within it. If the CD and CR methods are within the object itself, it allows for data encapsulation. The object knows what other object(s) it can collide with and has CD and CR methods built into it. As both CD and CR are application specific, it further suggests for CD and CR methods to be part of an application specific class.

Since balls take part in all collisions a discussion about collisions, their detection and resolution follows.

2.2.1 Collisions

The simulation needs to determine whether, when, and where several types of collisions will occur. Therefore, there is a collision data type containing the answers to "whether" and "when". There is no single answer to where since its meaning is different for each geometrical object type defined.

```
typedef struct
{
        AbsTime         at ;
        int             yes ;
}
   collision ;
```

The 'yes' attribute of a collision value will have the value YES if the collision occurs, and in this case, the 'at' attribute of the collision value will express the absolute time at which the collision occurs. The values YES and NO have the property that they can be used like Boolean values in conditionals and tests, allowing code to be shorter and simpler. For example, one might write:

```
Collision  my_collision ;
  ... find out whether the collision occurs ...
if ( my_collision.yes )
{ ... compute consequences of the collision ... }
```

2.2.2 Collision Detection And Resolution

Collision detection [Fuji86] and resolution play a vital role in this and many other applications. This section looks at these in detail with respect to the objects in OOSPG.

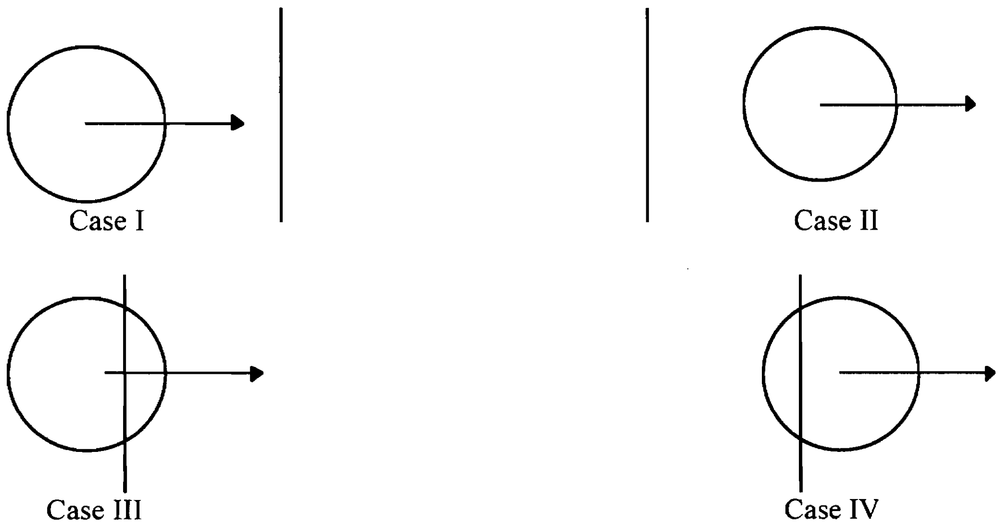
Ball and Line

If a moving circle crosses a stationary line segment the CENTER of the circle will touch the line segment at a particular instant of time, measured by an AbsTime value. The answer to "whether" and the answer to "when" are neatly packaged in a Collision value returned by the method CollD_BallCenter_Line (the method CollD_BallEdge_line predicts times at which the EDGE of a circle crosses a line).

Note that the collision might occur at a time in the past of the current time of the circle, that is, the 'at' attribute of the Collision value might be less than the 'at' attribute of the Circle input value. This condition will occur when the circle is heading away from the segment. Such cases should not be considered as collisions with the least collision times as the collisions have already occurred in the past.

When a moving circle is heading towards a line segment, the edge of the circle crosses the segment at a certain time, different from, and, in fact, earlier than, the time at which the center of the circle crosses the segment.

There are several cases (See Fig. 6) to explain what is meant by collisions between circles and line segments in OOSPG and to explain how these methods behave.



Various cases of collision between a circle and a line

Fig. 6

- Case I : A collision will be predicted at the moment the leading edge of the circle crosses the line segment. The predicted time will be greater than the reference time of the circle.

- Case II : No collision will be predicted by the method because the circle is regarded as moving away from the line.

- Case III : A collision WILL BE PREDICTED, because the circle is regarded as heading towards the line since its center is heading towards the line. However, the collision will be predicted for the time at which the leading edge of the circle crossed the line in the past, that is, the predicted time of the collision will be less than the current reference time of the circle.

- Case IV : No collision will be predicted because the circle is regarded as heading away from the line since its center is heading away from the line.

The final subtlety concerning the definition of circle and line segment collisions concerns collisions between the circle and the endpoints of the segment. The current method only predicts collisions where the line segment becomes tangent to the circle. Cases where the circle will strike the endpoints but will never lie tangent to the line segment are ignored by this method, and it predicts no collision. However, such cases can easily be taken into account by a method whose purpose is to predict collisions between circles and points. Such a method is discussed later.

Ball and Point

The edge of a circle can collide with a stationary point in the plane. The method CollD_BallEdge_Point predicts the time of such a collision and returns a collision type. To obtain this time we define a temporary vector 't' between the center of the circle and the point. Then if the discriminant 'd' is positive the circle will collide with the point. The term d is calculated as follows:

a = Squared magnitude of the circles vector 'Vec'

b = -2 * Cross product of the vector 't' & vector 'Vec'

c = Magnitude of 'Vec' - (square of the balls radius)

$d = b^2 - 4*a*c$

To calculate the time of collision we find the real roots and take the minimum of the two real roots. So,

$Root_1$ = -b + square root (d) / 2*a

$Root_2$ = -b - square root (d) / 2*a

The time of the collision then would be : Time + Minimum($Root_1$, $Root_2$) . The actual program follows:

```
collision ball::CollD_BallEdge_Point(point p) // Between ball edge and point p
{
collision Answer;
vector  Tmp(Center, p);  // Tmp : A distance vector between, circle and point p
double  a, b, c, d;
double  Mag;
double  R1, R2;
double  TmpR;
Answer.yes = YES; Answer.at = 0.0;
a = Vec.MagSqr();
b = -2.0 * (Tmp & Vec);
Mag = Tmp.MagSqr();
c = Mag - (Radius*Radius);
d = (b*b) - 4 * a * c;
if (d<=0)          // No hit or just grazing
{
 Answer.yes = NO;
 return Answer;
}
//  Case of two real root, take min. to be the time of the first collision
TmpR = sqrt(d);                  // More efficient implementation
R1 = ( (-b) - TmpR) / (2*a);
R2 = ( (-b) + TmpR) / (2*a);
if (R1 < R2) R2 = R1;
Answer.at = Time + R2;
cout << " Time collision at : " << Answer.at << "\n";
return Answer;
}
```

Ball and Ball

Pairs of circles can collide, also, and CollD_Ball_Ball method predicts the time of such a collision. CollD_Ball_Ball method uses the above method and is as follows :

```
collision ball::CollD_Ball_Ball(ball *b1, ball *t)// Between ball and ball
{
point   p;
        t->MoveBall(b1->Time);         // Move the temp ball to current time
        t->Vec = Vec - b1->Vec;        // Define the vector of temp ball to be the
                                       // difference between the vector of the ball calling
                                       // the method and the ball with which the collision
                                       // is being checked.
        t->Radius = Radius + b1->Radius; // Set the radius of the temp ball to be the sum of
                                       // the two radii
        p = b1->Center ;               // Point p is the center of ball b1
        return t->CollD_BallEdge_Point(p);    // Check for collision between temp ball and p.
}
```

The previous three methods do not have any side effects on the circles. They serve only to predict whether and when collisions occur. When dynamical effects of a collision on the objects are considered, side effects on values come into play. Dynamical effects change circle velocities and take into account the masses of objects. The next method treats a line segment as an infinitely massive fixed object, like a granite wall, and bounces a given circle off the wall. These methods modify the reference time, the position, and the velocity vector of the circle. These methods also answer the "whether" and "why" questions and returns an appropriate Collision value. If the collision does not occur, the input data will not be disturbed. In Case III, discussed above, where a collision is predicted in the past of the circle, the circle is backed up to the time where its leading edge first touched the granite wall before the dynamical effects on the velocity of the circle are calculated. The methods are discussed below:

This effect of backing up the circle can lead to unexpected results if one is not careful in the use of this method.

```
/*
Between ball and a departing line segment. This method returns the collision structure which
tells if a ball has departed from the specified line. A departure is defined as the last point on
the edge of the ball crossing away from a line segment. This function is used to calculate the
position and time when a ball moves away from a space boundary.
*/
collision ball::CollD_Ball_LineDepart(linear l, int d)
{
collision       when;
vector          v;
vector          dv;
switch (d)
{
  case POSITIVE_ONE:
          v = l.Vl * (Radius * 2);
        break;
  case NEGATIVE_ONE:
          v = l.Vl * (-Radius * 2);
        break;
  default:
          when.yes = NO;
          when.at = 0.0;
        return when;
}
linear   displaced(v.OtherEndPoint(l.Pl), l.Vl);
dv = displaced.Vl.Unit() * Radius;
linear  NL((dv.Reverse()).OtherEndPoint(displaced.Pl),displaced.Vl + (dv+dv));
vector           k(Center, NL.Pl);
if (((Vec & NL.Nl) * (k & NL.Nl)) <= 0.0 )
{
  when.yes = NO;
  return when;
}
when = CollD_BallEdge_Line(NL);
if (when.yes == NO)
  return when;
MoveBall(when.at);                    // side effect on the ball c
return when;
}
```

The following method is similar to the previous except that it produces the effects of a collision on a pair of circles. It also has side effects on its input circles, returning to the caller the positions and velocities of the two circles after a rigid- body collision. It sets the reference times of the two circles to the collision time, and the position and velocity attributes of the circles to their values after the collision. Note that if the collision does not occur, the input circles are not disturbed.

```
/*
This method returns the collision structure which tells if a circle edge has intersected with the
specified point. If an intersect does occur, the circle's position is updated to that
intersection. But unlike the collision function the vector of the ball is unchanged. This is
used in calculating the time when a ball crosses space boundary.
*/
collision ball::CollD_Ball_PointIntersect(point p)
{
collision when;
 when = CollD_BallEdge_Point(p);
if (when.yes == NO)
   return when;
MoveBall(when.at) ;                   // side effect on ball
return when;
}
```

The following method returns the collision data structure which tells the caller whether and when the ball will intersect with line segment (l). What is meant by intersect here is the time at which the first edge point of the circle touches a point on the line segment. If an intersect does occur, this function updates the ball's state. Intersection is different from the collision functions here in that the intersecting circle does not bounce of the line segment but passes through the line segment.

```
/*
This method returns the collision structure which tells if a circle edge has intersected with the
specified line. If an intersect does occur the circles position is updated to that intersection.
But unlike the collision function the vector of the ball is unchanged. This is used in
calculating the time when a ball crosses space boundary.
*/
collision ball::CollD_Ball_LineIntersect(linear l)
{
collision when;
when = CollD_BallEdge_Line(l);
if (when.yes == NO)
  return when;
MoveBall(when.at);                    // side effect on ball c
return when;
}
```

Finally, for dynamical effects, a point is considered to be an infinitely massive barrier from which circles rebound elastically. The method calculates the effects of such a collision and deposits the appropriate values in its input circle buffer, and computes and returns a Collision type.

2.3 Cue

Class cue is used to impart the initial force to start the simulation. The cue class derives from class mobile. It has the following declaration:

```
class cue : public mobile{
        line Ln;                // line not linear
public:
        cue(line CueStick);
        ~cue ();
};
```

The user imparts force, and gives direction to the cue ball using the cue. To do so he draws a cue using the mouse. Details on how to draw and define the cue are in the Graphical User Interface section (2.1.2).

The amount of force imparted to the cue ball in X and Y direction is proportional to the magnitude of the difference in the X and Y coordinates of the starting and ending point, that is, say the start point has coordinates (x_1, y_1) and the and point has coordinates (x_2, y_2) the magnitude of the vector given to the cue ball will be (dabs[4] (x_2-x_1), dabs(y_2-y_1)).

The direction given to the cue ball is same as the slope (y_2-y_1)/(x_2-x_1) of the cue line.

2.4 Hole (Pocket)

Class hole derives from classes stationary and circular. It has no attributes. The constructor of the hole class is given below. It inherits all its attributes from its base classes.

```
class hole  : public circle, public stationary{     // Multiple inheritance from circle and
stationary
public:
        hole (point c, int r, int n);
        ~ hole ();
        inline point GetPosition();
};
```

A hole is considered as a point object in the simulation. The ball method CollD_Ball_Point is used to project collision of balls with holes. After colliding with the hole the ball is removed from the simulation.

2.5 Wall

Class wall derives from classes linear and stationary. It has the following declaration:

```
class wall : public linear, public stationary{
linear  Ln;
public:
        wall(point p, vector v, int i);          // Constructor
        ~wall ();                                // Destructor
        inline linear GetLine();                 // Returns a line
};
```

[4] Returns the double absolute value

A line segment is represented by one endpoint, e, a unit normal vector, n, and a vector, l, pointing from the endpoint to the other endpoint[5] , the other endpoint is implicit and unexpressed; it can be found by calling the Vector method "OtherEndPoint".

2.6 Space

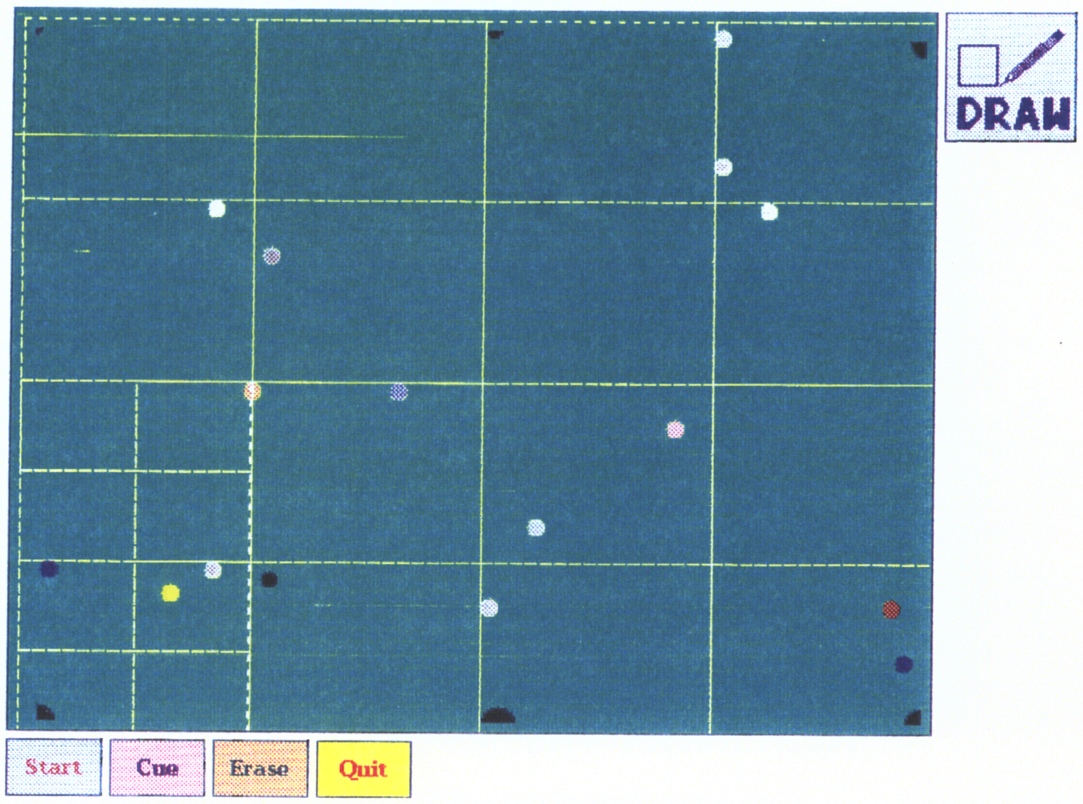
It is useful to subdivide the entire problem space into smaller subspaces. As pool balls on one end of the table will first collide with objects around them rather than objects on the other end. So, we should limit our collision detection system to within the subspace. Similarly, cars moving on the highway would want to guard against cars close to them rather than those far away.

It is important to take into consideration the dynamic nature of the application in selecting an appropriate decomposition factor (the number of objects based on which either the space subdivides or reunites). “Evaluation” section (5) has a comparison of number of collision checks versus different decomposition factors.

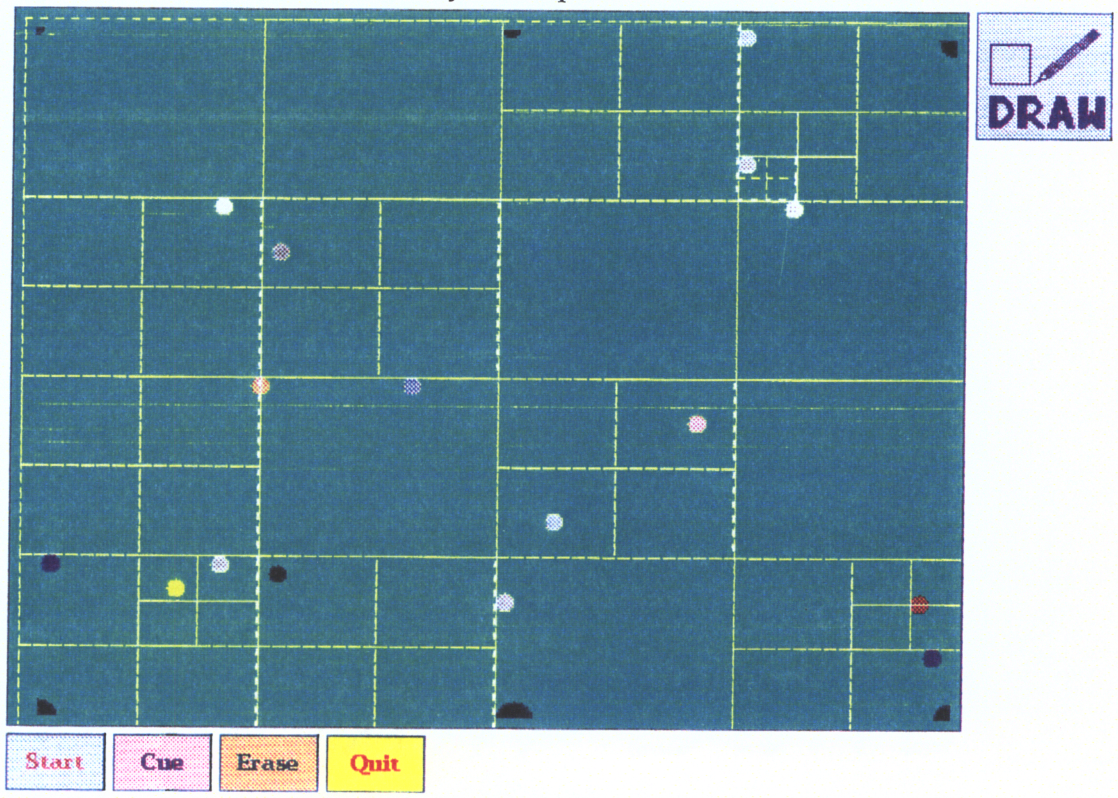
At the beginning of the simulation each space determines the initial position of all the objects and forms its initial object list. While the space is inserting objects into its list, if there are more objects than the decomposition factor the space calls its subdivision method and decomposes into smaller spaces. At this time all the objects are redistributed among these newly formed sub spaces. If an object is lying at the intersection of two or more spaces, all involved spaces have the object in their respective lists. Once all the rules specified by the rule class are meet and all space lists are formed, the space is ready.

Any time an object moves, it obtains from its space a list of all the objects with which it can collide. Each moving object in the space checks for collision with all other objects in the space list and with the boundary of the space itself. Whenever an object collides with its space boundary, the space removes the object from its lists and calls the transfer method to assign this object to the appropriate new space. Then the space asks its parent space to check if it needs to

[5] See Class linear for details



Dynamic Space with df = 4



Dynamic Space with df = 1
Fig. 7

recombine as an object has left its domain. The parent node in turn checks against the set of rules and reunites if needed. On the other hand the space that received this new object also checks against its rules to see if it needs to subdivide into smaller subspaces. This process continues during the course of the simulation.

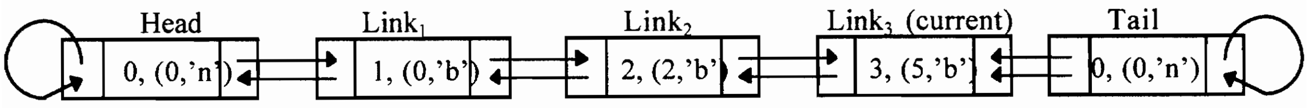
Let us now look at various type of spaces and how they can be used in various applications.

2.6.1 Dynamic vs. Static Space

A simulation could use one or more static subspaces, or dynamic subspaces which subdivide or reunite as objects cross space boundaries. The space class is designed to allow the above mentioned flexibility. The user can also dictate the rules for decomposing this space e.g. number of sub divisions, decomposition factor (after how many objects does the space subdivide) etc. Care has been taken to make the space class as flexible as possible to ensure adaptability to a wide class of space time simulations.

Fig. 7(a) shows a dynamic space in OOSPG with a decomposition factors of four (df = 4) and Fig. 7(b) shows a dynamic space with a decomposition factor of one (df = 1). The space boundaries, displayed in Fig. 7, show how the space would subdivide and reunite itself depending on the rules.

Each space class maintains the its and the space lists. These are doubly linked [Stro92] lists, where every link is a combination of a key and a value. Both the key and value are user definable classes. Each link maintains a pointer to its predecessor and successor. The list class has methods for inserting, finding, removing, and concatenating lists. It provides an efficient way of maintaining the space's object and subspace lists. Fig. 8 shows a sample space object list.



A sample Space Object List

Fig. 8

Each link in Fig. 8 is a combination of a key and a value. The first part of the link is the key and the second part, which is within parentheses, is the value (and combination of object number and object type). Each link in the space list has a unique sequential key except for the head and the tail which have a default key and a default value. The list of Fig. 8 has three objects of type Ball with number 0, 2 and 5, and with keys 1, 2 and 3 respectively. By using the above list class it becomes simple to maintain object lists for the space. For example, to find the type of the object with key = 2, one would say, find(2).obj_type. To concatenate lists add the size of $list_1$ to each link's key of $list_2$, make the predecessor of the tail of $list_1$ point to successor of the head of $list_2$ and then drop the tail of $list_1$ and the head of $list_2$.

3.0 Robot Arm Collision Detection System (RACDS)

3.1 Problem Definition

In NASA's proposed space station there are robot arms to move objects, perform repairs and construction, etc. [Shaf90]. The objective is to warn these real-time robot arms of imminent unintentional collisions. A collision could be between two robot arms or between a robot arm and other objects in the environment.

Each robot arm has seven degrees of freedom, with each section of the arm being nearly cylindrical in shape. The operating environment is dynamic - the system must accommodate movement of both arms and other objects. The motions of various objects are not pre-defined. The operating paradigm is one of receiving an indication of movement by certain objects, updating the representation of the environment to reflect that movement, and reporting any imminent collisions.

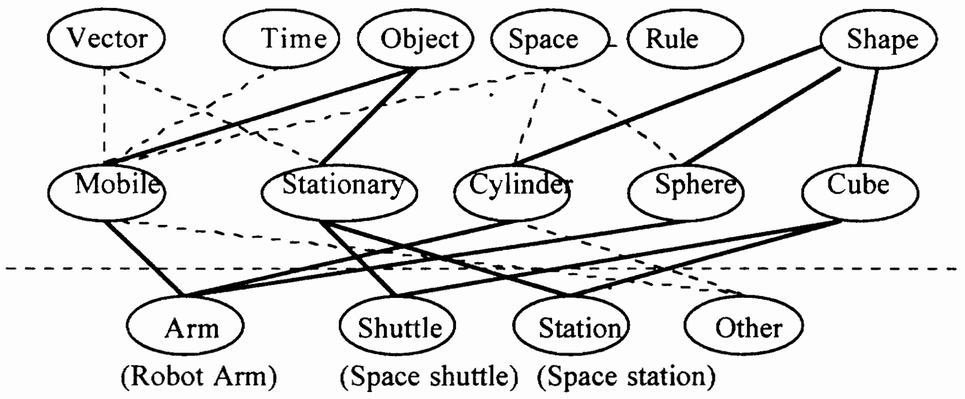
3.2 Solution

From the problem definition we see that RACDS has the following characteristics. First, we must determine in real time if robot arms/objects are about to collide. Second, the model must be constantly updated, adjusting to the movements of the robot arms and objects. Both updating and collision detection must be consistently performed within the permitted time period to be acceptable as a real time safety system. Third, the model must be reliable, but not necessarily exact. That is, since the objective is to warn and avoid imminent collisions, an exact representation of the objects is not required - as long as the approximation does not lead to missing imminent collisions, nor leads to reporting too many false alarms.

The approach to collision detection is to maintain a model of the working environment and through that model, detect when objects in the real world are about to collide. In effect, a real time simulation of the environment is performed. Information regarding the position of the robot arms is input to the system by placing sensors on the motors that drive various components of the robot arm. These sensors give indication to movements about to happen and the model is updated based on these inputs. Like in OOSPG each space calculates the collision time for all objects within its boundary. Among all the spaces if any collision time is “too short” an imminent collision is reported. As the space in RACDS is three dimensional so we would need an octree instead of a quad tree to model the space. The space shape would be of class cuboid.

Class vectors are used to represent motion of various objects and calculate their collision times. When a certain distance between two non-compatible objects is to be maintained, a standard technique for a static environment is to extend each object by 1/2 the minimum tolerance distance in all directions and then using vector arithmetic to determine any collisions that might occur.

For performance reasons the user can specify various rules to manipulate the space. As in OOSPG the space would subdivide into eight subspaces if there are more than ‘n’ objects within a space. Similarly eight adjoining subspaces would reunite if the total objects within their boundaries are ‘n’ or less. Since this problem is less dynamic than OOSPG (as the robot arm has seven degrees of freedom) and cannot traverse space boundaries rapidly a lower decomposition factor would give improved performance. This is explained in section 5 in detail.



The Generic Class Hierarchy applied to RACDS

Fig. 9

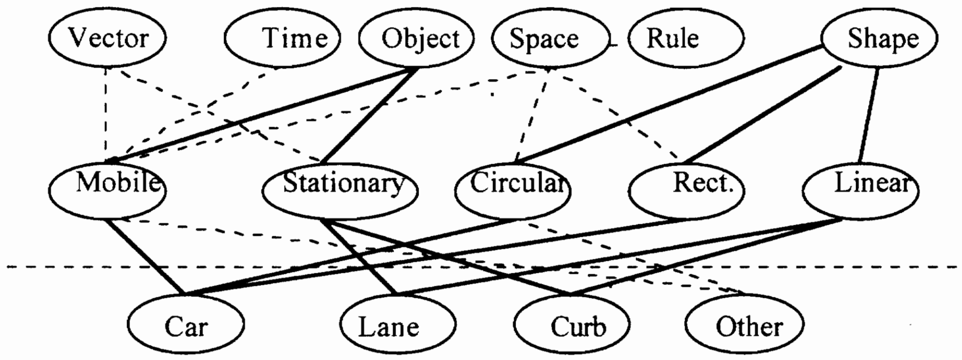
Fig. 9 shows the generic class hierarchy that could be used for RACDS. The robot arm inherits from mobile and various shapes. It has the collision detection system built into it. The space shuttle and the space station inherit from stationary and various shapes. Other mobile objects derive from class mobile and various shapes. Mobile objects, having vectors and time associated with them, are used to model and calculate collisions times. Upon receiving data from the sensors the objects that moved update their vectors. These new vectors are used by the collision detection algorithm. If a collision between two non-compatible objects is detected a warning is issued and the motion of the robot arms stopped. But if no two objects are close enough to report a collision all objects are moved forward in time and their vectors updated. This process continues as long as the robot arms are in use.

4.0 Automated highway problem (AHP)

4.1 Problem Definition

The intent is to devise a collision avoidance system that will warn the car drivers if the car is going to collide with other cars, with the curb, or even crossing the lane markers. These "smart" cars will look out for other cars, curbs and lanes, calculate their collision times with other objects and if the collision time is less than the "safe time", notify the driver of impending danger. The "safe times" are calculated based on the mass and velocity of the automobile. The greater the momentum (mass * velocity) the higher the value of safe time.

The generic class hierarchy will be used to model the automated highway problem and develop the collision avoidance system as shown in Fig. 10. All automobiles inherit from mobile and various shapes. Lanes and curbs inherit from stationary and linear. Other objects in the simulation would also inherit from various shapes and from either mobile or stationary.



The Generic Class Hierarchy applied to AHP

Fig. 10

4.2 Solution

The highway is divided into cells[6]. The cell is analogues to space in the pool game simulation and robot arm collision detection system. Each car checks for collision with other cars in the same cell. Cells have a list of all cars at all times within their domain.

The functionality of keeping a list of cars that belong to a certain cell for the automated highway problem is different than that the one used in the pool game simulation. In the automated highway problem each car keeps monitoring the signals via cellular phone from all the cells (spaces) to see which is the strongest. Whenever it discovers that a different cell has become stronger (as a result of the car driving into this new cell), the car informs the old cell which then hands it over to this new cell. This avoids the space having to go through a list of its siblings (neighboring spaces) to find out the new space that contains the object that collided with its boundary.

For collision detection between cars and cars, cars and lanes, cars and curbs, the car can be modeled by five circles as shown in Fig. 11, where each circle represents the zone of a radar attached to the car as follows:

1. One in the center to the car to check for curbs, as a car generally travels closer to curbs than to other automobiles.
2. One at the front and one at the rear of the car to check for cars that may collide from the front/rear.
3. One on both sides of the cars to check for cars coming from the side.

These circles may intersect each other in different ways, representing varying degrees of detection as depicted in Fig. 11(a) and 11(b). So, a car has five methods : CheckFront, CheckCenter, CheckLeft, CheckRight, CheckRear in its collision detection algorithm. These methods are invoked to calculate the collision times of the car with various objects. Each of

[6]Taken from cellular communication, same concept as the space for OOSPG and RACDS.

these methods in turn invokes basic collision detection methods described in the pool game simulation. For example, CollDCircleLine, CollDCirclePoint, etc.

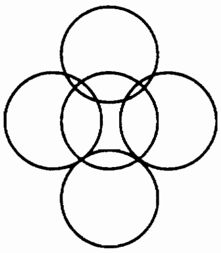


Fig. 11(a)

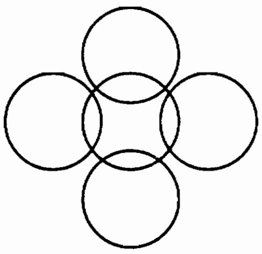


Fig. 11(b)

Proposed collision detection model for AHP

Fig. 11

An alternative to the above proposed five circle design is to model the cars by to two concentric circles. The inner circle for checking collisions with curb, lanes, etc. and the outer (larger) circle for checking collisions with other cars as shown in Fig. 12.

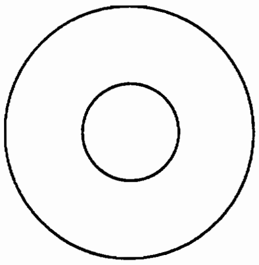

Alternative collision detection model for AHP

Fig. 12

Having two concentric circles makes the calculations relatively simple and the problem reduces to OOSPG. Instead of one ball, AHP has two concentric balls. When it comes to collision detection between ball and ball the outer circle is considered, and between ball and wall (curbs and lanes) , hole (pot hole!), then the inner ball (circle) is considered. Thus, instead of five methods only two methods CheckOuter and CheckInner are required.

4.3 The Simulation

At the beginning of the simulation all objects are at rest and the absolute time is zero. Once a car starts moving in the space, its clock starts ticking. The car's collision detection algorithms start calculating the collision times with other objects in its space. The collision times are checked at every clock tick, against the "safe times". AHP would have different safe times for collisions concerning cars and cars, and for collisions concerning cars and curbs/lanes. If any collision time is equal to or below the safe time then a warning is issued and appropriate action taken. If there is no collision warning all objects are moved forward in time by delta 't'. The reason for moving objects forward in time by delta 't' is because, cars do not have synchronized clocks (a car could start moving at any time).

5.0 Evaluation

5.1 When and How to subdivide the Space

In Section 2 it was shown that the OOSPG algorithm had complexity O(n*df) where, 'n' is the total number of objects and 'df' the decomposition factor. Recall that the decomposition factor causes the space to subdivide if it has 'df' number of objects, or reunites if there are less than 'df' number of objects in adjoining spaces. However, simply reducing 'df' will not necessarily yield the best overall performance, as will be seen. If there are very few objects per space, then the overhead of the space becomes significant as objects constantly move between spaces. That is, more collision checks are required between objects and spaces boundaries and consequently each space having to modify its object list.

For OOSPG, Fig. 13 compares the number of collision checks made during a simulation against different space decomposition factors (df) for varying number of mobile objects, the total number of objects being constant at fifteen. As seen in Fig. 13 for a decomposition factor of five (df = 5) among the chosen decomposition factors we get the best performance. Fig. 13 shows the advantages of a dynamic space. The significance of 'NS' curve is to show the overhead of space. 'NS' curve shows the number of collision checks if there where No Space. The '15' lines shows the number of collision checks if it was one big space. The delta between the above two gives the overhead of one space (objects colliding with space boundary). The '2' curve represents the least decomposition factor but it does not have the least number of collision checks, due to the collisions involving objects and space boundaries. The df = '5' curve has the least number of collision checks and the difference in the number of collision checks is even more significant as the number of objects increases. This signifies an optimal trade off between the advantages of the space and its overhead. The optimal value of 'df' the decomposition factor is application specific and depends on the dynamic nature of the application.

To illustrate the above statement let us take into consideration RACDS (Robot Arm Collision Detection System). In RACDS the robot arms have restricted degrees of freedom meaning the robot arm is less dynamic than pool balls, resulting in fewer collision checks between the objects and the space boundaries. Thus, for RACDS a lower decomposition factor 'df' would give better results. On the other hand AHP (Automated highway problem) is more like the pool game

# of Objects	NS	15	10	5	2
2	420	480	582	660	1396
3	1760	1850	2185	1679	1576
5	3290	3408	2980	2600	2400
10	11287	12908	8976	7658	9076
15	16756	17078	12081	10742	11210

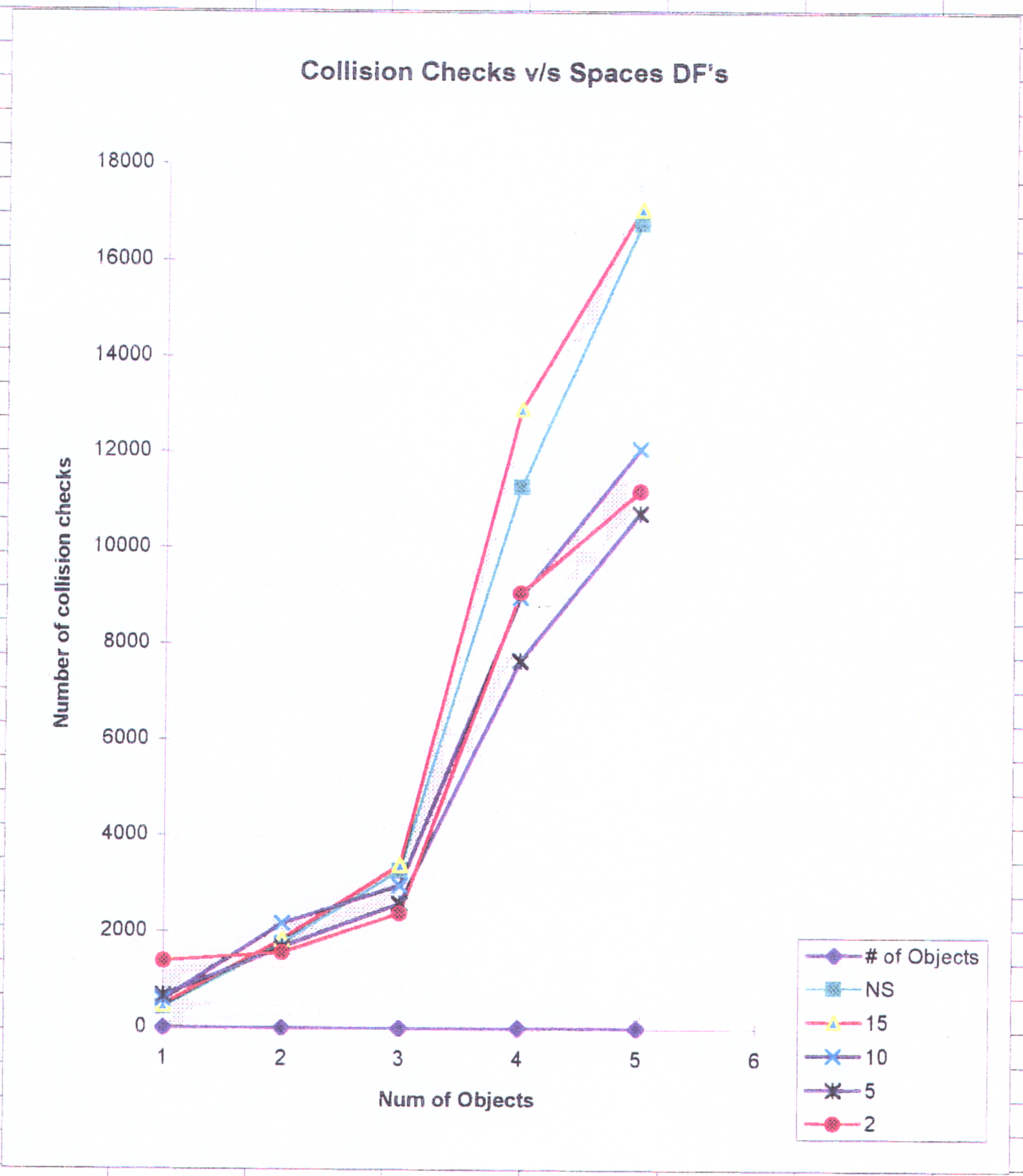


Fig. 13

simulation and would have an higher "optimal" decomposition factor than RACDS. Thus, it is vital to choose the right decomposition factor, and number of divisions to get optimal performance. Again both these factors are dependent on the dynamic nature of the application and can only be determined by empirical measurement. In general the decomposition factor depends on the dynamic behavior of the objects within an application and the role of space in the application.

5.2 How to model other problems in the problem domain

The following approach is recommended to solve other problems in the problem domain using the class hierarchy developed in this project:

1. Identify mobile and stationary objects in the problems. All objects should derive directly or indirectly from either mobile or stationary class.

2. Decide what shape classes various objects would derive from and how to model their physical and dynamic characteristics.

3. Decide how to proceed with the simulation and how to resolve the interactions during the course of the simulation.

4. Decide where the interaction resolver would fit in the class hierarchy.

5. Decide how to model the space and how the space would interact with the objects.

6. Decide the set of rules which would govern the behavior of the space, that is, how and when the space would subdivide and reunite. This involves choosing the right decomposition factor.

5.3 Limitations

Limitation 1. *OOSPG - Circle and Line Collision Detection Method:*

For detecting collisions between a circle and a line the method currently being used only predicts collisions where the line segment becomes tangent to the circle. Cases where the circle will strike the endpoints but will never lie tangent to the line segment are ignored by this method, and it predicts no collision. However, such cases can be taken into account by a method, whose purpose is to predict collisions between circles and points.

Limitation 2. *Absolute time stored as an attribute of mobile objects:*

The declaration of class mobile objects (section 1.2.8) has the time that has elapsed since the beginning of the simulation as an attribute. This “absolute time” could be associated with the global space and a method defined for each object to retrieve this time. This would eliminate the need to store absolute time as an attribute of each mobile object.

Limitation 3. *Jerky motion of the objects on the screen:*

At times if many objects are moving on the screen, and due to other CPU intensive processes running on the system the objects may have a jerky motion.

6.0 Conclusion

A generic class hierarchy has been developed and used to implement the pool game simulation. A design using the class hierarchy, to solve the robot arm collision detection system, and automated highway problem is presented.

6.1 Future Work

6.1.1 Applicability to Other Problems in the Problem Domain

The generic class hierarchy needs to be applied to other problems in the problem domain. This would prove the generic nature of the class hierarchy and bring forth any shortcomings of the current class hierarchy.

6.1.2 Enhancements to the Class Hierarchy

Based on the observations from 6.1.1 the existing class hierarchy needs to be enhanced to model more problem in the problem domain by adding more classes and making the existing classes more generic.

6.1.3 Better User Interface for OOSPG

- In OOSPG an option can be provided, where the user can move balls around in space before defining the cue rather than the system placing the balls in pre-defined locations.
- Using GUI (a menu option) the user should be able to add more objects to the simulation and change the properties of the existing objects.

7.0 REFERENCES

[Bart86]	Paul S. Barth. An Object Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2), April 1986 pp. 142-172
[Bing93]	Tim Bingham, Nancy Hobbs and Dave Husson. Experiences Developing and Using an Object-Oriented Library for Program Manipulation. *Software Development Technologies, Digital Equipment Corporation.* OOPSLA 1993, pp. 83-89
[Chie89]	C.H. Chien and T. Kanade. Distributed Quadtree Processing. *School of Computer Science*, Carnegie-Mellon University, Pittsburgh, PA
[Fuji86]	Kikuo Fujimura and Hanan Samet. A hierarchical strategy for path planning among moving obstacles. *Center for Automation Research.* University of Maryland, College Park, MD
[Hont89]	Philip Hontalas, Brian Beckman, Michael DiLoreto, Leo Blume, Peter Reiher, Kathy Sturdevant, L.Van Warren, John Wedel, Fred Wieland and David Jefferson (UCLA). Performance of the Colliding Pucks simulation on the time wrap operating systems. "Jet Propulsion Laboratory", Pasadena, CA. *The Society for Computer Simulation 1989.*
[Kors90]	Tim Korson and John McGregor. Object-oriented design: A tutorial. *Communications of ACM*, 33(9), 1990
[Nico90]	David M. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Department of Computer Science,* College of William and Mary, May 7, 1990.
[Shaf89]	Clifford A. Shaffer and H. Samet. Optimal Quadtree construction algorithms. *Computer, Vision, Graphics and Image Processing*, 37(3):402-419, March 1987
[Shaf90]	Clifford A. Shaffer and Gregory M. Herb. A Real Time Robot Arm Collision Detection System. *Department of Computer Science*, VPI&SU, Blacksburg, VA June, 11 1990
[Stro92]	Bjarne Stroustrup. *The C++ Programming Language.* Second Edition. Addison Wesley Publishing Company. April 1992
[Tarl90]	Mark A. Tarlton and P. Nong Tarlton. A Framework for Dynamic Visual Applications. "Microelectronics and Computer Technology Corporation".

1992 *Symposium on Interactive 3D Graphics*, Cambridge, MA March 29 - April 1, 1992

[Wass91a] Anthony I. Wassermann. Object-oriented software development: issues in reuse. *Journal of Object Oriented Programming*, 4(2), 55-57, 1991.

[Wass91b] Anthony I. Wassermann. From Object-oriented Analysis to design. *Journal of Object Oriented Programming*, 4(2), 55-57, 1991.