Celluloid

Language Reference Manual

Charlie Robbins
Sean McDuffee
Dan Federman
David Flerlage
Blake Arnold

1. Introduction

This reference manual describes the syntax and semantics of the Celluloid programming language and is not intended to be a tutorial. The document is divided into seven sections that give understanding of the language. While we hope to give the reader the full technical specifications of the Celluloid language and its constructs, we give a disclosure that some of the information will change while development and limitations become more apparent.

1.1 Motivation

As the landscape of digital media and technology evolve, so must the tools designed to support it. In particular, digital media as a performance art has become a much more prevalent and respected form of art over the past decade. Celluloid is designed to fill a gap in the current set of existing computer graphics and digital media tools; as a reactive media sequencing language. Celluloid enables artists to create and mix digital media in an intuitive and succinct manner.

The applications for Celluloid's reactive nature expand however, beyond just media sequencing into more complicated digital media installation art using real-time continuous inputs. Celluloid has therefore been designed with these applications in mind, and contains a syntax that allows compact and elegant representations of such systems.

1.2 Philosophy

When creating a new tool, one must recognize what niche the tool fills with regards to other existing tools. Celluloid is not meant to be a general purpose or graphics programming language, but rather a domain-specific reactive and media

sequencing language. However, it is obvious to the designers of Celluloid that many of the preexisting hardware and software abstractions necessary for media sequencing require the power a more general purpose or graphics language. To compensate, Celluloid circumvents its self-imposed limitations by including a construct for injecting executable Java code directly into the language.

The ability to weave Java code into Celluloid code will enable Celluloid to be extensible to many types of input and output. Celluloid provides a mechanism for encapsulating input and output methods that the user can expand upon with the media they are trying to use for sequencing.

Celluloid is also designed to be human readable; statements are designed to make sense in pseudo-natural language, while still having the flow of a programming language. This syntax also aims to be as simple as possible while still retaining the necessary functionality.

There are, however, several features that we are not able to implement in our initial release. The most notable missing feature is that Celluloid lacks the capability for a REPL (read-eval-print loop). This is largely due to lack of dynamic programming support in the target language Java 1.6. In addition, it was not feasible to implement synchronization primitives for time synchronization across multiple inputs and outputs. We plan to address these limitations in future releases, and many have made it into the road map towards Celluloid 2.0.

2. Lexical conventions

2.1 Line structure

2.1.1 Logical Lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more physical lines by following the explicit line joining rules.

2.1.2 Physical Lines

A physical line is a sequence of characters terminated by an end-of-line sequence. Any of the standard platform line termination characters can be used.

2.1.3 Explicit Line Joining

Two or more physical lines may be explicitly joined together into logical lines using the backslash character (\) as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following physical line forming a single logical line with the backslash and NEWLINE deleted. For example:

```
if condition1 and condition2 and condition3 \
  and condition4 and condition5 and         \
  condition6 and condition7
```

A line ending in a backslash cannot include a comment, and is not used to continue comments onto a following line. A backslash can be legally used only at the end of a physical line preceding the newline.

2.1.4 Implicit Ling Joining

Expressions between parentheses, *when … do*, *unless … do* can be split over multiple lines without backslashes. For example:

```
in timeline1 do
  when condition1 and condition2 and condition3  #when 1, 2, 3
       and condition4 and condition5 do          #and 4, and 5
    [code_block]
  end
end
```

Implicitly joined lines may have comments on them.

2.2 Code Blocks

Code blocks consist of multiple logical lines, and can be nested. Code blocks are used to determine the grouping of statements. The beginning of a code block is delineated with either a "do" or "{". The end of a code block that is begun with a "do" is delineated with an "end", while the end of a code block that is begun with a "{" is delineated with a "}". These two syntaxes are semantically equivalent and are available as a matter of style to developers.

2.3 Whitespace

Whitespace, as defined as spaces and tabs (not newlines), is solely used to separate tokens. A logical line that contains only whitespace or a comment is ignored.

2.4 Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of a logical line.

2.5 Identifiers and Keywords

Identifiers are described by the following lexical definitions:

```
identifier  ::=  (letter|"_") (letter | digit | "_")*
letter      ::=  lowercase | uppercase
lowercase   ::=  "a"..."z"
uppercase   ::=  "A"..."Z"
digit       ::=  "0"..."9"
```

2.5.1 Primitive Type Keywords

timeline	time	number
string	boolean	

2.5.2 User Defined Type Keywords

device	constraint	event
accepts	requires	input
output	audio	video
new	announces	announce

2.5.3 Function and Block Declaration Keywords

function	predicate	return
in	local	

2.5.4 Conditional, Loop, and Event Accessor Keywords

when	at	and
or	every	unless
not	start	now

2.5.5 Playback Modifier Keywords

play	pause	rewind
ffwd	stop	seek

2.6 Literals

2.6.1 Number literals

All number literals are parsed as double precision floating point literals. Floating point literals are described by the following lexical definitions:

```
floatnumber     ::=   pointfloat | exponentfloat
pointfloat      ::=   [intpart] fraction | intpart "."
exponentfloat   ::=   (intpart | pointfloat) exponent
intpart         ::=   digit+
fraction        ::=   "." digit+
exponent        ::=   ("e" | "E") ["+" | "-"] digit+
```

The implementation of the number literal is based on the hardware's implementation of the Java double. Range and overflow details are left to the individual machines Java implementation. Celluloid does not support single-precision floating point numbers nor integral numbers. Conversion of an appropriate number to a string is handled by the built-in function numberToString(). Conversions from a number to a string are implicit in the equality operator. Numbers are implicitly reactive in Celluloid.

2.6.2 Time literals

Time literals describe a unit of time. Time objects are implemented as numbers in milliseconds and are described by the following lexical definitions:

```
time        ::=  day
day         ::=  floatnumber "d" hour | hour
hour        ::=  floatnumber "h" minute | minute
minute      ::=  floatnumber "m" second | second
second      ::=  floatnumber "s" milisecond | milisecond
milisecond  ::=  floatnumber "ms" | emptystring
```

Units of time longer or shorter than those specified above are not supported. All time literals are stored in miliseconds. Therefore, the largest time possible is 3.402823466e+38ms.

2.6.4 Boolean literals

Booleans are primitive types that evaluate to either one or zero. A boolean in Celluloid is represented by a single unsigned integer in Java with the following grammar:

```
boolean  ::=  true | false
```

2.6.4 String literals

Strings are sequences of characters. There is no character type in Celluloid. Individual characters are considered strings of length one and represent at least 8 bits. String literals are described by the following lexical definitions:

```
string    ::=   [stringchar]
strinchar ::=   <any source character except "\"> | escapeseq
escapeseq ::=   "\" <any ASCII character>
```

String literals can be enclosed in matching double quotes ("). Strings can consist of ASCII characters only. Unicode characters are not supported. Conversion of an appropriate string to a number is handled by the built-in function stringToNumber(). Conversion of a number type to a string type is implicit under the equality operator. Strings are implicitly reactive in Celluloid.

Escape Sequence	Meaning
\newline	Ignored
\\	Backslash (\)
\"	Double quote (")
\n	ASCII Linefeed (LF)

Once recognized as such by the parser, Strings will be passed to and handled solely by Java.

2.7 Operators

The following tokens are operators:

=	+	-
/	*	%
>=	<=	
<	>	and
or		

2.8 Delimiters

The following tokens server as delimiters in the grammar:

+=	-=	/=
*=	=	local+=
local-=	local\=	local
local=	(	)
{	}	do
end		

The period can also occur in floating-point literals. The first half of the list (up through local), the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following ASCII characters have special meanings as part of other tokens or are otherwise significant to the lexical analyzer:

#	\	"

3. Language model

3.1 Primitive types

Celluloid is a strongly typed language where an object's type determines the operations that can be performed on it. The core of the language can be broken down into four principal types: Timelines, Devices, Constraints, and Events. Additional types are available for numbers, booleans, strings, and times (for single units of time).

Each of the four principal types in Celluloid are distinct and are consequently immutable. Programs in Celluloid are built up by the interaction of these principal types. For example, devices are held to certain constraints, such as being considered only an audio input, and are played out on a timeline. Events that occur along a timeline may trigger the execution of additional user defined code.

3.2 The standard type hierarchy

The following is an enumeration of the types that are built into Celluloid. Future versions of Celluloid may have additional types in this type hierarchy.

3.2.1 Literals

Number, string, boolean and time types are all literals and evaluate to their respective underlying representation in the target language Java.

3.2.2 Functions

Celluloid allows user defined functions in source programs defined by the grammar:

```
function_block     ::=  “function” [functionname] [function_arguments]
                        [function_block] “end”
functionname       ::=  identifier
function_arguments ::=  “(“ [expression_list] “)”
functionblock      ::=  [code_section]
```

These user defined function have a void return type by definition, but all parameters are passed by reference. Side effects on these parameters are therefore implicit, but may be avoided by prefixing the parameter name in the function_arguments with the local keyword.

3.2.3 Predicates

Celluloid does allow for a special class of functions called predicates that can return true or false defined by the grammar:

```
predicate_block     ::=  “predicate” [predicatename] [predicate_arguments]
                         [predicate_block] “end”
predicatename       ::=  identifier
predicate_arguments ::=  “(“ [expression_list] “)”
predicate_block     ::=  [returns_section]
```

Predicates are necessary for conditional statements such as *if … else*, *when … do* , and *unless … do* blocks. As with functions, all parameters are passed by reference with implicit side effects.

3.2.4 Timeline

The timeline type describes the object in which a Celluloid media stream can be played. A timeline object takes enforces a set of constraint functions on a set of inputs based on the progression of time and events.

3.2.5 Device

Devices encapsulate the types of hardware (though a file is a considered a device) that a developer can retrieve input from or send outputs to. Devices are declared to accept constraints that restrict the operations that may be performed on them. All devices have a corresponding stream. The only devices implemented included in core are AudioFile, VideoFile, and Monitor abstractions.

3.2.6 Constraint

Constraints are an important abstraction in Celluloid. They define the actions that can be taken and properties that can be accessed on a device. The basic built-in constraints are input and output. Defining a device as an input or output puts clear limitations of the kind of actions that can done with the stream connected by that device.

Constraints can also inherit from other constraints in a multiple inheritance hierarchy. Certain additional built-in constraints, such as audio and video, build on the input or output constraints in this fashion. Built-in constraints (see Section 6.2 Built-in Constraints) can also be placed on timelines. Constraints also announce events (described next), for example the constraint play announces the event started.

3.2.7 Events

Events describe state changes of a media stream along a timeline. The previous discussion on constraints presented the example of the event started. Celluloid has built-in events that correspond to each of the built-in constraints on inputs and timelines. Additional user-defined events can be announced from user-defined constraints. For example, consider an event that is announced when a certain

frequency from some device is reached or similarly when a certain red saturation is reached on the red channel of a RGB device input. Events can also be empty (i.e. void). An example of this is a theoretical event that would be announced when a constraint is placed on a device to be an input.

3.3 Mapping hardware with config.io

Celluloid programs are designed to be portable with regards to hardware, allowing a Celluloid program to generate the identical output on separate computers with different hardware configurations. In order to accomplish this goal, Celluloid must be able to dynamically map hardware inputs and outputs to I/O variables in the code. Yet Celluloid must also statically map devices used to hardware ports (i.e. USB3), displays (i.e. HDMI1), and files (i.e. ~/myfile.mov), if it is to achieve desired functionality. To achieve portability, this static tying occurs outside the main Celluloid code, in the file *config.io*.

The user must enumerate the ties of I/O hardware to device keys within *config.io*; these ties cannot be made from within Celluloid *.cll* files. A "device key" is a human-readable and user-defined name for a particular input or output hardware. For example, a midi input that is plugged into the first USB port might be given a device key "myMidiInput".

Abstracting these ties into a single file allows a user to prepare their code to run a different machine with minimal effort, and more importantly without having to perform tedious and dangerous search and replace commands.

Future portability note: In future releases, this abstraction will enable the user to ship compiled code (rather than source code) coupled with a modified *config.io* file

to a new machine and have the compiled code run as expected without modification.

3.3.1 Config.io File Structure

The line structure within *config.io* is relatively simple: there is no distinction between logical and physical lines, and both are terminated by a NEWLINE. Each logical line consists of three string identifiers separated by whitespace. No whitespace is allowed prior to or subsequent to the identifier list. The syntax for defining a device key within *config.io* is as follows:

```
Key DeviceType HardwareLocation
```

where Key is the user defined device key which must conform to type identifier, DeviceType is the type of the device being labeled (see section 7.3 Device Definitions for more on Device Types), and HardwareLocation is the physical location of the input.

4. Execution model

4.1. Reactive programming

Celluloid implements the reactive programming paradigm. That is, assignment and constraint operations will propagate change throughout the environment to other assignment and constraint operations. For example, consider the simple statement *a* = *b* + *c*, where *a*, *b*, and *c* are numbers. In a reactive programming environment, the variable *a* will be automatically updated when if the value of *b* or *c* change.

4.1.1 Local operations

All assignment operations in Celluloid are implicitly reactive in nature. It is possible however, to force an operation to be non-reactive by prefixing the operator with the local keyword. For example, consider the simple statement *a local*= *b* + *c*, where *a*, *b*, and *c* are numbers. By using the *local*= operator as opposed to the = operator, the value of *a* will not be updated if the values of *b* or *c* change. The variable *a* can still be used reactively, such as *d* = *a* will always have *d* update to the current value of *a*.

4.2 Names, blocks, and scope

Names refer to instances of primitive types that can be instantiated including literals, timelines, and devices. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of a Celluloid program that is executed as a unit. Blocks are defined by functions, as well as code following *if, in*, *when*, and *every* keywords.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

4.3 Language specific blocks

Celluloid is not designed to be a general purpose language, however the lower level aspects of input and output logic require some of the capabilities offered by a general purpose language such as Java or C++. Consequently Celluloid allows for Java code to be included in source text under the following grammar:

```
language_block  ::=  “in” [language] “do” [language_code] “end”
language        ::=  java
language_code   ::=  /(w*s+)*[end_of_line]/
```

These language specific blocks will be copied to the output program verbatim in the place that they appear. It is the responsibility of the developer to ensure that the code in these blocks conforms to the syntax and semantics of the target language.

5. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. Celluloid offers four different types of compound statements: *if*, *in*, *every* and *when*.

5.1 The if statement

The *if* statement is used for conditional execution:

```
if_stmt  ::=  "if" predicate_list "do"
              expression
              ( "else if" predicate_list
              expression )*
              ( "else"
              expression )?
              "end"
```

Celluloid selects exactly one of the suites by evaluating the expressions one by one until one is found to be true. That suite is then executed, and no other part of that *if* statement is executed or evaluated. If all expressions are false, the suite of the *else* clause, if present, is executed. The *if* statement can be executed either outside an *in* statement, or inside an *every* or *when* statement; these parent *every* and *when* statements, however, may be executed directly within an *in* statement.

5.2 The in statement

Sequencing in Celluloid is performed in blocks within timelines and outputs using the *in* keyword. The *in* statement is used to delineate a block of code that will act upon a timeline:

```
in_stmt  ::=  "in" timeline "do"
              expression
              "end"
```

The *every* and *when* statements are subsidiary to the *in* statement: they cannot exist outside an *in* statement.

Programmers Note: *in* statements, unlike other compound statements, cannot be nested.

5.3 The every statement

The *every* statement is used for repeated execution over a time:

```
every_stmt  ::=  "every" time "do"
                 expression
                 ( (when | "unless") ( predicate_list | event ) )?
                 "end"
```

This repeatedly executes on the interval time until the unless line is tripped; if there is no *when* or *unless* line, the statement will repeatedly execute asynchronously until the timeline is finished. The *when* or *unless* line is tripped only when the predicate_list evaluates to true or the event is triggered.

5.4 The when statement

The when statement is used to execute expressions upon the announcement of a particular event or the satisfaction of a predicate list:

```
when_stmt  ::=  (when_decl | unless_decl) “do”
                expression
                "end"
when_decl  ::= "when" ( timeline | input ) (predicate_list | event)
            |  "when" (predicate_list | event)
unless_decl  ::= "unless" ( timeline | input ) (predicate_list | event)
            |    "unless" (predicate_list | event)
```

A *when* statement is not executed as it is parsed, but rather the expression is executed when the event the statement is listening for is announced. The *when* statement is analogous to a Java ActionListener. Note that if no timeline or input is specified in the *when* statement, the statement is assumed to be listening on the timeline specified in the closest *in* statement which encapsulates the *when* statement.

6. Timelines and constraint semantics

6.1 Timelines

A timeline enforces a set of constraints on a set of inputs and reacts to events based upon how the developer wants execution to proceed. Within a timeline a developer can utilize three types of statements: constraint statements, when based event handlers, and time based loops. This hierarchy allows for a rich set of operations that can be composed is a variety of ways.

6.1.1 Delayed execution

Timelines are not executed automatically when they are created. The developer must specifically play a timeline on an output. Thus, a user can create many different timelines that can be dynamically added based upon some event. A consequence of this behavior is that there is no requirement for a "main" timeline, in the sense that Java requires a main method to know where to begin execution.

6.2 Constraint Semantics

```
constraint_stmt      ::= constraint_function_id playable_id arguments
playable_id          ::= device_id | timeline_id
constraint_member_id ::= identifier
device_id            ::= identifier | emptystring
timeline_id          ::= identifier
arguments            ::= expression_list
```

Constraint functions are declarative statements that specify a constraint (constraint_member_id), an input (device_id), and possibly some arguments depending on the constraint. However, if the device_id is not specified on the constraint statement, then it is derived from the context of the surrounding *in*

block. Additionally, the type and number of arguments specifically depends on the specification of the constraint function.

6.3 Built-in constraints

A core set of built-in constraint functions can be used to manipulate the input devices and timelines in Celluloid. In order to use the built-in constraint functions, the device being operated on must "require" the correct constraint (in this case input), as discussed in Section 7.3 Device Definitions. These constraint functions are essential for almost all media manipulation, which is why they are built-in. They can be applied to both timelines as well as any device requiring the input constraint. Users wanting to add additional constraint functions must first define a new constraint, as described in section 7.2. The core set of built-in constraint functions is enumerated below.

6.3.1 play

The *play* constraint function is used to play the specified *playable_id* on the timeline of the surrounding in-block.

```
play_constraint_stmt ::=  "play" playable_id time_to_play
time_to_play         ::=  time_literal
```

Programmer's Note: It is often tedious to be forced to write a play statement for each playable_id being consumed in a timeline. Especially because many of these may have been declared previously in the current *in* block. For this reason, Celluloid has a special keyword, all, which when called:

```
play all @start
```

will play all timelines or inputs that have been declared above that line in the scope of the current *in* block.

6.3.2 pause

The *pause* constraint function is used to pause playback of the specified *playable_id*. When *pause* is used, and the *play* constraint function is called again, playback resumes at the same point it was paused at.

```
pause_constraint_stmt ::=  “pause” playable_id time_to_pause
time_to_pause         ::=  time_literal
```

6.3.3 stop

The *stop* constraint function is used to stop playback and reset the *device_id*. When *play* is called again, playback will resumes at the start, not the point in which it was stopped.

```
stop_constraint_stmt ::=  “stop” playable_id time_to_stop
time_to_playing      ::=  time_literal
```

6.3.4 ffwd

The *ffwd* constraint function is used to move forward on the input at different rate faster than normal. It requires two arguments, the time to start fast forwarding, and the speed, as a multiplier, that should be used.

```
ffwd_constraint_stmt ::=  “ffwd” playable_id time_to_ffwd “,” speed
time_to_ffwd         ::=  time_literal
speed                ::=  number
```

6.3.5 rewind

The *rewind* constraint function is used to move backward on the input at different rate slower than normal. It requires two arguments, the time to start rewinding, and the speed, as a multiplier, that should be used.

```
rewind_constraint_stmt  ::=  “rewind” playable_id time_to_rewind “,” speed
time_to_start_rewinding ::=  time_literal
speed                   ::= number
```

6.3.6 seek

Seek is responsible for jumping to a specific point in the input or timeline. It requires two arguments, time to perform the seek at, and the time in which to move to.

```
seek_constraint_stmt ::= “seek” playable_id time_to_seek time_to_seek_to
time_to_seek         ::= time_literal
time_to_seek_to      ::= time_literal
```

6.3.7 size

This constraint is responsible for constraining a video to a specific section of an output screen. Since a video can exist in multiple timelines, it is necessary to specify its existence in the context of a timeline. This constraint takes one argument, the rectangular region in which to constrain the video. Unlike the other constraint functions just listed, this function is inherited from the Video constraint.

```
size_constraint_stmt ::= “size” playable_id x1 “,” y1 “,” x2 “,” y2
x1 | y1 | x2 | y2    ::= number
```

7. User defined types

Celluloid has semantics for extending the existing timeline, device, constraint, and event capabilities. Extending an existing capability involves defining a new type which builds upon or requires preexisting or user defined types. Code which expands upon existing capability must reside at the top of the source file, leading to the following source file structure:

```
Event definitions
Constraint definitions with no requires
Constraint definitions with requires
Device definitions
Source code
```

There is no whitespace convention to separate these different definition types within a file.

7.1 Event definitions

An event definition defines an event object (see section The standard type hierarchy, Section 3.2):

```
eventdef  ::=   "event" eventname "end"
eventname ::=   identifier
```

An event represents a decoupled change of state in a Celluloid program. An event definition is a declarative statement.

7.2 Constraint definitions

A constraint definition defines a constraint object (see section The standard type hierarchy, Section 3.2):

```
constraintdef   ::= "constraint" constraintname
                    “requires" [constraint_list]
                    "announces" [event_list]
                    [constraint_body]
                    "end"
constraint_list ::= [identifier_list]
event_list      ::= [identifier_list]
constraint_body ::= [code_section] [announcements]
announcements   ::= [announcement] [announcements] | ε
announcement    ::= "announce" [eventname] "when" [functionname]
                    [arguments]
arguments       ::= [expression_list]
constraintname  ::= identifier
```

A constraint definition is a declarative statement. It first evaluates the constraint list, if present, then the definitions of these constraints are added to the current constraint definition. The event list is evaluated next, if it is present; events declared in the event list may be announced by the constraint. The body of the constraint is then evaluated into a constraint object. The body of the constraint may contain both constraint functions (as in Section 6.2, Constraint Semantics) as well as constraint properties, which may be accessed through a simple dot syntax on devices that implement that particular constraint.

Programmer’s note: The relationship between constraints and devices is similar to that between classes and interfaces in other modern programming languages, but with one exception: when a constraint specifies an announcement, that function name is decorated by the announcement when that constraint is used by a device.

7.3 Device definitions

A device definition defines a device object (see section The standard type hierarchy, Section 3.2) using the grammar:

```
devicedef       ::=  "device" devicetype "accepts" [constraint_list]
                     [device_body] "end"
constraint_list ::=  [identifier_list]
devicebody      ::=  [code_section]
devicetype      ::=  identifier
```

A device definition is a declarative statement. It first evaluates the constraint list, if present. The body of the device statement is then evaluated to ensure that it satisfies all of the requirements of the constraint list. If it does not, a compiler exception is thrown. A device definition satisfies the constraint list if it implements all of the methods that are declared in each of the constraint definitions.