

King Pong Designer - Online Graphical Rule-Based Game Programming on Tablets

Mikaël Mayer

Supervisor: Viktor Kuncak

Laboratory of Automated Reasoning and Analysis,
EPFL, Lausanne, Switzerland

`mikael.mayer@epfl.ch`

January 29, 2013

Abstract

King Pong Designer is an innovative game engine which allows users to graphically and completely interactively program multi-touch games on tablets like multi-player Pong, Brick-breaker, Pacman and many others. It allows programming mostly using only a finger by combining implicit and explicit features. We are looking forward to use this game engine for educational and entertaining purposes.

1 Introduction

The way most people program is by writing explicit code, which gets compiled to binary code. The use of libraries allows to reduce the burden of specifying every single program behavior. Several programming languages allow the user to have high-level structures, such as object-oriented languages, partial functions or even constraint programming. But a lot of time is still spent writing programs without seeing the result immediately. This is why we tried a different approach to programming that makes it easier.

In this report, we present our game engine named King Pong Designer. King Pong Designer is an innovative game engine which allows users to graphically and completely interactively program multi-touch games like multi-player Pong, brick-breaker, Pacman and many others. Its interaction is finger-based and through backtracking mechanisms allows programming by finger by combining implicit and explicit features.



Figure 1: The two-player game of Pong in our game engine

2 Related Work

Modern programming languages, like Kaplan for Scala [KKS12] or Comfusy [KMPS10], start to adopt the use of constraint 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 [SCC06], which allows the programmer to work on a language that is closer to his needs. Our game engine has also a domain-specific rule-based language that is generated by the graphical selections made by the user.

The idea of programming graphically has already inspired some people to design functional GUI programming [Ell01]. Our graphical approach is comparatively more state/rule-based rather than function-oriented.

Quickdraw [CGL12] is a graphical system whichs rebuilds precise graphics based on vague input. It inspired us to enforce the robustness of our system against minor graphics specification errors.

The way we handle objects and context menus is very similar to the educational object-oriented framework Squeak [SRJ08]. Its specificity is to make the interaction more user-friendly by being able to modify the game state on-the-fly.

The way our game engine is designed is basically comparable to the one in [BEW⁺98] but less complex. For example, our collision engine only



Figure 2: Time History in the Pacman game

detects static collisions, not dynamic ones.

Modern debuggers allow the possibility of going back in time to solve complex problems [KM08]. Similarly, our system uses time-backtracking to be able to design and modify rules precisely.

Our constraint system is mostly based on input-output examples provided by the user and generalized by the system, an approach that has been explored by Gulwani et al. [SG12, Gul12]

The way the rules are refined according to multiple input-output examples is similar to the Version Space Algebra method which automatically learns programs from trace [LDW03]. Is also related to that process the Angelic nondeterminism [BCG⁺10], which also provides a methodology to fill the missing parts of code based on trace executions and specifications.

The problem of merging multiple states during symbolic execution found in [KKBC12] is similar to the problem of merging game changes to one single rule.

Assisting the programmer to choose the best code according to specifications [SVY09] inspired us to do our semi-automatic rule generator where the user can choose the code that matches its specifications, if multiple code possibilities were found. If specifications consist in the code context and the type of the current expression, such interactive code suggestions based on types has also been explored in [GKP11].

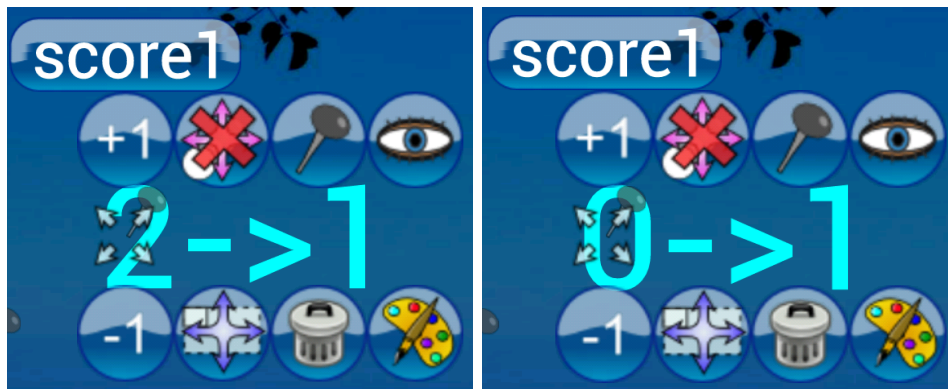
Finally, the way to manage coexistence between the code and its execution has been already source of many more or less fruitful experiments. The Khan Academy [Sal12] focuses more on “play with the code” than on its graphical output, which is vigorously criticized by [Bre12]. Our approach incorporates many important points of the second reference to make graphical programming enjoyable.

3 Lessons learned in this project

Our initial purpose was to create an interface that the user can program both graphically and/or with a text editor, with a preference to the first. During programming, we also learned a lot by recording bugs and interesting mistakes in the code, that might be used for future research.

Many upcoming challenges were not obvious at the beginning. First, what did we mean by graphical programming? How would we make it at least Turing-complete? What could be the purpose of it?

We will first introduce the challenges we had when designing the interface, then concerning the implicit and explicit user interaction, and then how to link the graphical and application programming interfaces by bootstrapping.



```
WhenIntegerChanges(score) { (oldValue, newValue) =>
  if(newValue == 5) {
    score1.value += -1 || score1.value = 1
  }
}
WhenIntegerChanges(score) { (oldValue, newValue) =>
  if(newValue >= 0) {
    score1.value += 1 || score1.value = 1
  }
}
```

(a.) Incoherent way of merging rules :

```
WhenIntegerChanges(score) { (oldValue, newValue) =>
  if(newValue == 5) {
    score1.value += -1 || score1.value = 1
  }
  if(newValue >= 0) {
    score1.value += 1 || score1.value = 1 //Double modification /\
  }
}
```

(b.) Coherent way of merging rules :

```
WhenIntegerChanges(score) { (oldValue, newValue) =>
  if(newValue <= -1) {
  } else if(newValue == 5) {
    score1.value = 1 // Merged code
  } else {
    score1.value += 1 || score1.value = 1
  }
}
```

Figure 3: Problem while merging two integer rules

3.1 Designing the interface

We first came up with the idea of programming a multi-player game of Pong (see Figure 1) in the following way. The user adds a moving ball, two walls, two paddles and two scores. To add game logic, she would run the simulation, add some input, pause the simulation, and specify rules modifying the game according to events and inputs.

Another challenge was to choose the platform. Instead of making it web-based or computer-based, we decided to implement our game engine on an Android tablet with the Android SDK. That way, we could use a simple finger-based interface, and our game could benefit from the multi-touch capabilities of the device to make multi-player games.

Later, as we were designing the API and GPI (Graphical Programming Interface), we faced the problem of storing the state of the game throughout the time, to be able to come back. After testing a few possibilities we learned that the most effective solution is to store an history inside each object, instead of storing all object histories in a single place. Our “ghost mode” allows the visualization of this history (see Figure 2). Histories consist in double-linked lists with timestamps. It should be noted that there are still minor unsolved problems with the time slider. Events can be recorded asynchronously, and sometimes when redesigning rules, events are not replayed correctly.

3.2 Designing the graphical user interaction

When it came to the rule generator, we had to deal with the possibility that two different input/output set for the same rule could contradict. We learned that having two successive shape modifications on the same rule for the same property would not help to keep the game rules in a maintainable state. Even more, by having two sequential rules, we could have undesirable effects such as double modification per rule execution.

For example, let us have the two rules in Figure 3. Because of ambiguity, the rule generator was able to find two different lines of code that would have matched the input to the output. If the user does not make an explicit choice, the two lines of code are stored into the program for further choice. Nevertheless, only the first line of code of the two is executed. For presentation purposes, we represented this behavior with the operator `||`, but in reality the expression is stored as a `ParallelExpressions(1: List[Expression])`.

We observed that just concatenating the rules (Figure 3 (a.)) makes the score to be modified two times in the same rule, which was not intended. Therefore, it was necessary to merge the *if* structures using disjoint intervals concerning conditions (Figure 3 (b.)). The two lists from

`ParallelExpressions` are merged by building a new `ParallelExpressions` with the following elements in order:

- The intersection of the two lists of possible codes.
- For each pair (a, b) of elements from the two lists, their merged behavior if applicable. Merging code lines is done in the following way:
 - If two code lines are again `ParallelExpressions`, we merged them as previously explained.
 - If one code line is a `ParallelExpressions`, we wrap the other into a `ParallelExpressions` as well and apply the previous algorithm.
 - If the two code lines only differ by a constant, we merge the constant into a function that will produce one of these two constants randomly. (see API page 15)
 - If the merge is not possible, we return nothing.

3.3 Designing the explicit user interaction

When the user provides an input/output example, we have seen that each code line is generated as a `ParallelExpressions` containing multiple code possibilities, where only the first one is executed.

To give the user more control about those possibilities, our initial idea was first to “unfold” a generated rule. For each `ParallelExpressions` containing n code lines in parallel, we would produce n times more rules where the `ParallelExpressions` has been replaced by some of its sub-expressions. Then we displayed a list from which the user could choose the rule corresponding to his need. For example, if the first change over x had 2 possibilities for a change from 20 to 25, and the second 3 possibilities for a change from 1 to 2, the system would have generated 6 rules, and would have let the user choose among them (see example below).

| | | |
|--|---|---|
| SomeRule { shape.x = 25 shape.value = 2 } | SomeRule { shape.x = 25 shape.value += 1 } | SomeRule { shape.x = 25 shape.value *= 2 } |
| SomeRule { shape.x += 5 shape.value = 2 } | SomeRule { shape.x += 5 shape.value += 1 } | SomeRule { shape.x += 5 shape.value *= 2 } |

The problem from this approach was the spatial exponential complexity of the interface. For example, for 4 lines of code for each there are 4 possibilities, the system let the user choose among $4^4 = 256$ different rules. This

| WhenIntegerChanges(score) { (oldValue, newValue) => | |
|---|-------------------------------------|
| : if(newValue <= 2) { | <input checked="" type="checkbox"/> |
| 2: ball.x = 176 | <input checked="" type="checkbox"/> |
| 2: ball.x += 53 | <input type="checkbox"/> |
| : } else { | <input checked="" type="checkbox"/> |
| : if(newValue <= 3) { | <input checked="" type="checkbox"/> |
| : } else { | <input checked="" type="checkbox"/> |
| 10: ball.y = 446 | <input checked="" type="checkbox"/> |
| 10: ball.y += -60 | <input type="checkbox"/> |
| : } | <input checked="" type="checkbox"/> |
| : } | <input checked="" type="checkbox"/> |

Cancel
OK

Figure 4: Expanding the code to let the user choose its specification

was not user-friendly. So many redundancies made the system unusable in practice.

To optimize that, we designed an algorithm to display the code in a multi-choice list in such a way that it allows to select the precise desired line of code each time there is a choice to make (see Figure 4). When the user selects OK, the algorithm does not drop unselected lines, it just places them in the queue of the **ParallelExpressions** just in case of a need to change the rule later.

For the previous example of 4 code lines with 4 possibilities, it would have produced only $4 \times 4 = 16$ items from which to choose instead of 256, which is been a spatial performance increase of a factor 16.


```

WhenFingerDownOn(textBox) {
  player.angle = -90 //more: player.angle += 90
  player.velocity = 0.066f
}.represents(List(
  ParallelExpressions(List(
    EApply(ESelect(ESelect(EIdentShape(player), "angle"),
      "$eq"),List(EConstantNumber(-90.0))),
    EApply(ESelect(ESelect(EIdentShape(player), "angle"),
      "$plus$eq"),List(EConstantNumber(90.0))))) ,
  ParallelExpressions(List(
    EApply(ESelect(ESelect(EIdentShape(player), "velocity"),
      "$eq"),List(EConstantNumber(0.066)))))

```







Figure 5: The compiled and interpreted version of a rule in the source code

3.4 “Bootstrapping” the code of games

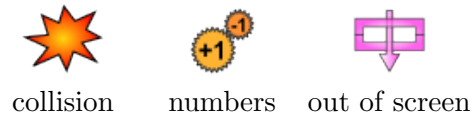
Although it is not possible to rewrite our game engine in our game engine itself, we introduce the possibility to bootstrap the code of created games. A game is currently saved in a valid Scala file, that uses the game API. Once linked to the original project and recompiled, this game can be loaded, and again modified and saved. The trick to do bootstrapping is to add a modifiable representation of the code such as in Figure 5.

4 Game Graphical Programming Interface (GPI)

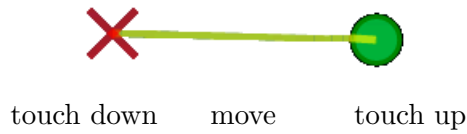
The general way to graphically program a game is the following:

1. Add shapes to the game and modify them : 
2. Launch the game  : The time elapses 
3. Pause the game 
4. At any time, to make permanent changes, go to the beginning by pressing the back button. 
5. Click on the “Create Rule” button 

6. Slide the time bar to the desired moment.



7. Select the event causing the rule



8. Change the game state with the menu



9. Choose the code from possibilities.

| | |
|----------------------|-------------------------------------|
| 2: score.value += -5 | <input checked="" type="checkbox"/> |
| 2: score.value = 0 | <input type="checkbox"/> |

10. Validate. To repeat the process, return to step 2

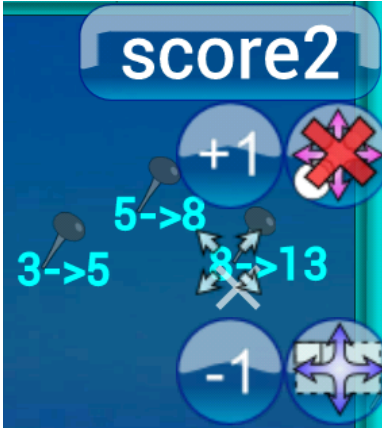
11. If you want to copy a shape, just select it and create the same shape type.

5 Graphically programmed Examples

5.1 Fibonacci sequence

It is simple to implement a simple interactive Fibonacci sequence. The way it works is the following: when we press on the first score, it goes through the sequence. When we press on the last score, it goes through the sequence in reverse.

1. Add the three scores to the game.
2. Set their respective values to 3, 5, 8
3. Launch the simulation
4. Press on the 8
5. Stop the simulation.
6. Go back in time, select the touchdown event on 8
7. Modify to 5, 8, 13
8. Select OK
9. Relaunch the simulation
10. Press on the 5
11. Stop the simulation
12. Go back in time, select the touchdown event on 5
13. Modify 5, 8, 13 to 3, 5, 8
14. Select OK

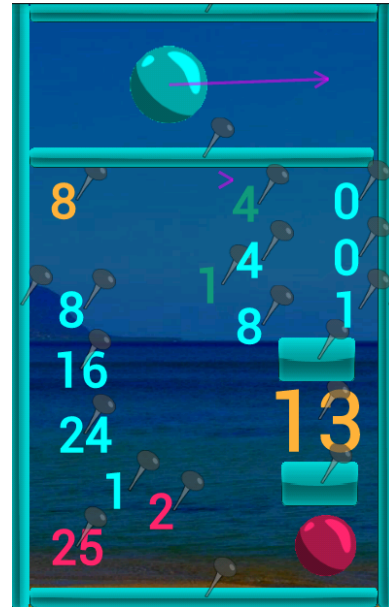


The screenshot shows a game interface with a blue background. At the top, there is a 'score2' display. Below it, there are several circular icons: a '+1' icon, a '-1' icon, and a red 'X' icon. In the center, there are three numbers: 3, 5, and 8. Arrows point from 3 to 5, from 5 to 8, and from 8 to 13. The text '3->5', '5->8', and '8->13' is displayed in cyan. Below the game interface, there is a list of events under the heading 'WhenFingerDownOn(score2) {'. The events are as follows:

| Event | Checked |
|--|-------------------------------------|
| 1: score.value = score2.value - score.value | <input checked="" type="checkbox"/> |
| 1: score.value = score1.value | <input type="checkbox"/> |
| 1: score.value += 2 | <input type="checkbox"/> |
| 1: score.value = 5 | <input type="checkbox"/> |
| 2: score1.value = score2.value | <input checked="" type="checkbox"/> |
| 2: score1.value += 3 | <input type="checkbox"/> |
| 2: score1.value = 8 | <input type="checkbox"/> |
| 3: score2.value = score.value + score2.value | <input checked="" type="checkbox"/> |
| 3: score2.value = score.value + score1.value | <input type="checkbox"/> |

5.2 Algorithm of Syracuse

It is possible to graphically program a game that computes the Syracuse sequence for any number, as well as factorial and any other functions. Here is a screen capture of the game that computes the Syracuse sequence. The game uses a ball to trigger the copy of values between $3n + 1$ (red) or $n/2$ (green) to top-left orange number, depending on the parity computed in the right-most digits. The bottom-right orange number can be customized through up and down rectangles. To copy the value from the bottom-right orange number to the top-left, we press the red button.



5.3 Brick-breaker

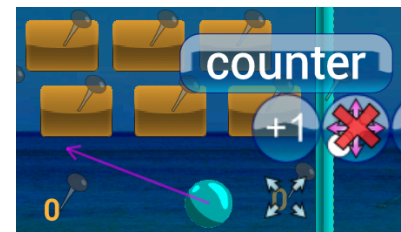
1. Add a ball, a rectangle named “block”, an integer named “score”, an integer named “count”
2. Change the ball’s velocity towards the rectangle



3. Launch the simulation
4. Stop after the collision
5. “Create rule”, select the collision event, edit effects
6. Move the block apart, increment the score, increment the count, OK

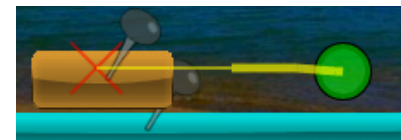


7. Duplicate the block to make several of them



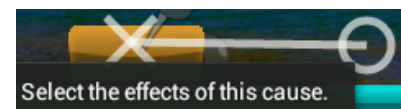
8. Add a rectangle, rename it paddle

9. Launch the simulation from the beginning, make the gesture to move the paddle.



10. Pause the simulation, select **create rule**

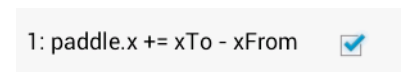
11. Select the finger trace



12. Move the paddle about approximately the same amount.



13. Confirm the rule.



14. Add a text displaying “Game over” in the middle, make it invisible

15. Make the ball go down and set up the velocity towards the wall below the paddle.

16. Launch the simulation. After the collision, stop

17. Create rule, select the collision event

18. Make the “Game over” textbox visible

19. Make the ball to stop moving. Confirm the rule



Shapes and Arenas constructors:

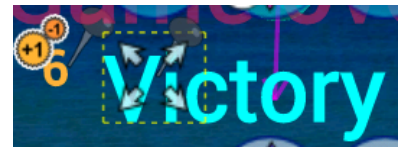
```
Shape
- Rectangular
  - Rectangle(x, y, width, height)
  - IntegerBox(x, y, width, height, value)
  - TextBox(x, y, width, height, text)
- Circle(x, y, radius)
Arena()
```

Modifiable properties:

```
shape.x : Float
shape.y : Float
shape.angle : Float
shape.velocity : Float
shape.noVelocity : Boolean // If true, the shape sticks to the game.
shape.color : Int
shape.visible : Boolean
rectangle.width : Float // For Rectangle, IntegerBox and TextBox
rectangle.height : Float // Idem
integerbox.value : Int
textbox.text : String
```

Figure 6: Shape properties

20. Go back to the beginning
21. Make a text displaying “Victory” in the middle, make it invisible
22. Launch the game.
23. Pause the game. Set up the count to $n - 1$, where n is the number of blocks. Relaunch the game.
24. After the ball hits a block, pause the game
25. Create rule, select the number change event, select condition “When count == n ”
26. Make the “Victory” text visible. Confirm the rule.



```

random(number1, number2, ...) // Chooses randomly between numbers
randomInterval(number1, number2) // Chooses a number in the interval
setCurrentArena(arena) // Sets the game to display arena
//Each new frame, if the condition is met, the code is executed
Whenever(condition) {
    codeModifyingGameState*
}
//If the finger press on the shape, the code is executed
WhenFingerDownOn(shape) { (x, y) =>
    codeModifyingGameState*
}
//If the finger is released on the shape, the code is executed
WhenFingerUpOn(shape) { (x, y) =>
    codeModifyingGameState*
}
//If the finger moves on the shape, the code is executed
WhenFingerMovesOn(shape) { (xFrom, yFrom, xTo, yTo) =>
    codeModifyingGameState*
}
//If the integer changes, the code is executed
WhenIntegerChanges(shape) { (oldValue, newValue) =>
    codeModifyingGameState*
}
//If a collision between shape1 and shape2 occurs, the code is executed
WhenCollisionBetween(shape1, shape2) {
    codeModifyingGameState*
}
//shape1 and shape2 go trough each other without testing collision
NoCollisionBetween(shape1, shape2)
//shape1 and shape2 go through each other but collision is reported
NoCollisionEffectBetween(shape1, shape2)

```

Figure 7: Game API constructors and rules

```

class PongGame extends Game {
  /** Game static values */
  var screenWidth = 480
  var screenHeight = 750

  /** Game Layouts */
  val arena1 = Arena() named "arena1"
  val wall = Rectangle(0, 0, 25, 750) named "wall"
  wall.noVelocity = true
  wall.color = -5266352
  arena1 += wall
  val ball = Circle(125.856f, 351.699f, 35) named "ball"
  ball.velocity_x = 0.098f
  ball.velocity_y = 0.242f
  ball.color = -19655
  arena1 += ball
  ...
  val paddle = Rectangle(179.924f, 694.477f, 120, 55) named "paddle"
  paddle.noVelocity = true
  arena1 += paddle
  val score1 = IntegerBox(50, 530, 60, 60, 0) named "score1"
  score1.noVelocity = true
  arena1 += score1

  WhenFingerMovesOn(paddle1) { (xFrom, yFrom, xTo, yTo) =>
    paddle1.x += xTo - xFrom // 2 more
  }
  WhenFingerMovesOn(paddle) { (xFrom, yFrom, xTo, yTo) =>
    paddle.x += xTo - xFrom // 2 more
  }
  Whenever(ball.y + ball.radius < 0) {
    ball.x = 241
    ball.y = 376
    score1.value += 1
  }
  Whenever(ball.y - ball.radius > screenHeight) {
    ball.x = 241
    ball.y = 376
    score.value += 1
  }
}

```

Figure 8: Pong game API example

6 Game Application Programming Interface (API)

The code syntax tree is Scala-like [OMM⁺04]. We added a few domain-specific language features to be able to efficiently create games.

First, we have 4 available shape types and constructors, plus one constructor for arenas, which are containers for shapes (see Figure 6).

These constructors are called in the class constructor to create the game. Also available are top-level functions to build rules (see figure 7) and to modify the game state. Figure 8 shows how to use everything to create a Pong game.

6.1 Language of actions

When the game state is modified, the rule generator compares the modification against the following list of possible changes (see Figure 9). The expressions are sorted by decreasing priority. When there is a star ★, it means that the system tolerates an error margin of 40% between the value specified in output and the code portion.

If multiple actions are available, the code generator wrap all actions in a parallel instruction, where only the first instance of the code is executed. For example, if the code is to move the shape from $x = 50$ to $x = 140$, the system will create the line `ParallelExpressions($x = 140, x + = 90$)`.

7 Future challenges

Such a game engine is likely to open a wide range of new concepts related to graphical programming. We were able to identify some directions of future research to make such products better. Here are some of them.

7.1 Improve execution speed

Based on the execution traces we observed, we found out where our engine speed could be improved.

Faster graphics About 10 to 30 % of the time is spent by software drawing. The use of OpenGL for example would help reduce this time.

Faster collision detection About 30 to 60% of time is currently spent detecting collisions. Better hashmaps and a quadtree could improve the efficiency of the current basic $O(n^2)$ algorithm.

Better game storing About 5-15 % of time is currently spent storing the state of the game at each time. Storing the state at longer intervals and then recomputing physics could allow us to improve the efficiency.

```

// Shape:
// The identifiers xFrom,yFrom,xTo,yTo are available only if
// the code is inside a WhenFingerMovesOn rule.
x += xTo-xFrom ★, x += xFrom - xTo ★,
x += Constant, x = Constant,
x += yTo-yFrom ★, x += yFrom-yTo ★
y += yTo-yFrom ★, y += yFrom - yTo ★,
y += Constant, y = Constant,
y += xTo-xFrom ★, y += xFrom - xTo ★

angle = Constant, angle += Constant
velocity *= Constant, velocity = Constant

color = Integer constant
visible = Boolean constant

// Rectangular shapes
width += xTo-xFrom ★, width = Constant,
width *= Constant, width += Constant

height += yTo-yFrom ★, height = Constant,
height *= Constant, height += Constant

// Circles
radius += Constant, radius *= Constant, radius = Constant
radius += xTo-xFrom ★

// Integer boxes
// If the code is inside a WhenIntegerChanges rule,
// then it can use the identifiers newValue, oldValue
value = newValue, value += newValue - oldValue,
value += oldValue - newValue, value = newValue / 2 if newValue % 2 == 0,
value = newValue * 2, value = newValue - oldValue,
value = oldValue - newValue, value = oldValue,
value += Constant if Constant == 1 or -1,
value = value1 + value2, value = value1 - value2,
value = value1 * value2, value = value1 / value2,
value = value1 * 2, value = value1,
value += Constant if abs(Constant) > 1, value = Constant

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

```

Figure 9: The language of actions that the game engine generates

7.2 Improve usability

Based on feedback, we have some ideas about how to make the game engine more user-friendly.

Direct rule modification Once a rule has been written, it would be good to be able to modify its constants by hand and to see their immediate effect in the game. A special display of its code would make this feasible. Deleting rules would also be a plus.

Typed objects and rules When an object is duplicated, all the rules linked to this object are currently duplicated. Instead, we could imagine that the rules would be based on types, not on objects. That way, we would avoid code duplication.

Saving and sharing If this game becomes popular, we should think of saving and loading games into a specific format that is suitable to send and share to friends.

Friction, gravity and cameras To create platform game, we would need to add friction, gravity and cameras in the game. That would allow us to create a broader range of games.

Better color and picture management What makes a game fun are the pictures and animations it is displaying. To allow to use predefined pictures and animations would make the game more interesting.

Programming abstractions To have support for arrays types and lambda expressions would provide a step forward to make game programming more enjoyable.

8 Evaluation to come

The evaluation of this experiment will come in a second phase. We are planning to have students of different ages and at different instruction levels to program or to modify an existing game. We will be able to compare several aspects of our game engine to state-of-the-art game engines. We plan to compare features like the speed of learning to program, the interest, the efficiency, and the robustness of programming with our tool.

The way we are going to record the user feedback will probably be with audio files.

9 Conclusion

King Pong Designer is a nice experiment that, if continued, will certainly have long-term effects on how to teach programming. To make everything

available to the programmer allows him to faster learn to program and to provide visual specifications. It also allows him to obtain the result immediately, without compilation efforts. The game engine allows the programming of a variety of 2D games, including multi-player games. The conclusion of this graphical approach is extremely positive.

We hope in the future to extend this game engine with other features and also to generalize this approach to other classes of program.

References

- [BCG⁺10] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. *SIGPLAN Not.*, 45(1):339–352, January 2010.
- [BEW⁺98] 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, February 1998.
- [Bre12] Victor Bret. Learnable programming. <http://worrydream.com/LearnableProgramming/>, September 2012.
- [CGL12] Salman Cheema, Sumit Gulwani, and Joseph LaViola. QuickDraw: improving drawing experience for geometric diagrams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’12, page 1037–1064, New York, NY, USA, 2012. ACM.
- [Ell01] Conal Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, page 41–69, 2001.
- [GKP11] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. Technical report, EPFL, SwissFederal Institute of Technology Lausanne, 2011.
- [Gul12] Sumit Gulwani. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, volume 10(2), 2012.
- [KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, June 2012.
- [KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe 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.

- [KM08] Andrew J. Ko and Brad 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.
- [KMPS10] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Comfusus: A tool for complete functional synthesis (tool presentation). In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, Proceedings*, volume 6174, pages 430–433. Springer-Verlag Berlin, Berlin, 2010.
- [LDW03] Tessa Lau, Pedro Domingos, and Daniel 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.
- [OMM⁺04] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.
- [Sal12] Shantanu Sal. Khan academy. <http://www.khanacademy.org>, 2012.
- [SCC06] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, October 2006.
- [SG12] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th international conference on Computer Aided Verification, CAV'12*, page 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.
- [SRJ08] Arturo J. Sánchez-Ruiz and Lisa 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.
- [SVY09] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.