

The SURPhACE

“The SURPhACE” (standing for “Simulation Utilizing Real-time Physics and Collision Engine”) is a vector-based graphics editing framework complete with networking functionality and Physics algorithms that allow users to drag, transform and push their various drawings across the canvas. Users can connect to one another over an online network in order to view the same drawing canvas and they can collaboratively draw objects and move them across the screen. Drawn objects will collide and interact with each other as if they were solid objects being pushed around a table. The SURPhACE will be valuable for various types of long-distance, collaborative conceptual planning, but will also boast an advantage over other network enabled drawing programs by utilizing basic physics to combine the best elements of a digital interface with the most beneficial aspects of a physical interface. One surely desires the ability to copy and paste in physical realm as much as the ability to interact with digital drawing software in a tangible and believable way.

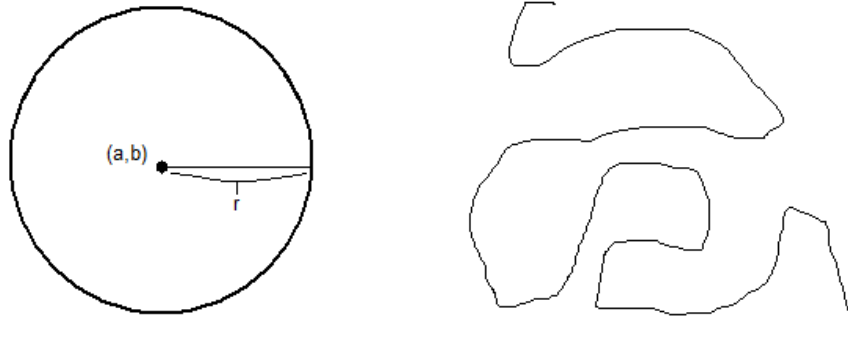
As it turns out, the creation of such a program brings up some intriguing issues regarding coding, geometry and real-world physical mechanics that must be considered in order for “The SURPhACE” to be successful. Not all such problems have been resolved, but all of them are being considered. My paper will outline everything involved in getting the SURPhACE to its current state, the project’s current features and capabilities, and the concerns and plans for the future of the project. For anyone who would like to see the source code, [my page](#) has a downloadable .zip file under “Files” for each major step (and some minor steps) that the SURPhACE went through.

Defining the Initial Goal – Squiggle Geometry

The program is a collaborative effort between myself and James Hoyt and began as a testing ground for additional features that could potentially be added to a different project called the Edge Table. Forrest Marvez and Lorin Petersen were working on the Edge Table for the company MITRE who desired a quick and efficient method of holding collaborative planning sessions over long-distances. The Edge Table was thus designed to be a multi-touch and network enabled drawing tool. In early September, the Edge Table was in an early state and so lacked some of the desirable features of other drawing software. Freehand drawings remained static and could not be moved or altered in any way save for being erased. Programs such as Adobe Flash, on the other hand, allow users to draw complex shapes and then move, scale and rotate them as individual objects. Such a feature would be advantageous for collaborative and technical planning, allowing users to adjust drawn objects as needed. Therefore, I took on the role of looking into how to emulate such features into a drawing tool within the framework used by the Edge Table, XNA 4.0, and its associated programming language C#.

Given the technical nature of the [MITRE](#) organization, I felt that using vector-based graphics would be the best approach. As opposed to raster graphics which considers an image as a grid of pixels, vector graphics uses mathematical expressions to define a set of lines, points and other primitive geometry in order to build an image. One might be familiar with the formula for a circle: $r^2 = (x - a)^2 + (y - b)^2$ with 'r' being the radius of the circle, 'a' being the x coordinate of the center of the circle and 'b' being the y coordinate of the center. Within a program one can

thus define a circle simply with a center point and a radius. However, when considering the context of a drawing tool, the question becomes how to represent a complex hand-drawn piece of geometry.



Left: A circle can be represented via $r^2 = (x - a)^2 + (y - b)^2$ **Right:** most free-hand scribbles are more intricate and are difficult to define by such a simple formula.

The simple solution is to define user-drawn shapes as a series of two-dimensional points each with an x and a y coordinate. XNA fortunately has a built-in `Vector2` class which is used to hold an x and y position. Therefore, the SURPhACE keeps track of the user's mouse position while they are drawing and periodically puts the mouse's x and y coordinates into a `Vector2` object and then adds each `Vector2` into data structure which will come to represent the drawn shape. Once we have a data structure consisting of a number of points defined as `Vector2`'s, our program goes through the series of points and connects each one to the next with a line (effectively playing a game of "connect the dots") until the end of the data structure is reached. With the complex shape being represented by a series of two-dimensional positions, we can allow the user to move and alter the object by applying basic pre-defined math formulas to each of these x and y positions. The new coordinates of each `Vector2` position after a rotation can be found thusly:

"Rotating a point from position (o.x, o.y) to position (oR.x, oR.y) through an angle "theta" about pivot point (oP.x, oP.y)

$$oR.x = oP.x + (o.x - oP.x) * \cos(\text{theta}) - (o.y - oP.y) * \sin(\text{theta})$$
$$oR.y = oP.y + (o.x - oP.x) * \sin(\text{theta}) + (o.y - oP.y) * \cos(\text{theta})"$$

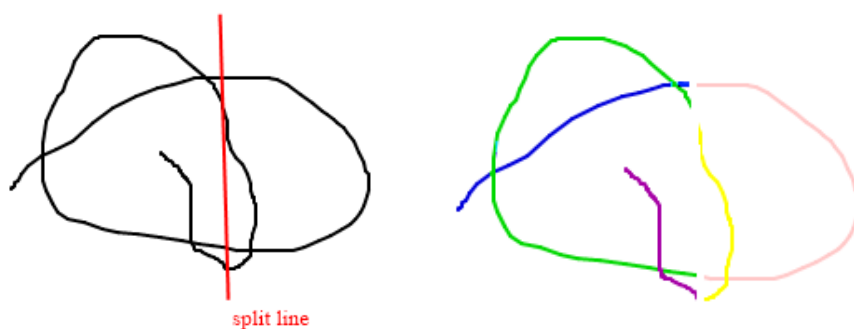
Source: Satish Goda (<http://www.openprocessing.org/visuals/?visualID=6782>)

The formulas for scaling and dragging the objects are even simpler. Scaling is performed by subtracting the center of the object from each Vector2, multiplying the x and y coordinates of each Vector 2 by a desired value (greater than 1 for growing and between 0 and 1 for shrinking) and then re-adding the center of the object to each Vector2. The reason we subtract and then re-add the center of the object (which can be found by averaging minimum and maximum Vector2's found in the data structure) is to ensure the object does not move towards or away from the origin (in our case the top-left corner of the canvas) while being scaled. Dragging is done simply by adding or subtracting values to each x and y position in the data structure depending on where we want the object to move. After each position in the data structure is changed appropriately, the object is redrawn, looping through the new coordinates. By implementing user-drawn shapes as a series of x and y coordinates along with basic transformation formulas, the SURPhACE allows the user to draw any shape they desire and drag, rotate, flip and scale them as needed.

Problem: What Kind of Data Structure?

A List is a built-in linear data structure available for use in C#. The structure functions as its name would suggest; the structure is essentially a laundry list of objects, in our case Vector2's that hold an x and y position. Although a simple List of Vector2's will work to define a squiggle

for the simple features mentioned before, such a structure causes problems once more complex features such as splitting are added. Now that we have drawings as individual objects the crucial eraser tool must be viewed in a different way. If one erases their object straight down the middle they should really now have two separate objects. In order to allow for a splitting feature one answer is to find the point at which the user's mouse intersects their drawing and then take every position in the list before the point of intersection and every position in the list after the point of intersection and place them into separate lists. A demo illustrating this type of implementation can be seen [here](#). While that approach works splendidly for drawings that do not intersect themselves, the fact that our shapes are drawn in a linear fashion from one point in the list to the next causes a unique problem for more complex drawings that intersect themselves once or many times. Rather than breaking the object into two clean pieces, the object is split in two each time the mouse intersects the drawing and thus numerous fragments are output as separate objects. The diagram below illustrates:



Top: A non-intersecting scribble is split successfully. The green and blue represent the two new objects.

Bottom: A scribble which intersects itself will break into many different objects due to the linear nature of the List data structure.

Although the problem may not seem very critical, the addition of physical interaction between drawn objects will cause more serious problems when self-intersecting shapes such as an '8' are split down the middle and form many intersecting objects as a result. Therefore,

finding a new data structure in which to store a drawing's Vector2's becomes necessary.

However, before such a structure can be found one must understand the nature of the problem being caused by the implementation of physics into the SURPhACE.

Adding Physics Adds a New Set of Problems

High school mechanical physics can be rather simple, but one must realize that such a simplification of real-world physics is not an accurate representation of actual physical behavior. Most problems found on a high school physics test neglect factors such as air-resistance and terminal velocity. They assume everything takes place in a vacuum, but in the everyday world this is almost never the case. Such questions are customized for what is necessary in the context of the classroom. Extraneous details are put aside for the sake of teaching students the general principles of mechanical physics. Similarly, real-world physics must be modified and adapted to fit within the context of the SURPhACE. Realistically, if one were to drag objects around an actual table, the interaction of the objects in question would be affected by factors such as friction and the angle at which an object bumps into another object. If an object were dragged into another off-center, the moving object would apply a rotation to the bumped object as well as a directional push. However, after some discussion, James and I realized that within most contexts in which the SURPhACE would be used, the user is likely to prefer that the act of pushing one drawing object into another will simply move both objects in the same direction. Cumbersome behavior such as unintentional rotation will annoy more than impress. To ignore such needs is to take the disadvantages of a physical interface along with its advantages which is

contrary to goal merging the best of the digital and physical realms. Therefore, factors such as friction, torque and rotational velocity need not be applied and simple pushing is the most practical implementation.

While collision resolution may be simplified, actual collisions detection is a less trivial affair considering the complexity of most hand-drawn shapes. A single bounding box encapsulating the entire drawing is far from accurate enough for collisions to feel accurate and natural. Each drawn object in the SURPhACE contains a list of numerous bounding boxes that are positioned along the contours of the shape. However, with many drawings on the canvas each having a large number of small bounding boxes, constant collision detections involving checking *all* of these boxes would be extremely processor intensive. The SURPhACE solves this optimization issue by utilizing a bounding box hierarchy that consists of a larger bounding box encompassing the shape's smaller bounding boxes. Smaller bounding boxes are not considered unless there are two large bounding boxes intersecting one another. Therefore the number of unnecessary bounding box intersection checks is greatly reduced, increasing the SURPhACE's performance.

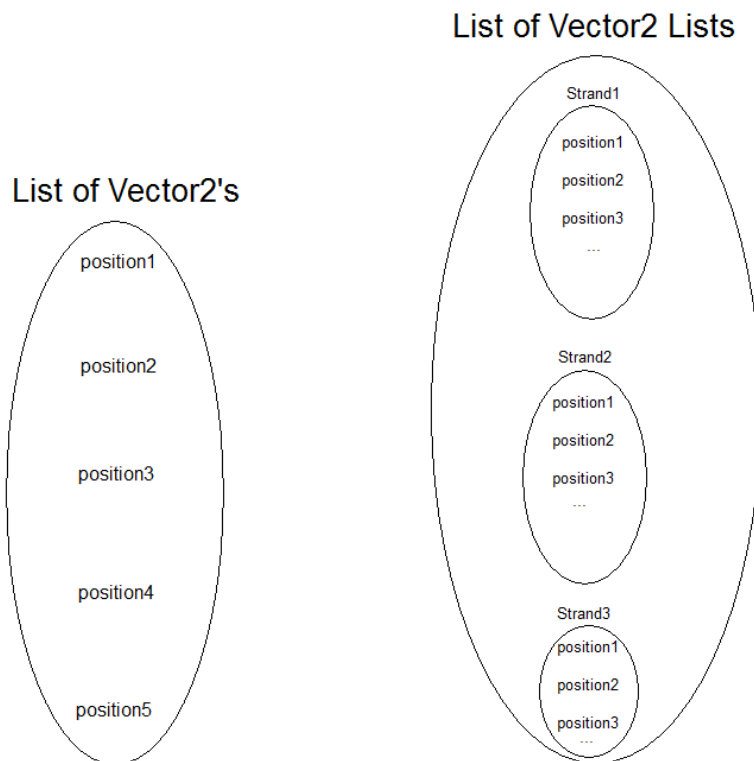


The smaller bounding boxes representing these two shapes (shown in a tan color) are not checked against one another in this instance due to the fact that the larger bounding boxes (shown in a pink color) are not intersecting

With the given collision detection implementation, however, the SURPhACE runs into trouble when presented with two separate drawings that are already intersecting one another upon their creation. Such instances can come about as the result of drawing one shape over a pre-existing object or, as mentioned in the previous section, splitting an object which intersects itself (such as an '8') right down the middle. As the SURPhACE resolves collisions caused via dragging by moving the bumped shape in the direction that the dragged object is moving, cases in which the shapes are already intersecting cannot be resolved. Meanwhile *both* shapes are still constantly triggering the now futile collision resolution code which causes the program to crash. The problem is illustrated in this [demo](#) at around the 3:00 mark. With a simple list data structure, a single object cannot hold more than a single unbroken strand and so a different sort of structure is needed. The necessary data structure would have to be able to hold a series of x and y positions, but also a way to signify that there is a gap or hole in the structure so that an object can consist of more than one strand of Vector2's.

I had my partner James Hoyt investigate the potential of utilizing the built-in Graph class in C#. The Graph structure consists of several nodes which can have any number of links to other nodes in the structure. However, James ran into trouble when trying to store Vector2 information (an x and a y value) into each node of the graph. The pre-defined Linked List class was also given brief consideration, but due to the fact that any node in the Linked List (aside from the first and last) had to have a link to a previous node and a next node. Therefore, the Linked List is just as linear as the regular List class. Finally, after talking to our professor, we arrived at the solution of using a List consisting of series of Vector2 Lists. Each List of x and y

positions would represent a single strand and an object would be able to store a number of these strands in another List.



Left: The old data structure was a simple list of positions given by Vector2 objects. **Right:** The new structure adds a new layer or dimension by having a List of Lists that each hold a series of positions.

The SURPhACE can then loop through each strand in the list and draw each based the strand's own series of Vector2's. If the user draws over a pre-existing object, the drawn strand will then be added into the pre-existing object's List of Vector2 Lists and thus the two will be combined into a single object. Split objects will be split as usual, but once the user lets go of the mouse, the SURPhACE will check every object on the canvas and if any two objects are intersecting one another, they will be merged into one object. Thus the implementation of a two-

dimensional List, or a List of Lists solves the problem of intersecting shapes and allows for drawing and splitting shapes to work in tandem with physical interaction between said shapes. The new implementation can be seen in action [here](#).

The Future: Networking and Bugs... Substantial Bugs

Although the SURPhACE has progressed substantially over the course of ten weeks, the current state of the program is far from perfect. Although the new splitting functionality works for the most part, splitting a shape a second time can cause sections of a shape to disappear randomly. The cause of this issue has yet to be discovered, but the investigation of the problem is currently the highest priority. Another bug involves the grouping of objects (this is different from the merging discussed in the previous section). Drawn objects can be grouped and moved together by holding shift and then clicking the objects one wishes to group. However, sudden movements can cause shapes within the same group to collide which results in a program crash. These are the two major problems that need to be addressed in the future.

Meanwhile, the SURPhACE lacks the networking feature that will turn an amusing toy into a practical and useful tool. James and I plan to implement a lidgren-based network. Lidgren is an open source and relatively user-friendly .net framework library that is designed to make networking simpler and easier (<http://code.google.com/p/lidgren-network-gen3/>). Despite the practical nature of the library in question, James and I lacked the time required to fully implement a networking feature into the SURPhACE. We do, however, have an idea of what needs to be sent over the network and at what point in the code such information needs to be sent. To perform an operation and communicate it across the network, the SURPhACE should

send a unique id representing the object involved in the operation, a string or character representing what operation is being performed (a draw, split, rotation, flip, etc.) and the data necessary to perform the given operation. A rotation can thus be communicated by sending the id of the object, then the word “rotate” and then the direction of the rotation. The other clients will then receive that information, find that object on the canvas that matches the id and then rotate the given object using its own rotate function.

The addition of networking also raises some other concerns. Many of the operations available in the SURPhACE assume that the user can only take one action at any given time. However, when networking is enabled this will no longer be the case. Scenarios such as one user attempting to split a squiggle while another is dragging it can cause unpredictable problems. Therefore it will be necessary to be able to flag an object as inaccessible and unalterable to other users when someone is already interacting with said object. Investigating the possibilities of lidgren networking, working out bugs in the code and exercising keen foresight will no doubt enable the SURPhACE to eventually become a desirable framework.