ECE395: Ideas for Senior Projects

Jonathan Lam

09/06/21

Three potential project ideas are proposed:

1 Typed Filesystem

Modern tree-like filesystems have never enforced any structure on their contents, other than through the use of permissions. It may be useful for package management, human recall, security, and programmability that certain conventions be followed in a file hierarchy. The *nix OS tend to follow some variant of the Filesystem Hierarchy Standard, but it has no means of enforcing it; installing packages in the wrong place (whether from bad install scripts or by user error), for example, may cause compatibility problems. We propose a new FUSE layer that lies on top of a working filesystem, which enforces a (user-specified) schema. To achieve this, we also introduce new filesystem error messages (caused by lack of adherance to the schema), conflict resolution strategies, and a type hierarchy for use in the schema.

2 SVG Video Compression for Variable-Fidelity, Low-Bandwidth Video

In the COVID-19 era, we live in the age of Internet calls. These are often in the form of voice or video calls. In the latter, a common problem is that of limited Internet bandwidth, which often limits us to the former. A common use for video calls is to transmit a user's casual presence (e.g., an image of their face, facial expressions, and surroundings) rather than fine details such as text, and often users may call from their phone in an Internet-constrained environment (e.g., outdoors) – in this case, we may desire a lower-fidelity, but also low-bandwidth, video streaming solution.

We propose a novel vector-based video format for transmitting vector video in an binary SVG-equivalent format, as well as a deep-learning model to convert videos to SVG frames in real-time based on YOLO-net's real-time object detection. The model will allow tweaking of the of "compression level" (fidelity) by adjusting the output shape density.

3 An Intentional Programming Framework for Programming and SWE Education

(This is an idea for my M.Eng. thesis, but in the spirit of proposing project ideas I'd like some feedback on this as well.)

There are many common difficulties among programming students; these include remembering syntax, comprehending error messages, mapping conceptual steps into code, structuring code in a meaningful way, and recalling what a piece of code is supposed to do. While integrated development environments (IDEs), beginner-oriented domain-specific languages (DSLs) (e.g., Scratch), and beginner-oriented techniques such as gradual programming (e.g., Hedy) aid the learning experience, these techniques are still usually locked into the mindset syntax of a single language and do not enforce good programming practices such as abstraction and self-documentation.

I propose an "intent-driven" method for programming education, which involves the creation of an IDE, a DSL, and drivers to transpile the DSL into target programming languages. The DSL will follow the "intentional programming" (Simonyi 1995) programming paradigm, which is a tree-like program representation independent of programming language that allows the user to encode their "intent" at various levels of abstraction. Intents separate purpose from implementation. An intent loosely corresponds to a language-level construct like a procedure, control-flow statement, or special form, but this is transparent to the user, who sees only a building block for abstractions. In this representation, the user describes their intent in an Lisp-like syntax-less manner, and is recursively prompted to define subintents until each intent is defined in terms of primitive intents. The purpose and API of each intent must be documented ("literate programming"). The IDE manages intent definitions, enforces the intent structure, allows "zooming" of abstraction level (similar to code folding), and transpiles to target languages (and relays error messages from the target compiler). The benefits are many-fold. IP: is easier to learn than a "real" programming language; allows experimentation with language features ("genes") across programming languages; encodes SWE best-practices such as top-down design, test-driven development, and self-documentation; relates errors to functional rather than lexical location; reduces cognitive load by intuitive code-folding; encourages learning by-example rather than by-error via a community of (well-documented) user-submitted intents; promotes polyglot programming; and allows grokking overall concepts in their own words ("language-oriented programming") rather than using prescribed formal languages.