Pong Designer

Another point of view

Lomig Mégard

LARA
Ecole polytechnique fédérale de Lausanne
lomig.megard@epfl.ch

June 13, 2013

Pong Designer

The user defines both the state and the **behaviour** using the programming by demonstration paradigm:

- The user creates a state where the pre-condition occurs
- Go back in time to select it
- Modify the state to show the post-conditions
- The game engine infers an appropriate rule

Issues of the old implementation

- Proof of concept
- Physics engine hard to maintain and debug
- Tunnelling effect
- Poor modularity

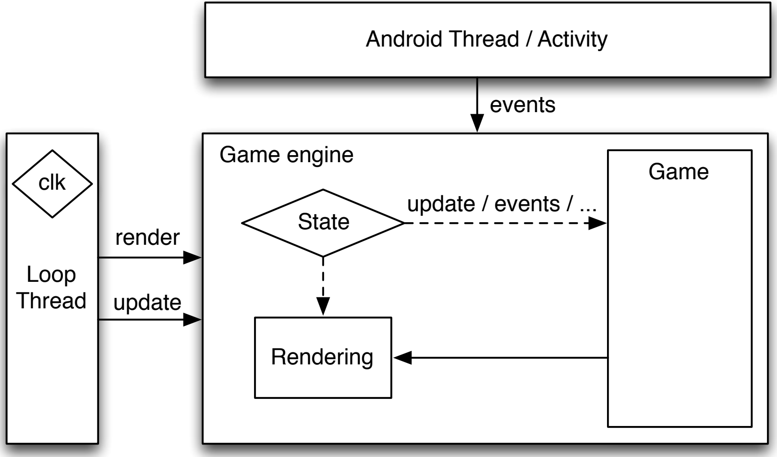
Goals of the new implementation

- Dedicated physics engine
- New ASTs for rules.
- Support to group of objects
- Maintainability and modularity

Overview

- Architecture
- Statements and expressions
- Type system
- Rules
- Categories
- Time management
- Physics engine
- Game objects
- One time step
- Future work
- Conclusion

Architecture - 1


Android Thread / Activity

events

Game engine

Game

clk

State

update / events / ...

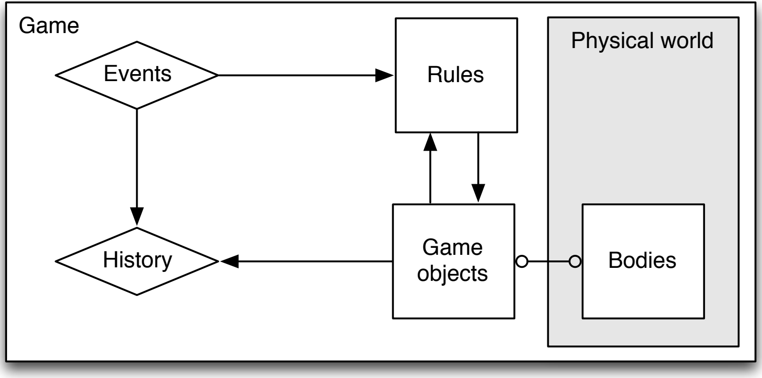
render

Loop Thread

update

Rendering

Architecture - 2

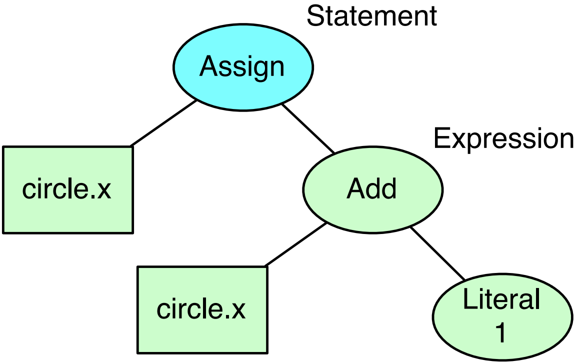


Statements and expressions - 1

- Permit to modify the rules, to reason about them
- Use of AST: convenient to manipulate
- Runtime typechecker and interpreter
- Statement with side-effects, without type
- Expressions without side-effects, with type

Statements and expressions - 2

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



Type system - 1

Each object property has two linked types:

- the expression type (use type classes)
- the value type (use Scala types)

```scala
abstract class Property[T : PongType](...) {
  def get: T
  def tpe = implicitly[PongType[T]]
  ...
}
```

Type system - 2

Benefit from the two types:

- the user can build expression with properties
- the game engine is typesafe

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

Rules

Permit to change the game state.

- One boolean expression for condition
- One statement for body
- Several triggers: `Whenever`, `On` and `Once`

```
whenever(Collision(ball, brick)) { Seq(
  brick("visible") := false,
  score("value") += 1
)}
```

Categories

- Unified behaviour for a group of objects
- Each object has one category
- Rules don't accept categories, use `foreach` to iterate

```
val bricks = new Category("Bricks")
rectangle("b1", x = 1, y = 0).withCategory(bricks)
rectangle("b2", x = 3, y = 0).withCategory(bricks)

val rule = foreach(bricks) { brick =>
  whenever(Collision(ball, brick)) { Seq(
    brick("visible") := false,
    score("value") += 1
  )}
}
```

Time management

Go back in time to create new rules.

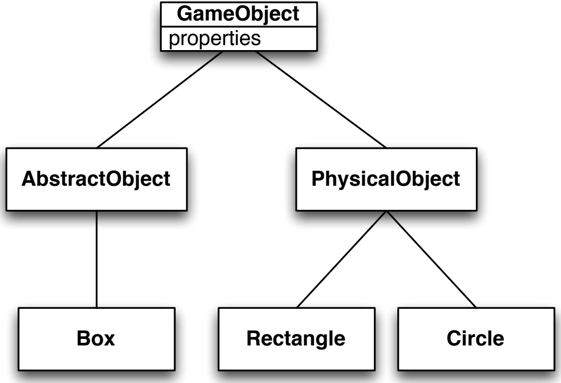
- Store game state at each time step
- Use a `RingBuffer` to handle bounded history

Physics engine

- Dedicated physics engine using JBox2D
- Only basic features are currently used
- Each JBox2D body is wrapped by a `GameObject`

Game objects

A GameObject handles its history and does the bridge with the type system



One time step

Fixed discrete time step. One game update is:

1. Evaluate the rules
2. New values are flushed to the physics engine
3. Update the physical world using JBox2D
4. Load new values from the physics engine
5. Save the current state in the history

Future work

This project produced multiple building bricks. The next step is to use them:

- Integrate the inferencer on top of these bricks
- Port the old UI to the new implementation

Conclusion

- Good design takes time
- JBox2D is fast but hard to learn
- Scala runs smoothly on Android