Game Programming by Demonstration

Mikaël Mayer
EPFL, Switzerland
mikael.mayer@epfl.ch

Viktor Kuncak [*]
EPFL, Switzerland
viktor.kuncak@epfl.ch

Abstract

The increasing adoption of smartphones and tablets has provided tens of millions of users with substantial resources for computation, communication and sensing. The availability of these resources has a huge potential to positively transform our society and empower individuals. Unfortunately, although the number of users has increased dramatically, the number of developers is still limited by the high barrier that existing programming environments impose.

To understand possible directions for helping end users to program, we present Pong Designer, an environment for developing 2D physics games through direct manipulation of object behaviors. Pong Designer is built using Scala and runs on Android tablets with the multi-touch screen as the main input. We show that Pong Designer can create simple games in a few steps. This includes (multi-player and multi-screen) Pong, Brick Breaker, Pacman, Tilting maze. We have made an open release of Pong Designer and editable games that we created using it. This paper describes the main principles behind pong designer, and illustrate the process of developing and customizing behavior in this approach.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Interactive environments

Keywords programming by demonstration

1. Introduction

Smartphone and tablet devices have dramatically increased the number of individuals with access to computing resources. The availability of these resources has an enormous potential to positively transform our society. Unfortunately, using traditional development methods for such devices is at least as difficult than for desktops, and possibly more difficult due to new constraints on device input size, energy consumption, and the complexity of the software stack. Furthermore, the benefits of these platforms can be fully realized only by specialization of applications to particular domains, or even particular users. We would like to enable domain experts that are not software developers by training to develop applications that support well their domain activities. Many educational, scientific, engineering and artistic domains would benefit from such end-user programming. For tasks such as scripting and home automation, we would like to deliver personalized applications at the low price that makes current phone applications accessible. To achieve this level of specialization, the number of developers needs to be much closer to the number of users. A promising approach to realize these goals is to empower users themselves to program, blurring the traditional divide between professional software developers and end users. This direction is especially appealing as the increasing computing capabilities of these ubiquitous devices enable more advanced runtimes and software development tools, and as new algorithmic techniques enable automated programming assistance and synthesis [7, 8, 12, 19, 21].

One of the challenges when programming using conventional text-oriented editors is the disconnect between program representation and its effect during execution. Several recent approaches support understanding the effect of a line of code with an enhanced editor [15] or with very high-level constructs such as behaviors and constraints [18], [22]. Others prefer to guide the programmer by providing code structures that can fit together, either modifiable during the game simulation [16] or in a special structured editor [18], [1]. These approaches reduce the burden of syntax checking. However, few of them provide a way to directly modify the running application by demonstrating the desired behavior using examples.

Recently, programming by demonstration has been revisited and shown very successful in domains such as spreadsheets [7, 21], which map inputs to outputs. We wish to understand the potential of programming by example on a new generation of touch-enabled devices and apply it to more complex domains, containing interactive behavior. This led us to the domain of graphical games running on tablets.

[*] This research is supported in part by the European Research Council (ERC) Project *Implicit Programming*

SPLASH Onward'13, October 26–31, 2013, Indianapolis, USA
Copyright © 2013 ACM . . . $15.00

1.1 Contributions

This paper presents a development approach for graphical games where developers use demonstration to describe not only application state, but also its behavior. The developer can pause the game and directly manipulate objects to demonstrate the desired effects through examples. The main contributions of our approach are the following:

- An on-the-fly editing principle: the users can pause, rewind, and modify a running game using a time progress bar, with graphical access to events from the past.
- Rule demonstration: a rule-based execution model, where developers dynamically create and update rules using concrete demonstrations on objects. The system automatically infers candidate rule conditions and actions from such demonstrations.
- A freely available working Scala/Android implementation that leverages these principles on runs on devices enabled with multitouch and accelerometer input. The system and several examples of (editable) games are freely available as "Pong Designer" app from Play Store, as well as at `http://lara.epfl.ch/w/pong/` .

2. Building Games by Demonstration

We illustrate the flavor of game development in Pong Designer by showing how to develop several games using a remarkably small number of steps on a tablet computer (we used Asus Eee Pad Transformer for our experiments).

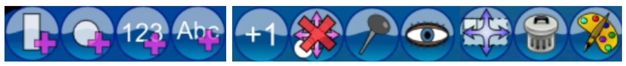
We first show how to build a brick breaker game, followed by a variant of Pacman with a moving camera and accelerometer input.

2.1 Breakout-Style Game

This example illustrates the use of time slider and demonstration of behavior in response of actions.

Suppose that we wish to program a classic breakout-style game, where the goal is to drive the bouncing ball, using a sliding paddle to ensure it does not escape the screen at the bottom, and aiming to ensure that the ball breaks all the bricks on the screen.

Describing the initial state. We begin by creating the game objects, using predefined shapes. This process resembles drawing in a simple vector graphic design application, such as the ones used to create conference presentation slides. To introduce shapes into the playing field or modify their properties we use buttons from an on-screen toolbar:



The buttons provide a way to increment or decrement a number, set up the velocity and angle value of any shape, whether the shape can move or not as well as its visibility, size and color. Setting up the game layout corresponds to defining data structures and objects in usual programming.

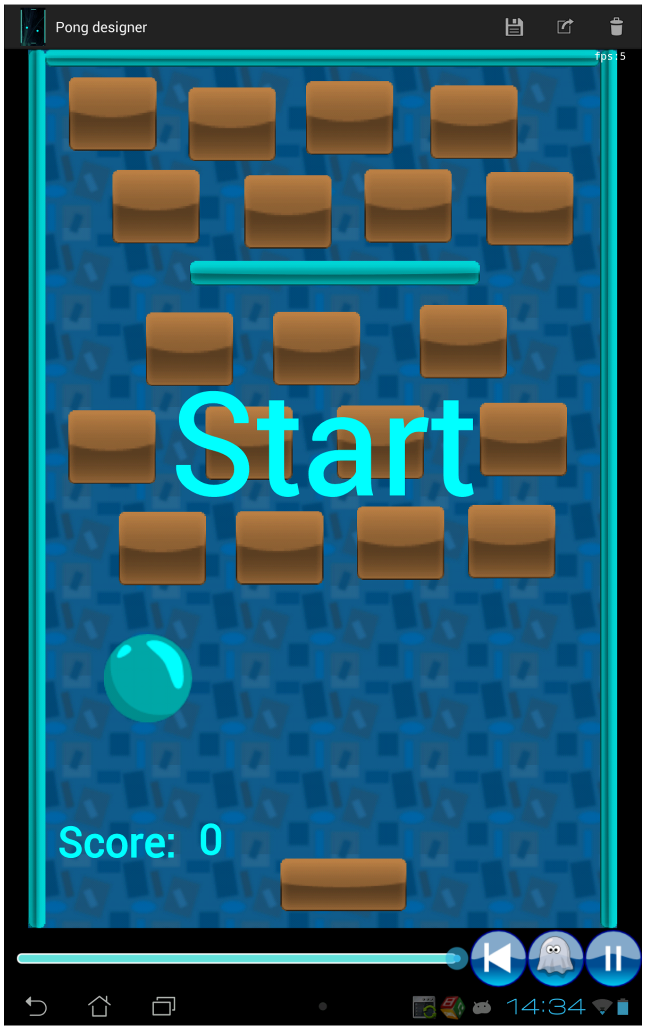
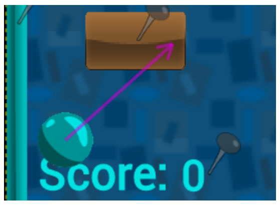


Figure 1. Static Initial State of a Breakout-Style Game

Figure 1 shows a possible result for our example, which includes the walls, bricks, the ball, and the paddle.

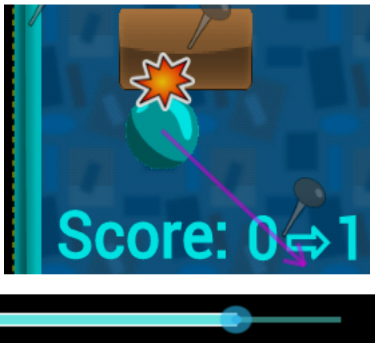
Setting dynamic properties such as velocity. In contrast to a drawing program, the objects we created are in fact models of physical objects with associated behavior in a two-dimensional physics world. Objects can be either non-movable (pinned to the ground), or movable according to Newton's inertial laws with friction. By default, all objects have zero initial velocity. If we select the ball object, we can change its speed by moving the endpoint of the velocity vector displayed on screen. In this example, we set the initial velocity of the ball to move towards a nearby brick:



Starting the game. Pressing the "play" button starts the game, running the physics simulation from the current state and displaying the outcome in real time on the screen (in this case, the ball moves towards the brick). As the game runs, we observe that the time bar at the bottom of the screen makes progress, indicating the passage of time, much like when playing a music or video stream.

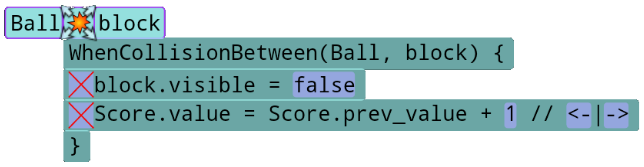
During this time the system silently records internal events, such as object collisions, as well as user input, in particular the multi-touch screen input.

Identifying events of interest. In our example we observe the ball hitting the brick and bouncing off it, with the brick staying intact. We would like to change this behavior to ensure that that, when the ball hits a brick, the score increments and the brick disappears. We press the "Pause" button to stop the simulation, which allows us to again edit objects in the last simulated state. This time, however, the game also contains the history of past events. Pressing the events button displays the events using appropriate graphical metaphors. The following representation shows that there was a recent collision between the brick and the ball:



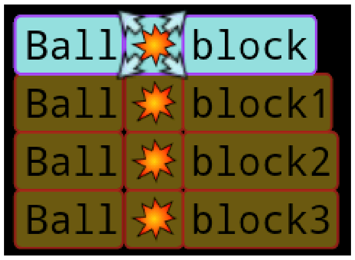
Navigation in time and space to select events. Note that the number of events recorded can be large, but the user can navigate them using the fact that an event is indicated near the relevant objects, and at the relevant time. The user uses the time bar to go back in past to the approximate point in time when the event of interest occurred. In our example, the user chooses the collision event between the ball and the brick.

Describing actions: breaking a brick. The selected event represents a condition for triggering a rule. The entire game is governed by such event-triggered rules. To specify the action that should take place in case of a given event, the user performs the action in the same editor. This action subsumes macro recording in simpler systems, but is followed by a crucial *generalization* step. In our current example, we would like to indicate that the brick should disappear in case of a collision with the ball. To do this, after selecting the collision event, we simply move the brick outside the visible screen, or set its visibility to *invisible*. We also increment the score counter. After pressing the OK button, the system automatically generates the corresponding rule (see below). The user can edit and delete some of its parts, if desired, or simply accept it as it is. The rule will now be applied whenever the ball hits this brick.



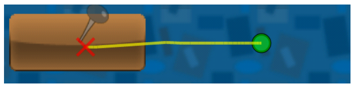
```
Ball💥block
    WhenCollisionBetween(Ball, block) {
    ✗block.visible = false
    ✗Score.value = Score.prev_value + 1 // <-|->
    }
```

If we copy any object, the system copies the rule along with the objects to which they apply. In this case we first create one brick and the rule. When duplicating the brick, the system will duplicate the rule. (This corresponds to a prototype-based object system; we are also adding support for sets and classes of objects.)

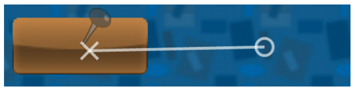


```
Ball💥block
Ball💥block1
Ball💥block2
Ball💥block3
```

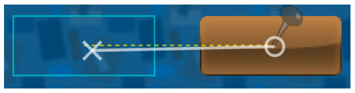
Second rule: moving the paddle. We next wish to specify that the paddle follows the horizontal movements on the touch screen when they occur on the paddle. To do this, we run the game and attempt to move the paddle. The paddle does not respond, but the movement is recorded in the event history. We can use the recorded movement to further correct the existing behavior. We pause the game and move the time slider towards the point where we made the movement on the screen. The movements are displayed using curves that describe the path traced on the screen, as in the following movement to the right:



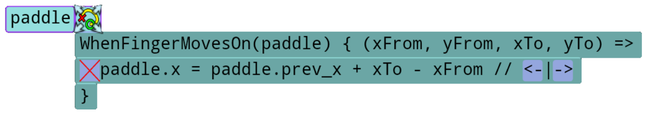
We select the move event, which acts as the condition of our rule. As the action for the rule, we demonstrate the corresponding change in the position of the object, moving it from the initial position



into the final position:



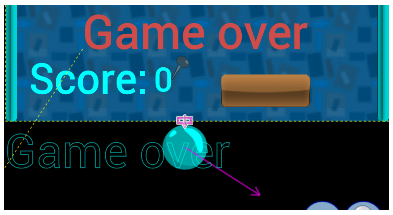
Because of a "snapping-to-position" feature, this movement is recorded as a perfectly horizontal movement. At this point the system performs a generalization from the concrete demonstration to a rule applying to these two objects. Even if the two movements demonstrated are not perfectly identical, our system finds that they are sufficiently close and derives the following rule that matches the local in finger position to the indicated horizontal movement on the screen.



```
paddle
    WhenFingerMovesOn(paddle) { (xFrom, yFrom, xTo, yTo) =>
    paddle.x = paddle.prev_x + xTo - xFrom // <-|->
    }
```

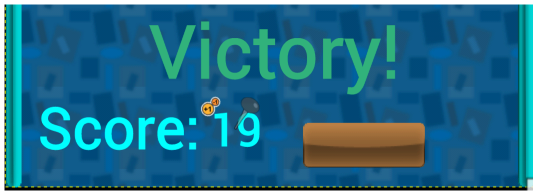
In general, the system uses a set of expected parametrized templates to compute a set of possible actions that could explain the demonstrated state change. If the user expands the generated rule, they are able to delete lines and select the intended result from the templates by pressing arrows.

Describing the losing condition. We would like to specify that a text displaying "Game over" appears when the ball is out of screen. For that, we first create the label and make it invisible by default. To specify when this text should become visible again, we follow these steps. We set the velocity of the ball towards the bottom of the game and launch the simulation. When the ball goes out of the screen, we pause the game. The engine detected the event of the ball going out of screen.



We select the out-of-screen event, make the "Game over" text box visible, and confirm this behavior by pressing OK. The system then creates the corresponding rule automatically.

Describing the winning condition. Finally, we would like to specify that a label "Victory" should appear when the score reaches 19, which is the number of blocks in the game. For this purpose, we make a text displaying "Victory" in the middle, initially invisible. We then change the score to 18 to indicate an interesting starting scenario for simulation. We then play the game until the score reaches 19. When we press the event menu, the system detected the previous change as a *number change event*. Select it, and accept the offered condition "When **score** == 19".



Make the "Victory" text visible and confirm the rule.

This completes the description of the basic functionality of a break out style game.

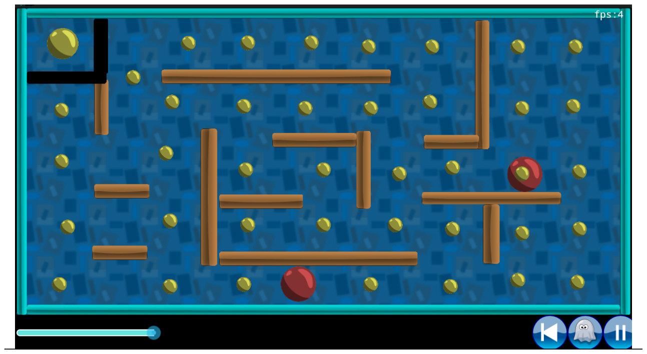


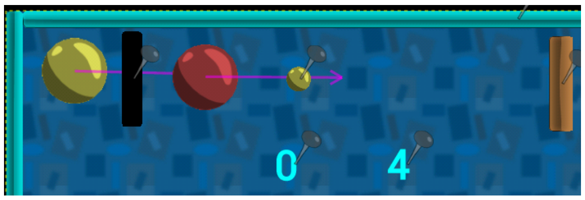
Figure 2. Screenshot of a Pacman-like game build in Pong Designer

2.2 Pacman with Accelerometer Input

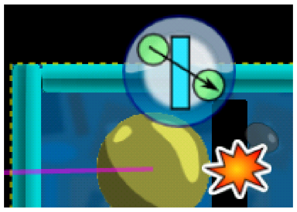
We next show how to build a Pacman-like game controlled using the accelerometer (the angle of the device). Only one part of the game field is visible on the screen, so the camera (viewpoint) needs to follow the player. The player moves in the labyrinth, and needs to eat all food chunks without touching any of the enemies. We will introduce three enemies; if they touch the player, the player lose a life. After losing four lives, the game is lost. The player can win by eating all food chunks.

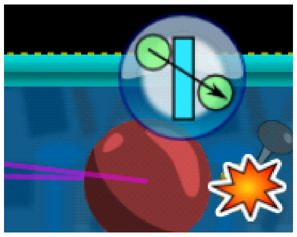
We will create the game logic from scratch, specifying that the player should go through black walls, eat the chunk, and when the enemy should go through the chunk and make the player loose a life.

1. Create a yellow circle for a player, a red circle for an enemy, a yellow circle for a food chunk, a brown rectangle for a wall and a black rectangle for "home" (a wall through which the player can go, but not the enemies). Add two numbers, one to 0 to count the number of chunks eaten, and the other to 4 to count the number of lives. Set the speed of the player and the enemy to the right.

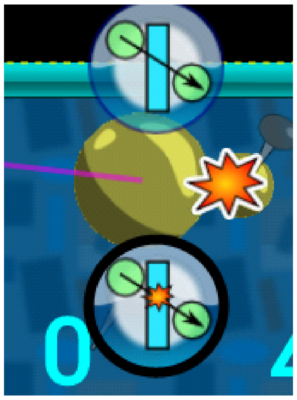


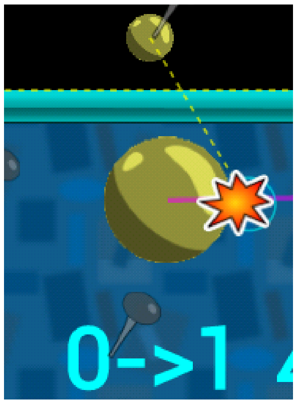
2. Start the game. After a few seconds, pause it again.
3. The player bounced against the black wall. Press the event menu, select the collision, and press on the button to remove the collision. Now the player goes through the black wall. Follow similar steps to remove the collision between the enemy and the food chunk.



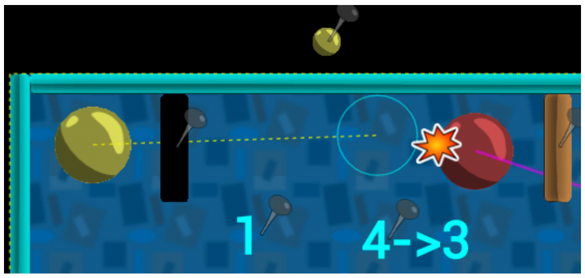


4. The player collides with the food. Select an instance of this collision, choose an action that the objects should go through each other but the collision is recorded, augment the score from 0 to 1, and move the food away from the visible part of the screen.





5. Move the time forward to observe the enemy bouncing against the wall and colliding with the player. Select the collision, select the option that the object should go through each other but the collision is recorded (as for the food), move the player back to the base, set its speed to zero, and decrease the number of lives.

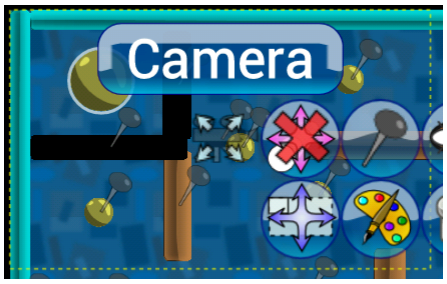


6. Now go back to the beginning. Create a maze, duplicating the walls, food and enemies as needed.

Second, we would like to let the player follow the gravity, and the camera to follow the player.

1. Press the accelerometer button to be able to select the shapes affected by this sensor, and select the player. Press the accelerometer button to stop selecting objects.

2. Press the camera button and then select the player. As a result, the camera will follow the player. To adjust the view screen, resize the camera.



3. To add "Victory" and "Game over" labels, follow the steps mentioned in the previous game.

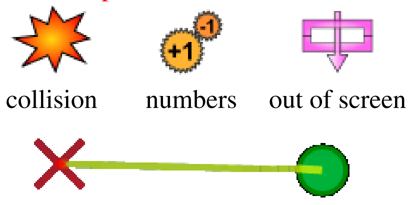
In addition to its immediate use when following the player in a large game field, the camera can be used to switch between several screens in case of a multi-level game, or a game with menus and options.

3. Pong Designer Principles

The key novelty of Pong Designer is the rule-based model with the ability to dynamically change rules through concrete demonstrations in a desired context. Starting from one or more concrete demonstration (which can be introduced incrementally at different points in time), the system performs a generalization to obtain a rule that applies beyond the concrete state in which it was demonstrated. Rules contain a condition (an event) and the action (state change).

3.1 Events

Pong Designer currently supports six kinds of events. Each kind of event may include several variations. For example, the user can specialize collision rules as "objects go through each other and no rule is executed", "objects go through each other and the rule is executed" or "objects bounce against each other and the rule is executed". It is not at all clear what it means to keep a collision



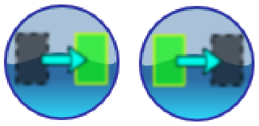
collision numbers out of screen

touch down move touch up

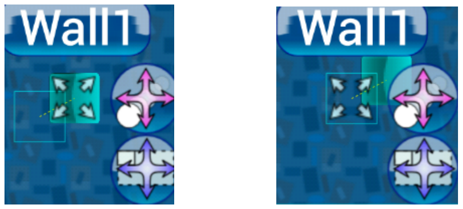
3.2 Specifying Actions

Selecting an event enables the user to specify an action though an example of input and output, which the system generalizes.

During the modification of the game state, the system draws two versions of objects being changed. The first version shows the original state of objects (input); the second state is the transformed state of the objects (output). It is possible to modify either the input or the output. To switch between these two, the developer toggles the input/output button.



When the user changes the output, moving a shape moves it as usual normally (the first picture below). This is the default mode. When the user changes the input state, moving a shape will move a shadow version of it.

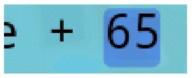


3.3 Editing Created Rules

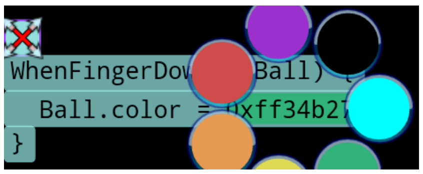
Once created through demonstration, it is possible to directly adjust the rules. This functionality is provided as a fallback in case demonstration does not achieve the desired effect or the users prefer to examine a more textual version of the rules. Note that, even in this representation we use graphical notation for events. Moreover, it is possible to edit constants in rules using selection or increment and decrement actions that requires no keyboard input. Finally, a preview feature makes it possible to quickly preview the effect of a single rule invocation on the current state.

Press OK button to create a new rule based on the modifications of the game, or to refine the existing open rule by providing a new demonstration.

After a rule is created, one can view its content for manual review and editing.

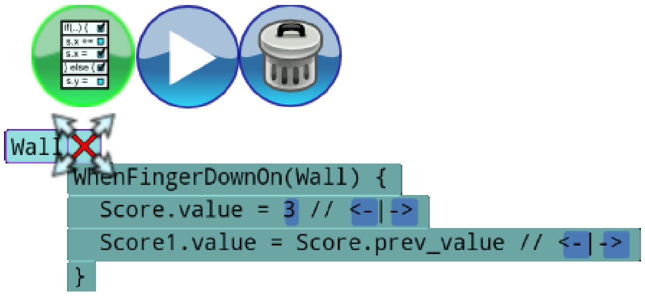


Drag the finger on a constant in dark blue to modify its value. Depending on the type of the number, an appropriate input method is invoked. For example, changing a color constant opens a color palette.



```
WhenFingerDow      Ball) {
  Ball.color = 0xff34b27
}
```

Press the arrows to choose different code possibilities that were generated by the system. For example, we can switch between all the three codes transforming 1 to 3, which are $= \ldots \times 3$, $= \ldots + 2$ and $= 3$.



```
Wall1
  WhenFingerDownOn(Wall) {
    Score.value = 3 // <-|->
    Score1.value = Score.prev_value // <-|->
  }
```

To apply the code of the rule to the game, press the play button. This can be used for example to modify parameters and to see how the rule behaves for them, in order to correct them. If the rule is opened, the game shows what the results of the rule would be.

3.4 Underlying Domain-Specific Language

Each game in Pong Designer can be described by its initial state and the set of rules. Although rules can be modified and displayed graphically, they also have a textual representation as a domain-specific language embedded into Scala. We use this domain-specific language as reference semantics, but also as a way to emit a compiled version of the game. Figure 3 summarizes this domain-specific language.

The game engine considers rules similar to logic gates. This means that the order in which the game executes their inner lines of code is not important. The game modifies all the parameters at the same time, through the use of a stacking mechanism illustrated in Figure 3. In all the assignments, the parameters names to the right of the equality start with "prev", which means that their value is the one before the rule started. In the sequel, whenever $v \mathrel{+}= C$ is written, it means in reality $v = prev\_v + C$ so that the previous value of v is left unchanged for other lines of code.

When the user provides an input/output example of how the game state should change, the code generator compares the example against the list of actions in Figure 4. If several actions are suitable for the same change, the code generator wraps all of them in a parallel instruction PARALLEL_EXPR, in the same order as in Figure 4. The meaning of such an expression is "Execute the first action, but keep the others in the case the first action is wrong." For example, if the code has to change a score value v from 1 to 2 when a certain event occurs, the system will create the line PARALLEL_EXPR$(v = 2, v \mathrel{+}= 1, v \mathrel{*}= 2)$. Now, if the same event occurs and the user specifies that the score should increase up to 3, the system will drop the first $v = 2$ and the last $v \mathrel{*}= 2$ and will keep only the second $v \mathrel{+}= 1$.

Furthermore, if the user enters a new input/output example which is contradictory regarding an existing piece of code, the code generator selects the most recent one. For example, if the code for a rule is PARALLEL_EXPR$(v = 2, v \mathrel{+}= 1)$, and the player asserts that the value should increase from 2 to 4, this is not consistent. Therefore, the code generator

```
GAME := class NAME extends Game
        '{' GAME_CONTENT '}'

GAME_CONTENT := layout_width = constant
                layout_height  = constant
                {SHAPE_DEF}+ {CATEGORY_DEF}+
                {RULE_DEF}+

SHAPE_DEF :=
  val NAME = new (Rectangle | Circle | IntegerBox | TextBox)
  '{' {property = value}+ '}'

CATEGORY_DEF := Accelerometer({NAME}+)
              | Gravity2D({NAME}+)

RULE_DEF :=
    WhenFingerDownOn(NAME) '{' CODE '}'
  | WhenFingerUpOn(NAME) '{' CODE '}'
  | WhenFingerMovesFrom(NAME) '{'
      (xFrom: Float, yFrom: Float,
       xTo: Float, yTo: Float) =>
      CODE
    '}'
  | WhenNumberChanges(NAME) '{'
      (newValue: Int) =>
      CODE
    '}'
  | WhenEver(BOOL_EXPR) '{' CODE '}'
  | WhenCollisionBetween(NAME, NAME) '{'
      CODE
    '}'
  | NoCollisionBetween(NAME, NAME)
  | NoCollisionEffectBetween(NAME, NAME)

CODE := SIMPLE_CODE
      | if(BOOL_EXPR, SIMPLE_CODE, CODE)
SIMPLE_CODE := {PARALLEL_EXPR}+
PARALLEL_EXPR := Parallel(MODIF_LINE+)
MODIF_LINE := NAME.property = FORMULA
FORMULA := FORMULA (+|−|*|%|/) FORMULA
FORMULA := NAME.prev_property
FORMULA := constant
FORMULA := newValue|xFrom|yFrom|xTo|yTo
BOOL_EXPR := FORMULA (⩽|⩾|<|>|==) FORMULA
BOOL_EXPR := BOOL_EXPR (|| | &&) BOOL_EXPR
BOOL_EXPR := !BOOL_EXPR
```

Figure 3. An overiew of the language and grammar used by our system to generate code

will overwrite the previous rule by producing the following code PARALLEL_EXPR($v \mathrel{*}= 2, v \mathrel{+}= 2, v = 4$).

If the event occurs when there is a finger move on a shape, then the variables describing the move can be used in the code. The system accepts relative coordinates imprecisions up to 40%. For example, if the user moved his finger from $xFrom = 90$ to $xTo = 140$, and he also moved the shape x from 80 to 128, the system outputs the code PARALLEL_EXPR($x \mathrel{+}= xTo - xFrom, x \mathrel{+}= 50, x = 128$) because $128-80$ is approximately equal to $140-90$. Enriching such expressions by considering an arithmetic interval solver to accomodate imprecisions is a problem we might investigate in the future.

Although each atomic modification MODIF_LINE could be, in principle, arbitrarily generated by the grammar, the generator may only use patterns from Figure 4, especially for conditional if-then-else statements. Whenever other shapes are involved in those patterns, it means that the generator loops over all the shapes having the desired property, such as a “width”. For each shape so that the pattern works, it outputs a code snippet.

These patterns ensure coherent code and might be extended in the future as we add other behaviors. For example, if we wish in the future to produce random numbers, we would add the random function as a primitive pattern.

3.5 Rule Creation Algorithm

Whenever the user performs a new demonstration, Pong Designer uses it to adjust the existing set of rules.

Creating and updating rules are similar activities. If the rule does not exist yet, the system creates it according to the type of the selected event, and its action is initially empty (see Figure 8)

The system extracts the code from the game state. It uses templates to generate the code (see Figure 6). Templates have access to the game state, so they can, for example, provide a code to align shapes, or set up a number as a combination of two other numbers. The template system in Figure 5 gives an idea of our template matching process.

When the system recovers the code from the game, it merges them with the existing code from the rule (see Figure 7). If there are number conditions for the new code, the system generates corresponding if-then-else statements or refines the existing ones. Generated if-then-else statements currently only check whether the number is less than constant, so the code is easily maintainable.

When the user duplicates a shape, if the condition of a rule contains the shape, the system duplicates the entire rule by replacing all occurrences of the old shape with the new shape. This can lead to an substantial increase in the size of the code, which we hope to reduce in the future by abstracting collections. If the condition of a rule does not contain the shape, the system duplicates every line of code modifying one of the shape’s properties for the new shape property.

In the future we expect to deploy more sophisticated algorithms for learning from examples, and, more broadly, machine learning techniques to infer the intended behavior.

4. Implementation Aspects

We next describe the architecture of Pong Designer that enables the modification of run-time behaviors.

```
/* Identifiers   xFrom, yFrom, xTo, yTo
   are available  only if  the code is
  inside  a WhenFingerMovesOn rule.*/
// Shape: (other.x  == x1, other2.x == x2)
cx = x1,       cx = cx1,       cx = x1+w1
x = x1−w,    x = cx1−w,    x = x1+q1−w
x = x1, x = cx1,              x = x1+w1
x += xTo−xFrom,               x += xFrom−xTo
x += C_N,                     x = C_N
x += yTo−yFrom,               x += yFrom−yTo
// Same for y,  by replacing  x by y and vice−versa
// Mirror position  of a center  with other  two centers.
(cx,  cy) = (2*cx1−cx2, 2*cx2−cx1),

// Then angle of the speed.
angle = C_N,                  angle += C_N
// Angle from the position  of a shape to the finger
angle = angle(x1, y1, xTo, yTo)
velocity  *= C_F,             velocity  = C_F
color    = C_N
visible  = C_B

// Rectangular shapes
width += xTo−xFrom,           width += C_N
width = C_N                   width *= C_F
height += yTo−yFrom           height = C_N
height *= C_F                 height += C_N

// Circles
radius += C_N,                radius *= C_F
radius = C_N                  radius += xTo−xFrom ...

// Integer  boxes
// The identifier   "nv" represents  the new value
// if  the rule  is  triggered  by a number.
/* We write v instead  of "value" */
v = nv
v = nv / i  if  nv % i == 0
v = nv * i
v += C_N if C_N == 1 or −1
v = v1 + v2,                       v = v1 − v2
v = v1 * v2,                       v = v1 / v2
v = v1 * C_N,    v = v1
v += C_N if |C_N| > 1
v = C_N

// Text boxes
text = text1           text = text1 + text2
text = Constant
```

Figure 4. The language of actions that the game engine generates by decreasing priority for each property. For each property, we write "property" instead of NAME.property. "other" and "other2" are identifiers to describe other shapes. We abreviate "other.property" to "property1" and "other2.property" to "property2". cx is "center_x", w is "width" and h is "height". C_N is an integer constant, C_F a float constant and C_B a boolean constant.

```
// Template definition.
trait  TemplateShape {
  var shape: Shape
  def variants(s:   Shape): List[Expression]   = {
    shape = s
    if(condition)  List(result)   else  Nil  }
  def condition:  Boolean
  def result:   Expression
}
// Template to compare against other shapes
trait  TemplateOtherShape extends TemplateShape {
  var other_shape:  Shape
  override def variants(s:   Shape) = {
    shape = s
    var expressions  = Nil
    for(o  ← game.shapes) {
      other_shape = o
      if(other_shape  ≠ shape && condition)
        expressions  += result
} } }
// Regrouping templates in  parallel
trait  TemplateParallel  extends Template {
  def templates:  Traversable[Template[T]]
  def result:   Expression  = {
    var expressions  = Nil
    for(template  ← templates)
      expressions  += template.variants(shape)
    Parallel(expressions)
} }
// Regrouping templates in  block
trait  TemplateBlock extends Template {
  //Replace Parallel  by Block in  the previous  code
    Block(expressions)
} }
// Template matching horizontal  finger  tracking.
object TX_DX2 extends Template[Shape] {
  def condition  = ofType(TOUCHMOVE_EVENT) &&
      approx(shape.x−shape.prev_x, xTo−xFrom) &&
      !movementIsVertical
  def result  = "shape.x = shape.prev_x+(xTo−xFrom)")
}
// Template matching alignments
object TX_AlignLeft1 extends TemplateOtherShape {
  def condition  = approx(shape.x, other_shape.prev_x,  20)
  def result  =   "shape.x = other_shape.prev_x"
}
// Variants  for  x
object TX extends TemplateParallel {
  def condition  = shape.prev_x ≠ shape.x
  val templates = List(TX_DX2, TX_AlignLeft1)
}
// Global  template
object TShape extends TemplateBlock {
  def condition  = true
  val templates = List(TX, TY, TColor ...)
}
```

Figure 5. Templates producing code for a given shape. The templates pattern follow those in Figure 4

```
def codeGeneration(game, event,
                        actionsCondition,  existingActions)  = {
  actions  ← ()
  initialize templates
  for{shape ∈ game} {
    variants  ← TemplateShape.variants(shape)
    if{variants  ≠ ()} {
      actions  += ParallelExpr{variants}
    }
  }
  mergeCode{game, event, actions,
            actionsCondition,  existingActions}
}
```

Figure 6. codeGeneration: Algorithm which takes a "game", an "event", an optional condition "actionsCondition" under which actions in the game should be performed, a list "existingActions" of actions that are currently performed when this event is triggered. Outputs a sequence of actions describing the intended action merged with the previous ones.

```
def mergeCode(game, event, actions,
                   actionsCondition, existingActions) = {
(actionsCondition, existingActions) match {
  case (true, ()) ⇒
    actions
  case (true, "if("cond")" code_T "else" code_F) ⇒
    "if("cond")" mergeCode(..., actions, true, code_T)
    "else" mergeCode(..., actions, true, code_F)
  case (true, _) =>
    - Group by assigned property existingActions and actions.
    - Intersect expressions parallelExpr for the same property
    - If intersection is empty, take the new code.
    - Return the resulting block code.
  case ("newValue ⩽" B,
        "if(newValue ⩽" A")" code_T "else" code_F) ⇒
    if( B < A ) {
      "if(newValue ⩽" B")" mergeCode(..., actions, true, code_T)
       "else (if(newValue ⩽" A")" code_T "else" code_F")"
    }
    . . .
}
```

Figure 7. mergeCode: Algorithm which takes a "game", an "event", an optional condition "actionsCqondition" under which actions in the game should be performed, a list "existingActions" of actions that are currently performed when this event is triggered. Outputs a sequence of actions describing the intended action merged with the previous ones. Notice how conditionals are merged: The structure of the program remains consistent. Texts in quotes are the representation of a program.

```
def ruleMerge(game, event,
                   existingRule,  actionsCondition)  = {
if{existingRule  is not defined} {
  existingRule  ← emptyRuleFrom(event)
  //existingRule.code is empty
}
actionsCondition  ← "true"
if(event  is  a "Number change event")  {
  if(event  is  a "Number equal event")  {
    actionsCondition  ← "newValue == event.shape.value"
  }
  else  if(event  is  a "Number greater  event"){
    actionsCondition  ← "newValue ⩾ event.value"
  }
  else  if(event  is  a "Number less  event"){
    actionsCondition  ← "newValue ⩽ event.value"
  }
  else  if(event  is  a "Number positive  event"){
    actionsCondition  ← "newValue ⩾ 0 "
  }
  else  if(event  is  a "Number negative event"){
    actionsCondition  ← "newValue ⩽ 0 "
  }
}
rule.code  = codeGeneration(game, event,
                        actionsCondition,  existingRule.code)
game.rules  += rule
```

Figure 8. ruleMerge: Algorithm which takes a game with input and output state available, an event and an optional existing rule. Outputs a rule to describe the complete event handling.

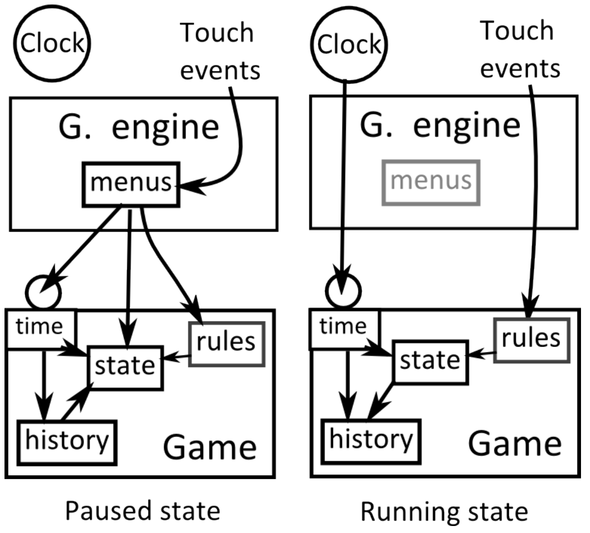


Figure 9. The two states of the game engine

4.1 Role of the game engine

Time plays an important role in the game engine. The game itself is not aware of the real time, only the game engine is. To manage time, Pong Designer can be in two different states. In the running state, the game runs naturally, whereas in the editing state, the game is paused and everything, including current time, is editable. Because of these two states, we made sure that the time of the game is tightly controlled by the game engine.

To control the time for the current game, the game engine performs the following actions synchronously and forever:

1. When in the running state, it updates the time of the game from a clock.
2. When in the editing state and if modified, it updates the time of the game from the time slider.
3. Displays the game.
4. When in the editing state, displays more information about the game state, such as if objects are static or if an object has been modified.
5. When in the editing state and activated, displays selectable events from the last 5 seconds in order to let the user create or modify rules.
6. When in the editing state, displays the game menu on top of it.

Concerning touch events, the game engine behaves differently. Because of the platform, touch events are received asynchronously. The game engine deals with them in two different ways:

1. When in the editing state, the game engine dispatches touch events to menus which in return modify the game state
2. When in the running state, the game engine dispatches touch events directly to the game

Figure 9 gives a summary of the interface between the game engine and the game in the two states The slider controlling time is included in the box "menus".

4.2 Time management by games

Games running with this game engine need to have the possibility to go back in time for at least a short period. Therefore, a game needs to store internally a 5-second history of all of its parameters and its shapes' parameters. Events are a particular case (see below) but are still recorded in a 5-second history. If the user wishes to start from the beginning, or if the game has been wrong and not corrected for the last five seconds, it is still possible to reset it to the initial state.

The data structure storing the history of parameters is a double linked list of values ordered by timestamps. This structure is parametrized in the type of the values, which can be anything among integers, floats, string and events. Except for events, we optimized the history by storing values only when they change, so that if the parameters of an object does not change, its history holds only one value.

When time elapses (forwards) to a new value set by the game engine, the following actions occur:

1. The game goes through all touch events that were stored asynchronously, and executes them according to its rules.
2. The game updates the physics by moving shapes and handling collisions and at the same time triggers synchronous events, like those from collisions, out-of-screen events and number change events.

The time can elapse backwards only when the game engine is in the editing state. In this case, if the user goes backwards in time with the slider (up to 5 seconds) the game reverts all its parameters and all its shapes' parameters to their value at the given time. If the user moves the time slider back to the right, the game executes forwards as if it was running.

4.3 Two kinds of events

How events are generated and stored is the key point to understanding how games are executed in Pong Designer. We distinguish two kinds of events.

Firstly, asynchronous events, such as touch gestures and accelerometer changes, are stored in a buffer. When the time increases to another value, the game flushes all these events. Because they are external to the game, they provide the necessary input to play it. After having triggered corresponding rules, these events are recorded in a 5-second history. When the game engine is in the editing state, they can therefore be selected by the user to create new rules or modify existing ones. When the user then lets the time elapse forwards in the editing state, the game replays these events from the edited history. It is thus possible to change the rules and to see their different outcome for the same touch input immediately, making it convenient to determine constants, for instance if rules are changing speed, position, etc.

The second types of events, synchronous events, such as collisions, out-of-screen events or number change detection, are detected after the physics is updated. When they are detected, these events might also trigger rules that are part of the game. They are also stored in a 5-second history. Although the user can still select them to create new rules and to modify existing ones, they will always be recomputed when the time elapses forwards, both in the editing state or in the running state.

4.4 Deployment on Android

We are compiling against the latest Android API version 17 (Jelly Bean) by using the SDK that Android provides. Our application is also compatible until the API 10 (Gingerbread). Because the SDK is written in Java, and because the Android virtual machine only deals with Java-like classes, we are using two different plug-ins to be able to program in scala: sbt and AndroidProguardScala. Because Scala libraries are not available on Android by default, the two plug-ins embed Scala libraries to provide a final stand-alone application. Our prototype application is available from the Android Play store as Pong Designer.

5. Discussion

Our purpose is twofold. The first objective is to reduce the gap between coding and testing, and the second is to allow the user to learn faster how to program by providing him a comfortable environment.

First, let us remark that there is an inherent duality between the code and the interface. This dual paradigm is representative of a major duality in the software development: compilation vs. testing, programmer vs. designer, engineering vs. marketing, developer vs. user, etc.

Because of too simple design decisions, Cooper et Al. discovered that for many systems, the interface often maps the code implementation, and do not meet the goals of the users[5]. For example, a program would like to ask if the user wants to save the changes, which in most cases should be done without asking. This happens because it reflects more or less the way the file system internally works.

The approach of self-reconfiguring interfaces is to try to reduce as much as possible the gap between the configuration and the execution. Reactive customization is at the core of self-reconfiguring interfaceS. The purpose is to empower the user with programming capacities, by specifying a desired behaviour on-the-fly. Coding should be done by the interface itself, so that the programmer would not spend too much time learning an API.

Bret Victor investigated the way courses currently teach programming [25] and depicted its bottlenecks. By comparing the program output to the code, Bret found principles for programming environments, if implemented correctly, would lead to a better understanding and a better learning curve for users and programmers.

> "Traditional visual environments visualize the code. They visualize static structure. But that's not what we need to understand. We need to understand what the code is doing."

To understand what the code is doing, we use visualization, debugging and verification systems. However, there are few programming environments that allow to directly manipulate what the code is acting on. Usually, any interaction of this kind only provides backwards pointers, such as retrieving the original position in the source of a compiled TeX file. We aim at providing more code What You See Is What You Get (WYSIWYG) customization features based on the manipulation of the outcome of the code, in order to demonstrate intended behaviors.

Bret identified the following list that users need for an enjoyable coding experience. We add a comment after each principle to show how we contribute to these guidelines.

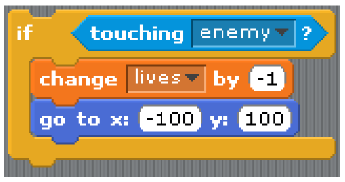


Figure 10. Code required in Scratch to describe consequence of a collision.

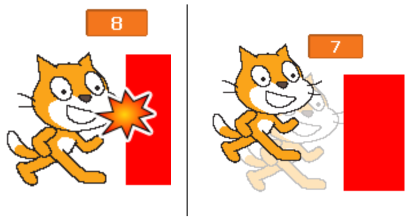


Figure 11. Hypothetical illustration of applying Pong Designer approach to the Scratch example. The developer pinpoints to a visually represented event, then changes the state into the desired one. The system infers the state transformation from the example demonstration by finding a function that maps the input to the output state.

- **Show the data** - We are showing the physical world
- **Show comparisons** - We have the time slider to compare the states between them
- **Get something on the screen as soon as possible** - Basic shapes can be added
- **Create by reacting** - We select events that happened to create rules.
- **Create by abstracting** - We are abstracting the demonstration into a general rule.

5.1 What needs to be improved

The main challenge will likely be to find the right trade-off between the complexity of the code we want to generate by demonstration and the visual simplicity. For example, it might be hard to design multiple winning conditions in our current game engine if they are described by some complex boolean formula. Moreover, generalizing conditions is currently impossible, for example checking if all numbers within a certain range are greater than 10, and adding a new number to the range.

6. Related Work

Most existing approach for game programming do not support inference of rules from demonstrations. We make an overview of some of those systems, as well as some of the approaches for inferring code from examples.

Scratch. Scratch is a game engine that helps to teach programming to children aged 8+. It provides all the typical structures of programming, loops, threads, if-constructs, tests, intersection detection, stylus, etc. There are two main screen areas in Scratch: one with programming blocks, and the other with the canvas. The objects on the canvas can be directly moved and rotated with the mouse, and programming has been made easier through assembling of compatible predefined blocks, which prevents the construction of programs that would not correct according to types and the syntax. The system is simple and seem appealing for teaching conventional programming through a graphical variation of the usual textual rendering of program text. While the code is running, it is possible to grasp objects, move and rotate them. However, we found that writing coordinates by hand, which is the only way to introduce specific coordinates into a program, can be cumbersome. Consider a situation where we wish to move the cat when a given line of code is executed to a position for which we do not know the coordinates. Currently, one needs to first move the cat round, note down the coordinates, and then enter them into the source code, which is much less immediate than in our system. In general, the programmer needs to define all constants by hand. Another important feature that we found lacking is the ability to go back in time to identify the desired behaviors. A major part of programming is to define the behavior of the interaction between two objects, but such behaviors cannot be defined on-the-fly in Scratch. If an object, let us say a cat, should loose a life when it touches a red enemy, the programmer normally creates the code such as in Figure 11. The creation of the analogous code is easier if we graphically set up the effects of the collision. To do so, in our system we just select the event on the screen, then change the position and the number of lives. The desired code is generated automatically. we believe that educational systems such as Scratch would also benefit from our demonstration-based approach

When it comes to program a single stand-alone behavior, e.g. for an enemy to try to reach a player, Scratch allows the user to program any looping constructions, branching conditions, lists and variables to achieve the desired result. Because of its programming paradigm, Pong Designer still lacks such explicit programming features. However, we could imagine in the future to program an AI by specifying a score which needs to be incremented, and the rules that increment the score. With some learning algorithms and input restrictions, the AI could learn to move a paddle towards a ball, or to go away from the player if the latter is in an invincible mode in order not to lose lives. For more sophisticated AI strategies and descriptions. we still need to enrich the interface and our language.

Programming by example. [14] reports that programming-by-demonstration paradigms are often Turing-complete, so is our engine. However, this completeness usually does not lower the complexity of programming non trivial tasks. Such paradigms become useful in the presence of a library, which directs the purpose of the programming. This is a reason why we choose to provide a direct support for physics, so that providing examples allows the programmer to quickly create games without having to worry about details. For example, he does not need to create the bouncing code when a collision occurs.

Gulwani et al. [7] [21] reports that programming constraints can be learned and generalized by their system through a given set of input/output examples. By using inductive synthesis with a DSL, they were able to find all the expressions that could match the inputs to the outputs. Their examples included text editing macros and spreadsheets. Our engine follows a similar algorithm on graphical input, although it is much less complex for now.

The way rules are refined according to multiple input-output examples is similar to the Version Space Algebra method which automatically learns programs from traces [12], as well as to Angelic nondeterminism [3], which also provides a methodology to fill the missing parts of code based on trace executions and specifications.

Quickdraw [4] is a graphical system which rebuilds precise graphics based on vague input. It also inspired us to enforce the robustness of our system against minor graphics specification errors.

Finding a way to manage coexistence between the code and its execution has already been a source of many more or less fruitful experiments. The Khan Academy [17] focuses more on "play with the code" than on its graphical output, which is vigorously criticized by [25]. Our approach incorporates many important points of this criticism to make graphical programming enjoyable.

Simula and Smalltalk. A pioneering object-oriented programming language Simula is an excellent programming model for physics-based games as well as other domains that can be viewed in analogy with the physical world. Smalltalk builds on this tradition and further emphasizes graphical environment and the ability to manipulate the state directly. Sánchez-Ruíz et al. [23] showed that 4th and 5th graders liked to program using the object-oriented programming interface Squeak and its graphics, but disliked correcting errors. Squeak provides a graphical interface, as well as contextual menus for on-the-fly editing purposes. Our tool also aims to provide a graphical interface that even children enjoy programming. The Morphic environment allows the user to program graphical interactions between objects called Morphs and has an object-oriented language inspired from Smalltalk. The structure of Morphs is organized around a hierarchy of traits and prototypes, which allows the user to factor behaviors and attributes. Similarly to our system, it let the user bind input events to actions. It provides a graphical editor as well as an interactive way of writing code for objects, especially prototypes.

Programming for phones. We also draw inspiration and insight from the TouchStudio/TouchDevelop project [24]. The TouchStudio/TouchDevelop project is related in the spirit to our work, because it also uses the hand to graphically program scripts on tablets and has a language that simplifies the general programming model.

Game engines. According to [2] there is a need to separate the game content from the game engine. For efficiency reasons, he asserts that there is a need to specialize the game engine according to the kind of game that can be produced. One of the main design goals of the game engine should be the speed of execution. With a proper scene manager, a dynamic collision engine and detection of visible objects, they were able to obtain a reasonable speed. The design of our game engine is similarly specific for the kind of games we would like to run. We are also taking inspiration from this paper to make our game engine faster, even if our collision engine is for the moment statical.

Construct 2 is a commercial HTML5 game prototyping engine with the associated community of developers and a portal for trading game components. It provides a game layout, a camera usually smaller than the layout, and lets the user add his or her own sprites. The game logic is on a separate sheet that is executed 60 times per second, resulting in professional quality of animation. We observed, however, that for some on-line games the authors mentioned that they had a hard time to debug their game logic. In our own experience, we observed that, for example, writing the specification to "spawn" objects (such as a bullet from a gun) requires the programmer to go back from the game to the game engine, to the image editor, and then to the event sheet. This process supports precise modifications, but misses the opportunity of intuitive contextual modifications. Therefore, we found this system would benefit from the techniques that we incorporated into Pong Designer.

Functional programming. Fruit [6] is a functional programming language that defines GUI logic as signals and signal transformers. Signals approximately correspond to continuous variables, and signal transformers are code that perform actions on signals, such as integrals or conditional assignments. With this approach, a Pong-like game is programmed with only 20 lines of code. Part of the efficiency of this approach can be found in our game engine, where parameters and events play the role of signals, and signal transformers resemble rules.

Sound processing. ChucK [26] is a strongly typed language designed to write functional audio synthesis programs. Its programming paradigm is to provide full control over time features, and to use an arrow operator which captures the sequential operations of programming. One of the specificity of ChucK is on-the-fly programming, which allows people to modify their program without having to interrupt the execution of the program, for example during a live performance.

Debugging environments. WhyLine [9] is a modern interactive debugging tool where the user can ask questions during debugging about why a certain change happened. By recording the execution trace, it is possible to solve complex debugging problems by navigating through history. Our system similarly uses time-backtracking to enable the precise design, refinement and modification of rules.

According to Lieberman [13], there is a huge gap between the environment of the code and the environment of the software. He suggests that visual users should be teachers for the interface of the software itself, which in return would act like as a learning student. His graphical programming environment includes the possibility to program macros by demonstration and to generalize them when translating them into code. He suggests that the generalization process is a key part of the learning of the software, and that small errors should be detected and corrected when generalizing. Similarly, our tool aims to be a learning student, which for example tries to correct small alignment mistakes made by the user who plays the role of the programmer. It produces macros that generalize the intended behavior provided by examples.

Tools and runtimes for existing languages. There are several tools that find, rank and present the most appropriated synthesized code portions to the programmer. InSynth [8] is an IDE extension which allows users to synthesize code snippets based on the type of the current expression. Although these approaches are not completely automatic due to the lack of complete specifications, they reduce the burden of the programmer. Similarly, our tool finds, ranks and presents different code portions, so that the user can choose among them based on their original intent. On another side of the spectrum, Chameleon [19] assists the programmer in the difficult task of choosing the best data type for the collections in a program.

Programming language extensions with constraints. Kaplan [11] and Comfusy [10] support the use of constraints as programming structures. Such structures allow programmers to work productively on explicit specifications rather than explicit code. The automatically generated code is thus less error-prone. Decreasing the number of potential errors is also the goal of domain-specific languages like those designed by Intentional Programming [20], which allows the programmer to work on a language that is closer to his needs. Our game engine also has a domain-specific rule-based language that is generated by the graphical selections made by the user.

7. Conclusions

Pong Designer enables to modifying games while they run, stepping back in time, and providing demonstrations of desired behaviors. The system infers corresponding rules and constraints, which can be manually modified afterwards. Based on object-based programming principles, users can create their game through moving and arranging elements while stepping through time. The system can generate code in a domain-specific language embedded in Scala, which runs on the Android platform using the standard toolkit.

We believe Pong Designer can be used to make games that are as fun to modify as they are fun to play. While there already exist games whose game worlds can be edited, the changes to behavior are currently limited, and there is a large gap between the sophisticated built-in behavior on the one side and simple customizations on the other side. We believe that Pong Designer leads reduces this gap, and we hope that this encourages experimentation and building of fun logic-based games and interactive games. We believe that the system can also be used for experiments exploring the learning and teaching of programming.

We are at this point confident that the approach can be successful in particular domains. The open question is the extent to which this success generalizes to broader domains, and the extent in which this paradigm can incorporate principles for managing complexity of larger applications.

Acknowledgments

We thank our shepherd and anonymous reviewers for useful feedback. We thank Sean McDirmid for useful discussions and comments. We thank Philippe Suter for his feedback about this project, as well as Eva Darulova for her feedback on the paper. We thank Lomig Megard, who greatly contributed to the new version of the project.

References

[1] H. Abelson and A. McKinney. AppInventor: app inventor for android, 2010.

[2] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46 –53, Feb. 1998.

[3] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, pages 339–352, 2010.

[4] S. Cheema, S. Gulwani, and J. LaViola. QuickDraw: improving drawing experience for geometric diagrams. In *SIGCHI Conf. Human Factors in Computing Systems*, CHI '12, 2012.

[5] A. Cooper, R. Reimann, and D. Cronin. *About face 3: the essentials of interaction design*. Wiley, 2012.

[6] C. Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, page 41–69, 2001.

[7] S. Gulwani. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, volume 10(2), 2012.

[8] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. Technical report, EPFL, SwissFederal Institute of Technology Lausanne, 2011.

[9] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, page 301–310, New York, NY, USA, 2008. ACM.

[10] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Comfusy: A tool for complete functional synthesis (tool presentation). In *CAV*, volume 6174, Berlin, 2010.

[11] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, page 151–164, New York, NY, USA, 2012. ACM.

[12] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on Knowledge capture*, K-CAP '03, page 36–43, New York, NY, USA, 2003. ACM.

[13] H. Lieberman. Mondrian: a teachable graphical editor. In *INTERCHI*, 1993.

[14] R. McDaniel. *Your wish is my command*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[15] S. Nasilowski. Codea: Create anything on your iPad with codea, 2011.

[16] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, and B. Silverman. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[17] S. Sal. Khan academy, 2012.

[18] L. Scirra. Construct 2: Create games. effortlessly., 2013.

[19] O. Shacham, M. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

[20] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, Oct. 2006.

[21] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, Berlin, Heidelberg, 2012.

[22] K. T. Stolee and T. Fristoe. Expressing computer science concepts through kodu game lab. In *Proc. 42nd ACM technical symposium on Computer science education*, SIGCSE '11, New York, NY, USA, 2011. ACM.

[23] A. J. Sánchez-Ruíz and L. A. Jamba. FunFonts: introducing 4th and 5th graders to programming using squeak. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, page 24–29, New York, NY, USA, 2008. ACM.

[24] N. Tillmann, M. Moskal, J. d. Halleux, and M. Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD, 2011.

[25] B. Victor. Learnable programming. http://worrydream.com/LearnableProgramming/, Sept. 2012.

[26] G. Wang and P. Cook. ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM international conference on Multimedia*, page 812–815, 2004.