Visual Programming of 3D Scenes

Semester Project
Contents

1 Introduction 3
  1.1 Overview .................................................. 3
  1.2 Semantic Information .................................. 3
  1.3 Visual Programming ................................... 5
  1.4 Goals of the project ................................... 6

2 OWL Information Extraction 7
  2.1 Jena Parser .............................................. 9

3 2D Representation / Data Manipulation 11
  3.1 Entities Representation ............................... 11
  3.2 Visual Libraries ....................................... 12
  3.3 3D Positioning ......................................... 12
  3.4 Virtual Human Features ............................... 13
    3.4.1 "Look at" action ................................... 13
    3.4.2 Key Frame Animation .............................. 14

4 3D Scene 15
  4.1 Java and C++ Sockets ................................. 15
  4.2 Communication Protocol .............................. 16

5 Evaluation and Discussion 18

6 Conclusion & Future Work 19
  6.1 Future Work ............................................ 19
    6.1.1 Taking Time into Account ....................... 19
    6.1.2 Walking Engine .................................... 20
    6.1.3 Collision Detection .............................. 20
  6.2 Conclusion ............................................. 21

7 References 22
1 Introduction

1.1 Overview

Creating a 3D scene or even composing a 3D scenario might not be the easiest and fastest thing to do. You will need to work with a graphical API (OpenGL, DirectX, etc.) or learn to use a 3D Engine that may ease the work. But that's not the hardest part! Unless the scene is empty, you will need to work with entities such as houses, plants, chairs or humans. They will need to be placed exactly where you want them to be and in the case of virtual “living” entities, doing exactly what you want them to do. Of course, such a task is far from impossible but may use a lot of time and resources.

This is where this project starts: with some a priori knowledge of semantic information concerning our virtual entities and using a visual programming paradigm, we will try to develop an application allowing anyone to setup easily any kind of 3D scene or scenario.

1.2 Semantic Information

One thing I need to explain first is this knowledge of the virtual entities. What is it and why is it needed in this application?

In order to answer the first part of the question, allow me to use an example: when you want to add a human in a 3D world, it will most likely consist of a certain amount of data. Maybe a file, describing the geometry of the person or a certain structure defined to represent triangles or points in space where the person will be. This information is necessary for the computer to display any kind of shape and it will also be a part of our a priori knowledge. But what about the name of the person, what about his or her gender, his or her weight? We would never talk about a pair of shoes saying the resolution of the laces is really high or the texture mapped into the outside is from a bmp file.

This kind of information is what we would use to describe a person. They are also important to complement the representation of the virtual character (we will see later how the application make use of such information to match animations with individual personality), but they’re not included in the geometry. That’s why they need to be incorporated in a kind of database. We will therefore use an ontology (knowledge representation) for our entities and also see shortly what is present in this ontology as well as how it is encoded.

Now, for the second part of the question, the answer lies in our motivation to make the process of 3D creation easier. People can work much faster with this kind of high level representation of information than with matrices or triangles. If the application provides a good interaction between those parameters (height, gender, position, etc.), anyone will be able to easily create a living 3D environment.

Finally, concerning the encoding of the information, the choice was made beforehand to use the OWL\(^1\) ontology language. The Ontology is a recent standardized way of representing information. It is used by the semantic web due to facility to decentralize information. Ontologies provides a higher structure level than

\(^1\)http://www.w3.org/TR/owl-features/
relational databases, as one can define concepts with properties, relationships between concepts and restriction rules (axioms). The Ontology Web Language (OWL) is one of many ontology languages available, based on earlier languages such as OIL, DAMN or DAMN+OIL and was build on top of RDF\footnote{http://en.wikipedia.org/wiki/Resource_Description_Framework} (a general method of modeling information) with a larger vocabulary and stronger syntax. This “machine-interpretable” language will carry the ontology of all our entities.
1.3 Visual Programming

Another really important model our application will use is Visual Programming. This paradigm is used when the user of an application, instead of specifying elements textually, will manipulate them graphically. It will help us again to tremendously simplify the task of creating a 3D world. For creating 3D environments, it is commonly necessary to have a lot of programming knowledge in C/C++ or Java for example. Moreover, someone would need to have basis of computer graphics. We would like to allow non experts to be able to create a 3D scene thanks to the integration of this paradigm. Instead of typing a few lines of code in order to load an object into the scene, you will simply drag and drop a box representing the object in the application. Instead of using pointers and variables to create a link between two entities, you will simply create an arrow with the mouse in between those two entities.

Everything will be done in order to provide the most easy-to-use interface while allowing the user as much freedom as possible in the creation process.

This approach of using “boxes and arrows” is not new and is also widely used for all kind of creation processes. One example which is used to process graphical data is Quartz Composer from Apple\(^3\). The interface which is displayed below will be the main model for our interface.

\[\text{Figure 1: Quartz Composer}\]

There exists softwares based on this paradigm to compose music, draw electrical logic schematics or even develop web applications.

\(^3\)http://en.wikipedia.org/wiki/Quartz_Composer
1.4 Goals of the project

In this project we will provide an application that allows a real time configuration of a 3D environment using semantic information and a visual programming paradigm. The next three chapters will each be dedicated to a certain section of the application: we will discuss the choice of technology for the section, the implementation process and the design decisions.

The reason behind this separation is that the process of creation is also split in three: you will first use an ontology file which will act as a database for all the objects on the scene then you will load in a 3D scene these objects and finally manipulate/configure them in a 3D world.

At the end we will also go over a few features that might be added to the application in order to improve the creation process or allow finer control over the scene. We will then discuss the result obtained with this application compared to a classical approach since we’re now inclined to use both.
2 OWL Information Extraction

The first step of our application development is getting the information from the ontology file. Everything concerning the entities is contained in an OWL file so the task here was to extract this information and store it in order to have a quick and easy access for the upper sections of the application.

We're using a simplified version of the Virtual Human Ontology defined in [3]. This project led to the creation of an ontology specially defined for Virtual Humans in order to make them more active and understandable for both humans and machines.

Only a few modifications occurred during the implementations to add objects in the OWL file.

Here is the graphical representation of an object:

Figure 3: Object Ontology

An object only has a few parameters to go with it: position in space, orientation and geometry. We could eventually create different classes of objects (housing, garment, food, etc.) in order to have more precised parameters but for simplicity we will consider only this model of object.

Now, since it would be uninteresting only to have objects in our scene and because I insisted on the importance of semantic information, I need to present the interesting part of the OWL file which contains the description of a Virtual Human:
There, we clearly have a more complex structure. I will not go into details for every class or link between classes (it is not the purpose of this section) but I will just highlight some of the parameters that are interesting for our application:

- First of all, I mentioned briefly the possibility to create 3D scenarios. For this to happen, we will most likely need movement and we will be able to add animation to our Virtual Human thanks to the “Animation” property.

- Also, the Virtual Human is not only described by a name, position and orientation but also morphological and emotional parameters. These parameters however will not be editable. (Changing the gender will have absolutely no impact on the geometry, for example.) But they will help us to organize or sort the data in an efficient way.

So, now that we have the structure of our ontology clearly defined, we will need to create a somewhat similar structure for our application, in order to store the information of the database. The way we’re going to proceed is that a class will be created for both our main entities (virtual human and objects) and each element of the hierarchy above will be either represented as a parameter or as
a class. Links will also be established between classes, so we can retrieve a particular information regarding an individual.

2.1 Jena Parser

We will now see how this parser was implemented. The programming language chosen to develop the application is Java. (However, for the last part of the project, C++ was used in order to support the 3D Engine but we’ll get into that later) There is a few reason behind this choice, but the main one would be because as far as we know, all development for ontologies are in Java. There is nothing done to work in other languages like C++.

Therefore, In order to accomplish the information extraction, I used the Jena framework\(^4\) for Java. Thanks to the API, understanding how the libraries interact with the owl ontology was easier and once I knew how to ask the file the information I needed to extract, the rest was simply to create an array of classes for each parameters and entities.

Here is a quick example showing how to extract the “Animation” class from the ontology:

1. First you need to regroup all information you know about the class: We must know its name (“Animation” in this case) as well as the name of its properties. Here we’re having two kinds of properties: simple variables such as “hasFile” and “hasDurationInSeconds”, and classes: “hasCategory”, “describesEmotion” and “hasMorphologicalDescription”.

2. When we go through the file, we must catch the classes who are named “Animation”. Once in this state, we create a new Java class Animation (the one we defined, not the one from the ontology) and we start defining the parameters. Simple variables are obtained by a “getProperty” function with the name of the property and for the more complex one, we will go in a loop looking, for example, at the emotions the animation describes. But, we won’t go down in the hierarchy yet! We only keep the name of the emotion(s) that interest(s) us.

There is also more thing you could do in this step, like looking if the structure of the ontology is respected by checking if the cardinality of a link is correct, or if a reference to a file is incorrect because the file doesn’t exist, etc.

3. When all classes are parsed, we can start creating the link between classes and not only keep the name as a reference.

We could be tempted to create the link present in the graph during the parsing, but since the individuals are not parsed orderly from top to bottom or bottom to top, we have to wait until every instance of classes is parsed and then we can check and create the links.

There is of course some simplification to be done: it is useless to create a class for each motion category (in the Virtual Human ontology). In this case, a simple boolean in the MotionCategory class for each “is-a” relation is sufficient.

\(^4\)http://jena.sourceforge.net/
One last thing needed was to make both the object and virtual human classes cloneable in order to allow multiple instances of such classes in the scene. Let’s suppose that in our OWL file, we have a stone. If we want to build a wall with the stone object, each stone will have different parameters. That’s why we need a bitwise copy of the object and not just a shallow one.
3 2D Representation / Data Manipulation

We now reach the core part of the application, the graphical user interface. This section will cover how the individuals are displayed and how the user will be allowed to manipulate them. Since we will define a short Visual Programming “language”, every manipulation will be graphical and a few rules will be set in order to deny faulty actions.

You may say that a picture is worth a thousand words, so before entering into details for each component, here is a quick snapshot of the interface:

![Image of the interface](image)

Figure 5: Visual Programming GUI

The screen is already filled with a few objects and a random scene, but every part of the interface will be explained in the following chapters.

3.1 Entities Representation

First thing first, we need to inform the user of the different individuals present in the database we just parsed. For this simple step, displaying the name of the entity correctly sorted in his class is sufficient. (We assume that the name is relevant enough for the user to differentiate two entities)

The result is showed in the first red square in figure 4: A simple JTree did the trick.
3.2 Visual Libraries

Simply showing entities would not be enough. We need to add, remove and play with those instances. To achieve this goal I was asked to look for a library that might facilitate the integration of visual elements in the interface. My choice went for the NetBeans Visual Library\(^5\) which offers support for graph-oriented modeling. The result can be seen on the second red square of the figure 4. Entities are represented in blue squares and can be freely moved around in the scene with the mouse. In order to lighten the representation, only a few basic parameters are displayed on those squares: name, class and geometry. The arrows and the other window inside the scene will be discussed in the following chapters.

The way the scene works is as follow: we have a main component, the GraphScene on top of which we add “Widget\(^6\) Layers”. Each of those layers will have its own purpose: adding widgets, making connections, defining the background, moving widgets, etc.

You might have guessed by now, the entities are widgets. Actually, the blue squares are a composition of a node widget (the top) and pin widgets (properties). Everytime an entity is added to the scene, all the necessary data belonging to the entity is copied and placed into an array. You can have multiple instances of the same object, as previously mentioned; only the name will change.

I would be lying if I say that integrating the so called “GraphScene” in the interface and insuring that the scene would only allow certain manipulations was easy, mainly due to the fact that not so many documentation and complex examples exist, but by the process of trials and errors, I’m proud to have reach a satisfactory level of interaction for the task at hand.

3.3 3D Positioning

Everything is well in the scene: we can now add and remove elements. But then, if we stop there we could only create a pile of object at the center of the world. We need to add rotation and translation on all axis! This way, we cover every orientation or position possible for any arbitrary entity.

Actually, one thing I omitted in the previous chapter was that the translation on the ground (plane XZ) is already implemented without any other buttons or scrollbars. One feature was to generate a position on the plane corresponding to the position of the widget in the scene. This way of moving the entities like we were seeing the scene from above created a stronger link between the 2D representation and the 3D representation which is more interesting because every user can now more easily move a widget in the environment without looking at the 3D world to confirm its position.

For the rest of the controls, they appear on the bottom left of the screen. We have three buttons representing the three possible axis of rotation, with a scrollbar to indicate the degree of rotation. There’s also a scrollbar for the vertical translation, since we already covered X and Z translations.

\(^5\)http://graph.netbeans.org/

\(^6\)http://en.wikipedia.org/wiki/GUI_widget
Everytime a modification is done on those parameters, the properties of the element is changed instantaneously and the change is reflected on the 3D world. (We will see how in the last part.)

3.4 Virtual Human Features

3.4.1 “Look at” action

Now that we’ve covered every possible position in 3D, we will shift our interest into more specific features only applicable to virtual humans. The first one we will talk about is the “look at” functionality. This feature was possible thanks to an already defined algorithm in the 3D engine. What was left to do is to make the link between this 2D representation and what we see in 3D. As you can see, the “look at” is represented by an arrow on the scene:

![Figure 6: Look at](image)

*Actually, the right part of the picture is not exactly representative of the left part, since the red_stone would be placed behind Brian, but it just to show how the connection is made.*

An arrow between two entities was the first graphical choice that came to mind in order to implement this functionality. It is probably how someone would quickly draw a person looking at something on paper. It is done thanks to the “connection layer” of the scene; the arrow itself is a widget. We just need to make some verifications before the connection is established since it is only valid between virtual humans and virtual humans and objects. The user only has to maintain the control key while moving the mouse to create a connection like this.
3.4.2 Key Frame Animation

The last feature implemented for the interface make significantly more use of the ontology of the virtual human. We talked about scenario and we've seen that a virtual human could possibly hold animations. That's why when you double click on a virtual human, you will see a small window appear on the screen, with the name of the virtual human and a list of animations. These animations are the one found in the OWL file and they are filtered specifically for the virtual human in question. Let me explain why: in figure 3 (the virtual human ontology), you can see that the virtual human has a morphological description and the animation tool. That's why, if they're not yet linked, we could check both morphological descriptions and allow only certain animations to be associated to a particular person. We can be more or less strict, but in the case of this application, we just made sure the gender is correct.

So, once you've associated one or more animations to a certain virtual human and you closed the window, the “Play” button on the top right part of the application will become available. If you press the button, the interface will freeze and the actors on the 3D part will play their animations until they are done. There is no special timing edition possible here, every virtual human will play his animation one after the other. We will discuss later what could be added to allow a more precise control of these scenarios.
4 3D Scene

We finally reached the last section of our application: the 3D world. So far, I’ve mentioned that the application would allow real-time configuration of a 3D world and that some modifications were reflected in this world. The main part of the program is developed in Java but for this part, we’re going to use the MVisio\(^7\) 3D Engine created here at the Federal Polytechnical School of Lausanne. Since the MVisio SDK is based on C++ architecture, this last part will be developed in C++ as well and we’re going to find a way for the two different programs to communicate.

Here is a picture showing the entire application:

![Visual Programming application](image)

Figure 7: Visual Programming application

We will focus our attention now on the “Visual Programming Viewer” running the MVisio engine.

4.1 Java and C++ Sockets

The first problem to solve was to establish communication. We have two different programs running in parallel, and we need real-time synchronization. For this, I’ve chosen to implement TCP Sockets on both applications. Since all the interaction with the user is done on the Java side and the C++ 3d world will only be there to display the result, we’re going to have one blocking\(^8\) Java tcp server and one non-blocking C++ TCP client.

\(^7\)http://vrlab.epfl.ch/~apeternier/mvisio/main.html

\(^8\)When a client or a server is in “blocking” mode, every time he sends a message, he will stop running until the message has been received on the other end. (Same thing when you receive, you stop until a message is available)
This choice was also made to simplify the implementation: instead of creating multiple threads, we can keep the C++ application simple and it will periodically check if there is a message waiting for it. If not, the application keeps running so that the user can navigate in the 3D world with his mouse and keyboard. On the other end, the Java server will make sure his message got through before letting the user continue his work. (I need to stress that the messages are very small, so even if the user keeps making changes, there will be no hiccups visible)

4.2 Communication Protocol

In order to make the two applications understand each other, I've created a few message "types" corresponding to a specific action. Those messages will be sent everytime a change occurred in the Java application. (Adding an object, moving it, adding a look at function, etc.)

Here is the list of messages along with their utility:

<table>
<thead>
<tr>
<th>Function</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quit the application</td>
<td>0</td>
</tr>
<tr>
<td>Add object/human</td>
<td>1(o/v)-name-path-file-orientation-posx-posz</td>
</tr>
<tr>
<td>Remove object/human</td>
<td>2(o/v)-name</td>
</tr>
<tr>
<td>Modify parameters</td>
<td>3o(o/v)-name-posx-posz-0</td>
</tr>
<tr>
<td></td>
<td>3y(o/v)-name-posy</td>
</tr>
<tr>
<td></td>
<td>3r(x/y/z)(o/v)-name-angle</td>
</tr>
<tr>
<td>Add/Remove animation</td>
<td>4(a/r)-name-path-file</td>
</tr>
<tr>
<td>Play the animation(s)</td>
<td>5</td>
</tr>
<tr>
<td>Add/Remove a &quot;look at&quot;</td>
<td>6(a/r)-source-target</td>
</tr>
</tbody>
</table>

Table 1: Messages

There is a few remarks I need to make about this list. Concerning the first message (quit), it is part of a synchronization problem. We have two applications in one, so you don’t want to launch or close twice the application while working. This is why the C++ application is launched at the same time as the Java application and is closed the same way. (We can run a command to execute the application, but to close it, it’s preferable to send a message telling the application to close instead of killing the application with another command.) Also, in order to avoid a synchronization nightmare, they both stay open while you work.

You should be familiar with the rest of the functions since we discussed them in the last chapter. We can also notice that it’s not necessary to send the whole object or human in the message: we only send what is necessary. When we load a human or an object, we only need his name, the name of the file holding the geometry and a position.

Concerning the parameters modification, we have one message for ground translations (on the plane XZ), one for vertical translations and one for each rotation. There’s also one last message to launch the animations (when you press play).

One more thing worth mentioning was a small difficulty that occurred during the implementation of a last feature in the program: the possibility to save a
scene and to reload it. Until then, the user was never fast enough to catch on
with the C++ application. What is implied here is that the C++ application
always has the time to process any message before another one is sent. (Unless
maybe the horizontal translations when the user moves the widget around for a
long time, but losing one or two coordinates in the process is far from dramatic
since what is important is that the entity is placed where the widget is at the
end.)

But, when the load functionality was implemented, the Java application had to
restore a whole scene as fast as possible, and sometimes one or two messages were
discarded because the C++ application was seeing two concatenated messages
and simply didn’t care about the second one, resulting in an avalanche of errors.
That’s why I had to enforce the C++ client to send a message informing the
Java server that he had done its work and he is now ready to receive a new
message.
5 Evaluation and Discussion

It is time now to see how the application behaves with different scenarios and users and if the project met our expectations. I will say, without taking too much risk, that the application does what it is supposed to do well. What has been implemented is working pretty fine and is quite easy to use. However, there is one limitation that bothered me during the creation of a few scenes (most of them are displayed in this document). The fact that we use blue squares to represent any entity and because they're almost the same size no matter what the object is, it might get confusing when the scene contains a very big object. We could take for example a house: we first add the house and then two virtual humans. If we suddenly feel the need to place the two virtual humans inside the house, we would have to go “blindly” by moving the two squares until we see that they're actually inside the house in the 3D representation. This limitation could be avoided by adding a richer representation of the scene. We could display in light blue an area representing the limit of the object as seen from above.

Aside from this limitation and a few feature that I wish I had the time to add, I believe the application is able to correctly handle any scene.

Now, concerning the goals of this project, again I believe that they were achieved. The application provides to the user (to a certain extend) a real-time configuration of a 3D scene using both semantic information and a visual programming paradigm. I would say that the most challenging task of this project was to choose and assemble every piece of this application together in order to provide to the user an intuitive and reactive interface. Every piece was essential to the whole application, whether it is the ontology or the sockets. The semantic information was efficiently stored and retrieved in the user interface (One could argue that we didn’t use all of the information at our disposal but once again, more features could be added that make use of those information). The visual libraries were a great help for the visual programming part and the sockets were correctly used to achieve the real-time constraint. Once this was done, we added layers of functionality for the creation (animations, look-at, etc.)

All in all, beside realising the limitations along with the possibilities for this kind of application, I’m quite happy about the result. I also hope that anyone using the application will feel the same too.
6 Conclusion & Future Work

6.1 Future Work

Now that we’ve seen every aspect of the application, I would like to quickly go through a few ideas that weren’t unfortunately implemented for this project, mainly due to time limitation or missing resources. So far, what we’ve done is creating all the necessary components in order to achieve the creation of a 3D scene by the means of Visual Programming. We’ve also added a few controls over our elements but achieving “total” control over the scene like a programming language would do would be time consuming and it wasn’t the scope of the project to create an application like 3ds Max\(^9\).

Without further ado, let’s jump into a few ideas that could allow more complex scenes creation.

6.1.1 Taking Time into Account

One of the first things that came to mind when we decided to use animations was to control when they would start. The way the application works as of right now is that all animations are played one after the other, and when multiple persons have animations, they would all start simultaneously. This, of course doesn’t tolerate much freedom for a scenario. (Unless you would create an entire animation for the whole scenario.)

\(^9\)http://www.autodesk.com/3dsmax
The way this feature would have been added is the following: instead of manipulating time textually, a “timeline” bar would be displayed at the bottom of the interface with one line for each person having an animation. Each animation file associated to a person would be displayed as a rectangle showing when the animation starts and when it finishes. You would be able to move this animation rectangle in order to modify those parameters.

6.1.2 Walking Engine

Another thing that might surprise you when you start creating a scene with this application is that the animations we talked about earlier are not locomotion animations. (Even though in the ontology, there is space for locomotion animations). So, our virtual humans are not nomads. Making them move around in the scene might not be as easy as it seems: first of all, we need to choose how the user will input the parameters for the movement. We need to know where the virtual human will go and how. The speed also needs to be taken into account and without already implemented algorithms or a walking engine, we would need to build the whole thing from the ground up. Speaking of ground, you wouldn’t like it if suddenly the virtual human goes through a wall or fly in the sky. This issue will be discussed in the next section, but seeing how many seemingly little things need to be taken into account in order to add movement, it was clear that this feature wasn’t even considered as a feasible add-on to the application.

6.1.3 Collision Detection

The last thing I’m going to discuss is a more generic feature. Although so far we’ve seen possible add-ons that would allow more freedom to the creation, this one is however limiting the possibilities. It might depend on what the user is creating, but maybe he won’t be too pleased about a virtual human embedded in a wall or one that could walk through doors. This is actually allowed because there is no control over the position of the entities and the 3D engine doesn’t mind merging objects. What we could do is make sure two entities don’t collide with each other. This would not be too hard on objects but could lead to complicated calculations with virtual humans. The geometry can be more complex and we have to be careful with animations! That’s essentially why we let the user rectify manually the colliding errors that might occur in the scenario.
6.2 Conclusion

The joint use of semantic information and visual programming to help the creation of 3D scenes showed promising results. I did find the process, although limited, really fun and easy. It is somewhat faster to accommodate to a certain visual programming paradigm than to learn a new programming language altogether. If the task to accomplish is in the boundaries of the application, I would not hesitate.

During these few weeks I also had the opportunity to touch different technologies and merge them. Whether in C++ or Java, I really enjoyed working to meet the expectations of my supervisors and I would really like to thank them for allowing me to work on this project, helping and guiding me along the way.

Thank you very much for your time!
7 References

[1.] Visual programming language,


[4.] MViSo (Mental Vision)
http://vhlab.epfl.ch/~apeternier/mvisio/main.html

[5.] OWL Web Ontology Language Overview
http://www.w3.org/TR/owl-features/

[6.] Jena Semantic Web Framework
http://jena.sourceforge.net

[7.] Quartz Composer
http://en.wikipedia.org/wiki/Quartx_composer

[8.] Netbeans Visual Library tutorials and examples
http://graph.netbeans.org/

[9.] Autodesk 3ds Max
http://www.autodesk.com/3dsmax