# GADGET: A Toolkit for Optimization-Based Approaches to Interface and Display Generation

**James Fogarty and Scott E. Hudson**
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{ jfogarty, scott.hudson }@cs.cmu.edu

**ABSTRACT**
Recent work is beginning to reveal the potential of numerical optimization as an approach to generating interfaces and displays. Optimization-based approaches can often allow a mix of independent goals and constraints to be blended in ways that would be difficult to describe algorithmically. While optimization-based techniques appear to offer several potential advantages, further research in this area is hampered by the lack of appropriate tools. This paper presents GADGET, an experimental toolkit to support optimization for interface and display generation. GADGET provides convenient abstractions of many optimization concepts. GADGET also provides mechanisms to help programmers quickly create optimizations, including an efficient lazy evaluation framework, a powerful and configurable optimization structure, and a library of reusable components. Together these facilities provide an appropriate tool to enable exploration of a new class of interface and display generation techniques.

**Keywords**
Toolkits, numerical optimization, display generation, layout algorithms, perceptually optimized displays.

**INTRODUCTION**
The exponential improvement of computing speed described by Moore's Law has enabled fundamental changes in user interfaces. Early graphical interface displays were monochrome, dynamic effects were very limited, and carefully tuned assembly code was employed to provide acceptable response times in the user interface. Yet these systems taxed computers of the day to their limits. In contrast, today's computers easily display dynamic color and spend most of their processor cycles idle. User interfaces now make common use of animation and other techniques that were once computationally infeasible. In the future, we can expect to see other new opportunities move from being infeasible to being commonplace.

Recent work, including the Kandinsky system [9] and the LineDrive system [2], is beginning to show the potential for one such technique – the use of numerical optimization in interface and display generation. Numerical optimization is the minimization or maximization of a function subject to constraints on its variables [22]. The Kandinsky system generates aesthetic information collages, which enhance the aesthetics of a space and also convey information. Because aesthetics are difficult to define algorithmically, the Kandinsky system uses aesthetic templates to define a matching problem, which it then solves with optimization. Similarly, the LineDrive system uses optimization to solve the variety of problems that arise in the design and rendering of perceptually simplified route maps. By using distortion, simplification, and abstraction, the LineDrive system generates route maps that resemble hand-drawn maps. These maps aid navigation and prevent the clutter created by information that is irrelevant to a route. The many constraints on how this information can be arranged would be difficult to manage algorithmically, but the problem is nicely described as an optimization.

While computationally expensive, optimization has several advantages as a general approach to interface and display generation. First, optimization seems to fit the way people think about elements of an interface or display. For example, it is common to want two elements of an information display to be near each other, aligned with each other, the same color, or non-overlapping. Similarly, we do not want distortions of a route map to create false intersections between roads that do not actually intersect. Programmers can identify such good and bad features of an interface or display and create an optimization by simply combining these independent criteria. This seems to be more intuitive than trying to construct an algorithm that simultaneously satisfies a variety of conditions. Second, optimization can work well with existing algorithms. As we will illustrate in our demonstrations, programmers can use an algorithm or heuristic to get a reasonable solution and then use optimization to explore similar solutions. Alternatively, programmers can have algorithms produce a variety of solutions, using optimization to improve upon and eventually choose between them. Finally, optimization offers a level of flexibility that is a sharp contrast to the fragile nature of algorithms. Adding a new goal or

constraint to an algorithm will commonly require a reevaluation of every part of the algorithm, and may break it entirely. In contrast, new goals and constraints can normally be added to an optimization and balanced with the existing requirements without starting over.

Optimization can be a difficult approach to pursue in current programming environments. Many programmers may be intimidated by or uncomfortable with the math required for programming an optimization. While optimization toolkits are available [7], they typically still require substantial specialized knowledge because they have mostly been designed for physics simulations and other traditional optimization problems. Further work on optimization as an approach to interface and display generation is hampered by the lack of an approachable toolkit designed specifically for these problems.

This paper presents GADGET, an experimental toolkit designed to support the exploration of optimization as an approach to interface and display generation. In the next section, we offer a simple example of an optimization created with GADGET. This is followed by a discussion of the major architectural features of GADGET, including a standard framework to abstract much of the mathematics behind optimization, generic property support integrated with an efficient lazy evaluation framework, a powerful and configurable optimization structure, and a library of reusable components. Afterward, we show three examples of larger systems: iterative improvement of Bubblemap layouts, generation of route maps like those created by LineDrive, and automated dialog layout. Finally, we discuss some related work and present some short conclusions.

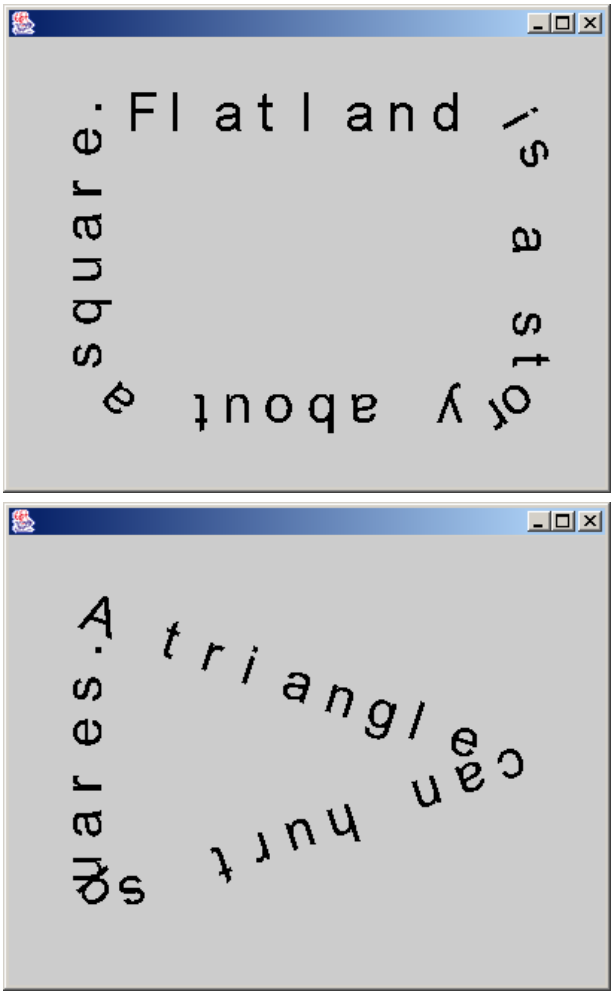


**Figure 1 – Simple Arrangements of Text on a Polygon**

## A SIMPLE EXAMPLE: TEXT ON A POLYGON

In order to explain how programmers create GADGET optimizations, we will now present a thorough explanation of a simple optimization. The problem we present is arranging text in the shape of an arbitrary polygon. Posing this problem as an optimization, each character in the text should be on an edge of the polygon, characters should be displayed in the correct order, and the characters should be spread out across the length of the polygon. The examples included later in this paper demonstrate the application of these same techniques to larger, more interesting problems.

A programmer creating an optimization using the GADGET toolkit needs to supply three components: an *initializer*, *iterations*, and *evaluations*. An initializer creates an initial solution to be optimized. This might be based on an existing or simplified algorithm, or done randomly. Iterations are responsible for transforming one potential solution into another, typically using methods that are at least partially random. Finally, evaluations are used for judging the different notions of goodness in a solution.

We will first define the evaluations used by our optimization. Derived from a base class provided by the toolkit, each evaluation examines some part of the current state of the problem. As we will discuss in a later section, GADGET provides a reusable library of evaluations. For the purpose of this introduction, however, we will assume that the programmer implements the necessary evaluations.

The first evaluation ensures that characters are placed on an edge of the polygon. To do so, the evaluation checks each character to determine how close it is to the nearest edge of the polygon. It creates an array of double values from these distances and returns this array to GADGET. It also indicates to the toolkit that the sum of the squares of the values in this array should be minimized. This component, therefore, is very simple and can be created without knowledge of complex optimization techniques.

Our second and third evaluations are equally simple. Our second evaluation ensures that characters are displayed in the correct order. It builds an array of double values by checking that each pair of characters is displayed in the correct order. If a pair is in the correct order, the evaluator adds a zero to the array of double values. Otherwise, it adds the distance between the pair of characters along the polygon to the array of double values. It returns this array

of doubles to GADGET and indicates that the toolkit should minimize the sum of the squares of the values in the array. Our third evaluation spreads characters across the length of the polygon. It creates an array of doubles by adding an entry to the array for the distance along the polygon between each pair of characters. Unlike our first two evaluators, this array is not minimized. The evaluation returns the array to GADGET with an indication that the toolkit should maximize the value of the smallest entry in the array. This will push the characters apart.

With these evaluations complete, the programmer next provides a set of *weights* to indicate the relative importance of each evaluation. By giving the evaluations weights of 1000, 100, and 10, respectively, each evaluation is an order of magnitude more important than the next.

After defining how GADGET should evaluate a possible solution, we need to provide iterations that indicate how GADGET should generate different possible solutions. Iterations can vary in complexity according to the optimization. For optimizations in which it is possible to identify that certain actions should be taken when certain conditions are met, an iteration might look to see if a condition is satisfied and then make an appropriate change. For optimizations in which it is less clear what steps will lead to a better solution, iterations can make random changes, relying on the evaluations to select appropriate changes. For this example, we will use a very simple iteration that selects a random character and nudges it a random distance in a random direction. Combined with our evaluations, this iteration yields the results in Figure 1.

At this point, it is worth noting that the solution we have presented does not require an optimization-based approach, as a very simple algorithm can achieve this result (and does so much faster). Picking an arbitrary starting point on the polygon, characters can simply be spaced at intervals equal to the total length of the polygon divided by the number of characters. However, this algorithm and our presented solution share a common problem illustrated by the overlap of characters in the corner of the triangle. To address this problem, Figure 2 shows the result of modifying our optimization by adding an evaluation that minimizes the overlap of the bounding rectangles of each character. Given a smaller weight than the other three evaluations, this additional criteria shifts characters around corners to avoid the overlap found in Figure 1. Adding this additional requirement to the algorithmic approach would be very difficult, but adding it to our optimization is very simple.

## GADGET ARCHITECTURE

The GADGET architecture has several features designed to support the exploration of optimization-based approaches to interface and display generation. A standard framework abstracts the concepts and constructs behind evaluations. Our base class, the GadgetObject, provides generic property support integrated with an efficient lazy evaluation framework. Our default optimization structure provides a variety of useful features, and our configurable optimization structure allows changes in the optimization structure to meet specific needs. Finally, GADGET supports a reusable library of iterations and evaluations.

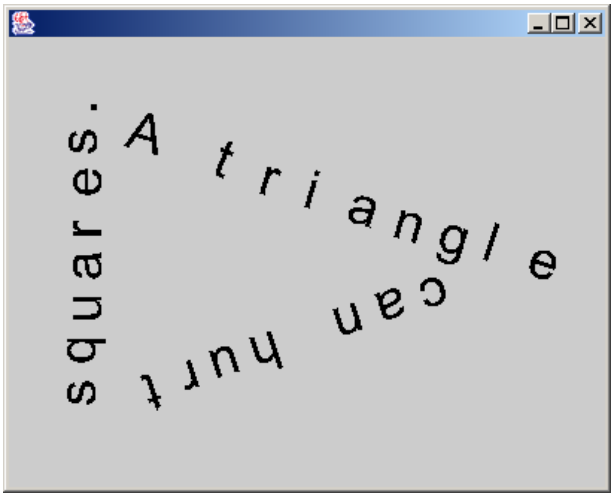


**Figure 2 – After Adding Overlap Minimization**

### Evaluation Standardization Framework

As illustrated in our previous example, GADGET allows programmers to focus on creating evaluations to measure criteria that are important to a problem. GADGET then combines these evaluations and uses them to choose between possible solutions to a problem. This process of combining evaluations and choosing between possible solutions has five stages. First, the framework presents each evaluation with the current possible solution, which we will call the *prior solution*. Each evaluation returns an array of double values representing its interpretation of the prior solution. We will call this collection of arrays of double values the *prior result*. Second, the framework uses an iteration object to modify the prior solution, creating a new possible solution that we will call the *post solution*. Third, the framework presents the post solution to each of the evaluations, and the individual evaluations return interpretations that are then combined to create a *post result*. Fourth, the framework uses a method described in the next paragraph to compare the prior result and the post result. Finally, the result of this comparison indicates whether the framework should accept the post solution or revert to the prior solution.

Up until this point, we have not imposed any particular structure on the arrays of double values that individual evaluations create to represent interpretations of possible solutions. Providing this flexibility to the programmer allows evaluations to use any appropriate representation. However, determining which solution is better requires a standard form for comparing results. We achieve this standard form by requiring each evaluation to be capable of comparing two arrays of double values that it has created and providing a double value in the range -1 to 1, where a -1 indicates the evaluation has a strong preference for the

prior solution, a 0 indicates the evaluation is indifferent, and a 1 indicates the evaluation has a strong preference for the post solution. To choose between a prior and post result, these values between -1 and 1 are multiplied by the weight associated with each evaluation and these multiplied values are summed. If this sum is greater than 0, the framework prefers the post solution.

While evaluations can use any method to create a value between -1 and 1 from the post and prior arrays of double values, they will most often use one of the methods we have built into the framework. These methods are "Minimize Sum of Squares", "Maximize Sum of Squares", "Maximize Minimum Value", "Maximize Maximum Value", "Minimize Minimum Value", and "Minimize Maximum Value". Each of these methods uses a formula resembling that in Figure 3 to create a value based on both the sign and magnitude of a difference.

```
priorSum = sumOfSquares(priorResult)
postSum = sumOfSquares(postResult)

if(postSum = priorSum) then result = 0
else result = sign(priorSum – postSum) *
             (1 – ( min(postSum, priorSum) /
                    max(postSum, priorSum) ))
```

**Figure 3 – Standardization Formula for the Minimize Sum of Squares Method**

### Generic Properties and Lazy Evaluation Framework

GADGET classes inherit from a base GadgetObject class that provides support for generic properties. Objects can create named properties to store a value on any object. Other objects can then use the stored value. As we will discuss in our section on a reusable library of iterations and evaluations, this support for storing and referencing property values helps programmers create general solutions. For example, the overlap minimization evaluation that we used in our earlier example is created from a list of objects and the property name for a bounding rectangle that is stored on each object. The evaluation uses the property to get the rectangles from the objects and compute their overlap. Therefore, the evaluation can be used to minimize the overlap of any set of rectangles, not just the rectangles associated with the characters in our earlier example.

Generic property support in GADGET is tightly integrated with an efficient framework for lazy evaluation. Based on the lazy evaluation algorithm presented in [13], the framework uses local flags and local dependency pointers to avoid unnecessary computations. This lazy evaluation framework is integrated with our generic properties by the GadgetComputedProperty class. Computed properties are declared just like any other property, by creating a named property and using it to store the computation on an object. Instead of storing a value, the property stores a computation. The first time the value of this property is requested, the computation is executed to compute the value. Each subsequent time the value is requested, the computation is executed only if the value might have changed since it was last computed.

Although the lazy evaluation framework requires careful bookkeeping to ensure that cached values are always correct and computations are not unnecessarily executed, the GADGET framework is responsible for all of this bookkeeping. Programmers define a computation by overriding two functions of a base computation class. The first function, computeValue, simply performs the desired computation and returns the computed value. The second function, declareDependentOn, is called by the framework immediately before the first call to computeValue. In this function, the programmer declares what objects this computation is dependent on. The framework then handles caching a computed value and monitoring the computation's dependencies.

As an example of the combined power of generic properties and the lazy evaluation framework, consider the evaluation from our earlier example that minimized the distance from each character to the nearest edge on the polygon. This evaluation is constructed from a list of objects, a named property for a location stored on each object, and the polygon that these locations should be resolved against. A naïve implementation of this evaluation would simply go through each object, extract the location of that object, compute the distance from the object location to each edge on the polygon, select the minimum distance, and build an array of minimum distances. Note, however, that our iteration only moves one character between evaluations. Therefore, the computation to determine what edge each point is closest to is wasted for all but one of the characters. Using the generic property and lazy evaluation framework, this same evaluation can be programmed much more efficiently. The evaluation declares a computed property that is the distance from a point to a polygon. It then instantiates this computed property on each object in the list. Each of these individual computations is dependent only on the location of the polygon and the location of that character. Therefore, each computation will be executed only if that character has moved or the polygon has changed since the computation was last executed. The cached value will be used for all of the other characters. When arranging twenty characters on a polygon, this change can represent as much as a twenty-fold improvement in the execution time of an evaluation. Because optimization can be a computationally expensive process, the ability to easily make these sorts of improvements without specifically burdening the programmer is important to this toolkit.

### Powerful Default Optimization Structure

By default, GADGET uses a simulated annealing approach to optimization [17]. Simulated annealing is a general approach that is characterized by a temperature variable. This temperature variable is initially high, indicating a "hot" system, and decreases over time, representing the system gradually "cooling" into an optimal state. This

temperature variable is used to probabilistically accept changes that do not appear to represent an improvement. By randomly accepting these changes, an optimization is less likely to become trapped in local maxima.

This temperature variable can also be used by evaluations and iterations to guide their decisions in an optimization. Iterations, for example, can make large changes to a solution when the temperature is high and smaller changes when the temperature is low. Similarly, evaluations can use the temperature variable as an indicator of whether to use estimation techniques instead of computing a full evaluation. For example, an evaluation that performs an expensive computation on objects in a list might use the temperature variable to randomly sample only a percentage of the objects in the list. Such estimates allow an optimization to approximate a desired solution while the temperature is high and settle into a more precise solution as the temperature cools.

Our default optimization structure also includes several other features that are helpful for creating optimizations. Because the weights associated with evaluations represent the maximum value that a particular evaluation can contribute to the decision on whether to accept the iteration, GADGET automatically avoids executing evaluations that cannot affect the outcome of the decision. We also provide event notifications intended to allow the programmer to visualize an optimization that is in progress. Finally, the default optimization structure uses knowledge about whether changes are accepted or rejected to maintain a cached version of the prior evaluation result, thus preventing unnecessary execution of evaluations.

#### Configurable Optimization Structure

In order to support the future development of GADGET and to allow optimizations to make changes according to their specific needs, GADGET uses a configurable optimization structure. An optimization structure is represented as a finite-state machine. Each state in the machine represents an action in the optimization, such as sending an event notification, executing an evaluation, or decaying the temperature variable. Each state has some number of exit conditions, and the optimization structure associates each exit condition with a transition to another state. This design allows changes to be made to an optimization structure by adding new states and changing the transitions associated with exit conditions. It also allows actions to be reused in entirely new optimization structures.

Because most optimizations will only need to use a standard optimization structure, GADGET uses Builder objects to abstract the configuration process. For example, our default optimization structure is created by a single function call to a Builder object that creates a simulated annealing optimization based on a handful of parameters, including a list of the evaluations to execute, a list of the iterations to use, and the rate at which the temperature should decay. Programmers that create custom optimization structures could extend this Builder to create an optimization structure that resembles our default structure, such as a simulated annealing structure that added support for logging all of the possible solutions that were considered. Programmers could also create entirely new Builder objects for structures much different from our default structure, such as an optimization based on a genetic algorithm. Builder objects then allow programmers to easily use the custom optimization structures.

#### Reusable Library of Iterations and Evaluations

GADGET supports a library of reusable evaluations and iterations intended to help programmers quickly create optimizations. Although our library is still relatively small, we have developed several promising approaches to creating an effective library.

One approach we are pursuing is a standard approach to reusable libraries. As discussed in our section on generic properties, the overlap minimization evaluation used in our text layout example can be easily implemented to minimize the overlap of any set of rectangles. Similarly, the iteration used to randomly nudge characters can be easily implemented to randomly nudge any objects with a property that defines their locations. The use of named properties on objects makes this type of reuse easy to include. This approach to a reusable library also results in reusable components that clearly implement a particular evaluation or iteration. Programmers who are new to GADGET can then create optimizations by combining these clearly named evaluations and iterations, treating them as black boxes.

Although the above approach works well for many components, coverage can be problematic. Given this problem, we are pursuing an approach that makes extensive use of computed properties. To understand how this approach is structured, consider the evaluation that ensures characters are displayed in the correct order in our text layout example. It penalizes each pair of characters that appears on the polygon in an order that is different from the order they appear in the string. If this evaluation is programmed to use a list of characters, a named property for the location of each character, and the polygon that the characters are being arranged on, the evaluation will have relatively little reuse potential. The reuse potential of the evaluation is improved substantially if it is instead programmed to use a list of objects, a named property for the desired order of each object, and a named property for the current order of each object. In this case, the desired order is the index of that character in the string being arranged. The current order is a computed property that determines how far that character is located along the edge of the polygon. The evaluation uses the desired order and current order properties to penalize each pair of objects that is not in the desired order. Structured this way, the evaluation can be used to maintain an arbitrary desired order for any list of objects. For example, our partial

implementation of the LineDrive system, presented in a later section, uses this evaluation to prevent short roads from appearing longer than long roads. In that case, the desired order is the undistorted length of each road and the current order is the length of each road after distortion.

The two approaches just presented help create reusable individual GADGET components. A third approach, intended for use with both of these approaches, allows groups of components to prevent unnecessary duplication of a computation. If several components require an expensive intermediate computation, any one of them can create a computed property, store it on a shared object, and expose the property name. The other components can then reuse this computed property. The framework will automatically ensure that the value is up to date prior to every use, but will never be recomputed unnecessarily. For example, several evaluations might share a computation to compute the convex hull of a list of objects, with one using the convex hull computation to efficiently find the two most distant objects and a second evaluation using the convex hull computation to compute the total area covered by the objects. An iteration could also reuse this convex hull computation to move objects on the hull towards the middle of the hull. The ability to transparently share these sorts of computations among components seems to be important for helping to create efficient optimizations.

## EXAMPLE: BUBBLEMAPS

In this section, we will shift from discussing GADGET as a general toolkit to illustrating how GADGET can solve a particular problem. The problem we discuss arises in the context of the Bubblemap layout algorithm created for PhotoMesa, a zoomable image browser that groups images according to a shared attribute [4]. The problem addressed by Bubblemap is the arrangement of semantically clustered rectangles into approximately rectangular regions without wasting space. In other words, the rectangles must fit into the smallest possible total area, but related clusters should be arranged in that area so that they approximate rectangular shapes. The method used by Bubblemap is a greedy pixel-based bucket fill algorithm. As can be seen in Figure 4, the algorithm yields fairly good results. Related rectangles, represented here by color, are kept together and the edges between clusters generally form clean boundaries.

Although Bubblemap produces solutions that are usually acceptable, it sometimes suffers from the fact that it is a greedy algorithm. As can be seen in the bottom of Figure 4, the last cluster of rectangles to be put into the solution must settle for whatever space is left. The border of this space is sometimes jagged and detracts from the overall quality of the solution. To address this sort of problem, we have created an approach that combines Bubblemap with an optimization. After using Bubblemap to create a reasonable solution, we explore other nearby possible solutions to see if they are better. This combination of algorithms and optimization seems to be a promising overall approach.

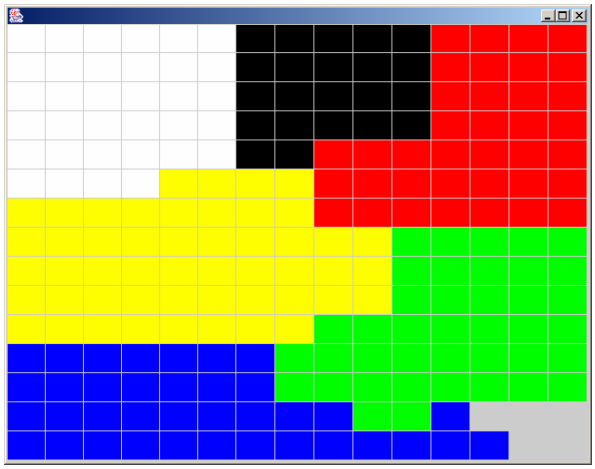

**Figure 4 – A Bubblemap Layout of Six Groups**

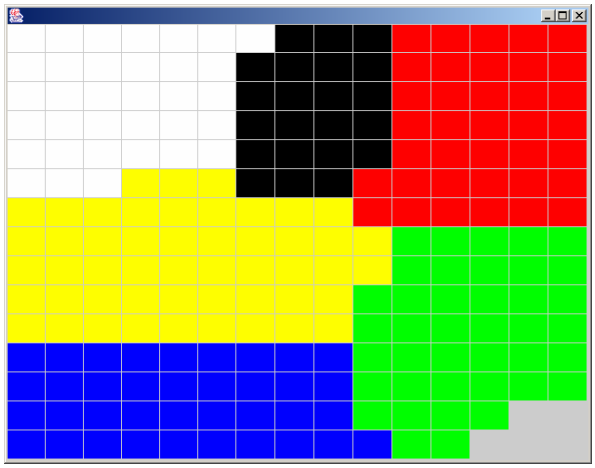

**Figure 5 – The Same Layout After Optimization**

Figure 5 shows the result of applying an optimization to the result created by Bubblemap that is shown in Figure 4. The optimization we applied uses a handful of evaluations. The most heavily-weighted evaluation ensures that all the rectangles of a particular cluster are connected. We then use evaluations that minimize the area of the bounding rectangle around each cluster and minimize the area of the convex hull around each cluster. Finally, an evaluation minimizes the number of times the clustered value changes in each row and column of layout.

The iteration used in this optimization demonstrates one approach to creating iterations for difficult problems. We use a swapping approach, selecting pairs of rectangles and swapping their values. While there may be a way to find pairs of rectangles that should be swapped, we have not found any simple criteria for selecting pairs of rectangles with a high success rate. While randomly selecting any two to swap could work, substantial efficiency is gained by instead using random selection from a list of rectangles that meet certain criteria. In this case, we only consider swapping rectangles that border two rectangles of a different color (counting edges of the layout as rectangles of a different color). This limits swapping to the corners of

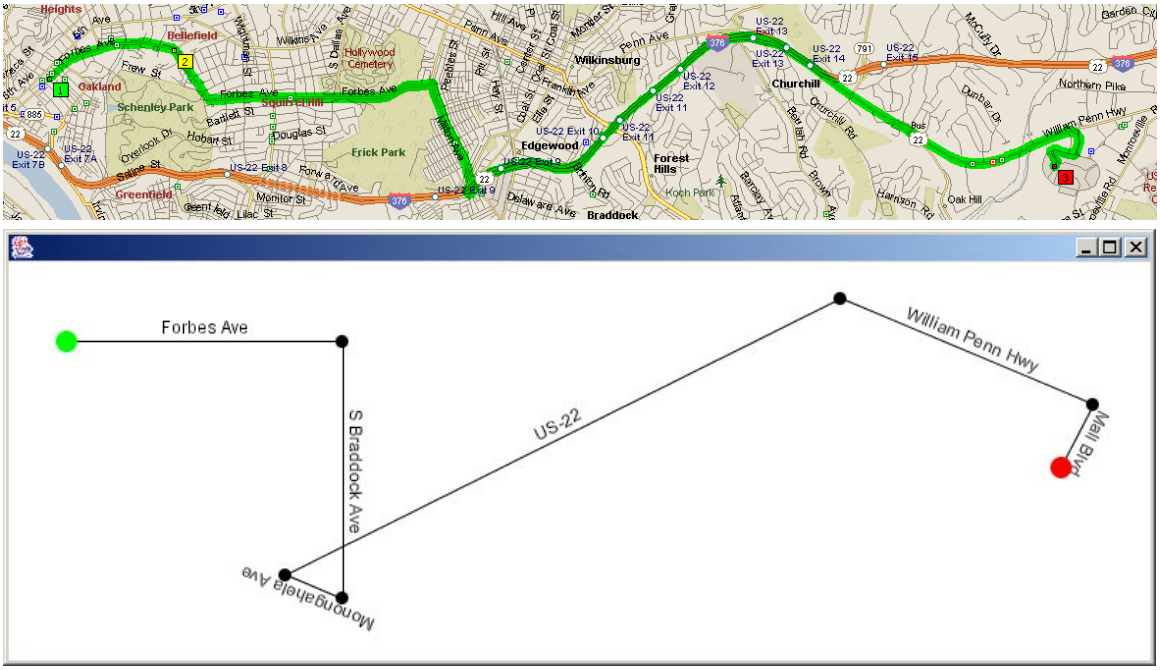


**Figure 6 – Undistorted and distorted versions of a route map.**

**Note that the loop at Monongahela Avenue is imperceptible without distortion.**

clusters, which limits swapping to pairs that are more likely to result in improvements when swapped. Further, the computation to determine which rectangles should be considered for swapping is very inexpensive, based on a Boolean computed property on each rectangle and a list that stores only the rectangles for which this property is true. In the general case, problems for which random selection is used as the iteration method can benefit from simple filters that increase the efficiency of the random selections.

## EXAMPLE: LINEDRIVE

The LineDrive system is used to generate route maps at http://www.mapblast.com [2]. These maps simplify and distort the information in a typical road map to more clearly convey the information critical to navigating the route. Figure 6 shows two views of a route from Carnegie Mellon University to a nearby major shopping center [19]. This route is particularly interesting because it includes a loop at Monongahela Avenue that can easily confuse a first-time driver of this route. This loop is imperceptible in a typical road map because only a very short section of Monongahela Avenue is included in the route. In the map generated by our partial implementation of the LineDrive system, the size of this route segment has been distorted to ensure that it is visible. The resulting map makes it clear to a first first-time driver of this route that they will not be able to go directly from South Braddock Avenue to US-22.

As a deployed real-world example of an optimization-based approach to interface and display generation, it is important that the techniques used by the LineDrive system are supported by GADGET. We can also use the LineDrive system to provide ideas about effectively designing optimization-based approaches to interface and display generation. In this section, we present a partial implementation of the LineDrive system and comment on how some approaches taken by the LineDrive system might generalize to other optimizations.

One interesting facet of the LineDrive system is the decision to split the optimization into multiple distinct stages, each of which optimizes a different set of conditions. The LineDrive system uses three such optimization stages: an optimization stage that distorts road length and direction to ensure all roads in a route are visible without creating false or missing intersections, an optimization stage that places labels on each road in the route, and an optimization stage that includes contextual features (such as cross-streets or landmarks). These optimization stages are complimented by algorithmic stages to simplify geographic data into road segments before the first optimization stage and to add decorative graphics to the route map after the last optimization stage. By dividing the many different parts of the optimization into stages, the LineDrive system significantly reduces the size of the problem that is solved by the optimization.

As a general approach, the division of a large optimization into smaller optimizations seems to have helpful properties. Because each optimization searches a much smaller space, each optimization can be executed much more quickly. Smaller optimizations might also make it clearer what evaluations and iterations will quickly lead to a solution. These helpful properties, however, need to be balanced against the possibility that an early optimization might create a problem that a later optimization cannot remove.

For example, with the road location fixed prior to the label placement optimization, it might be impossible to place labels without creating overlaps. If the label placement optimization were able to modify road placement, it could handle this problem. Given this possibility, the decision of whether it is appropriate to conduct an optimization in stages is problem dependent.

It is also worth noting that the LineDrive system uses very specific evaluation criteria designed to detect and prevent very specific problems. For example, false intersections are prevented with an evaluation that minimizes the distance from a false intersection to either end of the route. This pulls the false intersection until it eventually passes the endpoint, thus removing the false intersection. Another evaluation pulls together the roads from a missing intersection. Because these evaluations can conflict, a third evaluation identifies missing intersections contained in the loop created by a false intersection. This third evaluation very rarely applies, but it helps the optimization recover from situations where the first two evaluations conflict.

As a general approach, creating evaluations that identify and resolve very specific problems in an optimization seems to be beneficial. If a broad and general evaluation is effective for all but a very specific case, creating an evaluation for that specific situation seems preferable to trying to find a different broad evaluation that is effective in every case. As discussed earlier, this is one of the benefits of optimization as an approach to display and interface generation. Instead of trying to construct a single algorithm or evaluation to handle every condition, independent goals and constraints can be mixed to achieve the desired affect.

We have created a partial implementation of the LineDrive system to evaluate the compatibility of GADGET with the techniques used by the LineDrive system. Our implementation includes a full implementation of the road layout stage, which distorts road length and direction to ensure all roads are visible without creating false or missing intersections, and a partial implementation of the labeling optimization stage. In our implementation, we have focused on the portions of the problem that relate to optimization, rather than the portions related to geographic databases, rendering, and related graphics details. These optimizations were straightforward in the GADGET architecture. Figure 6 shows an undistorted route map and a distorted map generated by our implementation.

## EXAMPLE: AUTOMATED DIALOG LAYOUT

Automated dialog layout is a problem that has been investigated by a variety of systems [5, 16, 21]. One approach to this problem is the right/bottom strategy, which places each component either to the right of or below the previous component, according to a set of rules. An important difficulty with this strategy is the development of an effective set of rules. Other rule-based approaches can have similar problems.


**Figure 7 – An automatically generated dialog layout.**

Figure 7 shows an automatically generated dialog layout that was created using a combination of optimization and the right/bottom strategy. The desired order, text, and size of these components are taken from a previous discussion of automated layout [5]. Our layout was created using an optimization to determine whether to place each component to the right of or below the previous component. All of the components are initially placed into one long column. An iteration toggles a Boolean property on each component to indicate whether that component should be placed to the right of the previous component, and the layout that results from applying the right/bottom strategy with these Boolean properties is judged by a set of evaluations.

The evaluations used in this optimization are relatively simple. One evaluation tries to keep labels associated with a component (as indicated by a flag in the input specification) on the same row as that component. Another evaluation minimizes the size of the dialog and the size of each group box. A third evaluation minimizes the amount of unused space in the dialog and in each group box. The final evaluation penalizes layouts in which a vertically large component appears to the right of a much smaller component, a situation which has previously been identified as visually unpleasant [5]. While additional evaluations could improve the robustness of this approach, these four evaluations are sufficient for creating the layout shown in Figure 7. The rules used in our right/bottom strategy implementation are also very simple. Components are placed a uniform distance from each other, and similar components in two or more adjacent rows are left-justified. Additional rules could right-justify the OK and Cancel buttons and handle similar highly specific layout issues.

This hybrid strategy of using an algorithm inside an optimization demonstrates an interesting point. The algorithmic portion efficiently handles the well understood portions of the problem, such as spacing and aligning components. The optimization handles the less well understood portion of the problem, deciding on an overall arrangement of the components. The resulting system is both efficient, generating the layout in Figure 7 in just over one second, and flexible, appropriately arranging the groups of radio buttons even though it has no specific knowledge of radio buttons. This seems like a promising strategy for structuring optimization-based approaches.

## DISCUSSION AND RELATED WORK

The pursuit of optimization as a general approach to interface and display generation relates to previous work on constraints [3, 11, 12, 18, 25] and constraints in user interface toolkits [10, 26]. Some systems, such as Cassowary [3], explicitly use numerical optimization to maintain constraints. It is also common to associate weights with constraints, indicating that conflicts between constraints should be resolved in a manner that is consistent with the more heavily weighted constraint.

Many constraint-based approaches can be difficult to use when it is not clear how to express the desired constraint in the appropriate limited form, such as a linear equation or inequality. Recent work has demonstrated a technique for using sets of linear constraints to approximate nonlinear constraints [14], but the general problem still exists. Instead of requiring programmers to represent their evaluations as equations, GADGET allows programmers to use arbitrary code in an evaluation and provides a standardization framework that allows these arbitrary criteria to be combined. This flexibility is important to providing general support for the types of problems for which GADGET has been designed. There is clearly a tradeoff between this flexibility and efficiency, though we believe that the techniques illustrated in our examples, such as using algorithms to approximate a good solution and dividing large optimizations into multiple stages, provide some insight into how to manage this tradeoff. As a part of future work, we intend to explore additional methods to provide toolkit support for strategies to manage efficiency.

GADGET also relates to previous work on metric-based design [20, 23, 24], automated usability analysis [15], and automated dialog layout generation [5, 16, 21]. In order to emphasize the flexibility of GADGET, this paper includes examples outside of traditional dialog layout problems. Some metrics previously presented in the context of dialog layout, such as symmetry and balance, are likely to be useful as general aesthetic qualities in a large variety of problems. By supporting a library of reusable evaluations, GADGET can make it easy for programmers and designers to include these types of generally useful evaluations in their systems. Other metrics specific to dialog layout, such as Layout Appropriateness [24], suggest approaches to including task knowledge in evaluations. Layout Appropriateness evaluates the positions of components according to how often a user will move from one component to another, attempting to minimize the distance a mouse must be moved during common sequences. The flexible nature of GADGET evaluations supports the use of this type of task knowledge in a dialog layout problem. Stated more generally, GADGET supports not only evaluations based on the arrangement of visual elements, but also evaluations based on automated usability analysis, knowledge of human perception, and the amount of information conveyed by parts of an information display.

Optimization has also been used extensively as an approach to problems in graph visualization and VLSI layout [6, 8]. Many problems in these areas are NP-Complete, making optimization-based approximation techniques important. This work, appropriately, is often conducted at a level of mathematical complexity that is well beyond the comfort level of a typical programmer. GADGET works to make the benefits of optimization that have been demonstrated in these fields more approachable to typical programmers. We expect that making optimization easier to use will result in it being used for additional problems.

In our experiences with GADGET, we have found that our abstractions and reusable components make it easier to develop optimizations. A new evaluation can be quickly combined with an existing set to determine if the additional evaluation yields a better result. The lazy evaluation framework also seems to be very effective, allowing optimizations to consider many different small changes without the computational overhead of re-evaluating large parts of the problem that are unaffected by the changes.

The primary difficulties that we have encountered in using GADGET are those that we might expect in any approach to optimization. As the number of variables that can be manipulated in an optimization grows, it becomes difficult to ensure that an appropriate solution can be quickly found. This is partially because the there are many more points in the space to be considered, but is also due to the increasing difficulty of understanding the optimization space and deducing what new evaluations might improve results.

In discussing our examples and the functionality provided by GADGET, we have shown techniques for managing these problems. We are particularly interested in hybrid approaches that use a combination of an algorithm and an optimization to solve a problem. We are also interested in additional toolkit-level support for these hybrid approaches and other strategies to help people understand and improve upon non-trivial optimizations.

## CONCLUSIONS

Optimization-based approaches to interface and display generation seem to be a promising area of research, but important advances still need to be made. Carefully constructed optimizations can run quickly enough for use

with current systems, as illustrated by the LineDrive system, but other optimizations remain too computationally expensive for current systems. While part of the solution to this problem will come from the exponential improvement of computing speed described by Moore's Law, it is also important to develop strategies for managing efficiency and approachable abstractions of more efficient optimizations.

We have presented GADGET, a toolkit designed to support optimization-based approaches to interface and display generation. This toolkit provides several features designed to support the easy creation of efficient optimizations. These include a standard framework to abstract much of the mechanics behind evaluations, generic property support integrated with an efficient lazy evaluation framework, a powerful and configurable optimization structure, and a library of reusable iterations and evaluations. We have presented strategies for developing reusable and efficient components and have demonstrated the use of GADGET in three interesting optimization problems. As an appropriate tool for this new class of interface and display generation techniques, GADGET provides important support for the further exploration of these techniques.

## ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation under Grant IIS-01215603 and the first author's NSF Graduate Research Fellowship. We would like to thank Jodi Forlizzi and Joonhwan Lee for identifying the interesting route used in our LineDrive example and providing the undistorted image used in that example. We would also like to thank Jeffrey Nichols for his comments on automated dialog layout and versions of this toolkit.

## REFERENCES

1. Abbot, E.A. (1884) *Flatland: A Romance of Many Dimensions*.
2. Agrawala, M. and Stolte, C. (2001) Rendering Effective Route Maps: Improving Usability Through Generalization. In *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*, 241-249.
3. Badros, G.J., Borning, A. and Stuckey, P.J. (2001) The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *8* (4). 267-306.
4. Bederson, B.B. (2001) PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2001)*, 71-80.
5. Bodart, F., Hennebert, A.-M., Leheureux, J.-M. and Vanderdonckt, J. (1994) Towards a Dynamic Strategy for Computer-Aided Visual Placement. In *Proceedings of the Workshop on Advanced Visual Interfaces*, 78-87.
6. Cong, J., He, L., Koh, C.-K. and Madden, P.H. (1996) Performance Optimization of VLSI Interconnect Layout. *Integration, the VLSI Journal*, *21* (1). 1-94.
7. Decision Tree for Optimization Software. http://plato.la.asu.edu/guide.html
8. Di Battista, G., Eades, P., Tamassia, R. and Tollis, I.G. (1999) *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.
9. Fogarty, J., Forlizzi, J. and Hudson, S.E. (2001) Aesthetic Information Collages: Generating Decorative Displays that Contain Information. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2001)*, 141-150.
10. Gleicher, M. (1993) A Graphics Toolkit Based on Differential Constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1993)*, 109-120.
11. Hentenryck, P.V. and Saraswat, V. (1996) Strategic Directions in Constraint Programming. *ACM Computing Surveys (CSUR)*, *28* (4). 701-726.
12. Hosobe, H. (2001) A Modular Geometric Constraint Solver for User Interface Applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2001)*, 91-100.
13. Hudson, S.E. (1991) Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *13* (3). 315-341.
14. Hurst, N., Marriott, K. and Moulder, P. (2002) Dynamic Approximation of Complex Graphical Constraints by Linear Constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2002)*, 191-200.
15. Ivory, M.Y. and Hearst, M.A. (2001) The State of the Art in Automating Usability Evaluation of User Interfaces. *ACM Computing Surveys (CSUR)*, *33* (4). 470-516.
16. Kim, W.C. and Foley, J.D. (1993) Providing High-Level Control and Expert Assistance in the User Interface Presentation Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1993)*, 430-437.
17. Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. Optimization by Simulated Annealing *Science*, 1983, 671-680.
18. Marriott, K. and Stuckey, P. (1998) *Programming with Constraints: An Introduction*. MIT Press, Cambridge, MA.
19. Microsoft Streets and Trips 2002. http://www.microsoft.com/streets/
20. Ngo, D.C.L., Samsudin, A. and Abdullah, R. (2000) Aesthetic Measures for Assessing Graphic Scenes. *Journal of Information Science and Engineering*, *16* (1). 97-116.
21. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R. and Pignol, M. (2002) Generating Remote Control Interfaces for Complex Appliances. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2002)*, 161-170.
22. Nocedal, J., and Wright, S.J. (1999) *Numerical Optimization*. Springer, New York.
23. Sears, A. (1995) AIDE: A Step Toward Metric-Based Interface Development Tools. In *Proceedings of the ACM Symposium on User Interface and Software Technology (UIST 1995)*, 101-110.
24. Sears, A. (1993) Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. *IEEE Transactions of Software Engineering*, *19* (7). 707-719.
25. Vander Zanden, B. (1996) An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *18* (1). 30-72.
26. Vander Zanden, B., Halterman, R., Myers, B.A., McDaniel, R., Miller, R., Szekely, P., Giuse, D.A. and Kosbie, D. (2001) Lessons Learned About One-Way, Dataflow Constraints in the Garnet and Amulet Graphical Toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *23* (6). 776-796.