HPI

Exploring Visual Primitives
for Authoring Source Code

Master's Project Proposal, Winter Term 2021/2022
Software Architecture Group
Prof. Dr. Robert Hirschfeld

In computer programs, algorithms are usually implemented with textual code, even though there might be a better, often visual representation. When trying to understand such an algorithm, we thus often avoid looking at textual code, but develop a visual representation on paper or whiteboard.

In this project, students explore domains that take advantage of visual representations of code to build a framework allowing for rapid and easy creation of visual, executable representations of code. We aim for a programming environment where, rather than going to the whiteboard, programmers directly prototype their algorithms using visual primitives, means of combination, and means of abstraction provided by our framework and tools and, as they thus express executable code, get instant feedback from the system if their ideas will work in practice. Visual representations of algorithms coexist and integrate with the rest of the programmers' systems and tools.

The project will be based on a research prototype of a block-based programming environment [8] that is written in Squeak/Smalltalk [4] (see below). **Students will:**

- **collect and review visual primitives and workflows of existing visual programming environments in various domains,**
- **re-implement some of these visual primitives and corresponding visualizations in our block-based programming environment, and**
- **verify that the identified visual primitives can generalize by using them to implement visualizations of different domains.**

Background: Programming with Visual Representations

Figure 1: *A screenshot from the Unreal Engine animation editor [1]. Dataflow is visualized through a series of nodes and edges and users receive instantaneous previews of their system on the left side of the screen. Users can thus quickly experiment with values in their graph and receive feedback if their changes are desirable.*

Avoiding textual representations of code as much as possible is common in various domain specific programming systems. Especially tools that are design-focused or target programming novices or domain experts often adopt different representations of code [7]. For example, dataflow systems allow thinking about and arranging processing steps in terms of dataflow or events rather than sequential textual code.

Prior work has demonstrated that also more classical examples of algorithms lend themselves well to visual representations [2, 6]. For example, text books that explain algorithms such as tree transformations will likely employ visualizations to make their function clearer and it could be contemplated if an animation might be even more helpful. An implementation of a similar visualization integrated with code is seen in Figure 2.
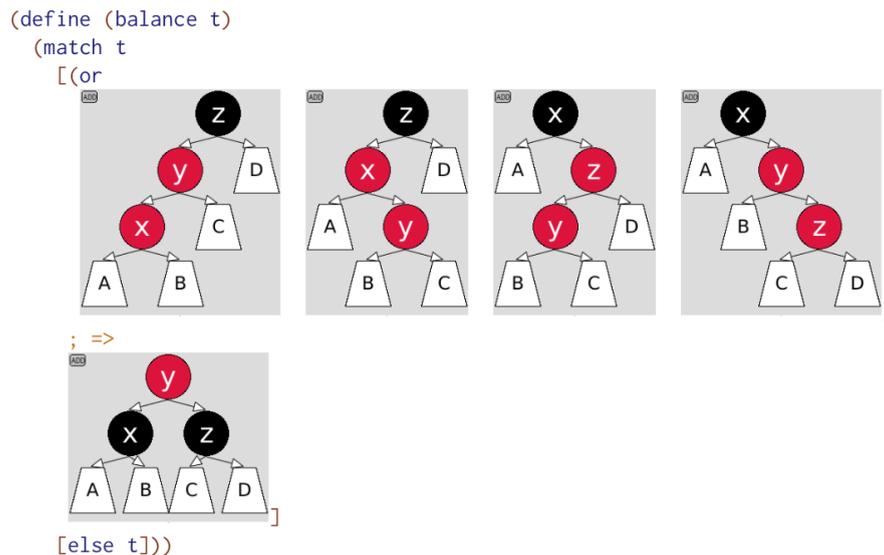
Figure 2: The red-black tree balancing algorithm expressed as a series of visualizations within textual code [2] to ease authoring and understanding. The top four visualizations match certain configurations of red and black nodes and assign variable names ("x", "y, "z") to these nodes in the tree. Below, we see the desired result, where the nodes assigned to the corresponding variables have been rearranged accordingly.

Approach: A Programming Environment that Supports Visual Representations

Students will work in and on the research prototype of our block-based programming environment [8], which is designed to support integrating visual representations with a large code base.

Contemporary programming environments and workflows pose challenges for an integration of visual representations of code with the rest of a system:
- It is unclear how debugging tools should interact with visual representations.
- If nested code inside the visualization is required, navigation between different text buffers may become challenging.
- Text buffers dictate a flow from top-to-bottom and left-to-right, while visual representations may want to make use of different layout and navigation strategies.
- It is unclear how cut/copy/paste of code with visual representations could and should work, especially when pasting to a different textual editor not supporting abstractions beyond text.

Our approach is to instead also display textual code in a flexible, block-based manner that can adapt to different requirements of visual representations inside or around it. Blocks themselves thus act as containers to hold primitives, such as other blocks, but also existing other GUI tools.
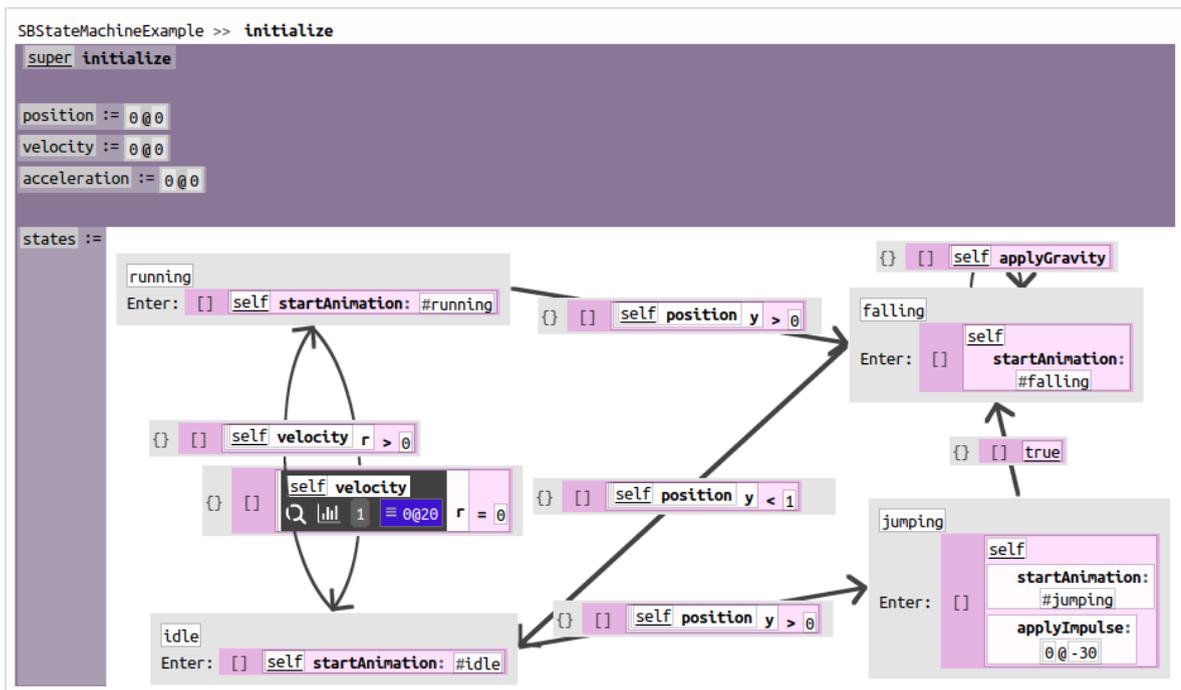

Figure 3: *Textual Smalltalk code projected in our programming environment as blocks [3]. We construct a state machine visually within this "initialize" method and assign it to the variable "states". The state machine itself also contains various pieces of code to express "enter" or "leave" triggers and conditions for when transitions should happen.*

In our programming environment, artefacts such as textual representation of code are *projected* as blocks based on the underlying syntax tree of the code. Visualizations such as the state machine shown in Figure 3 are largely made up of blocks that behave in a manner that makes sense for the visualization (e.g., attach to an arrow) and may stand in the place of a large part of the underlying syntax tree of the code.

Advanced programming tools, such as a debugger, are tailored to work with blocks: even if a visualization is used, our debugger is able to identify the next piece of code to be executed within the visualization, as it merely displays the code blocks in a different manner. If a visualization works on a higher level of abstraction combining multiple lower-level instructions, the programmer can choose to represent a visualization as the underlying code instead.

The programming environment has been built and integrated in Squeak/Smalltalk [4] and currently supports projecting textual Smalltalk and Scheme code as blocks as well as a subset of Javascript. The user interface is built using the Morphic framework [5], but the programming environment itself also provides a collection of blocks and widgets for quickly assembling block-based user and programming interfaces.

Other practices and values

Within this project, we want to follow certain values and practices:
- **Explorable implementation**
  The major portion of the solution will be implemented in Smalltalk to help programmers follow an exploratory workflow and interactively debug and resolve issues with Squeak's existing toolset.

- **Cross-platform compatibility**
  The proposed solution needs to work on recent versions of Microsoft Windows, Apple macOS, and established Linux distributions. Consequently, external dependencies should be avoided or made portable.
- **User testing**
  On a regular basis, parts of the proposed solution need to be prepared and packaged as contributions to the main branch of our block-based programming environment, if not all development happens on the main branch already. Regular feedback and testing with users once a stable base has been established needs to be strongly considered.

Organization

A group of three to five (3-5) students may participate in the project. Organization and tasks will be explored and mainly determined by the project participants. The project will be carried out at the Hasso Plattner Institute in Potsdam or virtually if necessary. Project participants are expected to communicate with the advisors on a regular basis. Communication tools include and are not limited to video chat tools, GitHub issues, projects, a wiki, and e-mail. Participants will regularly present their work to the chair to report progress and to obtain feedback. Communication is likely to be conducted in German or English. Expected results include a working software prototype accompanied by appropriate documentation including its architecture, design decisions, and framework and API usage.

Contact

Prof. Dr. Robert Hirschfeld, Tom Beckmann, Dr. Marcel Taeumel
Software Architecture Group, HPI, Potsdam
http://www.hpi.uni-potsdam.de/swa, {first.last}@hpi.uni-potsdam.de

Further Reading

[1] Unreal Engine 4 Documentation. "Blending Animations". Accessed on: 15.01.2021.
https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/AnimationBlending/
[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. "Adding interactive visual syntax to textual code". *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (November 2020), 28 pages.
DOI:https://doi.org/10.1145/3428290
[3] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. "Visual design for a tree-oriented projectional editor". In Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (<Programming> '20). Association for Computing Machinery, New York, NY, USA, 113–119. DOI:https://doi.org/10.1145/3397537.3397560
[4] D. H. H. Ingalls, T. Kaehler, J. H. Maloney, S. Wallace, and A. C. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1997, pp. 318–326.
[5] J. H. Maloney, "An Introduction to Morphic: The Squeak User Interface Framework," in Squeak: Open Personal Computing and Multimedia, Guzdial, Mark and Rose, Kim, Ed. Prentice Hall, 2002, pp. 39–67.
[6] Andrew J. Ko and Brad A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (*CHI '06*). Association for Computing Machinery, New York, NY, USA, 387–396.
DOI:https://doi.org/10.1145/1124772.1124831
[7] Markus Voelter and Vaclav Pech. 2012. Language modularity with the MPS language workbench. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, 1449–1450.
[8] Sandblocks - Block-based Programming Environment https://github.com/tom95/sandblocks