

A User Interface for Customizing Cane Layouts in Virtual Glass

by

Kimberly Baldauf

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2011

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

February 2013

Massachusetts Institute of Technology
All rights reserved.

Author:

Department of Electrical Engineering and Computer Science
September 13, 2012

Certified by:

Erik Demaine
Professor
Thesis Supervisor
September 13, 2012

Accepted by:

Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A User Interface for Customizing Cane Layouts in Virtual Glass

by

Kimberly Baldauf

Submitted to the Department of Electrical Engineering and Computer Science
on September 13, 2012 in Partial Fulfillment of the Requirements for the
Degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Cane pulling is a technique used in glass blowing to build up intricate patterns which come out in the final piece. Virtual Glass was created to model the cane pulling process from start to finish. There are a variety of standard cane layout patterns, but in truth the full range of possibilities is limited only by the glassblower's skill and imagination. For this reason the previous Virtual Glass interface, which provided only templates, proved to be too restrictive for users. In this thesis I describe an alternative interface we have developed which frees users from the limitations of template patterns. I provide a side-by-side comparison of several previous cane designs replicated in the program with and without the use of this interface. I also detail several unimplemented features and describe the foreseeable challenges with adding them, as well as how they might integrate into the interface when they are included.

Thesis Supervisor: Erik Demaine
Title: Professor

INTRODUCTION

The importance of cane in blown glass sculpture has increased substantially in recent years, but other than direct experiment there is a lack of tools for designing glass cane. Virtual Glass is an attempt to offer glassblowers the chance to see what their ideas would look like virtually, without having to go through the process of making a cane just to observe the results. Creating cane as well as glass blowing in general is time and labor-intensive activities. By using Virtual Glass, glassblowers can change small parts of their designs easily rather than having to pull an entirely new set of cane in order to make a change when they don't like the way an initial design turned out. It allows glassblowers without much experience to visualize every step in the process, or to design a cane that they hadn't necessarily thought about making because they lacked the experience to know it was something to try. By bringing the process of designing cane into the digital age, Virtual Glass will inject new life and new ideas into the field of pulling cane.

The team working on Virtual Glass consists of seven people. The coding team consists of Andrew Winslow, Benjamin Lee, Jim McCann, and me. The rest of the team consists of Erik Demaine, Martin Demaine, and Peter Houk, who along with Jim McCann provide ideas and feedback for the project from the glassblowing perspective. It can be downloaded for free from <http://virtualglass.org/>.

Virtual Glass is a program created for the purpose of simulating the glassblowing technique known as pulling cane. In cane pulling, pieces of molten glass are arranged in patterns and then elongated, often while a glassblower twists the ends to produce a periodic design. These can then be further arranged and formed into different shapes, creating a multi-step process with lots of freedom afforded to the glassblower. Virtual Glass simulates this entire process in an effort to make it easier and more enjoyable for a glassblower without lots of experience to visualize this progression.

Virtual Glass models this by describing the twisting mathematically. This was decided to be a better choice than a fluid simulation for several reasons. Virtual Glass is meant to be a program for casual

users, and fluid simulations take time and are computationally expensive. When trying to keep users interested, the speed of a program is one of the most important factors. Finally, the results achieved using mathematics produce a “perfect” cane: one with perfect symmetry and without potential flaws introduced by imperfect materials or human error, and these results are rather similar to the average cane produced using the process described. Because such cane is expected to be perfect, it’s expected that it should be easy to arrange cane in ways that will match this perfection: rotationally symmetric, or with equal spacing where it is desired.

Previously Virtual Glass fulfilled this expectation, but at the cost of the user’s ability to create a wide variety of styles. In collaboration with the team, I introduced a new interface to the program which now allows users the freedom of customization while still being able to build off these “perfect” bases. By combining the previous ability to make ideal layouts with new customization functionality, the user can take advantage of perfection without being constrained by it.

RELATED WORKS

There are quite a few virtualized glass projects, but none of them have the same focus as Virtual Glass. One of the major focuses of other glassblowing software is instructional, where programs are designed with various age groups in mind in order to teach about glassblowing. These are primarily web-based to allow access by a wide audience and involve stepping through the process of making a piece of glasswork. The emphasis is on the process itself, not the finished product, so the potential results are extremely limited in scope. Two examples of instructional programs are “Fireworks of Glass” (Children’s Museum of Indianapolis) and “School by Fire” (Museum of Glass, 2002). Both are available to the public online. “Fireworks of Glass” demonstrates the process of making blown glass and is intended for children, with a narrator who visually demonstrates the process interactively one step at a time. The achievable results vary only in color and how elongated the shape of the piece is. “School by Fire” demonstrates the

process of making a style of Chihuly piece called “macchia” and is intended for adults, relying more on text to describe each step of the process. The results for this program are different only by color. Beyond a few minor design choices, the user has very little control over the finished product.

Another focus of projects is creating freeform results which mimic glass in visual presentation. These programs don’t follow the process of glassblowing and often don’t impose any physical constraints on the results which necessitate that the outcome be possible to make in real life. Sometimes, these aren’t even specifically designed to create glass objects at all: one video I found showed the user creating a sculpture in the traditional CAD program SolidWorks (Kapsan, 2010). These sorts of programs don’t offer any suggestion how one might go about creating the sculpture one designed, resulting in few limits on creativity but a difficult time translating works into reality. “iGlass” (Bumbletech, 2009) treats glass similar to some sort of putty on the end of a stick: while the tools used in the user interface have real-life equivalents, the way that they operate on the glass is different from the way the same tools are used in real glass shops.

By studying these two particular focuses, we came to set the goal of striking a balance between being overly constrained in scale and having a program that’s increasingly complex to operate as well as translate to the real world. Too much restriction stifles a user’s creativity, but being overly free-form can leave a user with no idea how to bring their modeled creation into the real world, or even worse, how to operate the software at all.

Projects with academic focuses are heavily geared towards glasswork on the scale of mass production. One topic involves handling the dynamics of molten glass via fluid simulations (Giannopapa, 2006). The focus is on the cooling that happens when blowing glass bottles and these programs tend not to be accessible to the public. They are not used to design individual pieces, but rather to model existing pieces or to create a new template for large-scale production. Another topic of research has been creating machines or robots which can execute the process of making glass bottles without the need for human

interference (Ingle, 1924; Marechal et al, 2004). These are not meant to be used by a casual enthusiast, or someone interested only in making their own pieces.

APPROACH

Virtual Glass is intended to allow glassblowers to create their own cane designs while staying mostly in the realm of possibility. It does not constrain users to a single shape, or a limited color pallet. It is meant to be used to plan a piece from start to finish. With it, a glass blower can design pieces which can actually be made. To this end, it must be easy to use, yet still powerful. A strong user interface is essential to accomplishing these goals. The final users of Virtual Glass are not the software designers or someone who will receive special training, and thus the product must be simple to understand and work with.

For the first iteration of Virtual Glass, I worked almost exclusively on the user interface for modifying individual canes within a larger cane pattern. This interface included the way to specify a color selection, modify the size of a single cane, and position canes or bundles on the workspace using coordinates. I only worked on the interface and some of the features of it; I did not implement almost any of the back end.

One of the problems with the first version of Virtual Glass is that there was a poor interface for initially designing cane layouts. It was exceedingly difficult for the user to create regular patterns, which are standard in cane pulling. These patterns usually take the form of either straight lines or circles of cane evenly spaced around the center. In the first version of the software, canes were freely moved around with the mouse, and there was almost no “snapping” functionality allowing a user to line up canes with respect to each other. Additionally the center also lacked snapping functionality. Canes were placed in the center when first added to a layout, but once moved it became very difficult to put it back by hand. I added an interface with the ability to give strict coordinates as one attempt to mitigate this problem. By using either

radial or Cartesian coordinates, a user could line cane up in a perfect circle or along straight lines without too much effort. Another member of the team implemented a solution involving Inkscape, an outside program, and importing cane layouts into Virtual Glass. Through this method a user could externally create the layout they wanted and then bring it into the program. It was found that both of these methods were unwieldy and too freeform for the purposes of Virtual Glass. The Inkscape method required acquiring and learning to use an additional program, and the coordinates method lacked the intuitiveness of dragging around canes with a mouse.

In the second version of Virtual Glass, this problem has been remedied by allowing the user multiple parameterized templates to choose from which they can then customize further. In collaboration with the team, I have developed a customization interface which allows users to expand upon these templates in order to offer users a much wider variety of choices when it comes to designing the layout of their cane. The user interface that displays the templates available to select, implemented by other members of the team, is part of a larger interface where users can drag colors and existing cane designs from the library on the left side into a 2D top view of the cane, and the 3D results are displayed on the right side. The middle of this interface is where we decided to add the layout customization interface, which co-exists with the previous version and allows both to be used interchangeably.

It's impossible to include every single value the user could possibly change in a graphical template without it getting unwieldy, so I created a more "hands-on" way of modifying the templates to allow users more flexibility. This is similar to the way canes were arranged in the original version of Virtual Glass, however additional support for arranging cane was added, including the ability to modify multiple canes at once to keep an operation such as a resize consistent among elements within a template. Also unlike the previous version is that arrangement in this interface is done in 2D rather than 3D. While the results are still displayed on the side in 3D, 2D makes for an easier interface, since it doesn't necessitate the user manipulating the camera nor does it cause an issue with perspective.

This interface was specifically separated from the existing interface where the cane is dragged into a layout for two reasons. The first is that it limits the amount of functionality available on any given screen, so the operations can be kept relatively simple without fear of overlap. The second is because it is easier to keep the customizing operations distinct from changing the cane within a parameterized template, due to the customization removing the parameterization. We don't want the user to accidentally lose the parameters because they mistakenly clicked and dragged on the main interface; customizing a template should be an intentional decision on the part of the user. Because of this, it is a separate interface which can be worked on deliberately by the user.

BEHAVIOR

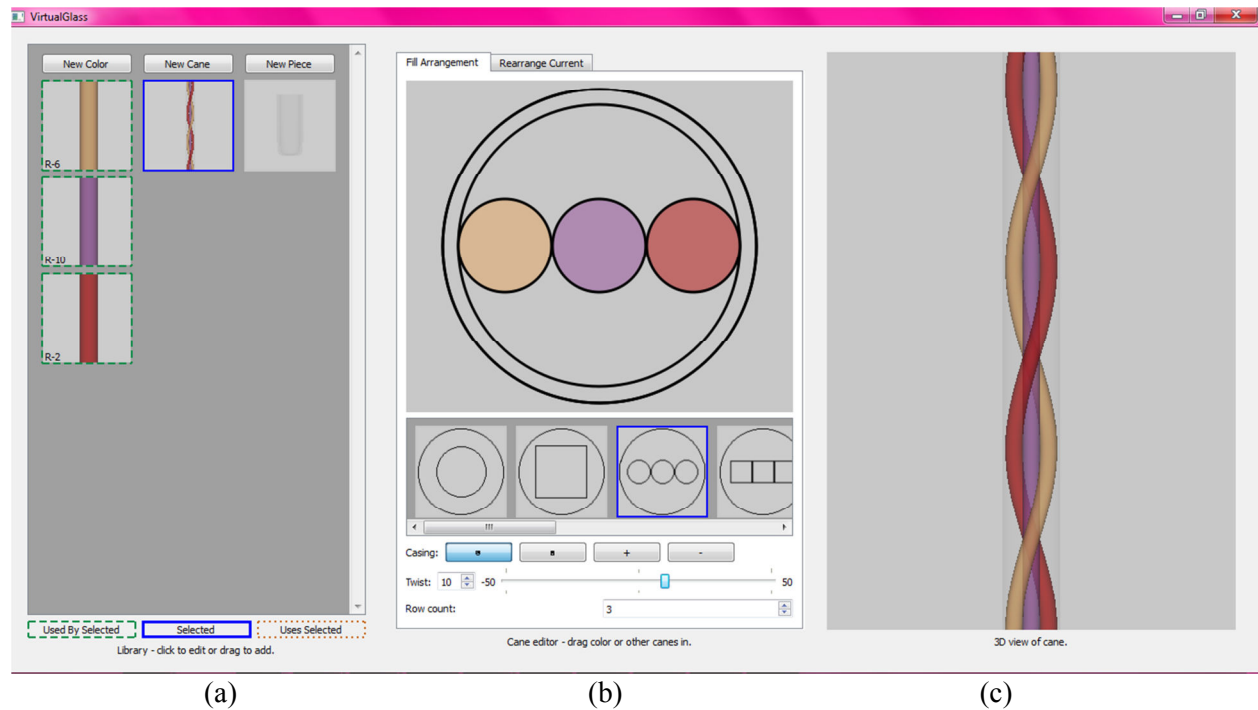


Figure 1: A view of the entire Virtual Glass program. The library (a) displays icons of the components, the editor (b) where most of the editing functionality resides for different components, and the preview (c) which displays a 3D rendering of the currently selected component.

The layout customization interface is just one component of the larger Virtual Glass program. The layout of the program is a three-panel design positioned horizontally. The leftmost panel (Figure 1a) is shared between the program's different modes and houses the library of components that the user creates. The rightmost panel (Figure 1c) shows a 3D visualization of the component currently being edited, and the view changes depending on what type the component is. The middle panel (Figure 1b) is the most dynamic, changing greatly between the modes of editing color, cane, or a piece. The customization interface is used for editing cane layouts alongside the previous existing interface, and the two can be toggled using a tab at the top of the panel.

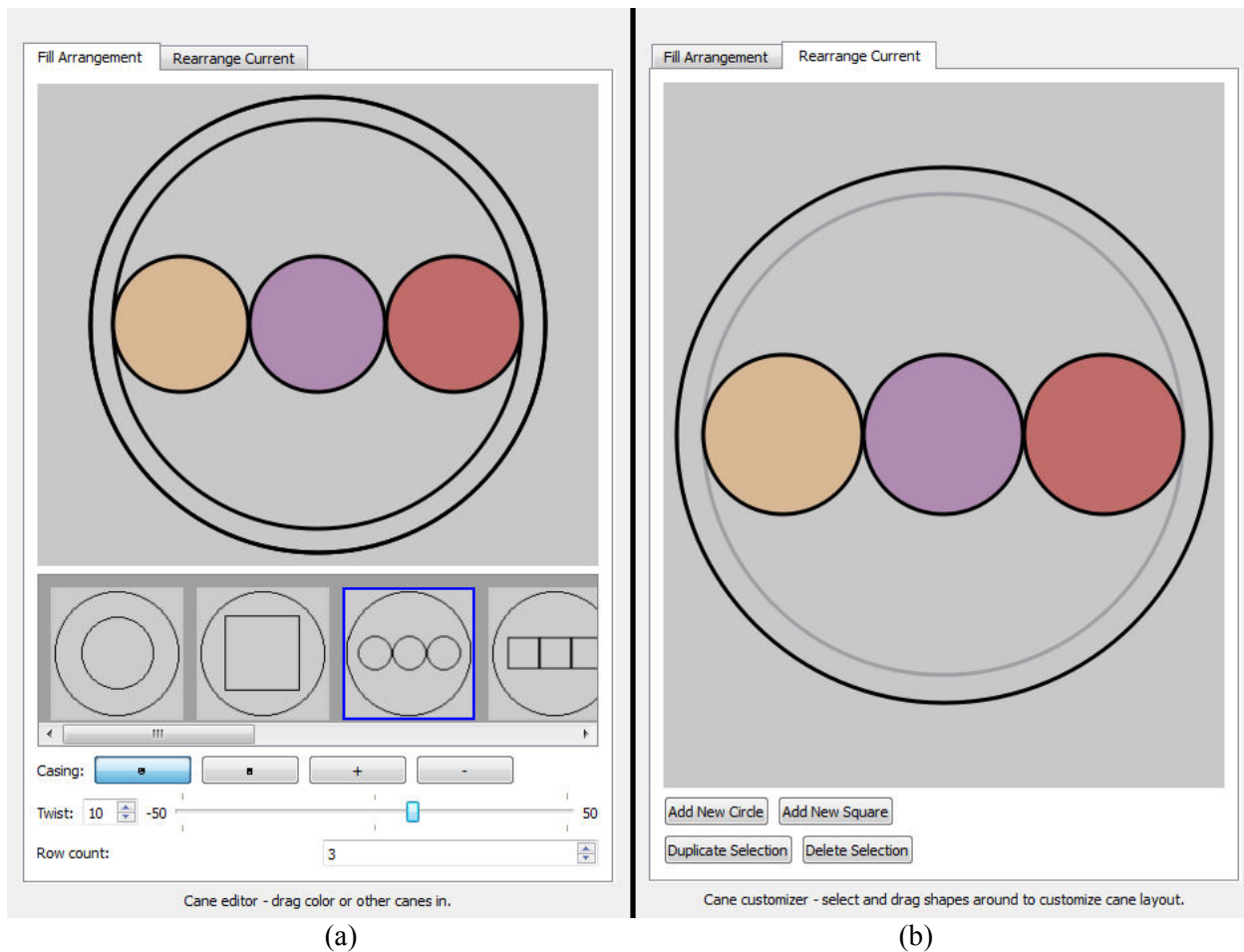


Figure 2: A side-by-side comparison of the two interfaces for cane editing. The fill interface (a) gives more space to the templates and parameter area at the bottom, while the customization interface (b) focuses primarily on the layout itself with less space delegated to buttons.

The cane editing interface is separated into two parts in order to prevent a collision of functionality. The fill interface (Figure 2a), labeled “Fill Arrangement”, is the default interface, and is the one with which users select a pre-made cane template and add casing, twists, and fill the template slots of the layout. Actions a user performs in the fill interface can almost all be easily described and replicated exactly with little to no visual aid required. The customization interface (Figure 2b), labeled “Rearrange Current”, is the one that I have developed with help from the team, and involves customizing pre-made templates. It allows for a much greater freedom of creation, and involves doing steps which may be difficult to exactly duplicate using a mere description of actions.

The primary functionality of the customization interface allows users to modify the template slots in order to customize their layout to exactly the way they envision it. These slots can be moved, resized, duplicated, or removed, or completely new slots can be added. The layout of the customization interface is intentionally similar to the layout of the fill interface, with the display of the cane's arrangement at the top and buttons beneath it. The top row of buttons is made of the two concerned with adding completely new slots to the layout: the Add New Circle and Add New Square buttons. When pressed these buttons add a new, empty slot to the center of the current cane. The new slot will either be a circle or a square shape depending on which button was pressed, as these are currently the only two shapes supported by Virtual Glass. The bottom row of buttons involves the operations that are dependent on the current selection: the Duplicate Selection and the Delete Selection buttons. The Duplicate Selection button makes a copy of the currently selected slot(s) and their fills, if any, and adds this copy to the current template. The Delete Selection button removes the currently selected slot(s) from the template. Neither button has any effect if no slots are selected. Moving and resizing are both done using the template display.

SELECTION

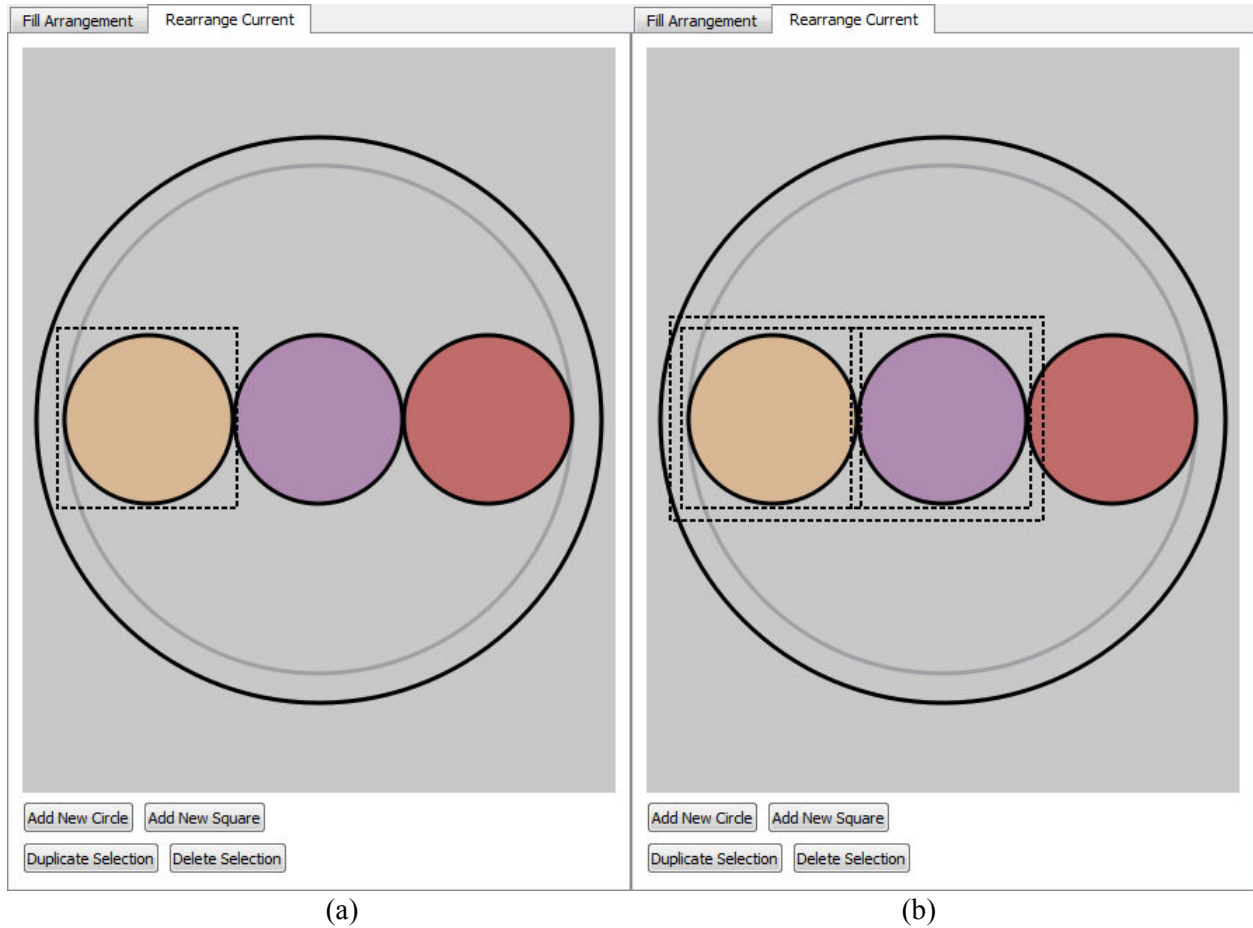


Figure 3: Two views of selection. The left figure has only a single slot selected. The right figure has two, with the additional box bounding the entire selection also visible.

Selection is done by clicking on individual slots in the template. (For this section, all references to the word “click” mean a complete progression of the sequence of actions: mouse button down, little to no movement, mouse button up.) Any click within the shape, not including the black outline around it, counts as clicking on a slot. A slot is visually designated as selected by a black dotted line border in the shape of a square around the outside of the slot (Figure 3a), with a radius slightly larger than the slot itself to ensure the selection boundary will be visible even around square slots in particular. When multiple slots are selected (Figure 3b), an additional selection boundary appears which takes the shape of the smallest rectangle that bounds all selected slots, again enlarged slightly to ensure visibility. When a slot is selected, clicking on a different slot will deselect the current selection and select only the one slot, or

clicking on a space within the display where there are no slots will deselect everything. Multiple slots can be selected by holding either the shift key or the command key (ctrl on Windows, apple on Macs) and clicking on additional slots, which will not deselect other selected slots. While one of these keys is being held down, clicking on empty space will not deselect the current selection, however clicking on a slot in the current selection will remove it from the selection. Only one slot maximum is selected per click, and in the case of overlapping slots, the topmost one will be the one selected.

MOVE

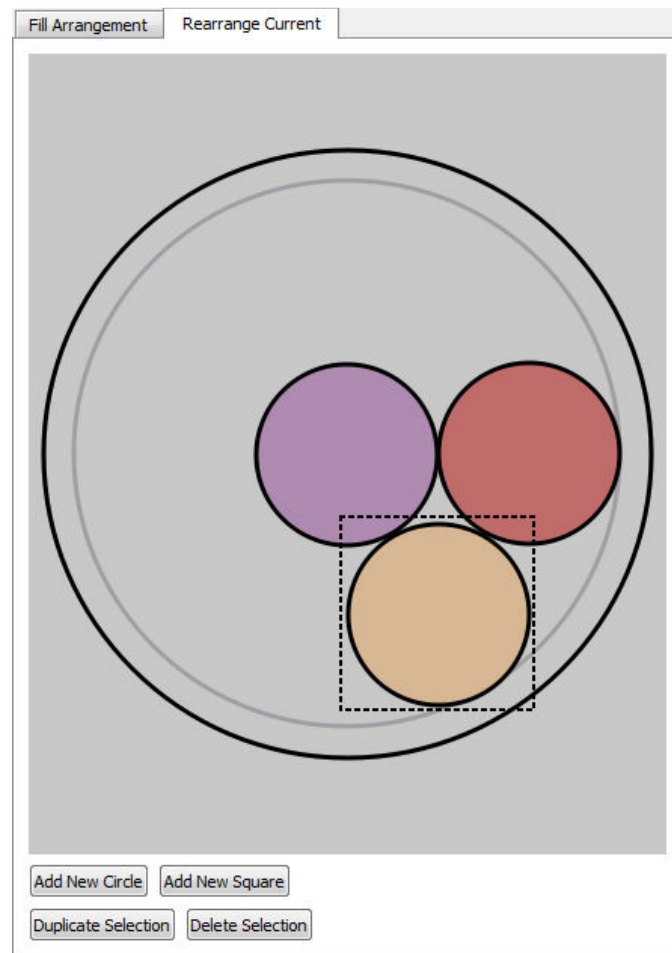


Figure 4: *An example of a moved slot.*

Moving is done by pressing the mouse down on a slot and dragging it then releasing the mouse button. Any drag where the mouse press is made within the selection box, even if it's outside the slot itself, will result in a drag. The exception is when a mouse down occurs on a different slot that isn't selected. If the shift or command key is held, this slot is added to the current selection and all are dragged; if neither key is being held then the new slot will become the only one selected and the only one moved. When performing a mouse down action anywhere within the selection boundary, exempting the previous case, the entire selection will be moved regardless of whether the shift or command key is held. This allows for slots to be rearranged to the user's liking, and large numbers of slots can be moved while still preserving their positions relative to each other.

RESIZE

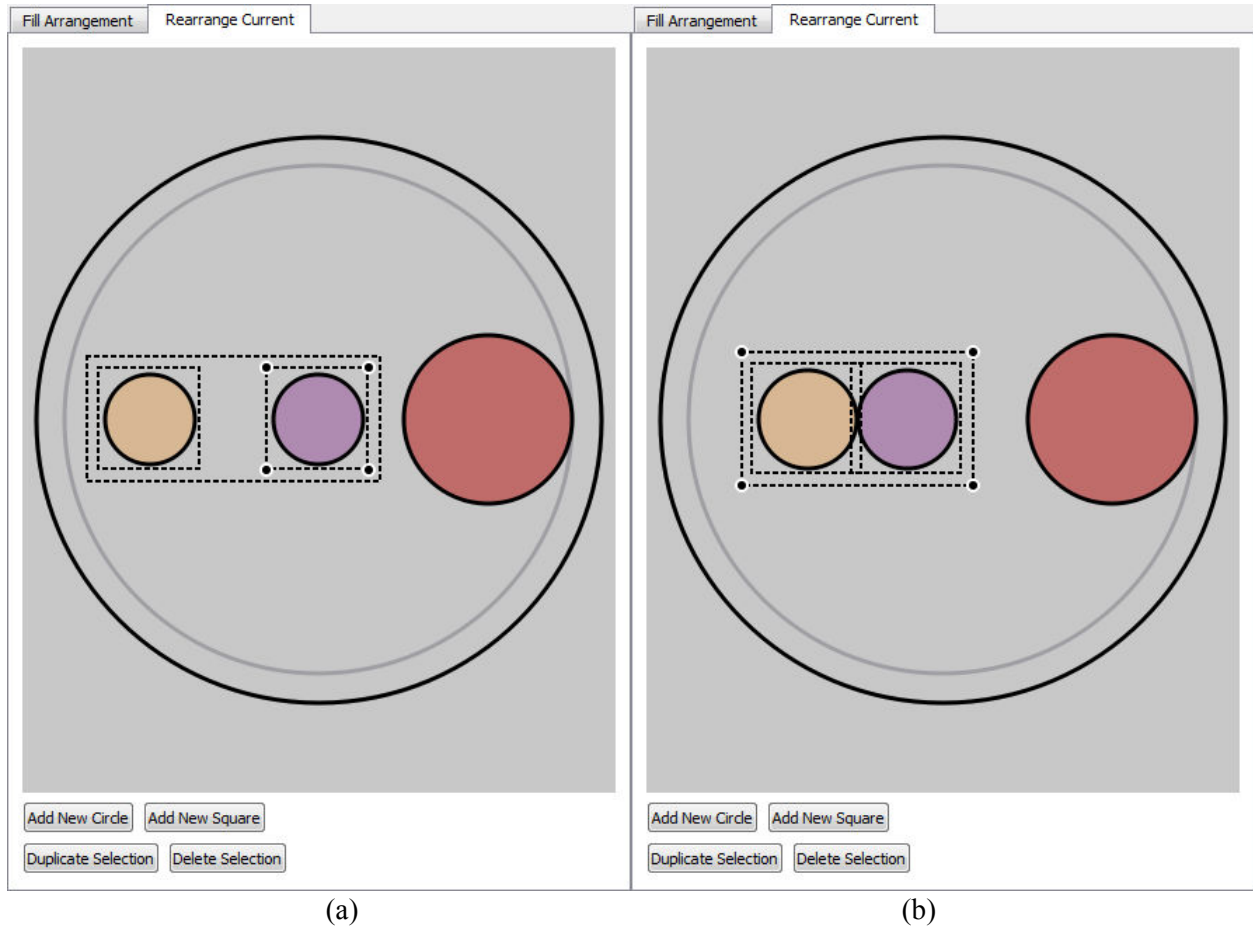


Figure 5: A comparison of two different resizing modes. Local resizing **(a)** scales all selected slots relative to their own centers, and global resizing **(b)** scales all selected slots around the center of the selection. In **(b)** the two slots remain flush against each other, while in **(a)** the purple (middle) slot is still in the center of the cane. The control points used for each resize operation are visible in their respective figures.

Slots can be resized by pressing the mouse down on any corner of the bounding box and dragging it. This functionality is indicated by small black circles with white outlines, selected so as to be impossible to confuse with template slots (which always have black outlines), which will appear at the corners of a selection box whenever the cursor is hovering within it. In the case of a box surrounding multiple selections, the resizing circles appear whenever the cursor is within the larger box but not within any of the smaller slot-specific boxes. These circles only appear on one box at a time, and unlike selection, will favor the bounding boxes of the most recent slot hovered over with no overlap regardless of if the

user hovers over a new slot in front of the previous. In the (physically impossible) case of one spot completely within another (better represented by nested canes), this means that the innermost slot will never be highlighted as resizable while both are selected, but as detailed later, this is irrelevant. During the act of resizing, selections always keep their proportions, and the bounding box remains such that the cursor is always on an edge of the box. There are two different types of resizing behavior, but regardless of which one the user does, all selected slots are scaled.

The first type is local resizing (Figure 5a). This happens when the user resizes slots using a control point on an individual slot's bounding box. All selected slots will be locally scaled by the same proportion, but each slot is scaled around its own center. This means that the slots do not move, only change size. Their behavior is unaffected by any other selection, except for the scale factor from the slot whose control point is being used. For this reason, the previously-mentioned case of the slot within a slot will behave the same regardless of which slot is used to resize if both are selected, meaning the smaller one doesn't need to be used.

The second type of behavior is global resizing (Figure 5b). This happens when the user resizes a selection using a control point on the box that bounds the entire selection rather than just an individual slot. In this case, the entire selection is scaled around the center of the bounding box, causing the center of some or all of the canes to move as the scale changes. This preserves features such as edges that are flush against each other, and the behavior of slots is affected by the position of other slots in the selection, since it all scales around a common center. Local scaling and global scaling are identical when there is only a single slot selected, and because of this, global scaling is unavailable unless the number of selected slots is two or more.

BUTTONS

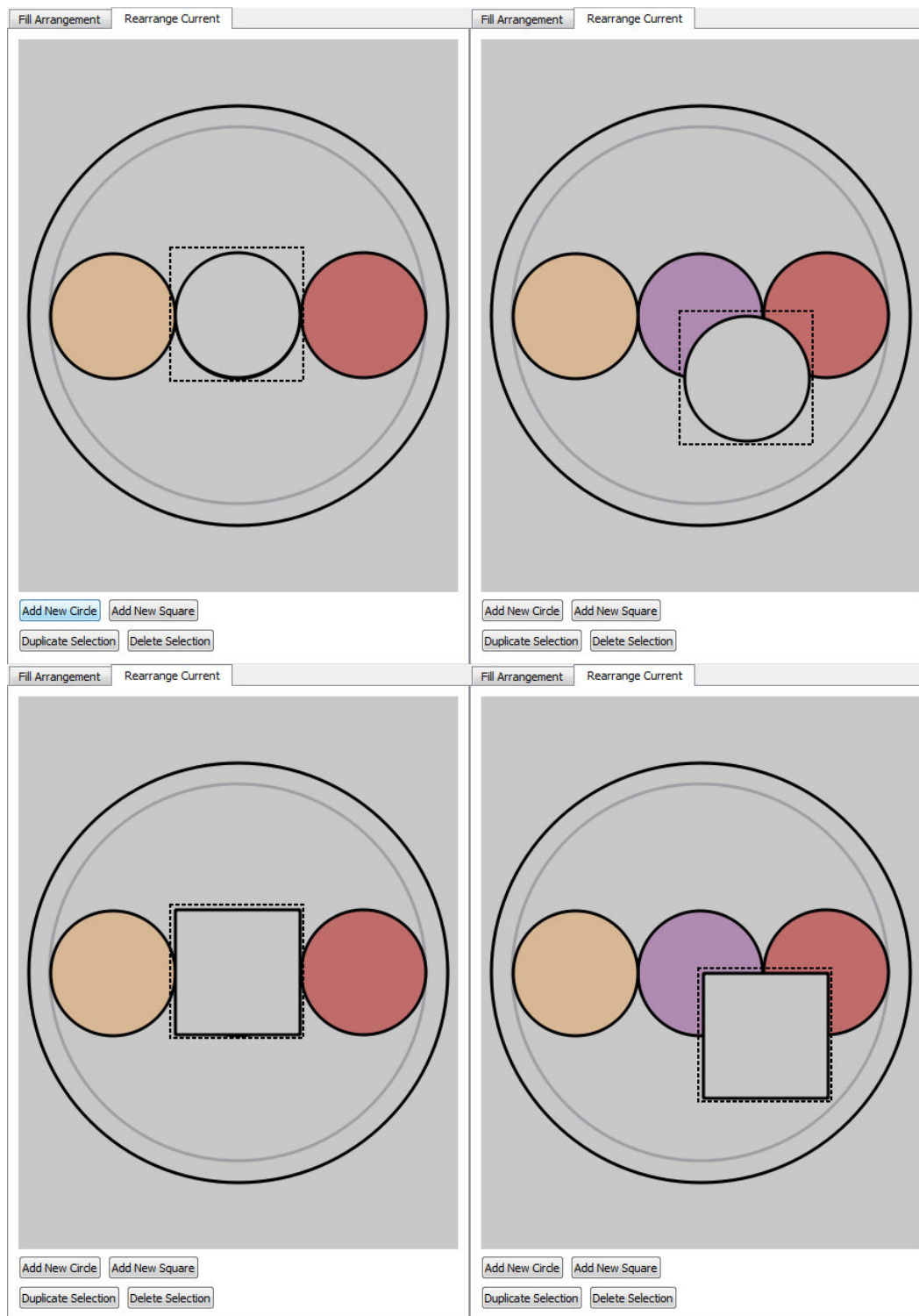


Figure 6: The Add New buttons in action. The top row is what happens when Add New Circle is pressed, and the bottom row is Add New Square. Since slots are placed in the center when first added (left side), they obscure the purple slot already in the center. The pictures on the right side of the figure have the new slots moved to show the older slot still underneath.

In case a user wants to add more slots to the layout than what they currently have, they can use either of the Add New buttons on the bottom of the interface. Using either one of these will deselect the current selection and select only the newly-inserted slot. The slot has a radius which is the average of the radii of all current slots in the template, or half the radius of the innermost casing if there are no other slots in the template. It is placed in the center of the template and on top of the rest of the slots, so it will always be visible unless another new slot is added. Currently buttons are a more convenient interface due to there only being two shapes for slots, however if more shapes are added later to the program then the interface will likely need to change to something more along the lines of a drop-down menu, in order to use less space for a wider variety of options.

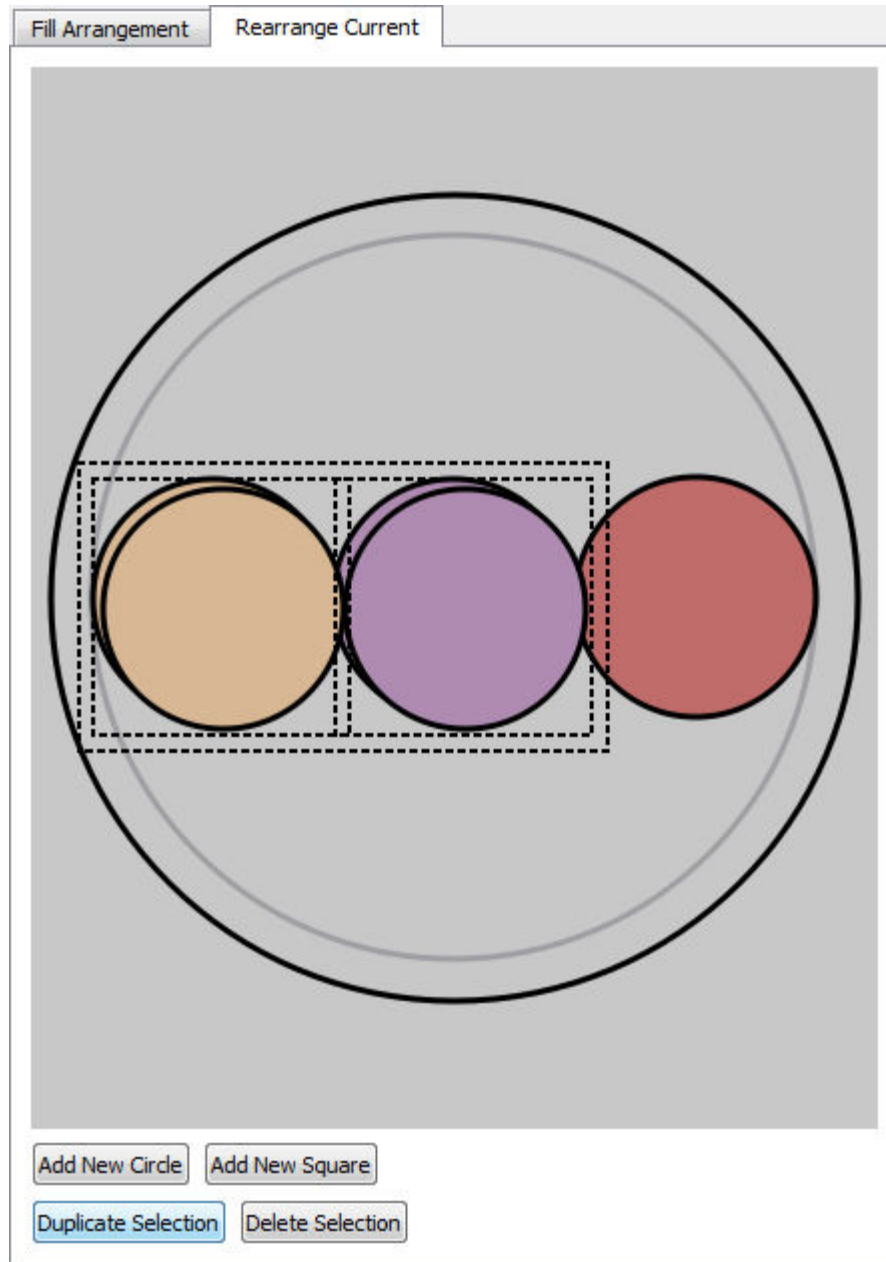


Figure 7: *Immediately after Duplicate Selection is pressed. Prior to this, the orange (left) and purple (middle) slots were selected. Now the two have been copied and offset by the program, and are automatically selected.*

The current selection can be duplicated to facilitate speed and ease in making templates involving identical patterns. Duplication is activated by pressing the Duplicate Selection button while there is at least one slot selected. When the button is pressed, all the currently selected slots are copied and exact duplicates are added to the template, including any colors or canes which are already filling the selection. These duplicates are offset by a small amount to distinguish them from the previous copies and are placed

such that they are on top of all previous slots, but they maintain their relative heights to each other within the duplicated selection. Additionally, the newly-made slots are selected following the duplication, with the original slots no longer selected. This allows the user to quickly manipulate the newly created slots without disturbing the originals.

In order to remove slots that the user may not want, there is a button labeled Delete Selection which will delete the current selection when pressed. Once the slots are deleted, nothing remains selected.

IMPLEMENTATION DETAILS

Virtual Glass is implemented in C++ using the Qt development environment. The main class for the cane interface is a widget that exists in the program on a library stack. This stack changes which widget is visible depending on what has most recently been selected in the library, and the cane interface is visible whenever a cane is the most recent selection. This widget contains the buttons for both the fill interface and customization interface, and switches between the two whenever the selected tab changes. Additionally, at the top of this widget is a nested pair of widgets: these are the “active” areas of the fill interface and the customization interface which display the 2D layout itself. Most of my implementation is in the one of these widgets which is visible when the user selects the “Rearrange Current” interface.

LAYOUT TYPES

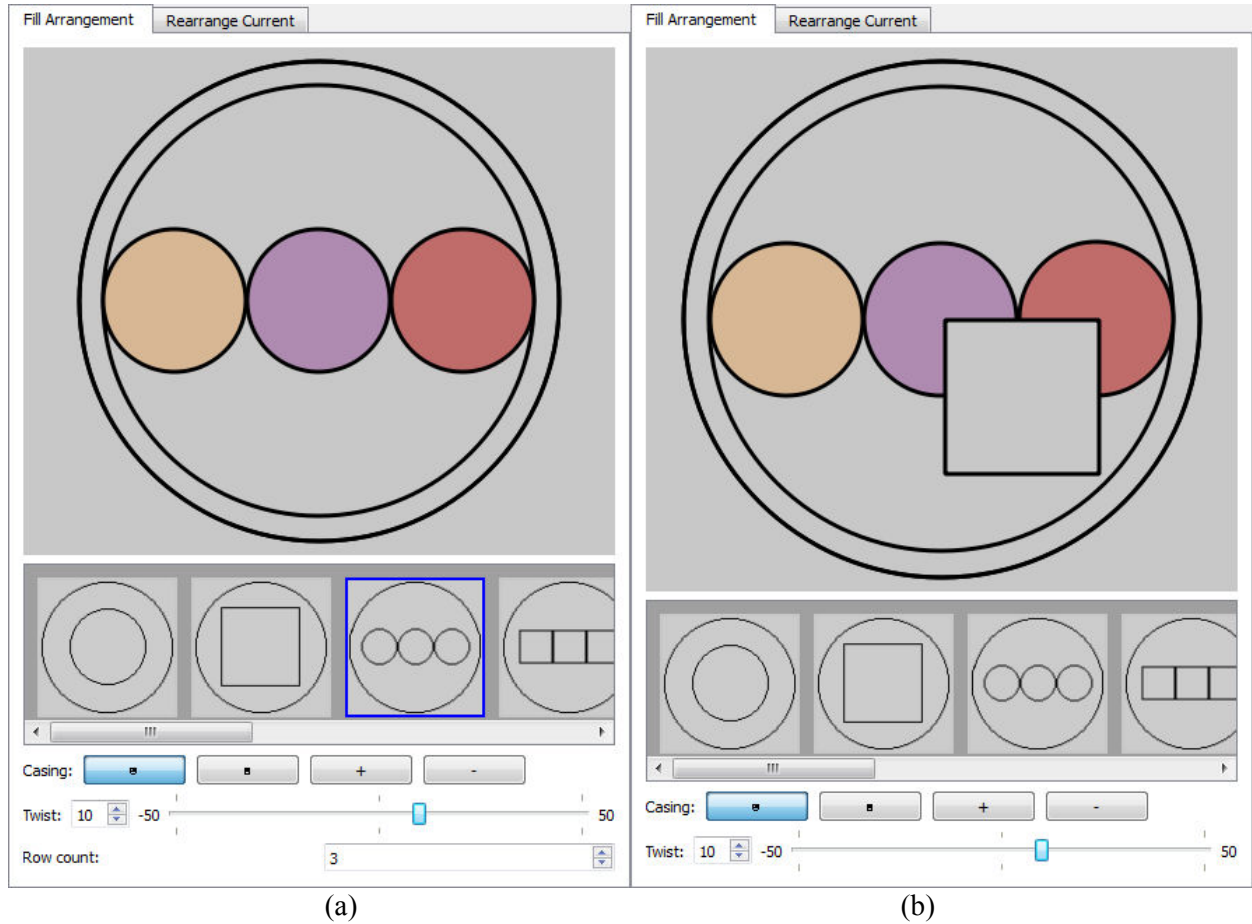


Figure 8: The effect of modifying a basic template (a) and changing it into a customized template (b). The icon is no longer highlighted, and the parameter “Row count” no longer exists.

All pre-defined layouts have different names associated with them. Using just this name and the value of the parameters, these layouts can be procedurally generated without supplying any other data. Customized layouts, however, do not share this property. Because customized layouts can be changed to an infinite number of configurations, it’s impossible to store them as a set of rules. Instead, customized layouts must store the slots themselves as data. This leads to them needing different treatment within the program from non-customized layouts. Most non-customized layouts have parameters to them which define some properties. For example, a layout with identical circular slots lined up next to each other has a parameter that stores how many slots are in the line (Figure 8a). Using just the type of layout and the number in this parameter, the program can generate this layout by calculating the radius of each slot and

the position of their centers. On the other hand, because customized layouts tend to ignore these nice orderly setups, customized layouts do not have parameters. When any layout is modified in the customization interface, it immediately is turned into a customized layout type, and any parameters it previously had are destroyed (Figure 8b). Rather than being regenerated procedurally in the interface, as non-customized layouts are, a customized layout passes along its list of slots as it is modified. There is no way of turning a customized layout back into a non-customized one without starting over, however with future advances to the software it may become possible by modifying saved files outside of the program.

DATA

Slots are stored in a vector in the cane data type itself, and the customization interface has a vector of indexes which it uses to keep track of the selected slots. Ideally, whenever a different cane is selected to be customized, all currently selected slots should be deselected. This can happen in one of two cases. First, the user can select a new layout in the fill interface. Second, the user can select a different layout that was previously created and is stored in the library. The first case is easy to handle, as a “change cane” signal received while the widget is not visible causes the selection vector to be cleared. The second however is not so simple. There is currently no signal that occurs specifically when the object selected in the library changes. Thus, a check which can catch most of these cases looks for if the new cane is not of a customized layout type (in which case it must be a different cane, as there would not be a modified cane signal that occurs within the customization interface which would not also cause the cane to turn into a customized cane layout). This doesn’t catch the case when switching from one customized cane to another within the library while the customization interface is visible in the program, so for this edge case the program checks all indexes in the selection vector. If any of them are greater than or equal to the length of the slots vector, the selection vector is cleared. This does have the potential to cause some strange selection artifacts, however it’s a rather extreme edge case. With this behavior the program no

longer crashes due to index out of bounds exceptions, and so the temporary fix is a reasonable one until the library is modified to send its own unique “selected object changed” signal.

The layout of the cane itself is drawn using a function programmed by Andrew Winslow for the fill arrangement interface which I adopted for the customization interface. Once the basic layout is drawn, the program goes through the selected cane vector to add boxes around all selected slots. For each selection, it draws a square centered on the slot with sides the length of the diameter of the slot, plus twice the length of an extra factor. This factor is 1/100th of the shorter end of the size of the interface as a whole, to a minimum of one pixel, and ensures that the selection box is always distinguishable from the slot. Additionally, if the size of the selection vector is greater than one, then a rectangle is drawn around the boundary of all selections, plus six times the length of the extra factor. This rectangle is calculated by going through all slots in the selection and finding the smallest as well as greatest x and y boundaries.

There are two values that are updated as the user moves their cursor around the screen which help facilitate interaction with the program. One of these values is the index of the slot the user is currently hovering over. This value is used when the user clicks the mouse down in order to quickly determine if the slot being clicked on, if any, is in the current selection, and to take appropriate action with possibly adding or removing it from the selection. The value is set to -1 when not hovering over a slot. The other value is used to determine which selection box should be active. Rather than having a valid value only when directly over a slot, it holds an index whenever the cursor is within a bounding box. Instead of giving priority to the slot which is more “in front”, it doesn’t change value until it leaves the current index value’s bounding box. It is set to -1 when outside the global bounding box, and `INT_MAX` when outside all individual bounding boxes but still within the global bounding box. Aside from these two special values, it never holds a value that cannot be found in the selection vector. When this value is not -1 , the program draws four circles at the corner of the bounding box indicated by the index. These are the circles used to indicate scaling.

MOUSE INPUT

The selection, moving, and scaling functions are all managed together in the mouse down, mouse move, and mouse up event handlers. Upon mouse down, the location of the event is saved, and then a check is performed to see if the scaling function should be activated. If the cursor is within two times the scale factor in both the x and y directions of any corner of the box indicated by the active selection box index, the mode switches to scale mode. Otherwise, the mode switches to move mode, and if the slot under the cursor is not in the selection vector, it's added and a movement flag is set. If the command key was not held down, all other slots are removed from the selection vector are removed. This leaves the selection vector empty if the cursor was not over any slot. If the mouse is moved after being pressed, the program first checks whether it's in move mode or scale mode. If it's in move mode, it checks if the current cursor location is more than a factor distance from where the mouse was pressed. If it is, the program sets a flag that indicates that the mouse has undergone movement. Then for all slots in the selection, it moves their locations by the distance the mouse has moved. If instead the program is in scale mode, it sets the same flag for mouse movement and checks if the active bounding box is the global box or an individual. If it's an individual, the program creates a proportion value which determines the factor between the previous size of the individual slot and the new size according to how much the user scaled it, and then multiplies the radii of all slots in the selection by that proportion. If instead the global bounding box is being used, the program calculates the same sort of proportion value, but instead of only scaling the radii, the selected slots also have their centers moved closer or farther away from the center of the global bounding box. Finally comes the mouse up. If the flag for mouse movement hasn't been triggered and a command key is being held down, the slot under the cursor is removed from the selection. If there isn't a command key signal, the selection is cleared and the current slot alone is added to the selection.

SLOT ORDER

In the vector of slots in a cane, slots that appear first in the vector will be drawn on “top” of the ones later than them. This means that when a new slot is added to a cane, such as when “New Circle Cane” is pressed, the created slot must be inserted at the beginning of the vector in order for it to be in front. At this point, the selection vector is cleared and given a single value of 0, since the new slot is at the beginning in index position 0. When duplicating the current selection, the same must be kept in mind. In order to make sure the duplicated slots are still distinct from the original slots, they need to be made by first constructing new slots using the properties of the old ones rather than doing a simple copy operation. By first sorting the list of indexes and then making duplicates of the old slots in a new vector, which is then inserted into the beginning of the original vector, the now-duplicated slots maintain their original depths relative to each other while still being at the front of the overall layout. The selection vector can then simply be filled with the integers 0 through $n - 1$, where n is the number of duplicated slots. The program also adds an x and a y offset to the position of each of the new slots in order to visually distinguish them from the old.

The only trick to implementing deleting slots is remembering that once a slot has been removed from a list, all slots after it now have an index one smaller than previously. In the customization interface, once an index has been processed and the equivalent cane deleted, all indexes later in the vector with a higher number are decremented by one. However another option would be sorting the selection list in descending order, such that no slot would be deleted once an earlier one had already been removed.

COMPARISON

The customization interface allows for a much wider range of possibilities when it comes to designing cane than strictly adhering to the provided templates. Here I will provide some examples of cane already made by glass artists, which I have attempted to replicate both with and without use of the customization interface. I will also include a more original design which is difficult if not impossible to design using layouts.

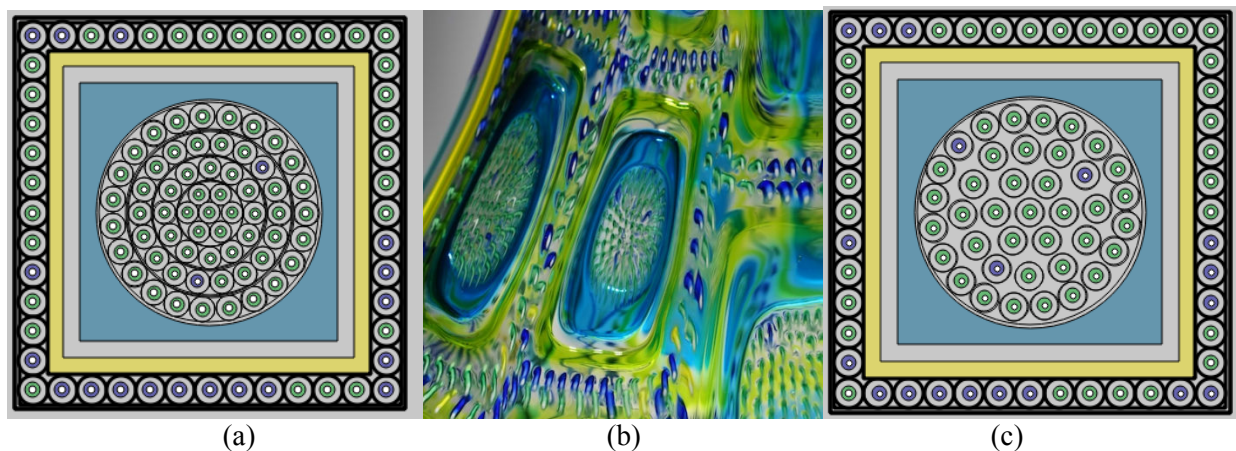


Figure 9: Two separate attempts to replicate a cross section of a cane designed by David Patchen (<http://www.davidpatchen.com/portfolio/resistenza>). The version on the left was made using only the fill interface, while the version on the right was made using the customization interface as well.

Figure 9b features a glass piece made out of murrini: short sections of cane arranged from straight on rather than lengthwise. To either side are attempts to reverse-engineer this cane using the Virtual Glass software. Figure 9a is a layout designed using only existing parameterized templates, and Figure 9c is a layout which was created using both templates and customization.

For most of the cane, the templates work just fine. The entire outside portions from the first square outwards are designed exactly the same for both layouts, with only slight differences due to user subjectivity. In this example, the inside is the key difference. There is no template for placing a large bundle of cane in a cluster, as is what happens in this layout, though the effect is somewhat replicable. One important difference to note is the prevalence of the black circles dividing the different rings of cane

in the layout: these signify that each circle is in fact its own layout surrounding those inside of it. What happens here is a mismatch with reality, rather dangerous when working with a program which is supposed to simulate a process. With the template layout, the process being described is as follows: take a small bundle of cane and fuse it, then surround it with a circle of cane and fuse it, then surround it with another circle of cane and fuse it, and then do it once more, and only after the cane has been fused four times already is it coated in blue. What is most likely a much closer representation of what happened in reality is in fact what is visible in Figure 9c: a whole bundle of cane is fused together all at once and then coated in blue. Visually the two may produce a similar effect, but when it comes to imagining how a cane will be formed, a user may want to create something which follows the actual process more accurately. Additionally, the customized layout uses less than half the number of nested layouts overall, making it much simpler to keep track of.

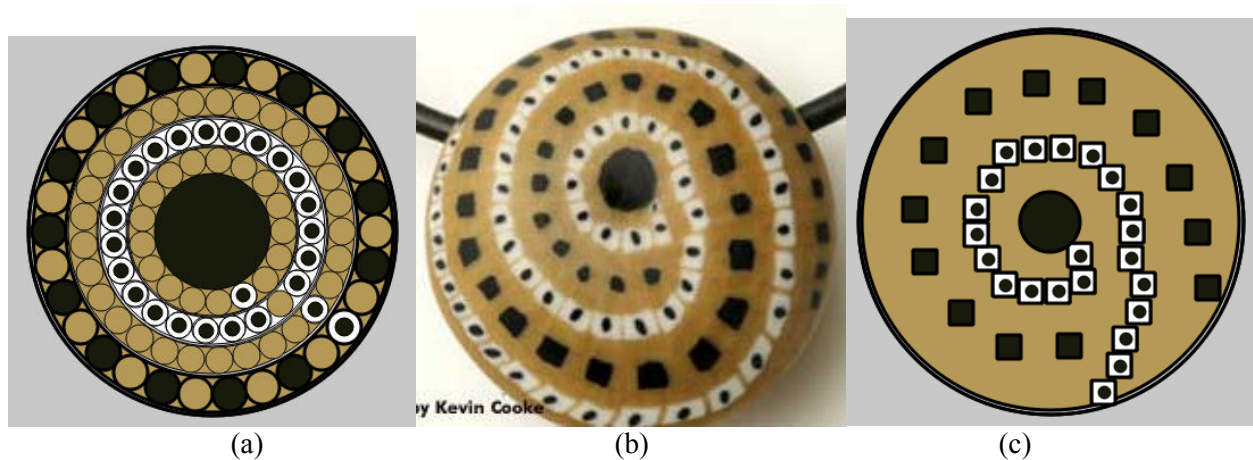


Figure 10: *Two separate attempts to replicate a murrini designed by Milissa Montini (<http://www.murrini.com/jewelry.htm>). The left attempt uses only the fill interface, while the right was made using the customization interface as well.*

In this case, the fill interface does a poor job of capturing the intended feel of the layout. This design is a multi-layer array of square cane spiraled around a center. The fill interface performs poorly when it comes to designs which are not symmetrical in layout, such as a spiral. The best it can do, as shown, is break the spiral down into concentric circles, which are then filled in such a way to attempt to evoke the impression of the spiral. The square shapes present in the original design are entirely absent;

they cannot be included due to the lack of a template with squares surrounding a circle. Using such a layout may make the user entirely miss what they were going for, and instead pack their spiral design into rings. The customize template allows the user to make their own spiral pattern and visualize their design with the square shape intended for the final piece. Additionally, not needing to fill in the brown space with more circles means less of a burden on the Virtual Glass program and the user's computer when rendering the entire cane.

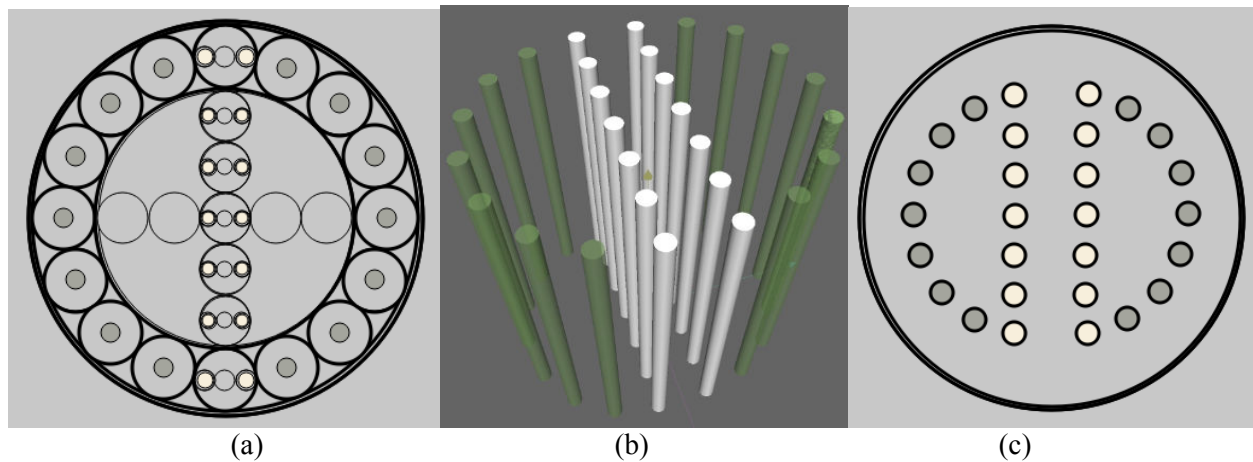


Figure 11: Two separate attempts to replicate a cane layout made by Erik and Martin Demaine using the first version of Virtual Glass. The layout on the left was made using only the fill interface, while the layout on the right was made using the customization interface as well.

Here I present a cane that has not been brought to life, but was instead created by Erik and Martin Demaine using the Inkscape importing functionality of the first version of Virtual Glass. It is an outline of two semi-circles facing each other and separated somewhat. The customization interface produced a layout (Figure 11c) that matches the original very closely, with the canes arranged as they are in the example. The fill interface alone made it much more difficult to replicate the design. The two parallel lines of the design are created by nesting a layout of two separated canes into a single line, which itself is then nested inside a circle which forms the outside of the semi-circles.

These examples highlight the stifling of creativity that using only the fill interface in Virtual Glass can cause. The templates are designed to match common cane patterns, but what we want with Virtual Glass is for glassblowers to go beyond the common and come up with new things. With enough

effort, it's possible to replicate all sorts of designs using templates, but most of them involve working *around* the templates rather than *with* them. The templates certainly have their uses: they are designs that most glassblowers are familiar with and know how to make. But when it comes to designing something beyond the scope of what's usually done, we want users to be able to break the mold just as easily as they can follow it. For that, the customization interface allows users to take what they know and arrange it in a way they never thought to visualize before.

IDEAS FOR IMPROVEMENT

There are still many more features that can be added to the customization interface, however some of them would require changes to the code which would prove difficult to implement without a long period of coding or major restructures to some parts of the code. Many of the features implemented in the customization interface were inspired by a user study of the Virtual Glass software conducted by Erik Demaine and Andrew Winslow. At the point of the user study, the ideas for how customization would be done were just beginning to be formed, and with the study the Virtual Glass team formed a clear direction for the direction we wanted the interface to go in, which I then took, developed, and implemented. Some of the features described here were suggested by the subjects, while others mimic features implemented in the previous version of Virtual Glass, and some were ideas put forth by the team as the software and the interface itself developed.

The feature which is most notably missing from the interface is an undo button. In this sort of an interface, the ability to undo is essential. Unlike the rest of the program, operations performed in the customization interface are not easily replicated, therefore manually returning to a previous state quickly becomes a difficult if not impossible operation. It is for this reason that the lack of an undo button is not a deliberate omission, but rather a yet-unimplemented feature which is planned for inclusion at a later point. Once the undo and redo functionality is implemented, these two buttons should be added into the

interface in a prominent location. It would be most useful to make them side by side above all the other buttons and give them vertical size of twice that of the other buttons. This puts them between the layout, where the user directly manipulates slots, and the buttons, where the user can do more complicated operations. In this location, they are easily accessible no matter what the user wishes to reverse, and with a redo button the undo operation itself is also reversible, making it permissible to press the undo button by accident. Undo and redo cannot be implemented until the Virtual Glass team decides how cane layouts are going to be saved. As that has not been finalized yet, undo is a function that is still waiting to be added.

Squares and circles are not the only shapes that cane can take. When arranged into a hexagonal grid and pressed together, canes will predictably morph into hexagons, making this a valid shape to create more layouts out of. Other bundling operations may also squish canes into various different sorts of shapes. Though a very desirable feature, there are several difficulties with including more shapes beyond circles and squares.

The simplest problem which nonetheless should be taken into account is drawing such a shape in the interface. The development environment of Qt provided easy functions for drawing ellipses and rectangles, making circles and squares very simple to create. Any other sort of shape is much less trivial. Polygons are still somewhat easy: they can be specified using a list of points, which in the case of regular shapes, can be done with a bit of math. It gets trickier, but still reasonable, for non-regular shapes such as trapezoids. Once a shape is included that doesn't have straight lines, drawing becomes much harder because a different class needs to be used in order to reliably draw and fill the shape properly. With some restructuring of the code, all shapes could be specified using this alternate class, the QPainterPath class.

Another problem becomes determining when a user is interacting with a shape. This involves identifying two things for any given shape: is the cursor within certain proximity of the shape's bounding box, used for the resize operation, and is the cursor within the shape itself, used for operations such as moving. For circles and squares, this is easier. A cursor is in the circle if its distance to the circle's center

is less than or equal to the radius. A cursor is inside the square if its distance to the square's center in the x direction as well as the distance to the square's center in the y direction are both less than or equal to the "radius". Both have simple, square bounding boxes. With regular polygons other than a square, the bounding box is still easy to determine but collision detection becomes more difficult since the math is not as trivial. Once curves and non-convex shapes are introduced, determining through math alone the bounding box and collision area becomes extremely difficult; however the QPainterPath class has functions for both collision detection and the bounding box, making it even more reasonable as a substitution for the current implementation. It even removes the necessity of a switch statement for determining just how to calculate whether or not the cursor collides with a given shape, as all shapes can use the exact same code.

The largest problem involves the rest of the code working around the customization interface. While switching to QPainterPath solves two problems in the customization interface, it would cause difficulty in other parts of the program. In order to properly render shapes in 3D, a polygon mesh needs to be constructed. With circles and squares, these are easy to create, but if everything were instead an arbitrary shape construction becomes much more difficult. The save format would need to include some specification for how a shape is constructed, especially if users can create and use their own shapes somehow, which would be the ideal eventual result. Even the customization interface would not work completely smoothly with the QPainterPath. Although there is a translate function, I could not find one which would scale a shape, implying that each control point in the shape would need to be individually changed in order to make the entire shape larger or smaller in the interface.

The visual interface would be able to handle more shapes with relative ease, with a few conditions. The customization interface currently allows the user to add new canes in circle or square shape. If more or arbitrary shapes were added, this can change to be a single button paired with a dropdown menu. The user would select the shape they wish to add from a list of available shapes and then click the button to add an instance of that shape to the interface. The development team would have to

decide whether or not they want the outer shape of layouts to be able to take on different shapes; if so, there is another space consideration in the casing buttons on the fill interface. Overall, the switch to a new representation is not impossible, but implementation would require a larger scale of work than the scope of this project.

In order to align designs differently, a user may want to rotate different slots. Even rotating a circle about its center can be a difficult operation when one considers that different layouts can be nested in the slots of other layouts. Currently, there is no support for rotation. For squares, it would be impossible in increments smaller than 90 degrees since squares are drawn with a specific rectangle-drawing function. For the same reason, circles would also need to be constrained to 90 degrees to allow for the case of a square shape nested inside a circle. Realistically, this limitation would need to be overcome in order to make rotation a reasonable option. Allowing only 90 degree rotation would not allow a truly interactive setup, as users would not be able to watch slots move while a theoretical rotation is in progress until they passed some arbitrary tipping point, at which point the slot would rotate.

Rotation also has the potential to introduce strange artifacts depending on how it is coded. When doing global scaling on a group of slots, the scaling operation occurs around the center of the bounding box. For rotation however (assuming the previous problem of being locked into only 90 degree rotations has been solved) consider a situation where the group consists of one square and one circle. As the circle rotates, its bounding box remains the same, but as the square rotates, its bounding box grows and shrinks. This would cause the center of the larger bounding box to move as the two slots were rotated. Taking into account only the centers of the slots would avoid this problem but needs to be accounted for, as current global operations are based on the bounding box itself and not the shapes within.

In the physical world, canes do not overlap, but Virtual Glass currently has no functionality to stop this. Programming in such a feature would cause the software to produce more realistic cane. Previous attempts were made to include this functionality which has since been removed. The way this

was previously done was every time a slot was manipulated, the program would check all other slots to see if any of them would have any overlap with the slot that was currently being modified, and if so, it wouldn't allow the change to go through. This was not a good implementation for several reasons. In the case of wanting to move slots around in multiple steps, it might be difficult to perform a set of actions which might otherwise be easy if the user was allowed to move a slot to an "invalid" position, but would later perform another operation which would made the result "valid". Additionally, this provided poor feedback to the user. There was no clear difference between being unable to move the slot because the position was invalid and being unable to move the slot because the program was freezing up. Finally, as the number of slots on a layout increased, the time it took to check each movement's validity increased, eventually causing noticeable lag in the program's response. This last problem is not strictly caused by the implementation's behavior, but is nonetheless something to be accounted for when re-adding collision detection functionality. A better solution than the previous implementation would be: rather than forbidding the user to make an invalid operation, it would be much better to simply visually indicate that there is an invalid set of positions, leaving it to the user to self-police their own layouts without the program mandating that they follow reality. This could be done by turning the shape of the slot a different color, perhaps red, when there is a collision to alert the user to this event.

Once collision detection is operational, an additional feature to add would be snapping. This would occur when a user drags one object close to another, and the object currently being dragged would jump to the location where it is closest to the position it was dragged to while still being flush with the nearby object. It should be possible to toggle this functionality, and ideally the radius in which it occurs as well. This would be nearly impossible to include without some form of working collision detection, making it a lower priority to implement than that. Rather than activating when two objects overlap, it would trigger when two objects have sides parallel to each other which are less than or equal to a certain distance away, and the smallest vector between these two sides would be added to the moving object's offset. This functionality could also exist for the resizing operation.

Another snapping functionality to include would be a self-constructing grid. This would take the slots currently in a layout and use their centers to create a grid, which other slots would then snap to. This could be toggled between triangular or rectangular shapes at the discretion of the user, or turned off entirely. The distance between the different “lines” of the grid would be determined by the distance between the centers of the existing slots, or if there were only one slot present, by using its size as well. This grid could be indicated using lines of a different color, perhaps dotted and white, drawn beneath the slots to provide the user with visual feedback that doesn’t obscure their view of the design as a whole.

DISCUSSION

Virtual Glass is a program designed to bring computer aided design to the art of glass cane. The goal of Virtual Glass is to be a robust program for designing and visualizing the cane pulling process from start to finish, giving users as much freedom as possible while still remaining easy to use. To this end, the layout designing interface based solely on template layouts, while a good base, proved to be not enough freedom for many users. Thus, in collaboration with the Virtual Glass team, I have built the customization interface to work smoothly alongside the fill interface to allow users a much wider range of options when it comes to building their layouts.

The primary focus in developing this interface was to make sure it was learnable by minimizing hidden functionality. All buttons on the interface have their own singular tasks which provide visual feedback when activated. Text aids the user in discovering selection, and in selection the indicator for resizing functionality is also revealed. The only mode change not explicitly visually indicated is the option of holding shift or the command key to select multiple canes, a feature which is still readily discoverable due to the same function being so prevalent in many other computer programs today. Even without accessing such functionality, the user is limited only in efficiency; they are still able to do the same operations in longer time. Possibility for confusion between the fill interface and the customization

interface is low: the fill interface prominently features a variety of templates to choose from which are notably absent on the customization interface, while the customization interface features the layout more prominently.

The interface design focused on learnability because the target audience of Virtual Glass is not necessarily expert computer users, but instead people who are skilled at glassblowing. Efficiency is not a huge concern because the only foreseeable time constraint is the user's own patience. Glassblowing itself is a complicated multistep process, so it's reasonable to assume that users of Virtual Glass are accustomed to keeping a design in mind for extended periods of time while they build up towards the final result. As long as they aren't distracted with other issues—i.e. learning how to use the software—taking a little longer to create their final result is greatly preferable to needing to devote more mental energy to discovering or remembering how everything works. Currently the user's ability to deal with mistakes in the customization interface is somewhat poor, but with the future implementation of undo and redo, it will be greatly improved.

I have designed this customization interface based on the feedback from Erik Demaine and Andrew Winslow's user study, along with input from the entire Virtual Glass team. This piece of the program is the part that combines the freedom of the first version of Virtual Glass with the usability of the second version. It is unlike any other part of the program that exists thus far, and may even pave the way for more customization options in different areas of the program in the future.

RESOURCES

Barr, W. E. and Anhorn, Victor J. *Scientific and Industrial Glass Blowing and Laboratory Techniques*. Pittsburgh, PA: Instruments Publishing Company, 1949.

Bumbletech, “iGlass.” 2009: <<http://www.bumbletech.com/w/>> (January 2012).

Children’s Museum of Indianapolis, “FireWorks of Glass: Games and Activities.” <http://tcm.childrensmuseum.org/themuseum/fireworks_ofglass/games_35.htm> (January 2012).

Giannopapa, C. G. “Development of a Computer Simulation Model for Blowing Glass Containers”. *Journal of Manufacturing Science and Engineering: Transactions of the ASME* Vol. 130, No. 4 (2008).

Ingle, Henry W., “Glass Blowing Machine and Method.” *United States Patent Office*. No. 1843159 (1924).

Kaptsan, Igal, “Glass Blowing in the virtual world.” 2010: <<http://www.youtube.com/watch?v=8hLQEIZmkQ8>> (January 2012).

Marechal, C., Moreau, P., and Locheignies, D. “Numerical optimization of a new robotized glass blowing process.” *Engineering with Computers* 19 (2004): 233-240.

Montini, Milissa, “Murrini by Milissa Montini.” <<http://www.murrini.com/index.htm>> (August 2012).

Museum of Glass, “School by Fire.” 2002: <<http://205.139.108.204/document.doc?id=142>> (January 2012).

Patchen, David, “Hand Blown Glass.” <<http://www.davidpatchen.com/>> (January 2012).

Winslow, A., Baldauf, K., Lee, B., McCann, J., Demaine, E., Demaine, M., and Houk, P. “Virtual Cane Creation for Glassblowers”, Studio Talk, 39th International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH 2012), August 2012.