7
Evaluation of Designs

7.1 Introduction

The evaluation software forms the fourth and final element of the generic evolutionary design system. This chapter describes how this software analyses designs, and gives details of the evaluation software that has been created as part of this work in order to allow the specification of a number of different design problems. The user-interface, which allows the user of the system to specify which parts of the evaluation software should be used (as well as all other system parameters) is described at the end of the chapter.

Evaluation software embodies the fitness functions used by the genetic algorithm described in the previous chapter. It is the job of the evaluation software to analyse how well the evolved designs, or phenotypes, fulfil the function of the desired design. The results of this analysis must be presented to the GA in the form of multiple fitness values for each phenotype, which the GA then uses to calculate the overall fitness of each individual. Hence, the evaluation software is responsible for guiding the evolution of designs, directing it to the areas of the search space that contain designs that perform the desired function.

As was stated earlier, the design system does not permit human intervention during evolution. In other words, only the evaluation software can judge the fitness of designs. This minimises the danger of the 'conventional wisdom' of a human designer influencing the outcome of evolution, and maximises the creative potential of the system. However, because the evaluation software is solely responsible for specifying the desired function of the required design, it is essential that it is adequate and correct. The GA in the design system will blindly follow the judgement of the evaluation software - if this software 'asks' for the designs to perform an incorrect function, then the GA will evolve an unsatisfactory design.

The evolutionary design system is intended to be generic, i.e. have the ability to evolve good solutions for a wide range of solid object design problems. Although the evaluation software must inevitably be specific to each particular design problem, the creation of entirely new suites of evaluation software for every new design problem would be laborious, and would require users of the system to have extensive expertise in programming. Consequently, the evaluation software must be as general as possible, consisting of as many reusable elements as possible. This would save the time required to specify the function of new designs, and would allow less computer-literate people to use the system with ease.

7.2 Software Specification of Design Function

7.2.1 Modularity

Traditional design analysis packages are highly specialised pieces of software that are only capable of assessing the performance of a single type of design (Libes 1990). Although attempts have been made to allow generic design optimisation systems to use such analysis tools, the amount of work needed to create an interface to every new analysis package is substantial (Tong, 1992). Moreover, such optimisation systems are limited to the available analysis tools.

For the generic evolutionary design system, a new and different approach to the analysis of designs was taken. Based on the hypothesis that it is possible to break down the function of any design into a number of desired characteristics, a number of separate modules of evaluation software are used, each module analysing a design for a single characteristic. These reusable modules can then be used in different combinations to fully judge the quality of designs for a wide range of design problems.

By building a library of such modules, new design problems can be specified for the evolutionary design system simply by the user selecting which of the existing modules are required. Although some very different design tasks may require the creation of one or two new modules, this will still be substantially less work than creating an entirely new suite of analysis software. Moreover, with every module of evaluation software being reusable, the more design tasks that are set for the design system, the larger the library of modules will become, and the quicker new design tasks can be set.

However, before modules of evaluation software can be selected (or created), the user of the system must determine exactly *which* modules of evaluation software are needed to specify a design problem adequately. In other words, the user must discover which desired characteristics comprise the desired function of the required design.

Fortunately, the problem of breaking down the specification of the function of a design can be achieved relatively easily, by using the evolutionary design system to build this specification from the bottom up. Because the GA in the system will always attempt to evolve designs that perform exactly as defined by the currently selected modules of evaluation software, if any module is incorrect or absent, the evolved designs will consistently show deficiencies in those areas. The entire specification of the function of a design can thus be quickly built up, by adding more and more modules of evaluation software (or removing some) and checking to see what the corresponding evolved designs look like (Bentley & Wakefield, 1995a/1996a). Indeed,

after a little practice, it is possible to surmise correctly which modules are needed for most design tasks almost immediately.

7.2.2 Example: Evaluation of a Table

To demonstrate how the desired function of designs can be built up using a number of modules of evaluation software, the specification of the very first design problem presented to the evolutionary design system will be described. The problem was to evolve a simple table (Bentley & Wakefield, 1995a/1996a). Note that for this initial problem, all designs were represented by five primitive shapes without intersecting planes.

Perhaps the most fundamental characteristic of the design of any table is its size: too big or too small and the design is useless. So, the first module of evaluation software to be used was the 'correct size' module. This allowed the user to specify acceptable outer and inner extents for the required design, penalising any designs that conflicted with these constraints (full implementation details and justification of all modules are given in the next section). Running the evolutionary design system with this single criteria, generated designs that were the correct size, see fig 7.1.

When these designs were examined, it became clear that they were usually very massive. A characteristic of most tables is a low mass, to reduce material costs and allow them to be moved more easily. Hence, the second module of evaluation software to be utilised was the 'correct mass' criteria. This gradually penalised any designs that have a mass greater or less than a user-specified value. Designs produced by the system with these two criteria were always the desired size and the desired low mass, see fig 7.2.

However, some designs were becoming fragmented (primitives becoming detached from the design, and 'floating' in mid-air). This was happening because the easiest way for the GA to create designs of low mass was to reduce the dimensions of the primitive shapes that make up the designs. When dimensions are reduced, the sides of primitives can pull away from the sides

of other primitives, causing the fragmentation of a design. To prevent this, a third module of evaluation software was introduced, which heavily penalised the fitness of fragmented designs.

Another deficiency in the designs was soon apparent - they would not always be able to stand upright under the force of gravity. Since an important characteristic of any table that is it should not fall over, a fourth module of evaluation software was added. This calculated whether designs were stable under gravity, and increased the fitness of designs in proportion to their ability to remain upright. Running the design system with these four modules of evaluation software activated, designs that were the correct size, mass, unfragmented, and stable were consistently produced. Often the stability was achieved by intricate counter-balance weights, see figs 7.3 and 7.4.

An obvious missing characteristic in the designs at this stage was the lack of a table top - an essential feature of tables. To remedy this, a fifth module of evaluation software was added. This allowed the user to specify that designs should have a flat upper surface at a particular height and covering a particular area. With these five criteria, the system was able to produce designs that resembled tables for the first time. Fig. 7.5 shows a table with a good table top and slightly unusual counter-balance at the front, fig. 7.6 shows a table with a slightly imperfect multi-level table top.

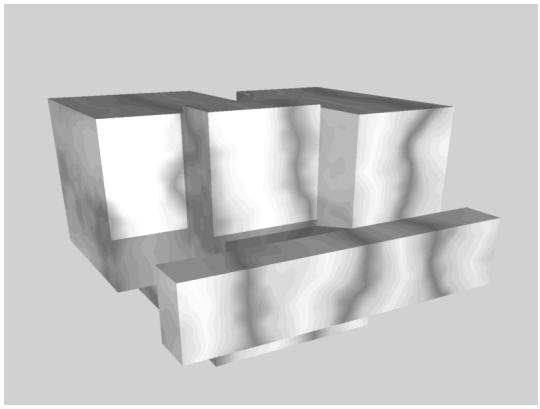

Fig. 7.1 Evaluation of size only.

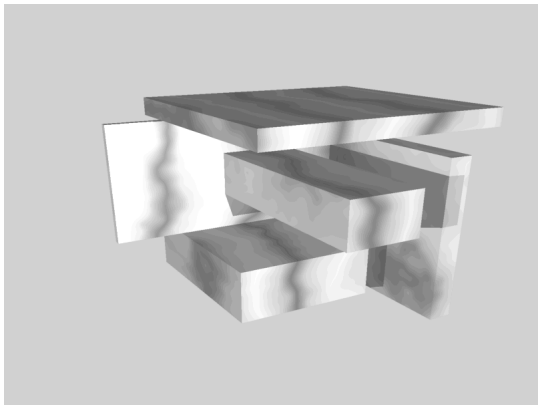

Fig. 7.2 Evaluation of size and low mass.

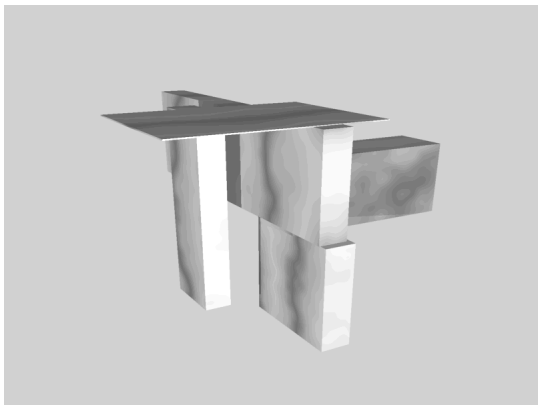

Fig. 7.3 Evaluation of size, low mass, no fragmentation, and stability.

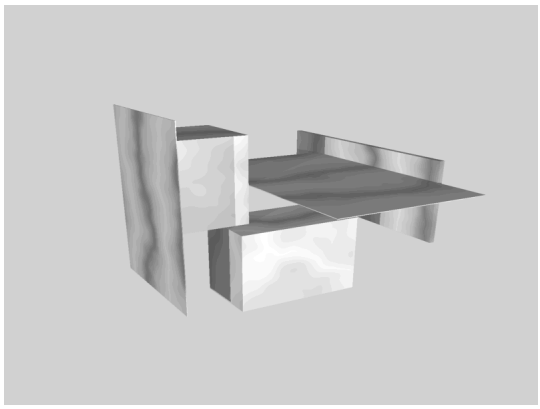

Fig. 7.4 Evaluation of size, low mass, no fragmentation and stability.

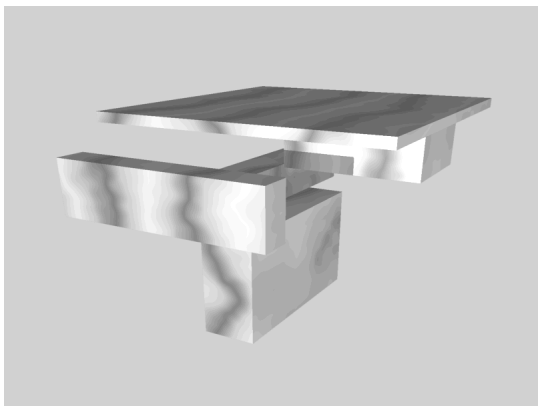

Fig. 7.5 Evaluation of size, low mass, no fragmentation, stability and flat top.

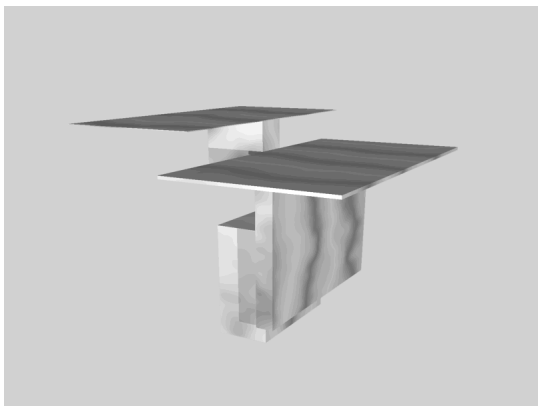

Fig. 7.6 Evaluation of size, low mass, no fragmentation, stability and flat top.

Fig. 7.7 Evaluation of size, low mass, no fragmentation, flat top and supportiveness.

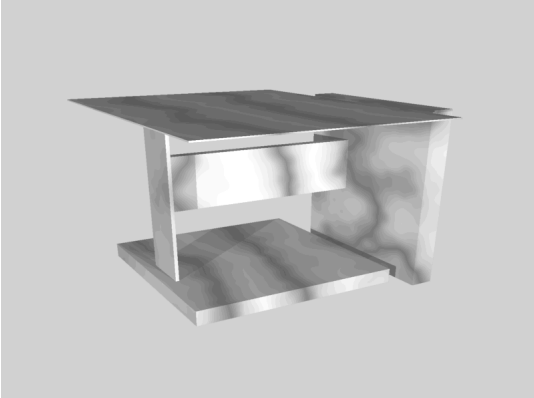

Fig. 7.8 Evaluation of size, low mass, no fragmentation, flat top and supportiveness.

Finally, a less noticeable deficiency was identified in some table designs. Although these designs were all capable of standing upright under their own mass, if an object was placed on them, many would still topple over. In other words, a table must be able to support objects on the edges of its table top, and remain standing. To complete the specification of the required function of the table designs, the 'stability' module was replaced by a 'supportiveness' module of evaluation software. This new criteria set the corresponding fitness value of a design depending upon how stable the design was with a heavy object placed on each edge of the upper surface in turn. With these five modules of evaluation software, the system was able to consistently evolve usable designs for tables, see figs 7.7 and 7.8 (Bentley & Wakefield, 1995a/1996a).

It seems likely that this modular approach to the specification of design problems could be expanded to allow the system to evolve realistic designs suitable for manufacture (e.g. by adding evaluation modules for criteria such as stress-analysis, cost, and availability of materials for the table design problem). It should be emphasised, however, that it was *not* the aim of this work to create perfect evaluation software suitable for real-world applications. The purpose of all of the evaluation software created was to allow the investigation and demonstration of the capabilities of the system to evolve a range of designs with different shapes.

Consequently, using this idea of building the specification of the desired function of designs from the bottom up, the generic evolutionary design system was applied to a number of different design problems.

7.3 Modules of Evaluation Software

7.3.1 Information Modules

Although the list of multiple groups of nine parameters contained within the phenotype does fully define the shape of a design, most modules of evaluation software require additional information about the phenotype they are analysing. For example, data describing the co-ordinates of the corners of the design is required by the 'size' evaluation module in order to calculate the fitness values for this criterion.

Since such additional information is often required by many different evaluation modules for a design problem, it seems logical to allow the information to be shared. In other words, only generate the additional information on phenotypes once, to supply all evaluation modules that require it. In this way, the speed of the system can be increased by minimising the amount of processing required for each evaluation module. Consequently, in addition to the creation of a library of reusable modules of evaluation software, a second library of *phenotype information modules* was created as part of this work. These modules are not directly chosen by the user, they are activated by the system. Hence, for every evaluation module selected by a user to partially specify a design application, there will be corresponding information modules that are automatically used by the generic evolutionary design system.

A total of six different information modules were created, each one generating a range of data about the phenotype being currently evaluated:

Information Module 1: **VERTICES**

Input: *phenotype*

Output: *- a list of the x,y,z coordinates of the vertices for each primitive in phenotype*

- a count of how many vertices for each primitive.

This module generates the co-ordinates of the vertices (i.e. corners) of every primitive in the phenotype, using the equations given in Section 4.2.5 and the Primitive Extraction Algorithm given in Appendix A.

Information Module 2: **PLANES**

Input: *phenotype & vertices of phenotype*

Output: *- a list of planes (A, B, C, D coefficients) for each primitive in phenotype*

- a count of how many planes constitute each primitive

- a Boolean list of whether each 'stretched cube' was intersected by the moveable plane or not.

The four coefficients of the equation of a plane are calculated for the planes forming the sides of every primitive in the phenotype. Each plane is identified by locating three corresponding coplanar vertices from the list of vertices produced by the first information module. Consequently, this module requires the prior execution of the 'vertices' module. Once all of the planes for the primitives have been calculated, they are translated by the centre (x, y, z) position of each primitive (see equation given in Section 4.3.4) in order to make the plane equations relative to the global origin of the phenotype, rather than the centre of each primitive. In addition, a Boolean list of primitives that have been intersected by their moveable plane is created (a primitive is not intersected if its moveable plane has no coplanar vertices).

Information Module 3: **PRIMITIVE EXTENTS**

Input: *vertices of phenotype*

Output: *- a list of the six extents of each primitive in the phenotype*

(left, right, top, bottom, back, front).

The six outer extents of each primitive in the current phenotype are calculated by examining the corners (vertices) of the phenotype. For example, the topmost extent of a primitive is denoted by the *z*-position of the highest vertex of the primitive. Once again, this module requires the prior execution of the 'vertices' module.

Information Module 4: EXTENTS

Input: *primitive extents of phenotype*

Output: *- a list of the six extents of the phenotype*

(left, right, top, bottom, back, front).

This module calculates the overall outer extents of the phenotype, using the extents of the individual primitives in the phenotype calculated by the previously described module. For example, the leftmost left extent of any primitive becomes the left extent of the phenotype. However, any primitive that has been defined as being 'inflexible' is ignored by this module. In other words, the extents of a phenotype are determined by the primitives that can be freely evolved by the system. Inflexible primitives are typically used as fixed 'skeletons', and hence are not considered part of the design that is being evolved. This module requires the prior execution of the 'primitive extents' module.

Information Module 5: MASS

Input: *phenotype*

Output: *- a list of the mass for each primitive in the phenotype*

- the value of the total mass of the entire phenotype

- the x,y,z coordinates of the centre of mass of the phenotype.

This module calculates the volume of each primitive in turn, multiplying this value by a pre-defined density constant to produce the value of the mass of each primitive. In addition to generating the separate mass of each primitive, the total mass of the phenotype is calculated, and the co-ordinates of the centre of mass of the phenotype is calculated (by taking the mean co-ordinate values of each primitive multiplied by the corresponding mass of the primitive). This module can only be activated when primitives are defined as having no intersecting planes.

Information Module 6: **SURFACE AREA**

Input: *phenotype*

Output: *- a list of the surface area for each primitive in the phenotype*

- the value of the total surface area of the entire phenotype

The final information module generates a list of the surface areas of every primitive in the phenotype (by calculating the area of each side and summing them). The total surface area is then calculated by summing the surface areas of each primitive and subtracting the total area lost by primitives touching each other. This module can only be activated when primitives are defined as having no intersecting planes.

7.3.2 Evaluation of Tables

As described earlier, the first design problem presented to the system was to evolve a simple table (Bentley & Wakefield 1995a/1996a, 1995b). This task was chosen because the table is an easily recognisable, every-day object, allowing the early performance of the prototype design system and accuracy of the evaluation software to be readily assessed. Five separate modules of evaluation software were used to guide the evolution. Figure 7.9 shows a block diagram of the evaluation modules selected to specify this design problem, and the information modules automatically used by the system.

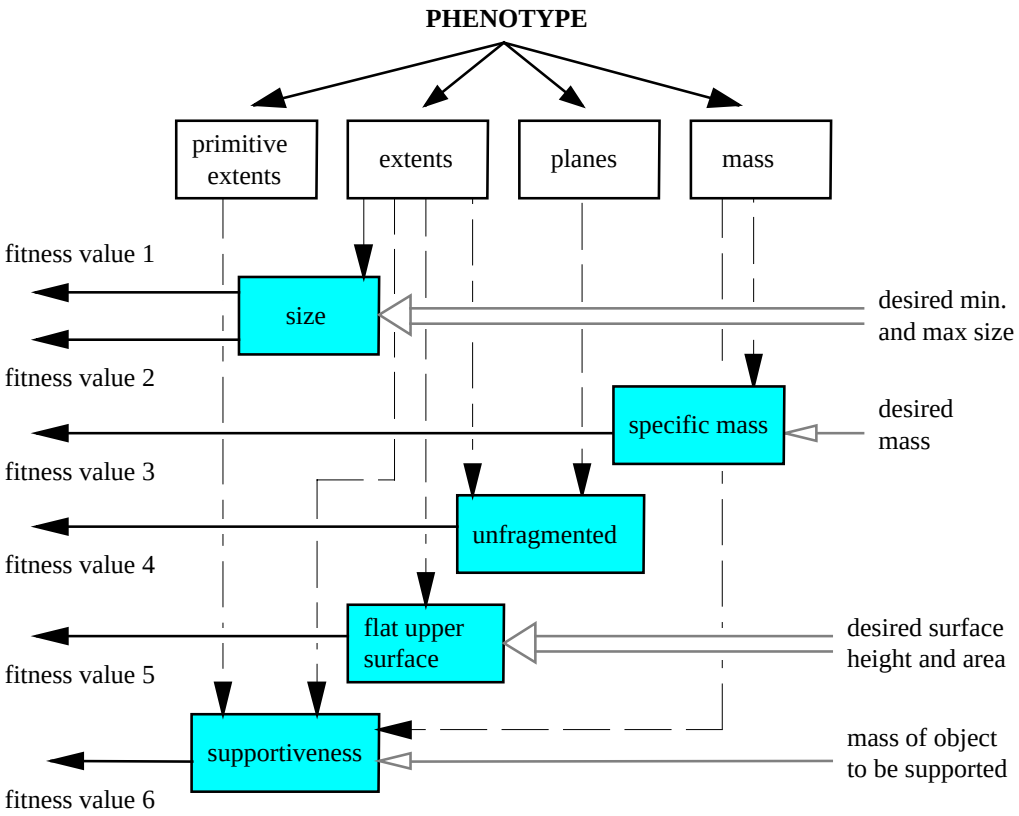

Fig. 7.9 Block diagram of the evaluation software used to evaluate tables.

EVALUATION MODULE 1: SIZE

Information module: *extents*

User-specified parameters: *12 (max. left, min. left, max. right, min. right,..., min. front)*

Number of fitness values: *2 (correctness of min. size, correctness of max. size)*

This was the first evaluation module selected for the table design problem. The size of the design is specified by user-defined minimum and maximum extents for the left, right, back, front, top and bottom of the design, see fig. 7.10. Two fitness values are returned by this module.

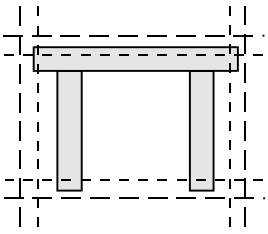

Figure 7.10 Desired minimum and maximum extents

(shown by dotted lines) of evolved design.

The module examines the six outer extents of each phenotype (generated by the 'extents' information module). Any extent that exceeds the corresponding desired maximum extent proportionately sets the fitness value for 'correctness of maximum size'. For example, if the front extent of a phenotype was 75 and the desired front extent was 50, then the 'max. size' fitness value would be set to the difference of 25. Thus, the more the size of a phenotype exceeds the limits defined by the 'maximum size' constraint, the worse the fitness value becomes.

Exactly the same process is used for the 'minimum size' constraint. For example, if the front extent of a phenotype was only 10, and the desired front extent was 40, then the 'min. size' fitness value would be increased by the difference of 30. Thus, the more the extents of a phenotype fall below the lower limits defined by the 'minimum size' constraint, the lower the second fitness value becomes.

An alternative approach to the use of an explicit size constraint for designs would be to use scaling, i.e. for each design use three scaling factors based on the *x*, *y*, and *z* extents of the design to fit the desired limits on size. However, it was felt that such a scaling operation would be disruptive to evolution, since it could radically alter the entire shape of designs if, say, a primitive was deleted. Moreover, when the system is used to evolve the relative positions of components of fixed dimensions or to evolve designs around fixed 'skeletons', it is essential that the shapes of some primitives are not distorted by scaling operations. For this reason, a soft 'size' constraint to simply penalise phenotypes of undesirable size was thought to be more appropriate.

EVALUATION MODULE 2: SPECIFIC MASS

Information module: *mass*

User-specified parameter: *1 (desired mass)*

Number of fitness values: *1 (correctness of mass)*

The second module of evaluation software defines an acceptable mass for phenotypes. This module returns a single fitness value proportional to how closely the mass of the phenotype (given by the 'mass' information module) matches a user-specified desired mass value. For example, if the mass of a phenotype was 190 and the desired mass was 25, the fitness value returned would simply be the difference of 165.

EVALUATION MODULE 3: NO FRAGMENTATION

Information modules: *primitive extents, planes*

User-specified parameters: *none*

Number of fitness values: *1 (degree of fragmentation)*

This evaluation module calculates whether a design is fragmented, and if it is, a fitness value equal to the sum of the distances from the origin of each primitive in the phenotype is returned. In other words, if a phenotype is found to have one or more primitives that are detached from the rest of the design, a fitness value proportionate to how far each of the primitives is from the origin, is returned. This means that, in the unlikely event of most of the population being initially fragmented, evolution will first attempt to move all primitives in designs closer together (at the origin), and thus generate unfragmented designs. If the design is not fragmented, a perfect fitness value of zero is returned.

Fragmented designs are detected by creating a network of the primitives and their connections in a design, and traversing it recursively. Any primitive that is not part of the main design will not be visited, meaning that the design is fragmented.

A primitive is connected to a primitive if it touches that primitive. Touching primitives are detected by first temporarily increasing the dimensions of all primitives by a negligible amount. This converts all touching primitives into overlapping primitives. Next, the conditional check described in Chapter Four (see fig. 4.9) is used to discover if the extents of any primitives (generated by the 'primitive extents' information module) in the phenotype overlap. If the primitives do not have any intersecting planes, then this means they must overlap, and hence

are touching. However, if the two primitives have intersecting planes and their extents overlap, the system then proceeds to identify whether the 'stretched cubes' intersect each other after being 'sliced' by their respective planes. This is performed by using the equations of the planes that form the sides of each primitive (generated by the 'planes' information module) to calculate the lines formed by the intersection of planes from each primitive. These lines are clipped to each primitive (i.e. any part of the line that falls outside either primitive is removed). As soon as a single line has not been clipped out of existence, it is known that the two primitives must overlap, and hence are touching, see fig. 7.11.

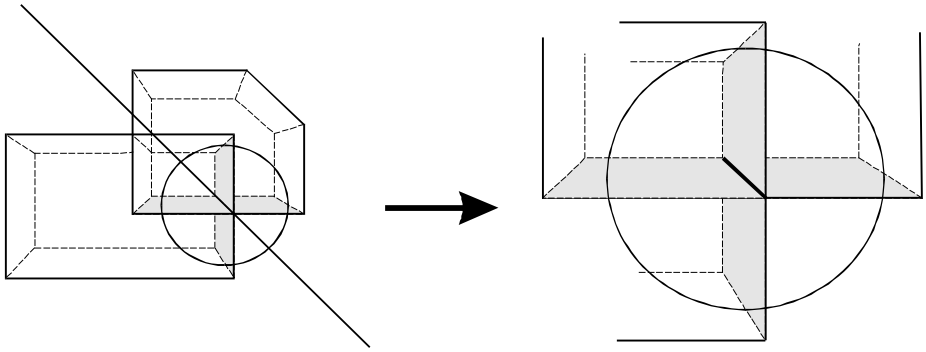

Fig. 7.11 A line of intersection between two planes forming the sides of primitives, that still exists after being clipped to both primitives, means that the two primitives overlap.

Although this algorithm may seem complex, it was found to be the most accurate and reliable method (compared to a number of alternatives tried) for detecting whether two primitives touch (Bentley & Wakefield, 1995a/1996a). It was implemented to ensure that the minimum number of calculations are performed; in use this process requires a negligible amount of computation time, even for large numbers of primitives in a design.

EVALUATION MODULE 4: FLAT UPPER SURFACE

Information modules: *primitive extents*

User-specified parameters: *3 (height of surface from ground, width & depth of surface)*

Number of fitness values: *1 (correctness of upper surface)*

The fourth module of evaluation software used to specify the 'table' design problem, analyses how well the upper surfaces of phenotypes match the desired flat upper surface as specified by three user-defined parameter values. These values define the required height of the table-top, and the desired two-dimensional area of the table-top (with the assumption that it is positioned centrally, above the origin).

To avoid the computational cost of exhaustively sampling the upper surface of a design, the evaluation is performed by picking five random points within the desired area of the table top. The position of the top of the highest primitive at each of these points is then compared with the required height of the table top. If the primitive is higher or lower, then the difference is added to the fitness value. If there is no primitive at a sample point, a value equal to the desired height of the design is added to the fitness score. This process is analogous to five objects of negligible mass, e.g. feathers, being dropped onto the design within the area the table-top is required. Should a feather be supported at too high or too low a height, or not be supported at all, the fitness of the design decreases proportionately, see fig. 7.12. This module does not consider the intersecting planes of primitives (i.e. the user should specify that primitives have no intersecting planes when using this module).

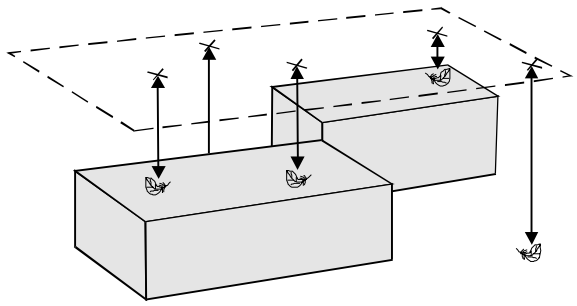

Fig. 7.12 Dropping 'feathers' within the desired area of the table-top and measuring how the height at which they are supported compares to the desired height of the table-top.

EVALUATION MODULE 5: SUPPORTIVENESS

Information modules: *primitive extents, extents, mass*

User-specified parameters: *1 (mass of object to be supported at edges of upper surface)*

Number of fitness values: *1 (degree of stability when supporting object)*

The final evaluation module used for the 'tables' problem calculates the fitness of table designs based on their ability to support a 'virtual object' and remain standing. This module first evaluates the degree of stability of a phenotype with no object on its upper surface. A table will topple over if its centre of mass lies outside its base. Hence, this module finds the lowest primitive(s) in the phenotype (the primitive(s) with bottom extent equal to the phenotype bottom extent), and determines whether the centre of mass (calculated by the 'mass' information module) lies inside or outside the area defined by the primitive's base. If the centre of mass falls inside the base of the phenotype, the fitness value remains at a perfect score of zero. If the centre of mass falls outside, then the fitness value is set to the distance between the base and the centre of mass, see fig 7.13.

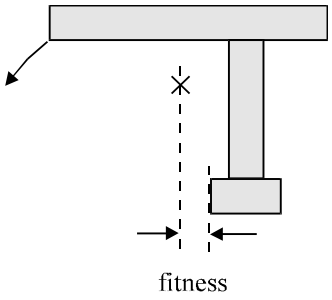

Fig. 7.13 The fitness of an unstable design is proportional to how far the centre of mass is from the base of the design.

Next, the primitive(s) defining the upper surface of the phenotype is/are identified, and a 'virtual primitive' of no dimensions, but having a user-defined mass, is positioned at each of the four edges of this upper surface in turn (i.e., the centre of mass of the phenotype is temporarily modified to produce this effect). The stability of the phenotype is re-evaluated each time, in the same way as before, with the fitness value being incremented further if the design is found to be unstable.

7.3.3 Evaluation of Steps

The second problem presented to the generic evolutionary design system was the task of evolving a set of portable steps, suitable for use in a library. Three steps were required, with each one being capable of supporting the weight of a person without the set of steps toppling over. This problem was defined by four evaluation modules used for the previous design task: 'size', 'specific mass', 'unfragmented', 'supportiveness', and one new evaluation module: 'flat surface' (which allows a desired flat surface at a specific 3D position to be defined). However, unlike the previous 'table' problem, this design problem uses two evaluation modules ('supportiveness' and 'flat surface') three times with different parameters in order to define the three separate flat, supportive surfaces required for the steps.

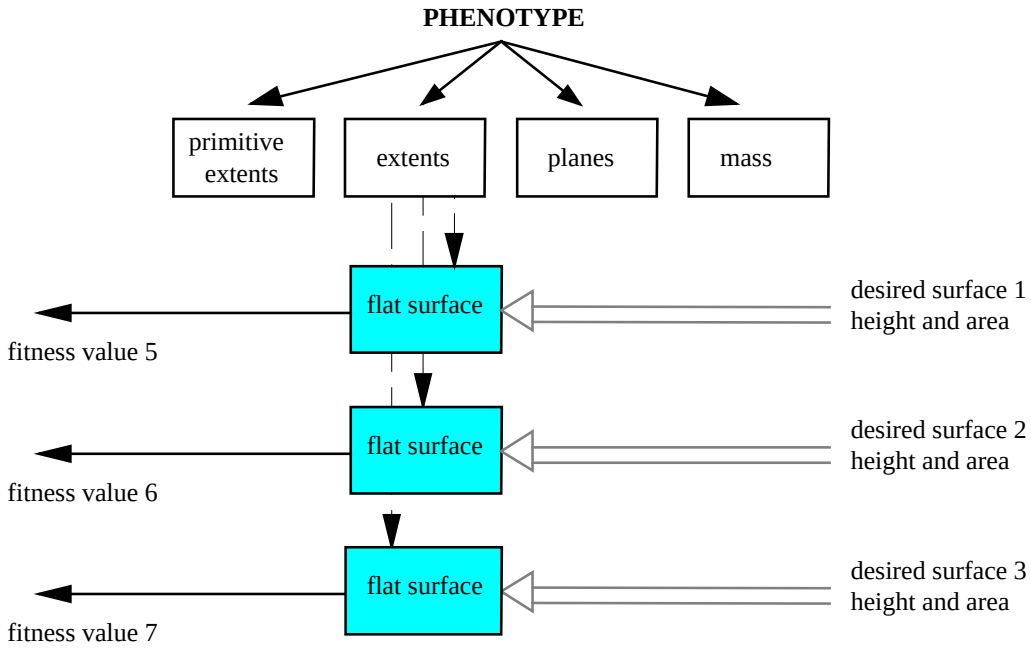

Fig. 7.14 Partial block diagram showing the new evaluation software used to evaluate steps.

This design task was chosen for three reasons: firstly it illustrates the reuse of existing evaluation modules for a new design problem. Secondly, it demonstrates how the same evaluation modules can be used more than once with different user-specified parameters to evaluate phenotypes, see fig. 7.14. Thirdly, a high number of fitness values are generated: two from 'size', one from 'mass', one from 'unfragmented', three from the three 'flat surface' modules

and three from the three 'supportiveness' modules (ten in total). This high number of fitness scores was deliberate, because the 'steps' design problem was used to test the six different multiobjective ranking methods in the design system (described in the last chapter). The more fitness values generated by evaluation software, the more difficult it is for the GA to calculate the overall fitness of a phenotype. Consequently, this design problem was found to be very effective in identifying unsuitable multiobjective techniques for the design system (Bentley & Wakefield, 1996f).

EVALUATION MODULE: FLAT SURFACE

Information modules: *primitive extents*

User-specified parameters: *5 (height of surface from ground, x position, z position, width & depth of surface)*

Number of fitness values: *1 (correctness of upper surface)*

This module of evaluation software was the only new module needed for the 'steps' design problem (replacing the similar 'flat upper surface' module used for the 'tables' problem). The 'flat surface' module allows a desired flat surface to be specified at any 3D position. Because this means that the surface may not always be the upper surface of a phenotype (i.e. the surface could be below another surface), this module cannot use the method of 'dropping feathers' onto the top of phenotypes, used previously.

Instead, five random sample points are picked in the desired area of the flat surface. The primitive with its top surface closest to each point is then identified, and the difference between this upper surface and the desired height of the flat surface is added to the output fitness value. Thus, the less the top surface of the primitive(s) in the vicinity of the desired flat surface matches the desired flat surface, the higher the fitness value (i.e., the lower the fitness) for the phenotype will be. Like the 'flat upper surface' module, this module does not consider the intersecting planes of primitives.

7.3.4 Evaluation of Heat Sinks

The third design task presented to the generic evolutionary design system was to create a heat sink for the processor of a computer. The purpose of a heat sink is to dissipate as much heat from a silicon chip as quickly as possible. Although the simulation of heat dissipation is not simple, because the surface area of heat sinks partly determines their ability to radiate heat, the quality of heat sink designs can be roughly determined by measuring the surface area of designs. Hence, heat sinks can be adequately evaluated by reusing three existing modules of evaluation software: 'size', 'specific mass', 'unfragmented', and adding one new module: 'specific surface area', see fig. 7.15.

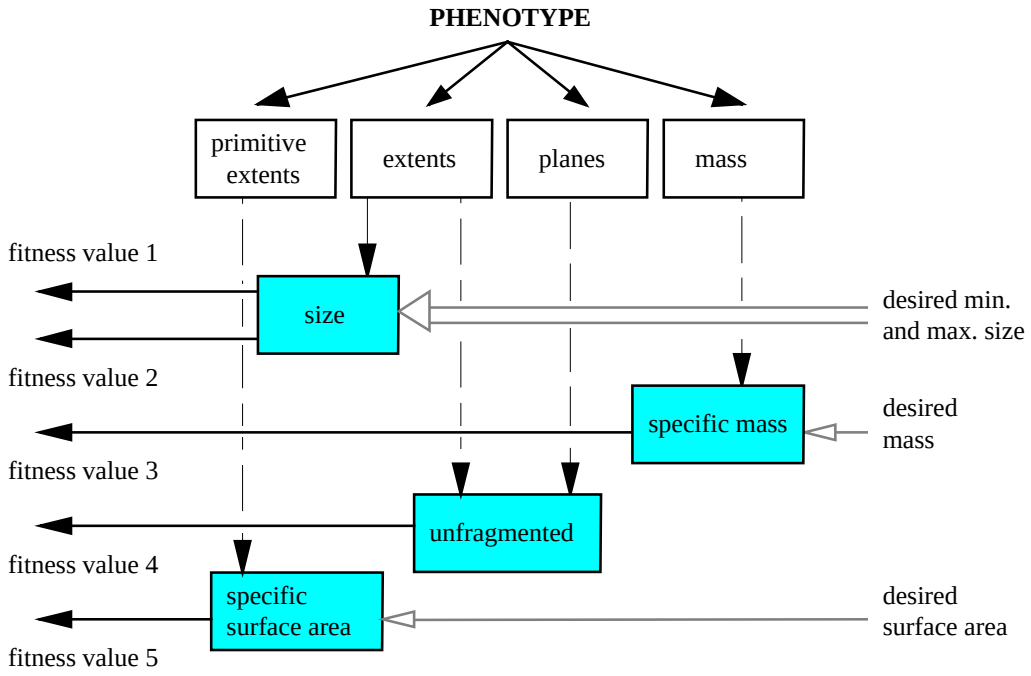

Fig. 7.15 Block diagram of the evaluation software used to evaluate heat sinks.

This design problem was chosen for two reasons. First, it demonstrates the ability of the design system to evolve a design on top of a 'fixed skeleton' (a base the size of the processor that it is cooling). Second, it demonstrates the ability of the system to add new primitives to designs (i.e. locating an appropriate design hyperspace). This is because the easiest way to increase the surface area of a design (when the overall size is limited) is to add more primitives. Indeed, for

this application, optimising the precise shape of primitives is far less important than optimising the number of primitives in designs.

EVALUATION MODULE: SPECIFIC SURFACE AREA

Information modules: *surface area*

User-specified parameters: *1 (desired surface area)*

Number of fitness values: *1 (correctness of surface area)*

This is the only new module of evaluation software needed for the 'heat sink' design problem. The fitness value returned is simply the difference between the desired surface area of designs, and the actual surface area of the current phenotype (as calculated by the 'surface area' information module). By specifying that an almost impossibly large surface area is required, this module can be used, in effect, to maximise the surface area of designs.

7.3.5 Evaluation of Optical Prisms

The fourth design problem presented to the system was to evolve a number of different types of optical glass prism. Prisms are used in many optical devices, for numerous reasons. Binoculars require a prism erecting system to both 'squash' their overall length to a more manageable size, and to keep the images erect (in the same orientation as the object being viewed) (Brown, 1966). Periscopes require derotating prisms to keep the image erect for the observer, no matter to what degree they are rotated (Meyer-Arendt, 1989). An SLR camera requires a constant-deviation prism (usually a penta prism) to ensure that the deviation of the optical axis is unchanged by rotation of the prism (Brown, 1966). Whilst mirrors can also be used for these tasks, prisms have a number of advantages. Firstly the relation between their reflecting faces is not subject to change because of mechanical misalignment or movement. Secondly, dust does not affect reflectivity in the same way as with mirrors, and finally, when total internal reflection occurs within a prism, reflectivity is higher than can be obtained with a mirror (Brown, 1966).

As fig. 7.16 shows, all optical prisms to be evolved by the system have their function specified by four modules of evaluation software: the 'size' and 'unfragmented' modules, and two new

modules 'raytracing' and 'intersected'. The 'raytracing' module is the most significant piece of evaluation software for this class of design problems. This software module traces 'light rays' through phenotypes, calculating the effect each design has on the light. Usually five separate 'rays' are used to specify the four corners of an image and a centre point, so the 'raytracing' module is normally used five times with different parameters. This allows the evaluation of the size and orientation of the output image, as well as the position and direction of the separate light rays.

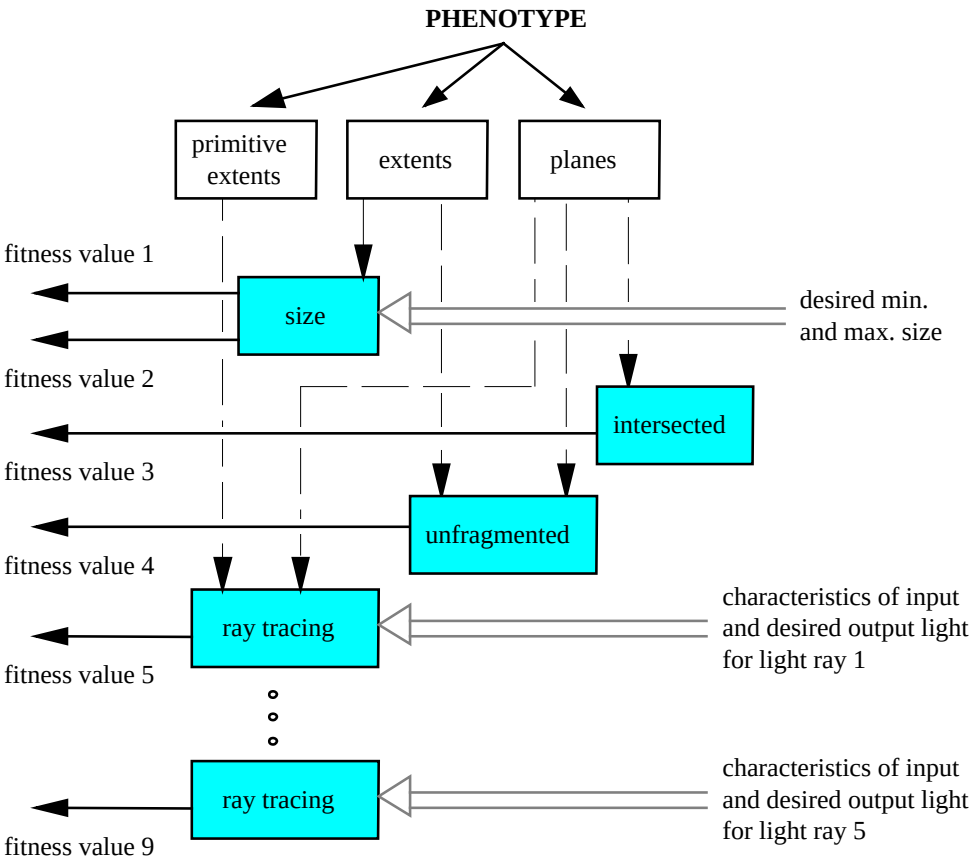

Fig. 7.16 Block diagram of the evaluation software used to evaluate optical prisms.

These prism design problems were chosen for three reasons. First, to demonstrate that the system can successfully optimise the orientation of clipping planes of primitives. (The first three design tasks did not require the use of primitives with intersected planes.) Optical prisms require the precise positioning of these planes in order to direct light correctly through them

(Bentley & Wakefield, 1996e). Hence, for these problems, the parameters that specify the planes of primitives are the most significant parameters to optimise.

The second reason for this choice of problem is that correctly evolving the shape of optical prisms is a very hard, or even deceptive problem, for the design system (Bentley & Wakefield, 1996e). Hence, these design tasks allowed the investigation of how the performance of the system and the accuracy of the functional specification of designs could be improved in order to overcome such difficulties (see Chapter Eight). Finally, these prism design problems allowed the system to be demonstrated evolving prism designs using previously evolved components, in addition to evolving new designs from scratch.

EVALUATION MODULE: INTERSECTED

Information modules: *planes*

User-specified parameters: *none*

Number of fitness values: *1 (whether a design is intersected or not)*

This is the first of the two new modules of evaluation software needed for the 'optical prism' problems. This module simply returns the number of primitives that have not been intersected by their planes (calculated by the 'planes' information module). If all primitives have been intersected, zero is returned (a perfect fitness score). Hence, the fitness of a phenotype for this criterion is proportional to the number of intersected primitives in the phenotype. This evaluation module is used to encourage reflections and refractions of light in phenotypes by ensuring that all primitives have some portion sliced off them.

EVALUATION MODULE: RAYTRACING

Information modules: *planes, primitive extents*

User-specified parameters: *20 (x, y, z coords of light source,*

x, y, z direction of light at source,

x, y, z, coords of required light destination,

x, y, z required direction of light at destination,

A, B, C, D projection plane coefficients,

x, y, z coords of undesired light destination,

Boolean 'refraction allowed' parameter)

Number of fitness values: *1 (correctness of output light)*

Optical prisms are designed to bend and refract light from source to destination along a specific path and in a specific way in order to perform their various functions. However, in order to give the design system complete freedom during the design process, this software module does not directly specify the path the light should take within designs. Instead, the co-ordinates of a light source and an initial direction vector is given by the user. This 'light ray' is then traced through the current design being evaluated and the emerging ray intersected by a user-specified 'projection plane'. The output direction and destination point of the ray is then compared to the required direction and destination point to allow calculation of the fitness of the design, see fig. 7.17.

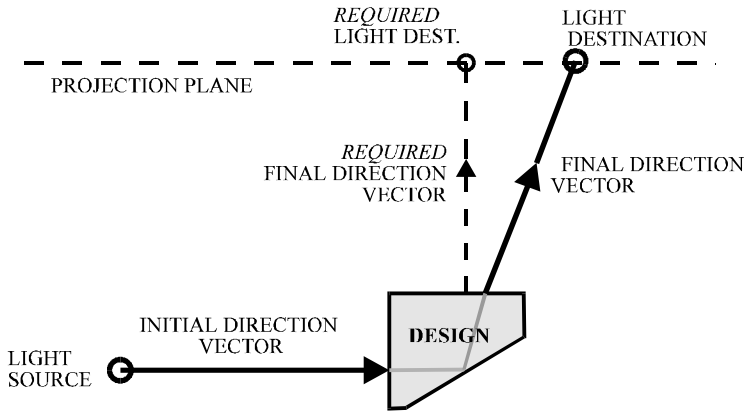

Figure 7.17 Specifying input and output light characteristics.

Light rays are traced through phenotypes by first calculating the line defined by the co-ordinates of the light source and the initial direction vector of the light ray. This line is then checked to see if it intersects any of the sides of the primitives in the phenotype (using the data produced by the 'planes' and 'primitive extents' information modules). The intersection closest to the light source (along the direction of the light) is picked, the light source is changed to the point of intersection, and the new direction vector is calculated. Depending on whether the ray

is travelling into, or out of the design, and on the angle of incidence at the intersection, the ray is either refracted (i.e. bent) or reflected (a total internal reflection in the design), using standard ray-tracing equations (Klein, 1970). This process continues until the closest intersection by the light ray is with the projection plane (or until the ray does not intersect anything), at which point the final direction vector and point of intersection are evaluated.

The single fitness value is formed by summing the *x, y, z* differences between the final direction vector of the emerging ray and the desired direction vector as specified by the user. In addition, this fitness value is incremented by the *x, y, z* differences between the point of intersection of the emerging ray with the projection plane, and the desired point of intersection as specified by the user. Thus, the further the actual direction vector differs from the required direction vector, the less fit the design is. Likewise, the further the actual destination point of the ray is from the desired destination point, the worse the fitness of the design for this criterion. Any designs with output direction vectors so incorrect that they do not intersect the projection plane at all (resulting in no destination points) are penalised heavily, by incrementing the returned fitness value by a large amount.

The 'raytracing' module also allows the definition of two additional characteristics of phenotypes: whether refraction is permitted, and the co-ordinates of an undesirable destination point. Some prisms must not refract light - if the user specifies that refraction should be forbidden, then the fitness value for a phenotype is penalised by a large incrementation for every time the ray of light is refracted by the phenotype. If the user defines an 'undesired destination point' for the light, then the fitness of a phenotype is gradually penalised as the actual destination point of the output light approaches this undesired point (when it is closer than a pre-defined constant distance).

The module assumes that all phenotypes consist of optical glass with the common refractive index of 1.5 (Meyer-Arendt, 1989). Phenotypes are assumed to be surrounded by air (with a refractive index of 1.0003). Only monochromatic light (light of a single wavelength) is

considered, with surface reflections being ignored. These restrictions are purely to simplify and speed up the evaluation process; more realism could be introduced by repeatedly evaluating designs using light of varying wavelengths. Surface reflections are usually insignificant with optical prisms (as opposed to total internal reflections) and can be safely ignored without significant loss of realism for most applications (Klein, 1970).

7.3.6 Evaluation of 'Streamlined' Shapes

The fifth and final design problem to be presented to the generic evolutionary design system was to evolve a number of different 'streamlined' shapes (Bentley & Wakefield, 1996c,d). Examples of such designs include the hulls of boats, the fronts of trains, and aerodynamically shaped cars (Pope & Harper, 1966).

Although it is possible to simulate the flow of air and water over a wide range of designs with some accuracy (Kuethe & Schetzer, 1959), the creation of such analysis software is not trivial, and hence is beyond the scope of this project. However, it is possible to crudely model the flow of liquid and gas past a design by 'firing' separate particles at designs, and calculating their trajectories over the designs, and the forces generated by them (Foley, et. al. 1990).

This is the approach used to judge how phenotypes react to water or air flowing past them. A new evaluation software module, 'particle-flow simulator' is used to allow the evaluation of how well the flow of particles over the shape of a phenotype generates user-specified desired forces on various areas of the phenotype. In addition to this new module, two existing evaluation modules are used: 'size' and 'unfragmented', and also one other new module: 'must have vertices', see fig. 7.18.

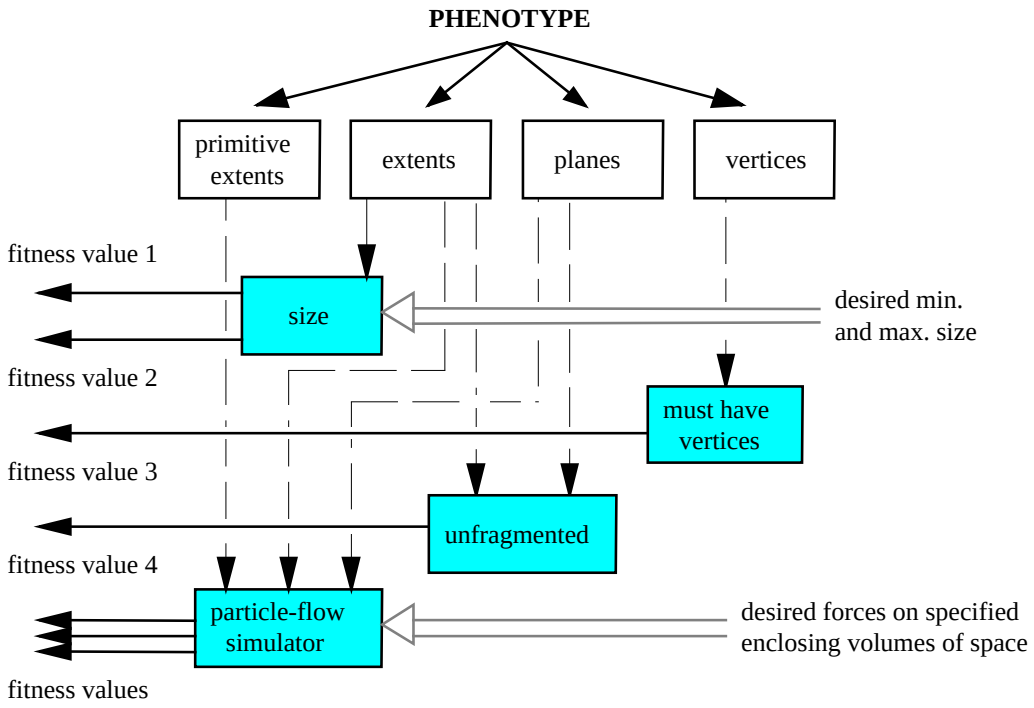

Fig. 7.18 Block diagram of the evaluation software used to evaluate streamlined designs.

The 'streamlined' class of problems was chosen for two reasons. First, they allow the demonstration of the system to evolve good solutions for more realistic applications. Unlike the previous design tasks, the creation of streamlined, or aerodynamic designs is a common and very challenging problem faced by designers in the real world (Obavashi and Takanashi, 1995). Second, this type of design problem allows every part of the system to be demonstrated in combination, e.g. the evolution from scratch or using parts of previously evolved designs, the evolution of the shape and number of primitives, symmetry, fixing genes, and so on.

EVALUATION MODULE: MUST HAVE VERTICES

Information modules: *vertices*

User-specified parameters: *none*

Number of fitness values: *1 (whether a design has vertices or not)*

This is the simplest of the two new modules of evaluation software used to analyse streamlined phenotypes. It simply returns the number of primitives in the current phenotype that do not have any vertices (i.e. if every primitive has at least one vertex, a perfect fitness of zero is returned). The purpose of the module is to discourage the system from 'removing' primitives by

slicing them out of existence with their intersecting planes. Although this effect is relatively uncommon, the use of this module ensures that every primitive in a phenotype contributes to the shape of the phenotype.

EVALUATION MODULE: PARTICLE FLOW SIMULATOR

Information modules: *planes, primitive extents, extents*

User-specified parameters: *variable (a number of required x, y, z forces on specifically defined volumes of space enclosing the phenotype)*

Number of fitness values: *variable (correctness of x, y, z forces on phenotype)*

This module returns a variable number of sets of three fitness values which denote how closely the actual *x, y, z* forces generated by particles passing over the phenotype match the desired *x, y, z* forces, for a corresponding number of user-defined volumes enclosing the phenotype. The user is required to specify the number of volumes, the dimensions of these volumes, and desired values for the *x, y, z* forces to be produced on the design within each volume, see fig. 7.19.

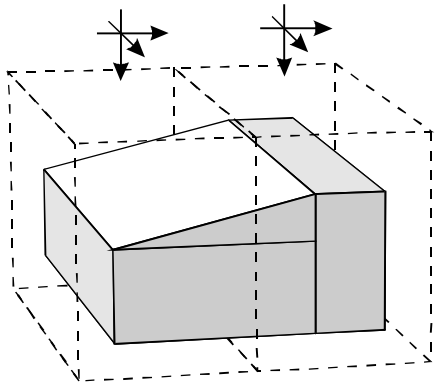

Fig. 7.19 Two user-defined volumes enclosing a phenotype, and the desired forces on the part of the phenotype that is enclosed by each volume.

The use of volumes to specify that different areas of designs should have different forces on them, allows the definition of more representative real-world problems. For example, some racing cars require substantial forces to push the back wheels down onto the track and improve the grip of the tyres, but less force on the front wheels to aid steering (Scibor-Rylski, 1975). Moreover, returning a separate fitness value to denote the accuracy of the *x, y, z* forces instead

of a single, overall force, allows the user to specify that some forces are more important than others.

The module calculates forces produced on a design by firing a pre-defined number of particles, one at a time, at the phenotype (typically between ten and twenty per design, see the next chapter). Each particle begins at an initially random point (within the extents of the sides of the phenotype), at a pre-defined distance in front of the phenotype, with a random velocity (between pre-defined ranges). The position of the particle is then repeatedly updated in proportion to this velocity (to emulate movement over time). Collision detection is performed by continuously checking to see if the line formed by the current position and the next position of the particle intersects any part of the phenotype. Using the same method as for the raytracing module, the closest intersection point of the trajectory of the particle and the side of a primitive is identified, and the particle is reflected (or 'bounced') off the primitive it has hit, by updating its velocity vector. In order to approximate the effect other particles would have on this reflected particle, its velocity is subsequently slowly modified over time until it is travelling in the same direction and towards the same distance point as it was before colliding with the design. Although this means that factors such as turbulence are not considered, it does allow a reasonably realistic curved 'flow' of particles over designs to be produced, see fig. 7.20.

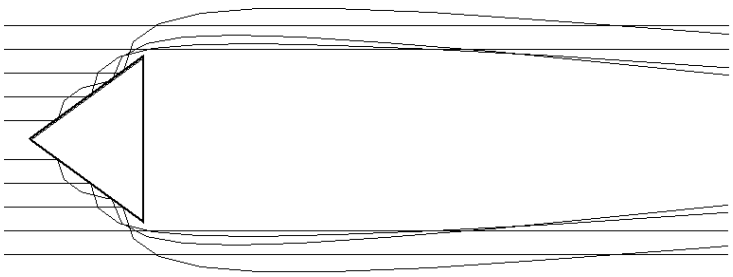

Fig. 7.20 Actual path of particles as calculated by the particle flow simulator.

Whenever a particle hits a phenotype (and this can happen a number of times for a single particle), it generates a force on that phenotype, at the position of the collision. This force is proportional to the velocity at which the particle was travelling, and is in the direction of the

normal of the side of the primitive that was hit. If this position lies within a volume of space defined by the user, the *x, y, z* forces are added to the values of the overall *x, y, z* forces produced so far on the phenotype in that volume. The trajectory of each particle is updated up to twenty-five times (to stop a trapped particle from being 'bounced' around forever), or until it has passed the back extent of the phenotype.

When all particles have been fired, the forces generated in each volume are averaged (because some regions of the phenotype may be hit by more particles than others, giving artificially higher forces in those areas). The difference between every desired *x, y, z* force specified by the user, and the corresponding actual forces is then returned in a number of corresponding fitness values.

7.4 User-Interface

7.4.1 Choosing Evaluation Modules and Specifying Parameter Values

Before the generic evolutionary design system can begin evolution, the user must define which modules of evaluation software are required, and give the necessary parameter values for these modules. In addition, the user has the option of changing default system values such as population sizes, mutation rates, and whether designs should be symmetrical, should have fixed genes, primitives without intersecting planes, inflexible primitives, and so on.

All such user-definable options and parameter values are input into the system by a single initialisation file which is automatically read by the system upon execution. The file, named 'GADESIGN.INI', consists of a selection of commands and values, which allow almost every variable of the system to be controlled by the user. Commands take four forms: *compound commands*, *Boolean commands*, *single parameter commands*, and *multiple parameter commands*.

Compound commands are used to group a number of other (non-compound) commands. For example, the command 'EVALUATE ..... END' informs the system which modules of evaluation software are selected. Compound commands may have a single parameter value, for example, the command 'VALUES 3 ..... END' informs the system that user-specified alleles should be used for some of the genes of the third primitive in every genotype.

Boolean commands are used to activate certain options of the system that are inactive by default. For example, the command 'NO_IPLANES' informs the system that primitives should not have intersecting planes. Likewise, the command 'Y_SYMMETRY' informs the system that phenotypes should be reflected in the plane $y = 0$, in order to produce symmetrical designs.

Single parameter commands are used to define a single value for some variable of the system. For example, the command 'EXT_POP 200' specifies that the external population should hold a maximum of 200 individuals. Alternatively, when used within the 'VALUES' compound command group, the command 'WIDTH 35' specifies that the value of the width gene (for the primitive in every genotype specified by the 'VALUES' command) should be given the initial value of 35.

Finally, multiple parameter commands allow the user to specify a number of different parameter values for a single element of the system. For example, when used within the 'EVALUATE' compound command group, the command 'SIZE 50 -50 30 -30 50 -50 45 -45 25 -25 45 -45' specifies that the 'size' evaluation module should be used, and defines the twelve desired extents.

When reading the initialisation file, the system ignores all whitespace and all lines beginning with a hash: '#'. This allows comments to be added to the file, in order to explain the use and function of every command. A complete list of all system initialisation commands is given in the appendix. Figure 7.21 shows an example of a 'GADESIGN.INI' file.

```
# define the design problem:
EVALUATE
    SIZE
    50 -50 30 -30 50 -50 45 -45 25 -25 45 -45
    MASS 15
    SURFACEAREA 9999999
    UNFRAGMENTED
    MUSTHAVEVERTS
END

# define population sizes
INT_POP 180
EXT_POP 200

# specify that we want 7 primitives in all genotypes *before reflections*:
PRIMITIVES 7

# specify that we don't want any intersecting planes:
NO_IPLANES

# give initial values for 3 genes of primitive 0 in all genotypes:
VALUES 0
    XPOS 0.0000
    WIDTH 100.0000
    ANGLE2 0.0000
END

# specify that all designs must be symmetrical in z=0 and x=0:
Z_SYMMETRY
X_SYMMETRY

# define fixed genes for primitive 0
# (0=not fixed 1=fixed, order: x y z wdth hght dpth ang1 ang2 plndist):
FIXED 0
1 0 0 1 0 0 0 1 0

# specify that genotype primitive 0 is inflexible
INFLEXIBLE 0
```

Fig. 7.21 A sample 'GADESIGN.INI' file.

7.4.2 Using the System

The system was implemented in ANSII standard 'C' on an IBM compatible PC and can be recompiled and executed on any computer with a 'C' compiler with negligible changes to the code required. As stated above, when executed, the system first reads and parses its initialisation file. As the various commands are parsed, the system outputs to the screen the options that were selected by the user. Evolution then begins automatically. In order to increase the speed of evolution and keep the system platform-independent, the system does not graphically display the designs that it is evolving. Instead, it saves a text file containing the phenotype of the fittest design, every generation (in addition to displaying the fitness values of this design on the screen). Evolution continues until a pre-defined number of generations have passed (the default is 500), or until the user presses 'ESCAPE'.

Although this is the fastest way to use the generic evolutionary design system to evolve designs, because the designs are not displayed during evolution, it is difficult to determine how well evolution is proceeding. Consequently, a simple graphical user interface was developed for Microsoft Windows™ 3.x on IBM-compatible PCs to control the system and allow designs to be displayed during evolution, see fig. 7.22.

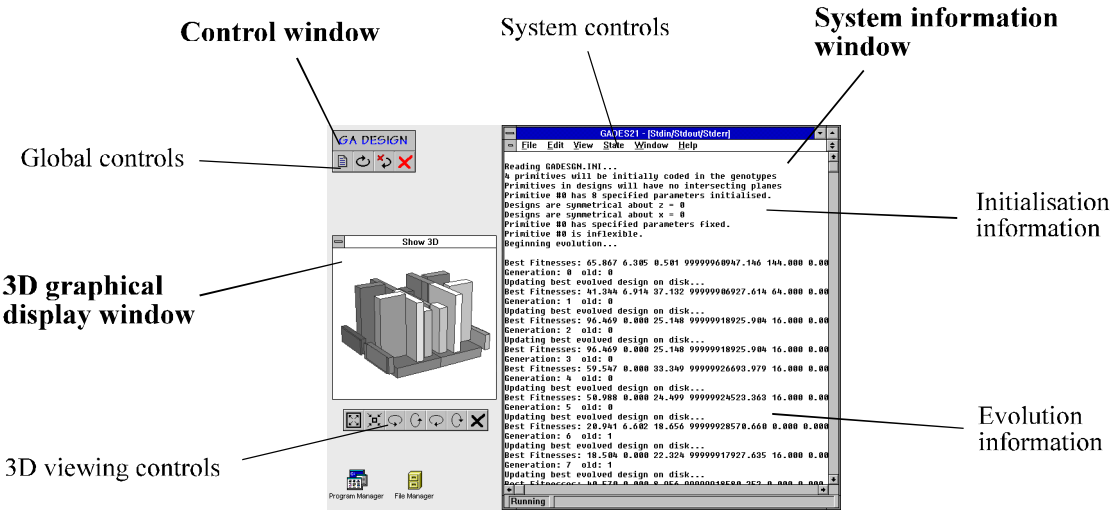

Fig. 7.22 The graphical user-interface of the generic evolutionary design system.

As figure 7.22 shows, the graphical user-interface consists of three main windows: the 'control window', the 'system information window', and the '3D graphical display window'. These three windows correspond to three entirely distinct programs: a control program, the evolutionary design system, and a 3D graphical display program, all running concurrently under MS Windows™ 3.x. To launch all three, the control program is executed. This places the control window at the top left of the screen and then launches and positions the evolutionary design system and the 3D graphical display program.

To permit the design system to run in parallel with the other two programs, it was compiled as a partially Windows™-compatible program, allowing all of its output to appear in a separate window. This 'system information' window also allows evolution to be temporarily paused or resumed, and permits the system to be terminated, by the user clicking on the appropriate,

standard menu items. The window is automatically positioned by the control program to cover two-thirds of the right of the screen, whatever the screen resolution.

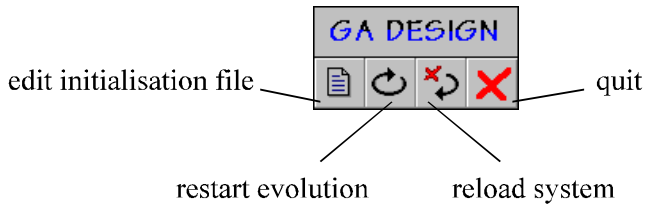

Fig. 7.23 Global controls in the control window.

In addition to launching the design system, the control program also allows various global operations to be performed. As figure 7.23 shows, the control window has four iconised buttons. When clicked on with a mouse, the first of these buttons allows the initialisation file to be edited by launching a standard text editor with the 'GADESIGN.INI' file. The second button informs the design system that evolution is to be restarted (re-reading the initialisation file). The third button terminates the design system and relaunches it (allowing another design to be evolved after evolution is complete) and the fourth button terminates all programs. Communication between the control program and the design system takes place via messages in a temporary file (standard messages handled by Windows™ 3.x are not possible since the design system is not a true Windows™ program).

While the generic evolutionary design system is running, the control program also monitors the output of the design system. Whenever the phenotype of a new, fit, evolved design is saved to disk by the system, the control program automatically informs the 3D graphical display program (this time using a standard Windows™ message) to update its current image by re-reading the saved phenotype.

The graphical display program created as part of this work continuously shows a three-dimensional rotating image of the phenotype it was instructed to display when launched. It first reads the '3D' format phenotype output file produced by the system, and calculates the position

of the vertices of the design using the algorithm given in Section 4.2.5. The vertices for each side of a primitive are then ordered by calculating the bounding envelope around them, and transformed by standard rotation and perspective equations, allowing separate sides to be realistically displayed as polygons (Foley, et. al., 1990). Hidden-surface removal is performed using a simplified variation of the z-buffer method (Foley, et.al., 1990), i.e. polygons are displayed in order of the average z-position of the vertices of the polygon, the furthest first.

The program continuously and smoothly animates the design by using a bit-block-transfer to move the next image from memory to the window. The image can be toggled from a wire-frame, to unshaded solid, to shaded solid by clicking in the window. (The shade of a polygon is simply determined by calculating the average proximity of its vertices to a light source.) In addition, this program allows the image to be rotated in any direction, at any speed, and permits the user to zoom in and out, by clicking on the appropriate iconised button, see fig. 7.24. Although at the beginning of this research a version of this display program was also created to run under X-Windows, because of the proliferation of advanced accelerated graphics cards in most PCs today (Bentley & Thorn, 1994), the Windows™ 3.x version was found to be substantially faster.

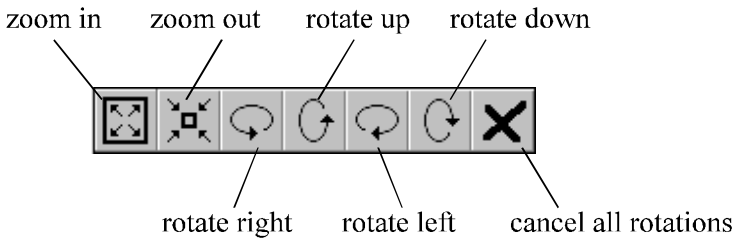

Fig. 7.24 3D viewing controls for the graphical display window.

7.5 Summary

This chapter has described the fourth and final element of the generic evolutionary design system - the evaluation software. Evolution of solid object designs by the system is guided solely by such software, i.e. the evaluation software is used to specify the desired function of designs, by judging the fitnesses of phenotypes.

Based on the hypothesis that the function of designs can be broken down into a number of smaller functional elements, the evaluation software is modularised into a number of discrete, reusable elements, known as 'evaluation modules' and 'information modules'. These modules permit a range of different designs to be analysed, with the user simply needing to pick which evaluation modules should be used in combination. If necessary, the appropriate combination of modules can be built up slowly, by adding new modules and running the design system to identify any deficiencies of evolved designs. As was shown by the 'table' example, this allows the complete specification of the function of the desired design to be assembled quickly and easily.

Evaluation modules were used in different combinations to define the desired function of five different types of design problem: 'tables', 'steps', 'heat sinks', 'optical prisms', and 'streamlined' shapes. Each problem was designed to allow the investigation of a particular aspect of the design system. For example, 'heat sinks' problem was created to check that the system could cope with variable-length genotypes as the number of primitives was optimised, while the 'steps' task was to allow the comparison of different multiobjective techniques.

A library of eleven different modules of evaluation software was described: 'size', 'specific mass', 'unfragmented', 'flat upper surface', 'supportiveness', 'flat surface', 'specific surface area', 'intersected, 'raytracing', 'must have vertices' and 'particle flow simulator'. When selected by the user, each module shared appropriate information from some of the six information modules: 'vertices', 'planes', 'primitive extents', 'extents', 'mass', and 'surface area'.

The modules of evaluation software are selected and their appropriate parameters (in addition to all other system parameters) are input to the system in the form of commands in an initialisation file. This text file is automatically read by the design system upon execution of the system, prior to evolution beginning.

The quickest way to use the system to evolve new designs is to execute it in isolation. However, if designs are to be viewed during evolution, and if additional control over the system is required, the graphical user-interface running under Microsoft Windows™ 3.x should be used. This consists of three windows: the 'information window' of the design system, the 'control window' of a control program, and the '3D graphical display window' of the graphical display program which shows a rotating shaded image of the current best evolved design.