CAGE and CARL

Seth Marinello
Dave Parent
Senior Project

November 30, 2007

1 Description of the Project

Our project is divided into two separate components. The first is referred to as CAGE (for Compact Adventure Game Engine) and the second CARL (for Compact Action/Reaction Language). Our purpose was to create a simple development platform for traditional first person adventure games that did not require writing complex code. CAGE is the application which displays the content in the game and CARL is the language used to script events. CARL provides a simple way to script events in CAGE, hiding the complexity of the underlying Java implementation. In the following sections we will lay out the specific goals of the project, as well as implementation details for both components.

1.1 Goals

We formulated a series of goals in the initial project proposal and expanded upon these over the course of the project. The process of developing the program and creating the demo productions lead to a better understanding of what features should be in CAGE. This is a summary of the goals for CAGE and CARL, created from the informal goals outlined in the project proposal and team discussion over the course of development.

- Ease of Use - The program should be easy to setup, develop for, and distribute
- Platform Independent - The program should be easy to port to different environments and behavior should be consistent across platforms.
- Low Entry Cost - No special tools should be required to use the program.
- Small Size - The API should be small and easy to understand and the released binary should be small to ease distribution over the internet.
- Flexible - While ease of use will be favored over extensive features, the program should be capable of varied uses. For example, the language must not be tied to a single game design.

These goals are suited to our target audience, which is users with computer skills and motivation, but without knowledge of advanced programming techniques. This is a program for our high school selves, one that would make the time between downloading the engine and getting the idea on the screen as short as possible. We believe that software is generally too complex, and while a game engine may not be the best place to start in improving computer usability, we have tried to keep the first time user in mind while preparing our release version. CAGE should be something anyone can download and get something running in a weekend. We do not expect amazing commercial products to be made in CAGE, but hopefully some fun spring break projects are made possible by our efforts. This has been our overall vision for the product's use, and we think the above goals, as well as the final product, reflect this aspiration.

2 Cage

2.1 Design

In this section we will give an overview of the design of CAGE and its various packages. This should serve as an explanation of our high level design and provide all information required to understand CAGE within the context of this document. For more detail on the actual implementation we refer you to the provided `JavaDocs` as well as the CAGE manual.

The original design for CAGE called for a number of domain specific "engine" classes which would control whole aspects of the functionality provided by the overall product. Our package layout is built around these specific engines. The final code does not contain as strict a separation of packages as originally intended, the reasons for this are discussed in the individual sections below.

2.1.1 Audio

The `audio` package was originally intended to have its own engine class controlling the playback of all sound in the program and providing synchronization timers as well as global volume control. This was not really in line with the role we had always been envisioning for audio within the CARL scripts. From the beginning we wanted the user to be able to script "playSound soundName" and have the rest handled by CAGE. If the interface did not support advanced sound manipulation, there was no reason to implement that functionality within the `audio` package. It now contains one class, `Sound`, which wraps the underlying `JLayer` library used for `MP3` playback. Sound provides a simple public interface which allows for playing, stopping and looping of playback, all of which are controllable from CARL scripts.

2.1.2 Carl

Most of this package consists of automatically generated classes from `antlr`. A lengthy discussion of its contents can be found in the second part of this paper. The parts directly

related to the construction of CAGE are covered in **Interfacing CARL with CAGE**.

2.1.3 Core

The `core` package contains the classes most fundamental to the operation of CAGE. The unifying theme between all classes in `core` is all other packages depend on them to function. `Cage` is the driver for the program. It sets up the program for operation and ties all of the various engines together. `XMLEngine` is part of the core because `Cage` depends on it to load all information about a production and resources. `Logger` resides here because `Cage` sets up the initial `Logger` which all classes will output through.

2.1.4 Graphics

The `graphics` package is the largest part of the CAGE codebase. It contains all classes related to the graphical display of content in a production. Elements of the production which can be displayed are `VisualItems`, including `Image` and `Animation`. A `Sprite` contains a visual item as well as a metadata like location and visibility. `Sprite` objects form the core of the graphical display in CAGE. A `Sprite` can have an associated action to be called when clicked as it implements the `Clickable` interface. A `Sprite` may contain several `HotSpots` if different actions are to be associated with different parts of the `Sprite`. A `Scene` is a collection of `Sprites` and can be added to the `GraphicsEngine` to be displayed. In addition to displaying `Scene` objects the `GraphicsEngine` controls the `Display`, which is a layered window where graphics are rendered.

2.1.5 Input

The `input` package contains only a few classes. `Binding` is essentially a structure associating an action to call with a key press event. The `KeyboardHandler` and `MouseHandler` process input events by executing the proper action via a binding or a collision with a `Clickable` respectively.

2.1.6 UI

In the `ui` package we have classes to support `Overlay` objects. An `Overlay` is a collection of Swing components created by `GUIFactory` that can be associated with various `Scenes`. See the sections **Displaying Over the Canvas** and **Controlling Swing from CARL** for more information on the workings of this package.

2.2 Challenges and Solutions

Cage presented numerous design challenges throughout the development process. Here we have recounted the most significant of those challenges as well as the solutions we used in the final product. These specific examples give a good look at the design of the most important parts of CAGE. We have tried ti incorporate the lessons we learned where appropriate.

2.2.1 Unifying Resources

An early challenge we encountered was providing a way for CARL to treat all aspects of CAGE as if they were part of the CARL environment. A user would place an image on the screen and expect to be able to manipulate it from CARL without having to declare it both in the Scene XML and in a CARL script. To solve this issue we require a "name" attribute on virtually all parts of a production. All resources have a name as well as a file source and are stored by name in `HashMap's` to provided acceptable lookup time. This way a `Sprite` can refer to an image via the same string as a CARL script.

By referring to all resources within the scenes and scripts, we get the added benefit of data abstraction. It is now a simple matter to change a single image, which may be used by many `Sprites`, to a different image file. As long as the resource does not change its name, no alterations need to be made to the scenes or scripts.

2.2.2 Interfacing CARL with CAGE

Our design has always contained limited access to CAGE functions from CARL scripts. Internal logic within CAGE would be wrapped by simple functions acting as built-in API functions in CARL, giving the user the ability to control much of the public interface of classes like `Sprite` and `Sound` without having to write Java code. Registering these Java functions with the CARL interpreter was a challenge we faced when connecting the two parts of this project.

Our initial solution created the class `Builtin`. This class extends `Closure` and defines the translation between the CARL function call to the proper methods in CAGE, including parameters passed to the interpreter from the script and the return type.

The first version of `Builtin` contained a set of enums which defined each method which could be invoked from CARL. When a `Builtin` object was created, we would assign an enum value to that instance. Methods in `Builtin` contained switch statements which would call the proper methods based on the assigned enum, as well as setup the right parameter and return types during construction. This method worked well for our initial merger of the two components, but an issue came up as the project moved forward.

The problem with the original `Builtin` design was scalability. As more and more features were added to CAGE, ways to control the new abilities needed to be added to CARL. This required continually expanding the number of enums and the length of the switch statement. Not only was this tiresome, but the code was harder to read. As the interface between CAGE and CARL is one of the parts most likely to experience heavy expansion in the future, we needed a cleaner solution.

The solution we came up with uses reflection to create the `Builtins` whenever CAGE is launched. A class called `Mediator` was created which contained only methods to be invoked by CARL users. Using the Java reflection system we are able to get an array of the methods in `Mediator` and extract the information required to create a closure for each. A modified version of `Builtin` is constructed from this information and placed in the global map of functions CARL scripts can invoke. By storing the `Method` object returned by the reflection

in the `Builtin`, we can invoke the proper CAGE method without having to repeatedly do lookups using reflection. In some simple tests showing and hiding `Sprites` we did not see a major performance hit moving to this new system. Now providing access to CAGE methods is simple: adding a method to `Mediator` will make it invokable from CARL by name. This also solves most of the scalability issue as the methods can be easily documented and self contained. If a large amount of functionality were added, another class like `Mediator` could be easily added in the future; the required modifications would be minimal.

2.2.3 Displaying Over the Canvas

Our requirements for CAGE make it a strange beast. The original intent was to create an adventure game engine, but that morphed into a more general purpose 2d multi-media platform in the final specification. On one hand, CAGE is very much a 2d sprite based game engine, but we did not want to limit it to simply games. This meant that the traditional bitmapped text of old game engines would not cover all possible uses. A user wanting to make a presentation would want to define text areas which automatically line wrap and provide scrolling in addition to simply placing some characters on screen.

The solution to this problem implemented in the final version of CAGE is to use the built in Java Swing components and display them over the 2d canvas of our engine. This is accomplished by using a `JLayeredPane` with the canvas at the very lowest layer. In theory this solution is very elegant; Swing is already provided with Java, providing a full range of text control for essentially no cost. Reality, of course, did not align with theory.

The first issue to overcome was the mixing of heavyweight and lightweight Java GUI components. Without going into the boring specifics, trying to place a Swing `JTextArea` over a AWT `Canvas` results in the `JTextArea` being covered, even if it should be the other way around. The solution to this is to wrap the `JTextArea` in an AWT `Panel`. This puts the lightweight Swing component on a heavyweight surface that can "overpower" the `Canvas`. Of course, this requires creating the `Panel` with the right size and location, and generally increases the complexity of GUI overlay.

2.2.4 Controlling Swing from CARL

Once we had the GUI elements displaying properly, they need to be put to use. As explained in **Unifying Resources**, CARL works with CAGE by referring to various components by name. Unfortunately, Swing components don't have a built in name, and even if they did, all our GUI pieces are wrapped in generic AWT `Panel` objects.

To solve this problem and allow CARL to control the GUI components we created `NamedPanel`. This class provides a way to name the `Panel` wrapping our Swing components, as well as provide some logic to help controlling location, etc.

We now encountered another problem, `NamedPanel` could be wrapping any number of different Swing classes, and we have no way of knowing which from CARL. The one thing we do know is that we want to be able to set the text displayed on the Swing component regardless of type, since GUI components without text could have just been a CAGE sprite.

Somehow, there is no interface indicating if a Swing component has `setText(String)` or `getText()`, even though most do implement these methods. We created an interface with these methods named `TextContainer` and created dummy classes which extend the Swing components we want to display in CAGE. These dummy classes simply call their parents `setText(String)` and `getText()` methods and implement `TextContainer`. A `NamedPanel` only accepts a `TextContainer`, so CARL is assured that it will be able to control the text and location of any GUI elements without having to check types on its end. This simplifies the code for the user working on a CARL script, helping us reach our ease of use goal.

2.2.5 Scaling Productions

When we first sat down to defined what our graphics engine would be capable of doing, it occurred to us that allowing the presentation to be scaled to a different resolution could be useful, especially if someone were to later want to use CAGE as an applet. In initial prototypes, we came up with the scheme of rendering the game not directly to the window but first an image and then scale that using one of the graphics libraries we were building the engine upon. This worked very well and was a simple addition to the design with a big payoff, so it was added to our list of features.

At the end of summer we began making a test program to remind us where we had left off and found the program consuming a huge amount of CPU time. While the new test scene had many more sprites than our previous prototype code, it did not make sense for rendering an essentially static image to take as much CPU time as Mozilla Firefox. Upon investigation, it was found that the the image scaling, done every frame, was very expensive. Rendering directly to the screen without scaling returned us to a sane amount of CPU usage, but we could no longer scale productions.

In order to keep the ability to scale, but avoid constantly performing a costly operation, CAGE now scales the entire production when first loading. This increases load times for a production, but only when it is not displayed at its intended resolution. Users who do not scale the production will experience no performance hit with this new design and those who do will not experience slowdown once the production is done loading. This gives us the feature we wanted at a performance cost we can afford.

Where this final design cost us really was in development time. Originally we scaled the images and their locations. Then we realized that Overlays needed scaling, so we scaled those. HotSpots on sprites need to be scaled. The list goes on and on. Most of our final show-stopper bugs have been related to this feature, one that was not in our original requirements, and many users will not use. If we had known the cost it would have there is no way it would have been added to our specification. Unfortunately we did not realize the complexity it would add to the graphics code in CAGE until we had already invested so much time that it was not worth turning back. From this we have learned to carefully plan and test under real world conditions any prototype features before adding them to the core specification. Even if a solution to a problem occurs, first evaluate the cost of that solution rather than charging in because you are excited to have "fixed" the issue.

3 Carl

CARL (Compact Action/Reaction Language) is the scripting language that allows the game/presentation designers to control what happens on screen when an end user interacts with the program. All scripts are loaded and parsed with a production when CAGE launches.

3.1 Design

CARL is an interpreted, functional language designed to be as easy as possible for anyone to pick up and use while still being capable of doing interesting and novel things in conjunction with CAGE.

CARL has a minimal amount of syntactical rules, and the statements and data types are as simple as possible. Although this limits the things one can do in CARL (you can't work with arrays or design your own complex data structures, for example), it reduces the time and effort required to master the language. We have tried to anticipate as many of the common, more complex tasks as possible and provided built-in functions to perform these tasks. More built-in functions can be created later by developers as their need is discovered, since the `Mediator` class is designed to be easily expandable.

CARL's complete grammar is in the appendix. The language and its features are documented in detail in the **Programming in CARL** document.

CARL is implemented mostly in ANTLR. This allowed us to focus more on the language design and less on lexing and parsing. Lexing, parsing, static type checking, and interpreting are all done at load time. Each function in the script becomes a `Closure`, which can be later executed by CAGE when the production runs.

3.1.1 The Grammar

CARL's grammar is most easily compared to C-like languages, but there are a few significant differences. The most obvious is the syntax for calling and declaring functions, especially regarding parameters.

In the declaration, we use the keyword `takes` to begin the list of parameters in a function, rather than simply surrounding them in parentheses, and the keyword `returns` followed by a type. These keywords make the function declaration read more like a description than a C function:

```
function foo
    takes
        a : int
        b : real
    returns string
```

"The `function foo takes` the parameters `a` of type `int` and `b` of type `real`, and `returns` a `string`." We felt that this was a nice balance between natural language, which can be

frustratingly verbose when programming, and a typical C-like language, which can be difficult for beginners to understand quickly. We later added the optional keyword `nothing`, which is used in place of a parameter list or a return type, and helps the declaration flow more naturally when there are no parameters and/or the function returns void.

When calling a function, there are no keywords, but it's arguments are separated only by whitespace and are not surrounded by parentheses. Again, this was to make calling a function seem more natural, like writing a short sentence. Many of our built in functions have active verb names, so one would often write things like

```
hide "tree";
```

This is most effective when the function is an active verb. Calling functions in this way also had the unfortunate side effect of making expressions difficult to parse when they contained a function call. We decided to use square brackets to denote a function call that returned with a value, much like Objective C (except in Objective C, they are required for all function calls).

3.1.2 Objects vs Names

Originally, we had planned to have CARL receive a collection of resource names from the parsed XML file. These names would be used like global constant variables, and would be statically type checked when the script was loaded. This would produce errors at load time if there were bad object names in the script, making it easier for the user to debug. We decided instead to simply reference objects by name with strings. This had a number of advantages.

First, there are less types for the user to worry about. Since the user can't create or manipulate an object from the script, it might be confusing to have these extra type names that can only be used to pass objects of that type to functions, both user defined and built in.

Second, and perhaps most importantly, it allows the user to do interesting things by manipulating the object names as strings. We did this in our TicTacToe example game; the names of the sprites for the X and O icons have prefixes that correspond to the player, and suffixes that correspond to the position of the icons on the board. We were then able to pass bits of names to various functions, eventually concatenate them and pass them to CAGE as valid names. It made the script significantly shorter than if we were forced to use each name without breaking it into pieces to be passed around.

This can, however, make debugging a script significantly more difficult since the CARL interpreter will pass any string to CAGE regardless of its validity.

3.1.3 Abstraction From The Engine

We tried to design CARL in conjunction with CAGE to be as simple as possible to do complex things by abstracting out all of the resource management and lengthy operations one would normally have to do to accomplish the same task in Java. Data is defined and

loaded from the XML files, and CARL has can tell CAGE to perform actions on that data by name. To play a sound in Java, for example, one would have to instantiate a sound object, and deal with threading and exception handling:

```
new Thread() {
    public void run() {
        try {
            do {
                if (player != null) {
                    player.close();
                }
                player = createPlayer(fileName);
                player.play();
                while (!player.isComplete()) {
                }
                player.close();
            } while (loop);
        }
        catch (Exception e) {
            // ...
        }
    }
}.start();
```

In CARL, this is abstracted down to one very obvious line:

```
playSound "MySound";
```

3.1.4 Dynamic vs Compiled

We had to choose whether CARL would be implemented as an dynamic language, or compiled to byte code. Making it a compiled language would offer performance benefits, but it would either require us to implement our own compiler (which is outside the scope of the project) or require the user to have third party tools and knowledge of their use. As a dynamic language, there is much more overhead (both in load time and run time), but no build environment is required. We decided to make it dynamic because the simplicity for the user is more important than being optimized for speed and memory in this case. Luckily, the resource usage of the finished product is quite reasonable on modern computers (considering the Java Runtime is fairly heavy, too), especially since CAGE is not intended for rendering complex graphics. There is still a lot of room for optimization in CARL, which could reduce the resource usage much further.

3.1.5 Implementing in Java vs C/C++

We chose Java for a number of reasons, but in CARL's case, we wanted the benefit of automatic garbage collection. Our target audience would have a very difficult time doing their own memory management, and implementing our own garbage collector for CARL would be a large project in itself.

3.2 Challenges and Solutions

3.2.1 Multiple Script Files

ANTLR only accepts one file, but we realized it was feasible for someone to write a very large script that would be difficult to work with in a single file. To solve this, we added a way for the interpreter to take the existing set of `Closure`s and add to it. This allows for multiple files, but there cannot be any duplicate function or global variable names across all the files in a production. The order in which each script is loaded is defined in the XML.

3.2.2 Nondeterministic Cycles in the Grammar

There is one place in the grammar where a cycle exists: expressions can consist of multiple expressions. The way the grammar was originally defined, this cycle was nondeterministic. To solve the problem, we require all operators with two operands to be surrounded by parentheses. Unfortunately, this adds an unnecessary syntactic rule that isn't entirely obvious. One would normally expect the following to work:

```
x = a + 1;
```

but because the `+` operator has two operands, CARL requires:

```
x = (a + 1);
```

It's good to be in the habit of using parentheses for complex arithmetic to avoid order of operations mistakes, but it shouldn't be forced. We were unable to come up with another solution without significantly redesigning parts of the language.

3.2.3 Error Handling

When testing any source code, it is good to have meaningful error messages and an indication of where the error occurred. CARL is parsed in ANTLR, and unfortunately ANTLR does not retain line numbers, so we cannot give the developer a precise location of the parse error. Instead, we keep track of the number of statements in each function, and when an error occurs, we give the name of the function it occurred in, the number of the statement, and a descriptive message. The developer must hunt down the exact location themselves, but hopefully the message gives them a good indication of where the error occurred. Here is some example code and its error message:

```
function foo
    takes
        x : int
    returns int
{
    x = (x * 3);
    return y;
}
```

```
 [java] cage.carl.InvalidException: Error in function: foo
 [java]     At statement: 2
 [java]     variable used without being declared: y
```

4 Future Development

While we feel our final release is a full product, there are features which would increase the utility of CAGE. This section is added on as either a reminder to our future selves where to go next, or as a starting point for a future developer wishing to expand our engine.

The first, and most important feature to add would be a fully implemented graphical editor. This could be done by extending the already existing edit mode in CAGE. This was not included in our first release because it was not planned into the original schedule. When we began to work on demos we realized how useful a graphical layout mode would be and created edit mode as a sort of prototype. Upon evaluation we found that doing it right would take a lot of effort. To implement it we would have to reduce the number of features we were adding to the engine, thus limiting the kinds of games that could be created with our product.

A new input library could be added to provide support for 2d action games. Our desire for a simple, single jar, pure Java solution limited us to the input system provided by AWT. This system is both limited in its control over input event ordering and handling of paired press/release events. At the time of writing there is not a pure Java, cross platform way to consistently detect held down keys using the AWT events because different platforms behave differently and are not abstracted by the JVM. Solving this issue was simply beyond the scope of the project as we set out to create a primarily mouse driven engine.

Dynamic loading and unloading of content. This would increase the scalability of the engine, as CAGE currently loads all the resources at launch. This makes scene changes quick, but also means CAGE will take up a lot of memory if a very large game were created. The resource engine provides an easy way to implement content loading/unloading with scene changes.

This is certainly not all the ways that CAGE could be expanded, but these are the major features we envision for version 2.0.

5 Appendix

5.1 Terminology

- **Production**: A production is a game, story, or presentation created to run in CAGE. While the primary intent is to be an adventure game engine, no restrictions are placed on what can be made with CAGE, and so we wanted a term that did not imply it was only for games.
- **Resource**: A resource is a media or image file that CAGE can use as part of a scene.
- **Scene**: A scene is a single screen in the production.
- **Sprite**: A sprite is an image or animation displayed in a scene. It is different from a image because it has a location and other properties stored with it, rather than just pixel information.

5.2 CARL Grammar

script -> **declaration*** (**action** | **function**)*
declaration -> ID : **type**
type -> `int` | `bool` | `point` | `real` | `string`
action -> `action` ID **block**
function -> `function` ID `takes` (`nothing` | **declaration**+)?
`returns` (`nothing` | **type**) **block**
block -> { (**declaration** | **statement**)* **}**
statement -> **if** | **while** | **command**
if -> `if` **expression block** (`elseif` **expression block**)* (`else` **block**)?
while -> `while` **expression block**
command -> (**call** | **assign** | **return**) ;
call -> ID (**expression**)*
assign -> ID = **expression**
return -> `return` (**expression**)?
expression -> (`!` | `-`)? **subexpression**
subexpression -> **item** | `(` **boolop** `)` | `[` **call** `]`
boolop -> **comparison** ((`&&` | `||`) **comparison**)*
comparison -> **addop** ((`==` | `!=` | `<` | `>` | `<=` | `>=`) **addop**)?
addop -> **multop** ((`+` | `-`) **multop**)*
multop -> **expression** ((`*` | `/`) **expression**)*
item -> ID | **number** | `true` | `false` | **point** | **string**
point -> < **expression** , **expression** >
string -> " .* "
comment -> # .*