Racket on the Playstation 3? It's not what you think...

Dan Liebgold

Naughty Dog, Inc.
Santa Monica, CA

RacketCon 2013

THE LAST OF US™

Motivation

- In games, programmers create code; artists, designers, animators, sound designers create data
- We often want to create data like we create code
  - Effect definitions, animation states & blend trees, event & gameplay scripting/tuning, sound metadata
- We want powerful abstractions, flexible syntax, and language well matched to each domain
- Domain Specific Languages to the rescue!

Motivation

- In games, programmers create code; artists, designers, animators, sound designers create data
- We often want to create data like we create code
  - Effect definitions, animation states & blend trees, event & gameplay scripting/tuning, sound metadata
- We want powerful abstractions, flexible syntax, and language well matched to each domain
- Domain Specific Languages to the rescue!

NAUGHTY DOG

Motivation

- In games, programmers create code; artists, designers, animators, sound designers create data
- We often want to create data like we create code
  - Effect definitions, animation states & blend trees, event & gameplay scripting/tuning, sound metadata
- We want powerful abstractions, flexible syntax, and language well matched to each domain
- Domain Specific Languages to the rescue!

NAUGHTY DOG

Motivation

- In games, programmers create code; artists, designers, animators, sound designers create data
- We often want to create data like we create code
  - Effect definitions, animation states & blend trees, event & gameplay scripting/tuning, sound metadata
- We want powerful abstractions, flexible syntax, and language well matched to each domain
- Domain Specific Languages to the rescue!

Motivation

- In games, programmers create code; artists, designers, animators, sound designers create data
- We often want to create data like we create code
  - Effect definitions, animation states & blend trees, event & gameplay scripting/tuning, sound metadata
- We want powerful abstractions, flexible syntax, and language well matched to each domain
- Domain Specific Languages to the rescue!

NAUGHTY DOG

Scheme

- We had experience using Common Lisp before to create our own implementation language (GOAL)
- Lisp supports creating data like code
- We built *DC* in Racket
  - Used Racket (MzScheme initially) because it's a good Lisp, is open source, and has quality libraries and implementation

Scheme

- We had experience using Common Lisp before to create our own implementation language (GOAL)
- Lisp supports creating data like code
- We built *DC* in Racket
  - Used Racket (MzScheme initially) because it's a good Lisp, is open source, and has quality libraries and implementation

Scheme

- We had experience using Common Lisp before to create our own implementation language (GOAL)
- Lisp supports creating data like code
- We built *DC* in Racket
  - Used Racket (MzScheme initially) because it's a good Lisp, is open source, and has quality libraries and implementation

Scheme

- We had experience using Common Lisp before to create our own implementation language (GOAL)
- Lisp supports creating data like code
- We built *DC* in Racket
  - Used Racket (MzScheme initially) because it's a good Lisp, is open source, and has quality libraries and implementation

How?

- Racket program that evaluated typed “Racket-ish” code that generates data usable by C++ runtime.
- Usage of syntax was the prime enabler of rapid DSL development, but also a source of much inefficiency and confusion.
- Error reporting was slow to develop, since it required careful usage of syntax info, which was difficult and confusing.

How?

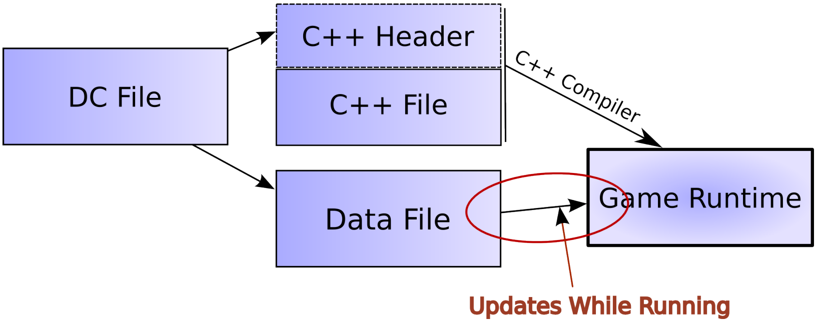
- Racket program that evaluated typed "Racket-ish" code that generates data usable by C++ runtime.
- Usage of syntax was the prime enabler of rapid DSL development, but also a source of much inefficiency and confusion.
- Error reporting was slow to develop, since it required careful usage of syntax info, which was difficult and confusing.

How?

- Racket program that evaluated typed “Racket-ish” code that generates data usable by C++ runtime.
- Usage of syntax was the prime enabler of rapid DSL development, but also a source of much inefficiency and confusion.
- Error reporting was slow to develop, since it required careful usage of syntax info, which was difficult and confusing.

NAUGHTY DOG

Architecture



Example

Let’s define a player start position:

```
(define-export *player-start*
 (new locator
      :trans *origin*
      :rot (axis-angle->quaternion *y-axis* 45)
      ))
```

Start with some types

```
(deftype vec4 (:align 16)
  ((x float)
   (y float)
   (z float)
   (w float :default 0)
   ))
```

Start with some types

```
(deftype vec4 (:align 16)
  ((x float)
   (y float)
   (z float)
   (w float :default 0)
   ))
```

```
struct Vec4
{
  float m_x;
  float m_y;
  float m_z;
  float m_w;
};
```

Types continued

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))
```

```
(deftype locator ()
 ((trans point :inline #t)
  (rot quaternion :inline #t)
  ))
```

Types continued

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))
```

```
(deftype locator ()
 ((trans point :inline #t)
  (rot quaternion :inline #t)
  ))
```

Types continued

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))
```

```
struct Locator
{
  Point m_trans;
  Quaternion m_rot;
};
```

Define a function

```
(define (axis-angle->quat axis angle)
 (let ((sin-angle/2 (sin (* 0.5 angle))))
  (new quaternion
       :x (* (-> axis x) sin-angle/2)
       :y (* (-> axis y) sin-angle/2)
       :z (* (-> axis z) sin-angle/2)
       :w (cos (* 0.5 angle))
       )))
```

Define some instances

```
(define *y-axis* (new vec4 :x 0 :y 1 :z 0))
(define *origin* (new point :x 0 :y 0 :z 0))
```

```
(define-export *player-start*
 (new locator
      :trans *origin*
      :rot (axis-angle->quaternion *y-axis* 45)
      ))
```

Define some instances

```
(define *y-axis* (new vec4 :x 0 :y 1 :z 0))
(define *origin* (new point :x 0 :y 0 :z 0))
```

```
(define-export *player-start*
 (new locator
      :trans *origin*
      :rot (axis-angle->quaternion *y-axis* 45)
      ))
```

How we use these definitions in C++ code

```
...
#include "dc-types.h"
...
const Locator * pLoc =
  DcLookupSymbol("*player-start*");
Point pos = pLoc->m_trans;
...
```

THE LAST OF US™

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

The Last of Us on the Playstation 3

- 16 programmers on the game project
- 20 designers
- 100 artists & animators
- 6000 DC files
- 120Mb of DC source, 45Mb of DC target binary files
  - Dynamically loaded into about 5Mb of managed heap space

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

NAUGHTY DOG

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

Experience

- Racket power, library support a big win
- Syntax transformation source location and performance hindered us
- S-expression based language a tough sell to industry programmers, as well as designers, and non-technical types
  - ...especially when paired up with Emacs as the editing platform.
  - Although once learnt many programmers and designers were expand and extend the language effectively
- Functional nature of the system is a big win, allowing data to be flexibly transformed to just the right runtime representation

Questions?