# A Collaborative Approach to Newspaper Layout

**Citation**
Lubin, Benjamin. 1999. A Collaborative Approach to Newspaper Layout. Harvard Computer Science Group Technical Report TR-04-99.

**Permanent link**
http://nrs.harvard.edu/urn-3:HUL.InstRepos:23853808

**Terms of Use**
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

## Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

# A Collaborative Approach to Newspaper Layout

**Benjamin Lubin**

TR-04-99

May 10, 1999

## Acknowledgements

I would like to thank the following people for their kindness and support, withou which this project would not have been possible: my advisor, Professor Stuart Shieber, Professor Barbara Grosz, Elisa Cheng, Mat Glatthorn, B.J. Trac , Steve, Wendy and Nathaniel Lubin. This project is an expansion of the principles in the GLIDE syste created by Kathleen Ryall, Stuart Shieber and Joe Marks, and could not have been realized without their tremendous effort. I would like to thank Eric Feigin for reading a draft and generating such insightful comments and to thank Ingrid, Gus and Augusta for all their help. This material is based in part upon work supported by the National Science Foundation under Grant No. IRI-9618848 to Stuart M. Shieber.

© 1999 Benjamin Lubin
All Rights Reserved.

# Table of contents

Acknowledgement ........ 1
Table of content ........ 2
Chapter 1: Introduction ........ 4
1.1 The Problem ........ 4
1.2 Our Solution ........ 4
Chapter 2: Related work ........ 8
2.1 Existing commercial software ........ 8
2.1.1 Approach of existing commercial software ........ 8
2.1.2 Limitations of existing commercial software ........ 8
2.2 Existing research software ........ 8
2.2.1 Juno ........ 9
2.2.2 GLIDE ........ 9
Chapter 3: A new paradigm for single-user layout tool ........ 11
3.1 The layout hierarchy ........ 11
3.1.1 A description of the node elements ........ 11
3.1.2 A description of the layout constraints ........ 13
3.1.2.1 Low-level layout constraints and syntactic requirements ........ 13
3.1.2.2 Mid-level layout constraints ........ 14
3.1.2.3 High-level constraint ........ 14
3.2 The content hierarchy ........ 15
3.3 Relating the two hierarchies: duct constraints ........ 16
Chapter 4: The LILT user interface ........ 17
Chapter 5: An example interaction with LILT ........ 19
Chapter 6: Algorithms ........ 37
6.1 The overall document model ........ 37
6.2 Specifics of the layout model ........ 38
6.2.1 Node element algorithms ........ 38
6.2.1.1 The ApplyUpdates functi n ........ 39
6.2.1.2 The ApplyConstraints functi ........ 39
6.2.1.3 The FreshenNodes function ........ 39

6.2.1.4 Required node attributes and behaviors ............................. .................. 40
6.2.2 Constraint and Update element algorithms............................ ................. 40
6.2.2.1 low-level constraints............................. ............................. ............... 41
6.2.2.2 Mid-level constraint ............................. ............................. .............. 42
6.2.2.2.1 Width and Height............................. ............................. ............ 42
6.2.2.2.2 Horizontal and Vertical Alignment ............................. ................. 43
6.3 Specifics of the content model............................. ............................. ............ 43
6.4 Specifics of the connecting ducts............................. ............................. ........ 43
Chapter 7: Implementation Issues............................. ............................. ................ 46
Chapter 8: Possible future improvements for the layout paradigm............................... 47
Chapter 9: The multi-user case: concurrent editing within the workflow ...................... 48
9.1 The problem............................. ............................. ............................. ....... 48
9.2 Current approaches............................. ............................. ............................ 49
9.3 Towards a solution ............................. ............................. ............................ 49
Chapter 10: Conclusio ............................. ............................. .............................. .51
Chapter 11: Glossary ............................. ............................. ............................. .....52
Chapter 12: References............................. ............................. .............................. ..54

# Chapter 1: Introduction

Originally, newspapers were typeset by hand. Every letter on every page was formed by an individual metal stamp that was carefully fitted into a tightly packed frame. Pictures were included using either wood-block carvings or stock metal stamps that could be placed into the frames with the text. With the invention of photo-reproductive processes it became possible not only to use actual pictures, but to avoid using individual characters of type as well. The text was typed up and then "cut and pasted" into place on large layout boards. These boards were reproduced to make the pages. With the adven of computers, high-end systems appeared for fully-automated layout, though these systems had severe limitations. Then, with the rise of personal computers, the age o ubiquitous "desktop publishing" began. But even the current crop of desktop publishing programs do little more than enable the user to cut and paste text and graphics in a digita form, almost exactly as it used to be done with paper and glue.

## *1.1 The Problem*

The aim of this thesis is to change the paradigm used in layout software, making the task of layout easier and far more enjoyable. In accomplishing this, the presen project employs two related means. The first is to invent an engine that can automate portions of the layout process in an intelligent manner. The second is to create a user interface that facilitates collaboration between this engine and the user. The planning o such an interface is a complex problem, because newspaper production is almost always a group activity requiring a high degree of collaboration. Ideally, the software should mediate this group interaction. But before this can be accomplished, a sound single-user interface is required. The present project focuses on this necessary first step in solving the more general multi-user case.

## *1.2 Our Solution*

In order to address these issues, a program called **LILT** (E<u>L</u>astic <u>I</u>nteractive <u>L</u>ayout <u>T</u>ool) has been developed. It provides a framework for semi-automated layout in the newspaper domain and the means for humans and computers to cooperate in the endeavor. It introduces more intelligence into the software, letting the software handle the low-level details and allowing the user to concentrate on the larger aesthetic issues. Furthermore, **LILT** bypasses the cut-and-paste problem by being dynamically oriented to the newspaper as a whole, rather than to single pages.

In formulating this novel solution, our first task is to create a formalism that can represent the newspaper domain within a constraint satisfaction framework. In doing this we first divide the problem space into two interrelated hierarchies: layout and content. The layout hierarchy is comprised of all the information pertaining to the look of the document — both the elements to be laid out and the constraints that relate them. The content hierarchy contains all the information in the newspaper: text and pictures. Creating this division has two main advantages. First, it allows for modification of each hierarchy predominantly independently of the other. Second, it gives explicit control to the user over the implicit hierarchical structure which is present in the domain. But in order for the division to work, there must be communication between the two parts, since

they are clearly not entirely independent. The task of binding these hierarchies together within the automation framework represents an important part of the present study.

A solution to the newspaper domain problem requires a method for the machine to satisfy the constraints the user has specified in the layout hierarchy, either overtly or indirectly. The method employed in this project is a mass-spring simulation based upon the one used in the GLIDE graph editor,[1] as described in section 2.2.2. However, the approach taken in GLIDE has been greatly extended in numerous ways. Among these are, first, that in developing **LILT**, we have made possible the application of constraints to a much more complicated set of objects. Also we have introduced an explicit update step into the simulation. By considering each portion of the layout as an individual mass and casting each of the constraints into a language of either springs or codable heuristic rules (updates), the machine can attempt to satisfy the user's specifications for the layout. For example, if the user specifies that two objects should be adjacent, the software would interpret this constraint as a single spring, as illustrated in the following figure:

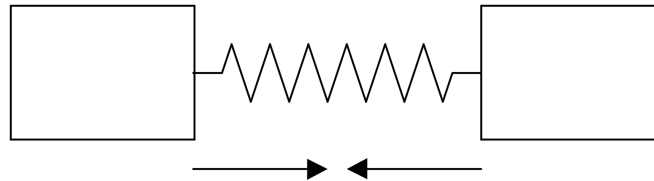

**Figure 1: A spring-mediated constraint**

The spring is attached to the right side of the left object and to the left side of the right object. It is given a rest length of zero, which causes it to apply an inward force to both objects as the simulation progresses. This will tend to cause the two objects to move towards each other whenever they are separated, and away from each other whenever they overlap (at least in the x-axis).

It is worthwhile examining a short example of how this type of automation works. Suppose the user instructs the system to create three new **Blocks**. **Blocks** are areas for displaying some type of content (note that all the words in **Bold** type are included in a glossary on page 52 that describes their specific meaning within the **LILT** framework). The user can then click and drag these blocks to new locations and end up with something like the following partial screen capture:

[1] See Ryall et al. "An Interactive Constraint-Based System for Drawing Graphs"

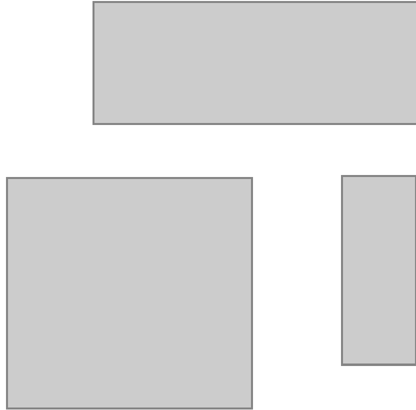

**Figure 2: Layout after creating three Blocks**

The next thing the user is likely to want to do is specify how these **Blocks** relate to one another in order to produce an elegant layout. For example, the user might specify the following constraints:

- The bottom two **Blocks** should have the same widths
- The combined width of these **Blocks** should be the same as the width of the top **Block**
- The left edge of the top **Block** and the left edge of the left-bottom **Block** should be co-linear in the y-axis.

After the user has applied these constraints, the system will move the blocks in order to satisfy them, producing the following:

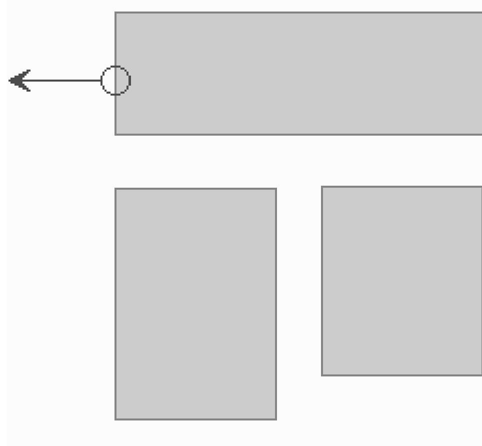

**Figure 3: Layout after applying constraints**

Because these constraints are updated dynamically, if the user drags the top **Block's** left edge to widen the top **Block** (as indicated by the arrow in Figure 3), the system will adjust the other **Blocks** to maintain the indicated relationships. Thus after the mouse-drag, the layout will look like this:

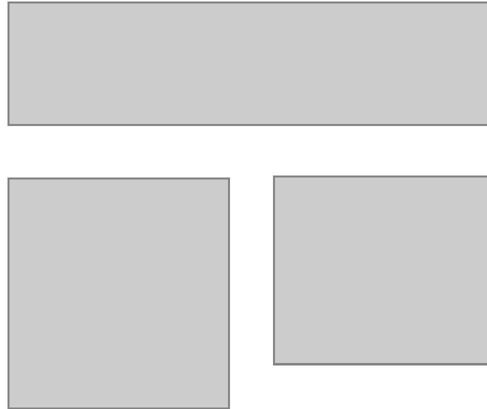

**Figure 4: Layout after mouse-drag maneuver**

Although this example has shown only a tiny fraction of the **LILT** system, it is already clear that mass-spring-update constraint satisfaction can be very effective in the newspaper domain. Other constraint satisfaction methods can produce layouts by employing some limited, rigidly defined aesthetic evaluation function, but in the newspaper domain, where flexibility is essential, it is all but impossible to define an aesthetic evaluation function, which strongly selects for **LILT's** approach. In this syste there is no need for such an evaluation function, as the user him- or herself acts as the evaluator. In this sense, the layout process becomes a true collaboration between the machine and the user: the machine does its part by satisfying the constraints to the best o its ability, and the user views the results and modifies the constraints to better match the desired layout. The mass-spring-update model provides an excellent medium for this communication, as its physical nature and real-time evaluation create an intuitive user interface. The user can anticipate how the computer will be likely to react to new input, an important aspect of any collaboration. Further, the model is a good choice because it gracefully handles both under- and over- constrained layouts, conditions that arise frequently. This mass-spring-update approach should scale well to the multi-user case also, for reasons that will be explored further on.

The code for **LILT** has been written in such a way that it should be able to provide a base for developing programs based on the same technology, but for very different applications. (One such application might be the placement of objects in architectural models, which would be a rather close three-dimensional analog to this project.)

# Chapter 2: Related work

## 2.1 Existing commercial software

In light of our goal of ease of use, we target the lower end of newspaper production software. We are concerned with layouts that go beyond simple word-processing, but are not as particular as layout for *The New York Times*, where one can afford to do everything by hand and ease of use is not an issue. Currently there are two major commercial products for doing this type of layout: PageMaker and Quark XPress.

### 2.1.1 Approach of existing commercial software

Both PageMaker and Quark provide tremendous flexibility in what they can lay out, but this comes at the price of requiring very tedious manual placement and arrangement of all the graphic elements. There are ways of making sure things line up relative to each other, but for the most part these are 'guides' and 'rulers' — rather stric analogs to ways of lining things up in a traditional, non-computer, cut-and-paste layout This provides an important sense of continuity for layout professionals used to the traditional methods, and is a very intuitive interface in its own right.

### 2.1.2 Limitations of existing commercial software

However, the insistence on using a traditional cut-and-paste paradigm places severe restrictions on the utility of the program by prohibiting any type of collaborative approach. These programs are incapable of capturing a user's intentions. For instance, all relative positioning is done statically. That is, if one tells the machine to align two blocks of text in some given way, and then the sizes of the blocks change, there is no provision for retaining the specified relationship between the blocks.

Moreover, these programs do little to solve the issues of multi-user collaboration. Since newspapers must be created under strict time deadlines, it is desirable to do as much of the process in parallel as possible. Extensions to these programs, such as the Quark Publishing System, do provide centralized storage and access locking for sourc materials. This enables users to ensure that they are not destroying each other's work and facilitates the transfer of content from one person (or machine) to another. But it also serializes much of the workflow, as final layout cannot be executed until the content that it is based upon is "checked in." Moreover, none of these programs has any means of enabling a number of people to work on the same section of layout at the same time. In these programs, only completely isolated layouts, stored in separate files, can be modified concurrently.

## 2.2 Existing research software

**LILT** is based primarily upon two pieces of previous research: Greg Nelson's Juno (and more recently Juno-2) [2] drawing program, and the GLIDE network graph program.[3] Short descriptions of these programs and of how they relate to **LILT** follow.

---

[2] See Nelson and Heydon. "Juno-2 Constraint Based Drawing Editor"

### 2.2.1 Juno

Juno enables a user to create pictures by specifying control points and constraints that determine relative positions of these points. PostScript style drawing primitives can then be added to paint an image using these control points. Specifiable constraints can be non-linear, but there are severe restrictions: inequalities and discontinuous functions (like modulo, floor or ceiling) are not supported. The system keeps track of the layou information (points, constraints and drawing instructions) in a powerful description language. This language is hierarchical in nature, letting the user employ system-defined primitives to build larger constructs. These large constructs can themselves be used as primitives in even larger constructs.

The system has a two-pane user interface: one pane shows the currently rendered picture and the other shows the Juno-language program that produces it. The user can edit in either pane. This provides a means to manipulate directly the constraints in the system without relying on the visual representation. This two-pane interface is a powerful feature which **LILT** uses as a model in its three-pane interface. In Juno, the complexity of the constraint language makes the system difficult to use — it is a graphics package for a programmer, not an artist. Still, the language lets the user define macros (through Juno functions and procedures) enabling the re-cycling of graphic elements. Adding a similar feature to **LILT** would certainly enhance it.

Juno's constraint satisfaction algorithm is useful, but it can make counter-intuitive assignments, particularly if the "hints" the user provides to aid in these assignments are off the mark. This makes it harder for a user to anticipate what the machine will do when the drawing state is changed. But in general, the "hints" are sufficient to keep the constraint satisfaction engine from producing wildly unexpected results.

To summarize, Juno has limitations in its constraint satisfaction algorithm, and it user interface requires learning a cryptic programming language. But there are rea strengths to the system: it *does* use constraint satisfaction effectively to create attractive graphics. Also, its approach of creating a drawing framework is very extensible. The macro feature prevents this flexibility from coming at the price of an inability to describe high-level objects, and alleviates the tedium of constantly dealing with minutiae.

### 2.2.2 GLIDE

GLIDE is a program for drawing network graphs that are either directed or undirected. The system was created specifically with the goal of producing a more collaborative interface in this domain. It does this through a constraint satisfaction system based upon a mass-spring simulation. Each node in the graph is modeled as a mass in this simulation. Any of a set of constraints can be applied between these nodes in order to position them in the desired arrangement. Typical constraints include alignment ordering, clustering, equal-spacing, symmetry and creation of a tree-shape. Each of these constraints is represented in the system by a set of springs that cause the nodes to move into the appropriate arrangement.

---

[3] See Ryall et al. "An Interactive Constraint-Based System for Drawing Graphs"

The constraints are set up in a manner allowing the user to specify relationships among the nodes that are maintained through time. For instance, if the user states that space between two pairs of nodes should be equal and then moves one of the nodes at a later time, the other nodes will move to maintain the relationship. This persistence is a tremendous advantage when one is trying to align elements appropriately in a graph.

The system animates the movement of the nodes in real-time, which provides excellent feedback for the user. It is the spring-model itself that creates this feedback: the moving of the nodes in the graph by the model is due to a force generated by a process with a direct physical analogy, one that the user can easily intuit.

The collaborative approach that GLIDE uses has the very important consequence of decreasing the precision required in the user's mouse movements. Most commercia graph-drawing programs limit the level of required mouse-precision by using a grid tha objects will "snap to." Anyone who has used such a system knows that in practice grids do not work well: either they are too coarse to permit objects to be placed where the need to be, or so fine that they do not really lower the precision required to place them. GLIDE's approach is a tremendous improvement, in that it lets the user place the node far from the desired location, trusting that it will be moved to the appropriate place.

Most newspaper layout programs (such as Quark and Pagemaker) allow the user to specify guidelines which objects tend to snap to. Clearly this is a generalization of the grid approach that we see in mos commercial drawing programs. But these guides are subject to many of the same problems that afflict grids. First of all, they do not allow for maintaining relationships between elements dynamically. Also, they require very precise mouse movements to set up and to make an object snap to a specific guideline among several. **LILT** uses an approach similar to GLIDE in order to overcome this problem.

GLIDE lets the user specify the constraints--that is, the high-level conceptual look of the graph diagram--while it takes care of satisfying the details. A truly effective collaboration results, one that **LILT** strives to emulate in its own domain.

# Chapter 3: A new paradigm for single-user layout tools

Since our aim is to redesign the low-level details of the layout process by creating a set of collaborative semi-automatic tools, we need the ability to specify the layou constraints to the machine. Here, the work on GLIDE for semi-automated graph layou provides a good guide (see Ryall et al.). The types of structural relationships that can be specified in GLIDE are similar to those that are needed in the newspaper domain. However, newspapers are far more complex, since the individual elements (text, pictures or graphic elements like boxes) take up area, unlike the nodes of a graph. Further, these elements are chained together and have exceedingly complex interrelations that must be maintained. Moreover, newspapers have multiple pages, and content often must flow across more than one page. As mentioned above, the approach taken here divides the newspaper domain into two separate but related hierarchies--layout and content--each o which will now be discussed in detail.

## *3.1 The layout hierarchy*

The layout information in a newspaper seems naturally to form a tree structure, which is how it is represented in **LILT**. There are three different types of elements in the tree: **Node** elements, **Constraint** elements and **Update** elements. **Node** elements are, for the most part, physical layout objects that have a visual form. **Constraint** and **Update** elements both represent entities that can modify these **Node** elements (that is, they act as constraints in the more general sense). The distinction between **Constraint** and **Update** elements is transparent to the user, and is a subject to which we shall return in Chapter 6: Algorithms.

### 3.1.1 A description of the node elements

In this section we will examine the fundamental part of the layout hierarchy: the **Node** elements in the tree. **Nodes** have five subtypes: **Papers**, **Pages**, **Blocks**, **Edges** and **Clusters**. **Node** elements are either visual objects, such as **Pages**, **Blocks** or **Edges**, or they are elements that represent a grouping of these objects called **Clusters**. All these different subtypes o **Node** elements share certain attributes in common, as described in section 6.2.1.4. All but **Edges** (which are always leaves) maintain three separate lists o children, one for each of the three element types, **Nodes**, **Constraints** and **Updates**. We will only deal wit **Node** child-lists here, leaving the discussion o **Constraints** for section 3.1.2. The discussion of **Updates** is introduced in section 3.1.2 and dealt with extensively in section 6.2.2.

Since the **Node** elements define the structure of the tree, it is worthwhile examining each **Node** subtype in detail

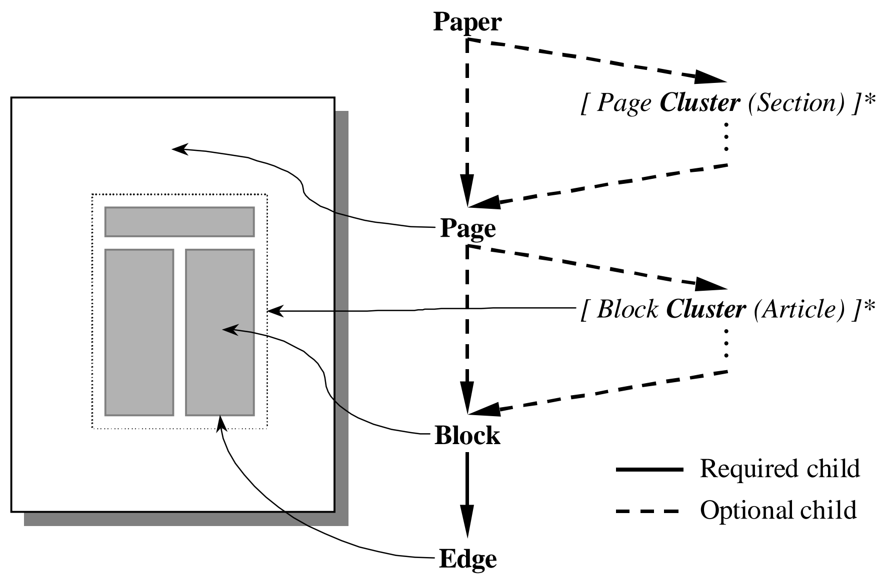


**Figure 5: The Node hierarchy**

Figure 5 shows all of the **Node** subtypes within the hierarchical structure they form. The root of the layout tree is always a node of subtype **Paper**, which contains all of the layou information. Only two **Node** subtypes are permitted in the **Node** child-list of a **Paper Node**: **Page Nodes** and **Page-Cluster Nodes**. The **Paper Node** maintains the correct page numbering for all of the **Page Nodes** among its children. The **Page Nodes** contain information about a page in the paper such as its width and height. The **Page-Cluster Nodes** have **Node** child-lists that contain only other **Page-Cluster Nodes** or **Page Nodes**. We can think o these **Page-Cluster** as **Sections** within the paper, though they are allowed to nest beyond one level for more complex organization. For example, one could have a hierarchy with two nested **Cluster** layers such as: **Paper** → **Cluster** → **Cluster** → **Page** to represent a structure like book → chapter → section → page. Thus a paper is made up of sections, and these sections are made up of either other sections or pages.

Eac **Page Node** contains the elements to be laid out on that page, and follows a similar pattern to that just described for the **Paper Node**. **Page Node** can only have **Block-Clusters** or **Blocks** in its **Node** child-list. These **Block-Clusters** are differen from the **Page-Cluster** described above in that they can only have **Blocks** (or other **Block-Clusters**) in their **Node** child-lists.

**Blocks** are the fundamental units of the **Node** hierarchy. They are rectangular areas containing text or graphics and are the elements within this framework that most closely resemble the cut-and-paste text chunks that we see in Quark and PageMaker. As in these programs, **Blocks** have a position on the page, a width and a height. However, in contrast to what happens in these programs, the **Blocks** themselves do not mediate this information. Each **Block** is defined to have an **Edge Node** for its top, bottom, right and left sides. These four **Edge Nodes** indirectly specify this position information. Each **Edge** has a single, one-dimensional position on the page, either vertical or horizontal.

In order to use the hierarchy just described, it is necessary to be able to impose a complex set of constraints on various **Node** elements; these constraints are described below.

### 3.1.2 A description of the layout constraints

When thinking about specifying layout constraints, it is important to consider the layout as a dynamic entity. Only by ensuring that solutions to all the constraints are constantly being sought will the system be able to react to changing data in such a wa that our secondary goal of multi-user editing/layout can be attained. We can divide the layout constraints roughly into low-level, mid-level and high-level constraints. The low-level constraints are largely syntactic in nature and, while configurable, will usually be maintained by the machine automatically. The mid-level constraints have defaults letting typical configurations be made quickly, while allowing fine control when necessary Finally, the high-level constraints will need to be extensively set up by the user and then rendered by the machine.

All constraints present in the system are represented either by **Constraint** elements or by **Update** elements. These elements act upon some subset of the other elements in the layout hierarchy. In this system there are two different ways tha constraints can be configured to determine the set of elements that they affect: an explicit and an implicit set definition.

Using an explicit definition, the user simply specifies all the elements the constraint should act upon, and the constraint maintains this list. In this case, the constraint will reside on the appropriate child-list (**Constraint** or **Update**) of the **Node** that is the common ancestor of all of the **Nodes** in this explicit list, unless this lis contains only one node, in which case the constraint will hang from that **Node's** parent. Because of this rule, the constraint is always the child of an ancestor for every node that i controls.

The implicit definition is based upon the following domain-specific heuristic: very often constraints will be applied to all of the**Node**-type children of a given **Node** element, and to no others. As a consequence, constraints can be assigned to hang on the appropriate child-list of a given**Node** *n,* and then to act on all of the **Node** elements that are children o *n*. When this approach is used, the constraint will begin to act on any Node that is added as a child to the node, without an explicit command to do so.

Let us now consider each of the different kinds of constraints in turn.

#### *3.1.2.1 Low-level layout constraints and syntactic requirements*

There are only three low-level constraints. The lowest of these is the syntactic condition that all the content must remain on the page. The system must also maintain a second, related constraint that no two **Blocks** on a **Page** may overlap. Fundamentally, this is a syntactic constraint as well, but it is also configurable: each **Block** has its own margin settings. Provision is also made to allow the user to specify that certain elements can, or should, overlap. For instance, there are times when text needs to flow over a picture or shaded box. The last low-level constraint is the option to anchor certain elements in place, preventing the system from modifying them (note that this can be a any level of the layout hierarchy). Overuse of this constraint will decrease the

effectiveness of the system, but this user-specified constraint can still be useful in man cases.

### 3.1.2.2 Mid-level layout constraints

The system must also be able to organize the relationship among the elements close to the leaves of the layout hierarchy forming the complex structures mediated b elements closer to the base of the tree. The mid-level constraints are responsible for establishing this relationship. As such, they act only on **Block-Clusters** and **Blocks**, and as a result hang from either **Pages** or other **Block-Clusters**. In some sense we can think of these constraints as rules that **Node** elements high in the tree use to control the placement of and relationships between their children. The mid-level constraints include:

- Controlling the width of all the attached **Nodes**.
- Controlling the height of all the attached **Nodes**.
- Horizontally aligning all attached **Nodes**, referring to each **Node** as a whole, or to its right- or left-most edges
- Vertically aligning all attached **Nodes**, referring to each **Node** as a whole, or to it top or bottom-most edges

When combined, these constraints are sufficient to group the **Blocks** and create complex layouts. But there is a critical piece missing: the layout of newspapers revolves around the information (i.e. the text and pictures) that needs to be laid out, and our system needs to understand this structure (and in particular it needs to know the amoun of space necessary to render this information), a topic to which we will return in section 3.2, The content hierarchy.

### 3.1.2.3 High-level constraints

There are many high-level constraints that would be advantageous to implement in this domain. Implementing them is beyond the scope of this project; but the current work does provide a sound foundation for future efforts in this area. Fundamentally, high level constraints require a tight integration between the two hierarchies: they consist of imposing layout constraints that use the content hierarchy to dictate aesthetic alterations in the layout hierarchy. Often the analysis of the content hierarchy will have to occur at a very high level, possibly even requiring natural language processing and, certainly, importance ranking. The following is a list of some of the most important constraints that fall into this category and would be excellent additions to this paradigm:

- Continuations should be near the top of a page.
- There should be more text in the first and last pages of a section and more ads in the center.
- Ads usually should be at the bottom of the page.
- Smaller ads should come first.
- Conflicts between ads should be avoided (i.e., two ads for the same product or service should not be on the same page).
- Allocation of spare white space should be improved.

- Placement of the layou **Node** elements should be according to the relative importance of their related elements in the content hierarchy (described in section 3.2). If in the content articles are given a rank of importance within sections, then the computer could place articles in different locations based on this information. More specifically, **Page** elements containing the **Tracks** of important **Bundles** (sections) would gravitate toward the front of the **Paper**, while on individual **Pages** important stories would gravitate toward the top right and unimportant stories would move down and to the left.
- The corollary of this constraint: ordering articles to ensure optimum packing.

These last constraints warrant some additional comments. While white space is handled well already by individualized margins on **Block** elements and by the ability to space **Blocks** and **Clusters**, there are significant improvements that could be made. I would be helpful if a constraint could be applied that would allocate the white space according to a heuristic that took some Gestalt approach, but still allowed the user to override its decisions. Further, the machine might want to request to insert a callout to take up additional space if it deems it necessary

There is also the question of figuring out the best overall order for stories, according to importance and space considerations. Ideally, the machine could provide suggestions as to possible article ordering and jump locations in order to optimize the paper's look and length, correlating this with the contrasting goal of keeping related content together and important stories in certain locations.

## *3.2 The content hierarchy*

The content information also forms a hierarchy structure: each paper is comprised of **Articles** (which for practical purposes could also be ads, banners or isolated graphics). Eac **Article** is comprised of a set of distinct pieces of information, or **Tracks**. A typical article might have the following **Tracks**, but there are several others that might appear as well:

- Title
- Byline
- Body Text
- Graphic
- Caption
- Jump (continuation) text
- Continuation Title

Eac **Track** contains a specific amount of information that requires a certain amount of space to lay out. Most of these **Tracks** are laid out within a single block, bu some of them, notably Body Text, will almost certainly exist in multiple blocks that need to be chained together.

**Articles** can be grouped together further to form sections, chapters or other divisions. Articles are then a special case of a more general type, a **Bundle**, that can contain either other **Bundles** or **Tracks**. Thus we have a tree structure with two types.

The root of the tree is a **Bundle** that contains all of the content in the paper. Thi **Bundle** contains an arbitrarily deep tree of other**Bundles** (often two layers deep: sections and **Articles**). These **Bundles** ultimately contain **Tracks** that house the constraint information itself.

### 3.3 Relating the two hierarchies: duct constraints

We have a very complicated interrelationship between the structure of the conten and the structure of the layout. Fundamentally, we have two hierarchical structures describing the same data. From root to leaf they are, respectively:

*Content:* **Bundle → …→ Track**

*Layout:* **Paper → Page-Cluster → … → Page → Block-Cluster → … → Block → Edge**

However, there is some overlap in the two structures. **Tracks** will be laid out using a set of **Blocks**. There needs to be a means of specifying what content data should flow into each of the **Blocks** in the layout. Because this involves the "flowing" of data from the content hierarchy into the layout hierarchy, these constraints are called **Ducts**. **Ducts** are mid-level syntactic constraints, and they must mediate another important factor as well: the **Blocks** assigned to lay out a given **Track** must have the exact amount of area necessary to lay out the material in this **Track**. Because of this control over layou **Nodes**, **Ducts**, like the other constraints, are clearly part of the layout tree. However, we can also see **Ducts** as being part of the content hierarchy, as they in effect divide **Tracks** into the pieces that are placed in the various **Blocks**. Although **Ducts** do belong to bot trees to this extent, practical considerations argue for **Ducts** hanging from the layout tree alone just like the other constraints. Still, each**Track** in the content tree remembers which **Ducts** are mediating its data flow in order to properly supervise this flow.

# Chapter 4: The LILT user interface

The user interface in **LILT** attempts to use the strongest features of Juno-2 and GLIDE and to supplement these with some new and powerful elements. The following screenshot is a typical view of the system

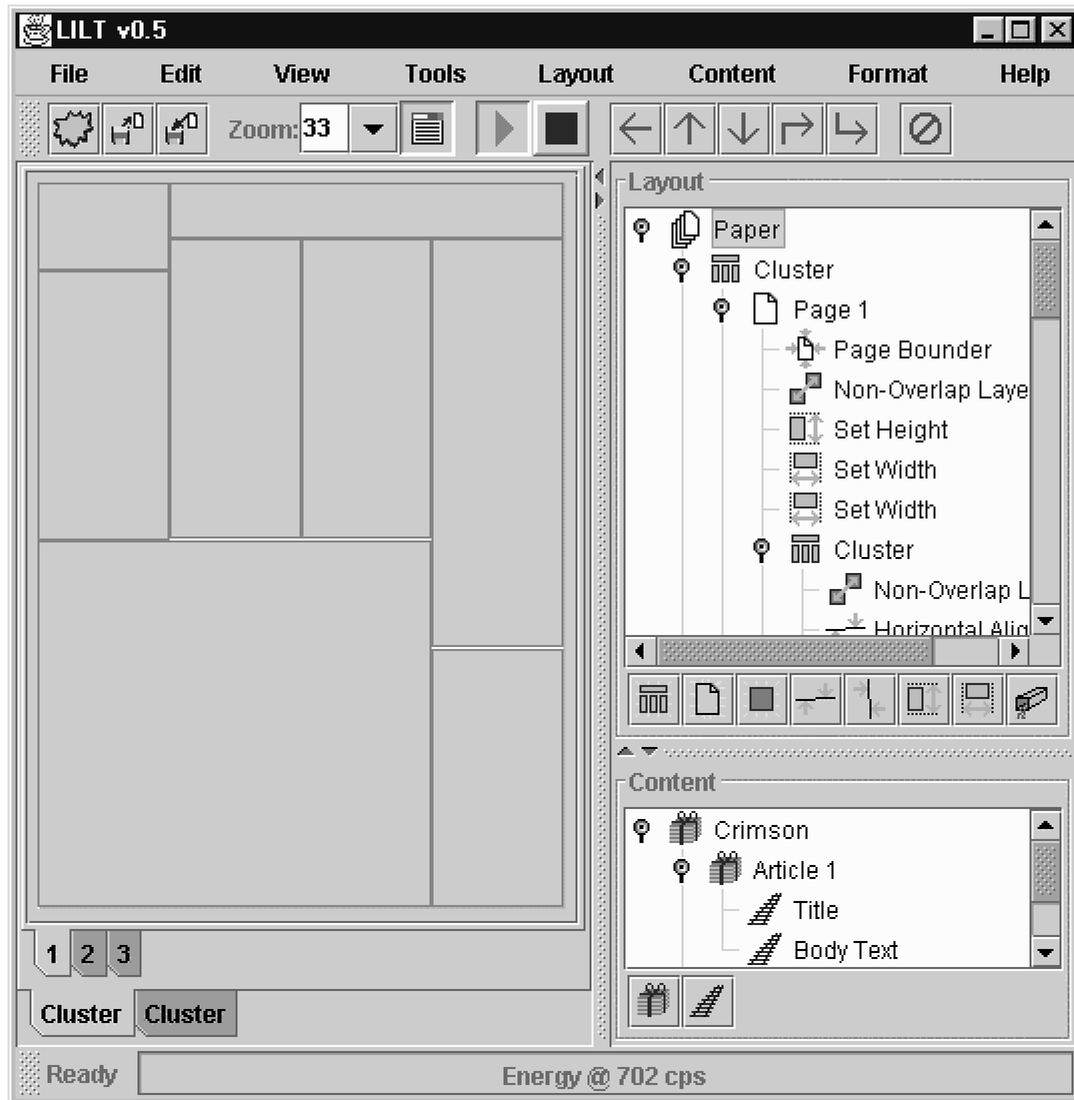


**Figure 6: A typical view of the system**

Like the other programs discussed, **LILT** has a WYSIWYG (What You See Is What You Get) interface that displays the current layout. This resides in the panel tha takes up the left portion of the screen. Like GLIDE, **LILT** provides buttons for adding objects to the system, but it places these buttons on two separate toolbars (right side): one for layout elements and the other for content elements. It provides a third toolbar (top) with many useful controls, including the button for deleting elements from the system and the button for bringing up **LILT's** "Properties" window.

**LILT's** "Properties" window is separate from the main GUI and shows details about the currently selected element, letting the user modify these properties. For

instance, when an **Edge Node** is selected, the window enables the editing of the **Edge Node's** two properties: whether or not it is **Anchor** , and its current position:

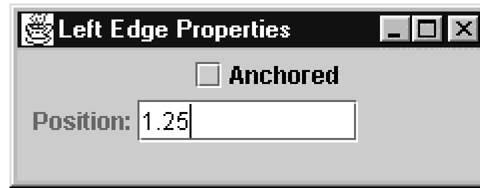


**Figure 7: The "Properties" window with an Edge selected**

Other elements in the system have different and much more complicated sets o properties that appear in this window when they are selected.

Juno-2 uses a dual-view interface to let the user more effectively manipulate the model data that represent the picture being drawn. One view is the WYSIWYG view o the picture itself, which does not show the constraints involved. The second is a text control that lets the user see, and explicitly modify, the Juno-2 Language program that produces this picture. The user can modify either view, while the other one refreshes to reflect these changes. **LILT** also uses the document-view user-interface model to allow multiple views of the same data (in this case, information about a newspaper). In addition to the WYSIWYG pane on the left **LILT** has the two tree controls on the right Each presents a means for editing its portion of the newspaper model: the layout and content hierarchies respectively. By manipulating the elements in these trees, the user can change the underlying model and watch these changes reflected in the WYSIWYG view. Conversely, changes in the WYSIWYG view will cause the tree-views to refresh. The tree controls provide an intuitive means of visualizing not only both the layout and content data, but also the constraints themselves, as the constraint language **LILT** uses (described in section 3.1.2) inserts these constraints into the hierarchy. The panel at the bottom of the screen displays status information. On the left, a string provides feedback to the user about the state of the system, including changes to either of the hierarchies and any errors that occur. On the right, a meter shows the amount of energy in the mass-spring-update model (discussed in section 6.1). Inside the meter, a string is displayed that indicates the number of "cps" or "cycles per second" the model is performing, a measure of how fast the simulation will converge on a solution to the constraint problem.

# Chapter 5: An example interaction with LILT

In order to better understand the system, it is useful to examine an extended and detailed example of a typical interaction within it. The goal here is not to specify every aspect of the system, but rather to show what a typical “hands-on” experience with the program is like, and how it behaves as the user interacts with it.

Let us suppose that a user has just started up the system and wants to begin tolay out a new paper. In order to produce the paper, the user will need to add new elements to both of the hierarchies in the system: layout and content. Let us assume the user adds to the layout hierarchy first. In this case, the user selects the root element of the layout hierarchy (a **Paper** node), and clicks the "new page" button in the toolbar below the layout-tree (second from the left. See Figure 8, below). This introduces not only a new page into the layout but also adds both a page-bounding and a non-overlap constraint below the new **Page Node**. Because these constraints are defined with the implicit attachment rule, they will affect any other **Nodes** that are added beneath the newly added Page. This placement for these constraints is consistent with a rule that **LILT** imposes: all constraints live in the hierarchy below the common ancestor of all the **Nodes** they affect. Usually this has the result of a given constraint being a sibling of the **Nodes** it affects.

After adding the **Page Node**, the user proceeds to add a **Cluster** element to the new **Page**, by clicking on "Page 1" in the layout tree, and then clicking on the "New Cluster" button in the toolbar below the layout tree. This adds the new **Cluster Node**, and its associated non-overlap constraint. In a similar manner, the user adds a **Block Node** and a **Cluster Node** below this new **Cluster Node**, and then adds two **Block Nodes** beneath this second **Cluster Node**. After all of this is done, the system will look as follows:

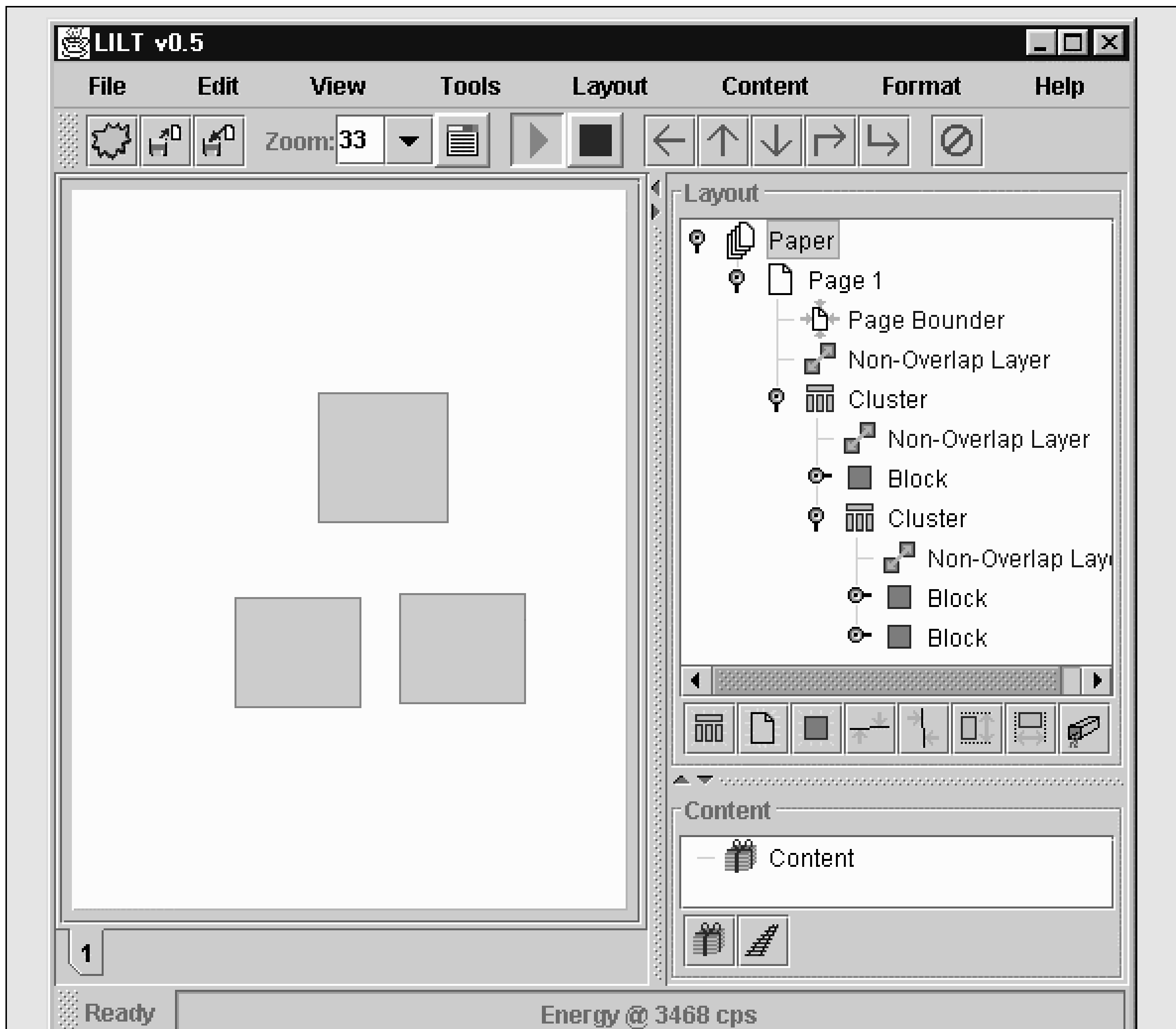


**Figure 8: Step 1**

The next step in creating the layout is to specify constraints between these nodes that enforce a desirable arrangement. In order to do this, the user first selects the nodes in the hierarchy that he or she wants a new constraint to apply to, and then clicks the button in the layout-toolbar corresponding to the new constraint type that he or she wants. This will create the new constraint and place it at the appropriate place in the hierarchy. The user may then want to tweak the parameters of this constraint (if it has any). This is done by selecting the constraint in question and then using the "Properties" window to change these parameters. For example, by default, width constraints try to make all of the **Nodes** they apply to have a fixed, absolute width. But, often a situation may instead call for a width constraint that attempts to make all of its attached Nodes have the *same* width, a change we see illustrated below:

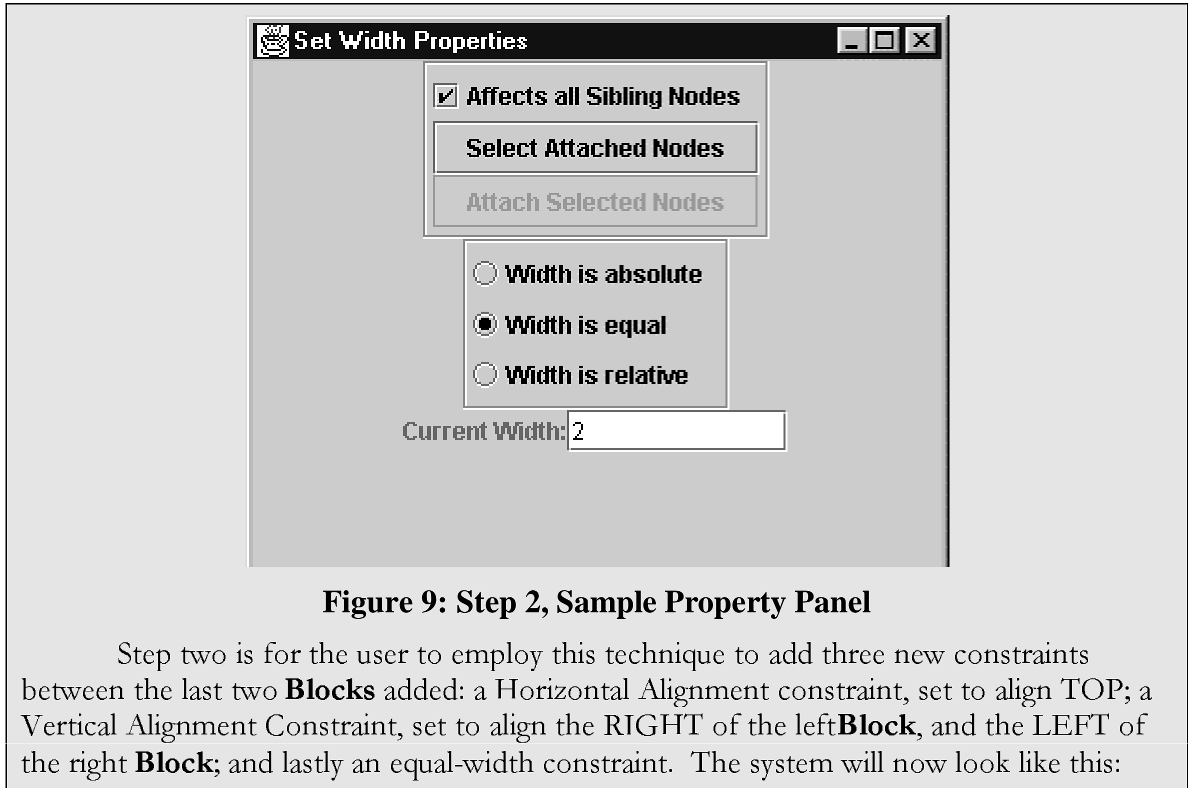


**Figure 9: Step 2, Sample Property Panel**

Step two is for the user to employ this technique to add three new constraints between the last two **Blocks** added: a Horizontal Alignment constraint, set to align TOP; a Vertical Alignment Constraint, set to align the RIGHT of the left**Block**, and the LEFT of the right **Block**; and lastly an equal-width constraint. The system will now look like this:

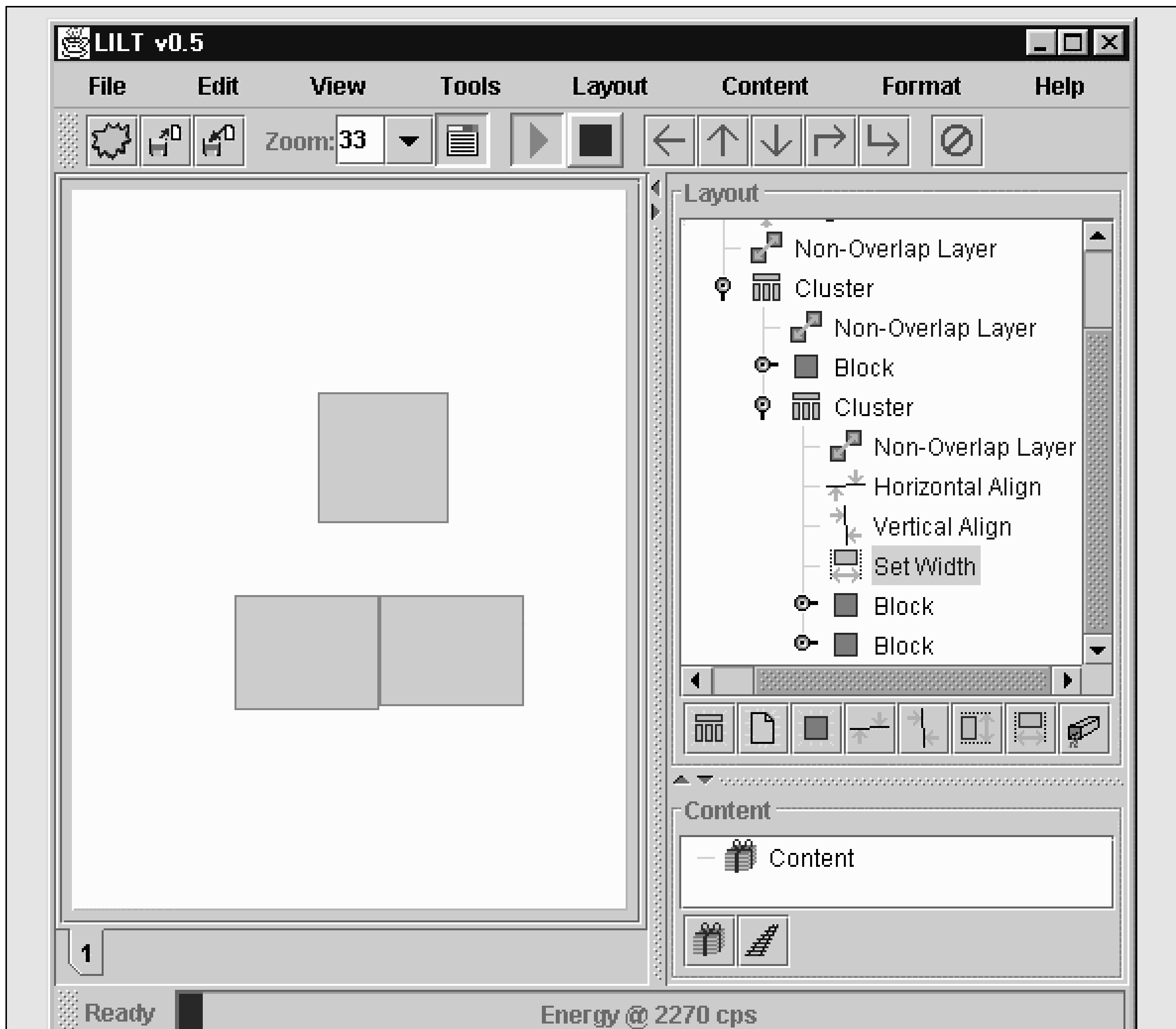


**Figure 10: Step 2**

The third step is for the user to apply a set of constraints between the remaining **Block** and the low-level **Cluster** containing the **Nodes** and **Constraints** described in step 2. By using the same technique as used in step 2, three additional constraints are added: a Vertical Alignment constraint that aligns the **Block** and sub-**Cluster**'s LEFT sides; another equal-width constraint; and finally a Horizontal Alignment constraint between the bottom of the **Block** and the top of the sub-**Cluster**. After these constraints are added, the system looks as follows:

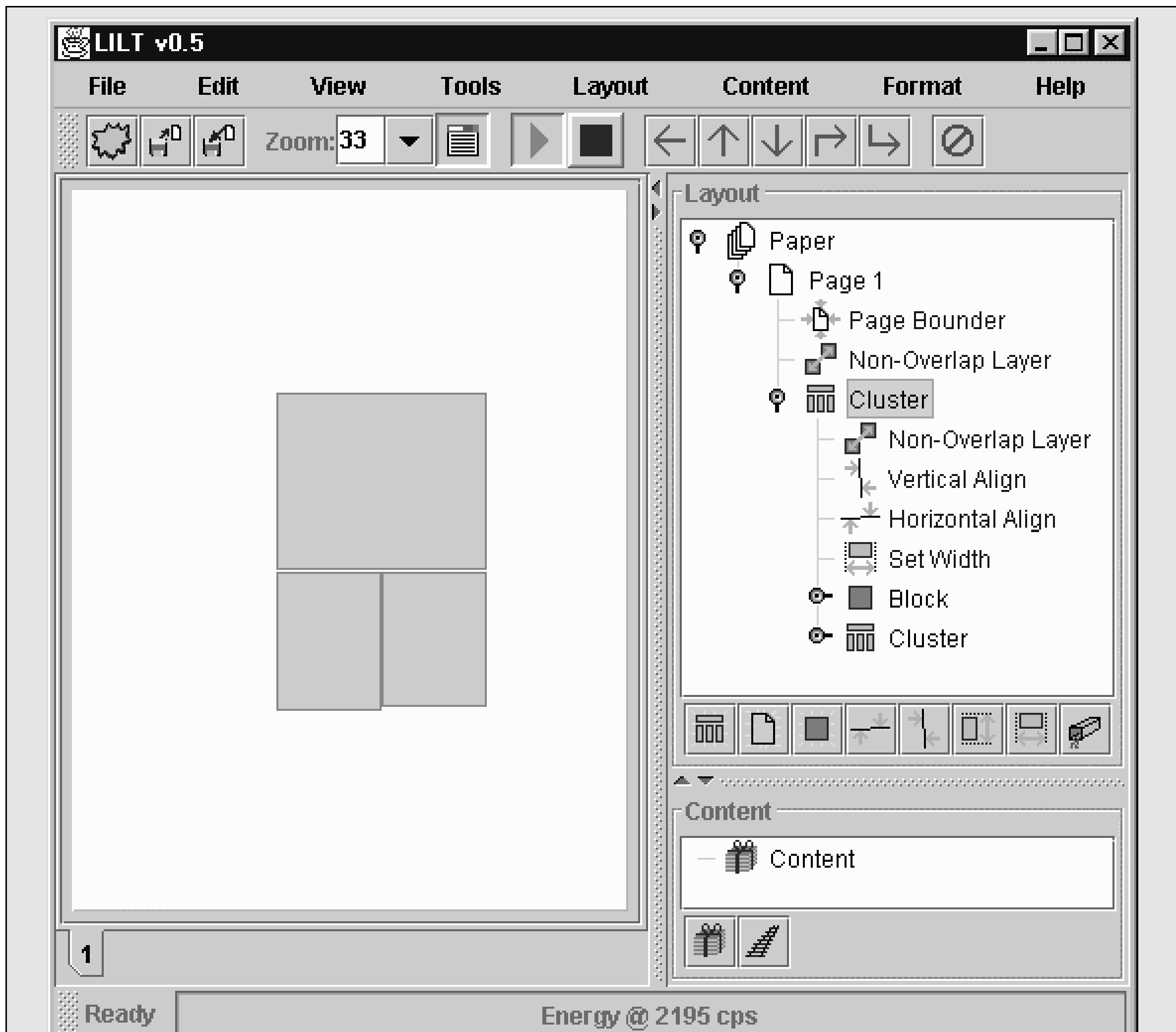


**Figure 11: Step 3**

For each of the alignment constraints that act on the different **Edges** on each of its attached **Nodes**, it is necessary to specify explicitly the affected **Edge** for each **Node**. In order to accomplish this, the user first selects the constraint in question, which loads it into the "Properties" window, which we can see below in Figure 12. Next one needs to ensure that the constraint is being applied to a specific set of nodes, rather than all of the **Nodes** that are its siblings in the hierarchy (in other words that it uses an explicit attachment definition). This is done by deselecting the check-box at the top of the "Properties" window. It is then possible to select individually each of the attached **Nodes** in the list box at the bottom of the window, and then to select the corresponding **Edge** that should be affected by choosing among the radio buttons in the middle.

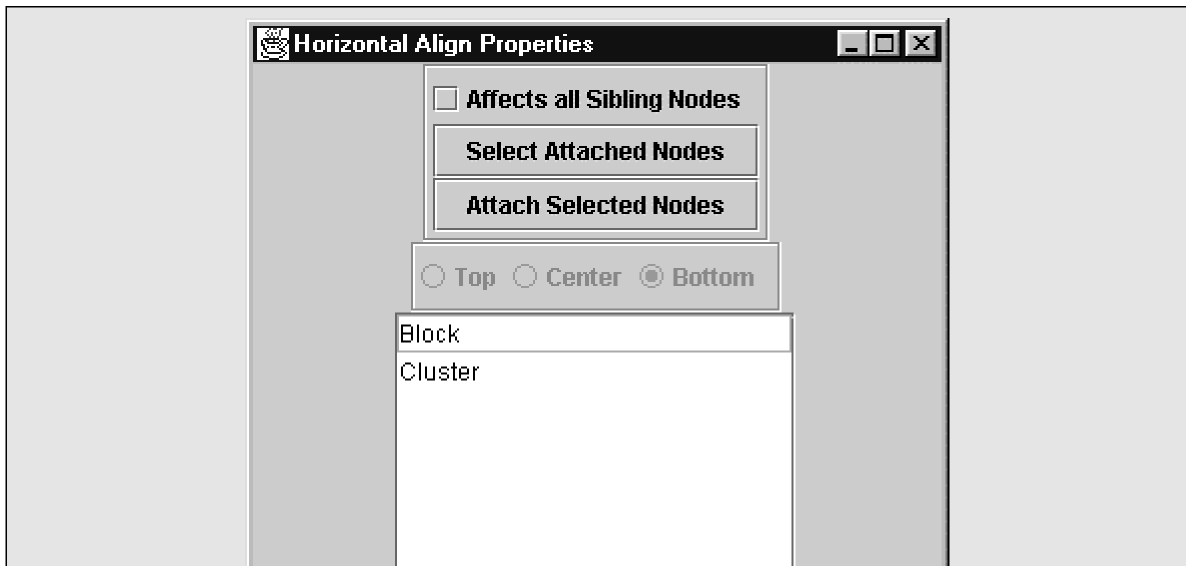


**Figure 12: Step 3, Sample Property Panel**

The fourth step is to provide a constraint for the width of our entire high-level **Cluster** (remember that both the width constraints specified above were of the "equal" type, and thus the actual width of all of the blocks is still unconstrained -- only their widths with respect to each other are constrained). This new constraint could be added in exactly the same way, but it is useful to note that **LILT** supports an important shortcut for adding width and height constraints. First, the user selects the **Node** that the new constraint should control, in this case the **Cluster** in question. Next, the user clicks (likely a right click, which will not affect the selection) and drags the **Edge** of the **Node** that he or she wants to constrain. In this case, this should be either of the right-most or left-most **Edges**. This will cause **LILT** to determine if there already exists an absolute width constraint that applies to this node, and add a new one if there is not. The drag motion is then used to set the size parameter of this constraint. After completing this process, the system should look like this:

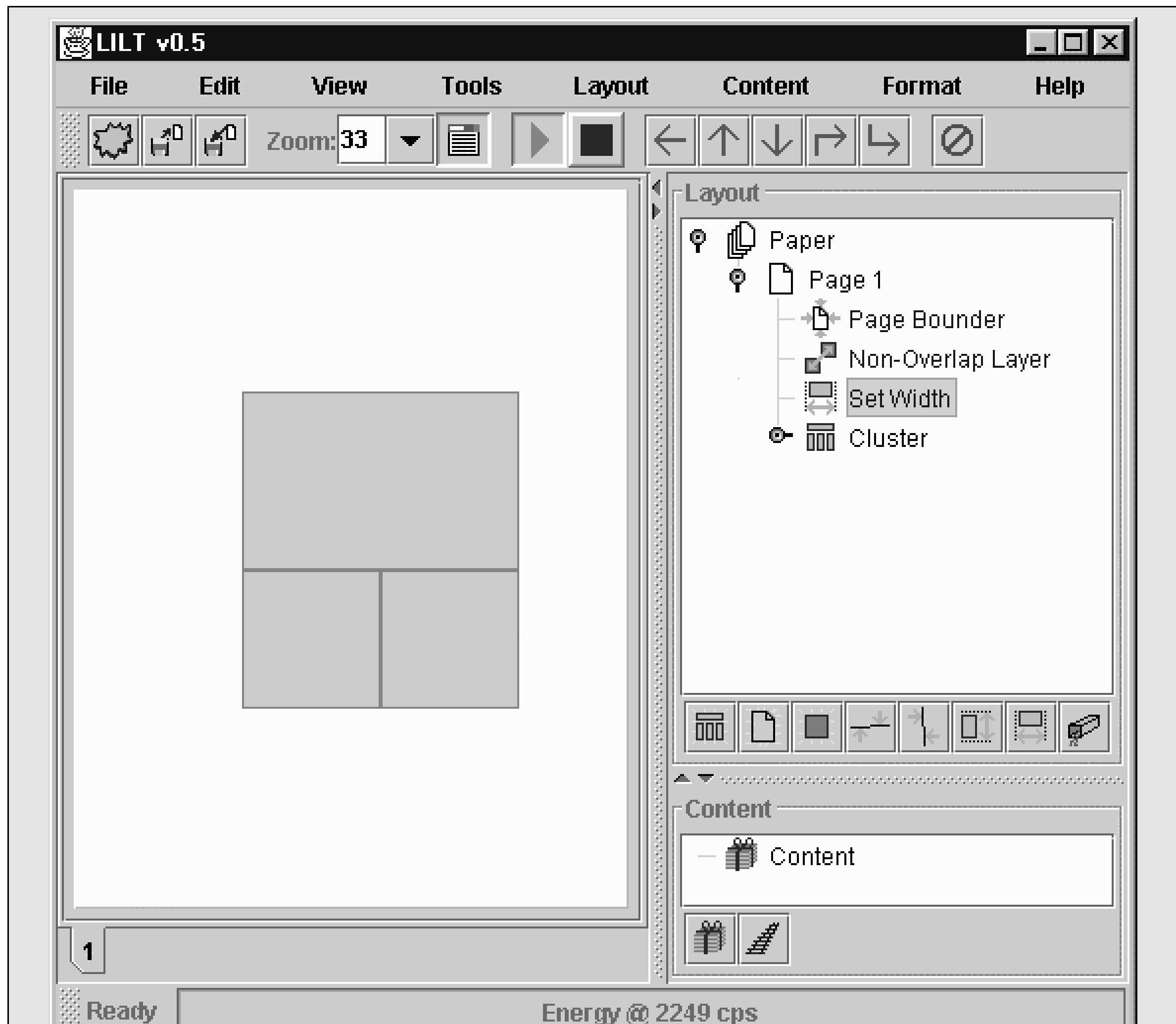


**Figure 13: Step 4**

After step four, the user has produced an attractive layout for three blocks, in a style that is typically used for a single **Article's** worth of text. This structure is so typical that it is useful to coin a term for it: a **Clump**. The next step in this example is to add some content to this document, which can then be directed in a later step into the **Clump** that has just been added.

Adding content elements to the content hierarchy is very similar to adding layout elements to the layout hierarchy. By selecting an element already present in the tree, and then clicking either the "add Bundle" or "add Track" buttons on the bottom right of the screen, the user can build up an arbitrary hierarchy. In our example, the user has added a **Bundle** (an element that contains other **Bundles** and/or **Tracks**) for a new lead story. Then he or she has added two **Tracks** (the basic element of the content tree which represents a linear flow of information, be it text or a graphic) beneath this **Bundle**. After these additions, each element is selected in turn and its parameters are edited in the "Properties" window.

The **Track** properties look like this:

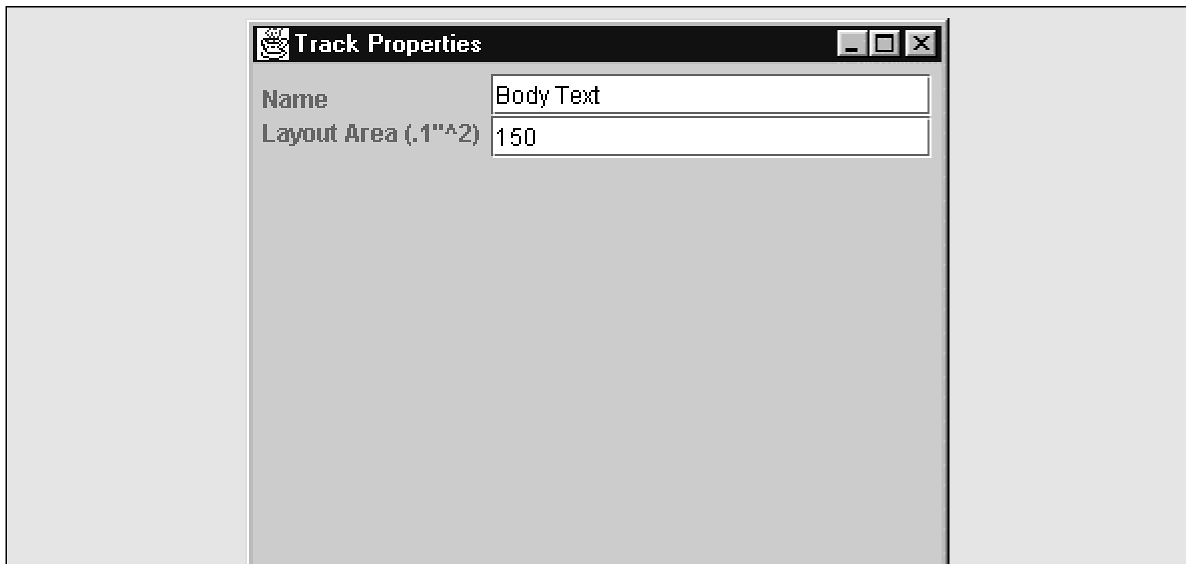


**Figure 14: Step 5, Sample Property Panel**

On this panel, there is a space to edit the name of the **Track** and to specify the amount of content this **Track** has. In a real working system, this second parameter would instead be the actual content to display (or some reference to it, like a filename). It is worth noting that this value is integral, which stems from the fact that most content is of an integral nature -- there are only a fixed number of characters in a given story, for example. **LILT** currently makes the assumption that each additional character takes exactly .1 more square inches to lay out. But the exact size is not as important as that the algorithms used within the system be capable of dealing with the granularity these integral values impose.

After adding the content and editing its parameters the system should look like this:

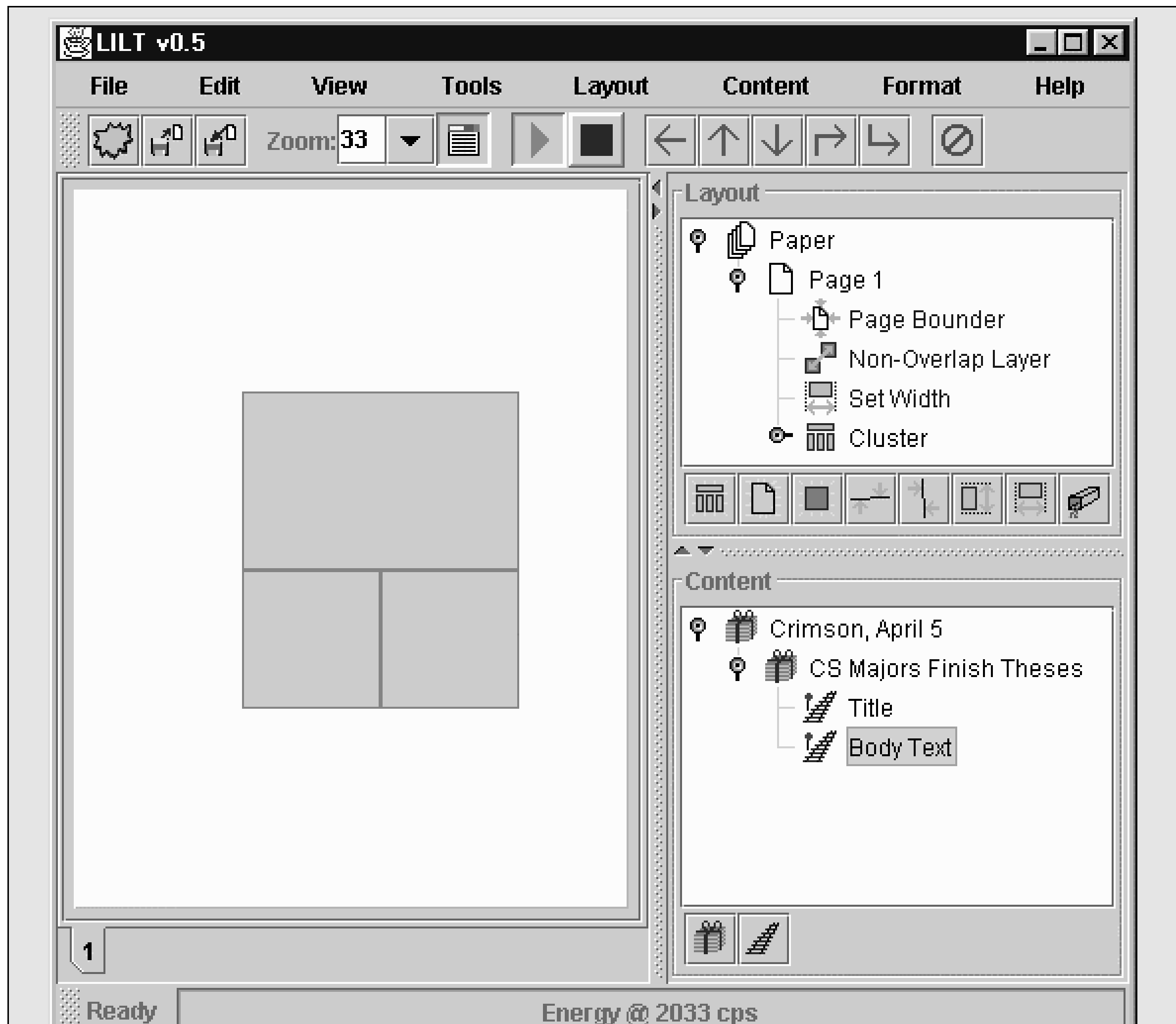


**Figure 15: Step 5**

The 6th step in the example is to connect the **Article's** amount of content that was added in step 5 to the **Clump** that was created in the first 4 steps. In order to do this, each block that is going to receive content needs to have a **Duct** constraint applied to it. In this example, two **Ducts** are added, one for the **Block** on top to receive the title content, and one for the other two **Blocks** to receive the **Body Text**. These constraints are added just like the others: by selecting the nodes they should be attached to, and then clicking the "add Duct" button. After the selecting of a newly added **Duct**, the "Properties" window looks like this:

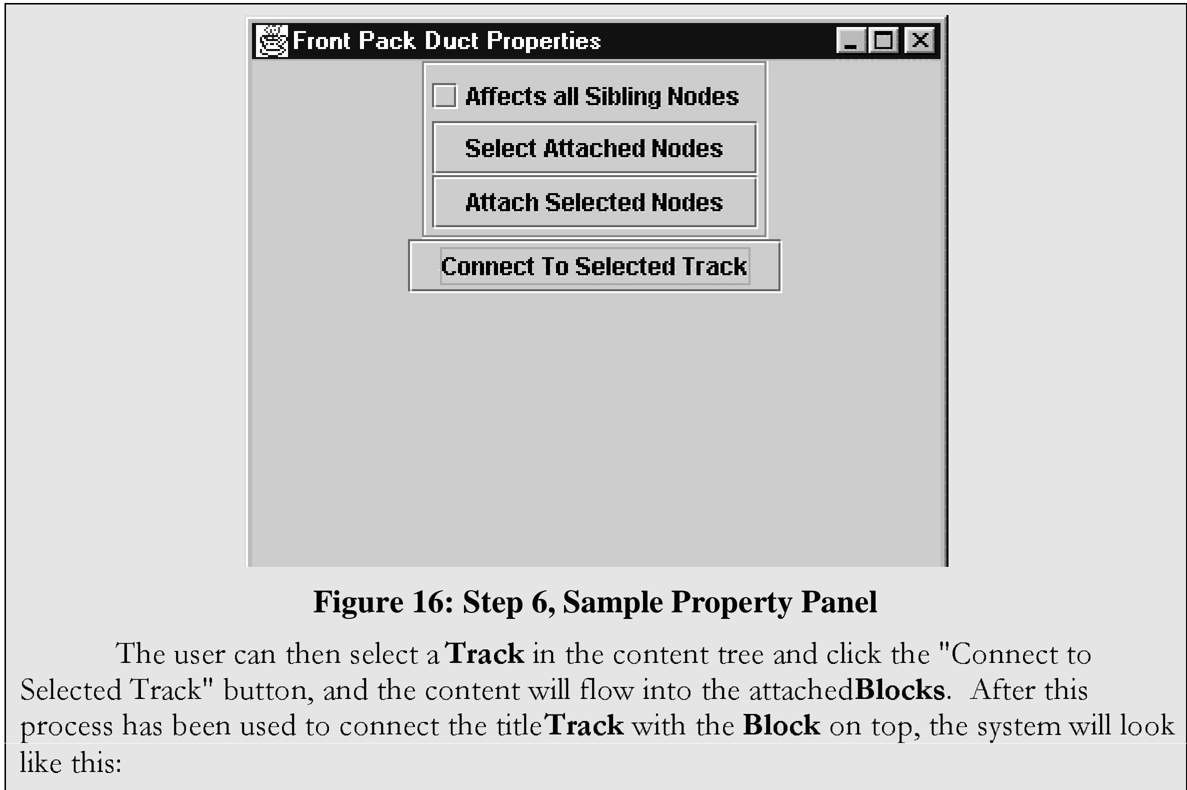


**Figure 16: Step 6, Sample Property Panel**

The user can then select a **Track** in the content tree and click the "Connect to Selected Track" button, and the content will flow into the attached **Blocks**. After this process has been used to connect the title **Track** with the **Block** on top, the system will look like this:

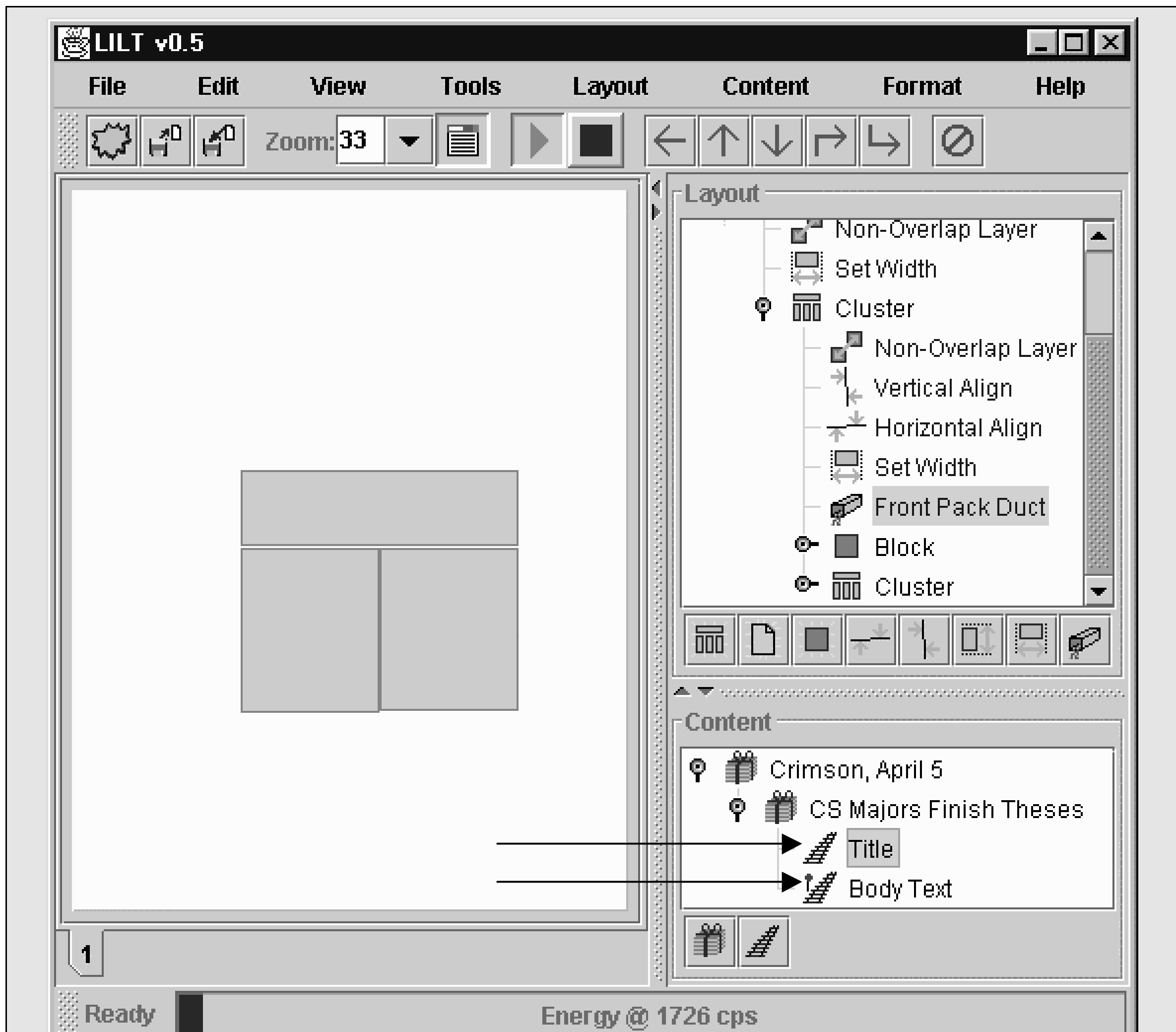


**Figure 17: Step 6**

Note that the little red stop sign on the icon for the title**Track** in the content tree has disappeared (as shown by the upper arrow), indicating that the content for this track has been laid out, and that the system was able to find enough space to lay out all the content. The stop sign is still present in the Body Text Track (as shown by the lower arrow), as a **Duct** has not yet been added for it as of figure 17.

The 7th step of the example is to reposition our**Clump** to its appropriate place on the page. Selecting the **Clump** and dragging it with a right mouse click does this (again, a right click does not change the selection state). The following figure shows the repositioned **Clump** and an additional block that is **Anchored** in place in order to simulate additional content that would be placed on the page:

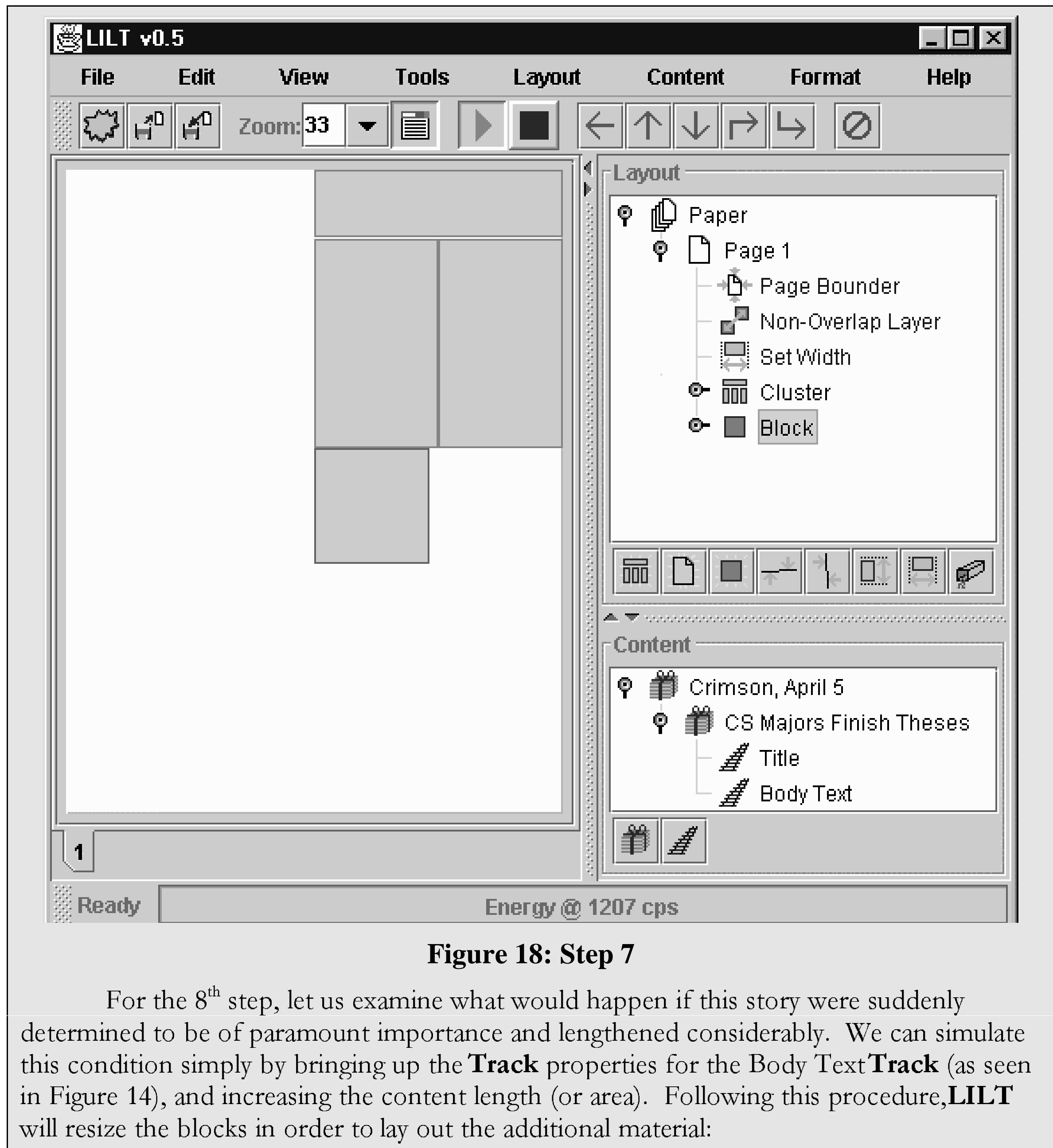


**Figure 18: Step 7**

For the 8th step, let us examine what would happen if this story were suddenly determined to be of paramount importance and lengthened considerably. We can simulate this condition simply by bringing up the **Track** properties for the Body Text **Track** (as seen in Figure 14), and increasing the content length (or area). Following this procedure, **LILT** will resize the blocks in order to lay out the additional material:

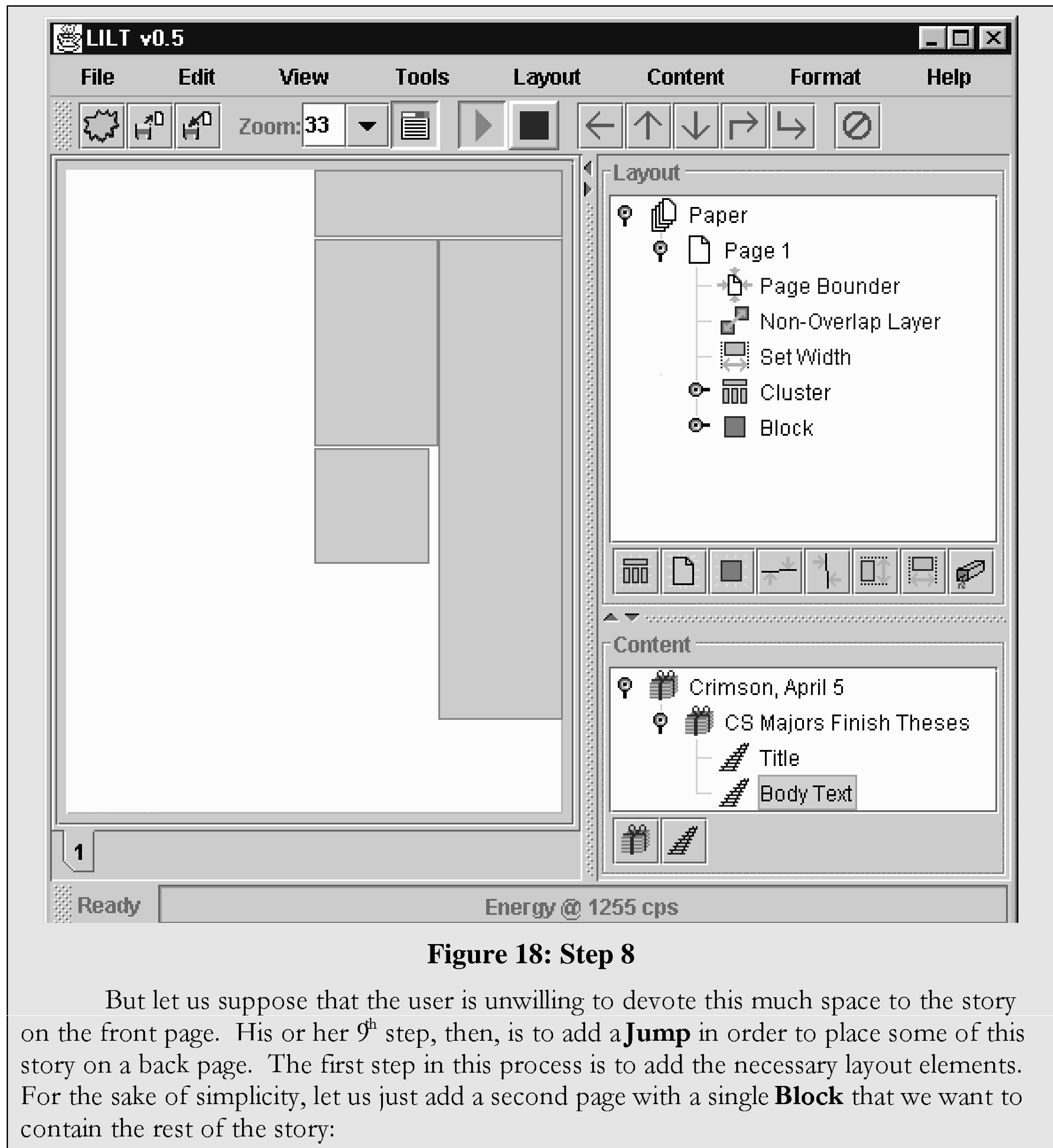


**Figure 18: Step 8**

But let us suppose that the user is unwilling to devote this much space to the story on the front page. His or her 9th step, then, is to add a **Jump** in order to place some of this story on a back page. The first step in this process is to add the necessary layout elements. For the sake of simplicity, let us just add a second page with a single **Block** that we want to contain the rest of the story:

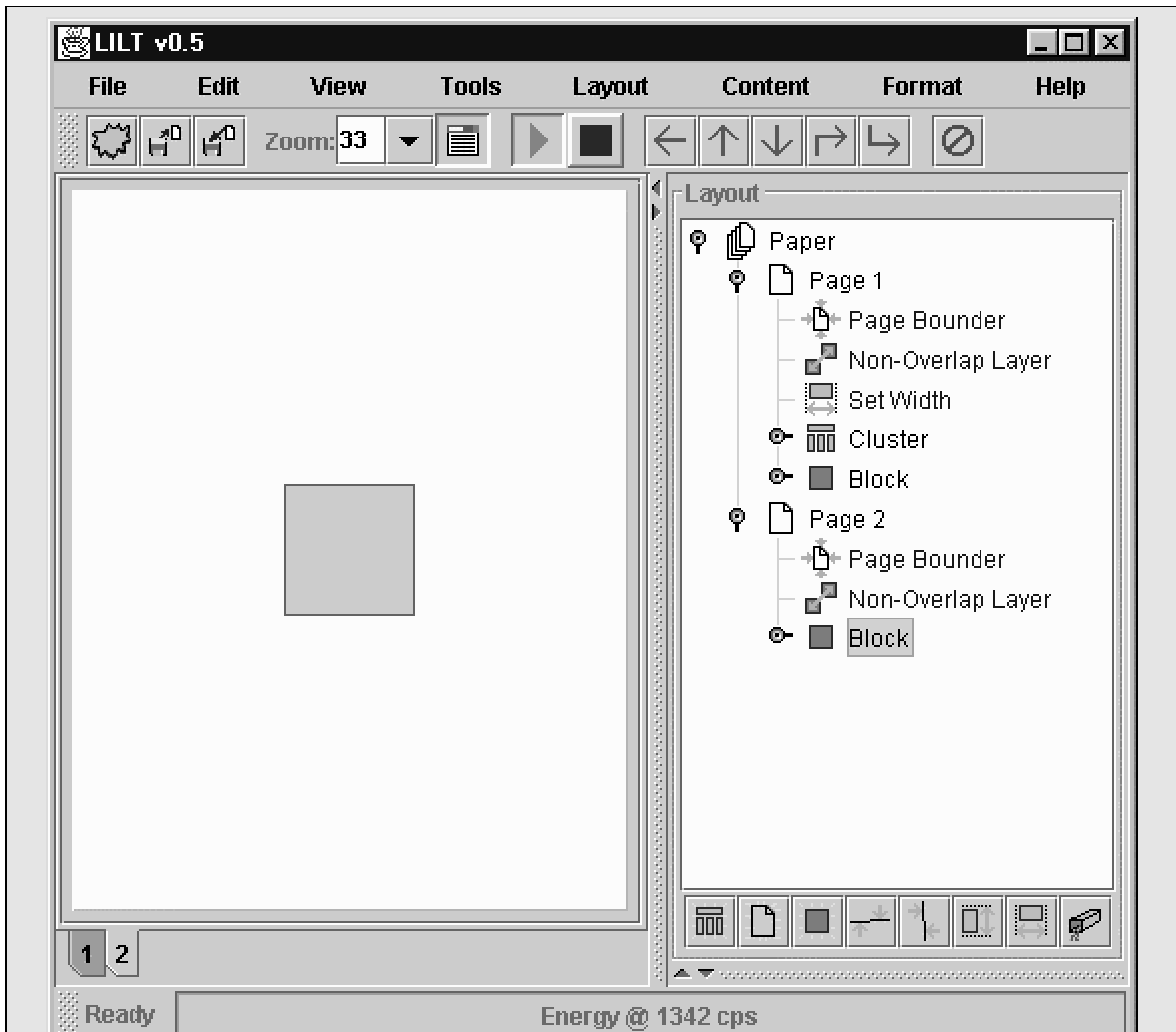


**Figure 19: Step 9**

Consequently, the 10th step is to redefine which **Nodes** are attached to the **Duct** that is mediating the layout of the **Body Text Track.** This is accomplished by using the two remaining buttons at the top of the **Duct** "Property" window shown in Figure 16. The two remaining buttons, "Select Attached Nodes" and "Attach Selected Nodes," provide a means for the user to determine what **Nodes** are being affected by the constraint (by having them selected) and a means for altering which nodes are affected respectively. After clicking on the **Duct** in question, and then hitting the "Select Attached Nodes" button, the two Blocks that are already attached will be selected. Next the "Affects All Siblings" check box needs to be deselected, in order to allow a **Block** that is on a distant branch of the tree to be connected. Then a new **Block** is selected, giving us the figure below:

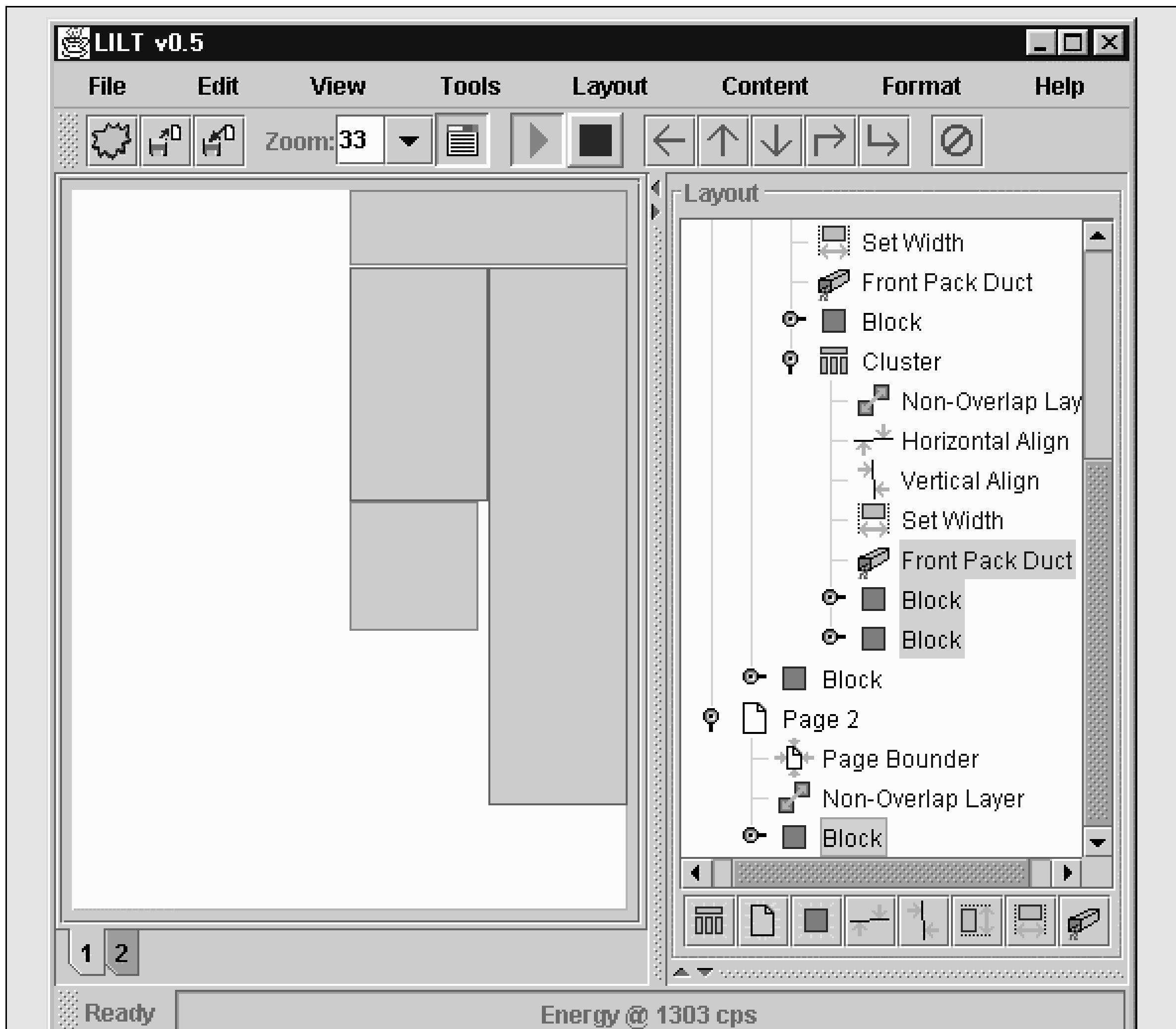


**Figure 20: In the middle of Step 10**

Next, the "Attach Selected Nodes" button is pressed. This causes the **Duct** constraint to be moved to the appropriate place in the hierarchy, and the new **Block** to be included in the Body Text content flow:

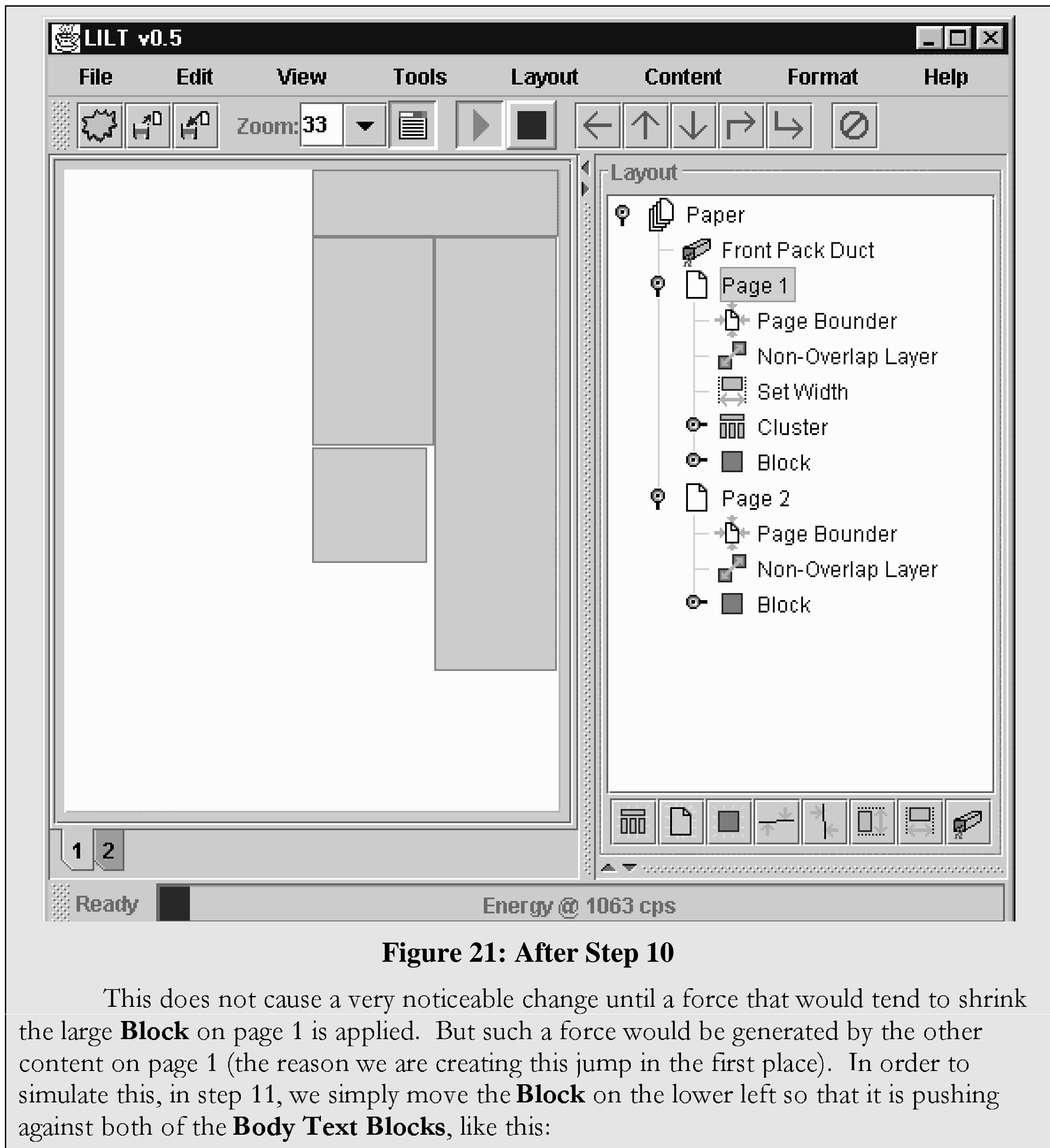


**Figure 21: After Step 10**

This does not cause a very noticeable change until a force that would tend to shrink the large **Block** on page 1 is applied. But such a force would be generated by the other content on page 1 (the reason we are creating this jump in the first place). In order to simulate this, in step 11, we simply move the **Block** on the lower left so that it is pushing against both of the **Body Text Blocks**, like this:

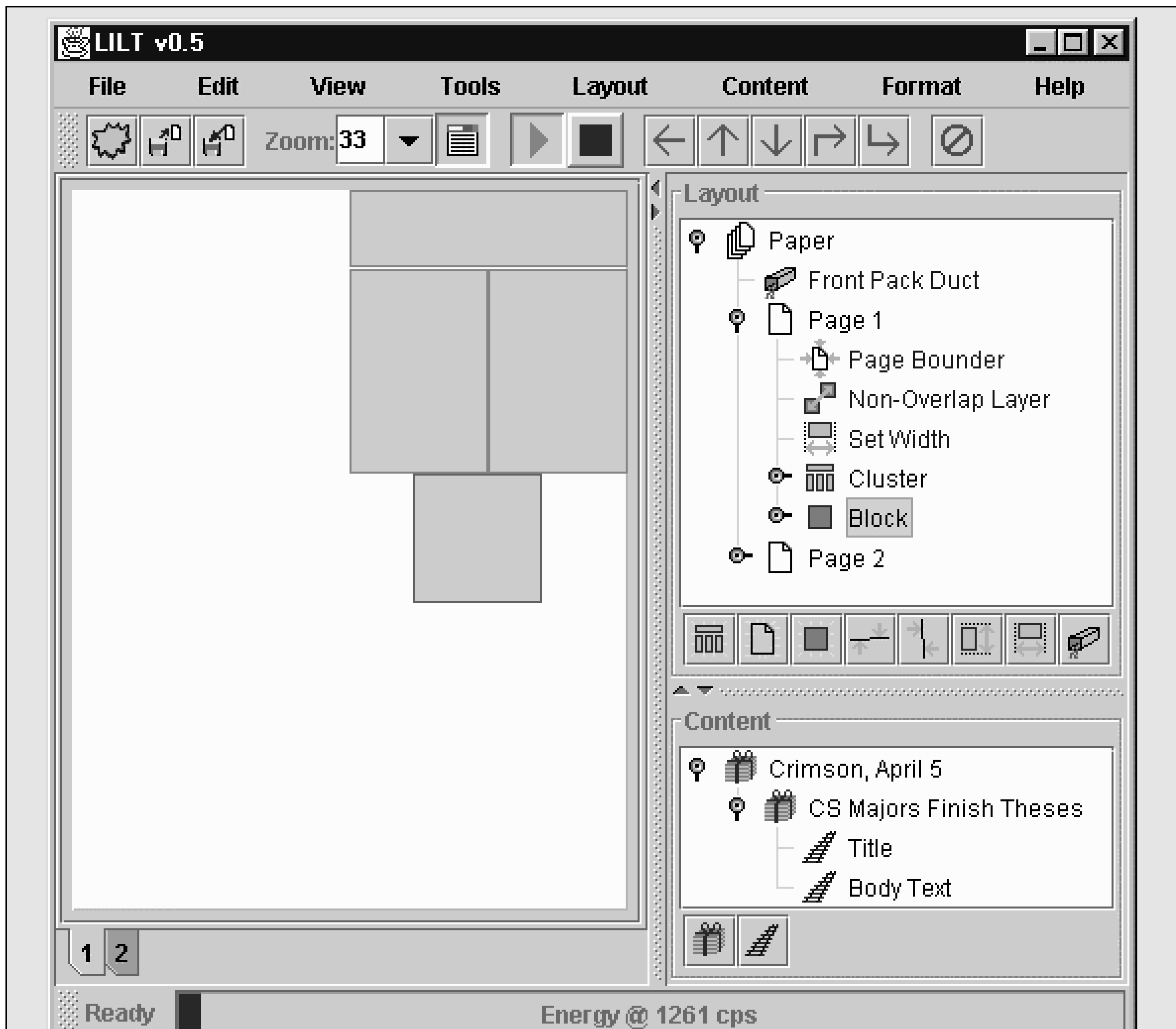


**Figure 22: After Step 11, page 1**

Because the Block on the back page has been linked into the text flow, it will grow as a consequence of step 11:

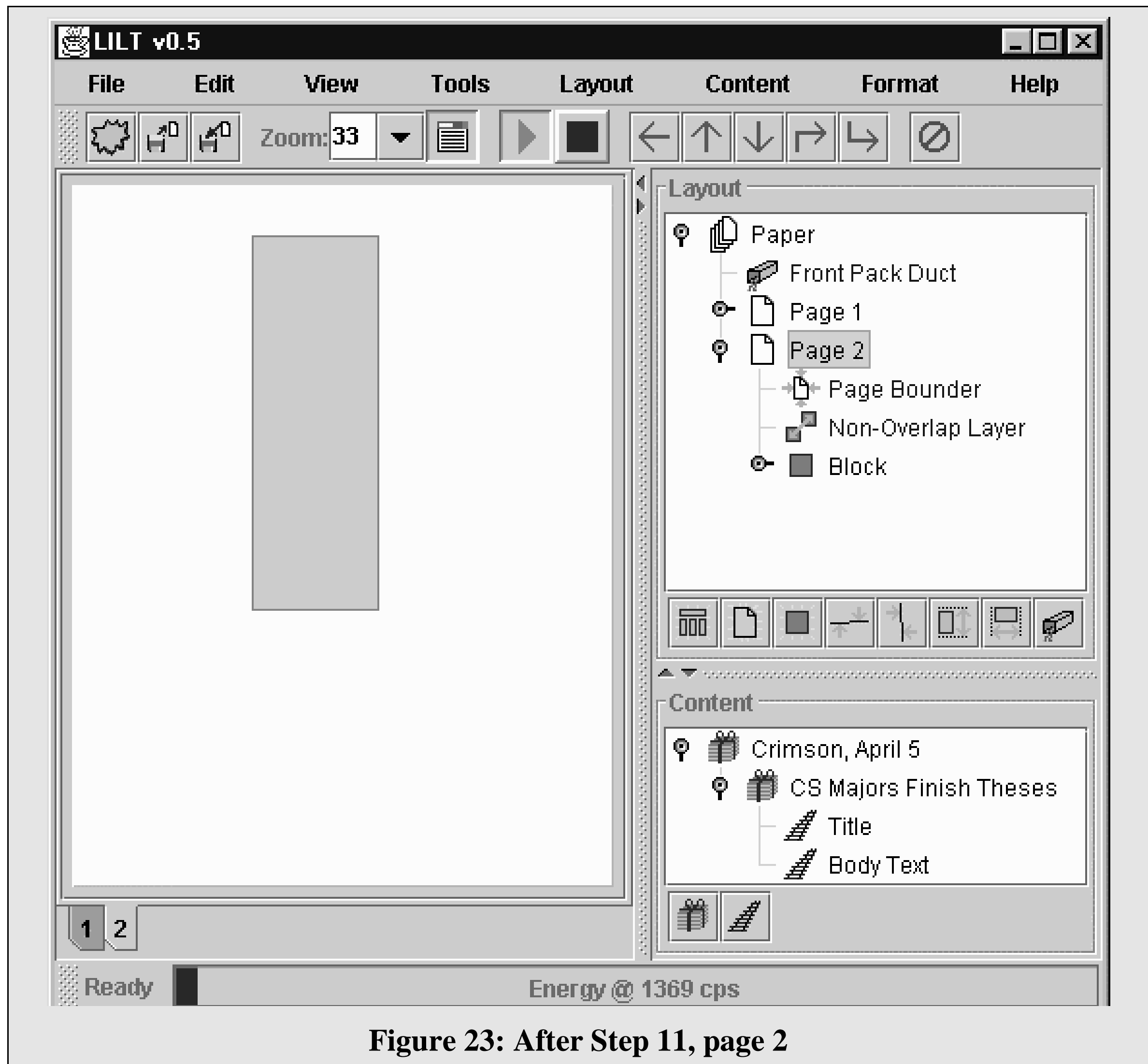


**Figure 23: After Step 11, page 2**

This brings our extended example to a close. The example has shown how an Article/Clump-level content and layout can be handled. Clearly, higher leve organization is also necessary to generate a complete layout. But the process for generating higher-level constraints is very similar to that of this example, the major difference being that all the constraints apply to **Clusters**. This being the case, let us turn to a more comprehensive look at the components of the **LILT** system.

# Chapter 6: Algorithms

There are a number of unique algorithms that have been developed for this projec , the most important of which is the constraint satisfaction scheme. In creating this scheme, the goal was to produce a system that provides for as many of the constraints described above as possible in a semi-automated layout framework. Toward this end, a system based on the mass-spring-update model in GLIDE was used. A mass-spring-update model provides the necessary semi-automated qualities allowing the user to anticipate the motion of objects in the simulation, much as one might predict the motion of billiard balls. However, significant modifications had to be made in order to render such a scheme compatible with the newspaper layout domain, and it is these modifications that are described below.

It is important to note that the dozens of constants in the algorithms described in this section must be chosen very carefully. As in GLIDE, it is vital for the various spring-constraints in the system to have the appropriate relative strengths in order for the system to behave as expected. It is even more vital for the constants affecting the simulation itself to be correct if the simulation is to converge quickly without introducing unacceptable levels of numerical instability.

## *6.1 The overall document model*

As in GLIDE, the mass-spring model here is based upon a modified Euler method for numerically solving for the positions of the masses. However, the algorithm in GLIDE has been modified to accommodate the dual-tree data structure described above. Instead of maintaining simple lists of nodes and springs, we use the tree structure itself t find the nodes and springs to update. Further, the simulation needs to be able to operate on both the layout and the content hierarchies, and it needs to be able to enforce constraints that do not conform to the mass-spring model (such as those that modify the attributes of other constraints in the system), or updates. Based on these criteria, the high-level algorithm for the constraint satisfaction algorithm is as follows:

Let us define the following variables:

The root node of the layout tree to be n
The root node of the content tree to be c,
The energy in the system on a logarithmic scale to be E (has a range from 0 to 1),
The time in the simulation to be T,
The change in time to b dT,
The damping constant to be Damp, and
The minimum energy for the simulation to continue to be $_{min}$
Various constants $k_i$

T = E = 0
Initialize both trees appropriately.

Repeat
{
    dT = $k_1$ + $k_2$ * E
    Damp = (2 - E/$k_3$) * Sqrt(the maximum spring constant)

```
    Call ApplyUpdates on n.

    Call ApplyConstraints on n, and note the maximum spring constant applied.

    Call FreshenNodes on n, and note the maximum squared-momentum engendered.

    Call UpdateContent on c.

    Update the Energy:
    E =log( 25 * the maximum squared-momentum / number of Edges in N)
    if( E > 1)
        E=1

    if(E < K_min)
        pause the simulation until the user changes something
}
```

Each of the four function calls in this code is discussed in detail below, but the other code deserves some explanation first. Getting the mass mass-spring simulation to work in the newspaper domain is much more difficult than it is in graph layout, since the model must be driven harder. The numerical methods used in GLIDE have been carefully tweaked in order to overcome the severe numerical instability that can appear when many of the springs in the model are strongly pressed against each other, a situation that occurs constantly in the newspaper domain. The key improvement that reduces the amount of numerical instability is to calculate a scaled value for the amount of energy in the system and then to vary dT and the damping constant based upon the value of thi energy. This allows the simulation to proceed quickly when there is high energy, but as the energy falls off, the simulation slows and gains precision in order to reduce the numerical instability that occurs as it approaches its equilibrium state. This enables the simulation to converge quickly, while still maintaining numerical stability in all but the most unusual circumstances.

## *6.2 Specifics of the layout model*

The actual behavior of the layout model is dictated by how the three calls on the root element (which is always a **Paper Node**) made in the algorithm above are implemented. It is therefore necessary to describe the attributes of each of the elements in the tree, and how they work together to make the model operate.

### 6.2.1 Node element algorithm

All Node elements are expected to provide a set of behaviors that can be called upon by other parts of the system. First, there are the three functions used in the listing above: ApplyUpdates, ApplyConstraints and FreshenNodes. In general, these are functions called by a parent on each of the elements in its **Node** child-list in order to update a specific aspect of the simulation and constraint solver (note that the root node represents a special case, as the functions are instead called by the high-level code shown above). **Node** elements provide at least a minimal set of attributes and behaviors that can be used by **Constraint** or **Update** elements when doing their calculations. Each **Node**

element also has certain values associated with it for use in the mass-spring simulation which include: Momentum, Position and Force (Masses are assumed to be unit sized).

#### 6.2.1.1 The ApplyUpdates function

This function provides a means for all the **Update** elements in the Tree to perfor their task. Each **Node** element in the tree performs the following algorithm when ApplyUpdates is called:

```
Perform any pre-processing this Node-type requires.
Call ApplyUpdates on every element in the Node child-list.
Call Update on each of the elements in th Update child-list.
Perform any post-processing this Node-type requires.
```

For efficiency reasons, some of the **Node** attributes are actually calculated during the pre- and post-processing steps in this function, and then cached in the node for one simulati cycle.

#### 6.2.1.2 The ApplyConstraints function

The ApplyConstraints function is very similar to the above, but operates on **Constraint** elements instead. As part of the simulation algorithm above, it is important to track the largest spring constant seen, so ApplyConstraints does this as well:

```
Perform any pre-processing this Node-type requires.
Initializ maximumSpringConstantSeen to 0.
Call ApplyConstraints on every element in th Node child-list, updating maximumConstantSeen
        after every call.
Call ApplyForces on each of the elements in th Constraints child-list, updati
        maximumConstantSeen after every call.
Perform any post-processing this Node-type requires.
Return maximumConstantSeen.
```

#### 6.2.1.3 The FreshenNodes function

Most **Node** elements implement this function to do a recursive descent similar to the previous functions:

```
If this Node is Anchored
   Return 0.
Perform any pre-processing this Node-type requires.
Initializ maximumMomentumSquaredSeen to 0.
Let F be the Force that has been applied to this node during the last ApplyConstraints call.
For each Node element n in the Node child-list
   Apply the force of F * (Number of Edges in n / Number of Edges in this Node) to n.
Call FreshenNodes on every element in th Node child-list, updating
        maximumMomentumSquaredSeen after every call.
Set F to 0.
Perform any post-processing this Node-type requires.
Return maximumMomentumSquaredSeen.
```

However, **Edge Nodes**, which are the base case, use a very different algorithm for this function:

```
If this Node is Anchored
    Return 0.
Set Momentum += dT * (Force - Damp * Momentum)
Set Position += Momentum * dT
Set Force to 0.
Return Momentum squared.
```

#### *6.2.1.4 Required node attributes and behaviors*

Node elements must provide a whole host of attributes and behaviors to make the system robust and workable. But there are certain vital ones that are worth enumerating (note that values like Force or Position are two-dimensional vectors in most **Nodes**, but are one-dimensional scalars for **Edges**):

- Enumeration of all child element
- Enumeration of each child-list individually (**Nodes**, **Constraints** or **Updates**)
- Accessing children by index
- Inserting/removing children
- Obtaining the current Position
- Moving the **Node** to a new Position
- Offsetting the **Node** to a new Position
- Obtaining the currently applied Force
- Applying a new Force
- Obtaining the current Momentu
- Obtaining the total number of **Edges**
- Obtaining the left/right/top/bottom-mos **Edge**
- Obtaining a bounding rectangle for this **Node**

These last three attributes are requested so often that it is very inefficient to calculate them each time, and as a result they are calculated once in the ApplyUpdates function, and cached for later use.

### 6.2.2 Constraint and Update element algorithms

Both **Constraint** and **Update** elements are responsible for changing the state of the simulation, but they act in different ways. **Constraint Nodes** act as **Springs** within the simulation and act by calculating and then applying forces to their attached **Node** elements. **Updat Nodes** (which are not present in the GLIDE system) differ in that they are not part of the mass-spring simulation. Instead they are rules that are evaluated once every simulation cycle (through a call to the Update function) that can modify the state of other elements in the layout hierarchy (including creating or destroying other elements in the tree).

**Constraints** (or **Springs**) have many attributes and behaviors, but the most important is the ApplyForce method. This method calculates Forces for each of its attached **Nodes** from the current state of these **Nodes** and the **Constraint** itself and applies it. Most of the **Constraints** implemen **Springs** that calculate their forces based upon Hooke's Law: F = k*(length - rest length); however some of the springs act in non-physical ways in order to effectively produce a layout. Most of these non-physica springs add to the numerical instability problem because they are non-linear in nature, requiring the algorithm that performs the simulation to be as forgiving as possible. Clearly in order to calculate the Force based on Hooke, a **Constraint** must have attributes for the spring constant, current length and rest length.

### *6.2.2.1 low-level constraints*

As we have seen, the lowest level constraint, **Ancho** ing, is a special case and is accomplished by the addition of a special flag in **Node** elements. This is equivalent to defining an **Ancho** as having an infinite mass, and attaching this mass to the appropriate Node. The next low-level constraint, **Page** bounding, is enforced by an **Update** element that can be applied only to **Page Nodes** and whose Update function uses the following recursive algorithm, first called on the **Page Node** itself:

If this is a **Block Node**
    If any of th child Edges are outside of the Page, move their Positions to be at the extremity of the Page.
Else
For each **Node** child element of this current **Node** do the following:
    If the child's bounding rectangle is outside the page
        make a recursive call on that Child Node.

This algorithm ensures that all **Nodes** below a page are bounded by it, since every descendent, not just the immediate children, will eventually be checked.

The non-overlap constraint is enforced by a **Constraint** element that sets up a new **Layer** and puts each of its attached nodes on this **Layer**. **Layers** are invisible sheets that exist above the parent **Node** upon which a set of child **Nodes** can sit. Layers work in a manner very similar to that of layers in traditional desktop publishing systems: the enable objects to overlap each other, and confer control over which object is on top of which. However, in contrast to what happens in traditional approaches, the ordering o the layers is easily observed and modified: it is defined by the ordering of the non-overlap **Constraint Nodes** in the layout tree.

The non-overlap **Constraint** only applies to its directly attached nodes, but in order to do this, it effectively maintains a spring among all the **Blocks** within each attached **Node** and all the **Blocks** in every other attached **Node** (but not among the **Blocks** within each of these sub-trees). These non-overlap **Springs** are non-physical in nature: the spring constant drops to zero when the blocks do not overlap, but is exceedingly high when they do. They are arranged in such a way that they always force blocks to move apart from each other and in such a way that they typically will seek the shortest route to a state where they no longer overlap. Also, because this non-physical situation promotes numerical instability, it is useful to have the springs remain active for a very small distance past where the blocks are no longer overlapping. These principals

are put to work in the following algorithm that is used for the ApplyForces function of the non-overlap **Constraint** elements:

```
Call the following recursive function between every pair of attached Nodes and return the
            nonOverlapSpringConstant.
ApplyNonOvelapForceRecursor(Node1, Node2)
{
    if(the bounding rectangles of Node1 and Node2 do not intersect)
        return
    else
    {
        if(Node1 is of subtype Block)
            if(Node2 is of subtype Block)
                Calculate the forces between Node1 and Node2 and apply them.
            else Node2 is a Block-Cluste
                Make a recursive call for each Node child of Node2 specifying Node1 as th
            other argument.
        else Node1 is a Block-Cluste
            Make a recursive call for each Node child of Node1 specifying Node2 as the other
            argument.
    }
}
```

### *6.2.2.2 Mid-level constraints*

Each of the mid-level constraints can be represented as a **Constraint** element in the layout Tree. Each acts as a set o **Springs** that apply forces to their attached **Nodes** in such a way as to enforce the constraint. There are four differen **Constraint** element subtypes (not including non-overlap and **Duct** constraints), and each applies its springs in a different way, as described below:

#### 6.2.2.2.1 Width and Height

Both these constraints simulate a set of physically accurate **Springs** that are connected between the appropriate paralle **Edges** of each of the attached **Nodes** and apply forces that tend to give these **Nodes** the appropriate dimensions. **Cluster Node** can be attached to a width or height constraint just as easily as a **Block** can. In this case, the force is always applied between the extreme **Edges** of the **Cluster** in the appropriate directions. For example, when a width constraint is applied to a **Cluster**, it will ac between the rightmost and leftmost **Edges** of the **Cluster** as illustrated in Figure 24.

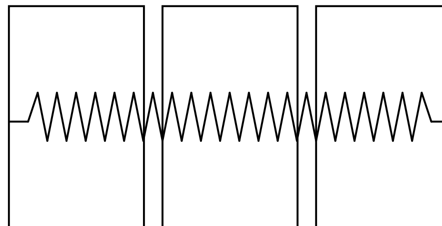

**Figure 24: A Width constraint applied to a Cluster**

The appropriate size is attained by giving the **Springs** applied in this manner a rest length that is equal to the desired width/height of the **Node**, a distance that is determined in one of the following three ways:

- A fixed value specified by the user
- The average current width/height of all of the attached **Nodes**
- The distance between any two parallel edges in the system.

#### 6.2.2.2.2 Horizontal and Vertical Alignment

These constraints also use **Springs** that directly follow Hooke's law. However, instead of attaching these springs within the same **Node**, alignment constraints apply them across their attached **Nodes**. These constraints can be applied to one of three locations in each direction. For horizontal alignment these are TOP, BOTTOM and CENTER. Center really means moving the entirety of the **Node** itself. For vertica alignment, we similarly have LEFT, RIGHT and CENTER. By default, these constraints apply to the same location on all of their attached **Nodes**. That is, they apply to all of the left **Edges** or all of the right **Edges**, or to the **Nodes** themselves. However, they can be set up to explicitly use a given location on each individual attached **Node**. In order to obtain the appropriate behavior, the average of the positions of the locations of all of the attached **Nodes** is taken. (**Clusters** calculate their CENTER location by taking the weighted average of all of their sub-**Nodes**' positions). A force is then calculated and applied to the appropriate location of each **Node**. This calculation uses a rest length that is the difference between the previously calculated average position and the position of the curren **Node's** appropriate location.

## *6.3 Specifics of the content model*

The call to the UpdateContent function in the high-level algorithm given in section 6.1 provides an opportunity for the content tree to update itself, and to modify corresponding elements in the layout tree if necessary. In practice, this means handling the content side of **Duct** constraints, and the rendering associated with the actual flow o content into the layout

This is accomplished by a simple depth-first descent down the tree, where each tree element evaluates all of the elements beneath. Whenever a **Track** is reached (and **Tracks** are always leaf elements), it calls the UpdateRange function for each Duct tha has been connected to it, which in turn mediates the data flow, as described in the nex subsection.

## *6.4 Specifics of the connecting ducts*

The connecting **Ducts** are actually updated twice, once as Constraints within the Layout Hierarchy, and once through their association with the Content Hierarchy. As a result, they must maintain some state between each of these different update calls in order to do their job correctly. First, the **Duct** keeps track of the amount of content (as an integer) that it was able to lay out in the last model-cycle. Secondly, the **Duct** maintains a record for each **Block** for which the **Duct** is responsible. The record has two fields: a rest length for sizing the **Block**, and an integral offset for how many characters the **Block** will contain. As mentioned in Chapter 5: An example interaction with LILT, it is

important that the data for the amount of content that has been laid out be integral. **Ducts** must be able to handle the fact that content comes in discrete units (usually characters, but it could really be anything).

The **Duct's** role as a **Constraint** is really rather straightforward, and like all **Constraints** is mediated by a call to the updateForces function:

```
For each attached Block
{
    Obtain the record for this Block
    Calculate and apply forces to this Block exactly like a height constraint, but using the rest
            length in the record in Hooke's Law.
}
```

The real work in a **Duct** is performed when the content tree updates it. This occurs when an attached **Track** calls the **Duct's** updateRange function, which takes as a parameter the length of the content to try to lay out (endPosition) and returns the actua length that it was able to lay out:

```
First it calculates the difference between this position and the amount it laid out last time:
If there are no attached Blocks, just return 0.
Integer Diff := endPosition - endPositionLastCycle.
If( diff != 0)
    Dif += the number of attached Blocks.
Integer BlockContribution := Diff/ the number of attached Blocks.
Integer FirstBlockExtraContribution := the remainder of Diff/ the number of attached Blocks.
StartPos:=0
For each attached Block:
{
    Get the record for this Block.
    Set record.OffsetField += BlockContribution
    If(This is the first Block)
        Set record.OffsetField += FirstBlockExtraContribution
    If(record.OffsetField < 0)
        record.OffsetField = 0
    Now call renderer.getNeededSiz (Block, attached Track, StartPos, record.OffsetField)
            Which returns the size needed  to lay out all this material.
    Set record.HeightRestLength := K * (this needed size - record.HeightRestLength)
    StartPos += record.OffsetField
}

Integer ActualOffset := 0
For each attached Block:
{
    Get the record for this Block.
    Call renderer.render(Block, attached Track, ActualOffset, record.OffsetField) Which should
            render the content into the block, and return the amount of content that actually fit.
    Set record.OffsetField := the length returned by this function.
    Set ActualOffset += the length returned by this function.
}
return ActualOffset.
```

This algorithm requires the two supporting functions: renderer.getNeededSize(**Block**, attached **Track**, StartPos, record.OffsetField) and renderer.render(**Block**, attached **Track**, ActualOffset, record.OffsetField). In **LILT** these functions are a simple calculation based on the assumption that each unit of content takes .1 inches to lay out. But in a real system, these functions would be very involved and would take care of actually laying out the data. For such a system to be run in a reasonable time, it will be vital for these functions to execute quickly. Thus these functions will probably have to use caching with some form of incremental update in order to produce a reasonable execution speed (and consequently model convergence speed).

# Chapter 7: Implementation Issues

**LILT** has been developed as a Java application, and as such can run in any Java 1.2 environment with a **JIT compiler**, provided that the machine the environment is running on is fast enough. Due to Java's slow nature and the use of real-time simulation, the system only runs adequately on a fast Pentium II machine or better.

Throughout **LILT**'s development, an attempt has been made to draw a distinction between the code that is used for creating the mass-spring-update model and the code tha uses that model to encode the newspaper domain. This should enable other collaborative applications based on mass-spring-update models to be developed using the same code base. This distinction was drawn very clearly in early versions of the program where each of these portions of the code existed in its own Java **package**, but these packages were eventually merged for efficiency reasons. Still, changing the specifics of the mode (by changing the properties of individual **Node** elements, or even by creating a different, non-hierarchical data structure) is very feasible.

It is important for a program that uses this framework to be multi-threaded. It is vital for the user to be able to manipulate the user-interface effectively, while the CPU-intensive simulation is running concurrently. This can only be accomplished if the mode is running on a separate thread that can synchronize changes in its state with the UI. In practice, this requires the model thread to accept changes to the state from the UI thread (caused by the user), and for the UI thread to periodically refresh its information from the model in order to make sure that the user is seeing the current state.

In order to facilitate both the reuse of code in other domains and the effective maintenance of the two threads and their associated timers, the current version o **LILT** has two **packages**:

- The Document **package**, which is now responsible for both the mass-spring-update model itself, and the two tree structures used to represent the newspaper domain.
- The UI **package**, which contains all of the code used to create the **LILT** graphical user interface.

# Chapter 8: Possible future improvements for the layout paradigm

**LILT** is not a piece of commercial software, and the system is not capable o coming close to *really* producing a newspaper right now. For an obvious example: it is currently not possible to place actual content into Tracks, but only to specify a parameter that indicates a hypothetical amount of content. It would not be hard to add the capability of accepting actual content, and the capability of rendering this data, but it still needs to be done. Additionally, the current implementation of **Ducts** only modifies the height of attached Blocks. It will be an important but relatively easy extension to this work to generalize the **Duct** constraint to work in both directions. As our focus here is not primarily practical, however, let us turn to the numerous additional improvements that could be made that are of more theoretical interest.

The current system provides no means to add a complex layout sub-tree all a once. Chapter 5: An example interaction with LILT shows a prime example of this. The user needed to add a new layout sub-tree (a **Clump**) that would contain all the blocks for a new **Article** in the newspaper. But instead of being able to add a generic **Clump** sub-tree, each individual element had to be added one at a time. In order to facilitate this, a macro language could be developed that might let a whole new sub-tree be added easily with default layout characteristics. Such a scheme could work in a manner similar to the way that Juno-2 lets users create libraries of drawings that can be added and built upon to create new and more complicated images. In this endeavor it could prove useful to have **Articles** contain a minimum set of standard tracks, such as the ones listed on page 15. These standard tracks could be specified in a macro that adds a new **Bundle** to the content hierarchy representing a single **Article** that includes a **Track** for each of the standard tracks. Macros for adding layout elements could then expect to find these **Tracks** in the **Article** level **Bundles**. Ideally, the macros themselves would be configurable, so the user could create his or her own buttons in the user interface for precisely adding objects he or she uses most, as in Juno-2 (see Nelson).

Another very significant improvement would be to implement some or all of the high-level constraints described in secti 3.1.2.3. This will require more information to be gleaned from the content hierarchy and will call for a far tighter integration between the content and the layout, so that automatic inferences can be drawn. Such relationships should be possible, but are very difficult to work out. Further, within this process, the more information that can be implicitly gathered from the existing content hierarchy the better, since the supplying of additional information will only take up the user's time.

It would also be highly beneficial to add the capability for the program to column- or page-break text only in specific places chosen either by the user or by some sort o natural-language processing. Usually there is a fixed set of locations in the text where a page or column break is desirable. It would be advantageous if the layout could be constrained by such restrictions, but only if they could be easily overridden.

# Chapter 9: The multi-user case: concurrent editing within the workflow

Now that we have considered the layout problem in depth, it is time we turned our attention to the question of how to get multiple humans to collaborate on the shared goal of producing a single newspaper.

## 9.1 The problem

In order to understand the problem, it is useful to see how a small newspaper (using one of the commercial packages) handles the inherent difficulty in trying to get so many people to cooperate on a single goal: a readable, attractive newspaper. Using the Harvard Independent and Bronx Science's Science Survey as appropriate models, generally the following flow is used:

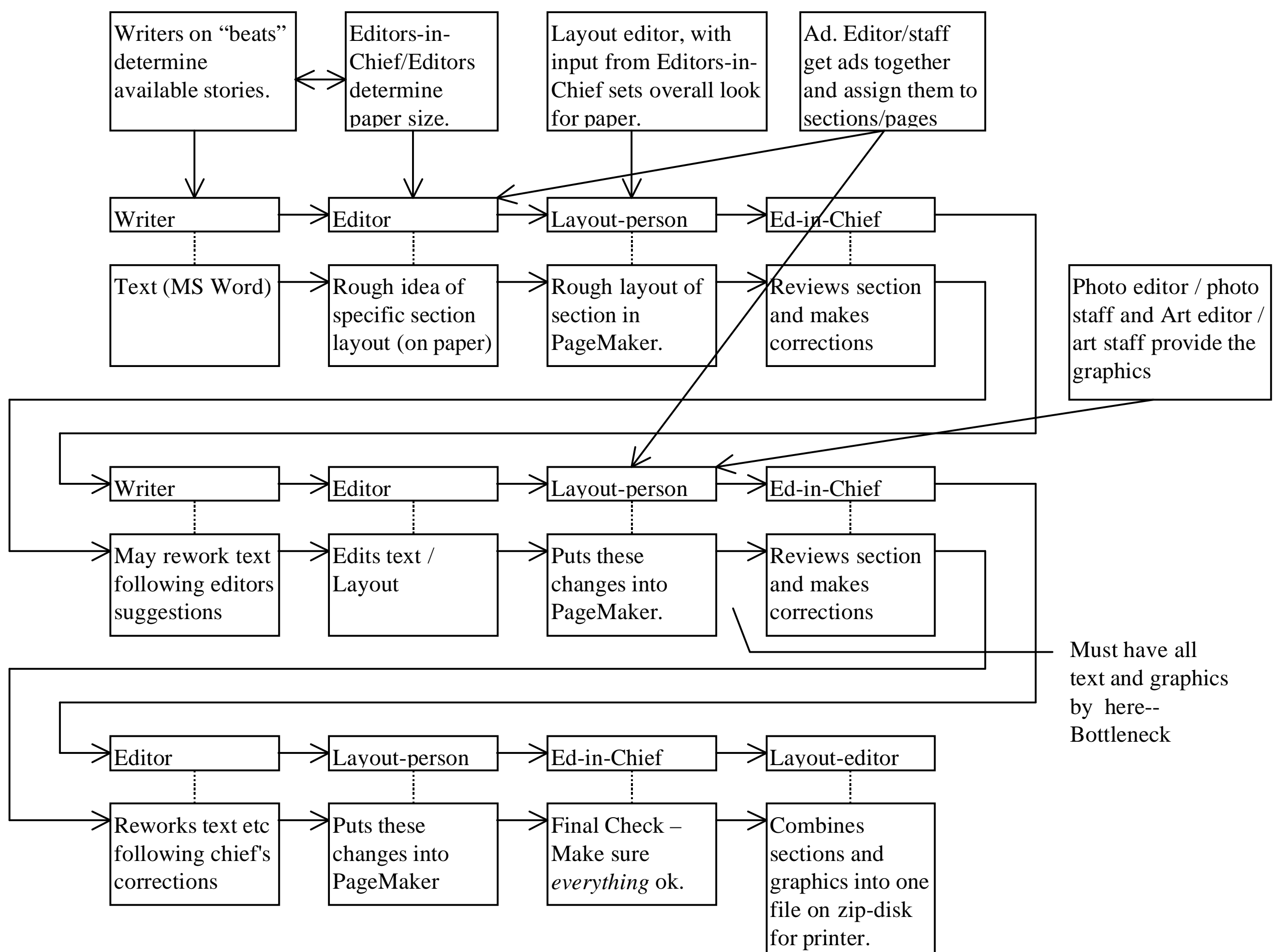


**Figure 25: An idealized newspaper production workflow**

This is the ideal workflow that people strive for. It is never actually attained for a million reasons. But if the software were able to handle multiple users more effectively,

perhaps orchestrating and displaying different people's contributions in a visual manner, a very smooth flow could be achieved.

The goal is to have all the text, ads and graphics complete by the first row and then to do corrections, breaking stories, bylines, titles and captions in the second. The third row is then for a final clean-up. However, what often happens is that some conten is not available until the flow has already hit the second row. Then that section of the paper hits a bottleneck, as indicated.

When this happens, only one person can be working on that section's layout at a time. This means that the editors and writers who are working on the text, not the layout cannot modify the text while the layout is going on. It also means that each section can only be laid out by one person at a time, and that a combining step must be included a the end in order to put all the sections (in separate files) into one document in PageMaker (or Book file, as Quark calls it). These and other limitations suggest that the following scenarios might be handled far better:

- A breaking story arrives soon before the paper goes to press and potentially the entire paper has to be laid out afresh. We want automation in this layout and the capability for multiple people to collaborate on various aspects of the layout in parallel.
- Stories, ads, photos, graphics arrive very late in the process. We want everything else to proceed smoothly until these items arrive. We want to be able to shrink/expand other stories or in other ways cope if these items do not materialize.

## *9.2 Current approaches*

There are commercial systems currently available that will do source-file monitoring on publishing data similar to the ones commonly used for computer source code. One good one is the Quark Publishing System. These systems allow the data of a paper, including layout information, to be stored on a centralized server. Different users can then check out portions of the data and modify it, later checking it back in. Systems like these enable the editors and writers to make their corrections right into the layout package instead of wasting the layout people's time by retyping all the corrections. They succeed in keeping the layout and the text distinct so that the layout and the editing can go on in parallel. However, there is an important limitation to this: these systems require that when text is edited, care be taken not to change the area it takes to lay it out. If this area changes, then the layout person has to do an explicit and manual re-layout.

## *9.3 Towards a solution*

Ideally, a system could be developed that updates the layout information dynamically as text lengths change, and that lets multiple people work on the same layout at once. The layout paradigm presented here provides a good foundation for attempting to produce a multi-user newspaper production environment that has these important new collaboration-promoting features. **LILT** provides a good basis for such an ambitious goal for the following reasons:

- The strong separation between the data model and the user interface should allow for an easy transition to a client-server type environment.

- **LILT** uses relative constraints rather than absolute positioning to create a layout. This provides the automation necessary to have the layout update based on dynamic changes to the content
- The dynamic nature of the system, where the effects of a change are expressed immediately, is an important first step for letting multiple users (or computers) affect the layout simultaneously.
- In the mass-spring-update model, the effect of a given local change tends to fal off with the square of the distance from that change. This is an importan property in a multi-user environment, where people will be making changes simultaneously: one wants the changes users make to be as local as possible to prevent them from ruining each other's work, or rendering it obsolete.

This last property is important, as it addresses the most fundamental problem of creating systems that promote collaboration: making the agents involved aware of each others' intentions. Clearly, the falloff of the spring model is a very useful property, although it does not yet come close to solving this important problem—a great deal of additional work is required.

# Chapter 10: Conclusion

Newspaper production is a very complicated affair with many, often conflicting, requirements that need to be resolved within a short timeframe. Traditional layou techniques simply do not provide robust methods for aiding users in solving these constraint problems. A system that can enforce the low, middle and high leve constraints and harness the dual hierarchical nature of the domain should be able to increase the productivity of its users immensely. This constraint satisfaction is a form of human-computer collaboration in which the computer assists the human in an informed way, by taking on the more tedious aspects of the layout problem, allowing the human user to concentrate on more aesthetic decisions. **LILT** is also a step towards an even more comprehensive system that is capable of letting multiple users work on the same newspaper in parallel, as it solves many of the problems such a scenario engenders.

## Chapter 11: Glossar

*Anchor*........... The constraint that a given **Node** should not be moved by the constraint satisfaction engine.

*Article*............ All the content information pertaining to a single story in the paper including both graphics and text. That is, a **Bundle** that contains only **Tracks** pertaining to a single story.

*Block*............. A box containing either text or a graphic. It is defined by its 4**Edges**, which indirectly specify its location and size. It also maintains information about its desired margins.

*Block-Cluster*. A **Cluster** that is a descendent of **Page Node**, and thus has only **Blocks** and other **Block-Cluster** on its **Node** child-list. Therefore, a **Block-Cluster** cannot contain another **Page Node**.

*Body Text* ...... The text in an **Article** that is the actual story itself, not including any of the other details such as title, **Byline**, or **Callouts**.

*Byline*............ The text in an **Article** that attributes the piece to its author.

*Callout* .......... Short, usually striking, phrase that is singled out from the body text and placed in a larger font, often as a means of wasting space.

*Clump*............ Any **Cluster** in the layout hierarchy that is comprised entirely of **Blocks** that are from the same **Article**, but whose parent cluster contains **Blocks** from multiple articles. **Clumps** are typically second or third level **Block-Clusters**.

*Cluster*........... A **Node** in the layout tree that represents a group of sub-units whose layout is constrained with respect to each other in one of several ways. Although **Clusters** are general organizational elements, at any given time they must either be **Page-Cluster** or **Block-Clusters**.

*Constraint*...... 1) {generally} A condition that is specified either by the problem domain or by the user, which the machine must attempt to satisfy.
2) {specifically} An element of the layout tree that acts as **Spring** in the mass-spring-update simulation and thereby modifies the **Nodes** in the layout.

*Edge*.............. A **Node** in the layout tree that represents one side of a **Block** and is defined by its position either horizontally or vertically.

*Duct*.............. The constraint that is responsible for ensuring that the same amount o space be reserved in the layout as is required by the content, and dictates what content flows into its attached layout**Nodes**.

*Jump*............. Newspaper-speak for what happens when an article begins on one page and is continued on another.

*Layer*............. A virtual plane above each page, upon which blocks can be placed. All blocks on a given layer are forced to not overlap.

*LILT*............. ELastic Interactive Layout Tool: The name of the program that has been developed in this study.

*JIT Compiler*. Just In Time Compiler: A compiler used by some Java environments to compile Java byte codes into native machine code on the fly, often resulting in a tenfold speed improvement.

*Node*............. Any element in the layout hierarchy that represents a displayable objec and not a constraint. Such elements include **Papers**, **Pages**, **Clusters**, **Blocks** and **Edges**.

*Package*......... The Java term for a collection of related code that works together as a cohesive whole and which can be reused in different programs.

*Page*............. A **Node** in the layout tree that represents a single page of the paper. **Pages** can contain either **Blocks** or **Block-Clusters** and are defined by their width and height and color.

*Page-Cluster*.. A **Cluster** that is only a descendent of other **Page-Clusters** and the root node (which is always a **Paper Node**). **Page-Clusters** only have other **Page-Clusters** or **Pages** on thei **Node** child-list and thus cannot contain **Blocks** or **Edges**.

*Paper*............. A **Node** in the layout tree that represents the paper as a whole. **Paper** node is always the root of the layout tree and can exist nowhere else in the tree. **Paper Nodes** can contain either **Pages** or **Page-Clusters** and handle the assignment of page numbers to these pages.

*Spring*............ Any element of the layout hierarchy that acts as a spring in the mass-spring-update simulation. That is, it is has a spring-constant and acts on attached **Nodes** by the application of force.

*Track*............. One piece of information about a given article, usually a single sequence of text, or a single graphic element.

*Update*........... An element in the mass-spring-update model that modifies other elements in the model in a way that does not conform to the mass-spring metaphor. More specifically, in **LILT**, this is any element of the layout hierarchy that modifies the state of another element in the layout hierarchy but does no act as a **Spring**. These are often used for syntactic constraints, but can be used to modify the parameters of any other object in the simulation (or to add and remove other elements).

# Chapter 12: References

Adobe Inc. Adobe PageMaker 6.5

Armstrong, Eric, Tom Santos and Steve Wilson. "Understanding the [Swing] TreeModel." Available at `http://java.sun.com/products/jfc/tsc/tech_topics/treemodel/treemodel.html`. 1999

"G & S House Style." Unpublished Guide by G & S Typesetters, Inc. Austin, Texas, 1997.

Johari, Ramesh, Joe Marks, Ali Partovi, and Stuart Shieber. "Automatic Yellow-Pages Pagination and Layout." Journal of Heuristics. 1997 : 321-342.

Nelson, Greg. "Juno, a constraint based graphics system." *Computer Graphics (Proceedings of SIGGRAPH '85)*, 19(3):325-243, July

Nelson, Greg, and Allan Heydon. "Juno-2 Constraint Based Drawing Editor." Digital Systems Research Center: SRC Research Report 1994-131a. December, 1994.

Nelson, Greg, and Allan Heydon. "Juno-2 Language Definition." Digital Systems Research Center: SRC Technical Note 1997-009. June 30, 1997.

Ryall, Kathleen. "Computer Human collaboration in the Design of Graphics." Unpublished doctoral dissertation, Harvard University. 1997 : 15-33, 71-76

Ryall, Kathleen, Joe Marks, and Stuart Shieber. "An Interactive Constraint-Based System for Drawing Graphs." Harvard University, 1997.

Sun Microsystems Inc. "The Java Tutorial: A practical guide for programmers." Available a `http://java.sun.com/docs/books/tutorial/index.html`. 1999

Quark Inc. QuarkXPress 4.0

Quark Inc. "QuarkXPress Brochure." Available a `http://www.quark.com/pdf/brochures/qxpbroch_us.pdf`. 1999.

Quark Inc. "Quark Publishing System Brochure." Available a `http://www.quark.com/pdf/brochures/qpsbroch_us.pdf`. 1999.