

MULTIMEDIA PROJECT

CONCEPT GRAPH APPLICATION

SERGIO GONZÁLEZ

JULY 9, 2008

SUPERVISOR:
PROF. ANTON ELIËNS

VRIJE UNIVERSITEIT AMSTERDAM

Contents

1	Introduction	2
1.1	What is a concept graph?	2
1.2	Concept Graph Application	3
2	Basic Concept Graph	4
2.1	ConceptGraphBasic.as	4
2.2	CharacterHandler.as	5
2.3	Character.as	7
2.4	Relation.as	8
2.5	data.XML	8
3	Medium Concept Graph	10
3.1	ConceptGraphMedium.as	10
3.2	CharacterHandler.as	12
3.3	Character.as	13
3.4	Relation.as	14
3.5	data.XML	14
4	Advanced Concept Graph	16

1 Introduction

1.1 What is a concept graph?

Many of you will be probably wondering what a concept graph is and for what it can be used. As the project is based on it, a definition becomes really necessary.

Wikipedia explains *concept mapping is a technique for visualizing the relationships among different concepts. A concept map is a diagram showing the relationships among concepts. Concepts are connected with labelled arrows, in a downward-branching hierarchical structure. The relationship between concepts is articulated in linking phrases, e.g., “gives rise to”, “results in”, “is required by”, or “contributes to”.*

Concept graphs are an easy way to communicate complex ideas. When there is too much information to be processed, sometimes is easier to handle if there is a complete visual interface where all the relations among the concepts are clear and easily accessible. This is the strong point of concept graphs.

Concept graphs are widely used: brain-storming, summarizing key concepts, instructional design, increasing meaningful learning, improving language ability, showing hierarchical structures, facilitating the creation of shared vision and shared understanding within a team and many more.

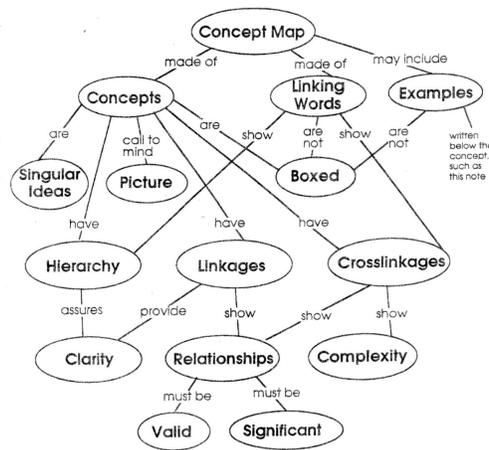


Figure 1: Example of a concept graph

This concept graph application tries to create a concept graph with lots of possibilities that can be fully adapted. All different kinds of information can be shown, such as video, photographs, images or text. User interface is also very important, so different characteristics such as zoom or drag & drop are available.

1.2 Concept Graph Application

At the beginning, the concept graph application was supposed to be an improved version of other currently available concept graphs (e.g. <http://der-mo.net/relationBrowser/>). Most of them were programmed in old versions of programming languages and they didn't include some features as video. Using the latest versions of an open-source programming language was another requirement for the project.

Flex Action Script 3.0 was a very new version of the programming language and with a lot of possibilities. This is the reason why this concept graph application has been made using Flex Action Script 3.0.

Nowadays, concept graphs are very interactive and dynamic thanks to multimedia. This concept graph application is an example of it. This manual contains all the information and help needed to adapt the concept graph to your own needs (You will need to download the compiler in case you want to re-program part of the application)- In order to be easily understandable, three different examples of three different concept graphs with different features are shown, from the easiest and simplest one to the most complex and powerful one.

2 Basic Concept Graph

The Basic Concept Graph (BCG) is the simplest application. It can be used to understand how the application works and to start “playing” with the source code.

The concepts are circles filled with different images or photographs. By clicking one character, the whole graph is redesigned showing the new characters and the relations among them and the clicked one (*relations*), and among themselves (*subrelations*).

The BCG is composed of the following files:

- ConceptGraphBasic.as
- CharacterHandler.as
- Character.as
- Relation.as
- data.xml

2.1 ConceptGraphBasic.as

ConceptGraphBasic.as is the main class. It reads all the information of the characters and relations from “data.xml”. This information is processed and stored in two arrays (*arrayChars* and *arrayRelations*) and they are sent to the CharacterHandler object (*handler*). In case of having problems reading the xml file it will report an error. This class also creates the *container* where all the characters will be placed.

The next source code indicates to load the data.xml file:

```
var loader:URLLoader = new URLLoader();
loader.addEventListener(Event.COMPLETE, handleComplete);
loader.load(new URLRequest("data.xml"));
```

The information stored about the characters is:

- *identification*: single name to identify the character by the application. Two characters cannot have the same identification name.
- *name*: name to identify the character by the user.
- *imageURL*: link to the character image.

This information is stored in the *arrayChars*.

```
var charData:Object = new Object();
charData.id = dataCharacter.@id;
charData.name = dataCharacter.@name;
charData.imageURL = dataCharacter.@imageURL;
arrayChars.push(charData);
```

The information stored about the relations is: who is related (*fromID*) and to whom is related (*toID*). This information is stored as a Relation object in the *arrayRelations*. The following source code shows how this is done.

```
var relation:Relation = new Relation(dataRelation.@fromID,
                                     dataRelation.@toID);
arrayRelations.push(relation);
```

If the user wants to add more information about the character or the relation, he has to modify the previous code adding the new fields.

2.2 CharacterHandler.as

This class is the most complex and difficult one to understand. It receives the *charData* and *arrayRelations* array from the *ConceptGraphicBasic.as*. It has to handle all this data to show the information on the screen. It handles the position and movement of all the characters and it process the different selections of the users.

The first customizable parameter is the position where the clicked character is going to be placed. This can be set in the *originX* and *originY* variables:

```
private var originX:uint = 400;
private var originY:uint = 365;
```

All the characters are shown in a circle shape, while the selected character is in the center of that circle. The radius of the circle (distance from the center to the first turn) can be set with the *radiusCircle* variable. The distance between the different turns can be set with the *distanceBetweenTurns* variable. The size of the character can be set with the *radius* variable. See figure 2.

```
private var distanceBetweenTurns:uint = 70;
private var radiusCircle:uint = 100;
private var radius:uint = 50;
```



Figure 2: Green arrow represents *radiusCircle*, red arrow represents *distanceBetweenTurns* and blue arrow represents *radius*

The function *overCharacter* shows the bigger character and the Bevel effect when the mouse is over it, while *outCharacter* restores the character to its default values. See figure 3.

The function *onEnterFrame* draws the lines connecting the related characters. The first step is cleaning the container of previous lines. Then, we draw the lines to connect the selected character with the related characters. Finally, we draw the curves to connect the characters with themselves, according to the subrelations. Different line widths have been set to distinguish the relations between the subrelations.

```

container.graphics.clear();
container.graphics.lineStyle(3);
for (var i:int = 1; i<arrayCharacter.length; i++) {
    ...
}
container.graphics.lineStyle(1);
for (var j:int = 0; j<subRelation.length; j++) {
    ...
}

```

The functions *click* and *checkExistence* processes the information of the characters, relations and subrelations when a new character is clicked.

Finally, the function *characterGeometry* sets the position of all the characters in the screen when a new one is clicked. When all the characters don't fit in the circle, a new turn is created. The number of characters that fit into the new turn gets bigger and bigger because the radius of the new circle is also bigger.

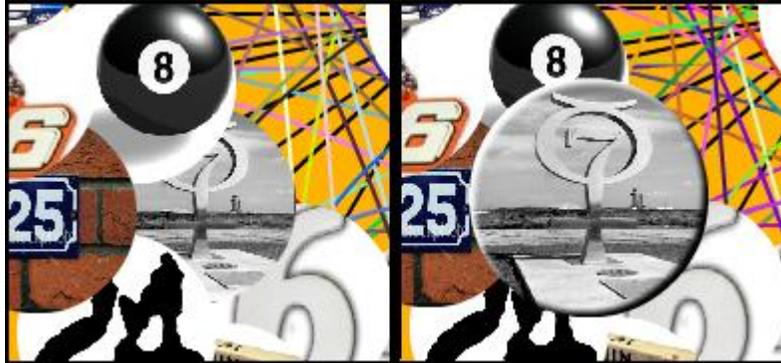


Figure 3: Mouse out and over the character

The number of characters that we want to be in the first circle can be set with the *numCharactersFirstTurn*. The variable *numCharactersFirstTurn* cannot be smaller than 3.

The number of characters in each turn gets increased by a power of *increaseCharactersFactor*. Recommended values are 2 or 3. The original value is 3. This means that, in the first circle we have *numCharactersFirstTurn* characters. In the second turn, we will have: $numCharactersFirstTurn + 3^2$. In the third turn there will be: $numCharactersFirstTurn + 3^3$, and so on. If the original values are not modified, we will have 15 characters in the first circle, 23 in the second, 39 in the third, ...

Again, here it is the code to modify these parameters:

```
private var numCharactersFirstTurn:uint = 15; //Minimum 3
private var increaseCharactersFactor:uint = 3; //2 or 3
```

2.3 Character.as

Character.as is the class that sets all the information, appearance and movements of the characters. In this basic application, the only information required to “create” a character is the *radius*, *id* and *imageUrl*.

```
public function Character (radius:Number, id:String,
                           imageUrl:String):void {
    ....
}
```

The function *onComplete* is executed when the image has been loaded. This function defines the shape of the character and “fills” it with the image. The default shape of the character is a circle,

but it can be changed easily to a rectangle or ellipse. The code to define the shape and fill it with the image is the following:

```
graphics.beginBitmapFill(bitmap,matrix);
graphics.drawCircle(0, 0, radius);
graphics.endFill();
```

Other functions such as *graphics.drawEllipse(0, 0, width, height)* or *graphics.drawRect(0, 0, width, height)* can be used to draw an ellipse or rectangle respectively.

Functions *increaseSize* and *decreaseSize* are called by the *CharacterHandler* when the mouse is over or out the character respectively.

Finally, the movement of the characters when a new one is clicked is programmed in the *onEnterFrame* function. A realistic and nice movement needed to be used, so a spring-movement was decided to be used. To customize this movement, to variables can be set: *spring* and *friction*. Here is the code with the default values:

```
private var spring:Number = 0.3;
private var friction:Number = 0.45;
```

2.4 Relation.as

Relation.as is a class with a formal description of how a relation is defined. In the basic example we are using only two attributes:

- *fromID*: sets who creates the relation
- *toID*: sets who is affected by the relation

In more complex applications, where more information about the relation is required, this class will need to be updated.

2.5 data.XML

The information about the characters and the relations is stored in a XML file. The Extensible Markup Language (XML) is a general purpose specification for creating custom markup languages that allows the user to define their own elements.

Here it is a portion of the XML file used in the basic application example:

```
<Example>
<Nodes>
<Person id="one" name="der_mo" imageURL="1.jpg"
  linkURL="http://lijit.com/users/der_mo"/>
<Person id="two" name="progressiveLoadingTest" imageURL="2.jpg"
  linkURL="" dataURL="moreNodes.xml"/>
<Person id="three" name="grahagre" imageURL="3.jpg"
  linkURL="http://getoutfoxed.com/users/grahagre"/>
.....
</Nodes>
<Relations>
<DirectedRelation fromID="one" toID="two"/>
<DirectedRelation fromID="one" toID="three"/>
<DirectedRelation fromID="one" toID="four"/>
.....
</Relations>
</Example>
```

This file is the most important one to be modified according to the user data to customize the concept graph. Only modify the data, the structure of the file should be kept as it is.

3 Medium Concept Graph

The medium concept graph contains the same features as the Basic Concept Graph, but with some new characteristics and functions.

In this example, the concept graph relates the famous singers who have collaborated with another famous singers. The relations have been done randomly, so it does not represent the reality, but it is enough for the example.

The new features of the Medium Concept Graph, compared with Basic Concept Graph, are the following:

- Zoom out/in with the wheelmouse to the characters. Very useful when there are too many or too less characters respectively
- Drag/Drop of the characters, so you can move the concept wherever you want
- Information display with customizable data: name, description, ...
- Possibility of including a video in the information display

The number and name of the classes remains the same as the Basic Concept Graph (except the main class that is named `ConceptGraphMedium`). We explain now the new functions and characteristics, so the user can adapt them to their own.

3.1 `ConceptGraphMedium.as`

The zoom feature is programmed in the function `scaleAt`. This function scales the area centered in `originX`, `originY` by a factor `scale`. This is the definition of the function

```
public function scaleAt(scale:Number, originX:Number,
                       originY:Number):void {
```

The `originX` and `originY` are the mouse position, so the zoom area depends on where the mouse is placed. The `scale` is the `zoomFactor` variable, that can be set in the following code:

```
private var zoomFactor:Number=21/20;
```

The `zoomFactor` must be slightly bigger than 1. The bigger it is, the more zoom is applied with the movement of the mousewheel.

The drag & drop function is very easy to implement. Two functions were create to handle the `MOUSE_DOWN` event and the `MOUSE_UP` event:

```

private function onMouseDown(ev:Event):void {
    container.startDrag();
}
private function onMouseUp(ev:MouseEvent):void {
    container.stopDrag();
}

```

The `MOUSE_DOWN` and the `MOUSE_UP` event are linked to the previous functions by the following `EventListener`:

```

stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);

```

The information display is configured in the `setDisplayInfo` function. There are two `textField` that indicates the name and the description of the selected character. The video is shown below the textfields. In figure 4 you can see a visual description of the variables. This is the code with the default values:

```

private var originInfoY:uint=50;
private var originInfoX:uint=975;
private var rectWidth:uint=350;
private var rectHeight:uint=600;
private var selectedLabelWidth:uint=300;
private var selectedLabelHeight:uint=20;
private var selectedLabelX:uint=25;
private var selectedLabelY:uint=130;
private var descriptionLabelWidth:uint=300;
private var descriptionLabelHeight:uint=150;
private var descriptionLabelX:uint=25;
private var descriptionLabelY:uint=160;
private var imageWidth:uint=100;
private var imageHeight:uint=100;
private var imageX:uint=125;
private var imageY:uint=15;
private var videoWidth:uint=300;
private var videoHeight:uint=240;
private var videoX:uint=25;
private var videoY:uint=330;

```

With all these variables, the user must be able to set the info display as he prefers, changing the size or the position of the elements. For example, if the user prefers to have the info display at the left

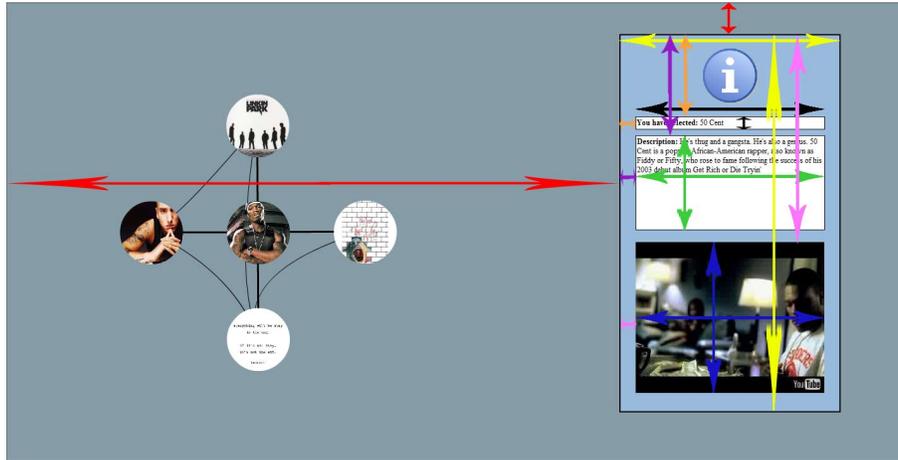


Figure 4: Red arrows represent *originInfoX* and *originInfoY*. Yellow arrows represent *rectWidth* and *rectHeight*. Black arrows represent *selectedLabelWidth* and *selectedLabelHeight*. Green arrows represent *descriptionLabelWidth* and *descriptionLabelHeight*. Blue arrows represent *videoWidth* and *videoHeight*. Orange arrows represents *selectedLabelX* and *selectedLabelY*. Purple arrows represents *descriptionLabelX* and *descriptionLabelY*. Pink arrows represent *videoX* and *videoY*

part, and the concept graph at the right part, he would only need to change the variables *originInfoX*, *originInfoY* in the *ConceptGraphMedium.as*, and *originX* and *originY* in the *CharacterHandler.as*. As we are need to read more information of the character from the XML file, we have to update the *XMLLoader* adding the new character fields. The updated function with the new fields is the following:

```
var charData:Object = new Object();
charData.id=dataCharacter.@id;
charData.nameChar=dataCharacter.@name;
charData.imageURL=dataCharacter.@imageURL;
charData.videoURL=dataCharacter.@videoURL;
charData.description=dataCharacter.@description;
arrayChars.push(charData);
```

3.2 CharacterHandler.as

Some changes according to the information display have been added to this class. Now, the constructor of the class also contains the textfields and the video references. The code of the constructor is the following:

```

public function CharacterHandler (charData:Array,
arrayRelations:Array, selectedLabel:TextField,
descriptionLabel:TextField, ns:NetStream):void {
    .....
}

```

Everytime the user clicks a character, all the information needs to be updated and displayed on the screen. This is processed in the *click* function. We can see in the following code how this information is processed and displayed:

```

public function click(characterClicked:Character): void {
    selectedLabel.htmlText = "<b>You have selected: </b>" +
        characterClicked.getName();
    descriptionLabel.htmlText = "<b>Description: </b>" +
        characterClicked.getDescription();
    if ((characterClicked.getVideoURL()==null) ||
        (characterClicked.getVideoURL()=="")) {
        ns.pause();
    } else {
        ns.play(characterClicked.getVideoURL());
    }
    .....
}

```

The textfield uses the *htmlText* to so formatted text. The video will not be shown either in case there is no videoURL or if it is empty.

3.3 Character.as

In the character class we have to include the new attributes of the character: name, description and video. We include this attributes in the constructor of the class:

```

public function Character (radius:Number, id:String,
    imageURL:String, nameChar:String, description:String,
    videoURL:String):void {
    .....
}

```

Other functions to return the name or description were also created:

```

public function getDescription():String {
    return description;
}
public function getName():String {
    return nameChar;
}

```

3.4 Relation.as

The relation amongst characters remains the same as in the basic concept graph, so no changes have been added to this class. However, in the advanced concept graph, this class will be renewed.

3.5 data.XML

As in the basic concept graph, all the data information is contained in the XML file. The structure remains the same, but new information has been added to the characters. This new information is the name of the character, a description, and a link to a Flash video. Relations remains the same.

This is a fragment of the data.XML file of the example:

```

<Example>
<Nodes>
<Person id="allsingers" name="All singers" imageURL="allsingers
.jpg" videoURL="" description=""/>
<Person id="50cent" name="50 Cent" imageURL="50cent.jpg"
videoURL="video/50cent.flv" description="He's thug and a
gangsta. He's also a genius. 50 Cent is a popular African-
American rapper, also known as Fiddy or Fifty, who rose to
fame following the success of his 2003 debut album Get Rich
or Die Tryin' "/>
<Person id="aliciakeys" name="Alicia Keys" imageURL="aliciakeys
.jpg" videoURL="video/aliciakeys.flv" description="Alicia Keys
is an American R&B/soul singer, composer, record producer, and
pianist. She released her debut album Songs in A Minor in 2001,
the title being a reference to both her classical aspirations
and to the fact that she wrote most of the songs at a very
young age."/>
....
</Nodes>

```

```
<Relations>
<DirectedRelation fromID="allsingers" toID="50cent"/>
<DirectedRelation fromID="allsingers" toID="aliciakeys"/>
<DirectedRelation fromID="allsingers" toID="amywinehouse"/>
....
</Relations>
</Example>
```

Once again, this file has to be customized to meet the customer needs. The structure must remain the same, but the information should be updated.

4 Advanced Concept Graph

It's coming...