CEL (Camera Evaluation Language)

DOCUMENTATION

"Where we pretend to know what we're doing."

Doc. Version: 0.1.0

Created June 10, 2010 and last modified: June 22, 2010

---

by Marc Christie, Roberto Ranon, Tommaso Urli

Contents

1 Introduction

The *Camera Evaluation Language (CEL)* is a language for reasoning about images produced by rendering a 3D scene from a camera. Expressions written in CEL can evaluate, for example, the relative size of an object in the image, or how much it is occluded by other objects. This kind of information is useful to check if an application is properly displaying some relevant objects (e.g., in a videogame, to check of the main character is at least a certain size on screen, and not too much occluded), or in automatic camera control approaches, to check how much a camera is satisfying some requirements.

This document is organized as follows: Section 2 describes how one can write CEL scripts and their semantics; Section 3 explains how to build CEL on MacOS, Linux and (not yet) Windows; Section 4 describes the operations which can be performed using the Test application; Section 5 contains details about the licensing of CEL and other third party software we used. Appendix A describes the grammar of the CEL language.

2 The CEL Language

CEL is composed by:

- *Rendering Operators*, that given a camera and part of a 3D scene will derive a *Pixel Set*;
- *Pixel Set Operators*, that act on Pixel Sets and derive other Pixel Sets or numerical values;
- common mathematical, logical and set operators and relations.

CEL operators are listed in Table 1.

A pixel set PS is a set of pixels, each with x, y, z coordinates (and no color). A PS derived from a rendering operator is roughly the content of the Z-buffer after rendering the subscene (more exactly, not all the Z-buffer, but just the Z-buffer cells where a value has been written during rendering). Additionally, each pixel in PS has an attribute called *side*, whose value has meaning when the rendering operator performs more renderings (as in the case of the `CR` operator).

A simple CEL script is the following:

```
Declare
   // Area of an object in pixels
   Area(obj) = Count(R(obj));

Evaluate
   Area("transporter1");
```

A CEL script has two sections, `Declare` and `Evaluate` (both mandatory). As the names suggest, the `Declare` section can contain declaration of new operators written in the language, while the `Evaluate` section will contain a sequence of CEL expression, possibly using the operators declared previously, which will be evaluated when the script is parsed. In the following, we describe the CEL operators that can be used when declaring new operators or writing expressions.

The expression `R(obj)` takes the object called *obj* in the 3D scene and derives a pixel set by rendering just *obj* from the current camera (CEL assumes that a default

camera is defined). We can use the operator `!` and write `R(!obj)` to render, instead of the *obj*, all the 3D scene but *obj*. In this case, all pixels in the resulting pixel set will have *side=front*. When *obj* is not in the camera view volume, the resulting pixel set will be empty.

The expression `CR(obj)` takes the object called `obj` in the 3D scene (more specifically, a node in the .scene file) and derives a pixel set by rendering it six times, the first time using a camera as the default one (but with 90 degrees FOV, both vertically and horizontally), and the other five times rotating the camera 90 degrees around the camera axes (i.e. performing rendering to a cube map). In this case, all pixels from each rendering will have a different value of the *side* attribute (*front, right, top, ...*).

Starting from a pixel set `PS`, we can compute some numbers:

- `Count(PS)` is the number of pixels in `PS`;
- `MaxX(PS)` is the maximum x coordinate of all pixels in `PS`; we can also compute `MinX(PS)`, `MaxY(PS)`, ...
- `AvgX(PS)` is the average x coordinate of all pixels in `PS`; we can also compute `AvgX(PS)`, `AvgY(PS)`, ...

While `Count` works for each pixel in a pixel set, min, max and avg operators only take into account the pixels which have $side = front$. These operators can be used, for example, to declare operators that compute the relative sizes of an object on screen:

```
Declare
   Area(obj) = Count(R(obj))/(imageWidth*imageHeight);
   Height(obj) = (MaxY(R(obj) - MinY(R(obj))/imageHeight;
   Width(obj) = (MaxX(R(obj) - MinX(R(obj))/imageWidth;
```

`imageHeight` and `imageWidth` are predefined constants in CEL, and respectively correspond to the height and width of the viewport to which the default camera renders.

Additionally, we can compute the minimum distance between two pixel sets by using `Distance(PS1, PS2)`. This can be useful, for example, to evaluate the relative distance of two objects on screen.

The operator `Overlap(PS1, PS2)` returns a pixel set `PS3` with all the pixels in `PS1` that have the same $x, y, side$ coordinates of some pixels in `PS2`. For example, a framing property (i.e. checking if an object is inside some frame in the rendered image) can be defined and evaluated by the following script:

```
Declare
   HERE WE SHOULD DECLARE A SCREEN-ALIGNED FRAME CALLED QUAD1
   Framing(T, SAS) = Count(Overlap(CR(T), CR(SAS)))/Count(CR(T))

Evaluate
   Framing("transporter1", QUAD1);
```

The operator `CoveredBy(PS1, PS2)` returns a pixel set `PS3` with all the pixels in `PS1` that have the same $x, y, side$ coordinates, but greater z coordinate, of some pixels in `PS2`. In other words, *CoveredBy* returns those pixels in `PS1` which overlap pixels of `PS2`, and are "behind them" (i.e. more distant from the camera). This

can be used, for example, to create operators to evaluate how much an object is occluded in the image rendered from the current camera:

```
Declare
   Occluded(obj) = if (Count(R(obj)>0)
      Count(CoveredBy(R(obj)),R(!obj))/Count(R(obj))
   else -1;

Evaluate
   Occluded("transporter1");
```

The script shows that we can also use conditional expressions in CEL scripts.

Finally, we have some operators to reason about relative positioning of objects on the screen. The operator `Left(PS1, PS2)` returns all pixels of `PS1` whose x coordinate is less than any pixel in `PS2`. Similarly, the operators `Right(PS1, PS2)`, `Above(PS1, PS2)`, `Below(PS1, PS2)` (the last two considering the y coordinate) respectively return the pixels of `PS1` that are right of, above or below all pixels in `PS2`. These operators only take into account the pixels which have $side = front$. With the `Left` operator, we can for example define an operator that evaluates how much an object is left of another object in the image generated from the camera:

```
Declare
    LeftOf(obj1,obj2) = if (Count(R(obj1))>0)
        Count(Left(R(obj1),R(obj2)))/Count(R(obj1))
    else -1;

Evaluate
    LeftOf("transporter1","transporter2");
```

Operator or Relation	Returns
R(obj)	PS = the set of pixels p that results from rendering *obj* from *camera*, with $p_{side} = 0$ for each p
CR(object)	PS = the set of pixels p that results from rendering six times *object* from the position of *camera*, using 90 degrees FOV, perspective projection and view direction towards a face of a cube centered in the camera, with $p_{side} = 0, \ldots, 5$ depending on which side of the cube p belongs to
$Max_x(PS, side)$ Min_x, Max_y, Min_y Max_z, Min_z	$max(\{p_x \mid p \in PS \wedge p_{side} = side\})$ ... ...
$Avg_x(PS)$ Avg_y, Avg_z	$avg(\{p_x \mid p \in PS\})$ ...
$Overlap(PS_1, PS_2)$	set of pixels in PS_1 that have the same $x, y, side$ coordinates of some pixels in PS_2
$CoveredBy(PS_1, PS_2)$	set of pixels of PS_1 that would be covered by pixels of PS_2 if we rendered together the subscenes that produced PS_1 and PS_2
$Left(PS_1, PS_2)$ *Right, Above, Below*	set of pixels of PS_1 that are left of any pixel in PS_2, considering only pixels with $p_{side} = 0$ ...
$Distance(PS_1, PS_2)$	$min(distance(p, p')$ where $p \in PS_1, p' \in PS_2$, , considering only pixels with $p_{side} = 0$

Table 1: Operators and relations in CEL. PS, PS_1, PS_2 denote pixel sets.

3 Compiling CEL

At the current state of development, CEL can be compiled under MacOSX, Linux and Windows. The code bundle available from the CEL website contains an Xcode project, a set of Makefiles and a Visual Studio 2008 solution, therefore building it is pretty straightforward whatever your platform.

Follow the instructions in the next sections to know what to do before running the build phases.

3.1 MacOSX

To compile CEL under MacOSX you need:

- the **Xcode** developer's tool chain, which you can get from your MacOSX setup disc, and
- the **Ogre SDK**, which is freely downloadable from the Ogre website, we have used Ogre SDK 1.6.3, but any latter version should be good.

Most libraries, under MacOSX, are shipped in form of a framework bundle (a file with extension *.framework*) which contains all the files needed to develop using the library, Ogre SDK follows this convention. After downloading the Ogre SDK, you will find the framework bundle under **OgreSDK/lib/release/Ogre.framework**. Make sure to copy it to **/Library/Frameworks** before compiling.

The Xcode project shipped with the CEL code is configured assuming that the Ogre SDK is installed in **/Applications/OgreSDK**. If you stick with this convention everything is going to be at the right place when compiling. Otherwise we assume that you're expert enough to modify the paths in the project.

The last thing to do before building the project is to set an `OGRESDK_HOME` environment variable to point **/Applications/OgreSDK**. You can set this variable by typing, in your Terminal:

```
open ~/.MacOSX/environment.plist
```

Note: always remember to log out and log in to make the environment variable effective.

After all these operations have been carried out, you can just open the CEL Xcode project and click on "Build" or "Build and Run".

3.2 Linux

A set of qmake files is ready on the repository. You can use these files to build CEL under Linux.

3.3 Windows

A ready-to-use Visual Studio solution to build CEL under Windows will be ready as soon as possible. However if you manage to get it build before us, just make us know so that we save some time ;)

4 Running CEL

Even if the CEL API can be invoked from any standard Ogre application, integrating the library into one's source code can take some time and, moreover, requires some programming skills. For this reason, together with the CEL's source code, we provide a simple test application which can be used to give CEL a try and understand its functioning; the application is compiled during the normal CEL build procedure.

The test application allows one to load a 3D scene from a DotScene (*.scene*) file, load some CEL scripts (*.cel*) and evaluate them against the scene. This section will describe ways to operate the test application.

4.1 Conventions and defaults

When the application starts, the test scene located ad `trunk/media/scene` is loaded, together with a test CEL script. You can modify the default script to make some tests or learn by example about the language syntax. By convention all the test scripts (even the default one, `omniscript.cel`) have a lowercase alphanumeric filename ending with ".cel" and are located in the `trunk/build/Scripts` directory.

Within the application, you can use **W**, **A**, **S**, **D** and mouse pointing to fly around the 3D scene.

4.2 Using the test console

The test application is equipped with a console which allows you to perform some basic operations. You can spawn or hide the console by pressing the **Tab** key on your keyboard. Here is a quick summary of the available commands.

- `help` prints a summary of the available commands.
- `load <filename>` allows to load a CEL script from the scripts directory, you must specify the filename without ".cel" extension.
- `reload` reloads the currently loaded script, this enables one to modify the script and execute it again. The same result can be obtained by pressing **R** on the keyboard when the console is inactive.
- `evaluate` evaluates the currently loaded script against the scene. The same result can be obtained by pressing **E** on the keyboard when the console is inactive.
- `clear` clears the console.
- `exit` or `quit` terminates the application.

The result of the evaluation will be flushed in the console, if you have multiple expressions in a single file (this is the case of `omniscript.cel`), you will have numbered results:

```
> evaluate
  Result[0] = 29372
  Result[1] = 2233
  ...
```

To exit the application just press **Esc** with the console closed or use the **quit/exit** command.

5 Licensing and Acknowledgements

CEL (Camera Evaluation Language) is a research project by Marc Christie (INRIA-Rennes, France), Roberto Ranon and Tommaso Urli (both from HCI-Lab, University of Udine, Italy), for the latest info see www.cameracontrol.org/language. If you use, modify, or simply find this code interesting please let us know at info@cameracontrol.org.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, or go to www.gnu.org/copyleft/lesser.txt.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

Details about CEL's licensing are included at the top of every source file.

5.1 Ogre

A fundamental part of CEL relies on Ogre rendering engine (www.ogre3d.org) and our most sincere thanks go to the Ogre community and Torus Knot Software, for developing such a great piece of software. At the current moment, we are not redistributing any of the code from Ogre project and we don't take any responsibility about its licensing. Read more about this topic at www.ogre3d.org/licensing before installing or redistributing it.

5.2 OgreMax

Portions of this work is provided by OgreMax (www.ogremax.com), OgreMax was written by Derek Nedelman. Derek Nedelman assumes no responsibility for any harm caused by using this code.

5.3 TinyXML

OgreMax makes use of TinyXML, an open-source piece of software which is used to parse and visit XML documents. TinyXML is currently maintained by Lee Thomason (www.grinninglizard.com).

A Grammar

In the following, we describe the BFN grammar for CEL scripts.

```
<CEL> ::=
  [ <list_of_includes> ]
'Declare'
  [ <list_of_declarations> ]
'Evaluate'
  [ <list_of_evaluators> ]

<list_of_includes> ::= [ <list_of_includes> ] 'include'  '"' filename '"'

<list_of_declarations> ::= [ <list_of_declarations> ] <declaration>

<declaration> ::= <leftbody_declaration> '=' <expression> ';'

<leftbody_declaration> ::=  <symbol> | <symbol> '(' <list_of_leftArguments> ')'

<list_of_leftArguments> ::= [<list_of_leftArguments> ','] <symbol>

<expression> ::= DOUBLE
    | '(' <expression> ')'
    | "sqrt" '(' <expression> ')'
    | "sqr" '(' <expression> ')'
    | "sin" '(' <expression> ')'
    | "cos" '(' <expression> ')'
    | "tan" '(' <expression> ')'
    | "asin" '(' <expression> ')'
    | "acos" '(' <expression> ')'
    | "atan" '(' <expression> ')'
    | "atan2" '(' <expression> ',' <expression> ')'
    | "exp" '(' <expression> ')'
    | "count" '(' <pixelsetExpression> ')'
    | "maxx" '(' <pixelsetExpression> ')'
    | "minx" '(' <pixelsetExpression> ')'
    | "maxy" '(' <pixelsetExpression> ')'
    | "miny" '(' <pixelsetExpression> ')'
    | "distance" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | '-' <expression>
    | <expression> '-' <expression>
    | <expression> '+' <expression>
    | <expression> '*' <expression>
    | <expression> '/' <expression>
    | <expression> '^' <expression>
    | <symbol>
    | <symbol> '(' <list_of_rightArguments> ')'
    | "if" '(' <condition> ')' <expression> "else" <expression>

<condition> ::= <expression> <CMP> <expression>

<list_of_rightArguments> ::= [ <list_of_rightArguments> ','] <expression>

<pixelsetExpression> ::=  "Cuberender" '(' <targetExpression> ')'
    | "Render" '(' <Targets> ')'
    | "Silhouette" '(' <pixelsetExpression> ')'
```

```
    | "CoveredBy '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | "LeftOf" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | "RightOf" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | "AboveOf" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | "BelowOf" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'
    | "Overlap" '(' <pixelsetExpression> ',' <pixelsetExpression> ')'

<targetExpression> ::= <target>
    | '!' <target>
    | '!' '[' <list_of_targets> ']'


<list_of_targets> ::= [ <list_of_targets> ',' ] <target>

<target> ::= '"' targetName '"'

<list_of_evaluators> ::= [ <list_of_evaluators> ] <evaluator>

<evaluator> ::= <expression> ';'
```