**Prock VR IDE**

Final Project Post

Mickey Barboi | Ken Sun | Logan Dirkx

12 . 18 . 16

**Motivation**

Humans are wired to interact with their physical surroundings efficiently through learned and innate spatial reasoning skills. Programming languages use syntax to communicate a programmer’s intent that is far removed from either the resulting operation or the level at which the programmer reasons. Modern, high level languages offer increasingly more powerful abstractions and tools with which to program, however, these languages are still far removed from the world of mocks, flowcharts, and specifications that drive software development.

Before any project, many programmers begin with a whiteboard sketch of the class hierarchy and interactions he or she will need to implement. We believe that physical representations of the intangible abstractions in a program help designers reason about functionality. Instead of being forced to switch from an intuitively drawn model to complex syntax, we want to allow programmers to use the same pseudo-physical objects to manipulate code in virtual reality. Specifically, we believe this offers distinct benefits when writing, reading, or running code.

**Contributions**

**Mickey Barboi:** Started off by embedding a Python 2.7 runtime into Unreal Engine 4, using a metaprogrammer to generate the cpp classes to wrap python objects. Set up physics, threw together some basic meshes, and spawned it all in the world. Worked on debugger and fancy splines, but didn’t finish them for the demo.

**Ken Sun:** Designed input mappings and implemented teleportation, box and line grabbing and box spawn. Mickey and I had a long discussion on how the world should look like. Shall we use the full room scale or just use a table top model. For example, shall we encourage the user to sit on chair and operate or encourage them to walk around and operate. We committed to the later idea and decided to pack as much information in the world as we can. Therefore we implemented teleportation. By pressing trackpad in vive or operating joystick in oculus touch, user can initiate teleportation and change the front direction.

**Logan Dirkx:**

Before starting this project, we knew that we’d be wrestling with profound design and ergonomic challenges. Code is the near-definition of abstraction. We attempt to interpret the physical processes of a computer in a series of increasingly abstract languages, that once combined, allow us to control the operations of a computer. Over time, languages have become more human friendly. However, programming still requires extensive knowledge in the mind in order to manipulate the design of a program. VR allows us to create an immersive platform to manipulate virtual objects in intuitive ways. What’s more, clever design allows us to embed additional knowledge into the virtual world to make control and manipulations align with real-world expectations.

Natural interfaces and intuitive designs have increased the accessibility and usability of tools for the masses. Computers used to be a specialized tool for individuals trained in a litany of command-line prompts. Then, Apple introduced the GUI that made computing power accessible to non-specialist users. Simultaneously, GUI made human-computer interactions more streamlined for specialist users. With this in mind, I worked on designing an interface to map with real-world interactions. Through intuitive, error-proofed design, I hoped to design an environment that was not only more useful to a veteran programmer, but also accessible to those without extensive programming knowledge.

All together, I wanted to design a space that accomplished the following:

1. A human-centered 3DUI with natural mappings
2. Physical representations of abstract objects
3. Tiered hierarchy of code specificity
4. Auto-complete functionality to streamline development

First, I wanted to ensure that the unique advantages of an immersive world are captured. Namely, the interactive 3D space. So, I designed the ‘code track’ to wrap around the user. In this way, the user can focus on what is immediately relevant directly in front of them, while being able to gaze to their peripherals and see the previous / upcoming code. The 3DUI also enables users to pull the appropriate code into their field of view. We feel like this is a more natural way of traversing the thread of execution in a program.

One of the grand challenges of coding in VR is the lack of a keyboard. Keyboards allow users to access any function, variable, or command by simply typing the corresponding command. In VR, we needed to find a way to display only the most relevant commands / objects to the user. Our preliminary design included auto-populating boards that show only the physical representations of valid code to the user (similar to autocomplete or input validation offered by many IDEs). Furthermore, we designed the shape & color of the object to create physical limitations for invalid user input. In the example environment, the d.insert function is orange with circular holes. The valid input of int(a) and int(b) spheres auto populate the backboard in the same color informing the user of the validity of these arguments. What’s more, the list(z) object is a cylinder of a different color suggesting that the input could be valid, but it needs to be modified (with a get function for instance). By physically error-proofing incorrect input, the VR IDE reduces the knowledge in the mind required by developers. We also identified that sometimes written confirmation is helpful to a user to ensure that the physical representations match with their expectations. So, we included terminal windows to show this information to the user.

Finally, we realized that most projects are far too large to navigate via an extremely precise pull mechanism. So, we also designed an expandable network of increasing granularity to allow the users to select the scope of the program they wish to view. In this way, users can visualize the connections between classes or libraries. Then, he or she can selectively pull themselves into any class ‘orb’ for further analysis of the variables & methods in that class. Once the desired section of the code is found, users simply walk into the control room discussed above to edit and modify the code. We feel that the ‘big picture’ view grants users a more comprehensive understanding of the interactions between the web of classes.

All together, I feel like this design offers a first step into physical programming in a virtual environment. I anticipate that the main limitation we will experience during implementation is the auto-completing backboards. Determining what objects are most relevant to the users and populating them in real-time seems like a massive technical challenge. Also, finding meaningful shapes and colors for increasing complex programs without creating confusion will not be trivial. Finally, programming languages use nesting / indentations to represent layering. So, handling local vs global variables of the same name add an additional level of complexity to visual representations.

**Outcomes**

* Describe the operation of your final project. What does it do and how does it work?
* How well did your project meet your original project description and goals?
* As a team, describe what are your feelings about your project? Are you happy, content, frustrated, etc.?

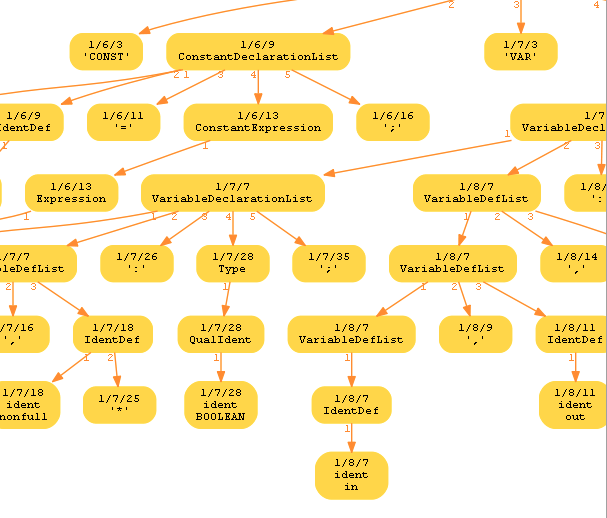
At the start, we had 3 functionality goals we wanted to implement into our IDE.

1. Reading: Read in source code from python and populate a VR environment with meaningful objects
2. Ritting: Make changes to the environment / objects and save the new code back out
3. Running: Run the code in the environment at real time and watch the interactions occur via animations. Tie in a debugger to physically see where errors occur.

These goals are definitely not trivial, but we were able to make significant progress towards all three. The initial setup of dissecting an AST and transforming the objects into VR was the critical backbone we needed to perfect before moving on to modifications in VR. By the end of the semester, we were able to read in basic arithmetic from a python script and visualize the code in boxes and lines in VR.

**Problems encountered**

We have to deal with an arbitrarily complex AST, in a language agnostic way, without implementing a programming language or syntax checking. Deciding what information is most valuable to show to the user, then representing it in a meaningful way is far from trivial. Though we started with simple arithmetic, more interesting programs exponentially more complex once control flow and nesting are intertwined with simple function calls and operations. As complexity increases, simple questions like what to show and where to put objects quickly become roadblocks for implementation.

****

Additionally, we realized there there must be two separate representations of the code. One for writing & editing code, and one for observing execution. The same design and interactions are unlikely to be valuable for both circumstances. Similarly, the way that people ‘white-board’ can differ dramatically. So, creating a one-size-fits-all physical representation might only aid those who share the same mental model of code.

From a technical perspective, we know that text input is not a reasonable way to name objects or make commands. So, all new objects will likely be named incrementally. At the time, we cannot think of a way to make variable naming more intuitive in VR than with a keyboard. The best choice now is randomly assigned letters.

Organizing the visible nodes in the game world in an extensible way is tricky. We’re not sure if there’s a good static way to do it, but for now we’re relying on a crude attraction/repulsion physics model. The point of it is to not have to worry about coordinates for each in-game node while still having them space appropriately. It might not scale in terms of performance, and it doesn’t work while scaling. Best bet here is a very simple home-grown physics simulation instead of relying on the engine’s.

Lastly, game engines are not designed to zoom in. So, it becomes very challenging to create functionality to ‘blow up’ and block of code and see increasingly granular information. Specifically, we’d love to be able to leverage LOD settings already built into the engine to control actor culling when zoomed way outside the scope of a given actor. This is going to have to be done in house.

**Next Steps**

Now that we can read code into a virtual environment as boxes and lines, we want to be able to create new boxes and lines to add to the existing code. Then, we can connect the newly created code objects to the existing thread and write the code back out to a python script via the AST. This could get tricky as far as checking for “valid” changes to the AST tree, not to mention the 3 or 4 layers that the AST ends up being pulled into (Python AST node, cpp wrapper, in-game actor, and higher-order representation.)

After we’ve implemented writing basic operations, we can continue to expand the ‘vocabulary’ of our IDE to handle control flow, classes, instances, collections, and invocation. To do this, we’ll need to design meaningful ways of representing these things in VR that make intuitive sense to the user. This involves designing and creating meshes and customizing existing functionality.

Finally, we want to allow the user to execute the code in VR and watch the objects physically interact in VR. The culmination of all efforts will allow the user to create a ‘Hello, World!’ program and watch it execute in real time- all from VR. This means hooking into a running python runtime, which turned out to be surprisingly easy. When running, we’ll move the code graph to the background and only show the variables in scope. Then, just like a debugger, we’ll step through each statement in our AST graph. Where the static code has lines connecting nodes we’re instead going to animate nodes, making them fly together, then stepping the real debugger and updating the values represented in the game world, and finally animating the nodes apart,

**Video Content:**

Prock Project Demo: https://youtu.be/RdVaZoD8ptc

Prock Physics Demo: <https://youtu.be/SrJamnDWqHo>

Future State Design: <Logan will embed in post>