

Anybody who's ever sat down at a table has pushed something along its surface. Whether it's moving a mouse, sliding a note to somebody, or just spinning your phone because you're bored, even you, my current reader, have interacted with an object that was resting on a table. And you, like everybody else, have for the most part not given a single thought to the underlying physics that allow the object in question to move. But why should you have to? You don't need to know the static coefficient of friction of the table's surface, or the kinetic force required to keep the object moving once it has already started. But imagine trying to break down those physics, and all the parameters involved, into code to implement on a virtual table surface: a projection onto an actual table that holds virtual objects that can be pushed, slid, or spun by your hands just like real objects. Think of all the variables in play, the give-and-take balance of realism vs. computational efficiency. And then wonder, why is such a mundane action so difficult to recreate?

This problem was realized into the Simulation Utilizing Real-time Physics And Collision Engine, or SURPhACE, a software prototype that allows the user to draw vector-based graphics, and then move them around and have them collide via a physics engine. The goal was to replicate the physics present on an actual table, while keeping in mind the limitations of the hardware we were working with. In addition to the physics features, we also implemented a number of direct interactions with the drawn objects, including a rotation tool, a scaling tool, and a split tool to cut objects into pieces while keeping the separated pieces intact. The development of the SURPhACE was rife with questions, roadblocks, and solutions, all of which assisted us not only in creating this program, but in learning more tricks and possibilities in the hardware and software we decided to use. So, in collaboration with Philip Moccio, who was interested in the vector-based graphics aspect of the project, we set out on building our prototype.

The SURPhACE was built in C#, a high-level programming language, using the Visual Studio 2010 development environment with the XNA extension library, which is useful for 2D applications like ours. We chose this software platform not only because we had prior experience with it, but because the project we were building it for, the MITRE Corporation's EdgeTable, was built in the same platform. Also, Visual Studio's multitude of libraries gave us a number of options for what type of data structure we could use to hold the objects the user would draw. This proved immensely useful, because we went through a lot.

We started out using an unlinked list of Vector2 objects. A Vector2 is basically a point that holds an x and y position for vector calculations and 2D drawing operations. As the mouse moves along the “canvas,” we generate a series of Vector2s and repeatedly draw a line to the latest Vector2 from the previous one. Because the Vector2 points were sampled in rapid succession, the line looked smooth. The advantages of this data structure included not only the ability to give each point an explicit position, but also the ability to enumerate through all the points as they were essentially a straight line, as opposed to some of the later data structures we used. I'll elaborate on the distinction after I've described the other type of data structure: the graph.

The graph was born out of an issue we had while using unlinked lists involving more complex drawings with overlapping lines. Remember how I said the Vector2 list was essentially a straight line? Rather, it was more like a single strand of spaghetti coiled into a shape: sometimes complex, sometimes simple, but always made of a single line . While it may be overlapping itself at some points, it runs into an issue with the SURPhACE's split function, which is essentially like dragging a knife through the object it's being used on. When you drag a knife through a coiled strand of spaghetti, it doesn't become two intact pieces, it becomes a

bunch of smaller strands that can be easily pulled apart. When this happens to a drawing, it loses its structural integrity and becomes much harder to manage.

With graphs, however, the nodes of a graph can connect to multiple other nodes out of sequence, making for a much more two-dimensional object. Then, when the split tool is used, some of the edges of the graph are deleted, but the drawing is still only in two pieces. While this fixed the “straight line” issue, it came at the cost of another: the built-in Node class in C# does not have a position variable, which makes drawing the graph objects much harder. We attempted to fix this by making a subclass for Node called NodeData, so the NodeData class would have everything the Node class has (being a subclass of it), plus a position variable. Unfortunately, this failed because all of the operations that use Nodes will accept only Nodes, no subclasses. So, as the last two attempts taught us, we need a nonlinear data structure that has a position variable. C# doesn't have one built in, so we made our own: A List of Lists of Vector2 objects.

The List of Lists of Vector2s differs from the standard List of Vector2s in that the multiple lists can connect to each other at points where the drawing overlaps. As a result, the Vector2 object that starts one List could also be a midpoint for another List. So, instead of being one strand of spaghetti, it forms a type of spaghetti that can branch out in multiple directions. Would you want to eat it? Probably not, but it's an interesting, useful structure that suits our exact purpose.

So now that the problem of “what data structure to use for the drawings” has been solved, we can get to the main event of this project: the physics engine. What we wanted originally was a full representation of physics, with the objects having mass and inertia and velocity and the like, and the table having a coefficient of friction that would allow the objects to slow to a stop after a time. Objects would collide and bounce off of each other, and would also bounce off of the

boundaries of the table. This would allow for not only a fully realized simulation of actual physics, but would also allow us a number of variables to tweak to get the simulation we desire. While these variables would be helpful, they are the reason why that mundane action I described is so difficult to replicate. As a result, we had to simplify the replication for our prototype.

Even though what we got was much simpler, in a way it made more sense. The SURPhACE instead uses more basic collision and simply has the objects push each other in one direction rather than have them bounce off of each other and go in two separate directions. Additionally, the objects don't have a velocity and the table doesn't have a friction, so objects are only moved by dragging and dropping, rather than sliding. However, this is not only simpler and therefore easier on the computer, it is still somewhat realistic. While we were looking for a perfect representation of Newtonian physics, where actions have equal and opposite reactions, the truth is that the friction objects have on a surface makes Newton's Third Law almost negligible. Objects moving on a table may still slide to some degree, but will very rarely bounce away from each other. Instead, the object moving just pushes the object it collides with a short distance before it stops, which is what happens here. Overall, while it wasn't a full physics simulation, it still adequately represents moving objects for the purpose of telecollaboration.

But the fact that those physics are still there means that pushing objects isn't that mundane after all. There's a big difference between what people can see when moving an object and what's really happening underneath it all. And if, in the future, we implement that full physics engine, we can go beyond just a realistic representation of table physics. That's the benefit of working in a virtual environment. As Morpheus said, "Some rules can be bent; others can be broken." A friction variable can be turned down to nothing for air hockey, or rotational velocity can be made perfectly elastic so an object can spin forever. But all of those calculations,

if they work, will go unnoticed by the user. And that's the true test of a physics engine like this. If it works, you won't give it a single thought. You'll just sit down at that table, and push something along its surface.