CapStudio: An Interactive Screencast for Visual Application Development

Koumei Fukahori
The University of Tokyo
Tokyo, Japan
furaga@is.s.u-tokyo.ac.jp

Daisuke Sakamoto
The University of Tokyo
Tokyo, Japan
d.sakamoto@acm.org

Jun Kato
The University of Tokyo
Tokyo, Japan
jun.kato@acm.org

Takeo Igarashi
The University of Tokyo
Tokyo, Japan
takeo@acm.org

CHI 2014, Apr 26 - May 01 2014, Toronto, ON, Canada
ACM 978-1-4503-2474-8/14/04.
http://dx.doi.org/10.1145/2559206.2581138

Abstract

Programmers write and edit their source code in a text editor. However, when they design the look-and-feel of a game application such as an image of a game character and an arrangement of a button, it would be more intuitive to edit the application by directly interacting with these objects on a game window. Although modern game engines realize this facility, they use a highly structured framework and limit what the programmer can edit. In this paper, we present *CapStudio*, a development environment for a visual application with an interactive *screencast*. A screencast is a movie player-like output window with code editing functionality. The screencast works with a traditional text editor. Modifications of source code in the text editor and visual elements on the screencast will be immediately reflected on each other. We created an example application and confirmed the feasibility of our approach.

Author Keywords

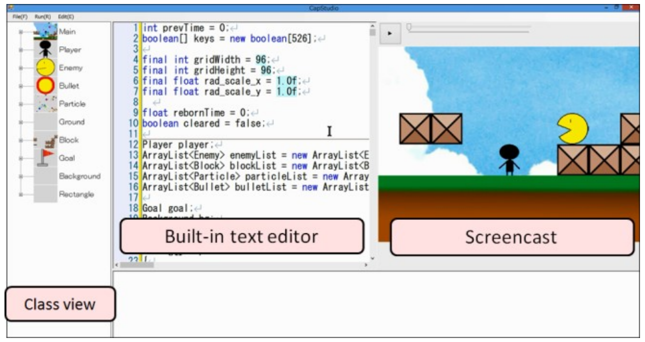
Development environment; design alternatives; Live Programming

ACM Classification Keywords

H.5.2 [Information interfaces and presentation]: User Interfaces- Graphical user interfaces

Introduction

Creating interactive visual applications, such as video games, is a major field of software development. Designing the right look-and-feel of an application is important for making an attractive visual application. For example, subtle differences in the design of objects affect the impression the user has of the application. The usability of a game strongly depends on the arrangements of the interface elements (Buttons, Labels, etc.). However, conventional text-based programming environments do not help programmers to edit these visual elements.



```
int prevTime = 0;
boolean[] keys = new boolean[526];

final int gridWidth = 96;
final int gridHeight = 96;
final float rad_scale_x = 1.0f;
final float rad_scale_y = 1.0f;

float rebornTime = 0;
boolean cleared = false;

Player player;
ArrayList<Enemy> enemyList = new ArrayList<E
ArrayList<Block> blockList = new ArrayList<B
ArrayList<Particle> particleList = new Array
ArrayList<Bullet> bulletList = new ArrayList

Goal goal;
```

Figure 1. The interface of CapStudio.

On the other hand, modern Integrated Development Environments (IDEs, e.g. Eclipse [2] and Visual Studio [9]) and game engines (e.g. Unity [12]) help the programmer to create and edit the visual elements of the application. These systems provide interactive preview windows called a GUI editor and a scene editor, respectively. These display the result of the modifications of the parameters in the application immediately without re-running the application. In contrast, the programmer can also adjust the parameters for the location of a game object by directly moving the object on the preview window. However, these systems provide a highly structured framework, which limits what the programmer can edit such as rendering of the output window. Moreover, they cannot fast-forward, rewind, and/or stop the sequence of the application at their specific time just like a movie player.

To address these problems, we present CapStudio, a development environment for game applications (Figure 1). It has a traditional text editor and an interactive screencast, a movie player-like output window. A programmer can write the whole source code of the application in the text editor and edit the look-and-feel of the game window in the screencast. When the programmer edits a source code on the text editor or a resource file (e.g. an image file), the screencast displays the modification without re-running the application. We call this functionality *Forward Editing*. In contrast, the programmer can directly interact with a game object on the screencast to edit corresponding code snippets and resources. We call this functionality *Backward Editing*.

Although the screencast is an interface like a movie player, the system does not capture screenshots of the game window but records the rendering history (e.g. the location and the size of the game objects displayed on the game window) and reconstructs the game window with it. CapStudio also uses the rendering history to realize the Forward/Backward Editing.

Related Work

This research is inspired by Bret Victor's talk "Inventing on Principle" in 2012 [13], in which he proposed collaboration between the text editor and output window. However, he only explained the concept and did not give any technical details. On the other hand, we achieve this facility by introducing an interactive screencast and give the details of its implementation.

There are systems that provide captured output windows for debugging [1, 7]. Especially, *Whyline* [7] provides an interactive captured output window as CapStudio. When the programmer clicks on an object on the captured output window, Whyline navigates her to a code snippet that computed the property (e.g. color or location) of the object during the execution. However, unlike CapStudio, these debugging systems are not tools for editing a source code. They cannot reflect the modifications of the source code into the captured output window. Also, the programmer cannot edit parameters in the source code by interacting with an object on the captured output window.

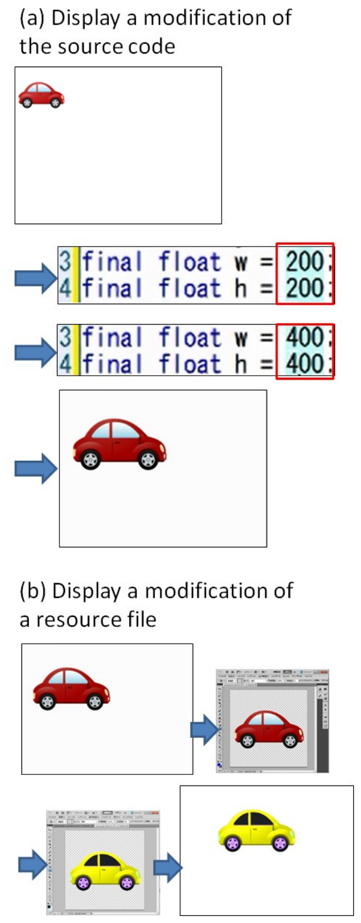
On the other hand, *Physics Storyboards* [4] provides a captured output window for parameter tuning. When the user tunes parameters for a physical simulation, the window displays the result of the modifications immediately. Also, when she specifies space-time regions in the window and declares an objective function for each region, the system automatically determines parameters that satisfy the objective functions. However, the captured output widow is not interactive, unlike that of CapStudio. The user cannot edit parameters by directly moving an object to a location where the object should be at a time point.

An approach for displaying code modifications without re-running the application explicitly is to compute the result of the modifications by re-building and re-running the application in the background [5, 6]. However, when the application communicates with an external process, this implicit re-running would modify the state of the external process unexpectedly. On the other hand, CapStudio records all of the necessary information during execution and re-computes the result of code modifications only with the recorded information. Therefore, our system does not disrupt any external processes or resources unexpectedly. Some live programming languages [3, 8, 11], which do not distinguish the running mode from the editing mode, can also display the code modifications without re-running. When the programmer develops a network application, however, these systems also need to keep the network connection during editing the application.

CapStudio IDE

We implemented a working prototype called CapStudio, an integrated development environment with an interactive screencast. Our current implementation is for Processing language [10], but we consider that it is possible to design this system for other languages. The interface of CapStudio is implemented as a WinForm application in C# and works on Windows 7 and 8.

The most important contribution of CapStudio is that it provides a text-based development environment with an interactive preview window similar to a scene editor of the game engines. Generally, it is almost impossible to predict the behavior of the application or generate a preview window. Therefore, we use a captured game window as a pseudo preview window. The system captures the rendering history of the game window to generate a captured game window as a screencast. It also uses the rendering history to achieve the collaboration between the screencast and the text editor (Forward/Backward Editing).


(a) Display a modification of the source code
(b) Display a modification of a resource file

```
3 final float w = 200;
4 final float h = 200;
```

```
3 final float w = 400;
4 final float h = 400;
```

Figure 2. Forward editing.

Generating a Screencast
First, the programmer writes a source code with a text editor on CapStudio and then runs the application. An output window pops up after the execution, and she can play her application on it.

During the execution, the system captures the call history of the application's rendering methods in background. Each time the application calls a method related to the rendering of the game window, the system captures information of the method call: the current location in the code, the values of the variables used in the argument expressions, and the return value if its type is not void (Figure 3). We suppose that the game window is rendered only with the Processing rendering API. Therefore, the system can identify the methods related to the rendering and the arguments that decide the location and the size of the game objects. When we want to use another rendering API such as OpenGL or Java Swing, we additionally need to implement the specification of the API in our system.

When the program exits, the system reconstructs the game window with the captured rendering history and shows it on the screencast window. The screencast basically works as a standard movie player with a seek bar, which allows the programmer to playback history of the application.

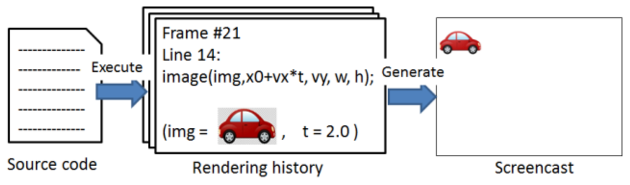
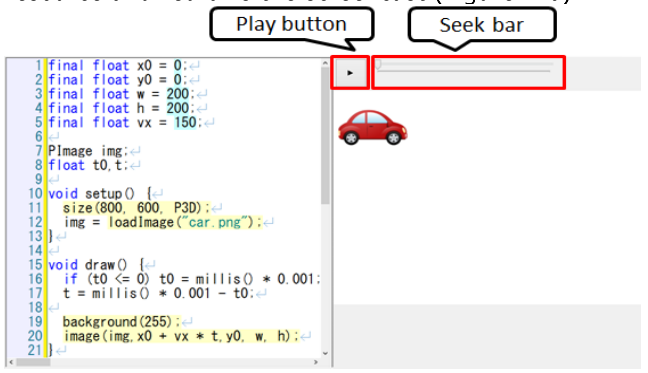


Figure 3. Generating a screencast

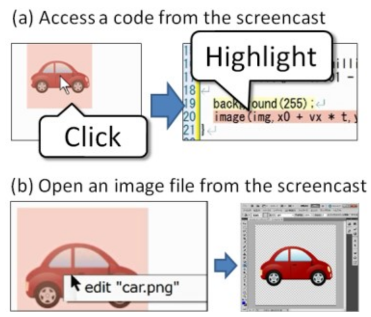
Forward Editing
CapStudio can update screencast immediately when the programmer changes the source code or a resource file. This functionality is called Forward Editing. After the application execution, code snippets related to the look-and-feel of the game window are highlighted. In CapStudio, they are the arguments of the rendering method calls (e.g. the arguments of image() at line 20 in Figure 4) and the constant values (e.g. the value 0 of x0 at line 1 in Figure 4). When the programmer rewrites one of these code snippets, the system reparses the source code and redraws the screencast with the new argument expressions and the constant values (Figure 2-a). Likewise, when the programmer edits a resource file, the system reloads the modified resource and redraws the screencast (Figure 2-b).



```
 1 final float x0 = 0;
 2 final float y0 = 0;
 3 final float w = 200;
 4 final float h = 200;
 5 final float vx = 150;
 6
 7 PImage img;
 8 float t0,t;
 9
10 void setup() {
11   size(800, 600, P3D);
12   img = loadImage("car.png");
13 }
14
15 void draw() {
16   if (t0 <= 0) t0 = millis() * 0.001;
17   t = millis() * 0.001 - t0;
18
19   background(255);
20   image(img,x0 + vx * t,y0, w, h);
21 }
```

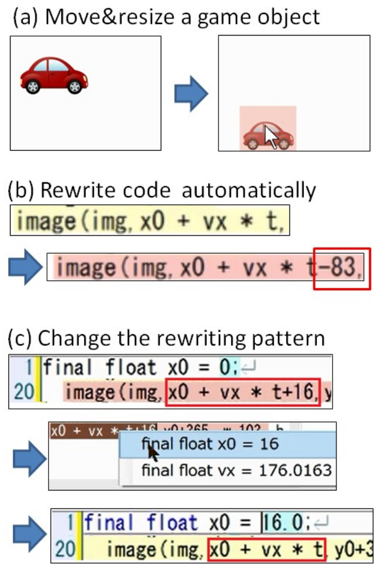
Figure 4. A simple application developed in CapStudio. The image of a car is in a uniform linear motion.

Backward Editing
All game objects in the screencast are clickable, and the programmer can interact with them to edit the code and resources. This functionality is called Backward Editing When the programmer left-clicks the screencast, the system finds which game object has been clicked


(a) Access a code from the screencast
Highlight
Click
(b) Open an image file from the screencast
edit "car.png"

```
background(255);
image(img, x0 + vx * t, y
```

Figure 5. Access code/resource from the screencast.


(a) Move&resize a game object
(b) Rewrite code automatically
(c) Change the rewriting pattern
final float x0 = 16
final float vx = 176.0163

```
image(img, x0 + vx * t,
image(img, x0 + vx * t-83,
```

```
1  final float x0 = 0;
20   image(img, x0 + vx * t+16, y
```

```
1  final float x0 = 16.0;
20   image(img, x0 + vx * t, y0+3
```

Figure 6. Edit the source code from the screencast.

by scanning all of the rendering method calls executed during the current frame. Then, the rendering method call that rendered the object is highlighted, and the cursor moves there (Figure 5-a). Likewise, when she right-clicks the object, a menu appears and shows a list of the resource files used in the corresponding method call. When she selects one of the menu items, the system opens the selected resource file with an external application (Figure 5-b). As described above, when she has edited this code or these resources, the screencast displays the result immediately.

The programmer can adjust constant values in the code by interacting with the screencast (Figure 6). She can change the position and the size of a game object by dragging it (Figure 6-a). Then, the system automatically rewrites the code so that the application renders the object in the same location and size in the next execution. As a default, the system adds constant values to arguments of the corresponding rendering method call (Figure 6-b). When she right-clicks one of the rewritten arguments, the other rewriting patterns are displayed (Figure 6-c). Each pattern modifies a value of constants appearing in the argument expression. The system chooses a constant as a rewriting target if it fulfills the following two conditions. 1) The constant affects only the arguments of the rendering method calls. For example, if a constant is used in an *if* statement, the constant is not a rewriting target. 2) It is possible to compute what value the constant should be. Currently, CapStudio can compute the value only if the argument expression is a linear expression of the constant. Therefore, if the argument expression is $a * c1 + b / c2$ ($c1$ and $c2$ are constants), only $c1$ is a rewriting target.

Additional Functions
In addition to the functions for code/resource editing, we implemented the following three functions by applying the screencast generation method (Figure 7). 1) Generating a thumbnail of a class by rendering only the game objects rendered in the member functions of the class. 2) Rendering a trajectory of a game object by overlapping images of the object rendered in other frames. 3) Shifting the camera by adding x-y offset values to corresponding arguments of the method calls.

Example Application

To discuss the feasibility of CapStudio, we created an example application (Figure 8). This application is a simple multi-user 2D game like Bomberman[1]. The users move their player characters and place bombs to kill the others. This game is a server-client system. Each user plays in a client process in his PC (written in Processing with CapStudio) and all the client processes communicate with each other through a server process (written in C#). In the game window of a client, the image of the player's character and a button to revive her player character are shown above the map.

We investigated two advantages of CapStudio from this implementation. First, the programmer can benefit from the Forward and Backward Editing. Once game objects are displayed on a game window during an execution, the programmer can move or resize them by directly dragging them on the screencast. She can also easily access the image files used in the application from the corresponding game objects on the screencast. Specifically, when she wants to access an image of a game object that appears only for a short time (e.g. an

[1] http://bomberman.jp/

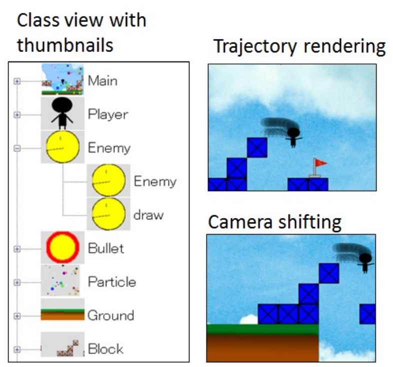


Figure 7. Applications of the screencast generation method.

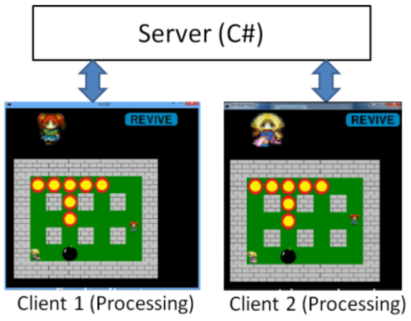


Figure 8. An example application.

explosive flame of a bomb), she can catch it easily by adjusting the seek bar, which was difficult even with a scene editor of game engines.

The other advantage is that the screencast works after the connection with the server is closed. When a programmer modifies the code, an existing method of re-building and re-running the application in the background [5, 6] cannot compute the results of the modification correctly. On the other hand, because the rendering history contains all log data of the communication, it can display the modification correctly.

Conclusion and Future Work

We proposed CapStudio, a development environment for interactive visual applications, especially video games, with an interactive screencast. The programmer can write the whole source code of an application in the code editor. At the same time, she can intuitively edit the look-and-feel of the game window in the screencast.

The problem with our approach is that the screencast cannot display all the modifications in a source code. Currently, the system observes only modifications of rendering method calls and constant values. Therefore, when the programmer rewrites a code snippet related to logic such as a collision-detection among the game objects, the screencast cannot display the modifications. Our system is for editing the look-and-feel of the game window, such as colors or the arrangement of a game object. CapStudio currently supports only subset of the processing rendering API. It supports the rendering methods for developing a 2D game such as loading and rendering an image. In the future, we will support the fullset of the rendering API including 3D rendering methods. Likewise, we will implement an interface to handle a game object on the screencast three-dimensionally. Also, we will support low-level rendering APIs such as OpenGL and Java Swing.

References

[1] Burg, B., Bailey, R., Ko, A.J. and Erust, M.D. Interactive Record/Replay for Web Application Debugging, I*n Proc. UIST 2013*, 473-484.

[2] Eclipse. http://www.eclipse.org/.

[3] E dwards, J. Subtext: Uncovering the Simplicity of Programming. In *Proc. OOPSLA 2005*, 505–518.

[4] Ha, S. McCann, J. Liu, C. K. and Popovic, J. Physics Storyboards. In *Computer Graphics Forum 2013*, 32, 133-142.

[5] Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S.R. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST 2008*, 91-100.

[6] Kato, J., McDirmid, S. and Cao, X. DejaVu: Integrated support for developing interactive camera-based programs. In *Proc. UIST 2012*, 189-196.

[7] Ko, A.J. and Myers, B. A. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. ICSE 2008*, 301–310.

[8] McDirmid , S. Living it up with a live programming language. In *Proc. OOPSLA 2007*, 623–638.

[9] Microsoft Visual Studio. http://msdn.microsoft.com/ja-jp/vstudio.

[10] Processsing. http://processing.org

[11] Self. http://selflanguage.org/

[12] Unity3D Game Engine. http://unity3d.com.

[13] Victor, B. Inventing on Principle. http://vimeo.com/36579366.