Programming as an Interactive Experience

Andrej Ivanis

4th Year Project Report
Computer Science and Electronics
School of Informatics
University of Edinburgh

2016

Abstract

Have you tried using a code editor that underlines syntax errors as you type? Isn't this easier than having to compile and see that you made an error, then go back and add a single bracket you forgot? Modern editors also automatically complete the code, not forcing you to memorise the exact name of every function in the library. As computers become more powerful we can expect more features like these. Watch the video "Inventing on Principle" by Bret Victor and you will see that there are many more interactive features we can add to the next generation of code editors. This video inspired me to do this project.

I made a JavaScript development environment that interprets the code as you type it. It also lets you change the hard coded constants using sliders, switches and colour pickers. Rest of the features are made for game developers, animators or graphics designers. You can quickly find which part of code draws which part of picture by just clicking on that part of the picture. If you are making a game or animation you can rewind it like it is a video, change the code and replay it with the new code or just look at the player's motion path on a still image. As you change the code the path is changed immediately. If you played around with animation tools like Flash, you may know this as onion skinning.

The finished system was evaluated using speak-aloud protocol by 7 potential users.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Andrej Ivanis*)

Table of Contents

Chapter 1

Introduction

1.1 Inspiration: Inventing on Principle

I got inspired to do this project after watching the video "Inventing on Principle" by Bret Victor [16] where he showed how interactive programming can be. Many features were demonstrated, but I will show you ones that got my attention because of my interest in video game programming. First demo showed a picture of a cartoonish tree on the left side of the screen and some JavaScript code on the right side. The code was using Canvas API [11], which is a library for making animations or games within webpages, and Bret used it to draw the tree programmatically. Nothing special about it, until he stared changing the code and the tree was immediately updated. The code was executed as he was typing it. When he clicked on some number in the code a slider would pop up above it as you can see in Figure 1.1. As he dragged the sliders tree would change the height or number of leaves or any other property immediately. Similarly, he could change the colour of the leaves or the sky with colour pickers that popped up when he clicked on a string representing an RGB colour in hexadecimal format, for example "#FF4521".

As Bret moved the mouse over the sky the line of code that drew the sky was highlighted as in Figure 1.2. Whole sky turned red so you could clearly see the region drawn by this line of code. Then he clicked on some line bellow the highlighted one and sun would turn red so he immediately knew: this is the line that draws the sun. He had a two-way interaction between the code and the output program.

Then Bret demonstrated a game where a player could move around, jump on platforms and bounce of enemies, similar to Super Mario Bros. He also had a time rewind slider, so he could walk around a bit, rewind and watch the gameplay again like it was a video. After rewinding he changed the jump height in the code and the player was jumping higher while doing the same moves as before. What was even more impressive, he had onion skinning, which is a feature of animation tools used to show previous frames while you

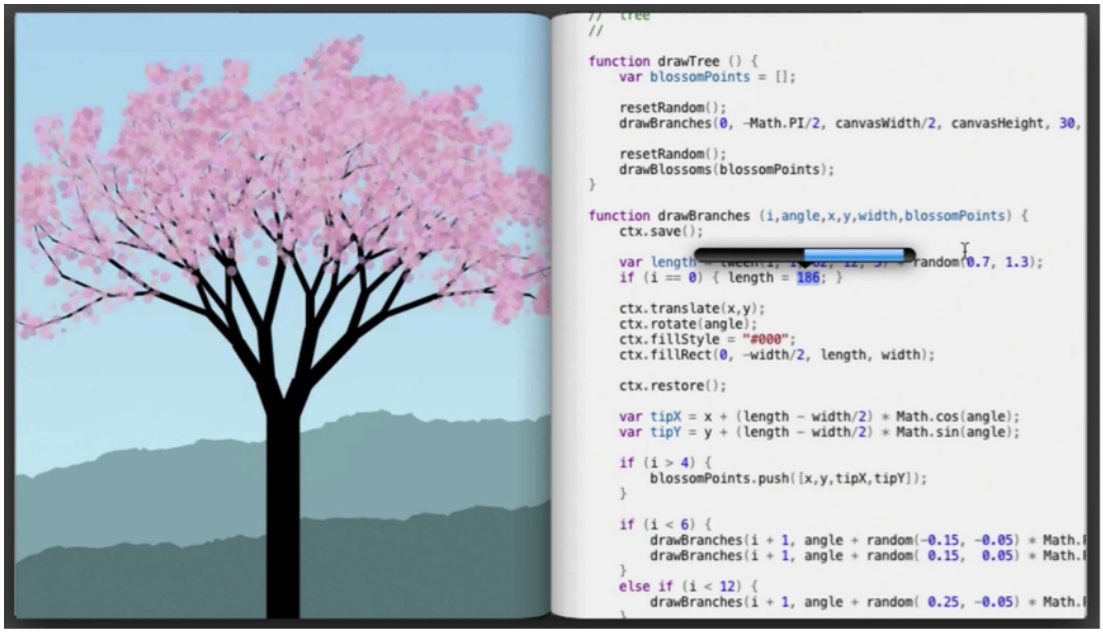

Figure 1.1: Inventing on Principle, A slider

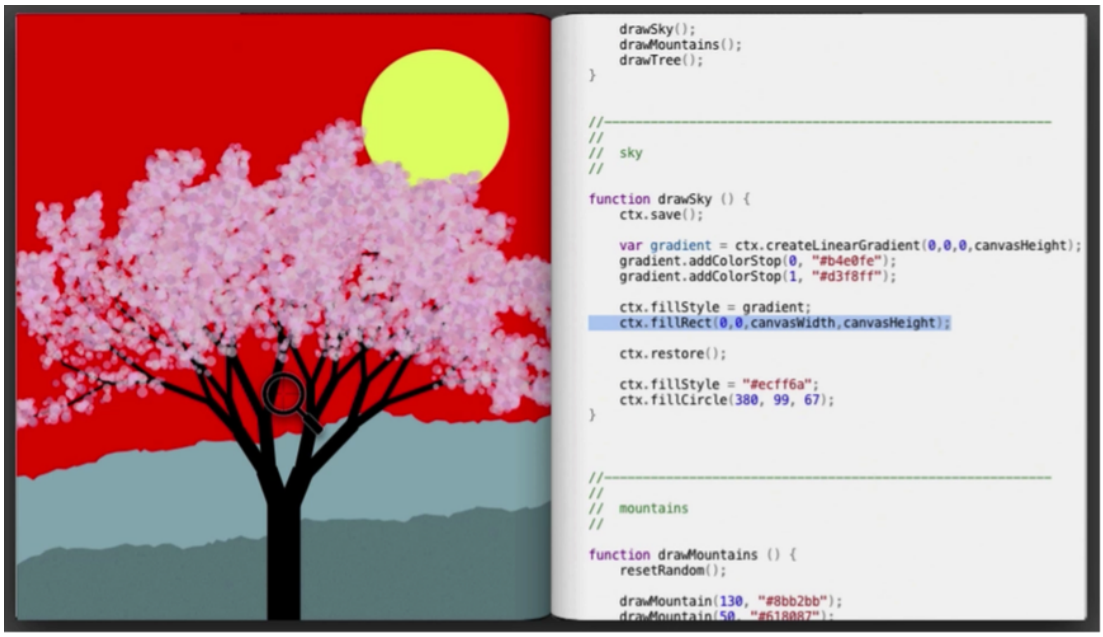

Figure 1.2: Inventing on Principle, Sky highlighted

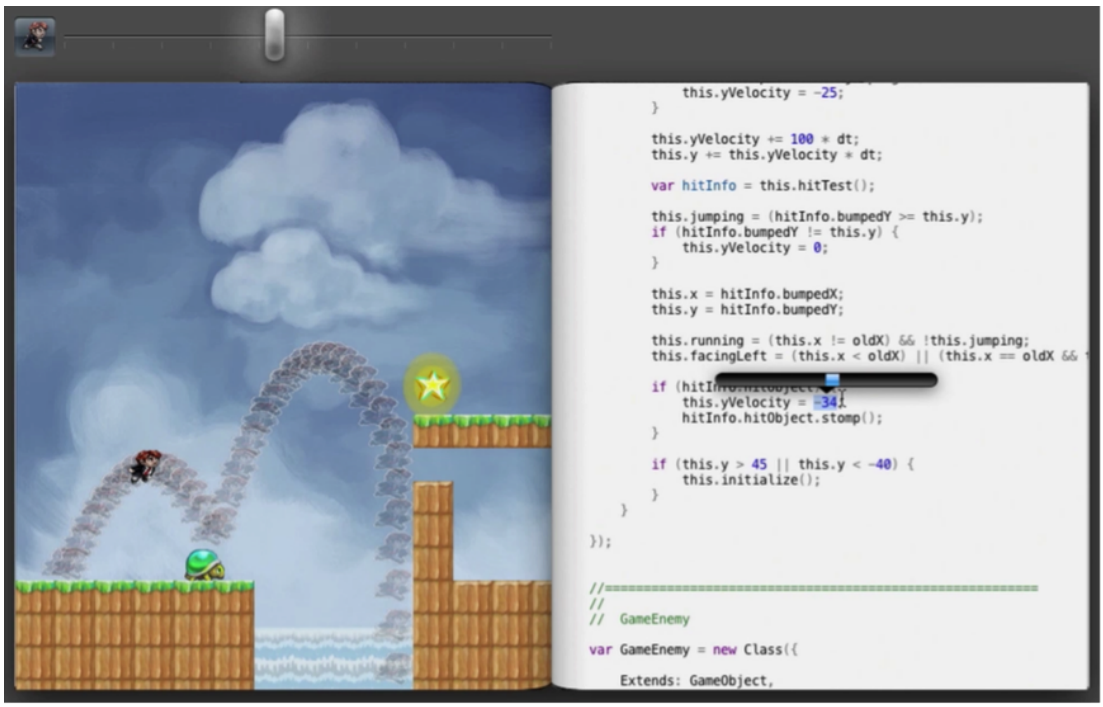

Figure 1.3: Inventing on Principle, Onion skinning

draw the next frame. In Bret's case player left a trail of semi-transparent copies like in Figure 1.3 that was changing as he was changing the jump height.

None of Bret's demonstrations show the implementation. We don't even know if the demos are implemented or just a flashy animation.

1.2 Previous work

Has anybody implemented any of Bret's ideas? There are some attempts.

1.2.1 tributary.io

The most popular attempt to implement these ideas is a website "tributary.io" [15]. They made the sliders and the colour pickers, but the highlighting of an object drawn by a line of code is missing. Animation features are limited. Their animation library only supports animations that are expressed as function of time. You can't make a game as you can't use mouse or keyboard to control the animation. This is not generic enough to be included in some IDE.

1.2.2 MInf Project by Craig Innes

Craig Innes did MInf Project at University of Edinburgh on the same topic "Programming as an Interactive Experience" [4]. This is the most complete implementation of Bret's ideas. All features of the tributary.io are there and you can highlight the object drawn by a line of code and vice versa. Some of the ideas are even extended, for example, you can move objects in the output image and the code changes. Craig did some work on animation like added onion skinning: you are able to see couple of frames in the future, you can rewind it and even fast forward it.

The animations in his system must be defined using functional programming, but what is mostly used in games/simulations/animations is imperative approach. We usually have an update loop, executed many times per second, which moves the objects a tiny bit every frame. The loop, for example, checks if an object is going to collide with another one and moves them away or if the left arrow is pressed it makes the player goes left.

Craig commented about his time rewind feature:

> This simple implementation of this feature would be significantly more complicated in the imperative style of animations. In the imperative style, the state of the animation is altered slightly on each loop, so in order to rewind time, one would have to figure out how to reverse each of those incremental changes made on each loop. [5]

My approach is different. Instead of figuring out how to reverse the changes I will record the memory every frame so I can jump back anytime.

1.2.3 Light Table

[10] Unlike the two above, which are research projects, this is actually fully-featured IDE and is already used by developers. It contains only the basic idea: user can see the output program as soon as he changes code. It is worth mentioning as it shows that next generation of IDEs will be more interactive.

1.3 Project Goal

My goal was to implement following features:

- Immediate interpretation - The resulting image should change immediately after the code is changed.
- Graphic interface for hard coded values

Sliders: A slider should pop up when you click on a number and by dragging it the number should change.

Colour pickers: A colour picker should pop up when you click on a string that starts with # and change the string when you select the colour.

Boolean switch: A button should pop up when you click on a boolean value (true/false). Clicking it should change between true and false. This is not in Bret's demo, but users expect this after they see sliders and colour pickers.

- Code line highlighting - When you click on a part of the image a line of code that drew that part should be highlighted. Also this goes the other way, when you select a line of code the part of the image drawn by that line should be highlighted.
- Partial interpretation - If we reinterpret the whole program in a traditional way, where we stop the old script and load the new one, player would be sent to the beginning of the level every time we make the smallest change because the position values will be reloaded too. We don't want the player to be affected when we change the colour of a platform so only part of the program that is changed needs to be interpreted.
- Time manipulation - Gameplay should be recorded and there should be a slider to rewind it like it is a video. Any change in the code should affect the gameplay.
- Onion skinning - You should be able to see future frames as semi-transparent overlays of the current frame.

I am not making fully featured IDE nor focusing on any features IDEs already have. The goal is to experiment with the features that may be added to the next generation of IDEs and evaluate them with potential users to measure their usefulness.

Chapter 2

Implementation

Figure 2.1 shows the finished project. It is a webpage with similar layout to Bret's demo. Why did I make a web page? Well, I got cross-platform compatibility and easy setup, which helped with user evaluation. More importantly, I got JavaScript interpreter as part of the browser. JavaScript scripts have infinite freedom to change themselves. For example, you can ask user to type some code and make that code become the part of the program that asked for the input, using single function call:

```
eval(user_input);
```

There is no limit on what the user can do, he can input:

```
eval = 5;
```

After this eval function will no longer exist, it will become a integer variable with value 5. This makes it simple to change the program while it is running, something that would be much harder to do with other environments and programming languages.

Every modern web browser comes with a 2d drawing library called Canvas [11]. It is already integrated and incredibly simple for anybody who wants to try this project so I'm using it to draw the output image on the right. Bret used JavaScript and Canvas in his demo too, but I realised that after I decided to use them too.

On the left side of the screen is a code editor called Ace [1]. It makes the code properly formatted, highlights the keywords and checks the syntax as you type. If you want to autocomplete the word you are typing press ctrl+space and you will see the suggestions. This is a web equivalent of popular code editors. I haven't used online code editors before and Ace had simple API (application programming interface) that can be learned quickly.

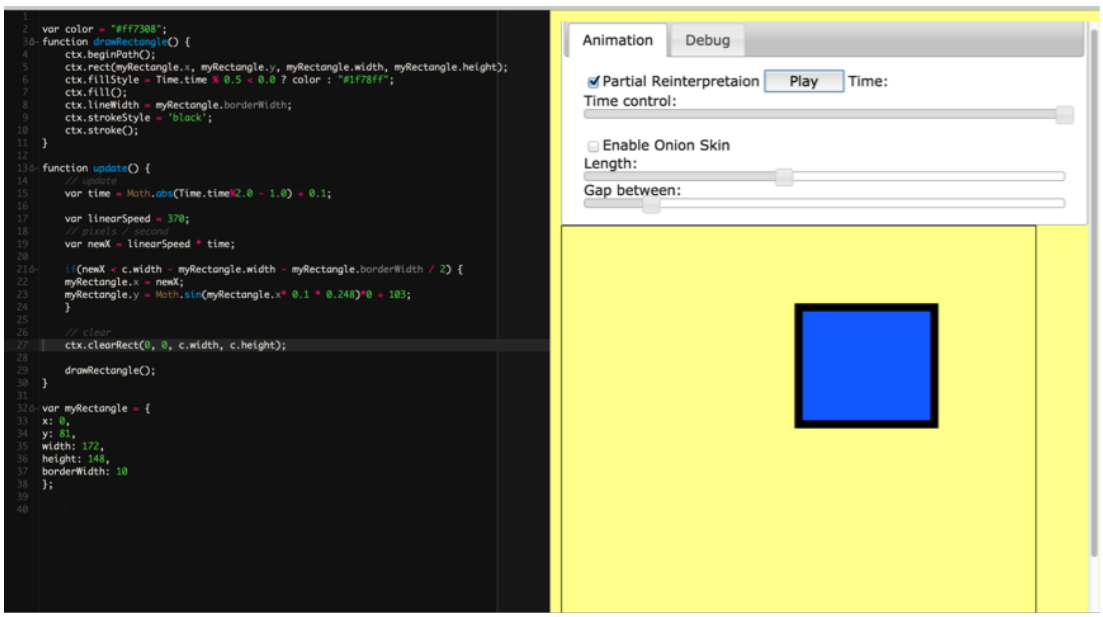

Figure 2.1: My implementation, finished project

2.1 Immediate interpretation

First of all, the image has to change as you change the code. The Ace text editor has an event happening when the code is changed and I made it evaluate the code using globalEval function [7]. This was enough to get the output immediately, but it has a limitation, if you add code that schedules a timeout, that code will be reinterpreted on every keystroke and therefore scheduling new timeout every time. There is no easy way to track current timeouts and cancel them so if I was to fix this I would have to replace built in library functions. This was a big task so I decided not to support them.

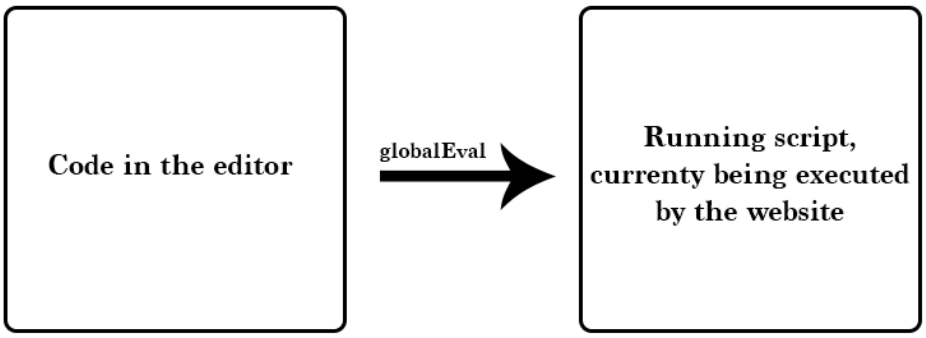

Figure 2.2: Immediate interpretation. On each keystroke content of code editor is loaded and replaced dynamically using globalEval

2.2 Graphic interface for hard coded values

```
68
69
70
71    if((ball.x < 0 && [redacted] ball.x > 600 && ball.vx > 0)){
72        ball.vx = -ball.vx * 0.9;
73    }
74    if((ball.y < 0 && ball.vy < 0) ||(ball.y > 500 && ball.vy > 0)){
75        ball.vy = -ball.vy * 0.9;
76    }
```

Figure 2.3: My slider

```
18      ctx.arc(ball.x, ball.y, 24, 0, 2 * Math.PI, false);
19      ctx.fillStyle = "blue";
20      ctx.fill();
21      ctx.lineWidth = 5;  003302
22      ctx.strokeStyle = "#003302";
23      ctx.stroke();
24  }
25
26  var ball = {x: 49, y: 32,
27
28  function goTo(pos, vel){
29      //if(pos[0])
30
```

Figure 2.4: My colour picker

```
79  }
80
81  var grid = [                    false
82              [false, false,  false,  false,  false,  false],
83              [false, false,  false,  true,  false,  false],
84              [false, false,  false,  false,  false,  false],
85              [false, false,  false,  false,  false,  false],
86              [false, false,  false,  false,  true,  true],
87              [false, false,  true,  false,  false,  false],
88              [false, false,  false,  false,  false,  false],
89              [true, true,  false,  false,  true,  false],
90              ];
91
```

Figure 2.5: My switch: On click value flips between true and false

There are many web GUI (Graphic User Interface) libraries available online. I chose the popular jQuery UI [8] for sliders and buttons, and jscolor [9] for colour pickers. Then I used Ace's built in functions to calculate the pixel position of a character on the screen so I know where to put the slider, button or colour picker. Ace can replace the words programatically so I can change the number as the slider is moved or change a colour string with the colour picker. Similar goes for boolean buttons.

I also check if the selected number is decimal or integer. Decimals should stay decimals and integers should stay integers. If you have a loop doing 5

iterations you don't want to see 5.531 after you moved the slider, but 6, 7, 8 and so on.

If the number 5 is selected and you move the slider to the far right you get 55. If you move it to the far left you get -45 so the range is ±50. But if we select the number 1000, we don't want to move the slider all the way right and have 1050 so I made that if the number is not between -25 and 25 the range becomes ± double that number so between -1000 and 3000. These limits seamed reasonable to me, but if you are implementing an IDE you may want to add setting for these ranges. I also made up rules for decimal numbers: if the number is between -1 and 1 range of the slider is ±1, otherwise the range of the slider is ± double that number.

2.3 Code line highlighting

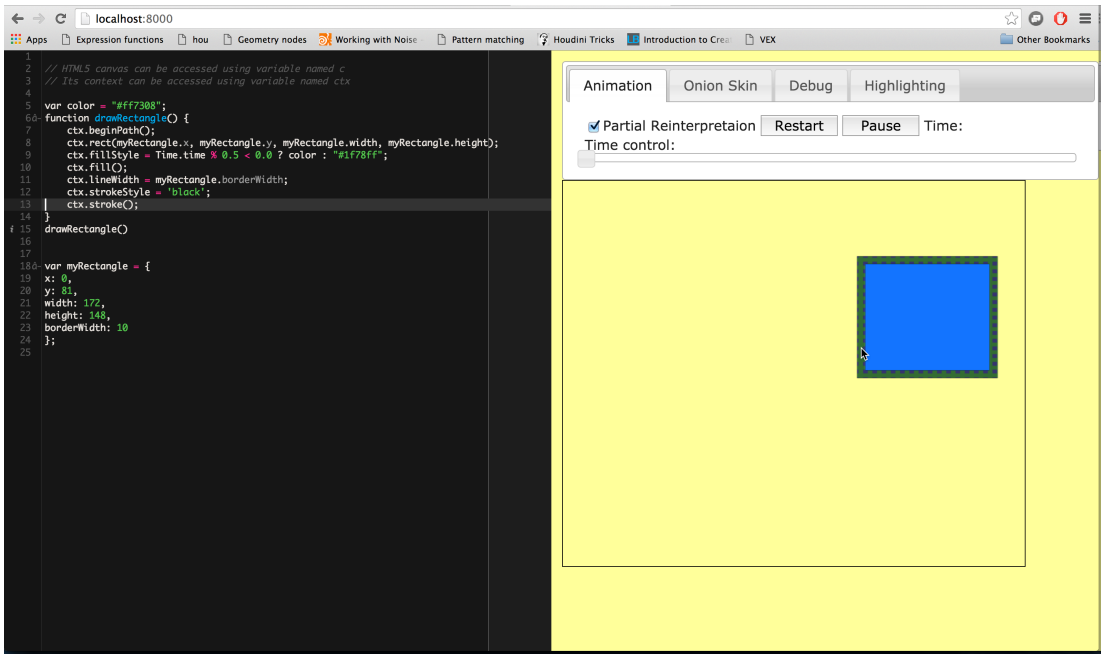

Figure 2.6: Mouse is over the stroke of the rectangle so drawing function for the stroke is automatically highlighted

Each pixel of the output image is represented using 32 bits. Those values are interpreted as a colour by graphics card, but they can represent whatever you want. I use them to store integers. There is a separate image (Figure 2.8) of the same size as the output image and every pixel of this new image is actually a 32 bit number representing a line of code containing the function that drew this pixel.

I mentioned previously that you can easily modify JavaScript function just by redefining them. I redefined all of the Canvas drawing functions. Apart from

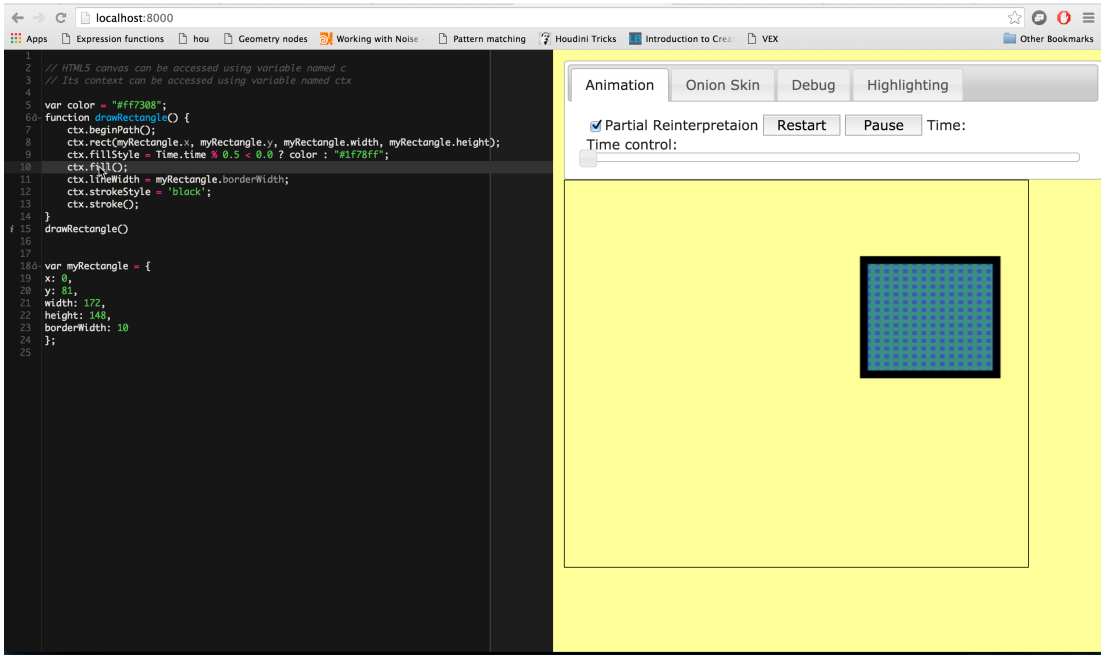

Figure 2.7: Line of code for filling the rectangle is selected so inside of the rectangle is highlighted

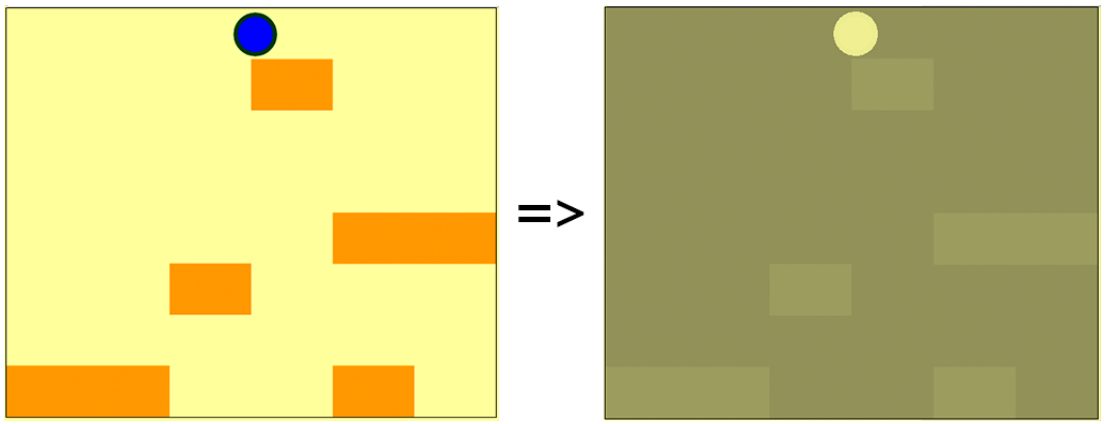

Figure 2.8: Invisible code line canvas. Each colour represents a line of code

drawing the shape only on the output image I draw the shape one more time on this invisible image containing lines of code, but I colour it according to the current line of code. Colour has 4 channels: RGBA for red, green, blue and alpha (transparency), each of them is 8 bit so I encode a 32-bit integer representing the line of code as a colour.

When the mouse is over the output image I get the location of the pixel right under the mouse and look up the colour of the pixel at the same location on the invisible code line image. That colour represents the number of the line in the code, which I need to highlight.

If you click on an object, the highlight will stay on that object so you can safely move your mouse back to the code and edit it without jumping to every object you move the mouse over. When mouse is out of the canvas highlighting will be unlocked again.

2.3.1 Getting the current line number at runtime

When you throw an error in JavaScript and catch it you get a complete stack trace with numbered lines of code. So I made every drawing function throw and catch an error so I can get current line of code by parsing stack trace. Every browser has different format of the stack trace, but I used stackframe [13] library for parsing them.

2.3.2 Highlighting part of the image

If a line of code is selected the part of the output image drawn by that line should be highlighted. So I overlaid a transparent canvas on top of the output image and if a line of code is selected I look up the code line image for pixels that are drawn by that line and colour them in the overlay canvas. You can see that in Figure 2.7, instead of solid colour overlay that Bret used I used this turquoise dotted pattern as it is easier to spot if object has similar colour as overlay.

The overlay canvas can be coloured by going through code line canvas pixel by pixel and checking if they belong to the selected code line. This can be slow for large output images so instead I wrote WebGL fragment shader [12], which is a program executed on the graphics card in parallel for each pixel of an image. The shader gets the code line image and selected line of code, then it colours the part of the image drawn by the given line of code with the turquoise pattern, makes the rest transparent and outputs it to the overlay canvas.

2.3.3 Alpha blending

I mentioned before 32 bit integers representing lines of code are encoded as RGBA colour and drawn on an invisible code line image. If I was creating this image using the Canvas API, I would get into trouble every time A channel is not 255, because it is made to represents transparency, not last 8 bits of an integer. The new semi-transparent shape will blend with the background mixing its colour with the background colour according to transparency. I don't want this to happen, when I draw a shape on the code line canvas it needs to store the exact RGBA colour I gave it. Unfortunately, there is no way to turn off alpha blending in the Canvas API. I might have used only RGB channels and leave A at 255, which means the image is opaque, this would give me 24 bits

which is over 16 million lines of code. This would be fine as Ace editor supports around 4 million lines, but there is no way to ensure alpha will be 255 for all drawing methods. Method drawImage is used to draw an image you specify inside the rectangle and it uses alpha channel from the image and there is no way to ignore it and set alpha to 255. So I made another WebGL fragment shader to ignore blending. The input of the shader is the current code line and the image produced by that line and it outputs RGBA values to the code line canvas without blending.

2.3.4 Performance

At the end, apart from drawing a shape once I draw it second time and send it to WebGL fragment shader, which draws it third time on the overlay canvas. So it is 3 times slower then just drawing a shape without code line highlighting plus it is expensive to send an image from Canvas to WebGL on some systems [14]. As WebGL is a new technology you may have trouble using this with slightly outdated browser, I have tested and developed this using latest version of Chrome. This would work better when implemented as an actual part of the web browser or some IDE as client side JavaScript has too many restrictions. This way it could be 2 times slower instead of 3 times as alpha blending can be disabled and you can draw the shape only twice as you don't need to use 2 libraries and copy it between them to avoid alpha blending.

2.4 Animation

```
function update() {
    ctx.clearRect(0, 0, c.width, c.height);
    if(state == "right"){
        ball.vx += Time.deltaTime * accel;
    }
    if(state == "left"){
        ball.vx -= Time.deltaTime * accel;
    }
}

function onkeydown(event){
    if(event.which == "D".charCodeAt(0)){
        state = "right";
    }
    if(event.which == "A".charCodeAt(0)){
        state = "left";
    }
}

function onkeyup(event){
    state = "idle";
}
```

Figure 2.9: Functions provided to make animation easier with example code

I provide the programmer with an update loop. I also give him the time since

the beginning of the program and the length of the current frame (deltaTime) as global variables. And I added functions that are executed when keys are pressed or released. This is what most of game engines do for you and it will simplify both my task to track the loop and record the keystrokes and programmer's task to program a game. Figure 2.9 shows an example with these functions.

2.4.1 Partial interpretation

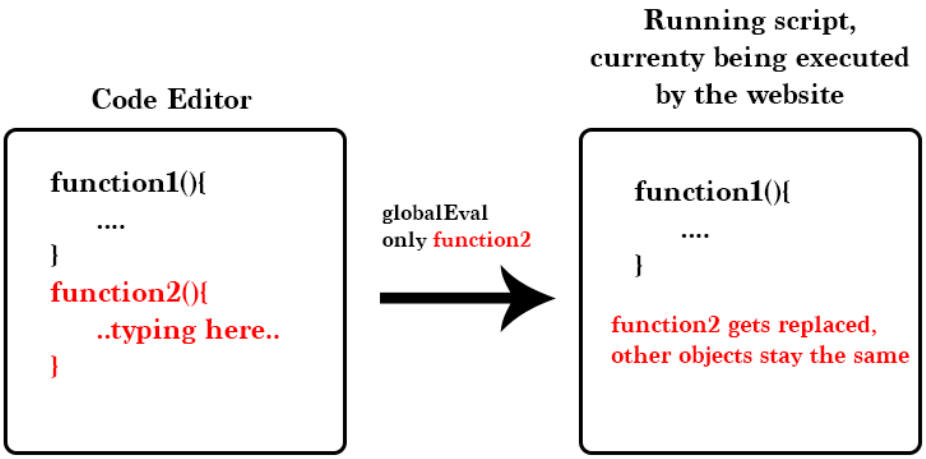

Figure 2.10: Partial interpretation. Edited global scope object is replaced dynamically using globalEval

If we program a platformer or a side scroller and come near the end of a level only to realise that last platform is too far away to jump to. If we move the platform by changing the code, then whole code will be reevaluated and level will reset so we can't test if the platform is close enough now. A way around this is to find which part of the program that has been changed and evaluate only that part without affecting the rest.

JavaScript has no encapsulation, so you can overwrite any variable or function you want. Go to facebook.com and type "Array = null" in web browser's console and you can't scroll the page any more or open a menu. You just deleted built in Array object and most of the scripts will fail. How do I use this in my tool? When user changes part of code, firstly I find which global object or function does that part belong. This was done by constructing an AST (abstract syntax tree) and going from the changed token up the hierarchy to the global scope. Then I just evaluate the last object I visited before reaching the global scope. If the object exists in the global scope it will be replaced or if it doesn't it will be created. For example, if you modify the background colour only background colour will be replaced in running script, leaving the position

of the player the same. Most likely this is what you want, but if you want to reevaluate whole script there is a restart button in the graphical interface. Also if a user deletes an object from the code, I made it so it is deleted from the global scope to avoid a memory leak.

The AST construction was done using esprima [3] parser. After the AST is constructed I find to which global scope object does changed code belong and I pass the syntax tree of that object to escodegen [2], which is a library that generates JavaScript code from the given AST. Browser's parser and AST interpreter are not accessible from JavaScript and there is no eval function that accepts AST instead of code so I have convert it back to code this way.

2.4.2 Time manipulation

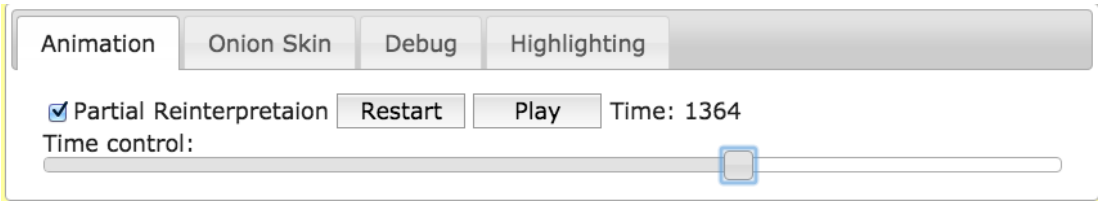

Figure 2.11: Time manipulation: Graphical Interface

The animation is recorded by taking a snapshot of the global scope every frame, that means making deep copy of all objects in the global scope, storing keystrokes pressed and released during that frame and storing duration of that frame. Deep copies are made using built in jQuery's extend function [6]. Making a deep copy of a large program can take a lot of memory so I store only 2000 frames in a fixed size array, partially shown in Figure 2.12. This is equal to 33 seconds at frame rate of 60 frames per second. To save some memory I store only objects and leave out functions. Although JavaScript allows for the body of the function to be changed dynamically at the run-time that is used rarely. I haven't exposed the settings for these optimisations in the graphical interface, but they can easily be added. The frame array can be as large as the memory of user's computer allows. There is a slider so you can scroll backward and forward in time. When you move the slider a snapshot is copied back to the global scope and therefore values of all variables are replaced with ones stored in the snapshot.

Lets see happens when you scroll back in time and change the code. The changed code will be evaluated, the current snapshot is modified and a dirty flag is set for that snapshot. If you scroll forward in time all the snapshots between the dirty one and the current point in time will be reevaluated one by one and the data from previous one is passed to the following one. I actually simulate program's execution, but I disable any drawing functions to make it run faster. After evaluation of each snapshot I store the new data back in the

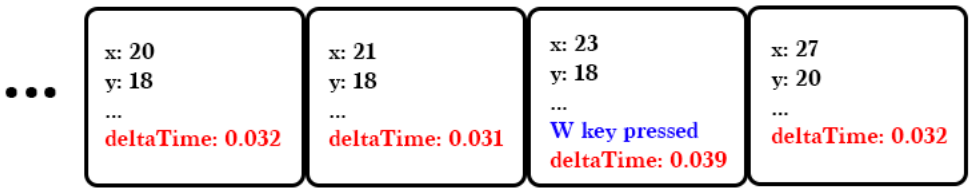

Figure 2.12: Time manipulation: Snapshot buffer

snapshot array and move the dirty flag to the last evaluated snapshot. Dirty flag tells the system that any frames before it are up-to-date and it doesn't need to recalculate them if user slides back. Time and length of frame variables as well as keystrokes are left unchanged in the snapshots because we want to simulate what would happen if the game was played exactly like the first time but the new code is used.

2.4.3 Onion skinning

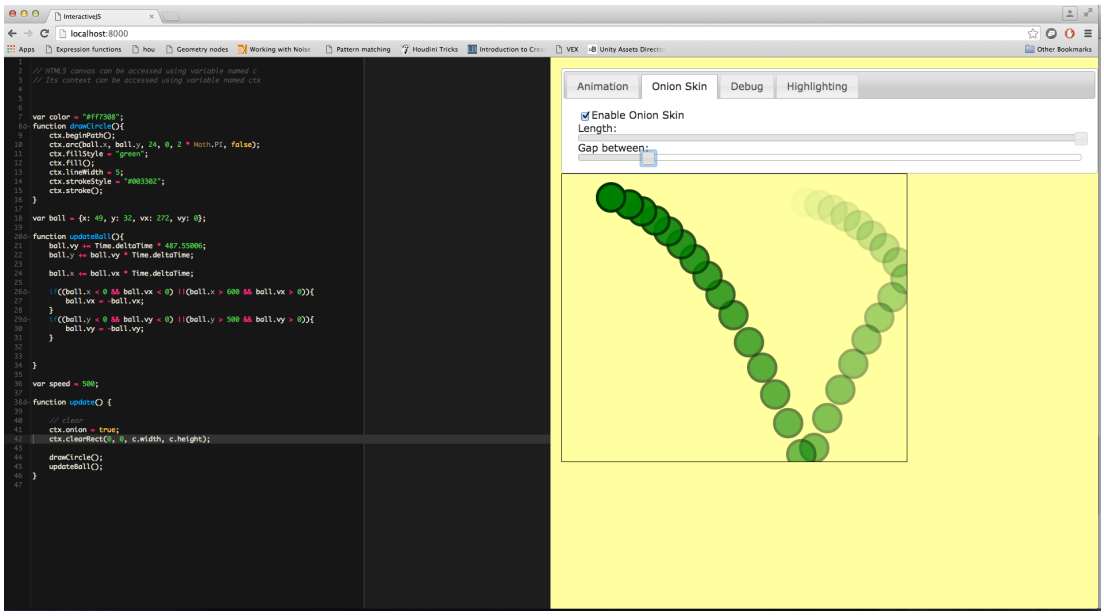

Figure 2.13: Onion skinning: Ping pong demo

With the snapshot system in place, I can jump around the program, skip frames and still have exact representation of what will happen. Onion skinning draws the current frame with full opacity, skips a couple of frames using snapshot system, decreases opacity and draws a frame again and so on. In Bret's demo of onion skinning all frames were rendered with same opacity, but this way you can't see the direction of the player's movement. The number of frames

rendered and the gap between frames can be set by sliders above the canvas. I also added a property ctx.onion that user can set to false if he wants to ignore an object from onion skinning, for example the background. My implementation of onion skinning is shown in Figure 2.13

Chapter 3

Evaluation

System was evaluated by 7 undergraduate Computer Science students at University of Edinburgh.

3.1 Evaluation Process

Each of the students evaluated the system separately and evaluation followed speak-aloud protocol. Each evaluation session was recorded and students were giving their suggestions while trying out the system. I decided to do this type of evaluation because I wanted to give each evaluator as much time as he wants to get familiar with the system and try out his own ideas apart from only doing tasks I gave him. I printed a script to use during evaluation which was more like a plan so that the process wouldn't differ to much between users, but as I said they were free to experiment with the system. None of them had any experience with HTML5 canvas, which is the drawing API used in the project. Only few of them said they heard the name HTML5 canvas, but I gave them all the functions they need to use during evaluation. You can find my notes and scripts used during evaluation in appendix.

Evaluation session started by giving users simple tasks: draw a rectangle, change the width and height using sliders, draw more rectangles and hover over them to see which line of code drew each of them. Then I showed them the ping pong demo and the keystrokes demo.

The ping pong demo is the one shown in Figure 2.13. I let the users play around with the gravity and see how the onion skinning updates and displays many frames immediately. Keystrokes demo is shown in Figure 3.1 and it lets you accelerate the ball using WASD keys. The ball's inertia makes it hard to change direction. When you hit an orange rectangle you lose and red Game Over text appears, after this happens users need to rewind, turn on the onion skinning and modify the positions of orange blocks so the ball passes between them. This shows how onion skinning can be a level design tool as you can check if

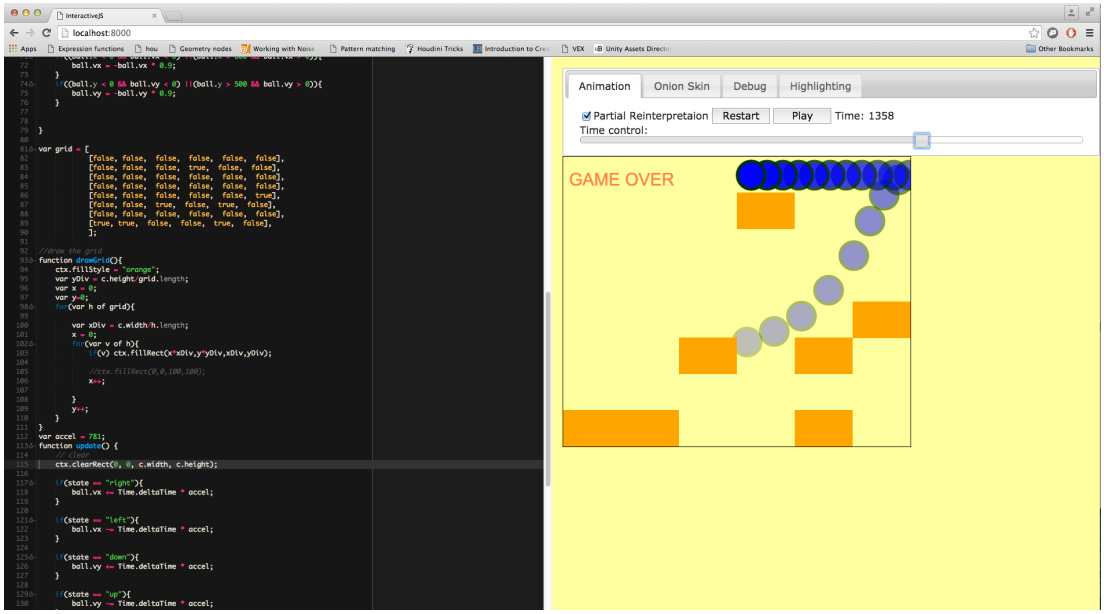

Figure 3.1: Onion skinning: Keystrokes demo

the level is possible to complete while designing it. To design a new level start without any rectangles on the screen, move the ball around randomly, then rewind, turn on the onion skinning and add the blocks around the motion path of the ball.

After completing all the tasks I gave evaluators the feedback sheet shown in Figure 3.2 to fill in.

Four of the evaluators did basic tasks only, while 3 of them were really interested in how the system works and wanted to know implementation details. One of the evaluators made his own procedural image made of diagonal array of smaller and smaller rectangles which gave it 3D look that was adjustable using sliders. Another evaluator made an animation of different shapes moving around and changing colour. Average evaluation session lasted 30 minutes.

3.2 Evaluation Results and Analysis

3.2.1 Immediate Interpretation

Immediate interpretation got good reviews. In users words:

> That definitely happened.
>
> It was very immediate.
>
> It doesn't get any more immediate.

Programming as an interactive experience
Feedback Sheet

Rate features below between 1 (bad) and 10 (good):

	Feedback was immediate (no lag)	The feature is easy to use	The feature is useful
Immediate Interpretation			
Sliders/Colour Pickers			
Code line highlighting			
Time Manipulation			
Onion Skinning			

Figure 3.2: Feedback sheet

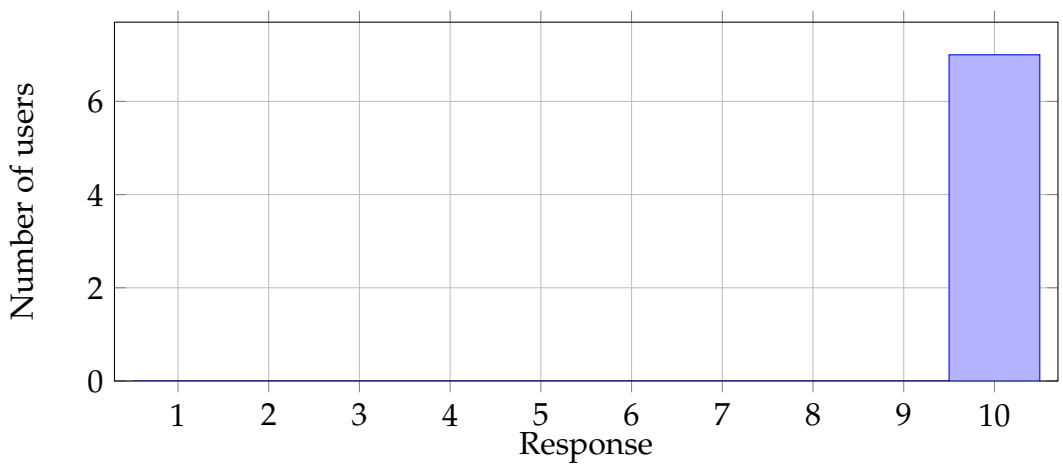

Figure 3.3: Immediate Interpretation: Feedback was immediate

However, one user discovered a problem with immediate interpretation. His program got stuck in an infinite loop when he wanted to change a condition in a for loop. As he deleted the terminating condition the website froze. The problem is that I'm running the editor and the user's code in the same thread, so when the user makes an infinite loop the whole website gets stuck too and

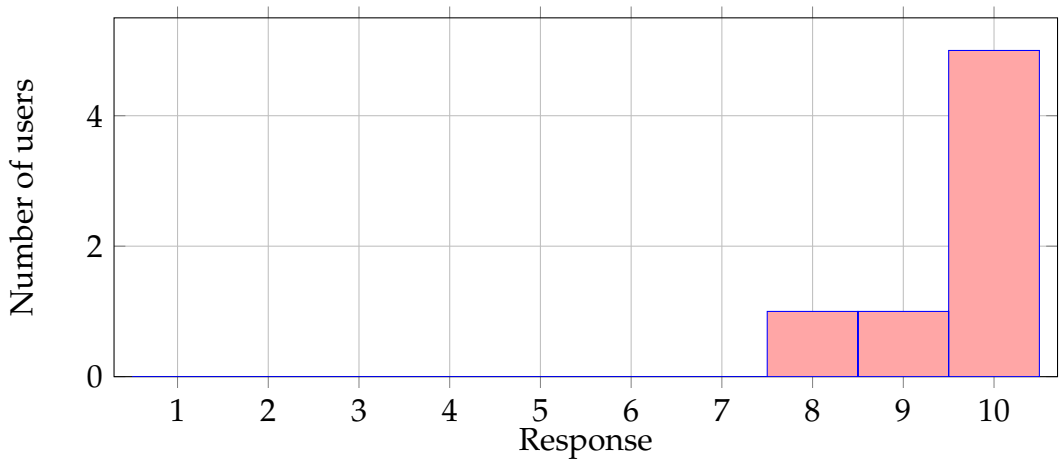

Figure 3.4: Immediate Interpretation: The feature is easy to use

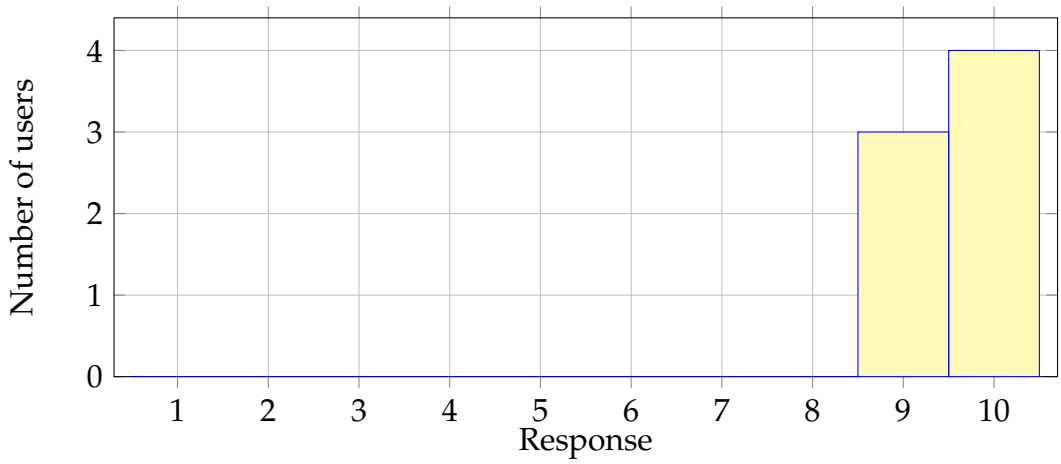

Figure 3.5: Immediate Interpretation: The feature is useful

it can't stop the execution of the loop. Solution is to separate them in different threads, but this is not possible to do in a webpage.

3.2.2 Sliders/Colour Pickers

Again good results. Evaluators especially liked this feature and described it as cool and neat. In their own words:

> Definitely useful, especially the colour picker.
>
> I like these colour pickers and sliders. That's neat!
>
> I really like the sliders.
>
> I really, really like this, especially the colour picker. That is very useful. It is amazing.

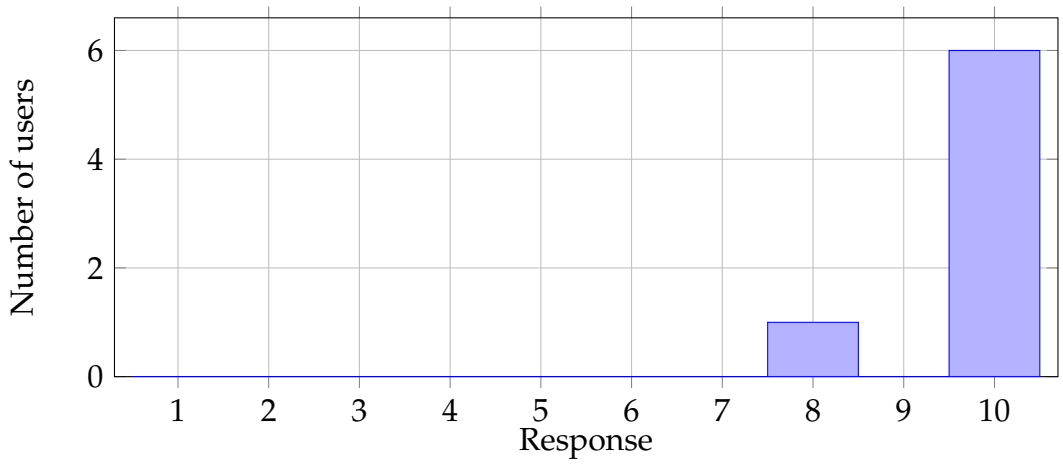

Figure 3.6: Sliders/Colour Pickers: Feedback was immediate

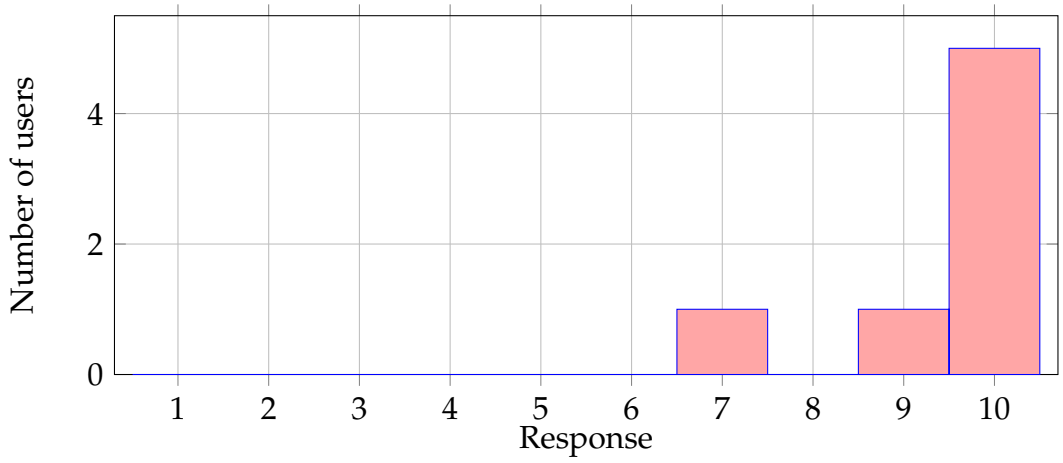

Figure 3.7: Sliders/Colour Pickers: The feature is easy to use

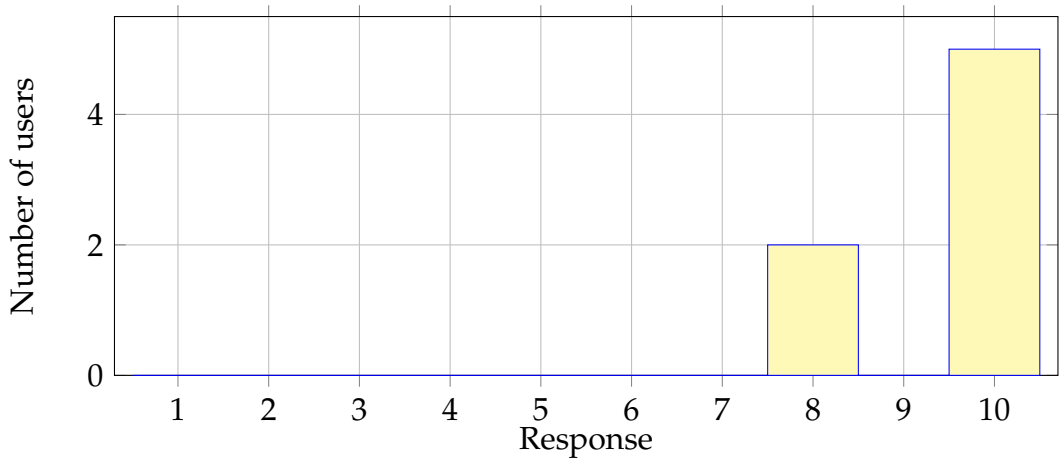

Figure 3.8: Sliders/Colour Pickers: The feature is useful

There are some improvement suggestions. Colour picker could be bigger, have a colour pallet and text boxes to type in RGB values. One evaluator asked me

if I could get wider range for sliders, but didn't propose specific solution. Bret had allowed dragging sliders outside of their range, they would expand as you drag them. I didn't do this and it can improve the range significantly, unless the slider is near the edge of the screen.

3.2.3 Code Line Highlighting

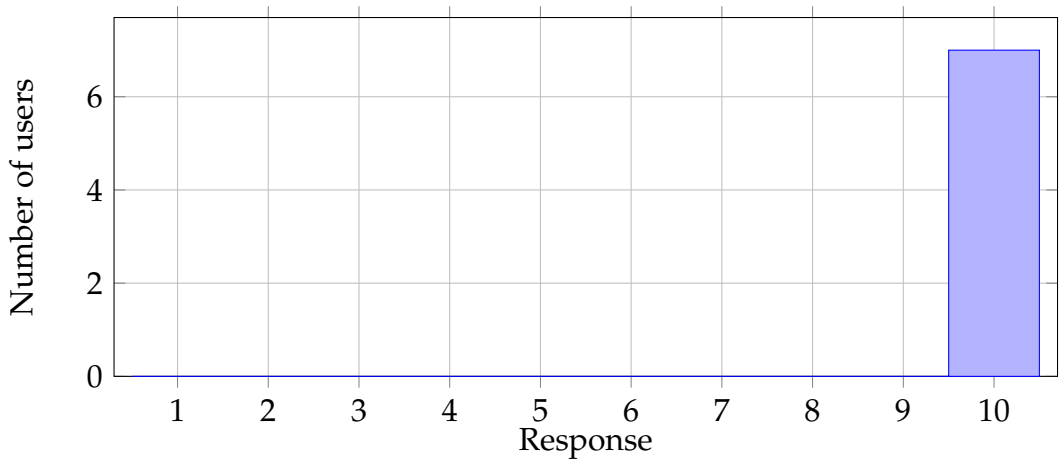

Figure 3.9: Code line highlighting: Feedback was immediate

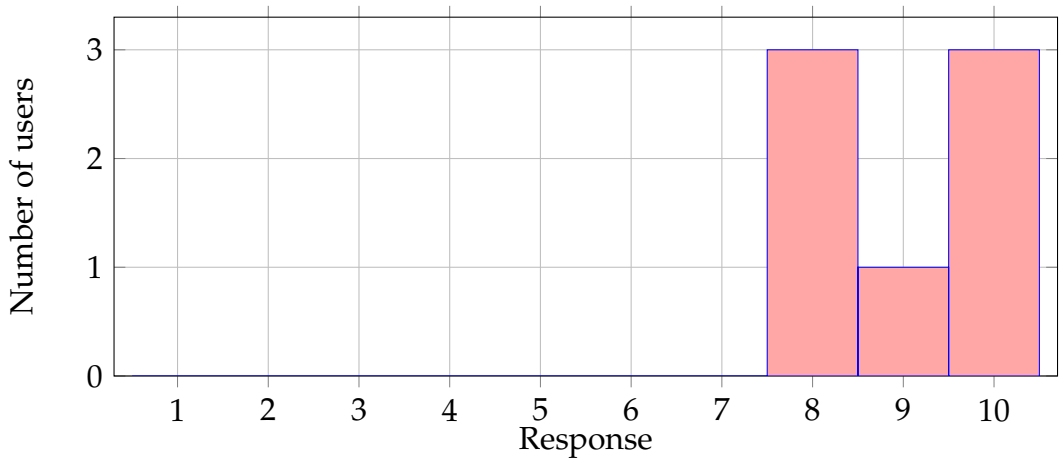

Figure 3.10: Code line highlighting: The feature is easy to use

> That's neat. I really liked the feature.
>
> That is really useful and works immediately.
>
> That is very impressive. I don't know how you do that.
>
> That was quick and super useful.

Apart from compliments I got one complaint about canvas locking the highlight when you click on an object. I made that when you hover over objects on

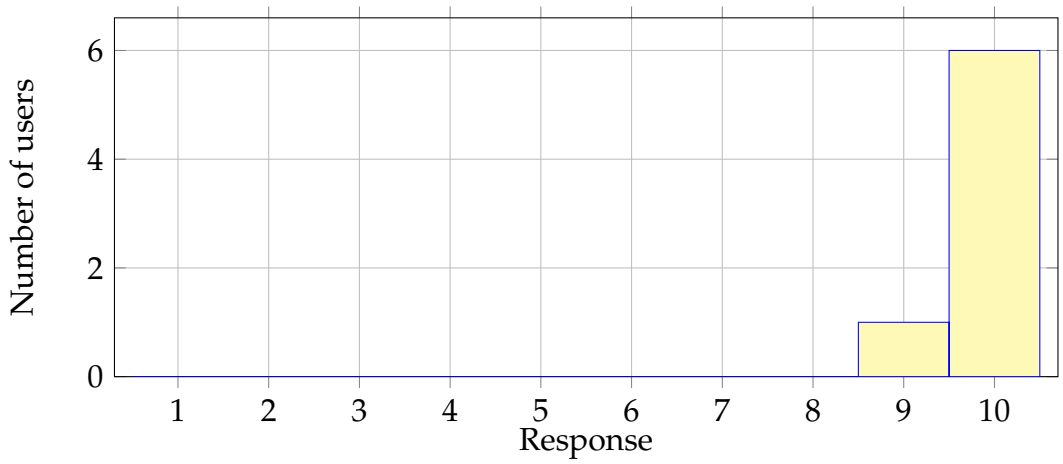

Figure 3.11: Code line highlighting: The feature is useful

the canvas they get highlighted, but when you click on one this feature stops working and the object you clicked on remains highlighted until you move the mouse outside of the canvas. This way you can safely move mouse back to code while highlight remains on the line you want to edit. The user that complained found it confusing that the feature stopped working after clicking and suggested that even when the highlight is locked he should be able to click on another object and get it highlighted without moving the mouse out of the canvas. He is right, this way it would be less confusing.

3.2.4 Time Manipulation

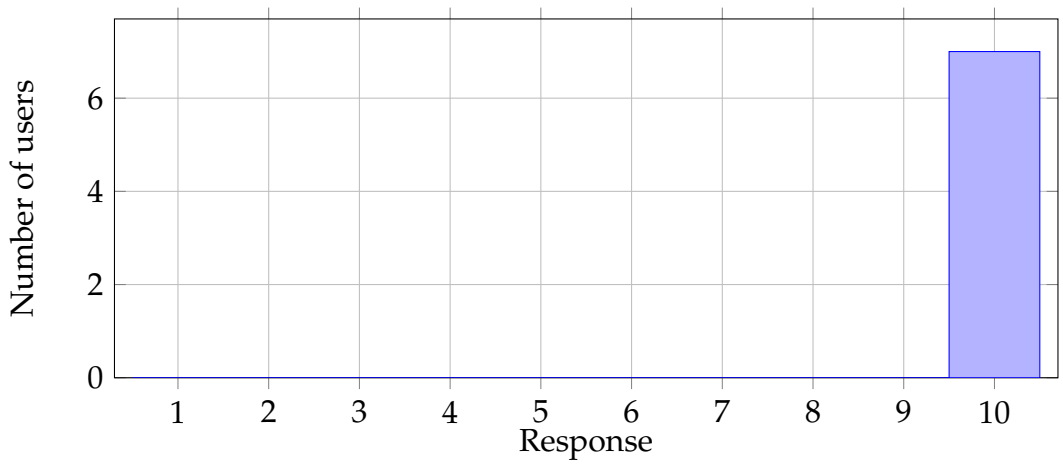

Figure 3.12: Time manipulation: Feedback was immediate

> That is cool. Really cool.

Users are satisfied with the feature. I was asked to make the system pause automatically when you rewind. Currently you can rewind using the slider and if you didn't press pause before rewinding game will continue playing from

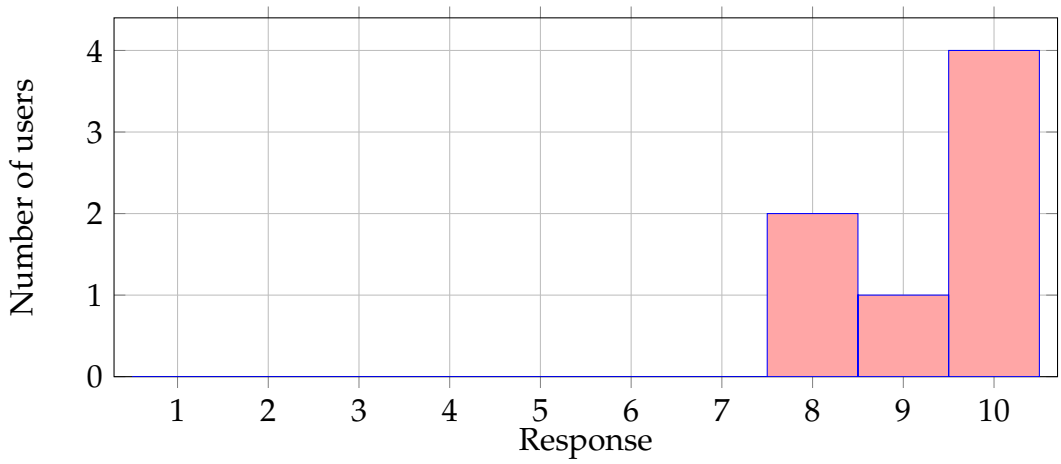

Figure 3.13: Time manipulation: The feature is easy to use

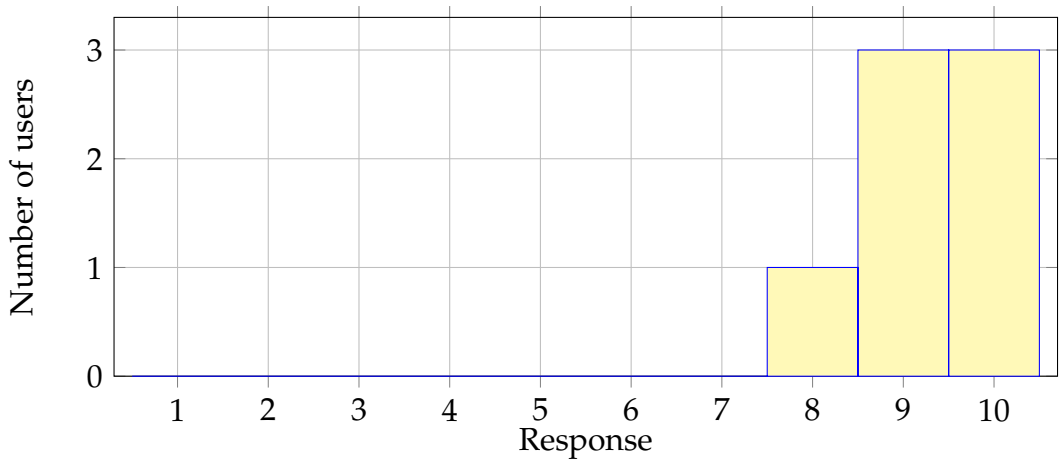

Figure 3.14: Time manipulation: The feature is useful

the point where you rewinded. This currently behaves like a movie player, but I agree with the user because if you rewind you usually want to pause and make a change there and then play.

3.2.5 Onion Skinning

Here are few positive comments first:

> That's good. You can see straight away what is happening.
>
> Wow! That is so cool!
>
> Onion skinning is really cool. I really liked it.
>
> It is easy to understand. I like the use case you showed when you can design the level after playing it.
>
> It is very impressive how fast it reacts.

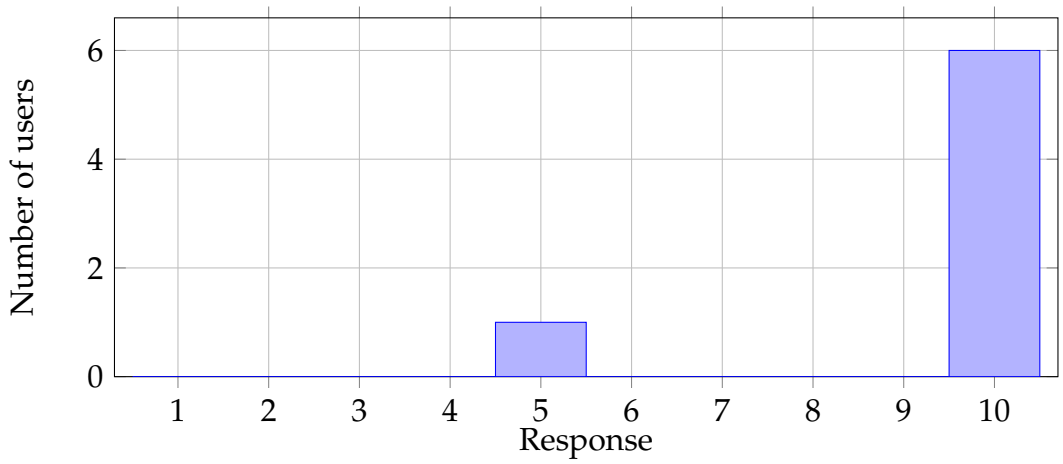

Figure 3.15: Onion Skinning: Feedback was immediate

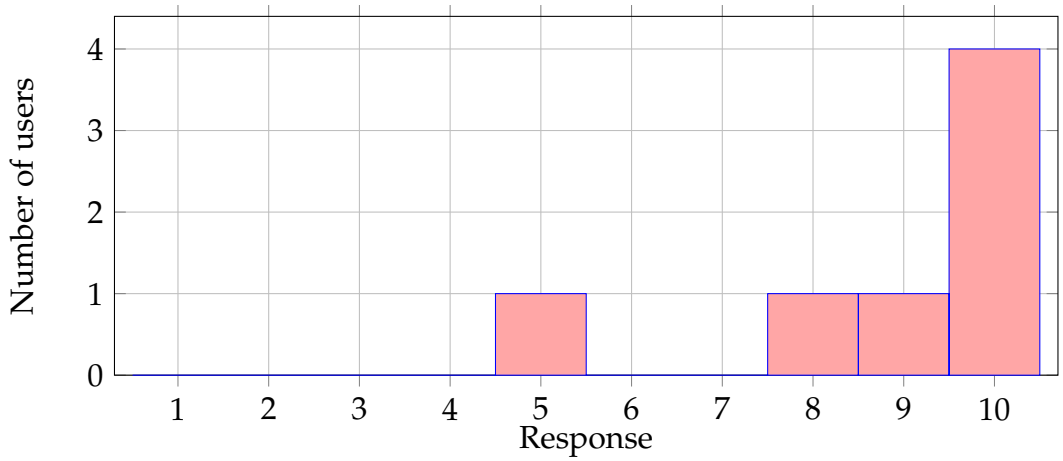

Figure 3.16: Onion Skinning: The feature is easy to use

This feature received much more critique the the other ones. Most evaluators were impressed by the feature, but some had trouble finding the use for it or understanding some parts of it.

> Why I can't see onion skinning while playing.
>
> Shouldn't it be the other way around.
>
> You need to rewind in order to see onion skinning. You should add a note about that.
>
> I personally wouldn't use this much.
>
> I don't know if it so useful. I can't think of use cases.

As you can see 2 evaluators didn't find it so useful. They still liked the feature and were satisfied with the responsiveness even when a lot of frames are drawn at once, but couldn't see themselves using it. One of them said that she is not working with animations often so she hasn't seen it used anywhere. Other one didn't see it as widely used tool in the future.

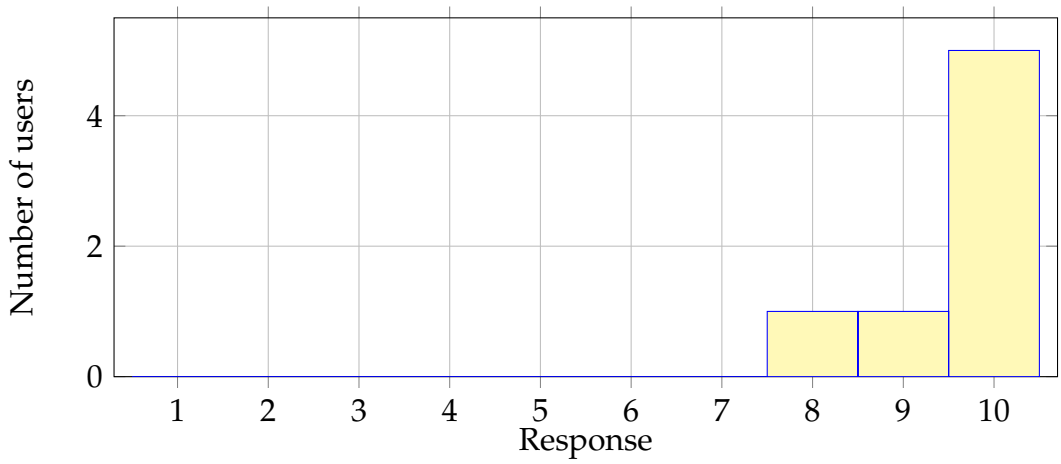

Figure 3.17: Onion Skinning: The feature is useful

An evaluator wanted to see onion skinning without rewinding and another one wanted it while playing. This is useful if you make animations, but if you want to see the future in a game that extensively uses keystrokes, you would just see the player falling or moving in a straight line which is not helpful. In this case you need to record your keystrokes, rewind and than watch what happens while changing the code. Anyways, it is confusing that nothing happens when you click on a checkbox next to onion skinning unless you rewind and I should add a warning about that.

Last complaint was that objects appear to move in opposite direction than they really are when onion skinning is on. This is because selected frame is drawn with full opacity and each following one with lower and lower opacity. People are used to motion blur where old frames are faded out and new ones are opaque. Why did I make it the other way around? For example, when you want to change jump height, you'll rewind the time just before the jump so you can change it. After that you will turn on onion skinning and you're most interested what is going to happen immediately after, not before, the point in time you rewinded to so those frames should have highest opacity. Bret dealt with this by drawing all of the onion frames with the same transparency, but then you can't see the direction of movement at all. The fade out looked more natural to me than object suddenly disappearing so I deviated from his demonstration.

Onion skinning showed great performance even in keystrokes demo where over 10 shapes were drawn for each of 30 visible frames at once, giving total of over 300 shapes per frame. One user gave it 5/10 for being immediate, but I didn't notice any lag nor he commented on his answer.

3.2.6 More features

Overall, evaluators were impressed by set of features the system has and some of them suggested additional features they would like to see:

- Drag and drop shapes to the canvas and get code generated - I like this one and it is already available in some GUI editors, but not as generic as other features as this would have to be specialised for the each function of the drawing API.
- Drag shapes around on the canvas and get code changed automatically - As I mentioned before, Craig Innes did this in his MInf Project. Again, this isn't generic and needs to be specialised for the drawing API as each drawing library has its own way of specifying positions and sizes of drawn objects, some of the libraries even use different functions to set the position of a shape and draw the shape which makes it hard to implement universal tool for this.
- Add API auto completion - I have autocompletion for the words used in the code, but I don't have it for Canvas API. I didn't focus on adding this as it is already widely available feature of IDEs (integrated development environment).
- Export project as HTML5 website - One of the evaluators wanted to export his game as HTML website. Good idea, but although project was implemented as a web site it wasn't focused on web specific features, but those that can be added to any IDE.
- Add separate red, green, blue channels - An evaluator suggested that it is easier to animate colours if you can assign each channel separately, apart from the default #RRGGBB string in the canvas.
- Shift the lines of code around - This is already available in some editors.

Chapter 4

Conclusion

This project showed that ideas from "Inventing on Principle" are already accomplishable and well accepted from potential users. It is not as difficult task as people think, it's more of an engineering problem than a research one. We already have libraries containing graphical elements like sliders and colour pickers and there are many customisable code editors and parsers that let you modify the syntax tree easily. We can dynamically modify scripts at run time and computers are already fast enough to reload part of the script on each keystroke. Also, OpenGL can be used for any type of parallel computing, not just graphics. Of course, we need to write a few custom algorithms, but most of the work is putting all of this together and getting good performance, every time you can get away without drawing a frame you need to do it.

I'm happy with how well users responded to the project. Evaluation sessions usually ended with "That was really impressive.", "That is so cool!" or "Wow!" and my reaction was the same the first time I watched "Inventing on Principle" [16]. Speak-aloud evaluation showed me how much design preferences differ between users. Because of many different feature requests, a good way to proceed with the project would be to set up online repository where users can evaluate, request or vote for the features. This way I would let users choose development path.

During evaluation system showed to be reliable, apart from the for loop bug. There was no lag whatsoever, output was immediately changed with the code and none of the users complained about that.

Chapter 5

Appendix

5.1 Feedback script

My Feedback process

1) Immediate interpretation and sliders/colour pickers (still image), partial interpretation is disabled Starting with an empty file with two comments on the top

```
// HTML5 canvas can be accessed using variable named c
// Its context can be accessed using variable named ctx
```

Add a rectangle using:

```
ctx.fillRect(20,20,150,100);
```

Four parameters represent x, y, width and height. Click on a number and the slider will appear. Move the slider and see what happens. Click on the number again to reset range.

Change the colour of the the rectangle by adding:

```
ctx.fillStyle = "#1f78ff";
```

before the fillRect function. Change it with a colour picker. Add another rectangle. Hover over one of the rectangles and line of code that drew it will get highlighted. Click on the line that drew other rectangle and the rectangle will get highlighted.

Anything you want to try before we move on to animation

2) Animation, partial interpretation and the onion skinning Start with a ping pong demo. Pause the time and rewind. Turn on onion skinning. Change the gravity

Load keystrokes demo. Move the ball to lower right corner using WASD keys. When you hit yellow block rewind and use onion skinning to redesign the level so it isn't so hard to complete.

5.2 Ping Pong Demo

```
var color = "#ff7308";
function drawCircle(){
    ctx.beginPath();
    ctx.arc(ball.x, ball.y, 24, 0, 2 * Math.PI, false);
    ctx.fillStyle = "green";
    ctx.fill();
    ctx.lineWidth = 5;
    ctx.strokeStyle = "#003302";
    ctx.stroke();
}

var ball = {x: 49, y: 32, vx: 272, vy: 0};

function updateBall(){
    ball.vy += Time.deltaTime * 487.55006;
    ball.y += ball.vy * Time.deltaTime;

    ball.x += ball.vx * Time.deltaTime;

    if((ball.x < 0 && ball.vx < 0) ||(ball.x > 600 && ball.vx > 0)){
        ball.vx = -ball.vx;
    }
    if((ball.y < 0 && ball.vy < 0) ||(ball.y > 500 && ball.vy > 0)){
        ball.vy = -ball.vy;
    }


}

var speed = 500;

function update() {

    // clear
    ctx.onion = true;
    ctx.clearRect(0, 0, c.width, c.height);

    drawCircle();
    updateBall();
}
```

5.3 Keystrokes Demo

```
var color = "#bdff38";


function drawCircle(){
    if(winState == 'lost') {
        ctx.fillStyle = "red";
        ctx.font = "30px Arial";
        ctx.fillText("GAME OVER",10,50);
```

```
        return;
    }
    ctx.beginPath();
    ctx.arc(ball.x, ball.y, 24, 0, 2 * Math.PI, false);
    ctx.fillStyle = "blue";
    ctx.fill();
    ctx.lineWidth = 5;
    ctx.strokeStyle = "#003302";
    ctx.stroke();
}

var ball = {x: 49, y: 32, vx: 272, vy: 0};

function goTo(pos, vel){
    //if(pos[0])


}

function isOnGrid(posx, posy){
    var yDiv = c.height/grid.length;


    var y = Math.floor(posy/yDiv);
    if(!grid[y]){
        return false;
    }
    var xDiv = c.width/grid[y].length;


    var x=Math.floor(posx/xDiv);



    return grid[y][x];
}

var winState = "playing";
function updateBall(){

    var wasOnGrid = isOnGrid(ball.x, ball.y);

    ball.vy += Time.deltaTime * 0.0;
    ball.y += ball.vy * Time.deltaTime;

    ball.x += ball.vx * Time.deltaTime;

    var onGrid = isOnGrid(ball.x, ball.y);

    if(onGrid){
        winState = "lost";

    }
```

```
    if((ball.x < 0 && ball.vx < 0) ||(ball.x > 600 && ball.vx > 0)){
        ball.vx = -ball.vx * 0.9;
    }
    if((ball.y < 0 && ball.vy < 0) ||(ball.y > 500 && ball.vy > 0)){
        ball.vy = -ball.vy * 0.9;
    }


}

var grid = [
            [false, false,  false,  false,  false,  false],
            [false, false,  false,  true,  false,  false],
            [false, false,  false,  false,  false,  false],
            [false, false,  false,  false,  false,  false],
            [false, false,  false,  false,  true,  true],
            [false, false,  true,  false,  false,  false],
            [false, false,  false,  false,  false,  false],
            [true, true,  false,  false,  true,  false],
            ];

//draw the grid
function drawGrid(){
    ctx.fillStyle = "orange";
    var yDiv = c.height/grid.length;
    var x = 0;
    var y=0;
    for(var h of grid){

        var xDiv = c.width/h.length;
        x = 0;
        for(var v of h){
            if(v) ctx.fillRect(x*xDiv,y*yDiv,xDiv,yDiv);

            x++;

        }
        y++;
    }
}
var accel = 781;
function update() {
    // clear
    ctx.clearRect(0, 0, c.width, c.height);

    if(state == "right"){
        ball.vx += Time.deltaTime * accel;
    }

    if(state == "left"){
        ball.vx -= Time.deltaTime * accel;
    }

    if(state == "down"){
        ball.vy += Time.deltaTime * accel;
    }
```

```
    if(state == "up"){
        ball.vy -= Time.deltaTime * accel;
    }

    drawCircle();
    updateBall();
    drawGrid();
}

var state = "idle";

function onkeydown(event){
    if(event.which == "D".charCodeAt(0)){
        state = "right";
    }
    if(event.which == "A".charCodeAt(0)){
        state = "left";
    }
    if(event.which == "W".charCodeAt(0)){
        state = "up";
    }
    if(event.which == "S".charCodeAt(0)){
        state = "down";
    }
}

function onkeyup(event){
    state = "idle";
}
```

Bibliography

[1] Ace Editor. https://ace.c9.io. Accessed: 28 Sep 2015.

[2] escodegen. https://github.com/estools/escodegen. Accessed: 6 Nov 2015.

[3] Ariya Hidayat. Esprima. http://esprima.org/. Accessed: 6 Nov 2015.

[4] Craig Innes. Programming as an interactive experience, minf project (part 2) report. http://homepages.inf.ed.ac.uk/dts/students/innes/report.pdf, 2014. Accessed: 29 Sep 2015.

[5] Craig Innes. Time manipulation. In *Programming as an Interactive Experience, MInf Project (Part 2) Report*, page 50, 2014.

[6] jQuery. extend(). https://api.jquery.com/jquery.extend/. Accessed: 26 Oct 2015.

[7] jQuery. globaleval(). http://api.jquery.com/jquery.globaleval/. Accessed: 26 Oct 2015.

[8] jQuery UI. https://jqueryui.com/. Accessed: 20 Oct 2015.

[9] jscolor Color Picker. http://jscolor.com/. Accessed: 20 Oct 2015.

[10] Kodowa. Light table. http://lighttable.com/. Accessed: 30 Sep 2015.

[11] Mozilla Developer Network. Canvas api. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. Accessed: 30 Sep 2015.

[12] Alexander Oldemeier. Playing around with fragment shaders in webgl. https://blog.mayflower.de/4584-Playing-around-with-pixel-shaders-in-WebGL.html. Accessed: 1 Oct 2015.

[13] stackframe. https://github.com/stacktracejs/stackframe. Accessed: 29 Sep 2015.

[14] Bug 729385 Canvas to WebGL texture copies are slow. https://bugzilla.mozilla.org/show_bug.cgi?id=729385. Accessed: 21 Dec 2015.

[15] Tributary. http://tributary.io/. Accessed: 29 Sep 2015.

[16] Bret Victor. Inventing on principle. http://worrydream.com/#!/InventingOnPrinciple. Accessed: 26 Sep 2015.