Scripting 3D Human Animation

Student: **Reinhard Pointner**
Supervisor: **Bogdan Matuszewski**

A project report submitted in partial fulfillment of the Degree of BSc (Hons.) Computing.

April 2010

Abstract

The goal of this project was to create a scriptable 3D human model that can be deployed on the web. This requires the development of two major components: a scripting language specifically designed for human animation and a 3D visualization thereof.
The product of this software project is a proof-of-concept prototype that shows how the scriptable 3D visualization can be embedded in a website and how scripts can be executed dynamically form within this website using JavaScript.
The scripting language aims to provide an easy-to-read language for describing movements of a human body. The 3D visualization will then plays and render these movements. The project seeks for a sophisticated and elegant solution by relying heavily on existing technologies such as jMonkeyEngine for state-of-the-art 3D graphics and Scala as host language for the powerful new domain-specific language. The project succeeded in using these technologies to deliver comprehensive solutions to the various tasks that would otherwise not have been possible within the time constraints.
While the path that was chose for implementing the required components was undoubtedly a wise one. The actual results, while technically impressive, lack in terms of usability. The domain-specific language for example is extremely powerful yet the implementation itself is simple and advanced new syntactic elements can be added to the language easily.

Contents

1 Introduction

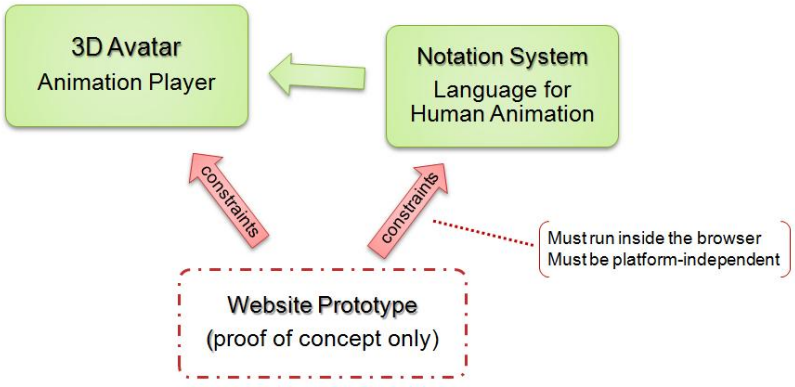
1.1 Background

There are many different sign languages used all around the world. This includes more complex sign languages used by deaf people, simplified sign languages that are sometimes used when working with people with mental disabilities but also military hand signals. There are also multiple notation systems which vary in simplicity and expressiveness.
Sign languages are visual by nature. A solely text-based description of a gesture will inherently be difficult to understand and to perform. Gestures that are described in natural language or using a specialized notation system are always accompanied by a set of pictures, an animation or a video illustrating how to perform the gesture. Human audiences will find it easy to comprehend and reason about visual material or descriptions in natural language, however doing the same on a computer would be an extremely difficult task. A standardized notation system on the other hand can easily be processed by a program but will be significantly more difficult to understand for human audiences in comparison to a simple animation.

1.2 Purpose

The general goal is to bridge the discrepancies between visual instructions designed for human audiences and text-based instructions that can be processed by computers. The project can be divided into two fairly separate tasks. One is to create a text-based language for describing human animation like arm or hand movements that can be processed easily by a computer while remaining simple enough so that new animations can be composed by humans. The second task is then to create a 3D application that can render a model of a human that can play animations given at runtime in the previously defined language.

The ultimate goal was to create a collaborative online sign language dictionary that would allow users to enter new words and gestures and thus participate in creating a universal database of gestures in a format that is both machine-readable and human-readable.
Due to the complexity of the problem and limited time available the main goals have been reduced to the 3D visualization and the language aspects while respecting the constraints imposed by the original idea which requires all software components to be deployed on the web.



1.3 Overview of Chapters

Chapter 2 describes and compares the different techniques and technologies that were considered to be used in this project. This includes libraries for 3D application development and possible approaches to creating a domain-specific language.

Chapter 3 describes what kind of transformations the new DSL is designed for and explains how the language is implemented using Scala language features. How this language can be interpreted dynamically using the Scala compiler from within an application and security considerations of doing so will be explained. Subsequently the approach of transforming an instruction tree into actual animation will be covered.

Chapter 4 describes the use of jME to deliver state-of-the-art 3D graphics. The main theme of the chapter is loading and programmatically animating models and what kind of problems were encountered when doing so.

Chapter 5 explains how the previously described components were deployed as an applet embedded in a website as well as additional techniques that were required to do so. This section also covers various problems that were encounter when deploying the application to the web.

Chapter 6 summaries and evaluates the techniques and technologies used in the various parts of the applications and explains in how far project goals were achieved and what could still be improved upon.

2 Analysis and Investigation

2.1 Domain-Specific Language for Human Animation

2.1.1 Language Design

2.1.1.1 Requirements

Before designing the language some basic requirements were determined. The language would have to be able to support commands for different types of movements. There has to be a way to synchronize these commands so that some actions can be performed in parallel while others will be performed sequentially. The language should also be general enough so that one can manipulate all moving elements of the body such as hands, arms, fingers, legs the also the head. One important aspect of the desired language is that it should be easily readable for average users and must therefore be based on natural language as far as possible while keeping within the constraints of an exact language that can be interpreted by a computer program.

2.1.1.2 Scripting Language for Embodied Agents

Z. Huang, et al. (2003) tackled a project similar to this one when implementing STEP (Scripting Language for Embodied Agents). Their scripting language is targeted at non-professional users and has very clear and easy to read syntax.

STEP works by controlling the joints or bones of the human body and allows all these joints to be turned towards a set of predefined directions such as “top” or “left_down”.

One interesting implementation detail of this approach is that it requires an interpolation between the original orientation of the bone and the targeted orientation of the bone. It has to be pointed out that rotation interpolation in 3D space is significantly more difficult than in 2D space. In STEP an algorithm called slerp (spherical interpolation) is used to solve this problem. By default interpolation is liner, but in order to achieve more natural looking movements easing can be achieved by stating interpolation points.

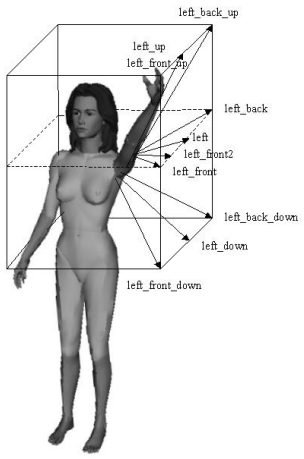


Figure 1: Direction Reference System in STEP, Z. Huang, et al. (2003)

STEP allows actions to be structured into sequential and parallel groups of commands. These groups may again contain actual instructions and groups again. A group can therefore be seen as a single complex instruction. This nesting of groups and commands in sequential and parallel groups allows all kinds of movements to be synchronized in a very simple and easily readable manner.

```
script(walk_pose(Agent), Action):-
     Action = seq([
     par([turn(Agent,r_shoulder,back_down2,fast),
           turn(Agent,r_hip,front_down2,fast),
           turn(Agent,l_shoulder,front_down2,fast),
           turn(Agent,l_hip,back_down2,fast)]),
     par([turn(Agent,l_shoulder,back_down2,fast),
           turn(Agent,l_hip,front_down2,fast),
           turn(Agent,r_shoulder,front_down2,fast),
           turn(Agent,r_hip,back_down2,fast)])
     ]).
```

Code Example 1: The walk animation in STEP, Z. Huang, et al. (2003)

As can be seen above STEP is a very simple yet powerful description language and fulfills all the predefined requirements. However there is still a big gap between natural language and this notation system which seems more like a programming language. Writing this kind of code for an inexperienced user might require an initial learning phase but reading and understanding should be straightforward from the beginning.

2.1.1.3 Sign Language Notation Systems

In addition to STEP the most common sign language notations where also investigated whether they represent a viable foundation for the new language. However these notation systems use lots of special symbols and have a steep learning curve. They look very cryptic and complex on first sight. The target audience for this application on the other hand does not only consist of sign language experts but average users that are not familiar with any of those notations. In addition these notation systems tackle the problem on a very high level and have predefined notations for complex hand shapes and movements. Implementing a system on that level would have been well outside of the scope of this project and the resulting system would not have been useful for scripting generic 3D animations.

2.1.2 Language Implementation

2.1.2.1 Requirements

Once there are some guidelines and design ideas for what this new domain-specific language (DSL) should look like, an approach to actually implement a parser or interpreter for this language has to be devised. The implementation doesn't only have to take the language design into account but also additional requirements imposed by the application itself while implementation itself has to be feasible within the time constraints.
Since the application should be able to execute syntactically correct instructions at runtime, the parser or interpreter must be able to perform its function within a running program. The new language must therefore be a so called external domain-specific language.
There are additional deployment considerations as well. Since the final application will be deployed on the web and run inside the browser, all components must support this kind of

environment. Naturally this means that the software must be platform-independent as well. While a working solution is important, flexibility, extensibility and maintainability have to be considered as well. It is difficult to design every aspect of the language beforehand because many problems or ideas for improvements will arise as a result of testing and working with the language. Therefore an implementation that can be grown easily to keep up with changing requirements may be vital for a practical solution.

2.1.2.2 Create a new language

The obvious solution is to implement a new language from scratch. However there is only limited time so implementing full-featured parser is not feasible. Even if a working solution can be achieved within the time constraints this solution would likely suffer from a severe lack of quality and as a result the language would be neither maintainable nor extensible.
However there are many well tested tools and APIs available that would simplify and speedup language development significantly up to the point where creating a new language can be considered a perfectly reasonable option. ANTLR (ANother Tool for Language Recognition) for example is a parser generator purely written in Java and can therefore easily be deployed in Java applet. The reason why ANTLR is preferred over any other Java-based tool because it is used for parsing in the Groovy scripting language so it is safe to assume that this tool would be also be capable of fulfilling the demands of this much simpler language.

2.1.2.3 JavaScript Object Notation

JavaScript is the first and only option to execute code directly within a webpage without resorting to browser plugins. Being dynamically-typed makes the language quite flexible in the way it is used and the syntax supports object literals as well as array literals so code that requires listing objects will be much less verbose than in languages such as Java. These features combined allow data structures to be fined as a tree of JavaScript objects in a quite concise matter. For this reason JSON (JavaScript Object Notation) has become a widely adopted way of serializing structured data in text-form.

```
{action:"sequence", children:[
    {action:"parallel", children:[
        {action:"rotate", bone:"leftLeg", degree:90, duration:1},
        {action:"rotate", bone:"rightLeg", degree:-90, duration:1}
    ]}
]}
```

Code Example 2: Description of an animation in JSON

The most notable advantage is that this script can be dynamically evaluated in any browser so there is no reason to actually create a new language. A notation similar to STEP (Z. Huang, et al., 2003) can be created in an extremely short amount of time using this approach but also shares a common disadvantage. The code is very different from a description in natural language.

2.1.2.4 Groovy DSL

Groovy is a dynamically typed scripting language similar to Java with lots of syntactic sugar which makes code concise and easily readable. It supports a way building a tree of objects in a similar manner as JavaScript does. These so called "Builders" are used throughout the Groovy standard libraries for creating markup or building user-interfaces. Groovy and JavaScript both support first-class functions meaning that functions are objects and thus can be used as parameters for method. In Groovy DSLs this feature is used quite often because the syntax for anonymous functions is very concise and looks just like a block of code. In JavaScript however creating a function requires the use of the "function" keyword which makes the code a lot more verbose. In Groovy it is also possible to end an expression simply with a newline so there is no need for semicolon ("`;`") unless multiple expressions are stated within the same line of code. Obviously Groovy cannot be used in the browser directly but since Groovy is a scripting language running within the Java-platform, the Groovy interpreter can easily be embedded into a Java applet.

2.1.2.5 Scala DSL

A statically compiled language like Scala may not look like a valid option at first glance. Even though it is a typed language, the code is extremely concise as types are almost never stated explicitly but automatically inferred by the compiler. Another aspect that might be considered is type-casting which can make Java code quite verbose. This problem is solved by so called implicit conversions. When a method is called on an object that doesn't have this method then the Scala compiler will look for any implicit conversions in the current scope that allows this object to converted to another object that does support this method.

```
// convert Int to String implicitly when necessary
implicit def int2String(i:Int) = ""+i
val b = 12345.startsWith(1)
// this expression is equivalent to "12345".startsWith("1")
// the return type of String.startsWith(String) is Boolean
// so the type of value b is Boolean as well
```

Code Example 3: Implicit conversion in Scala

The idea of a builder can easily be applied in Scala in a manner similar to Groovy. Some aspects of Scala syntax however trump Groovy syntax in terms of DSL friendliness. One of the design goals of the Scala language is that it should be possible to seemingly grow the language with libraries without actually changing the syntax. A so called internal DSL in Scala is merely a library that uses syntax-like features in a way so that the resulting code is very easy to read and to understand. First of all Scala doesn't require a dot ("`.`") when accessing a field or a methods of an object and since methods with no arguments don't require empty parentheses ("`()`") there is no difference in accessing a field or a method. Another syntactic feature that is extremely useful

for creating code that reads similar to natural language is the fact that the argument for a function that only takes one arguments can be stated after the function without any parentheses.

```
object scala {
    def is(attribute:String) = {} // empty function
}

scala is "interesting"
// syntactically valid expression equivalent to scala.is("interesting")
```

Code Example 4: Code similar to natural language in Scala

However it was questionable at first weather Scala is also a viable choice for creating an external DSL that allows the code to be interpreted dynamically at runtime after all the main application code is already compiled and deployed. Since Scala code compiles to Java byte code all Scala code is completely interoperable with Java and vice versa. The Scala runtime as well as Scala compiler depend solely on the Java-platform and therefore it is safe to assume that any given Scala code can be compiled and executed dynamically at runtime within a Java application. So after more in-depth research Scala was deemed to be an excellent choice for developing an external DSL.

2.2 Visualization of a Human 3D Model

2.2.1 Requirements

The 3D animation should aim for a natural and appealing look. Due to the video game industry users now expect stunning visuals from all kinds of 3D applications, so the visualization should live up to those standards as best as possible. It should be possible to display a textured and rigged model with a satisfying number of polygons as well as visually appealing elements such as reflections or shadows. The bones of a rigged model must be accessible so 3D transformations can be applied to the bones at runtime when an animation script is executed.
Using low-level libraries such as OpenGL directly can be ruled out immediately, but using higher-level 3D engines should allow for nice visuals and simplified application development. Along with the language component, the visualization component will be deployed on the web so the chosen technology must support this kind of environment.

2.2.2 3D rendering on the Web

2.2.2.1 jMonkeyEngine

jMonkeyEngine (jME) is the most prominent Java-based 3D game engine providing a rich set of features. As it is used as the foundation for computer games there is good support for working with all kinds of 3D models. Advanced visual effects like shadows or water simulations are built in and can be used in applications easily. There is a wide variety of tutorials and impressive

sample applications that are provided by both the developer team as well as the community. By the looks of those examples it is safe to assume that jME provides the required features and more. Due to the non-restrictive licensing, jME is also used in various commercial games. The project has an extremely active developer community so discussing problems in the forums and getting help on specific issues from more experienced users is easy. The libraries also supports common debugging functions such as drawing bounding boxes around 3D objects or drawing the bones of a rigged model. Most of the demo applications are already deployed via Java Web Start and there are a few test applets in the source tree that showcase that jME can also be used for Java Applets with ease.

2.2.2.2 Java3D

Java3D is an open-source project that provides a 3D scene graph API that can be used on top of OpenGL or DirectX to render 3D graphics. Like any Java library, Java3D can easily be deployed in a Java applet. It does support loading the most common 3D formats but more advanced features such as shadows are very difficult to implement. One major problem is that the community around Java3D is not very active so it might be hard to get help from developers or more experienced users. A few sample applications are provided by the Java3D project itself, but it is hard to find real-world applications based on Java3D that could proof that this library is the right choice for the task at hand. In fact Open Wonderland, which used to be one of the very few major Java3D projects, is now solely using jMonkeyEngine.

2.2.2.3 X3D

X3D is a web standard for representing 3D virtual environments. Rendering 3D content is however not implemented in any major browser and the X3D format itself is not widely used. Popular 3D modelling applications like Blender don't support this format natively. If a 3D model in a more popular format from the web is used in the application it has to be converted to X3D beforehand which could pose a problem. There is a Java applet called XJ3D which is used by Z. Huang, et al. (2003) in STEP to provide the X3D rendering and runtime engine. While visuals were not the main points of their research, their use of a very basic low-polygon model might suggest that it is difficult to get detailed 3D models in X3D format. X3D would be an interesting choice if it were natively supported by browsers, but since a Java applet needs to be deployed anyway, it doesn't hold much merit over using a real-world game engine like jME.

3 Scripting Language for Human Animation

3.1 Introduction

The scripting language for human animation is defined by the language syntax but also the interpretation and execution of instructions and how specific instructions are translated into specific transformations of the 3D model. Before any language constructs of the new scripting language can be defined, the semantics of what kind of instructions will be supported and what implications those may have on possible 3D transformations of the model as well as overall implementation of the language have to be determined.
Early on, the decision to make use of existing technologies was made so a full-featured language can be created within the limited amount of time. Scala was chosen as the foundation for the new scripting language because it allows a representation of tree structures in the code as well as some natural language like constructs.
Due to the lack of prior experience with the Scala programming language, Scala was only used for the definition of the language and for building up the data structure from a given textual input. These data structures are then processed in Java and corresponding code is executed resulting in an animation of the 3D model.

3.2 Transformation Model

3.2.1 Introduction to Forward Kinematics and Inverse Kinematics

There are two fundamentally different approaches for animating rigged 3D models. In 3D modelling application such as Blender, animation is achieved using so called inverse kinematics. Using inverse kinematics means, that an animation is defined by the translation or rotation of a body part (e.g. a hand) while all the transformations of connected joints (e.g. shoulder and elbow) that are required to perform the desired movement are inferred by the system automatically. For this to work each joint is associated with a set of constraints that allow the inference system to come up with physically possible and natural looking movements.
Using forward kinematics on the other hand simply means that transformations that are applied to a parent joints (e.g. shoulder) will apply to all child joints (e.g. arm and hand) as well. This kind of behavior is inherent in all modern scene graph APIs so it is much easier to implement than inverse kinematics. The ability to control every single joint directly also makes the animation system much more powerful and allows all possible (and impossible) movements.
There is however a steep price in terms of ease of use to be paid for this universal yet simple solution. In a system using inverse kinematics the movement of a hand is described solely by this very movement. In a system using forward kinematic however the same movement must be translated to a series of parallel joint transformations of the shoulder and elbow.

3.2.2 Forward Kinematics using Rotation and Tilt

For the final application it was decided to start with a simple approach which can later be extended if there is time. The basic idea is that there are two operations namely “rotation” and “tilt” both of which are parameterized with a degree or angle. Only these two transformation types can be applied to a joint, but how those are then mapped to actual 3D rotation transformations is not further specified. How rotation and tilt correspond best to actual 3D rotation can differ depending on the location and orientation of the bones in the model. It should be possible to directly map rotation and tilt to two of the tree Euler angles (yaw, pitch and roll). When doing so the expected behavior for each joint has to be determined. Rotating a foot by a certain number of degrees for example should have the same logical behavior for both feet meaning the foot should turn toward the body or away from the body. If the same 3D rotation is applied to both feet then both feet will turn toward the same side. The expected behavior however is defined by two movements that mirror each other, which means that different 3D rotations have to be applied.

3.2.3 Evaluation

Z. Huang, et al. (2003) use a similar approach but they allow a 3D transformation to be given in form of a target direction which may be named (e.g. “top”). This allows for all kinds of 3D rotations while the method implemented in this project only allows rotations by two of the three axes. It would also be easier to read and envision an animation using this kind of description in many cases.

The reason this approach was not chosen initially was because it would require interpolation between two 3D rotations which would add even more complexity. In retrospect this consideration has been flawed as this functionality is already provided by jME and would be immediately available to the application without effort. The chosen solution on the other hand requires a mapping between rotation and tilt and to corresponding 3D rotation of the bone. This mapping cannot be inferred programmatically but has to be adjusted manually for each bone making it very difficult to exchange the 3D model used in the visualization.

3.3 Language Implementation

3.3.1 Architecture

Designing a DSL requires more than just basic programming skills. The different syntactic constructs provided by Scala and combination thereof have to be understood and explored in order to be able to design an easily readable language within these syntactic limitations.
Since new DSL is designed within Scala it is defined using object-oriented concepts like classes, methods, properties and inheritance. Furthermore all language elements are defined in a singleton object and since Scala has support for importing the members of any given object into the current scope the DSL as a whole can imported with a single statement.

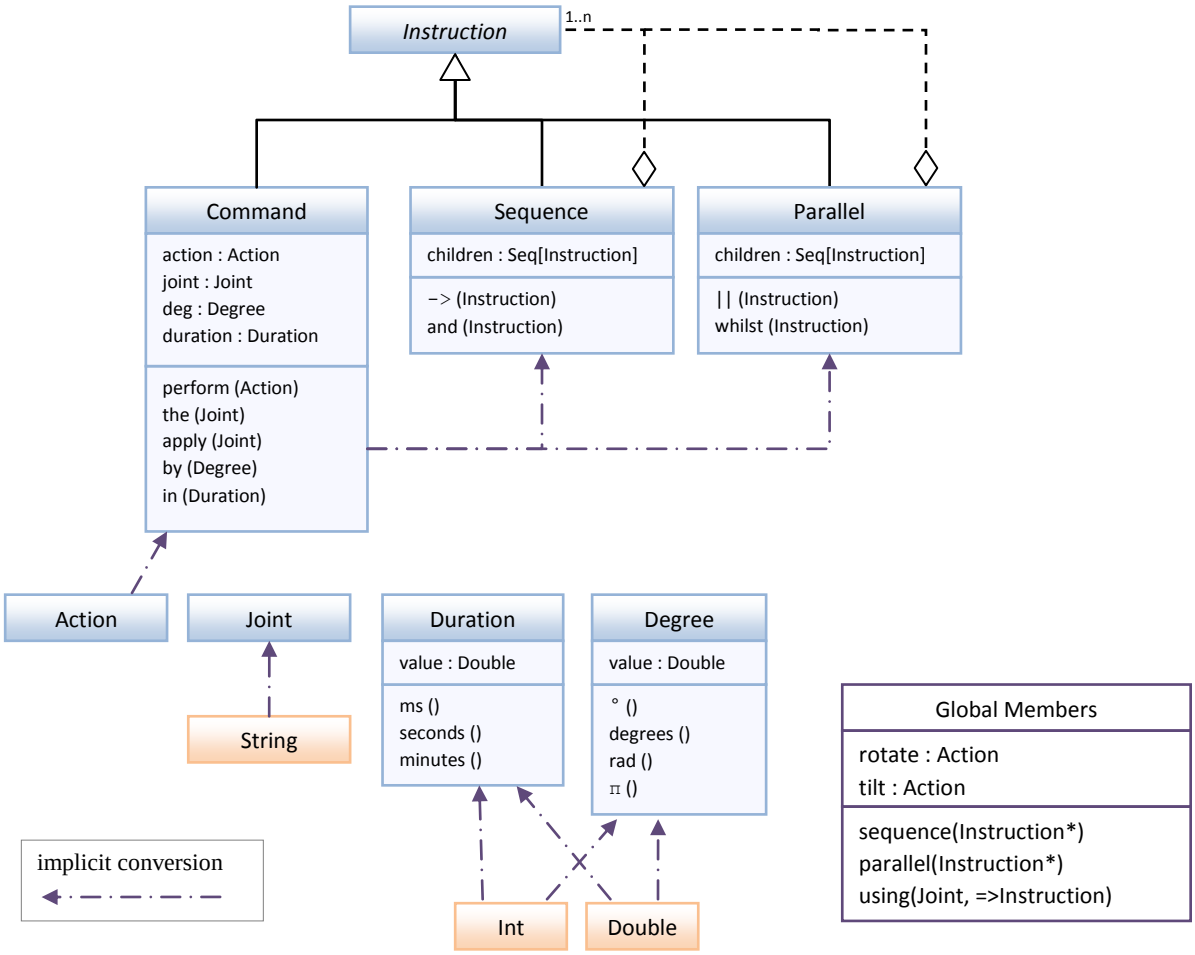


Figure 2: Architecture of the Scala DSL

An animation is represented by a tree of instructions. The leafs of this instruction tree are commands. Nodes are either a list of sequential or parallel instructions, which themselves may again represent a list of instructions. The language provides two additional factory methods that take a variable number of instruction arguments which allows for builder syntax. Thus code can be written in a tree-like form similar to the internal representation.

```
import avatar.dsl.AvatarLanguage._ // import language elements
import avatar.dsl.AvatarBody._ // import body data structure

sequence(
     parallel(
          rotate the (left arm) by (45°) in (2 seconds),
          tilt the (left arm) by (45°) in (2 seconds)
     ),
     rotate the head by (0.5 π) in (500 ms)
)
```

Code Example 5: Instruction builder

This example shows how more complex animations can be built using tree-like composition of elemental commands. Commands can be constructed simply by calling methods which is much less verbose in Scala than in many other languages.

3.3.2 Implicit Conversions

One of the most important aspects of the DSL are implicit conversions which essentially provide the glue logic between two different types that would otherwise not be interoperable. For example numbers are implicitly converted to Degree or Duration types as required and objects like “rotate” and “tilt” are actually of type Action and not Command. There are also some additional implicit conversions from and to existing Scala types that will allow more experienced users to make full use of Scala features like tuples or sequence comprehensions.

3.3.2.1 Number to Degree or Duration

This implicit makes numbers and Degree or Duration objects interchangeable. Units are defined as methods (e.g. “ms”) so if one of these method is called, a new object is created using the original value and the numeric conversion appropriate for the unit.

```
t:Duration = 5 minutes // t.value is 300
r:Degree = -0.5 π // r.value is -90
```

Code Example 6: Implicit Conversion: Number to Degree or Duration

3.3.2.2 Instruction to Sequence or Parallel

Sequences support concatenation of Instructions using methods such as “->”. Due to implicit conversion, all these methods can be used on plain Instruction objects as well.

```
(rotate(right arm) by 45) -> (tilt(left arm) by 45) // sequence
(rotate(right arm) by 45) || (tilt(left arm) by 45) // parallel
```

Code Example 7: Implicit Conversion: Instruction to Sequence or Parallel

3.3.2.3 Tuple to Command

Commands may be defined using the previously seen syntax but also via Scala tuple literals which might be considered more concise.

```
c:Command = (rotate, left arm, 45°, 500 ms) // Scala tuple notation
```

Code Example 8: Implicit Conversion: Tuple to Command

3.3.2.4 Scala Sequence to Instruction Sequence

The support for sequence comprehensions makes loops very useful for defining sequences of instructions.

```
s:Sequence = for (part <- "head" :: "rightArm") yield rotate(part) by 45
```

Code Example 9: Scala Sequence to Instruction Sequence

3.3.3 Enforcing Fully Parameterized Commands

One aspect of the current implementation might be considered a problem. All parameters of a command like the target joint or the degree are optional on the language level even if they are required for actually executing the command.
R. Ferreira (Type-safe Builder Pattern in Scala, 2008) suggests a solution to this problem which uses generics in an unorthodox way in order to make the compiler flag incomplete commands. This idea has been applied to a subset of the language as a proof-of-concept prototype.

```
private[StrictAvatarLanguage] final class O // missing data
private[StrictAvatarLanguage] final class X // check

// define new type for valid commands (no missing data)
type ValidCommand = IntermediateCommand[X,X,X]

case class IntermediateCommand[A,J,D](act:String, jnt:String, deg:Degree) {
     def the(joint:String) = IntermediateCommand[A,X,D](act, jnt, deg)
     def by(deg:Degree) = IntermediateCommand[A,J,X](act, jnt, deg)
}

// rotate prototype (requires more data)
object rotate extends IntermediateCommand[X,O,O]("rotate", null, null)
```

Code Example 10: Strict Avatar Language (Excerpt of Language Definition)

This method uses type-parameters for every single required property which flags whether this property is set or not. When a property is set, a new command with adjusted type-parameters is created thus this new command is of a different type than the original. For convenience and readability a type alias that represents a fully constructed command without any missing data is created. This type will be used in all the other language constructs like sequences.

```
// Flagged as type mismatch because the type is IntermediateCommand[X,X,O]
c:ValidCommand = rotate the "leftArm" *ERROR: TYPE MISMATCH*

// valid assignment because the types match
c:ValidCommand = rotate the "leftArm" by (7°)
```

Code Example 11: Strict Avatar Language (Example)

In the final implementation however default values were chosen over this method of handling missing properties. If this language where to be designed as internal DSL used only by other developers enforcing correct usage would be very desirable and could help prevent errors. However in this case, as the language is used by average users, it is preferable to perform at least some sort of action instead of just returning an error.

3.4 Language Interpretation

3.4.1 Architecture

The previously described language is just syntax and executing code in this language will only build up a data structure. It will not perform any action. The language definition in Scala can be seen as a lower-level parser API while the actual semantics can be implemented in the higher-level application. This separation of concerns would make it possible to completely replace the technologies used for the language without losing much of the application logic. This architecture of course resembles the language definition in the sense that they both try to model the same problem.

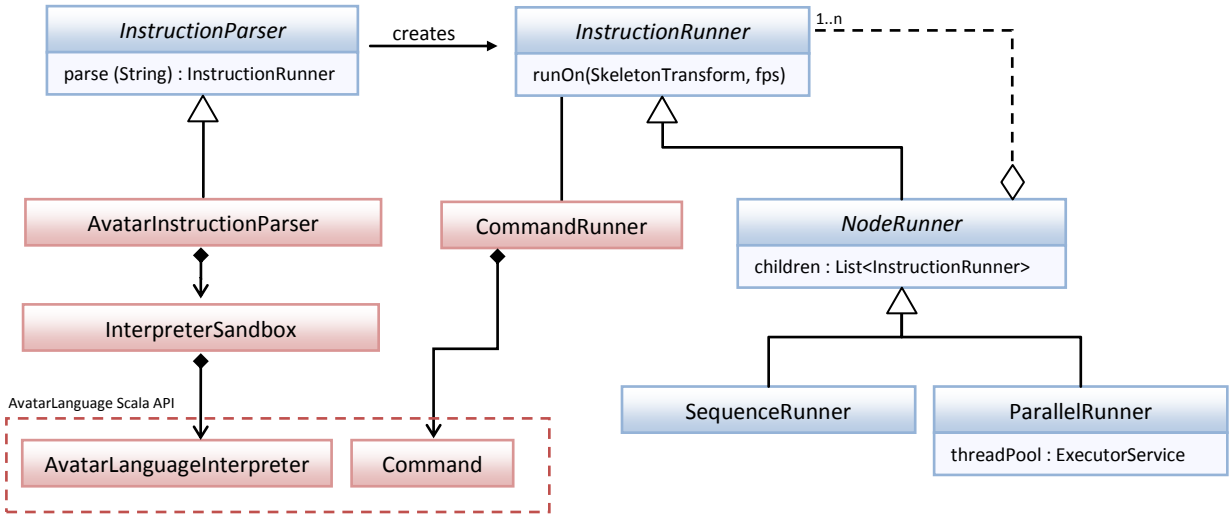


Figure 3: Architecture of the instruction interpreter

3.4.2 Performing Animations

The process of performing an animation is accomplished by updating joint transformations in small intervals as defined by a given set of instructions. The problem however is finding out the transformation values for all joints for any given point in time within the animation sequence. There are two ways this can be solved. One option would be to analyze the sequence of sequential and parallel and instructions and by reasoning about the succession and duration of each of the commands, the exact transformation values of any point in time can be interpolated. Naturally only one execution thread is required so there is no need to consider problems of concurrency. The other option is by simulating the sequential and parallel running commands using multiple threads and synchronization thereof. Many threads may run concurrently continuously updating joint transformation values which due to their shared mutable state would require additional synchronization. This solution might sound more complicated at first but in Java this kind of concurrent logic can be implemented quite quickly so this solution was chosen.

3.4.3 Exposing the internal DSL as external DSL

The Scala compiler API provides an interpreter that will dynamically compile and execute code at runtime. But this interpreter is designed for providing a shell-like environment. There is no direct method of for simply evaluating an expression and getting the result. There is however support for adding bindings to the interpreter context so a simple binding that would hold the result of the expression was defined. The type system is used to enforce an acceptable return value. An expression like "`4+2`" would be valid Scala code, but since the type of this expression is Int, it will be flagged by the compiler as type mismatch.

```
val capture = new Capture
bind("_capture", capture)

interpreter.interpret("_capture {" + code + "}") match {
     case Success => capture.result
     case _ => throw new Exception("Error: " + output.toString())
}
```

Code Example 12: Capturing the value of an expression

3.4.3.1 Scala compiler classpath issue

There is an important issue that needs to be considered that is far from obvious. The Scala compiler uses its own class lookup mechanism for loading classes and has its own classpath. It does not use the class loading mechanism of the main application. That means that even thought classes of the Scala library can be loaded in the main application without problem, the Scala compiler will immediately abort due to missing dependencies if these dependencies are not explicitly stated in the compiler classpath as well. In order to solve this issue all libraries required for Scala and the DSL must be added to the bootclasspath of newly created Scala interpreter objects. Naturally the file paths to these application libraries cannot be hard coded but must be

determined dynamically at runtime which is possible by querying the protection domains of classes that are part of the required libraries.

3.4.4 Error Reporting

Basic reporting of syntax errors is supported by simply forwarding compile time errors in the interpreter. But since those errors are directed at developers these errors may make little sense to average users. Error reporting is however an inherently difficult problem in this kind of setting. The Scala compiler itself can be considered a black box so it is next to impossible to translate the original compiler error message into a more user-friendly error message on DSL level.

3.4.5 Privileges and Security

In a traditional parser, security is not an issue because text is merely analyzed and converted into a structure. But in this case a full-featured language is used and text or rather code is converted into a data structure by executing that code. Instructions can be illegal in terms of the DSL because code is only required to conform with Scala syntax. Code is given by the user, who is considered an untrusted source. Consequently that code must not be trusted. If there is no consideration about security and privileges then this untrusted code will inherit the privileges of the parent application and runs without any restrictions, allowing access to the file system (e.g. deleting files) or the virtual machine (e.g. terminating the VM). Unless the application is run as an unsigned applet which is sandboxed from the beginning, great care has to be taken when executing user-defined code.

The Java platform has a comprehensive security model. Each thread is associated with an access control context which means that privileges can be restricted for a single calling context. This effectively means that the interpretation of given code can easily run in a sandboxed environment without access to any critical functions or system resources.

```
AccessController.doPrivileged(new PrivilegedExceptionAction<Instruction>(){
     public Instruction run() {
           return interpreter.evaluate(expression);
     }
}, getSandboxContext());
```

Code Example 13: Run code with restricted priviledges

The sandbox context is defined by a set of privileges that are granted to the execution thread. Some runtime permissions must be granted to the Scala compiler in order to work, but access to all critical resources can be eliminated.

3.4.5.1 Default Security Manager

When the sandboxing of code was first tested it didn't seem to work with no explanation as to why. Code was executed as if the access control context was never restricted yet the code for doing so was perfectly valid. The reason for this unexpected behavior was found in the way of how classes of the Java standard library check if the current calling context is allowed to perform a given action. Usually the security manager will be consulted which in turn should than check against the current access control context. However by default there simply is no security manager. Java application that requires access control checks therefore must initialize the security manager. Subsequently all access restrictions will be respected and access violations will result in expectations.

3.4.5.2 Dealing with Infinite Loops

Even though access of the DSL code is restricted, there are still ways of writing malicious code. Program structures such as loops are allowed to make the DSL more versatile but this also allows infinite loops to be written which means that even though the DSL code is syntactically valid, running the code will never result in anything as the code will run indefinitely. Since the code is just looping and not accessing any resources the access control context is meaningless.
This problem was however left as an open issue as instructions are written by the user and executed indefinitely on the machine of that user until the application is terminated. No harm apart from wasting CPU cycles can be done. It would be possible to solve this problem by defining a timeout and forcefully terminate the interpreter thread if execution takes too long.

3.5 Maintainability and Extensibility

The language is first tested within a Scala test application making sure that all use cases of the DSL are working. Whenever the DSL is changed in way that would break code in the test application, the compiler will immediately flag the corresponding line as problem so there is no need for basic tests. Unit tests however are used to make sure that the resulting data structure for a given piece of DSL code is correct. Animations are only tested by checking that the correct transformations are applied in the correct amount of time making sure that synchronization of instructions works. Intermediate states of the animation are not tested programmatically.

There are two types of language extensions that have to be separated. One kind of language extension would be additional parameters or structures that need to be respected when running the instructions. While this is can be achieved fairly easily, work in two parts of the system as to be done. However if only syntactic features are added that only change the way of building up the data structure without making changes to the capabilities of the actual objects, then this can be accomplished solely by changing the Scala code of the language definition.

One problem that was found early on was that if multiple instructions for the same joint are written, then the name of the joint has to be stated repeatedly. This issue was solved by allowing "using blocks" that look as follows.

```
using(left arm) {
  (rotate by 30) and (tilt by 45)
}
```

Code Example 14: Usage of the "using" feature

The code required in the language definition is surprisingly short and concise making use of Scala language features such as closures, pattern matching and sequence comprehensions.

```
def using(joint:Joint)(block: =>Instruction):Instruction = block match {
  case command:Command => command the joint
  case Sequence(seq) => Sequence(for (it <- seq) yield using(joint) { it })
  case Parallel(par) => Parallel(for (it <- par) yield using(joint) { it })
}
```

Code Example 15: Implementation of the "using" feature

This method takes two arguments. One is the joint that should be used and the other is a function that results in an instruction tree. This instruction tree is recursively traversed and a new instruction tree is recreated by applying the given joint to every command.

3.6 Evaluation

Using Scala as a host language for the new DSL enabled a fairly comprehensive and extremely extensible language to be created. The Scala language itself is a well thought-out and very concise language. It is syntactically very different from Java so a high learning curve has to be expected. However there were problems in using it to dynamically execute scripts due to Scala being a compiled language. An important point to consider is that the Scala compiler cannot be used within an unsigned Java Applet or Java Web Start application due to its class lookup mechanism. One problem that may be an issue is that the Scala compiler is a quite large and complex software component. Just loading all the required classes may take one or two seconds when the interpreter is first initialized which results in noticeable delay when the first instruction is interpreted. Additionally the Scala compiler is very slow due to complex tasks such as type inference. Even compiling a simple instruction of only a few lines will take a couple of hundreds of milliseconds. Even though execution is then extremely fast, it doesn't matter as every instruction is only compiled and executed once. These small delays however would not be noticeable by the user and since this language is directly targeted at the user these performance hiccups are not considered an issue.

4 Visualization of the 3D Avatar

4.1 Introduction

The 3D visualization is realized using a Java applet that will render a 3D world via jMonkeyEngine. The primary task is to load and display a 3D human model and then define an interface which allows individual joints to be rotated programmatically. The visualization represents the main application that is able to accept instructions using the previously described language and play the corresponding animations.

4.2 Human 3D Model

4.2.1 Loading of 3D Models

Loading 3D models into the scene was one of the aspects that were completely underestimated. It was assumed that loading 3D models downloaded from the internet in any format supported by jME would be trouble-free. However as it turns out, while jME supports multiple formats, most of them are only supported partially. There may even be multiple loaders provided by the community all of which support a different feature set for the same format. Almost none of the models that can be found online could be loaded correctly into the scene. Another aspect that was not thought of before is that the model that is used has to be rigged which means that the 3D body has to have bones which can be animated resulting in a change of the skin geometry. This requirement makes finding a suitable freely available model even more difficult.

jME defines its own XML-based format for 3D models and an export plug-in for Blender. Using Blender and exporting to the jME format works great for simple geometries. It was however not possible to export more complex models. Blender itself already has trouble importing non-native formats. The geometries were never a problem but textures and bones where usually missing.

4.2.2 3D Modelling

Due to the problems with loading complex 3D models, an attempt was made at creating a model manually using Blender. However this endeavor was vastly underestimated due to the lack of prior experience in 3D modelling. Even creating a simple rigged stick model in Blender is exorbitantly difficult for users that are not familiar with the tool. The more time was invested into modelling the less likely it seemed that a visually appealing model could be created in a reasonable amount of time so this endeavor was eventually given up without producing any notable results.

Poser, a modelling application that is solely designed for creating human models was tried as well, but the models created with this application have such a high number of polygons that they cannot be rendered in real-time. In addition the supported interchange formats have already been confirmed to be troublesome for use in jME.

4.2.3 Default Model

After all attempts of loading a rigged model from the internet and manually modelling failed and since 3D modelling was never considered to be a part of this project anyway, it was decided to use one of the models used in jME for demonstration which was known to work perfectly. While this model is rigged and nicely textured there are some pitfalls. Most importantly only the main body and the limbs are rigged but the hands and fingers are not. One of the original ideas was to be able to describe simple signs of sign languages. While rest of the system would in theory supports this use case, it could not be realized as the fingers of the model are not rigged and thus bones cannot be animated.

Each bone is identified in the model by a unique name and each single bone can therefore be easily addressed separately be accessing the node graph of the skeleton. Bones are structured in a tree so that when a parent bone is transformed, this transformation will then apply to the child bone as well. This means that when an arm is rotate the hand is rotated as well along with the arm. Because each bone is solely addressed by its name the model could theoretically be exchanged as long as the bone structure and bone names are compatible with the original model that is used for testing. The main difficulty however is either creating a model or finding a model that can be loaded into jME. Therefore the final application does not allow for a selection of models but only uses the default model.

4.2.4 Playing Animations

When playing an animation the language system will continuously adjust variables in an internal data structure that represents the skeleton with all joints. This data structure is however completely independent from any kind of visualization. 3D animation is achieved by applying the current rotation values of this internal data structure to the bones of the 3D model on every frame. There are some aspects that have to be considered when applying logical values of the internal representation to actual 3D transformations of a bone node in the scene graph. First of all properties like rotation and tilt have to be mapped to a 3D rotation, but this 3D rotation cannot be applied to the bone directly. Every bone has an original 3D rotation which is defined in the model which must be taken into account when changing the transformation of a given bone node. The desired behavior can be achieved by combining the original transformation in the model with the additional transformation defined by the animation.

4.2.5 Skin Deformations

The 3D model consists of two aspects. One is the skin which represents the visual geometries and the other is the skeleton which is an invisible construct that is used to control and animate the skin. Each bone is associated with a set of vertices of the skin geometry that build up this body part. When the bone is transformed the corresponding vertices are move accordingly which results in the transformation of the skin geometry and thus visible movement. However certain

type of rotations or tilts of joints my affect the model in unnatural ways. The upper body for example is deformed when the shoulder bone is rotated. These unnatural deformations could be fixed by fine-tuning the model which defines how each vertex is affected by corresponding bone transformations. Due to the lack of experience in 3D modelling this issue was left unresolved.

4.3 Evaluation

Despite the problems with loading of models, jMonkeyEngine turned out to be a very good choice to deliver state-of-the-art 3D graphics. The library itself is huge and can be daunting at first especially for developers that lack a background in 3D programming. There are however many resources that help in learning key concepts and the API itself contains a vast number of test cases and demo applications that showcase all aspects of the library. This comprehensive library also speeds up development as many components commonly used by 3D applications are readily available. Features of the API such anti-aliasing, shadows and bone debugger can easily be added to the application making it much more visually appealing.

5 Deployment on the Web

5.1 Introduction

The final goal is to deploy the 3D visualization along with scripting support for the DSL as a proof-of-concept Java applet embedded within a website. The applet should be able to execute DSL statements dynamically given via JavaScript. This is designed to proof that all the components that have been built and all the technologies that have been used can in fact be deployed to the web. Since this website is proof-of-concept only, it will consist solely of the visualization applet and a plain html test area where DSL code can be entered. In theory however the DSL statements that are executed may as well have been provided from a database of animations.

5.2 Applet Deployment using JNLP

Deploying pure Java code in form of an applet is a non-issue. jMonkeyEngine however depends on a native library which is required for hardware accelerated rendering using OpenGL. Normal applets must run in a very restricted environment. Calling native code on the other hand would allow a sandboxed application to bypass any security restrictions imposed by the virtual machine. In the past a dependency to a native library would have made deployment as an applet impossible.
Java 6 Update 10 however introduced major improvements in the Java plug-in architecture. The most important change was added support for the Java Network Launching Protocol (JNLP) which allows Java applets to be deployed using jnlp descriptors similar to Java Web Start

application. Web Start applications can be launched either in a sandboxed environment or with unrestricted permissions. This applies to applets as well now. There however are additional security requirements imposed on applets or Web Start applications that run with unrestricted access. Most importantly all code used in the application must be digitally signed. In addition the Java plug-in itself will ask the user for explicit authorization before launching the application.

5.3 Pack200 Compression

When the applet is deployed on the web the amount of data that has to be downloaded has to be kept to a minimum in order to guarantee quick startup of the applet. Due to the nature of this project one would expect that most of the size is caused to the 3D model and textures. In fact however most data is cause by the Scala compiler and jME. Both of these APIs are extremely large and containing thousands of classes that take up multiple megabytes of Java byte code. This problem can be tackled by using pack200 compression. Pack200 is a compression algorithm that is specifically designed to compress Java byte code which makes extremely high compression rates possible. By default the pack200 command line tool will first apply pack compression and then further reduce file size by using gzip compression. By using pack200 the file size of an already zipped jar file can typically be reduced by about two thirds. Using this method the size of the applet could be reduced from 17 MB to 7 MB which results in a noticeable speedup when loading the applet for the first time. All the data will be downloaded directly into the Java Web Start cache so no download is required when the applet is launched for the second time.

5.4 Code Signing

All the code that is deployed on the web is digitally signed so it can be run with full permissions. Because a self-signed certificate is used to sign the jars the Java plug-in will ask the user to accept this certificate.

When code signing is used in combination with pack200 compression, there is an interesting dilemma. Pack200 will restructure class files as part of its compression algorithm which of course invalidates any digital signature. So the jar can't be signed before compression. But signing the jar in its packed form is also not possible. Pack200 however provides a solution. First pack200 is used to restructure the class files within a jar without compressing it, then the jar is signed and finally the actual pack200 compression is applied.

During development an odd problem was encountered. Sometimes the Java plug-in would reject a signed jar for being tampered with because the signature did not match. The problem would sometimes appear or disappear by making completely unrelated changes to the program. The issue was tracked down to a know bug in pack200. Due to this bug, pack200 would sometimes restructured already signed class files even though they have been restructured before thereby invalidating the digital signature.

5.5 Solving the Scala compiler classpath issue

As mentioned earlier there are problems with the Scala compiler not being able to lookup required classes. In the first instance this issue was solved by manually adding the paths of the library jars to the classpath of the Scala compiler. There is however a fundamental problem when the application is launched via JNLP. The idea of Java Applets and Web Starts applications is that they are launched directly from the web. So when the location of a library jar is queried via the protection domain then the result will be the URL of the remote location and not the file system path of the location of the cached jar. The Scala compiler requires local file system path to work with. This was a very serious and unexpected problem because it meant that the interpreter for the DSL cannot be deployed on the web making a crucial aspect of the project impossible.

The class lookup mechanism of the Scala compiler makes it impossible to dynamically compile and execute Scala code within an unsigned Applet. Nothing can be done about this fact and there is no workaround within the limitations of a sandboxed Applet.

Luckily, the Applet is already required to be launched with full permissions due to jME so in this case a workaround is possible because the application has full access to the file system and thus to the Java Web Start cache. First the file system path of the required jar in the cache has to be determined. Reverse engineering the folder structure and inner workings of the Web Start cache however is not a feasible solution as it could change with every update. Instead the JNLP class loader which allows the application to load classes from the cache is used to translate remote locations to local file system paths of the cached jars by using internal APIs from com.sun.jnlp. This jar is than copied into a temporary folder only to rename it to make sure that the file extension is ".jar". Otherwise it would not be accepted by the Scala compiler. This solution is far from elegant but it is only way to work around the limitations of the Scala compiler within a JNLP application.

5.6 Evaluation

Deployment on the web was expected to be smooth but there were more critical issues than in every other part of the project. During deployment it turned out that Scala may not have been the best options as a host language for the DSL. But because this application has to runs unrestricted access a workaround is possible. However if a solution that runs within a sandboxed Applet is required, Scala is simply not an option. In addition a bug in pack200 would seemingly randomly break code during deployment. Solving problems that only occur after deployment is extremely troublesome because the only way to debug already deployed applications on the web is using logging. Due to the size of the required libraries jar signing and compression plus subsequent deployment on a web server takes up 10 minutes and effectively breaks the flow of fixing bugs and testing the fixes.

6 Project Evaluation

6.1 Design Discussion

The project has set itself particularly ambitious goals. One was a modern 3D graphics system and the other a new domain-specific scripting language. There are many different ways in how these problems can be approached but finding good solutions for both of these aspects that can be implemented in the limited amount of time that is available is the main challenge of this project. The idea was to make use of existing technologies that would allow a sophisticated and elegant solution. The Java platform with jMonkeyEngine and Scala proved to be well suited to provide a powerful foundation that would make it possible to achieve the defined goals. If a different set of technologies was used, the projected could have easily failed in achieving its goals.

6.2 Planning and Scheduling

The project was about prototyping from the very beginning. Technologies like jMonkeyEngine and Scala were unknowns so the capabilities of these technologies were unclear in advance. The decision to use prototyping was both due to the fact that those technologies had to be explored step by step but also because it was uncertain if the goals of this project can be met. Prototyping would ensure that there is some sort of result that can be showcases at any point in time even if the project cannot be finished completely. Consequently both visualization and language were developed in parallel with incremental prototypes. The project plan was in general too optimistic so the project fell behind schedule little by little as time went past. However steady progress was made so this was not considered much of an issue. However the problems with loading 3D models which was initially expected to take less than a day and the subsequent attempts at 3D modelling greatly disrupted the plan as virtually no tangible progress was made for almost a month. Development however sped up when more in-depth understanding and experience with jME and Scala was acquired. Another critical problem emerged when almost everything was finished and only deployment on the web was missing. Even though everything worked in a development environment during deployment many new issues emerged so finishing up the implementation and deploying the applet took a lot much longer than initially expected.

6.3 Conclusion

The project has achieved its goals in the sense that it was shown how technologies like jME and Scala can be used to complete sophisticated systems. While impressive progress has been made on both the visualization and the language, these systems can only be seen as proof-of-concept prototypes. The language for example is easy to read on a line per line basis but the resulting possibly nested rotations are hard to visualize in the mind. While creating a readable language was a success, to fully comprehend the series of nested rotations required for more complex animations is difficult if not impossible. The way the language is implemented and how easy it is to extend the language demonstrates clearly that this approach was elegant indeed. While the

actual language itself is fulfilling its purpose, it is lacking in term of usability. The main point of criticism regarding the visualization is undoubtedly the lack of fine-grained control of the fingers even though one of the original ideas for this project was sign language visualization and even though the language system would support controlling the fingers it is not possible. The simple reason is that the model that is used is lacking the necessary rigging.

Compared to system implemented Z. Huang, et al. (2003) the DSL used in this application is clearly inferior to STEP in all aspects but probably realized in only a fraction of the time and probably much easier to extend than considering the approach that was taken in implement it. The visualization on the other hand, while not targeting a standard like X3D, is more visually appealing due to the use game engine which provides better graphics capabilities.

6.4 Further work

The main aspects that could significantly increase the usefulness of this project are a redefined language and a better 3D model in the visualization. Adapting the code to these new features would only require minimal time. Especially the language could probably be implemented in a similar manner as was done with the current language but the main difficulties lie in defining a powerful and comprehensive language for human animation that is easy to read and to comprehend.

7 References

Z. Huang, A. Eliëns, C. Visser. Implementation of a Scripting Language for VRML/X3D-based Embodied Agents. In: Web3D
8th international conference on 3D Web technology
Saint Malo, France, 09-12 March 2003. ACM.

R. Ferreira, 2008. Type-safe Builder Pattern in Scala [online], Updated 9 July 2008
Available at: http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html
[Accessed 17 April 2010]

8 Appendix A: Project Proposal

Department of Computing Degree Project Proposal

Name: Reinhard Pointner Course: Computing Size: double

Discussed with (lecturer): Bogdan Matuszewski / Maureen Nicholson Type: development

Previous and Current Modules

CO3401	Advanced Software Engineering Techniques
CO3512	Neural Networks
CO3708	Database Driven Web Sites
CO3716	Flash Programming for Interactive Entertainment
EL3105	Computer Vision

Problem Context

There are lots of different sign languages used all around the world. This includes more complex sign languages used by deaf people, simplified sign languages that are sometimes used when working with people with mental disabilities but also military hand signals. There are also multiple notation systems, some which vary in simplicity and expressiveness. Yet there are very few dictionaries available online, all of which rely heavily on animations or videos making them hard to extend. None of them use a notation system that would allow the user to query for signs and get a list of words in return.

The Problem

The main goal is to create a collaborative online dictionary for any and all sign languages. The idea is to provide the software that will allow users to input signs and meanings for existing sign languages (or create their own). Naturally this will be implemented in some kind of website to reach a high number of potential users and contributors and facilitate search not only “text to sign” but also “sign to text”, which is not possible in any of the current systems. Signs obviously are best displayed visually, so a fully animated 3d avatar will show users how to do any given signs. This 3d avatar will be animated by using some kind of notation system. This notation system must be both expressive enough so sign language experts will want to use it but also very simple to understand, so people are not scared away from extending and collaborating in creating a comprehensive sign language dictionary.

Potential Ethical or Legal Issues

None.

Specific Objectives

3D Avatar:
This entails writing an application that will display a 3D computer model of a human body. This program must expose some kind of API that allows other application to fully control this model in terms of arm, hand, finger and head movements as well as facial expressions.

Sign Description Language:
Some kind of domain-specific language must be worked out to describe signs. It must be able to express all kinds of hand and finger movements and facial expression and take concurrency and parallelism of gestures into account. But at the same time, another objective is to make it as simple as possible to understand and input these notations, as non-programmers will be using the system.

Collaborative online sign language dictionary:
The objectives mentioned above will be embedded in an online dictionary. Users must be able to search for specific terms or phrases but in contrary to ordinary dictionary the sign language hits will be demonstrated by a dynamically and automatically generated 3d computer animation. Time will be limited so the objective is not to create a fully featured collaborative online dictionary, but a mere prototype that will show off the above mentioned objectives and how they can be embedded in a website.

Mini-Paper:
The mini paper will discuss how different tools and technologies that can be used for the creation of domain-specific languages. Criteria for comparison will entail ease of use, expendability and limitaions of those technologies.

The Approach

First of all, I need to identify key aspects of sign languages to get a deeper understanding of how sign language actually works to convey information. Most sign languages convey meaning in form of arm movement, hand direction, easing of movement, facial expressions, and parallelism [4]. The main idea is to expose the final product in form of a website, so a Java applet will be developed that will be able to interpret the notation system that will be developed and render the 3d animation.

This project will heavily rely on 3rd party open-source libraries, such as JOGL of hardware accelerated 3d graphics and maybe even some kind of game engine [1] if this turns out to be reasonable. Once the 3d visualisation implements basic requirements, a control system can be

implemented that can convert written notation to 3d transformations. Prototypes of the notation system will be created and further refined in response to feedback from sign language experts. This may be accomplished by either by creating a DSL based on Scala [2] or, if this approach is not expressive enough, by using ANTLR Parser Generator [3] to build up the new language from scratch.

Once visualisation and notation system are sufficiently useful, these two systems will be incorporated in a very simple online dictionary.

Development itself will be mostly in Java using Eclipse as IDE as well as Blender for 3D modelling. A subversion repository will be used for both versioning and backup purposes.

Resources

- Maureen Nicholson - Senior BSL/English Interpreter

Potential Commercial Considerations

Estimated costs and benefits

The success project will depend on how viable the notation system and the 3d avatar turn out to be as alternatives to videos. Of course this project will not be able to capture full human expressiveness as perfectly as videos do, but if is at least reasonably useable, this project could easily be developed into a centralized open database of signs in a computer-readable format. This information could be used to support lookup of signs for given “sub-gestures”, which would be a unique feature that cannot be provided by traditional sign language dictionaries.

Mini-Paper Content

A comparison of tools and technologies used for developing your own domain-specific language

References

[1] JMonkeyEngine, http://www.jmonkeyengine.com (2009)

[2] DSLs - A powerful Scala feature, http://www.scala-lang.org/node/1403 (2009)

[3] ANTL Parser Generator, http://www.antlr.org (2009)

[4] Buttussi F., Chittaro L., Coppo M. (2007): Using Web3D Technologies for Visualization and Search of Signs in an International Sign Language Dictionary

9 Appendix B: Mini-Paper

An Evaluation of Scala as a host language for DSLs

Reinhard Pointner,

BSc (Hons) Computing

Supervisor: Bogdan Matuszewski

Second Reader: Chris Casey

22 April 10

Abstract

This paper discusses and evaluates the use of the Scala programming language in regard to domain-specific language development. This paper will outline the various advantages DSLs provide in general and explain the difficulties inherent in DSL development. In addition we will examine different approaches to DSL development and their respective characteristics. We will then evaluate the features of Scala DSL development. These features include easy reuse of existing language infrastructure, composability of multiple DSLs and pluggable DSL implementation due to the fact that DSLs are defined within the Scala language itself as well as performance due to compilation. Support for domain-specific languages is then further discussed by looking at Groovy in contrast to Scala including a brief explanation of how Scala can be used as a foundation for an external DSL. We will then look at how Scala can be used for the creation of a domain-specific language for human animation and some of the benefits of this approach.

Introduction

Domain-Specific Languages

Mernik (2005) defines DSLs as languages that are designed and optimized for a specific purpose or set of tasks, thereby offering substantial gains in expressiveness and ease of use in comparison to general-purpose programming languages. The idea of domain-specific languages is not new. DSLs for various tasks have been around since early ages of computing (Mernik, 2005). One of the main differences between GPLs and DSLs is that the latter is not necessarily executable by itself. It may merely be some kind of description language like regular expressions for text patterns, make files for building software or simply HTML and CSS for websites. Although there are many successful and widely used examples of DSLs, standalone applications rarely make use of small special-purpose languages. Mernik (2005) and Hofer, et al. (2008) agree that the primary reason for the small adoption of DSLs is simply that the extreme amount of effort that is required to create a new language from scratch almost always outweighs possible benefits.

Overview

This paper will introduce DSLs in general and evaluate Scala as a host language in particular. The following section will outline on various approaches to DSL development. The third section will evaluate the use of Scala in respect to certain set of desirable DSL development features and move on to a direct comparison between Scala and Groovy. The use of Scala for writing a DSL for human animation will be discussed in the fourth section.

Approaches to DSL development

Fowler (2009) generally distinguishes between internal and external DSLs. External DSLs are usually written from scratch as part of an application that uses this DSL for interpreting certain types of input. An internal DSL on the other hand is built upon or within an existing language. The aim is to reuse existing language infrastructure like syntax checker, parser, compiler and interpreter as far as possible.

Mernik (2005) points out however, that DSL development does not only require technical expertise in language design but also domain expertise and that few people have both. Since DSLs are typically designed to be used by non-programmers to accomplish a specific range of tasks, a deep understanding and insight into that purpose is required when defining syntax and semantics.

External DSLs

Mernik (2005) talks about language invention in this regard as external DSLs are typically implemented from scratch. Language design can be arbitrary and there are virtually no limitations which allows for syntax that is close to the notations used by domain experts (Mernik, 2005). This requires defining a grammar, then a parser for converting the textual or graphical input to some kind of data structure and finally an interpreter to evaluate the input expression. Therefore the implementation will require a considerable amount of time, effort and technical expertise in language design. There are many tools like parser generators that help substantially, but language development will nevertheless remain quite complex. It is for that reason that extensibility of the language is usually not the first priority. Mernik (2005) highlights this as one of the main disadvantages of this approach and argues that domain-specific languages are generally much more prone to language changes than general purpose languages because the domain experts will usually request more and more features as they are working with the language. In addition maintaining the language will proof to be difficult especially for developers who are not familiar with the specific implementation.

Internal DSLs

Internal or embedded DSLs are typically built within an existing language. The DSL will be able to reuse the complete tool chain of the host language. Therefore no grammar, parsers or any other language development tools are required. The language is defined solely by using language features of the host language which imposes certain limitations to the DSL syntax. Mernik (2005) marks this as one of the main disadvantages and argues that most languages don't allow for arbitrary syntax extension. Naturally languages that provide syntactic flexibility are more suitable for creating a DSL. One of the main advantages of internal DSL development is that it does not require any kind of special expertise in language development. Fowler (2009) suggests the term "fluent interface" in this regard, because internal DSLs are fundamentally a form of API. This is also one of the reasons why internal DSL are much easier to maintain and to extend because in essence developers only need to be familiar with the host language to be able to improve upon the DSL.

Mernik (2005) however defines this type of DSL in more general terms when he talks about language exploitation. When reusing an existing language the DSL can overcome limitations of the host language by either extending the host language itself or by using some form of pre-processor that converts DSL code to host language code before compilation. This approach of adapting an existing language to make it more expressive will make language development significantly more complicated when compared to the pure embedding approach that solely uses existing language features.

Evaluation of Scala as a host language

In the previous section the merits of domain-specific languages have been explained and general implementation approaches have been discussed. The idea of using Scala as a host language puts this approach into the domain of internal DSL development and therefor most of the previously mentioned advantages and disadvantages apply.

This paper focuses on Scala rather than any other language because Scala promises language flexibility that is intended to allow developers to capture their respective problem domains more naturally (Odersky, et al. 2006). The same kind of reasoning leads to domain-specific languages.

Mernik (2005) argues that embedding suffers from limited user-definable syntax in languages like Java and observes no trend towards more powerful languages. However Hofer, et al. (2008) illustrates the opposite with Scala compellingly but points out that there are certain syntactic constructs that cannot be modelled easily.

Hofer, et al. (2008) identifies the following set important DSL features:

- Reuse of Infrastructure
- Pluggable semantics
- Performance
- Composability

Reuse of Infrastructure

The approach of defining a DSL purely in Scala syntax immediately allows the language developer to reuse the complete Scala API as a well as all the additional tools. This includes syntax checker, compiler and maybe most notably IDE support. There is also debugging support but since the debugger will work on the host language level rather than the DSL level it is less useful (Hofer, et al. 2008). It is noteworthy however that debugging support is generally not implemented at all in most DSLs because it is usually not considered worth the effort.

Pluggable semantics

The idea of pluggable semantics is to decouple the DSL syntax from a single specific interpretation and thereby allowing multiple possibly very different implementations of the same language. Scala is perfectly suited for this purpose as this concept can be implemented using straightforward polymorphism as Hofer, et al. (2008) illustrates. Essentially the DSL syntax is defined by an abstract class definition and the concrete semantics are defined by the respective implementations thereof. Those implementations themselves may then again be extended and adapted by others. Where Mernik (2005) was not able to answer how to extend implementations in a safe and modular manner, Hofer, et al. (2008) seems to have found a simple solution with this approach.

Performance

Performance is generally not an issue for domain-specific languages. However it might be noteworthy that any Scala DSL code is compiled to Java byte code, virtually eliminating language related overhead.

Composability

Composability describes the concept of using multiple DSLs in the same code. This is virtually impossible for external DSL implementations because it would require merging two or more very different and possibly extensive codebases.

As far as Scala is concerned, DSLs are a merely a form of API. When implementing a new DSL the developer can simply make use of existing lower-level DSLs. Multiple DSLs may even work and interact on the same level as long as they agree on common interfaces and types.

Scala vs Groovy

Scala is only one of many new languages targeting the JVM. Each one of them tries to solve issues that are inherent in traditional programming languages. Scala and Groovy are among the most prominent ones.

Scala is a statically-typed compiled language. Groovy on the other hand is a dynamically-typed scripting language. Because of those features, it seems apparent that Groovy would be a far better choice as a host language for both internal and external DSLs. It is easy to evaluate Groovy expressions dynamically at runtime from within a Java application and languages targeted at non-programmers usually don't make use of type systems because it would unnecessarily increase complexity. In fact, G. Laforge (2007), one of the core developers of Groovy, argues that Groovy syntax is malleable and flexible which makes it the perfect choice for creating DSLs. However Groovy first and foremost improves upon Java and adds many features of popular scripting languages. The aim of Scala on the other hand is to create a new language that is in itself extensible enough in order to allow developer to model their respective domains easily and naturally in libraries and frameworks (Odersky, et al. 2006), instead of providing a certain set of built-in features. Even though Scala is statically typed it does not hinder DSL development significantly because of a sophisticated type inference system which permits to omit actual type (Odersky, et al. 2006). This means that the type does not have to be stated explicitly in most cases giving Scala many syntactic advantages that are usually associated with dynamically typed languages. Since Scala is a compiled language it stands to reason to question if it sensible at all to use Scala as a foundation for an external DSL because user-defined code will need to be evaluated dynamically at runtime. But this is actually only a minor issue because Scala provides API for dynamic compilation and interpretation. However Scala is more complicated than Groovy in this regard, because it is not primarily designed to be used for that purpose.

Even though Groovy seems more suitable to be used as a foundation for an external DSL initially, Scala extensive support for growing the language and internal DSLs, which can be exposed as external DSLs, may make it more worthwhile in the long run.

DSL for human animation

One aim of the project is to design and implement a simple user-friendly domain-specific language for scripting a virtual 3D avatar. Z. Huang, et al. (2003) tackled a similar problem when implementing STEP (Scripting Language for Embodied Agents) and chose to expose the internal logic via a DSL to the user. However due to the simplicity of the language some use-cases that would have been possible internally are not accessible via the external DSL.

A Scala DSL could help solve the dilemma of trying to simplify the input language and make it user-friendly while allowing advanced and much more complicated code as well. Since anything will be valid Scala code the user can choose to solely use the simplified DSL but may also directly access any language constructs, classes or APIs that are available to the programmer. Interestingly this concept is applicable vice versa. The developer may also choose to use the DSL in internal code. Therefore more complex parts of the DSL implementation could be defined in basic constructs of the very same DSL. The language definition can also be considered as an independent software component and may be reused for scripting any human-shaped body. However one of most significant reasons for using Scala in any kind of project is simply that a lot can be achieved within a limited amount of time.

Conclusion

The Scala language is DSL friendly by design. One of the major goals of Scala is to allow developers to produce expressive and concise code as well as easy-to-use APIs in form of internal DSLs. It is possible to implement DSLs in a modular manner that is impossible for external DSLs. First and foremost all DSL code is valid Scala code. Therefore multiple DSLs can be used in conjunction or on top of each other. In addition the DSL definition itself is a form of class that may be extended allowing multiple implementations of the same DSL using simple polymorphism.

As mentioned before, Scala is a compiled language and as such it is not primarily designed to be used as the foundation of an external DSL. There are however approaches of using the Scala compiler API to interpret new input dynamically at runtime.

Since everything is hosted completely within Scala, there is no actually language development in the traditional sense involved when implementing the DSL, allowing for rapid DSL development as well as straightforward maintenance and language extension.

References

C. Hofer, K. Ostermann, T. Rendel, A. Moors, 2008. Polymorphic Embedding of DSLs.
In: GPCE (Generative Programming and Component Engineering),
7th international conference on Generative programming and component engineering.
Nashville, Tennesse, USA, 19-23 October 2008. ACM.

M. Mernik (2005). When and How to Develop Domain-Specific Languages.
ACM Computing Surveys. 37 (4), 316-344.

Martin Fowler, 2009. Domain Specific Languages (WORK-IN-PROGRESS) [online]
(Updated 30 Jun 2009) Available at: http://martinfowler.com/dslwip/index.html
[Accessed 10 November 2009]

M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger (2006).
An Overview of the Scala Programming Language. 2nd ed. Lausanne: EFPL (École Polytechnique Fédérale de Lausanne). p18.

G. Laforge and J. Wilson, 2007. Tutorial: Domain-Specific Languages in Groovy [online]
(Updated 7 March 2007) Available at: http://glaforge.free.fr/groovy/QCon-Tutorial-Groovy-DSL-2-colour.pdf
[Accessed 15 November 2009]

Z. Huang, A. Eliëns, C. Visser. Implementation of a Scripting Language for VRML/X3D-based Embodied Agents. In: Web3D
8th international conference on 3D Web technology
Saint Malo, France, 09-12 March 2003. ACM.

10 Appendix C: Technical Plan

Department of Computing Final Year Project Technical Plan

Name: **Reinhard Pointner** **Size: double** **Mode: pt** **Time: 1**

Course: **Computing** **Supervisor:** **Bogdan Matuszewski**

Summary

The project is about 3D visualisation and a description language for sign languages. It can be devided into two general components. A 3D application displaying a human model (avatar) and some kind of easy-to-understand language to script movement.

The solution will entail basic 3D modelling but more importantly importing and animating the model within a standalone application. Part of the visualisation is also to reasonably mimic human movement (e.g. easing and constraints in movements). The notation system can generally be seen as creating your own domain-specific language (DSL) designed for describing the movements of the 3D avatar. These tasks seem to be a lot for a for the limited amount of time, but this project is also about using high-level APIs which should make this project feasible. The components that will be developed are intended to be used in websites so all technologies must be embeddable in modern web browsers.

Constraints

20. November 2009: Deadline for the mini-paper

22. April 2010: Deadline for software and the final report

Key Problems

The main problems concerning visualisation will be to import a 3D model but then animate parts (e.g. hands) of it dynamically in the standalone application. This will also include constraints and different body parts in terms of acceleration/easing and possible angles. One problem of the notation language is of course the language on a technical level. This may include creating a parser, evaluating expressions and of course allow for extensibility in the language. But the notation language will also need to be able to express complex and parallel behaviour and yet be as simple and easy-to-read as possible.

Risk Analysis

Risk	*Severity*	*Likelihood*	*Action*
A Scala DSL is not adequat for the notation system	*High*	*Low*	*Build language using ANTLR parser generator*
A proper 3D model cannot be obtained	*Medium*	*Medium*	*Use a simple stick model*

Options

This project will make use of rapid prototyping. It will first of all help evaluating the aforementioned set of technologies and APIs and decide if they are good choice to continue. Prototyping will also make sure that the project is always in an presentable state, even if some features have not been implemented yet. Step-by-step development will keep motivation high and help to monitor the overall project progress. Traditional more linear software development methods are not feasible because there my be many unforeseen complications for the lack of prior experience in this sort of thing. There must be a working application at a fixed deadline and prototyping will ensure just that. Unit tests will be used wherever it is reasonable, but since there is alot of visualisation involved, a lot manual testing will also be required.

The project will rely heavily on Java technology since this is the only choice to deploy complex high-performance 3D applications on the web. Flash might be considered an option too but there is a lack of key APIs that would definitely make this project impossible within the limited amount of time. In particular JMonkeyEngine, a game engine written in Java, will provide a feature-rich base for high-performance 3D visualisation. Some 3D modelling will be involved in the project, but open-souce software such as Blender should suffice for these kind of tasks. Additionally Blender objects can be imported by JMonkeyEngine. Scala and Groovy, languages that could serve as a foundation for the scripting language, are only available for the Java platform.

Potential Ethical or Legal Issues

None.

System & Work Outline

The system can be divided into three loosely coupled components. The focus of this project will be 3D visualisation of human movement and some kind of easy-to-understand notation system for scripting these kind of movements. The website will just serve as a proof-of-concept prototype that these components can be embedded in websites and run in the browser platform independently.

All components will rely heavily on the Java platform so everything can be deployed via Java applets and of course for the existing high-level libraries. The 3D visualisation will depend on

JMonkeyEngine to provide 3D acceleration, a scenegraph and loading of 3D models. The model itself will need to be designed in a way, that allows the Java application to access bones and joints and change properties like angles to dynamically control movement. If an existing model is used then information about rigid parts and joints will have to be retrofitted into the model using 3D modelling software such as Blender so those parts can be referenced from within the application.

The first step in implementing the movement description language will be evaluating if any existing languages like Groovy or Scala can be adapted for this purpose. Both of them allow APIs to define internal domain-specific languages with syntax-like structures. This feature can propably be used to quickly build an easily extendale language. Of course this approach has certain limitations, so it might actually be necessary define and implement a completely new language and interpreter for it. ANTLR parser generator may be used to simplify this task, but the first approach is preferred because it will propably be much simpler.

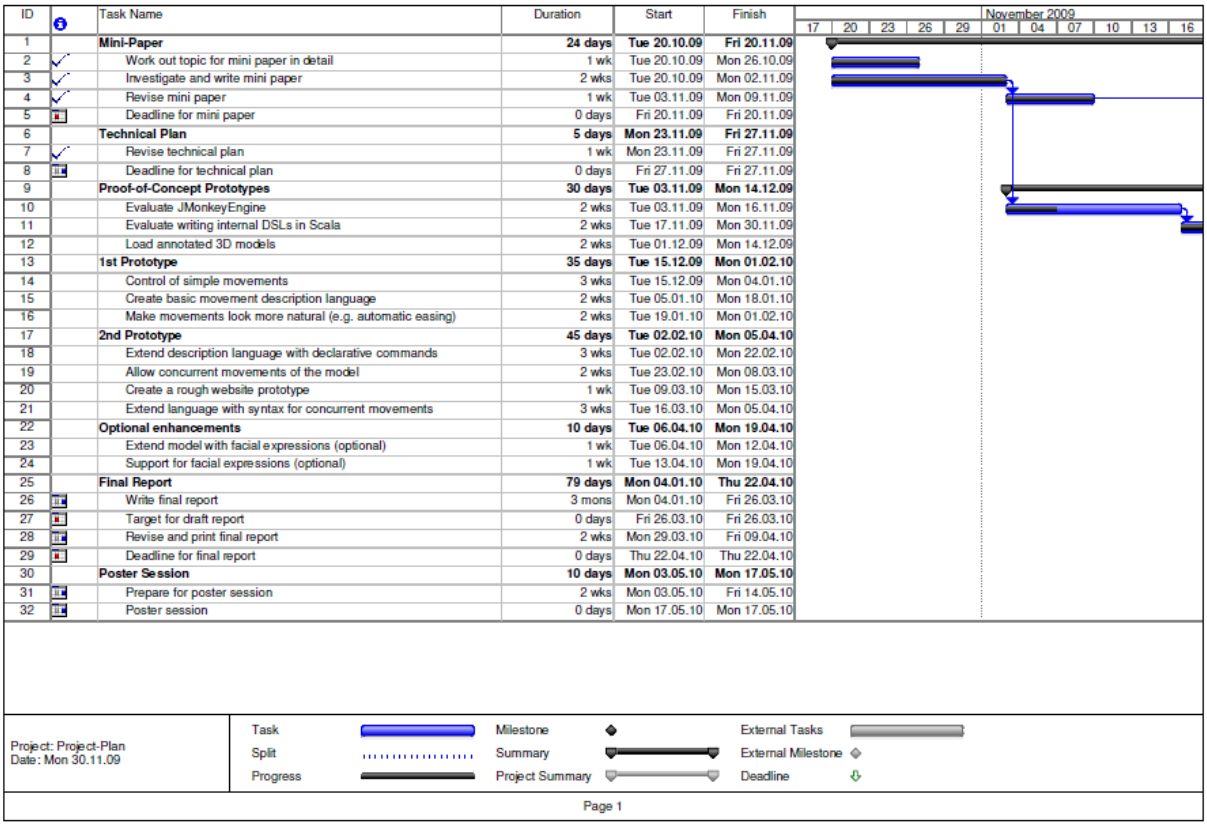
Commercial Analysis

Estimated costs and benefits

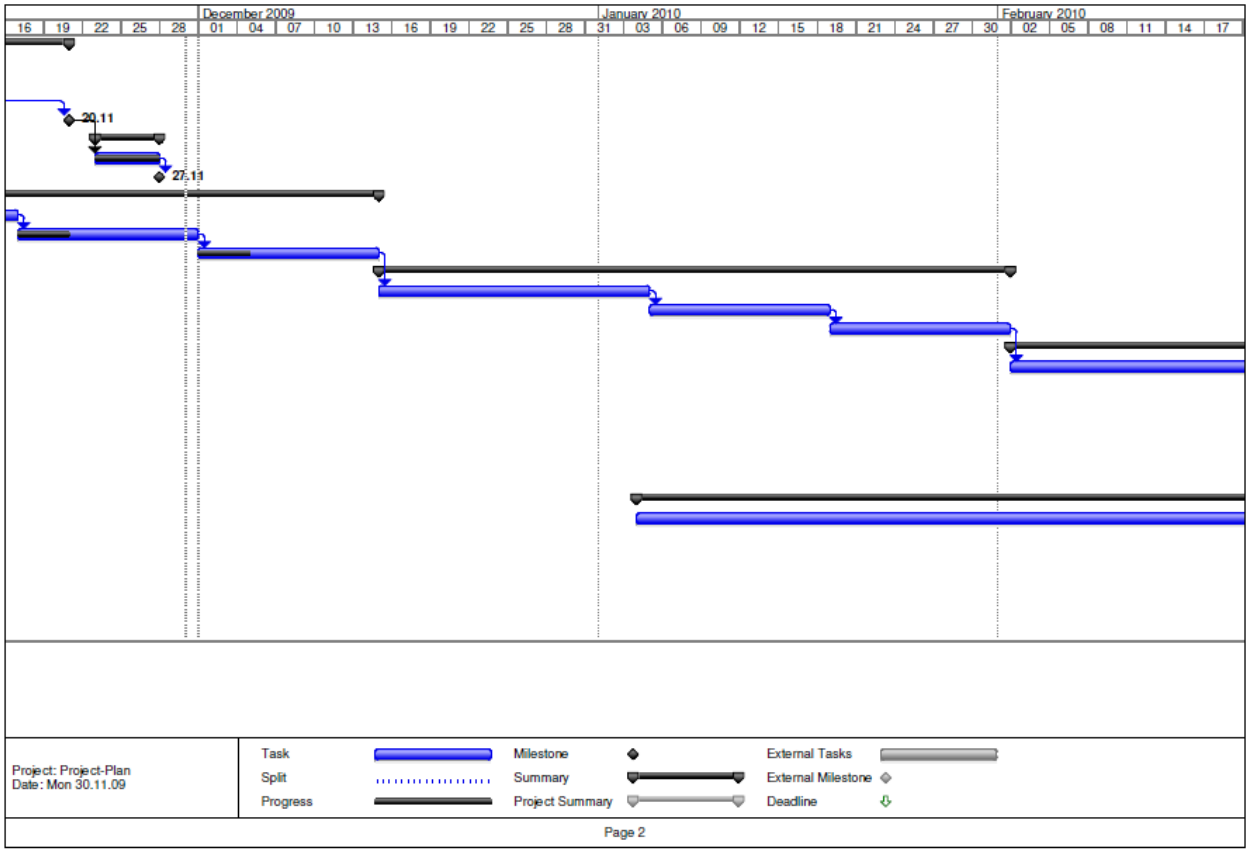
Factor name	Description	Is this a cost or a benefit	Estimated Amount	Estimate of when paid
Workplace	Place of work (e.g. office)	Cost	????	Montly
Salary	Salary	Cost	2500 €	Monthy
3D model	Higly detailed 3D model	Cost	200 €	Once, exact date doesn’t matter

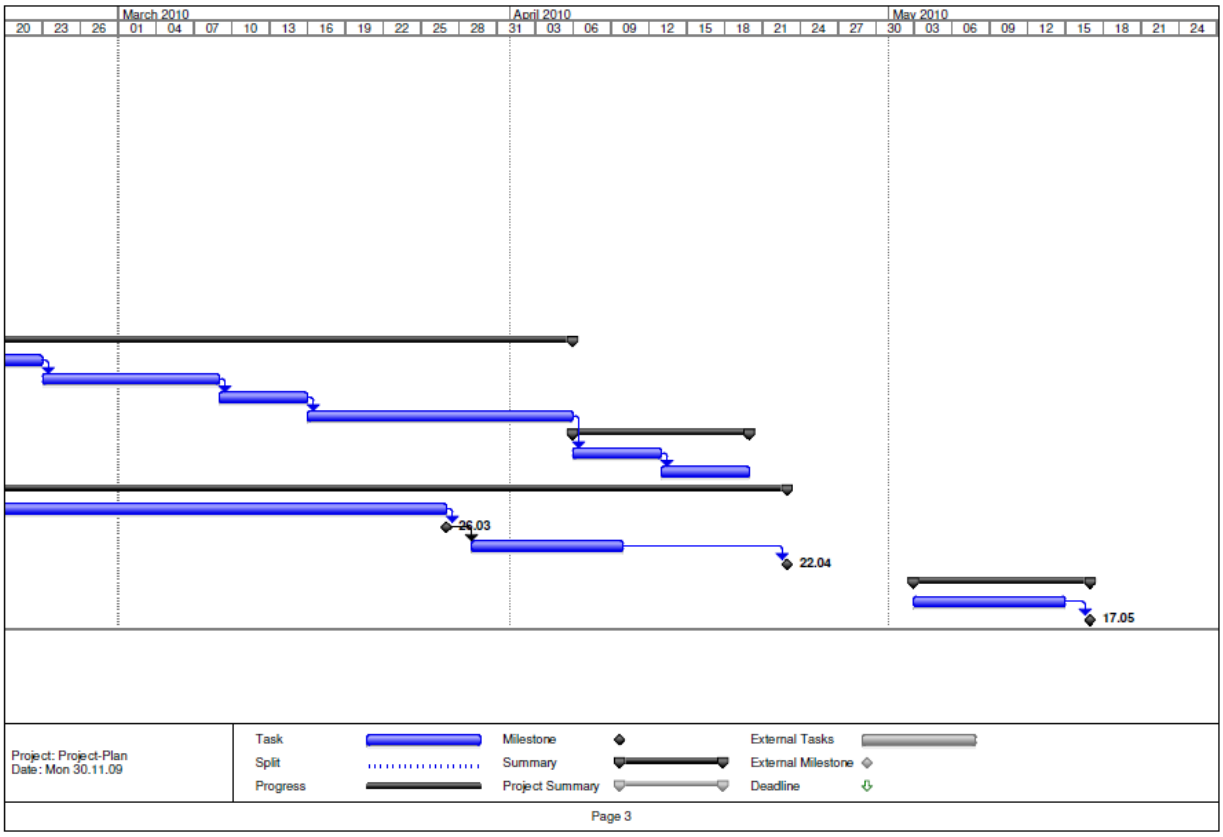
Commercial Conclusion

The components that will be developed cannot be exploited commercially directly by themselves. They could be used as a foundation for some kind of visualisation task which can then be marketed. If it were to be used to build a universal web-based sign language dictionary and sell advertisement or the collected data itself, it could be exploited commercially. This is not immediately a commercial idea, but since there would by no direct competition it could attract a lot of users and thus gain in worth.



ID	Task Name	Duration	Start	Finish
1	**Mini-Paper**	**24 days**	**Tue 20.10.09**	**Fri 20.11.09**
2	Work out topic for mini paper in detail	1 wk	Tue 20.10.09	Mon 26.10.09
3	Investigate and write mini paper	2 wks	Tue 20.10.09	Mon 02.11.09
4	Revise mini paper	1 wk	Tue 03.11.09	Mon 09.11.09
5	Deadline for mini paper	0 days	Fri 20.11.09	Fri 20.11.09
6	**Technical Plan**	**5 days**	**Mon 23.11.09**	**Fri 27.11.09**
7	Revise technical plan	1 wk	Mon 23.11.09	Fri 27.11.09
8	Deadline for technical plan	0 days	Fri 27.11.09	Fri 27.11.09
9	**Proof-of-Concept Prototypes**	**30 days**	**Tue 03.11.09**	**Mon 14.12.09**
10	Evaluate JMonkeyEngine	2 wks	Tue 03.11.09	Mon 16.11.09
11	Evaluate writing internal DSLs in Scala	2 wks	Tue 17.11.09	Mon 30.11.09
12	Load annotated 3D models	2 wks	Tue 01.12.09	Mon 14.12.09
13	**1st Prototype**	**35 days**	**Tue 15.12.09**	**Mon 01.02.10**
14	Control of simple movements	3 wks	Tue 15.12.09	Mon 04.01.10
15	Create basic movement description language	2 wks	Tue 05.01.10	Mon 18.01.10
16	Make movements look more natural (e.g. automatic easing)	2 wks	Tue 19.01.10	Mon 01.02.10
17	**2nd Prototype**	**45 days**	**Tue 02.02.10**	**Mon 05.04.10**
18	Extend description language with declarative commands	3 wks	Tue 02.02.10	Mon 22.02.10
19	Allow concurrent movements of the model	2 wks	Tue 23.02.10	Mon 08.03.10
20	Create a rough website prototype	1 wk	Tue 09.03.10	Mon 15.03.10
21	Extend language with syntax for concurrent movements	3 wks	Tue 16.03.10	Mon 05.04.10
22	**Optional enhancements**	**10 days**	**Tue 06.04.10**	**Mon 19.04.10**
23	Extend model with facial expressions (optional)	1 wk	Tue 06.04.10	Mon 12.04.10
24	Support for facial expressions (optional)	1 wk	Tue 13.04.10	Mon 19.04.10
25	**Final Report**	**79 days**	**Mon 04.01.10**	**Thu 22.04.10**
26	Write final report	3 mons	Mon 04.01.10	Fri 26.03.10
27	Target for draft report	0 days	Fri 26.03.10	Fri 26.03.10
28	Revise and print final report	2 wks	Mon 29.03.10	Fri 09.04.10
29	Deadline for final report	0 days	Thu 22.04.10	Thu 22.04.10
30	**Poster Session**	**10 days**	**Mon 03.05.10**	**Mon 17.05.10**
31	Prepare for poster session	2 wks	Mon 03.05.10	Fri 14.05.10
32	Poster session	0 days	Mon 17.05.10	Mon 17.05.10


December 2009
January 2010
February 2010
16 19 22 25 28 01 04 07 10 13 16 19 22 25 28 31 03 06 09 12 15 18 21 24 27 30 02 05 08 11 14 17
20.11
27.11
Project: Project-Plan
Date: Mon 30.11.09
Task
Split
Progress
Milestone
Summary
Project Summary
External Tasks
External Milestone
Deadline
Page 2


March 2010
April 2010
May 2010
20 23 26 01 04 07 10 13 16 19 22 25 28 31 03 06 09 12 15 18 21 24 27 30 03 06 09 12 15 18 21 24
25.03
22.04
17.05
Project: Project-Plan
Date: Mon 30.11.09
Task
Split
Progress
Milestone
Summary
Project Summary
External Tasks
External Milestone
Deadline
Page 3