Presenting: the Casanova language
the next generation of game programming

Abstract

Games are extremely complex pieces of software which give life to animated virtual worlds. Games require complex algorithms, large worlds filled with many intelligent entities and high-quality graphics, and it all must run in real time.

Building general, high-performance frameworks capable of representing any virtual world has, until now, proven to be an elusive task; existing solutions either sacrifice performance (X3D browsers) or generality (game engines).

In this paper we present a new language, called Casanova, that aims at incorporating the knowledge that game engines represent with libraries or framework inside a language. The language is built around the idea of a single game state which contains not only its shape but also a series of rules that define how the state updates itself at every tick of the game engine. Rules are well suited for describing the declarative portions of the game update, but in order to keep the language expressive and easy to use we introduce behaviors, a system of imperative coroutines that run combined with the rules.

Rules and behaviors are coupled with powerful optimization techniques such as parallel execution, memory recycling and query optimization (for set processing) which greatly improve the performance of a game without requiring careful manual implementation of complex optimization algorithms.

1 Introduction

Game are: - complex - require performance

Low-level languages do not cut it fully.

We will: (A) discuss game constraints - discuss traditional approaches and their limitations - discuss similarities between game genres and try and find a general framework - define a language that is built around this general framework, and which makes it easy - reason on why we'd rather have a new language and not just a library - give syntax, typing, semantics (B) give optimization transformations of the language (C) give a detailed case study - give benchmarks that show how effective our optimizations are and how they are completely automated (they require no effort on the part of the programmer)

2 Existing game engines

Games and their complex requirements.

2.1 Related Work

- Data driven games - Entity hierarchy - Components - Against OO in games - Handmade optimizations - Nightmare of concurrency - SGL - Dungeon Siege - The X3D/VRML Way (general but no performance)

2.2 A different approach

Game engines as world interpreters: - data driven = game data + scripts = an actual, complex, interpreter

Rather than interpret, we build a language to compile the world rather than interpret it (less overhead).

A general framework for games: - rules - behaviors Rules and behaviors in existing genres: - arcade - puzzle - racing - rts - fps - rpg

We define a language around the idea of rules and behaviors, but with optimization firmly in mind. Also, with a language we can: - do syntactic abstraction, reducing boilerplate code (recycling memory efficiently, repetitive algorithmic optimizations, traversing the game state for updating, etc.) - encouraging a declarative style of programming which can yield to an increase in productivity by writing less code at a higher leve of abstraction, that is with lower maintenance costs - to simplify the approach to game programming and encouraging the emergence of a single idiom by providing only aspects essential to this domain, rather than using a very general purpose programming language with lots of paradigms (all somewhat acceptable but ultimately inadequate to the domain) - to enforce certain important properties at the semantics level, in order to allow the automation of important constructs such as complex cross-module optimizations

Our language does not take care about graphics. Casanova is a language capable of building the simulation module which can then be linked with another program capable of performing the rendering.

3 The Casanova language

In this section we present the Casanova language syntax, typing rules and semantics. Before we start, we will give a general idea of how the language works with a small example. We will build a very simple game where asteroids enter the screen from the top, scroll down to the bottom at different velocities and then disappear.

Note on syntactic sugar:
We will use the following syntactic sugar to increase source code readability:
Rather than write:

```
let! _ = b1
in b2
```

we can write:

```
do! b1
in b2
```

Rather than write:

```
let x = t1
in let y = t2
in t3
```

we can write:

```
let x = t1
let y = t2
t3
```

A Casanova program is composed of two portions:

- the state of the game, a series of types arranged hierarchically (typically at least one for the global state and one for the state of each entity); each portion of the game state may contain exactly one rule
- the main behavior, which is performs a series of instructions on all the mutable fields of the state (those marked as `Rule T` or `Ref T`; behavior execution is suspended at the `yield` statement, and resumed at the next tick

The state of our simple program is defined as:

```
type Asteroid =
  {
    Y      : Rule float
           :: \(self,dt) -> self.Y + dt * self.VelY

    VelY   : float
    X      : float
  }

type GameState =
  {
    Asteroids
        : Rule(Table Asteroid)
        :: \state -> [a | a <- !state.Asteroids && a.Y
            > 0]

    DestroyedAsteroids
        : Rule<int>
        :: \state -> !self.DestroyedAsteroids + count
           ([a | a <- !state.Asteroids && a.Y <= 0])
  }
```

In a type declaration, the `:` operator means "has type", while the `::` operator means "has rule". Rules can access the game state, the current entity and the time delta between the current and previous ticks.

In the state definition above we can see that the state is comprised by a set of asteroids which are removed when they reach the bottom. Removing these asteroids increments a counter, which is essentially the "score" of our pseudo-game. Each asteroid moves in the direction of its velocity.

The initial state is then provided:

```
let state0 =
  {
    Asteroids              = []
    DestroyedAsteroids     = 0
  }
```

After defining the state we must give an initial behavior. As can be easily noticed, our game does not generate any asteroids and so the initial state will never change. Since creating asteroids is an activity that certainly must not be performed at every tick (otherwise we could generate in excess of 60 asteroids per second: clearly too many), we need a function that is capable of performing *different* operations on the state depending on time. Since rules perform the *same* operation at every tick, they are unsuited to this kind of processing. Behaviors are built exactly around this need. The behavior for our game is the following:

```
let main state =
  let random = mk_random()
  let rec wait interval =
    {
      let! t0 = time
      do! yield
      let! t = time
      let dt = t - t0
      if dt > interval then
        return ()
      else
        do! wait (interval - dt)
    }
  let rec behavior() =
    {
      do! wait (random.Next(1,3))
      do! state.Asteroids.Add
          {
            X      = random.Next(-1,+1)
            Y      = 1
            VelY   = random.Next(-0.1,-0.2)
          }
      if state.DestroyedAsteroids < 100 then
        do! behavior()
      else
        return ()
    }
  in behavior()
```

Our behavior declares a random number generator and then starts iterating a function that waits between 1 and 3 seconds and then creates a random asteroid. When the number of destroyed asteroids is greater than 100, the function stops and the game ends (games end when their main behavior terminates).

Notice that behaviors are expressed with two different syntaxes: an ML-style syntax for pure terms (those which read the state and simply perform computations) and an imperative-style syntax for impure terms (those which write the state and interact with time such as wait). The imperative syntax loosely follows the monadic syntax of the F# language, where a monadic block is declared within `{}` parentheses, monadic operations are preceded by either `do!` or `let!` and returning a result is done with the `return` statement.

3.1 Syntax

In the remainder of this section we will adopt the following conventions:

- capitalized items such as `Program` and `StateDef` are grammatical elements
- quoted items such as `'type'` and `'GameState'` are keywords that must appear as indicated
- items surrounded by `[ ]` parentheses such as `[EntityName]` are user-defined strings

The program syntax starts with the definition of the state (a series of type definitions with rules) and is followed by the entry point (the initial state and the initial behavior):

```
Program   ::= StateDef
```

```
               Main
StateDef ::= EntityDefs
```

A type definition is comprised of one of various primitive types such as integers, floating point numbers, two- or three- dimensional vectors, etc. combined into any of the usual composite types known to functional programmers such as tuples, functions, records and sum types. Also, type declarations may contain a rule (which is simply a term, even though with the limitation that only pure functional terms are allowed inside rules):

```
TypeDef  ::= TypeDef'
           | TypeDef' :: Rule

TypeDef' ::= '()' | 'int' | 'float' | 'Vector2' | ...
           | TypeDef × TypeDef
           | TypeDef → TypeDef
           | '{' Labels '}'
           | TypeDef + TypeDef
           | Modifier TypeDef
           | [EntityName]

Labels   ::= Label; Labels
           | Label

Label    ::= [Name] ':' TypeDef

Rule     ::= Term
```

A `Modifier` for a type definition allows to make a field mutable (`Rule` or `Ref`), or to use queries to manipulate that field (`Table`). Also, another important modifier is `Foreign` which can be seen as a programmer annotation that tells the compiler how a certain field is just a pointer to another portion of the state and as such it must not be processed recursively:

```
Modifier  ::= Rule
            | Ref
            | Table
            | Foreign
```

Entities are definied as a series of type definitions with a name which can be referenced anywhere in the state; the last entity to be defined is the game state itself:

```
EntityDefs  ::= 'type' 'GameState' = TypeDef
              | EntityDef
                EntityDefs

EntityDef   ::= 'type' [EntityName] '=' TypeDef
```

The various entity names are simply replaced with their type definition in the remainder of the program, according to the $[\![\bullet]\!]_{\texttt{MAIN}}$ translation rule:

$$[\![\text{type EntityName} = \texttt{TypeDef};\ \texttt{EntityDefs};\texttt{Main}]\!]_{\texttt{MAIN}} =$$

$$[\![\texttt{EntityDefs};\ \texttt{Main}]\!]_{\texttt{MAIN}}[\texttt{EntityName} \mapsto \texttt{TypeDef}]$$

$$[\![\texttt{TypeDef};\ \texttt{Main}]\!]_{\texttt{MAIN}} = \texttt{TypeDef};\ \texttt{Main}$$

The actual type definition of the state may be extracted from the program with the $[\![\bullet]\!]_{\texttt{STATE}}$ transformation, which extracts the type definition and erases all the rules from it; this means that two entities may have the same type with different sets of rules. The $[\![\bullet]\!]_{\texttt{STATE}}$ transformation inductively removes all rules from a type declaration:

$$[\![T{::}\ \texttt{Rule}]\!]_{\texttt{STATE}} = [\![T]\!]_{\texttt{STATE}} \qquad [\![T_1 \times T_2]\!]_{\texttt{STATE}} = [\![T_1]\!]_{\texttt{STATE}} \times [\![T_2]\!]_{\texttt{STATE}}$$

$$[\![T_1 + T_2]\!]_{\texttt{STATE}} = [\![T_1]\!]_{\texttt{STATE}} + [\![T_2]\!]_{\texttt{STATE}}$$

(...)

A term can be a simple, ML-style functional term (we do not give all these possible definitions because they are fairly known [ML syntax and types]) or an imperative behavior. Functional terms can read references with the `!` operator and can use a Haskell-style table-comprehension syntax:

```
Term         ::= ‘let’ [Var] ‘=’ Term
                 ‘in’ Term
               | ‘if’ Term ‘then’
                    Term
                 ‘else’
                    Term
               | Term Term
               | !Term
               | ... (* other ML-style terms: fun,
                  types, head, tail for tables, etc. *)
               | [ Term | Predicates ]
               | ‘{’ Behavior ‘}’

Predicates   ::= ϵ
               | [Var] ‘<-’ Term, Predicates
               | Term, Predicates
```

A behavior defines an imperative coroutine that is capable of reading and writing the state and manipulating time. Behaviors can be freely mixed with terms. The simplest behavior simply returns a result with `return`. The result of a behavior can be plugged inside another behavior with `let!`, which behaves like a monadic binding operator. A reference can be assigned inside a behavior with `:=`; a behavior can suspend itself until the next tick (`yield`) and it may read the current time with `time`.

Behaviors can be combined into more complex behaviors with a small set of combinators. A behavior may spawn another behavior with `run`, be executed in parallel with another behavior with ∨ or ∧, be suspended until another behavior completes (⇒), be repeated indefinitely (`repeat`) and be forced to execute in a single tick (`atomic`):

```
Behavior    ::= ‘return’ Term
               | ‘let!’ [Var] ‘=’ Term
                 ‘in’ Term
               | Term := Term
               | ‘yield’
               | ‘time’
               | ‘run’ Term
               | Term ∨ Term
               | Term ∧ Term
               | Term ⇒ Term
               | ‘repeat’ Term
               | ‘atomic’ Term
```

The main program is comprised of two terms: the initial state and the initial behavior:

```
Main   ::= ‘let’ state0 = Term
           ‘let’ main = Term
```

3.2 Type System

Our language is strongly typed. We will omit some type declarations when obvious, and our language will make use of type inference. Typing rules for ML-style terms are the usual ML-style typing rules [ML syntax and types]; for example:

$$\frac{\Gamma \vdash t_1 : U \quad \Gamma, x : U \vdash t_2 : V}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 \ : V} \ \text{LET} \qquad \frac{\Gamma \vdash c : \textit{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{if } c \texttt{ then } t_1 \texttt{ else } t_2 : T} \ \text{IF}$$

$$\frac{\Gamma \vdash r : \textit{Ref } T}{\Gamma \vdash !r : T} \ \text{REF-GET} \qquad \frac{\Gamma \vdash r : \textit{Rule } T}{\Gamma \vdash !r : T} \ \text{RULE-GET}$$

(...)

Table comprehensions are types thusly:

$$\frac{\texttt{decls}(\Gamma, \texttt{ts}) \vdash t_1 : T}{\Gamma \vdash [\ t_1 \mid t_s\ ] \ : \textit{Table } T} \ \text{TABLE} \qquad \texttt{decls}(\Gamma, \epsilon) = \Gamma$$

$$\frac{\Gamma \vdash t : \textit{Table } T}{\texttt{decls}(\Gamma, (\texttt{x} \leftarrow \texttt{t}, \texttt{ts})) = \texttt{decls}((\Gamma, \texttt{x} : \texttt{T}), \texttt{ts})}$$

$$\frac{\Gamma \vdash t : \textit{bool}}{\texttt{decls}(\Gamma, (\texttt{t}, \texttt{ts})) = \texttt{decls}(\Gamma, \texttt{ts})}$$

(...)

Terms cannot have types with rules in them, that is rules can only be used in the top-level state definition. We will use the above notation for typing rules, where to be more precise we should have written $[\![T]\!]_{\texttt{STATE}}$ instead of simply `T` for each type to make sure that rules do not appear in a type annotation inside a term.

We also state another informal restriction, that is function types may not have rules; so, a type such as $(U :: \textit{Rule}) \to V$ is forbidden and generates a compile-time error.

We introduce another type, `Behavior T`, which allows us to type behaviors. Instances of `Behavior` can be constructed but never eliminated within a program: behaviors are eliminated implicitly inside the semantics. The first typing rules for behaviors are the monadic typing rules which allow to build and consume basic behaviors:

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \texttt{return } x : \textit{Behavior } T} \ \text{RETURN}$$

$$\frac{\Gamma \vdash t_1 : \textit{Behavior } U \quad \Gamma, x : U \vdash t_2 : \textit{Behavior } V}{\Gamma \vdash \texttt{let! } x = t_1 \texttt{ in } t_2 : \textit{Behavior } V} \ \text{BIND}$$

Behaviors may also be suspended for a tick (to wait for an application of all rules or to synchronize between behaviors, for example), they may read the current time (in fractional seconds) or they may spawn other behaviors:

$$\frac{}{\texttt{yield} : \mathit{Behavior}\ ()}\ \text{YIELD} \qquad \frac{}{\texttt{time} : \mathit{Behavior}\ \mathit{float}}\ \text{TIME}$$

$$\frac{\Gamma \vdash t : \mathit{Behavior}\ ()}{\Gamma \vdash \texttt{run}\ t : \mathit{Behavior}\ ()}\ \text{RUN}$$

Behaviors are the only places where unrestricted assignment of the state may happen; behaviors may indiscriminately write any portion of the state:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Ref}\ U \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma t_1 {:=} t_2 : \mathit{Behavior}\ ()}\ \text{REF-SET}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Rule}\ U \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma t_1 {:=} t_2 : \mathit{Behavior}\ ()}\ \text{RULE-SET}$$

We also support a small behavior calculus. Two behaviors may be executed concurrently (the first one that terminates returns its result while the other behavior is discarded), or they may be executed in parallel (when both terminate their results are returned together). A behavior may also act as a guard for another behavior, that is until the first behavior does not terminate with a result the second behavior is kept waiting. Finally, a behavior may be repeated indefinitely or it may be forced to run inside a single tick:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Behavior}\ U \\ \Gamma \vdash t_2 : \mathit{Behavior}\ V\end{array}}{\Gamma t_1 \vee t_2 : \mathit{Behavior}\ (U + V)}\ \text{CONCURRENT}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Behavior}\ U \\ \Gamma \vdash t_2 : \mathit{Behavior}\ V\end{array}}{\Gamma t_1 \wedge t_2 : \mathit{Behavior}\ (U \times V)}\ \text{PARALLEL}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Behavior}\ (U + ()) \\ \Gamma \vdash t_2 : U \rightarrow \mathit{Behavior}\ V\end{array}}{\Gamma t_1 \Rightarrow t_2 : \mathit{Behavior}\ V}\ \text{GUARD}$$

$$\frac{\Gamma \vdash t : \mathit{Behavior}\ ()}{\Gamma \vdash \texttt{repeat}\ t : \mathit{Behavior}\ ()}\ \text{REPEAT}$$

$$\frac{\Gamma \vdash t : \mathit{Behavior}\ ()}{\Gamma \vdash \texttt{atomic}\ t : \mathit{Behavior}\ ()}\ \text{ATOMIC}$$

The inability to eliminate a behavior unless inside another behavior is important because it allows us to force rules to not contain behaviors; thanks to this limitation we can ensure that the execution of rules may only read from the state and never write to it, and so rules can be made to behave as if they are executed simultaneously without risking complex interdependencies. This simplifies many instances of game programming; for example, consider the rules seen in the example at the beginning of the section:

```
Asteroids
    : Rule(Table Asteroid)
    :: \state -> [a | a <- !state.Asteroids && a.Y >
       0]

DestroyedAsteroids
    : Rule<int>
    :: \state -> !self.DestroyedAsteroids + count([a |
        a <- !state.Asteroids && a.Y <= 0])
```

If rules are executed sequentially from top to bottom, then when an asteroid is eliminated then that same asteroid will not be available anymore when computing the sum of asteroids waiting for deletion.

Types are used for restricting rules. In particular, inside the definition of an entity `type EntityName = TypeDef`, all its rules must have type (given that the rule has type `Rule T`):

```
GameState × EntityName × float → T
```

For convenience any subset of `GameState, EntityName, float` may be used in practice.

The final restrictions limit the acceptable terms for the `Main` program; the program is defined as:

```
StateDefinition

let state0 = t₁
let main   = t₂
```

and we require that t_1 must have type:

$$[\![\texttt{StateTypeDefinition}]\!]_{\texttt{STATE}}$$

and t_2 must have type:

$$[\![\texttt{StateTypeDefinition}]\!]_{\texttt{STATE}} \rightarrow \texttt{Behavior}\ ()$$

where

$$(\texttt{StateTypeDefinition};\ \texttt{Main}) = [\![(StateDefinition; \texttt{let state0} = ...)]\!]_{\texttt{MAIN}}$$

3.3 Semantics

Rule semantics (->, =>), everywhere function, script semantics.

4 Optimizations and compilation

4.1 Optimization

Optimization with: - memory recycling rather than re-allocation - parallel execution - query optimization - on Windows, WP7 and the Xbox (and the iPad?)

4.2 Compilation

Compilation strategy: - syntax - F# expression tree - regular F# compiler - graphics and networking in other .Net languages - sample compilation from very simple example in Section “Casanova Language”

5 Case study

Player, Projectiles, Asteroids: - rules (position, collision detection, score) - behaviors (input for player, generation of asteroids, end game)

6 Benchmarks

- Windows, Xbox, Wp7 (, iPad?) - memory recycling - parallel execution - query optimization

7 Conclusions and future work

Taming the complexity of virtual worlds is a task that has never been fully accomplished: - general purpose but slow engines - single purpose but fast engines

In this paper we presented the Casanova language: - a language for games, rather than a library in a general purpose language - that reduces development costs - without sacrificing flexibility - without sacrificing performance (through aggressive automated optimizations)