Semester project – Pong Designer
Technical report

Author: Lomig Mégard
Supervisors: Prof. Viktor Kuncak and Mikaël Mayer

LARA/EPFL

June 6, 2013

1 Introduction

This report explains de different implementation choices taken during the semester. The reader should first read the theoretical report in order to have the context.

2 Type system

The type system relies on type classes in Scala. For example, the signature of a property restrains the generic type that can be used.

```
abstract class Property[T : PongType](...) {}
```

The `PongType` is defined in such a way we can translate an internal result of the interpreter to a Scala object, and the other way around.

```
trait PongType[T] {
  def getPongType: Type
  def toPongValue(v: Any): Value
  def toScalaValue(v: Value): T
  def clone(v: T): T
}
```

The following method illustrates how these types can be used. The `Context` contains some global information required for the evaluation, as for example the current finger movements.

```
def typecheckAndEvaluate[T : PongType](e: Expr): T = {
  typeCheck(e, implicitly[PongType[T]].getPongType)
  eval(e)(Context).as[T]
}
```

3 Expressions and statements

The AST of statements is illustrated in figure 1. All classes are immutable but can reference a property whom value can change. Statements can have side-effect, here modifying a property value with the `Assign` operation.

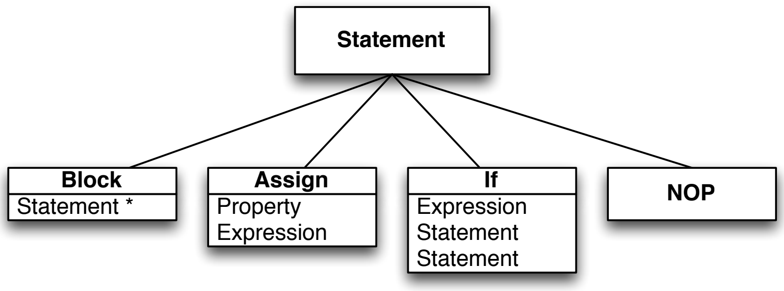

Figure 1: AST of statements.

The expressions cannot have side-effects. Here is a summarized list of the different kinds of expressions. The different expressions that use finger position and objects collision need a context to be evaluated. It is provided by the game engine that exposes the asynchronous events of the last terminated time step.

- Literals: `IntegerLiteral`, `StringLiteral`, ...
- Arithmetic: `Add`, `Minus`, `Mod`, ...
- Boolean: `And`, `LessThan`, `Equals`, ...
- Finger: `MoveOver`, `DownOn`, ...
- Collision
- Property

4 Game objects

This project uses JBox2D version 2.2.1.1 for its physical engine. Since Pong Designer needs to handle the history of all properties, a wrapper named `PhysicalObject` is built around each JBox2D body. It takes care of instantiating the body with arbitrary expressions in the world and of managing the different properties with their history.

The figure 2 describes how different kinds of game objects inherit from the same trait `GameObject`. It manages all the properties that belong to its implementation in two ways. First, each property is a class member of `GameObject` or of another sub-class if it is specific (an example is the circle radius). These members can be accessed only from the game engine itself, not from the user. This permits to have internally the right static Scala type for each property. Secondly, a map stores all the properties, even the

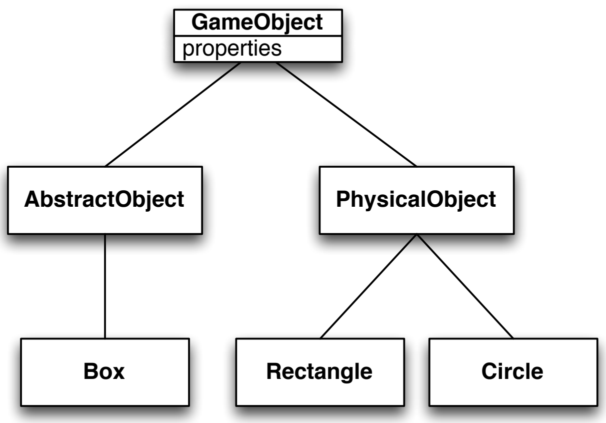

Figure 2: Hierarchy of game objects.

ones declared in sub-classes, with their name for key. This enables the user to have access to a property from a name, but the precise type will remain unknown (it is anyway not required by the user). The following example shows how we can build a statement to increment a value.

```
circle("x") := circle("x") + 1
```

This syntax is not very convenient but we assumed that this code would be generated by the game engine. It remains readable and meaningful.

5 Time management

To manage a bounded history with good performances, I implemented a `RingBuffer` collection. It permits to have a complexity in $O(1)$ for `append`, `head` and `last` among others.

6 Game loop

6.1 Time step

The game loop uses fixed time step. The following code corresponds to the main loop that calls `update` and `render` on the game engine. The `update` call is redirected on the game itself if it is running.

```
var canvas: Canvas = null
while(running) {
  canvas = null
  try {
    canvas = holder.lockCanvas()
    holder.synchronized {
      val beginTime = System.currentTimeMillis()
      var framesSkipped = 0

      // the view is the main game engine class
      view.update()
      view.render(canvas)

      val timeDiff = System.currentTimeMillis() - beginTime
      var sleepTime = FRAME_PERIOD - timeDiff

      if (sleepTime > 0) Try {
        Thread.sleep(sleepTime.toLong)
      }

      while (sleepTime < 0 && framesSkipped < MAX_FRAMES_SKIPPED) {
        // we missed a frame
        view.update()
        sleepTime += FRAME_PERIOD
        framesSkipped += 1
      }
    }
  } finally {
    if (canvas != null) holder.unlockCanvasAndPost(canvas)
  }
}
```

6.2 Update

The `update` method in the game follows a strict sequence of operations:

1. Evaluate the rules. The context can be used and contains all asynchronous events recorded since the last iteration. The collisions of the last physical step are also available.

2. Properties with new values are flushed to the physics engine.

3. Update the physical world using JBox2D. This performs new collisions and moves the objects.

4. Load new values from the physics engine to the properties.

5. Save the current state in the history.