Game Development as a Serious Game with Live-Programming and Time-Travel Mechanics

Anthony Savidis, Alexandros Katsarakis

Game Development as a Serious Game with Live-Programming and Time-Travel Mechanics

Anthony Savidis[1,2], Alexandros Katsarakis[2]
[1]Institute of Computer Science, FORTH, Heraklion, Crete, Greece
[2]Department of Computer Science, University of Crete, Greece
as@ics.forth.gr, akatsarakis@csd.uoc.gr

Abstract. Serious games for programming provide players with some type of algorithmic mechanics to accomplish game challenges. Such mechanics maybe formally algorithmic, or in some cases not theoretically linked to strict programming constructs, although still characterized as programming-related games. We discuss a serious game with visual programming where the primary mission is the development of a simple 2d game. Its primary novelty is the lack of separate build and run cycles. There is only one game mode, with gameplay and game development being inseparable, where every game object can be clicked, live-programmed, and live-edited during play. Additionally, time may be freely rewind and replayed, undoing or redoing internally all related user actions and game state updates. During such time travels, it is allowed to drop the entire history onwards, from any given point in time, and continue from there.

Keywords: Serious Games, Visual Programming, Live Programming, Time-Travel Mechanics, Learning Programming.

1 Introduction

Games with programming mechanics, either through some explicit language or via implicit algorithmic elements, exist for a long time. They are frequently targeted to a specific player audience and focus either on new forms of entertainment, offering at an abstract semantic level very challenging algorithmic puzzles, or may be used for learning purposes, falling in the category of serious games with the aim to teach and develop basic programming skills.

In this context, we discuss a serious game for acquiring basic programming skills, relying on visual programming systems. The target game itself is both developed and played in a single runtime mode, making gameplay and game development two inseparable tasks performed interchangeably. Essentially, programming and editing are treated as standard game mechanics, independently of the game that is actually being developed by the player. This feature allows changing the game-program in a live manner, while running, and is known as *live programming* [8]. This notion of live programming in our game goes beyond code fragments and covers all game assets, also becoming editable during gameplay, something we similarly call *live editing*.

This way, offering an exploratory programming system for crafting a game, all development operations and features are *game mechanics*, while the development environment is a live editor with only one operational mode being gameplay. Then, as part

of these mechanics, we support a time-travel facility, enabling player developers to browse back and forth in the game timeline, enabling to replay the history again from any previous point in time, with speed control options, or even select to restore an earlier point in time and continue thereafter. This facility is not merely a video recording tool, but works exactly like a global undo and redo system, storing precise snapshots and state differences taken within every game loop. This allows developers replay or rewind development actions (code or character editing) and also gameplay actions, under a single global timeline.

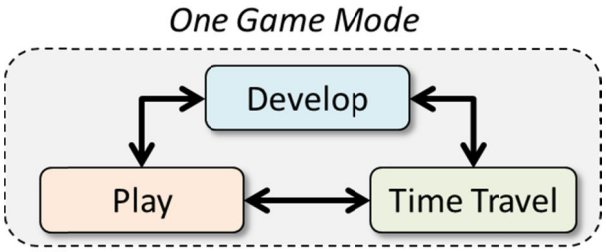

Fig. 1. All three general tasks can be interchangeably applied within one global mode, game mode, with develop and time-travel becoming essential game mechanics.

The idea for active development is also inspired by the Smalltalk 80 programming environment during early eighties where Smalltalk introduced object-orientation and reflection that were highly advanced concepts. The language designers took some decisions to make the language experience and its IDE very easy to use. They adopted an "*everything is an object*" approach that concerned not only program elements but also IDE components and access to the operating system via the language. In this sense, environment configuration, tool-chain control, underlying system management, etc., were all possible via a unified control mechanism either from code or the user-interface. Anything selectable with the mouse could be directly programmed.

Also, our work is based on the notion of active learning [1], emphasizing learner-centered and learner-driven processes, where learners are not strictly constrained in what to do, but are offered an open challenge to develop a game they prefer. This is in contrast to teacher-centered and system-driven perspectives, where the task is very specific and learners are expected to perform in a narrow path of activities.

1.1 Contribution

Our contribution falls in the general field of serious games for teaching programming, while in particular it concerns the design and implementation of a comprehensive toolset[1] for in-game live-development mechanics combining: (i) visual programming and direct manipulation of all game objects *during gameplay*; and (ii) time-travel on global game state recordings, enabling free rewinding or replaying for easier, iterative and playful live testing and in-game updating. In this serious game for programming, we offer a 2d game development toolset with no separate build and run

[1]https://www.youtube.com/watch?v=rpDWvi0Fejo, https://github.com/alexkatsarakis/GameEnvironment

modes. Everything is performed in one live game mode, and every single user action is effectively a gameplay action.

2 Related Work

Firstly, we consider worth mentioning the initial versions of programmable Lego bricks [3], which eventually led to Lego Mindstorms, as probably the first example of serious games for learning programming, even though consisting of mechanical toys. Next, we briefly review recent work on serious games for programming, referring to some commercial games with programming-related mechanics that are relevant, with an elaborate analysis provided under [1], including both research and popular commercial games. Compared to [1], we judge more strictly the theoretical foundation of the programming mechanics, as outlined under Fig. 2, clearly separating between explicit (programming language, code) and implicit forms (no language, no demonstrated Turing completeness). Then, we judge more loosely the compliance to the ACM Curriculum [18], which concerns programming as a profession, setting focus only on the fundamental programming concepts and algorithmic design. In other words, we treat serious games for programming as introductory tools for non-professional programmers, rather than as teaching alternatives.

Effectively, implicit mechanics are appealing and entertaining, but are not directly linked to an underlying formal algorithmic model. Thus, they cannot be theoretically treated as serious games techniques for the art and science of programming.

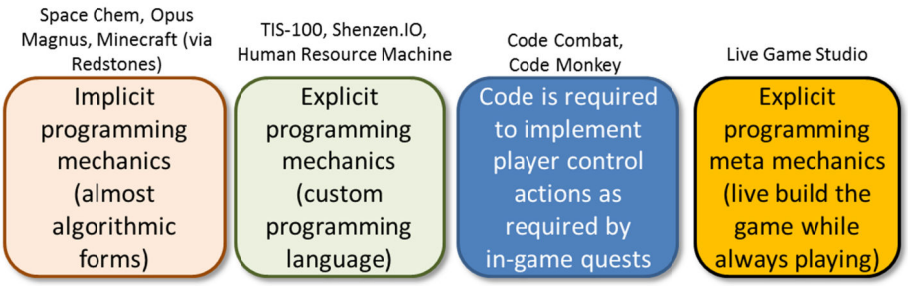

Fig. 2. Categories of games with implicit or explicit use of programming-based mechanics – the last category is our serious game, tentatively named "*live game studio*".

Commercial games with explicit programming mechanics like TIS-100 (assembly), Sehnezen.IO (circuit assembly) and Human Resource Machine (intermediate assembly) are popular to programmers, but require a programming background, in fact familiarization with machine-level symbolisms. Games with implicit mechanics like SpaceChem, Opus Magnus, and Minecraft, do not involve pure algorithmic reasoning or formally-related notations, thus we do not treat them as games for programming.

Code Combat [6], Code Monkey [14] and Code Hero [8] represent a category of games where programming is used as a control substitute, asking the player to write code fragments (source text) so that target actions occur in the game (like player char-

acter control). In [12], a study demonstrated that combining artwork and code editing in a player-extensible serious game resulted in increased motivation, something that we directly adopted in our work. Kodu Game Lab [10], a variant of Kodu originally by Microsoft Research, is a tool related to our game, since it is an environment to visually program games via a subset of algorithmic elements and a custom tile-based graphical language. As also mentioned earlier [13], due to the style and form of its visual language not being Turing complete, Kodu variants are not introductory programming laboratories, but mostly game prototyping environments. Finally, Tynker [7] is an on-line platform for introductory teaching of programming to kids through a visual block-based programming system, including a few challenges requiring complete sample games. Compared to Tynker we also adopt a visual language being Blockly [1], but then focus on live-programming, in contrast to traditional modal builds, and provide the time-travel play and test facility.

3 Live Development

3.1 System Architecture

The software architecture of our system is depicted under Fig. 3, showing the extensions we introduced on top of the game engine (in our case a 2d platformer engine). We particularly emphasize the management of objects, assets and events, although they typically belong in the game engine, since, depending on the originally supplied API, extra work may be required to introduce wrappers enabling scripted methods, state reflection (capture and restored), and live updating.

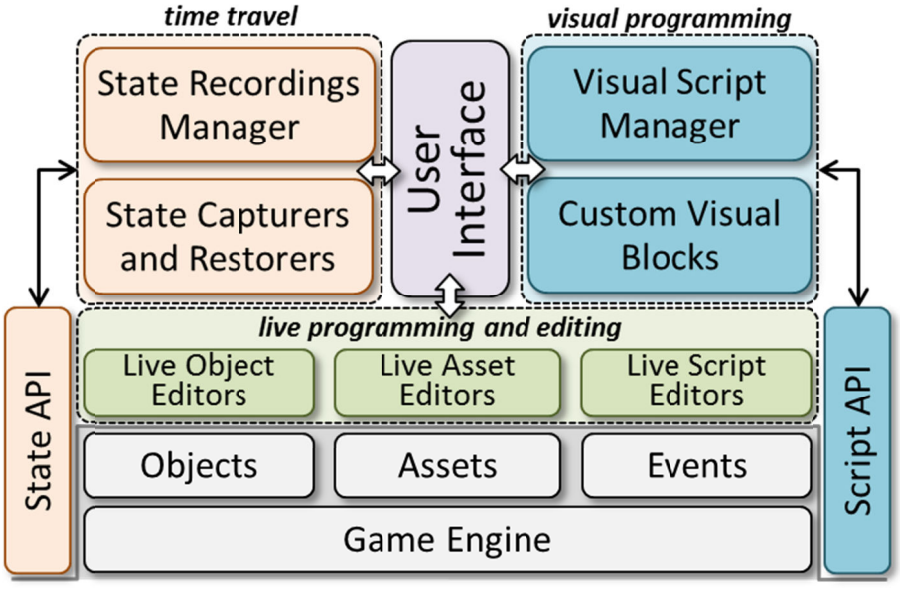

Fig. 3. System architecture, built on top of the game engine components (shown in grey).

Then, on top of such required extensions, the live editing and programming tools are implemented. The state and scripting APIs are also used for implementing the time-travel and the visual programming sub-systems respectively. In particular, within every game loop, the state API is used to capture state updates or retrieve full scene state snapshots, and chain them in sequences of tagged state recordings. For visual scripting, we have introduced custom blocks regarding object methods and properties, game event handling, and also character animation control. All scripts are kept and catalogued by a central script manager that invokes the visual editor if editing is requested. Finally, the User-Interface connects all dots interactively and allows players to handle the features we introduced immediately, at any point in time. In particular, to support live editing and programming, game object selection is enabled all the time over the game terrain. This way, players may pick any object, an action leading to opening the respective object property sheets and the visual script editors (for anything that can be scripted on objects), enabling to live make any desired changes.

3.2 Visual Programming

For visual game scripting we embedded the Blockly [1] editor in our system, while introducing new types of basic blocks, as well as composite block structures for handling all game event types and related object operations.

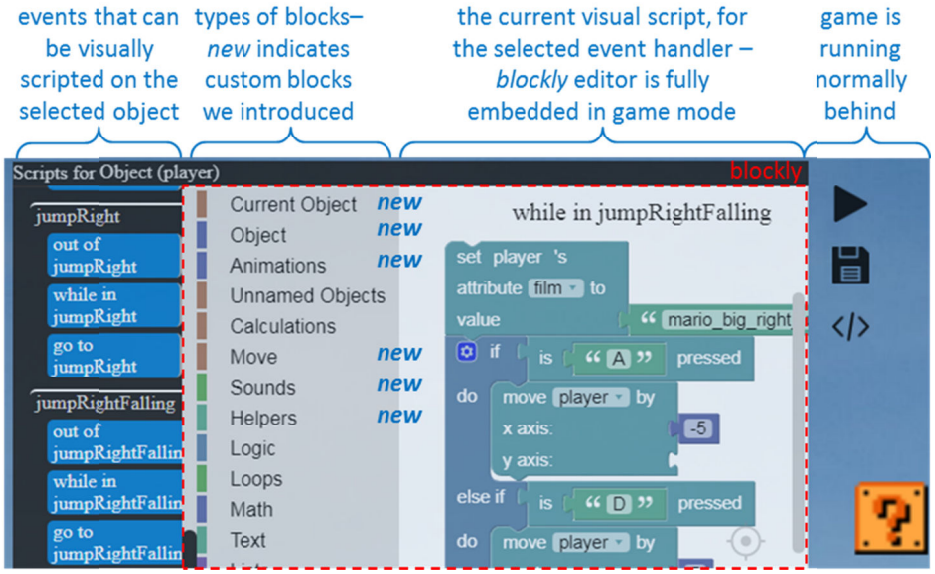

Fig. 4. Visual live-programming non-modal window (moveable, may be hidden or opened during gameplay, anytime), here opened once the player character is selected; the embedded Blockly editor is shown in a dashed red rectangle.

As shown under Fig. 4, the live programming window is non-modal, is displayed on top of the running game scene, and can be freely moved or hidden anytime. Being invisible by default is automatically opened when a game object is selected (Fig. 4

shows all scripts for Mario, the player object, after being selected). Game development functionality has been fully exported to Blockly through new custom types of blocks (indicated with *new* in the mid of Fig. 4), while the player can organize events, states and actions freely inside tabbed groups, with arbitrary naming (e.g. '*jumpRight'* group with '*while in jumpRight'* event handler).

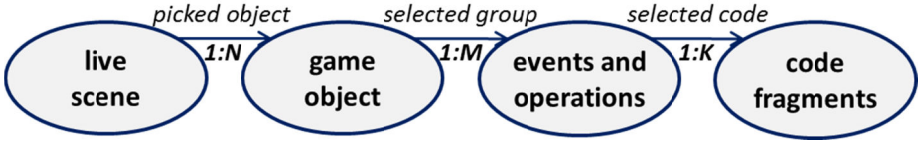

Fig. 5. Quick access to visual code fragments by organizing them around objects; with only a couple of clicks the target code can be edited.

Clearly, there are other game editing environments with such kind of visual scripting, the most notable being Tynker. But an improved feature in our system is the way such visual scripts are accessed, overall how all visual code snippets are organized, as illustrated under Fig. 5. In Tynker, and in most tools we know, visual code fragments are spread in a large canvas as floating elements, moved and grouped with the responsibility of the user. This, even for very simple games, results in a highly crowded space with code blocks that is difficult to handle and can be frustrating.

3.3 Live Editing

Live editing refers to the ability of updating everything on-the-fly, whether code, object properties, terrain structure, artwork or animations. As illustrated under Fig. 6, editing is possible after selecting any game object by a single click and can be done even when the game has been paused, and time-travel (for all available successive state recordings) is carried out by the user. As it is explained latter, to make editing changes applicable during time-traveling the *continue* operation must be selected, clearing all state from that point in time onwards (this is like text editing, then undoing, followed by an edit action which results in clearing the redo list).

The live editing feature allows immediately testing the effect of changes in the current play session, something that otherwise would require exit (interruption), update, rebuild and rerun phases, and then manually bringing the state of the program to the exact point just before interruption. The idea of live-editing was originally introduced in Microsoft Visual Studio debugger, called edit-and-continue [16]. It enabled programmers directly modify the code during debugging when execution is stopped in a breakpoint, and was originally limited to small-scale changes. Then, once execution continues, the new code becomes active and can be live tested in the same running session. A more elaborate example of the live programming feature, and also the advantages of the object-based organization of all visual code snippets, is depicted under Fig. 7, where all steps involved are sequentially numbered.

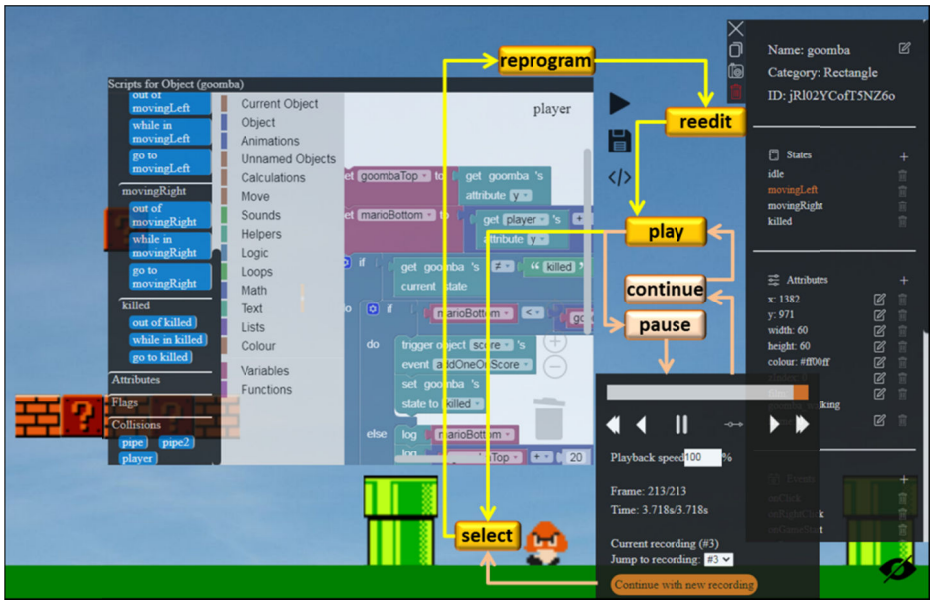

Fig. 6. Live editing and testing cycles as play, select, reprogram, reedit iterations; pausing and time-control (bottom-right with *fwd*, *bwd* and rest operations) is possible anytime (until *continue* is chosen to interrupt) with live-editing fully enabled all the time.

Firstly, the user clicks on a *coin* object, which opens automatically the respective property sheet (right) and the code browser window (left). Then, after browsing on the left column with the identities of all code snippets, the *Collisions* tab is located and the *player* entry is clicked. The latter opens the code snippet for *coin-player* collision logic that is then directly edited in Blockly. As shown, the defined code invokes the *addOneOnScore* method of the *score* object.

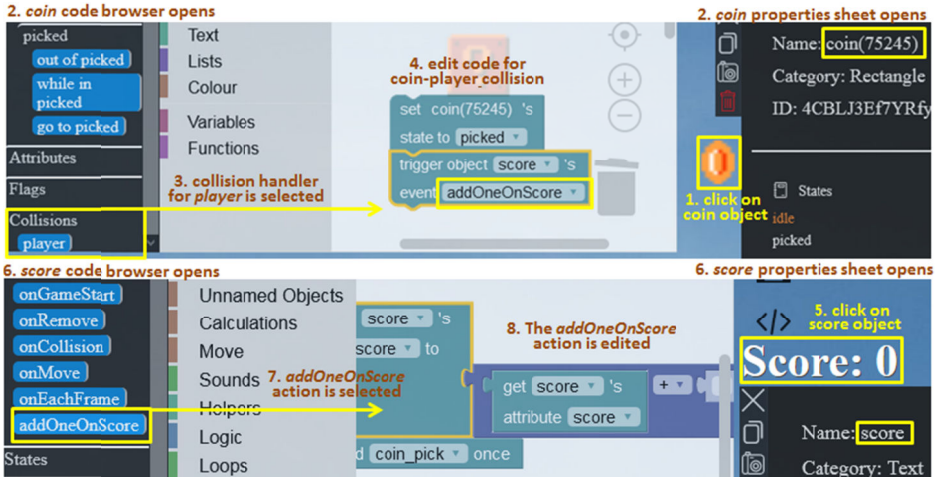

Fig. 7. Live programming of coin-player collision and its effect on score change; top part: coin-player collision, bottom part: score change action.

Then, the *score* object (displaying the score text) is clicked on the terrain area, opens its respective property sheet and code browser window. After browsing on the code groups at left, the *Events* tab is located and the *addOneOnScore* method is selected, opening Blockly editor to define or edit the associated code. The implementation of the method is completed, with the visual code denoting an increase of the respective numeric text field of the *score* object by one.

3.4 Live Clipboard

A standard clipboard is only an editing feature, not linked to serious game mechanics for acquiring programming skills. So, we focused on making the clipboard a tool also making life easier for junior programmers in crafting a simple game (see Fig. 8).

Fig. 8. The live clipboard showing copied objects (top), and object snapshots (bottom) ordered older to newer, displaying time and date of copy; restoring snapshots needs confirmation.

To this end, we integrated all clipboard operations to live editing, thus affecting the game while running. Then, we introduced two variations we consider very helpful:

- Object copy: (i) normal *copy* that is pasted as a new object; and (ii) special copy saving a *snapshot* of its state and restoring it on paste, even after deletion
- *Scene* copy: copies the fully game state and restores it precisely on pasting - this feature was also used very frequently during gameplay

4 Time Travel

4.1 Features

The time-travel feature is functionally the ability to move back and forth in the game time, by precisely storing in every game loop the state of all game elements, including code, objects, assets, terrain, animations, etc. Initially, it was designed to back-up learners with an easy-to-use testing tool, enabling them to trace backwards

the behavior of the game and directly inspect what possibly went wrong in terms of game logic programming. However, after adding all interactive recording and replay operations, facilitating to reproduce an entire sequence of recorded states as a film backward or forward, by also preserving the original game loop timing, it was immediately seen also as a nice game feature, just like time-rewind in Braid [15]. In terms of implementation, such a time-travel feature is demanding. However, it's dual role serving as testing tool and as a game mechanic makes it very helpful for learning programming being a genuinely experimentation-driven and exploratory process.

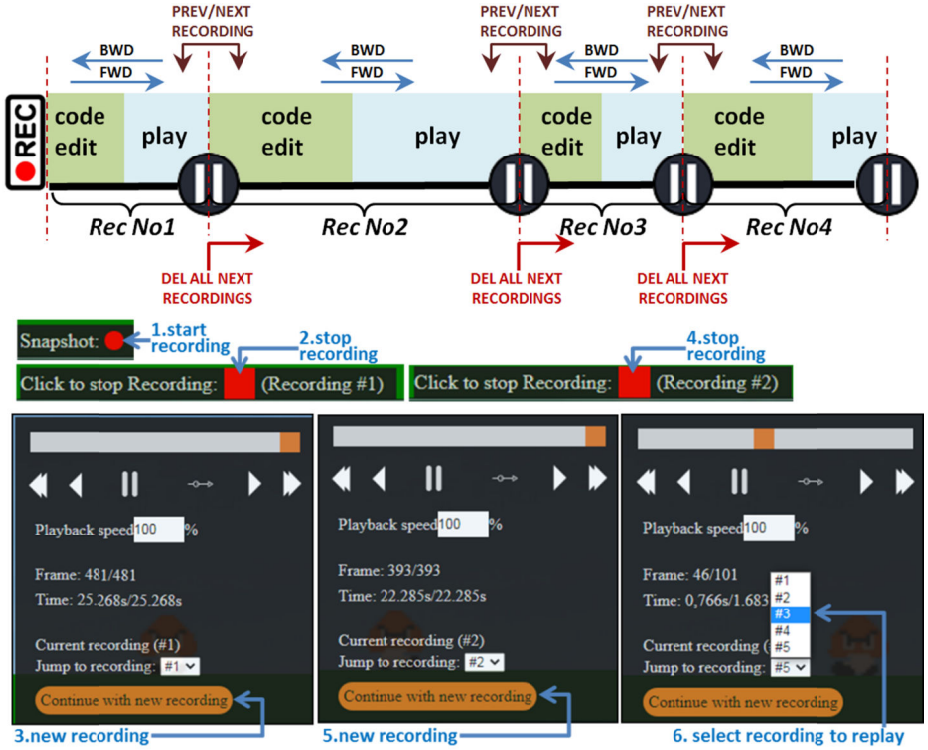

Fig. 9. *Top*: state recording (i.e. no video) of game actions enabling: (a) chain recordings by pausing; (b) move back and forth between them to navigate in events and state updates; (c) replay any recording forward and backward; and (d) delete all recordings and continue. *Bottom*: the actual record and replay user-interface.

The use of the feature is outlined under Fig. 9. More specifically, the user can initiate a recording session anytime. Then, by *pausing and continuing* recording, a new distinct recording chunk is created, following the previous one. After this, it is possible to play backward or forward any selected recording, or move to the next or previous recording, or alternatively select any of the recordings from a dropdown list.

Such recordings internally chain state updates for game loops including code modifications, editing actions, or typical game play events, all at the order they occur. This mixing of development and gameplay actions is illustrated in every recording of Fig.

9 as the two "*code edit*' and "*play*" rectangular areas. It should be noted that interleaving of an arbitrary number of such actions is possible to form a single recording. During a recording replay process, the user may pause and then freely review the state of objects, code, or artwork, thus observing the values they had at that point in time. If desired the game can be restored at that point-in-time by choosing to continue from there, effectively *deleting all states onwards in the current recording, and also all subsequent recordings, thereafter*.

4.2 Implementation

The time-travel implementation technique is outlined under Fig. 10, and we briefly discuss how it has been accomplished. As mentioned, recordings are semantic, chaining state modifications that are tracked within every game loop. We implemented two techniques, with the same end-result, but with different computation complexity and memory requirements, both relying on the Command [11] design pattern. Effectively, actions become objects encapsulating their side effect, all implementing a common interface though which their effect can be undone or redone at any point in time (illustrated by lists of change undo / redo objects).

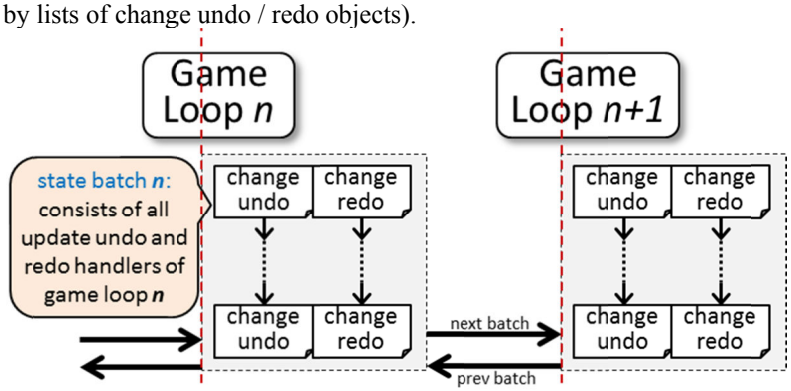

Fig. 10. The essence of state recording for play, coding and edit actions, all in a single game mode, through undo and redo handlers (Command design pattern) for each action, batched and linked across game loops to form chained structured semantic recordings.

For the first implementation, we keep one state-update per game loop, called a *snapshot*, being faster but requiring more memory, as it saves the state of all objects (linking to Fig. 10, only one undo / redo object per loop is stored). In the second implementation, we record the state modifications on objects for every action, called state *deltas*, being slower as it overloads all game actions with state storage processing, but results in smaller memory footprints for recordings since only differences (i.e. deltas) are recorded. Both options are offered to users, so with every new recording they have to choose whether it is delta or snapshot.

The handling of object *deletion* undoing (or object *creation* redoing) and its integration to the undoing of all object modifications prior to deletion was very challenging. Deletion as such is not related to some language operator, but implies the removal of an object from the game scene, something that may occur via a delete or dispose

operator in certain languages, or reference nullification with latter garbage collection on other languages. Essentially, not only an object reference after deletion is invalid, but cannot be restored on the same reference, since the way the memory manager handles memory blocks and memory references is implementation dependent and differs across platforms and languages.

```
using ObjId = uint64_t;
class ObjIdDir {
   void    Insert (ObjId, Obj*);
   void    Remove (ObjId);
   Obj*    LookupById (ObjId);
   ObjId   LookupByObj (Obj*);
   ObjId   Gen (void);
   static  ObjDir& Get (void);
};
class Sprite : Obj { native sprite
   void Move (int dx, int dy);
   void SetFrame (unsigned frameNo);
   void SetFilm (
      const AnimatinFilm& film
   );
};
```

```
namespace SpriteMethods { adapter methods
   void Move (ObjId id, int dx, int dy)
   {
      dynamic_cast<Sprite*>
      (ObjDir::Get().Lookup(id))
      ->Move(dx, dy);
     }
   }
   const ObjState GetState (ObjId id);
   Sprite* New (const ObjState&, ObjId);
   similar wrappers for all methods…
}
Instead of sp->Move(dx,dy) call is formed as:
SpriteMethods::Move(
   ObjDir::Get().LookupByObj(sp), dx, dy
);
```

In the game engine, in the callback where deletion of some `Sprite* sp` *needs to be handled, we create an undo command for object deletion as follows (also inserted in current recording):*

```
auto* f  = new DeleteUndo;
auto  id = ObjDir::Get().LookupByObj(sp);
f->SetState( SpriteMethods::GetState(id) );
f->SetObjId( id );
f->SetImpl(
   [f](void) { SpriteMethods::New(f->GetState(), f->GetObjId()); }
};
```

Fig. 11. Adopting generic object ids (like serial numbers) to handle undo (and redo) of object deletion (and construction) on top of a C++ game engine.

Our technique is based on persistent generic object ids on top of every native type of game engine object, with such ids mapped to the native class references when the actual invocation of object methods is required (see Fig. 11, shown in C++). Then, in the game implementation, the bidirectional mapping between generic ids and native objects is handled via an object directory (`ObjIdDir`), with object access being carried out in two steps. Firstly, the mapping from a generic `ObjId` to the native reference (like `Sprite*`) is done, and then the actual method is invoked. Both steps can be handled by special method wrappers, like `SpriteMethods` depicted under Fig. 11. Notably, to undo an object creation a deletion is required. Then, its redo action is exactly like undoing deletion, meaning *redo create* is identical to *undo delete*.

5 Evaluation

We briefly discuss our experience and the key outcomes from an evaluation process we carried out. The overall process took almost one month, more than usually expected due to pandemic rules. We intentionally tried to make the whole process not

looking like a usability study for participants, but like an exploratory programming laboratory assignment. The entire process was remotely handled, via telco sessions, with the following characteristics regarding setup and conduct:

- 11 high school students participated, of almost balanced genders but varying ages, with some experience in visual programming tools from school, and also playing regularly video games on their mobiles or home consoles
- We firstly explained they were going to make a simple version of Super Mario (NES classic edition, just first stage) and were asked to find information on the game online
- Then, we introduced the environment explaining and detailing all of its features, with extra emphasis on terrain and character authoring and the time-travel feature with all its options
- We provided all the required artwork (bitmaps for tiles, sprite sheets, sounds, and all details for motion and animation of game characters)
- We requested they work in small groups, giving freedom of choice for the communication and cooperation tools (eventually, they all used Discord)
- We had regular on-line plenary meetings to discuss the progress of their projects and provide explanations in using the tools
- They were also told they will exchange projects and play each other games, just to increase motivation

When the process was concluded, all students completed three standard SUS questionnaires, differing only on the titles: (i) editing tools for terrain, characters and animations; (ii) game coding tools; and (iii) tools for recording and time-travel. We wanted to have separate feedback for these three tools. Then, we provided another simple questionnaire where they could also insert free text, with the aim to get some additional information, with some of the questions listed below:

- What you think is the coolest feature and why
- What you think is the worst feature and why
- Do you think you learned something?
- Does this felt more of an exercise or a game to you?

The results of the SUS questionnaires gave the following overall positive scores (rounded): 87 for editing tools, 83 for the coding tools (Blockly) and 91 for the recording and play back tools. The feedback from the informal questionnaires gave the following highly interesting feedback (we actually summarize the responses):

- Time control is the coolest feature, following a unanimous opinion – interestingly, two of the kids said that this does not make sense for on-line multiplayer games
- They did not mention anything to be considered as a worst feature
- They reported it was very interesting that they have learned how to make characters animate and move with keyboard control
- They found this process more of a game, a challenge, rather than a typical school exercise as part of a programming lesson

During the process we regularly asked information on how live-development and time-travel were applied. In this context, we observed that live animation coding and editing were used very early almost by all, involving the few steps outlined in Fig. 12.

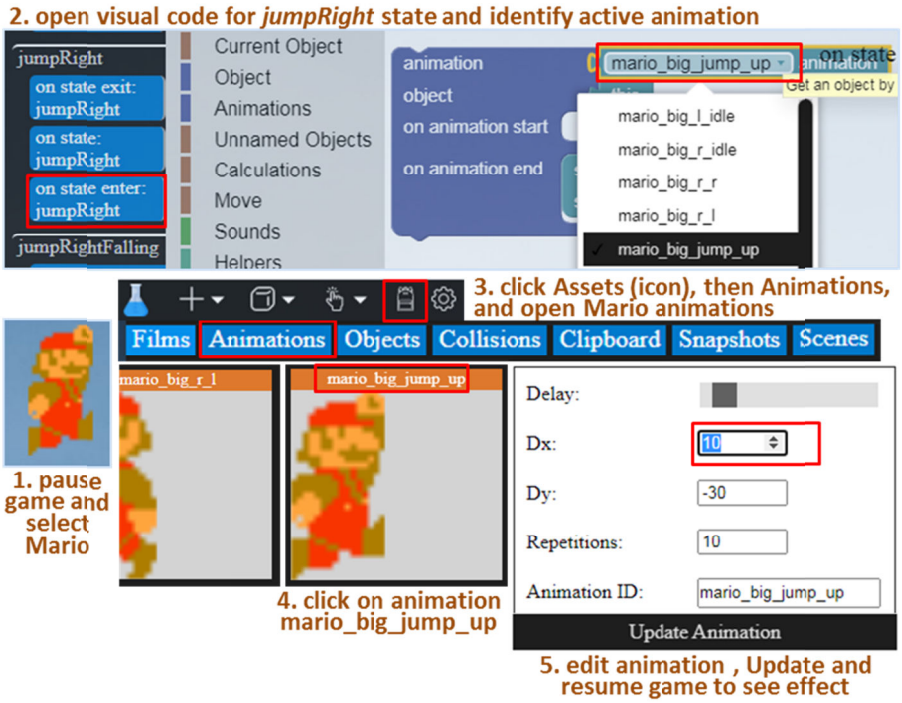

Fig. 12. Live updating of animations with just a few simple coding-and-editing steps.

6 Summary and Conclusions

In our work we focused on game-based learning of programming, putting initially emphasis on increased motivation and exploratory development. The adoption of game development as a task to stimulate learners is not new, and we consider that it rents in roots in applicable constructivist theory and the work of Logo [17]. To make the experience easier and more entertaining we supported immediate development, that is developing during play, and time travelling, that is being able to undo or redo anything in the original gameplay time. The goal was to realize a dual learning experience this way, with development and play interleaved as a unified task.

Initially, while trying to keep a balance between serious and entertaining, we decided that the bigger risk was making a game looking more of a programming-intensive laboratory rather than the opposite. Thus, we very early decided to drop separate build and run cycles, i.e. modal development, and also relieve learners from code management and typical overcrowded code canvases. Then, after early evaluation with users, the time-travel mechanic, originally meant as a testing tool, grew to

both a game-play and a game-testing feature. We carried out an evaluation study, where besides filling questionnaires, we discussed a lot with students. Most of them reported they are not intending for a programmer profession, but still they would like to do hobby programming activities with such visual tools. Now, based on this remark, serious games for programming may be stepping stones for professional programming, or just introductory tools for hobbyists, or support both if appropriately designed. Driven by this, and due to our early evaluation results, we consider that by combining live-programming and time-traveling, while keeping coding tools at a moderate level, we managed to achieve a good balance between these two worlds.

References

1. Blockly. https://developers.google.com/blockly. Accessed May 2021.
2. Barnes, D. (1989). Active Learning. Leeds University TVEI Support Project, 1989. p. 19. ISBN 978-1-872364-00-1.
3. Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O., Repenning, A. (1995). LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. IEEE Symposium on Visual Languages, Darmstadt (Sept 5-9), Germany, IEEE, pp. 172–179
4. Miljanovic, M., Bradbury, J. (2018). A Review of Serious Games for Programming. In Serious Games 4th Joint International Conference, Darmstadt, Germany (November 7-8), 2018, Springer LNCS 11243, 204-216
5. Goldberg. A., Robson, D. (1984). The Language and its Implementation. Addison-Wesley
6. Code Combat. https://codecombat.com/. Accessed May 2021.
7. Tynker. https://www.tynker.com/. Accessed May 2021.
8. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T. (2019). Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. The Art, Science, and Engineering of Programming, Volume 3, Article 1.
9. Code Hero. https://codeherogame.wordpress.com/. Accessed May 2021.
10. KoduGameLab. https://www.kodugamelab.com/. Accessed May 2021.
11. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995): Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
12. Barnes, T., Powell, E., Chaffin, A., Lipford, H. (2008). Game2Learn: Improving the motivation of CS1 students. In: Proc. of the 3rd Int. Conf. on Game Development in Computer Science Education (GDCSE '08). pp. 1-5
13. Buckley, C. (2012). Design and Implementation of a Genre Hybrid Video Game that Integrates the Curriculum of an Introductory Programming Course. Master's thesis, Clemson University (2012). https://tigerprints.clemson.edu/all_theses/1515/ Accessed May 2021.
14. Code Monkey. https://www.codemonkey.com/. Accessed May 2021.
15. Braide (2008). https://en.wikipedia.org/wiki/Braid_(video_game), Accessed May 2021.
16. Microsoft Visual Studio (2019). Edit code and continue. https://docs.microsoft.com/en-us/visualstudio/debugger/edit-and-continue?view=vs-2019. Accessed May 2021.
17. Papert, S. (1980). Mindstorms: Children, Computers and Powerful Ideas. Basic Books. Original edition.
18. ACM/IEEE-CS Joint Task Force on Computing Curricula: Computer science curricula 2013. Tech. rep., ACM Press and IEEE Computer Society Press (Dec 2013)