Integrating a Live Programming Role into Games

Krish Jain
Lake Washington
School District
Redmond, Washington 98052
USA
s-kjain@lwsd.org

Steven L. Tanimoto
Computer Sci. & Engineering
University of Washington
Seattle, Washington 98195
USA
tanimoto@uw.edu

Abstract

Web-based games can permit players to take on multiple roles, and in the past such roles have generally been defined in terms of characters in game narratives. In this report on early work, we propose adding a live-programming role to games that may involve the kind of problem solving that requires "thinking outside of the box." The live programmer can be empowered by the game designers to bend the rules, within certain bounds. We demonstrate the concept using a prototype multi-role game in which players must bring Covid-19 outbreaks under control by performing a sequence of pre-designed actions. The live programmer is able to adjust parameters of the actions, and even disable actions or create new ones. We suggest that having the live programming role in such a game can foster learning about the game domain and structure in different way than usual game playing or modification. Such a live programming role may also be appropriate in some simulation environments and emergency management systems. Finally, we discuss several issues raised by the existence of the live programming role: player power and fairness, "live scripting" (one form of live programming), and characterizations of game sessions in terms of evolution of game state versus evolution of game state plus code versions ("full trajectories").

1. Introduction

Live programming may occur in a variety of contexts. One is musical performance, in which there is typically one programmer writing code in front of an audience, and the code is controlling a music synthesizer, responding to changes in the code during the performance (Aaron & Blackwell, 2013). In another context, a programmer working on creating a dynamic web page writes HTML, SVG, or Javscript code in one pane of browser window, and in another pane, a graphic is shown that has been produced by the code. The objective is fast development of the graphics, so that the SVG can be incorporated into a new web page, especially a web application (JSFiddle-Staff, 2020).

Another kind of context for live programming is what we refer to in this paper as "live scripting." Here, we have a programmer who is editing code that affects the running of an activity. For example, that activity might be an architectural CAD session, in which an architect is drawing a CAD model or a blueprint for a building. The programmer, who may be a different person from the architect, is editing a script that controls how an alignment tool works. For example, the tool might be reprogrammed to cause drawing elements to snap to various horizontal positions on the basis of a hierarchy of attraction preferences involving nearby objects and the current distances to them. The live programmer (live scriptor) here is playing a secondary, behind-the-scenes, role that is supporting the architect.

The meaning of the word "live" in such live scripting can be understood to be "concurrent with the activity being modified." Here "activity" is not necessarily to be interpreted as the computer executing the edited code, as it typically is in live programming situations. The activity is more general and involves, in our architectural example, a session in which a human user (or several users) are interacting with a computational system to accomplish a task, such as designing a building. The live scripting is taking place while the designing is happening. If the scripting were not live in this sense, then it would have to be done in specific time intervals during which other activities such as drawing the blueprint were suspended or allowed to continue but determined to be independent of the scripting. We further discuss the notion of live scripting in section 4.9.

Live scripting has several potential benefits. One is the usual live programming benefit of reduced

latency between programmer action (e.g., editing) and programmer understanding of the consequences of that action. Another is allowing tooling limitations to be fixed without requiring the users of the tools to start their designs or other projects over again. Related to this is the possibility of real-time interaction between the programmer and the user, in the context of the session, so they can cooperate to achieve the joint goal of a successful design and improved tool.

This live-scripting form of live programming could play an important part in education and training, especially when the programmer needs to learn how a certain type of system works and can be modified. One of us has made the case that future systems for emergency management could benefit from facilities for live programming (Tanimoto, 2020). An important component of an emergency management system is a subsystem for conducting training exercises. A live-programming component could be used both to help create new training exercises and as the target of training exercise – i.e., for bringing a new live programmer on-board as an emergency response team member.

With that kind of scenario in mind, we developed a simple collaborative game that very roughly resembles an emergency management system which offers multiple user roles. Within that scenario is a special, unconventional role: a "live programmer." (Now that we have made a distinction between live programming in general, and live scripting, we will revert to the more common terminology of live programmer for the game role to be described; it is really live scripting in the above sense.)

The live programmer is empowered to edit program code that affects the running of the game itself. The overall system, which includes the facility for live programming, it set up such that more traditional game play and live programming not only can happen concurrently, but such that the changes made by the live programmer can take effect immediately, sometimes altering the course of the game, and without requiring players to restart the game.

2. Related Work

The design of programming environments has a major impact on the experience of computer programmers (Edwards, Kell, Petricek, & Church, 2019). In addition, the applications context and social context also have an impact on the programmer experience, especially with novice programmers (Guzdial, 2015).

In addition, by supporting "live" programming, a development environment can enable an interactive programming experience that is "tighter" and that can enhance either the process or the end result of a programming session (Victor, 2012) (McDirmid, 2013) (Church, 2017).

Liveness can be incorporated in a variety of styles within an environment, from full programming to merely parameter tuning (Kato & Goto, 2016). For more literature related to liveness, see the references in some recent works (Petricek, 2019) (Kato, 2017).

Programming is important nowadays not only for creating software products, but to control or modify the behavior of systems that already come with a lot of software. In a domain such emergency management, it can happen that new information-processing needs arise, or preconceived models need to be modified, and computer code may be the best vehicle for achieving the change.

Past work on emergency management systems (EMS) includes work from the decision-control systems standpoint (e.g., (Turoff, 2002)), and the commercial enterprise developers of EMSs such as WebEOC (Juvare-Inc., 2020) and intergovernmental agencies (World-Health-Organization, 2013).

3. The Game Context for Live Programming

Next we describe the specific game we developed that serves as a computer-based activity context for the live programming. The game is collaborative, providing explicitly named roles for multiple players, and allowing each player to join a session over the Internet. The basic game is turn-based. However, there is a special role of live programmer (LP) which is not strictly turn-based but more loosely synchronized with the activities of the other roles.

3.1. A COVID-19 Pandemic Management Scenario

We designed and implemented a game in which a collaborating team of players work to get an imaginary (and extremely simplified) version of the COVID-19 pandemic under control. The game is implemented within an online client-server software framework called SOLUZION, which is described in more detail in section 4.3. Figure 1 shows a screen shot from the beginning of a session. For some of the details about the game, please refer to Video 1 of the two associated with this paper. (URLs to the videos are given in the appendix.)

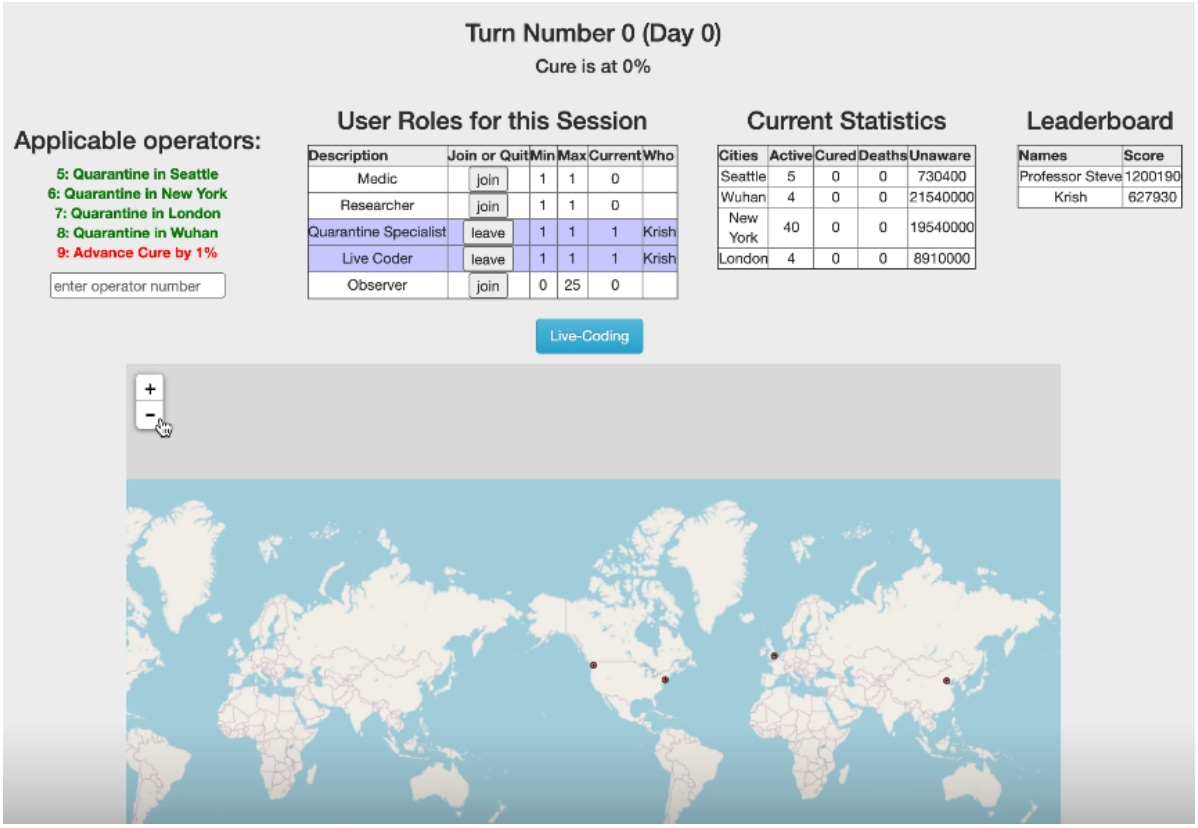

Figure 1 – Screen shot at the start of a game of pandemic management.

3.2. The Live Programming Role

We invented a special role for the game in order to begin to explore the possible ways in which live programming could be meaningfully integrated into games.

The progression of the game itself is controlled by a Python program (the formulation file) that operates within an enclosing client-server framework implemented in Javascript and Python. The formulation file specifies an initial state for the game and a set of game operators that can be applied by players to change the current state of the game.

The live programmer (LP) in the game is empowered to edit the formulation file such that the functions contained within operators change, and/or new operators are created. The effects of these changes occur at the very next turn in the game after the LP clicks on "Save."

The framework requires the LP to have a basic knowledge of Python programming, as well as of the relationship between operators and states. As to not overwhelm a novice programmer, the framework presents two options: a novice version and a complete version. The first comes with suggested events for the LP to follow so that the LP becomes familiar with the code. After developing a thorough understanding of the framework, the LP can choose to use the complete version to edit all the code in the program. In the context of this simulation, the LP can present novel ideas into the mix to better predict outbreaks. For instance, the LP could split an existing city into two to further illustrate the idea of quarantine or

create a new operator for the other roles to act upon. After introducing a new parameter to cities, the change might not be immediately visible, but over multiple turns, it will be observed that the growth of the virus will slow due to increased borders.

For more details about how the LP role works in the game, as well as its technical implementation, please see Video 2 of the two videos associated with this paper. (Again, URLs to the videos are given in the appendix.)

The game has been play-tested on a very limited basis at the time of this writing. However, the trial game showed that the LP could respond to needs arising in the game at scripted intervals, to improve the rudimentary COVID-19 propagation model to more accurately predict outbreaks and help the team score better in the game.

If this game represented an actual emergency management system, and there were a LP who could make functional changes to the system in response to new need during a developing crisis, the system and its team of operators might be better able to manage the crisis.

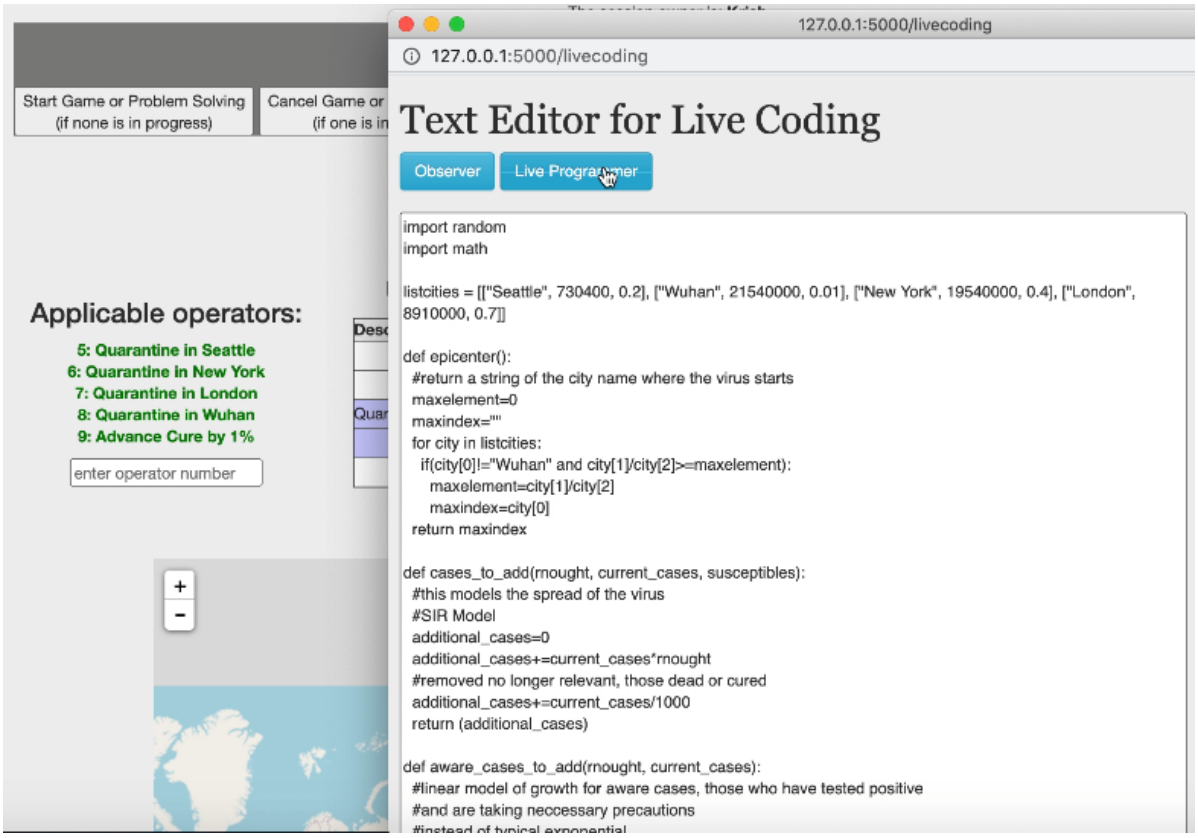

Figure 2 – Illustration of a live programmer's screen.

4. Future Work and Discussion

4.1. Future Work

We are considering several possible follow-on developments in this project: enhancements to the game to make it somewhat more realistic, studying the LP role in relation to other roles, and making live programming an essential part of playing and winning the game. In this next section we consider issues primarily related to the second of these (understanding and reshaping the LP role in relation to the other roles).

4.2. Discussion Overview

This section brings up a variety of issues germane to having a live programming (LP) role in a game. The first issue is the relationship between the LP role and the other game roles. This leads to a discussion

of how much power the LP has, and how it might be limited or expanded for various purposes. Playing the LP role typically requires more knowledge than playing other roles, and we next discuss what that knowledge is and how the requirements can be reduced. These same issues are affected by the purpose of the game in relation to application domains, which are then discussed. Then we return to the issue of live scripting vs. live programming generally, which was laid out at the beginning of the paper. Finally, we consider the issue of "divergence" which can be problematical in some live programming contexts; it turns out not to be a concern for us when our purpose is supporting LP for the sake of enriching game play, but it remains potentially problematical if the purpose of LP during the game is to improve the formulation of the game.

4.3. Computational Model

In order to discuss some of the more interesting issues associated with this work, we first present some background on the SOLUZION framework as well as a general model for a SOLUZION game that includes live programming, such as our game presented in this paper. One example of the model's use is that it will help us explain the possible application of this sort of game to emergency management training.

The SOLUZION framework permits a game designer to specify (using the Python language) an initial game state and a set of operators, as well as a set of player roles. For example, we have a SOLUZION formulation for a 4-disk version of the Towers of Hanoi puzzle, with a single player role, "Solver." The initial state contains a representation of a platform with three pegs, and 4 disks of different diameters, with holes in their centers, piled up on the leftmost (first) peg. There are six operators, with names such as "Move the topmost disk on Peg 1 to Peg 2." A player takes a turn by selecting an operator that is applicable in the current state. The computer then applies the operator, which updates the state and the players' displays (over the web, in their browsers). If the game has multiple roles, then turn taking is typically managed through the current state having a variable whose value specifies whose turn it is. A game session begins with the players adopting roles. SOLUZION does not prevent a user from taking more than one role, but normally each player will take one role in a session. When the session owner (the user who first connects to the game server after it has been started) starts the game, the SOLUZION system running on the server puts the game into its initial state. That specifies which player(s) may move first, based on the Python code in the game formulation file. Players take turns making moves until the game ends, typically when a "goal" state is reached. In the Towers of Hanoi, the goal state is the configuration of disks in which they are all piled up on the rightmost peg.

Thus a typical SOLUZION game session can be characterized as a sequence of moves: $\langle \mu_1, \mu_2, \ldots, \mu_n \rangle$, which induces a corresponding sequence of game states $\langle \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$. Here σ_0 represents the initial state, and σ_i for $i > 0$ represents the game state that results at the completion of move μ_i. Every game state σ_j is the result of zero or more applications of game operators. (Each operator application is one move.)

The incorporation of a live programming role makes the basic SOLUZION model insufficient to characterize a game session. This is because after the game starts, the set of operators may change at any time, due to the live programmer's editing. Existing operators may have their names changed, their preconditions changed, or their state-transformation functions changed. They may also be deleted. New operators may be added. In order to describe the evolution of the set of operators, we'll use a sequence of code versions: $\langle \Xi_0, \Xi_1, \ldots, \Xi_m \rangle$.

The symbol Ξ_0 refers to the problem formulation file at the beginning of the game, before a live programmer has made any changes. Each time the live programmer edits and saves the formulation file, the set of available operators changes from, say, the one specified by Ξ_i to the one specified by Ξ_{i+1}. The relative timing of LP code-saving steps and player moves is important. The *full trajectory* of a game session consists of a sequence in which game moves and LP save operations are interleaved (though not necessarily in a one-to-one fashion). Such a sequence can be represented as $\langle \zeta_0, \zeta_1, \ldots, \zeta_\omega \rangle$, and each ζ_i is either one of the μ_j or one of the Ξ_k. These sequences are ordered by time, and the full tra-

jectory should have associated with it a sequence of time stamps, $\langle t_0, t_1, \ldots, t_\omega \rangle$, such that in the game session event ζ_i happened at time t_i. The full trajectory, together with its sequence of time stamps, gives an accurate representation of what the players did during a game, insofar as the evolving game state and formulation changes are concerned. Such a record makes it possible to recompute the sequence of game states $\langle \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$, in spite of the live programmer having made changes to the operator definitions throughout the game.

When we discuss the possible application of these games to training emergency management personnel, the full trajectory is a fundamental object that might be evaluated to provide educational assessment relative to the desired training.

4.4. Live Programmer Role Relationships

We designed our game to have a live programmer (LP) role with the assumption that this LP role would be one more role for a team of collaborators. Although we tell players they are working together in a team, the software does not prevent them from either fighting over resources or getting in each others' ways; e.g., the live programmer changing something that another player has been counting on, such as the way funds are distributed when they arrive from governments. We have been assuming that a team wants to win and will try to avoid these conflicts.

Our assumption is not necessarily warranted — that the LP will make positive contributions toward the team's achieving the game goal of bringing the pandemic under control. The LP may have the ability, within the game's software, to help, but that does not mean the LP will know how to help or even actually wish to help. The LP could intentionally work against the efforts of the rest of the team, whether instructed to help or not. Also, the LP could break the smooth operation of the game by causing a semantic error, say, that sets the game into an extreme state. (Merely syntactic programming errors will block the activation of new code versions that contain them and are fairly harmless.)

This leads to the concern, whether the LP is well-intentioned, or sufficiently skilled, or not, of whether the other players trust the LP. If they are aware that the power of the LP is limited, this concern can be ameliorated. If they trust the intentions of the LP, and they can be confident in the skill of the LP, the concern can also be ameliorated. To reduce the worry on the part of other players, game designers can do three kinds of things, and we are considering doing these to our game: (a) putting more explicit limits on what the LP can do; (b) providing better training to the team, including helping teach the LP to be competent, and training the other players about how to communicate with and make requests to the LP; and (c) share the LP's power among players by rotating the LP role within a game. We consider each of these directions in more detail in subsections 4.5-4.7 below.

Before that, we discuss the particular role of "observer," for which we haven't provided any operators for making moves in the game. Our "Observer" role was originally created to support audience members who might wish to see the code being edited by the LP, as has been popular in live coding performances with music (Sorensen, 2005). As implemented now, the observer role can serve that purpose, so online sessions with the game can have audiences. However, the observer role can also serve as a supplemental affordance for any player other than the LP. Thus the Medic is allowed to not only be the Medic but also be an observer of the LP, and see the code that the LP is editing. Finally, we have neither intended it nor tried it, but the observer role could support pair programming in which the observer acts as a code navigator (communicating with the LP via Zoom, Skype, etc., or in person), while the LP does the editing.

4.5. Limits on The Live Programming

In our current implementation of the game, the LP is free to edit any part of the problem-formulation file, which specified the initial state of the game, and the operators that are available to the players to make moves. Let us now consider placing easily understandable limits on what the LP can do.

The game operators could be partitioned into two groups: mutable and immutable operators, such that the LP can only edit the mutable ones; which operators are which could be made known to all players.

Thus players who have learned what the immutable operators do can be assured that they will not change and they will probably not lead to surprises.

Alternatively, the game's state variables could similarly be partitioned such that the LP's new code cannot alter the existing behaviors with regard to the protected variables. This would be more difficult to enforce while still keeping the existing LP programming mechanism intact, because an operator is generally allowed by the SOLUZION framework to update the state in an arbitrary way. However, if there were a "budget" variable in the game, players might be reassured to know that it could not be tampered with by the LP; or perhaps they would be disappointed to know that.

Even more restrictive would be a constraint that the LP only create new code for a specific mathematical equation that is part of the embedded simulation model. This function could be prevented from having side effects or from accessing any but pre-selected state variables. In this way, the players could rest assured that game logic would not change as a result of LP errors, creativity, or mischief, and yet the LP could have a well-defined means of contributing to the game.

A different form of restriction would be to limit the LP's changes to the turn-taking logic of the game. It might seem like a good idea to a team to let the medic perform a succession of vaccination operations before the researcher or quarantine specialist get to proceed with a new move, and this could be enabled on a one-time basis by the LP, or whenever some particular condition is true in the current state, such as a vaccine being available and millions still need it.

This last example illustrates one sense of the phrase "bending the rules." The initial turn-taking order gets modified in response to the needs of simulation, in a manner not expected in games. Another example of a LP bending the rules is modifying a requirement that the Medic and Researcher never overspend their budgets, such that players can go into the red financially, at least temporarily, to a certain extent. Postponing a deadline or allowing one to be missed is another example of this sort of update that seems inconsistent with the original game formulation but adds a degree of realism. Allowing larger groups of people to be vaccinated in one turn, or cured of the disease, is another example of bending the rules – altering the dynamics of the game in a manner unexpected at the start of the game.

An interesting possible variation of the limitation on LPs according to which game operators they might edit is the following (which we have not implemented but are considering). In our game, each operator is associated with one of the non-LP roles: Medic, Researcher, Quarantine specialist (M, R, Q). By structuring game turns to follow an interleaved order (LP, M, LP, R, LP, Q, ...) and restricting the LP edits to only operators of the role next up, players would have an understanding that the LP is regularly providing coding services for each of them. Furthermore, each non-LP could specify through a menu, which operator they would permit the LP to edit. This might not prevent mischief, but could give clearer expectations about the role relationships than with the LP having complete freedom.

One more way to limit what the LP can do is to limit edits on operators to only the pre-condition portion of operators. This is a predicate that defines the scope of applicability of the operator. For example, undertaking a Covid-related research study may only be allowable in the game when 2 million dollars are available in the current game state. However changing that precondition to allow the study when only 1 million dollars are available widens the scope of applicability without altering the portion of the operator that changes the current state.

Before we leave the subject of protecting a team from the potential ravages of a rogue LP, we note that having an uncooperative or ill-intentioned group member is a problem for cooperative teams of all kinds, and not to be blamed on the existence of an LP role. However, the difference in power between a normal game role and the LP role justifies some special emphasis on the issue.

4.6. What Players Need to Know

Incorporating a live programming role into a game imposes some additional demands on the person filling that role, and can also raise expectations on other players. Let's consider what the LP role may require and then discuss effects on other players.

In our game, the live programmer needs three kinds of knowledge: (a) programming in Python, (b) the structure of a SOLUZION problem formulation, including some elements of the "classical theory of problem solving," and (c) the specific effects of the game's operators. If we add limitations on what the LP is permitted to change or add, then the LP needs to be aware of these limitations, as well.

The Python source code in the game's initial problem formulation is simple enough that advanced knowledge of Python is not required by the LP, in order to read and understand it. In addition, modifying this code is not expected to require advanced Python knowledge either. For example, new class definitions are not required, although modifications to the given State class, could be useful; adding a new state variable, through an assignment statement such as `new_state.num_virus_variants = 3` is allowed in Python, even if the State class' `__init__` did not set up any `num_virus_variants` member variable. If an LP wishes to use advanced Python features, however, there is nothing that we have put in the game to stop that.

The LP needs to understand that our game is implemented within a software framework (SOLUZION) that requires all potential player actions to be implemented as "operators" that may transform a data object, known as the "state" to effect moves or progress in the game. Each operator has three components: a textual name used in the game's human interface, a "precondition" function that determines whether the operator is allowed in the current state, and a "state-transformation function" that maps the current state to a new game state when the operator is used. Learning this structure is relatively easy, say, in comparison with learning to program. A new LP should either be given a 15-minute personal introduction to SOLUZION and the existing problem formulation code or watch a short video.

The third kind of knowledge needed by the LP is an understanding of what the existing formulation's operators do, and how they do it. This can be learned through a combination of the tutorial (which should combine an introduction to SOLUZION with information about how the existing operators work), game play – to see the operators put into action by the players, and examining the source code of the problem formulation. Comments within this source code assist the LP in this regard.

While the knowledge requirements for the LP role may seem formidable, one can argue that they are not so bad. Programming ability and fluency is more and more common, as computational literacy is taken seriously by K-12 educators. The SOLUZION structure is intentionally simple, almost minimal in its requirements, and the given problem formulation file is already in the proper form, so an LP never has to come up with a formulation file from scratch. The existing game is quite simple, and does not involve complicated code in its operators. That said, in the future, we could further simplify what the LP needs to know through one or more of the limitations mentioned earlier, such as limiting the changeable code to one particular mathematical function used in modelling the pandemic's spread.

What the non-LP players need to know is a little about their own roles (e.g., Medic, Researcher, Quarantine Specialist) in controlling a pandemic, how the basic game mechanics work (turn taking) and enough about the LP role to either trust the LP (if possible) or have a justified mistrust.

Game events must be understandable to all players. Certain events take place after a pre-specified number of turns have been taken – this could be a new outbreak of the disease, the discovery of a new variant, or a breakthrough in vaccine development. Players should also be able to read game state variables as shown on the screen and know generally what they represent.

Prompts to the LP need to be understandable, and understandable in terms of the problem formulation elements – current state, game operators including their preconditions and state-transformation functions. A LP might benefit from having a "cheat sheet" about how to get started in responding to such prompts.

4.7. Power Sharing

An alternative to greatly restricting the free-ranging power of the one LP in the game is to somehow distribute that power more evenly among players. Here are some ways we might go about re-designing the game for that.

Rotation of the role of Live Programmer could be incorporated and enforced, so each player who wishes to do so could have the chance to perform live programming as part of his/her regular turn. Such a policy might comport well with the earlier-mentioned constraint that such live programming be limited to editing only the operators associated with the role whose turn it is. This suggests all players would have to be able to program in Python to take full advantage of this policy.

This rotation approach is not very much different from saying that all roles in the game should be considered LP roles. Then the operators become simply a means of ordering edits into turns. A player would take a turn by making some edits as a LP and then applying an operator to signal the end of the turn. Whether players would be allowed to perform editing outside their turns is a design decision that might depend on how quickly players are expected to make changes, and how their edits might be restricted to operators they "own." If their editing time intervals and program scopes are allowed to overlap, then conflict-resolution methods might be needed.

A game with this many LPs might require more elaborate scaffolding to keep them all on track. Once again, each could be limited to specific functions, operators, or formulas.

Finally, we could achieve an approximation of this sort of power sharing without making any change to the current game implementation as follows. We would ask each non-LP player to take on both a regular role (Medic, Researcher, Quarantine specialist) and an observer role. These is no limit on how many observers there can be. We then instruct each player to be part of the "programming committee" by using techniques of pair programming, triple programming, etc., to assist the LP in making the changes needed by the whole team. We might call this manner of playing the "co-programming" game strategy. Co-programming this way avoids certain editing/versioning conflicts that could arise if all players were LPs simultaneously editing the problem formulation file. That is another possibility, however.

4.8. Contrasting Application Domains

We imagine three types of applications for the techniques used in our game to combine the LP role with the rest of the application: (a) training of live programmers to modify existing code in the context of a running activity, (b) game design, during which the modifications introduced by the LP get immediately tested in subsequent turns of the play-testing session, and (c) entertainment.

Training a live programmer involves not only helping that person master the three kinds of knowledge listed in Section 4.5, but giving the LP experience in communicating with a team in the context of solving a problem. Emergency management is one such context (Tanimoto, 2013). Such communication requires that the LP not only understand much of the existing code, but be able to discuss its structure and functionality with team members who might have next to no programming knowledge. This could mean that the LP learns to help the non-LPs develop mental models for the code functionality and for the challenges of the coding process.

Game design differs in a fundamental respect from training live programmers. The actual sequence of state changes during game design is of much less interest to the team than obtaining the final version of the problem formulation file. The game-play session, with the live programming role, is a means to an end in which the final game formulation Ξ_m will represent a game that no longer needs a live programmer, since the tweaking of game rules will have been completed.

A third aim for prospective designers of our type of game is simply to create entertaining challenges for players. Entertainment aspects include game storyline, problem/puzzle solving, and the social aspects of collaboration, or possibly competition.

Our main interest is the first category of applications. Live programming may turn out to be important in domains such as emergency management, or large control-system software maintenance (e.g., transportation systems, nuclear power plants, space stations, etc.) An example from emergency management where a live programmer may be required is patching a system for earthquake response management to accept a new format of map data (say, produced by a new model of drones), so that the data can be integrated with the existing maps to show locations of drones or beacons. The software must keep

running to keep supporting current rescue missions, but the new functionality needs to be added.

The game context can be helpful for training live programmers, as the environment may feel safer or more welcoming, or just simpler as a first step. A veteran of the LP role in our game may have gained enough confidence to consider working up to a LP role in, say, managing an earthquake response or other emergency.

To create an effective training tool for emergency management (EM) through a reworking of our game, not only should an initial problem formulation provide a good starting point for an EM exercise, but the full game trajectory described in section 4.3 should be automatically captured and analyzed, so that pedagogical feedback can be given to the players about their responses to particular game situations. This should include an analysis of the code changes made by the LP to determine which were effective, which were ineffective, and why or why not. Such a game and analysis system could be augmented with in-game questionnaires to further interrogate players on their beliefs and the reasons for their in-game choices, leading to even more accurate assessments.

A complex phenomenon, such as the evolution of a Covid-19 pandemic, would be modeled much more accurately in a computer simulation based on high-fidelity data sets and mathematical formulas, than in a simple turn-based game. One can imagine that a complex, scientific computational model for the pandemic could be set up to have a live programmer making adjustments to it while running. That might make sense if the simulation requires long run-times during which some of the assumptions on which it is based turn out to change. In that scenario, the LP would need deep knowledge of the computational model, but would perhaps not have to know about any game roles or much about other users of the system. That scenario is also a possible application of our integrated LP role, but unless the running simulation is also driving policy decisions with real-life consequences, this kind of system doesn't seem subject to the same risks as emergency management systems face, and so the safety and security issues are less in focus.

The distinction between game and simulation, already tricky, is further eroded in a game with a LP role. After all, isn't a game just a simulation with rules about what players can do? If the rules can change or disappear, what's left might just look like a simulation. Of course, our LP role can also change the simulation, or make it disappear, too. But by incorporating an LP role, we admit to "fuzzing the rules of the game," and thus we blur the line between game and simulation.

Yet one more possible application of our style of game is to support general learning and meta-cognition. The players' task could be thought of as having to quickly learn a set of rules and then track the rules as they change. This could prompt students to reflect on how game structures guide their own thinking as they live their lives, and how they might reshape those guides as their lives unfold.

4.9. Characterizing Live Scripting

The distinction we made at the outset is that live scripting is a special type of live programming in which an ongoing activity which involves computation has human users whose experiences are being affected by the programming as it happens. We further discuss this distinction here.

The category of programming commonly called "scripting" typically refers to developing relatively small programs that work with larger, existing software items either to (1) "glue" them together, as a Unix shell script might invoke a data-processing program and then pipe its output to a graphing program, or (2) customize a large software application, such as Microsoft Word, adding new functionality through Visual Basic programs. Customizing applications such as Gnu Emacs is accomplished by writing scripts in Emacs Lisp. However, it requires a much greater level of technical proficiency to script Emacs effectively than to simply use it as a text editor. Efforts have been directed to providing scaffolding for would-be scriptors, e.g., in the form of customizable buttons in a Xerox Lisp environment (MacLean, Carter, Lövstrand, & Moran, 1990). That group stresses the need for a culture of software tailoring in addition to any special features in the environment.

In our game, the scripting can also benefit from a culture of tailoring, so that the live programmer can

feel fully supported by the rest of the team in making changes to the game's formulation details. This culture is relevant regardless of whether game players are simply trying to win, to improve the game, or to learn about live programming or game structures.

In the context of collaborative design of software systems, including games, the argument for users and designers working together has been made by Bødker and Gronbaek (Bødker & Grønbæk, 1991). That teamwork situation is analogous to game players and live programmers working together to improve the effectiveness of game operators. Our use of the term live scripting is somewhat apt for the sorts of co-design in which users and programmers are testing and coding with a process of many very short cycles of editing and testing. However, we would argue that scripting is really live when the cycles form a continuous flow, and it is not necessary to explicitly restart tests whenever an update to the code is made.

An additional example of live scripting is a collaboration between an advertising layout specialist and a CSS specialist. They have a shared screen in front of them. The JSFiddle website (JSFiddle-Staff, 2020) is up. The CSS specialist is editing the CSS code as the ad specialist provides feedback about the look. In this case, the result will be CSS code that then goes into regular use on a corporate sales page. A variation of this involves a color-blind reader instead of the ad specialist, but again with the CSS specialist. The result of this session is a customized styling of a the material, optimized for the individual or for a relatively narrow set of viewers. In either case, the scripting is part of a close collaboration between concurrent participants one of who edits code.

Live scripting for design can be considered a subclass of co-design processes in which we have a specific LP role, and at least one other role whose job it is to test and/or use the computational affordances being changed by the LP, and in which liveness plays an important role. The liveness eliminates or reduces the need for the non-LP participant to repeat completed steps, due to having to restart the software. This definition may not always lead to a clear distinction between live scripting and other co-participation processes, but it seems to capture the nature of the relationships among the roles in our game, while also being applicable to other systems.

4.10. Divergence of Trajectories – Game play vs Program Execution

One of the sticky technical issues around live programming in a software development context is the "divergence" that can arise between the execution state (which typically can depend on past versions of the source code) and the current version of the source code (Basman, Church, Klokmose, & Clark, 2016). If users wish to be able to re-create an arbitrary execution state from a session, it may be infeasible, because the program's source code has been altered by live coding, and older versions have not been kept. Another aspect of divergence is that the continuing execution of a program that has been live-modified is no longer guaranteed to be representative of the code in its latest version. Thus the final code version Ξ_m from the session, cannot be accepted as an adequate formulation of the game solely on the evidence that the final game state σ_n from the session is a desired final state. The continuing execution is dependent both on code that exists and on code that existed in the past but no longer exists.

One way of preventing the loss of access to early execution states is to build into a system facilities for "remembering." One is to start by backing up the original program formulation file, and then logging every live-coding edit and every user-input event, and time-stamping them in such a way that the complete session trajectory can be replayed. Another is to build-in state-externalization methods that allow check-pointing the evolving session state at an arbitrarily fine temporal resolution. Such techniques can be useful not only in addressing the divergence issue, but also to enable flexible error recovery when live programming results in undesired execution states, such as errors or unexpected deletion of state data.

In our game, as in other SOLUZION framework games, all relevant execution state for a game session is embodied in the instances of a game "State" class, which, defined in Python, is easily externalized as a JSON text string, although we have not yet had sufficient reason to implement that. The sequence of formulation file versions, represented in section 4.3 above as $\langle \Xi_0, \ldots \Xi_m \rangle$, could be written out to file storage, with a new file (with index i as part of the file name of the i-th version), after each LP

save operation. Again, we have not implemented this. But the fact that we could do this suggests that divergence in the sense of Basman et al should not be seen here as impugning the incorporation of the LP role.

If it is desired that the final formulation file Ξ_m be a means to produce the final game state σ_n without need of old formulation versions $\Xi_0, \ldots, \Xi_{m-1}$, then an additional auto-replay mechanism should be added to the SOLUZION infrastructure to permit any move sequence prefix $\langle \mu_1, \mu_2, \ldots, \mu_i \rangle$ to be used as a regression test on the latest Ξ_j, and if ever a desired intermediate state is not attained, then the latest formulation change should be undone and replaced with a formulation that achieves the correct state before proceeding further in the game. Such a process may be worthwhile in a game-design context, but might lead to player confusion if added to our game.

In our game scenario, as in musical live-coding, one can consider that the possible disparity (between the current game state and the game state that would exist if the the game were re-run from the beginning using only the latest versions of the operators) to be "part of the game" and not usually of concern, since the live programmer is trying to help control the evolution of the game state rather than produce a new piece of software.

Provided that an original problem formulation file is not thrown away, we maintain the possibility of playing again along any particular full game trajectory, even without a recording of the original session. However, to replay an entire game the same way, under evolving rules, the players, including the live programmer, would need to perform the same actions and edits as happened in the earlier game, and with very similar relative timing. Some ineffective edits and redundant saves could be skipped, but the essential code updates and player moves would have to take place again in order to produce the same state sequence.

4.11. Safety, Learnability, Empowerment, and Engendering Trust

We discuss a few more topics before closing: safety, learnability, technical empowerment, and how to engender trust within a team having a live programmer.

By empowering one user (or a subset of users) to modify a running system or the rules of an in-progress game, there is a danger that the system will break or become worse off than before any live programming. This issue has been discussed by the second author elsewhere (Tanimoto, 2020). Using a game context rather than a real emergency management situation can reduce the risk of real damage while facilitating education and training about live programming and effective collaboration.

The learnability issue is particularly important in the emergency management context because it is relatively likely that less-than-fully-trained personnel may have to come on board to help deal with an emergency. Liveness in a programming environment can help a programmer new to that environment in coming to understand the possible effects of various code changes and additions.

High technical empowerment for live programming means designing the base system in such a way that a live programmer can change much of the system functionality. One way to achieve this is for the base system to be written in a language such as Smalltalk that supports inspection and modification. In fact, a commercial data analysis system known as Analyst was written with this in mind at Xerox PARC in the 1980s (Thomas, 1997) (Xerox-PARC, 1987). Earlier, we discussed ways to limit such power, but it's worth mentioning that expanding the power is sometimes of interest.

Finally, trust and collaboration, especially important during emergencies, can be fostered through training exercises, including games. A live programmer (and in particular, a live scriptor), working in a collaborative situation, a game or not, should be trustworthy. The observer role (as a supplement to the roles present in the game) allows a player watch the code in real time, serving as a referee and a trusted adviser for all the changes that the live programmer is making. This also allows the observer to function as a deterrent to the live programmer abusing the premises of the framework. Often, the live programmer only has the technical expertise to code, but not the expertise to decide what pressing issues need to be fixed; in this specific simulation, the live programmer would probably lack the medical expertise needed

to decide what needs to be changed. Through observing the code, the observers can better inform the live programmer on the action that needs to be taken. There is much more to these issues, but we leave the discussion here.

5. Acknowledgements

The authors would like to thank the organizers and insightful reviewers, including Mariana Mărășoiu, Luke Church, Colin Clark, Philip Tchernavskij, and those anonymous, as well as the play-testers who tried out the game. We also thank the PPIG December attendees who provided feedback after our presentation.

Appendix

There are two ".mov" format videos that support this paper's narrative. They are available at this time via the following URLs. Each is approximately 6 minutes in length. The first describes the collaborative game referred to in the paper, and the second describes how the live programming role works.

`http://xanthippe.cs.washington.edu/ppig20au/PPIG-Video1.mov`
`http://xanthippe.cs.washington.edu/ppig20au/PPIG-Video2.mov`

6. References

Aaron, S., & Blackwell, A. (2013). From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the first acm sigplan workshop on functional art, music, modeling & design* (p. 35–46). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2505341.2505346` doi: 10.1145/2505341.2505346

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the psychology of programming interest group annual conference 2016 (ppig 2016).*

Bødker, S., & Grønbæk, K. (1991, March). Cooperative prototyping: Users and designers in mutual activity. *International Journal of Man-Machine Studies*, *34*, 453-478.

Church, L. (2017). *Becoming alive, growing up.* Vancover BC, Canada. (Invited keynote, LIVE 2017, workshop at SPLASH/OOPSLA)

Edwards, J., Kell, S., Petricek, T., & Church, L. (2019). Evaluating programming systems design. In *Proc. 30th annual workshop of the psychology of programming interest group, ppig 2019.*

Guzdial, M. (2015). *Learner-centered design of computing education: Research on computing for everyone*. Morgan and Claypool.

JSFiddle-Staff. (2020). *Jsfiddle.* Retrieved from `https://jsfiddle.net`

Juvare-Inc. (2020). *Webeoc.* Retrieved from `http://juvare.com/webeoc`

Kato, J. (2017). *User interfaces for live programming.* Keynote presentation at LIVE 2017, Vancouver, Canada.

Kato, J., & Goto, M. (2016, 07). Live tuning: Expanding live programming benefits to non-programmers. In *Proceedings of ecoop live.* ACM. Retrieved from `https://junkato.jp/live-tuning/`

MacLean, A., Carter, K., Lövstrand, L., & Moran, T. (1990, March). User-tailorable systems: pressing the issues with buttons. In *Proceedings of the sigchi conference on human factors in computing systems* (p. 175-182).

McDirmid, S. (2013). Usable live programming. In *Proc. 2013 acm international symposium on new ideas, new paradigms, and reflections on programming & software* (p. 53-62). ACM.

Petricek, T. (2019). *Histogram: You have to know the past to understand the present.* Retrieved from `http://tomasp.net/histogram/`

Sorensen, A. (2005). Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the australasian computer music conference.*

Tanimoto, S. (2013). A perspective on the evolution of live programming. In *Proc. 1st international*

workshop on live programming (p. 31-34). Los Alamitos, CA: IEEE Computer Society.

Tanimoto, S. (2020). Multiagent live programming systems: Models and prospects for critical applications. In *Programming '20: Conference companion of the 4th international conference on art, science, and engineering of programming* (p. 90–96). New York: Assoc. for Computing Machinery. Retrieved from https://doi.org/10.1145/3397537.3397556

Thomas, D. (1997). *Travels with smalltalk.* Retrieved from http://www.mojowire.com/TravelsWithSmalltalk/DaveThomas-TravelsWithSmalltalk.htm

Turoff, M. (2002). Past and future emergency response information systems. *Communications of the A.C.M.*, *45*, 19-32.

Victor, B. (2012). *Inventing on principle.* video. Retrieved from https://vimeo.com/36579366

World-Health-Organization. (2013). *A systematic review of public health emergency operation centers (eoc).* Geneva, Switzerland. Retrieved 28 March 2020, from http://apps.who.int/iris/bitstream/10665/99043/1/WHO_HSE_GCR_2014.1_eng.pdf

Xerox-PARC. (1987). The analyst workstation system. In *Xerox special information systems.*