

ARTIFICIAL INTELLIGENCE TECHNOLOGIES, S.L.

aiSense Flash Interactions

PROGRAMMING GUIDE

www.aitech.es



VERSION 1.6

Contents

Preface	4
1 Introduction	5
1.1 aiSense architecture.....	5
1.2 Conventions.....	6
1.3 About aiSense examples	6
1.4 Using the aiSense interface library in a Flash project	7
1.5 Organization of this document.....	9
2 Connection to the aiSense server	10
3 Basic presence and gender detection	12
3.1 Presence detection.....	13
3.2 Gender detection	17
3.3 Other considerations.....	17
3.3.1 Id property.....	17
3.3.2 Showing a background video	18
4 Presence detection in real time	20
4.1 Updating properties in real time.....	21
4.2 Scaling objects.....	22
4.3 Other properties.....	23
4.3.1 Rgb property.....	24
4.3.2 Frontal property	24
5 Motion regions.....	26
5.1 A basic dynamic region.....	29
5.1.1 Definition of regions.....	29
5.1.2 Management of regions	32
5.2 Using multiple motion regions.....	34

5.2.1	Identifying an aiRegion.....	35
5.2.2	Creating/Removing multiple regions	36
6	Advanced example	38
6.1	Creating an interactive menu.....	39
6.2	Velocity property.....	43
	References.....	44

Preface

This manual is a tutorial for version 1.6 version of aiSense Flash Software Development Kit (SDK) from Artificial Intelligence Technologies (AITECH) [1].

To best understand the techniques described here, you should be familiar with ActionScript 3.0 (AS3) [2] or, at least, with basic object-oriented programming concepts such as classes, inheritance or polymorphism.

In addition, the tutorial introduces basic ideas related to the access of data through the aiSense Flash SDK from the aiSense computer vision modules in a logical order and progresses steadily. Therefore, it is recommended to follow the order of the chapters.

1 Introduction

1.1 aiSense architecture

aiSense provides an architecture for supporting computer vision-based interactivity in multimedia systems, as depicted in the following figure.

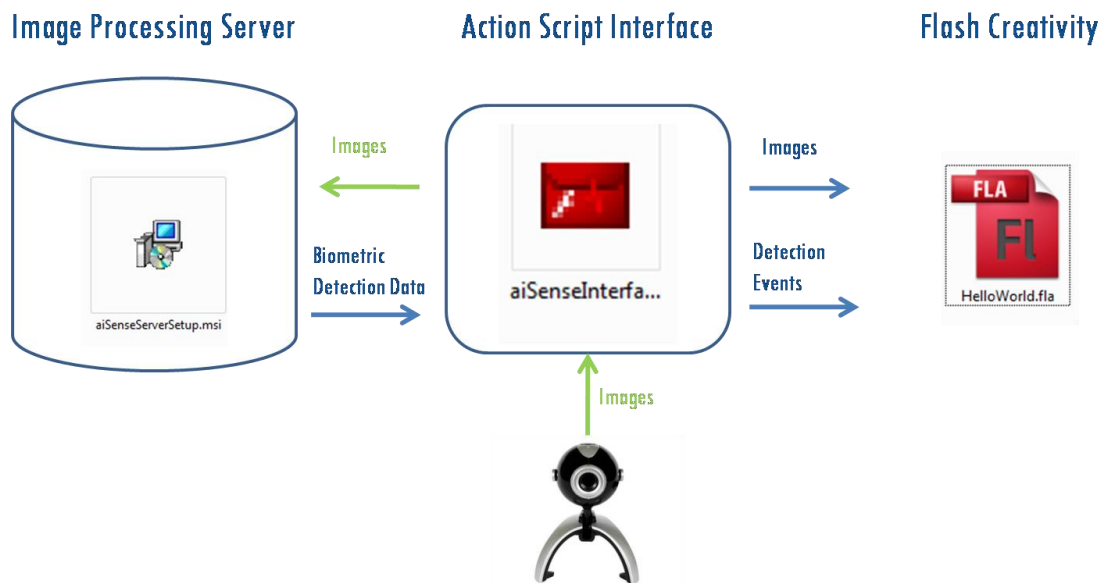


Figure 1.1: aiSense general architecture

This architecture is composed by the aiSense Server, the multimedia content (Flash creativity), the aiSense Flash interface and a web camera. The Flash interface obtains an image stream from the camera and sends it to the server for its processing. Then, the server detects biometric patterns in images of people, such as the location of their faces, their genders or movements, and sends them back to the interface so that Flash designers can use these data in their multimedia content.

This guide addresses the Flash SDK interface of the system by providing a set of examples which show how to use all biometric data obtained from the aiSense computer vision modules. These data are related to the following aiSense features:

- Presence and gender detection.
- Motion detection in defined regions.
- Color detection of people upper outfits.

1.2 Conventions

This document makes special use of computer voice (represented typically in `Courier` font), text in italics, bold and underlined text so as to discuss classes, methods and other programming elements. Computer voice denotes methods of classes or characters that are to be taken literally (typed as they appear). Text in italics denotes words that represent names of main examples and programming classes. Bold font indicates properties of some aiSense interface class. Underlined text combined with bold and italic fonts denotes a stage or step inside of a more complex procedure.

For instance, the syntax:

1.1) Removing a region: an *aiRegion* object is removed from the internal list of aiSense by using its `removeRegion()` method. Use its **definedRegion** to check the list of *aiRegion* objects which is maintained by the aiSense interface.

, means that **1.1) Removing a region** is a process stage, *aiRegion* is a class, `removeRegion()` is a method and **definedRegion** is a property.

AS3 code is shown in figures to separate it visually from the normal literature of this manual.

1.3 About aiSense examples

This aiSense interface guide aims to describe all concepts related to obtaining all presence and motion data through a set of visual examples. The code of these examples may be found in the attached folder called *Tutorial_Examples*, which contains the following instances:

1. BallGender_Presence
2. BallGender_RealTimePresence
3. BallGender_ShirtColor
4. FlyingRegion
5. InteractiveMenu
6. FootballHeading

There are two sub-folders in each example folder called *Source_Code* and *Executable*. The former has the source code written in AS3. The second contains the SWF file to be run in Flash Player and all necessary assets attached (images, sounds, etc). Likewise, the *GreenSock* library is used to manage animations for both *InteractiveMenu* and *FootballHeading* examples, while keeping the AS3 code as clean as possible. This library can be obtained from its original web site [3].

Moreover, every aiSense example has the same first image at the beginning of their execution (Figure 1.2). This first picture indicates the initial state of the application. When there are no detections, all examples shown in this tutorial go back to their initial state.



Figure 1.2: Initial state for the aiSense examples

Finally, note that there is an HTML documentation about the aiSense interface where all classes, methods and properties are described. This reference can be found in the folder called *HTML_Reference* which is placed in the same *Documentation* folder as this manual.

1.4 Using the aiSense interface library in a Flash project

The aiSense interface library is compressed in a SWC file called *aiSenseInterfaceLibrary_1.6.swc*. You may find this library in the *aiSense_Library* folder attached. The aiSense library must be referenced in the platform that we are working, such as *Adobe Creative Suite* [4] or *FlashDevelop* [5]. In our case, we describe the library path setting in the Adobe platform.

To set the document-level Library path (Figure 1.3):

1. Create a new Flash project and select *Public Settings* option of the *File* menu option.
2. Make sure *ActionScript 3.0* is specified in the *Script* pop-up menu and click *Settings*.
3. In the dialog box called *Advanced ActionScript 3.0 Settings*, click the *Library path* tab.
4. Add the path where the *aiSenseInterfaceLibrary_1.6.swc* is placed to the path list. You may add folders or individual SWC files to this list.

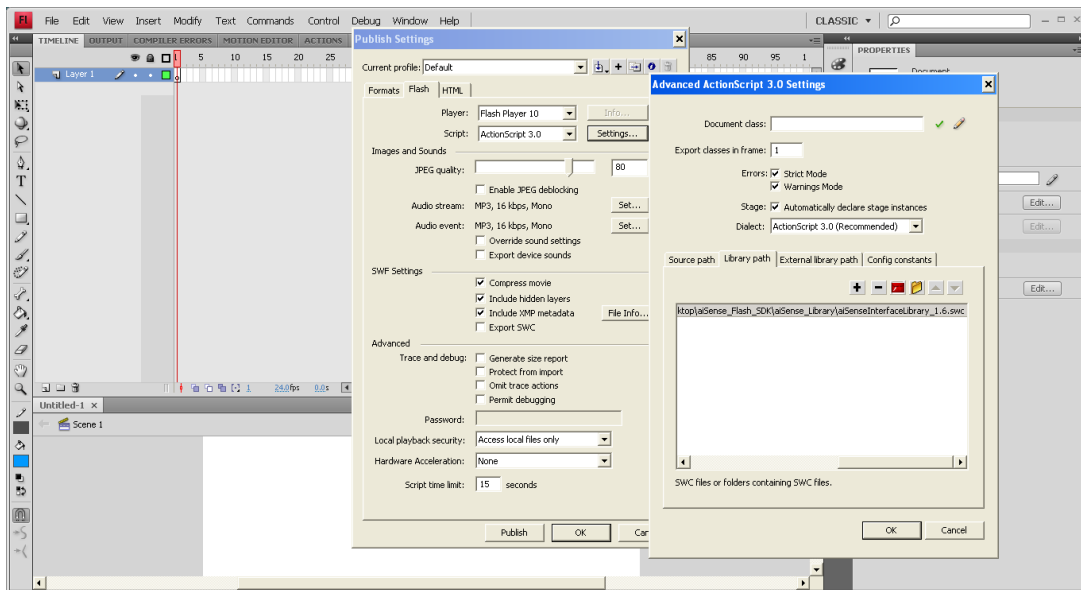


Figure 1.3: Setting the document-level aiSense library path

To set the application-level source path (Figure 1.4):

1. Choose *Preferences* option in the *Edit* menu option and click the *ActionScript* category.
2. Click the *ActionScript 3.0 Settings* button and add the path to the *Library path* list.

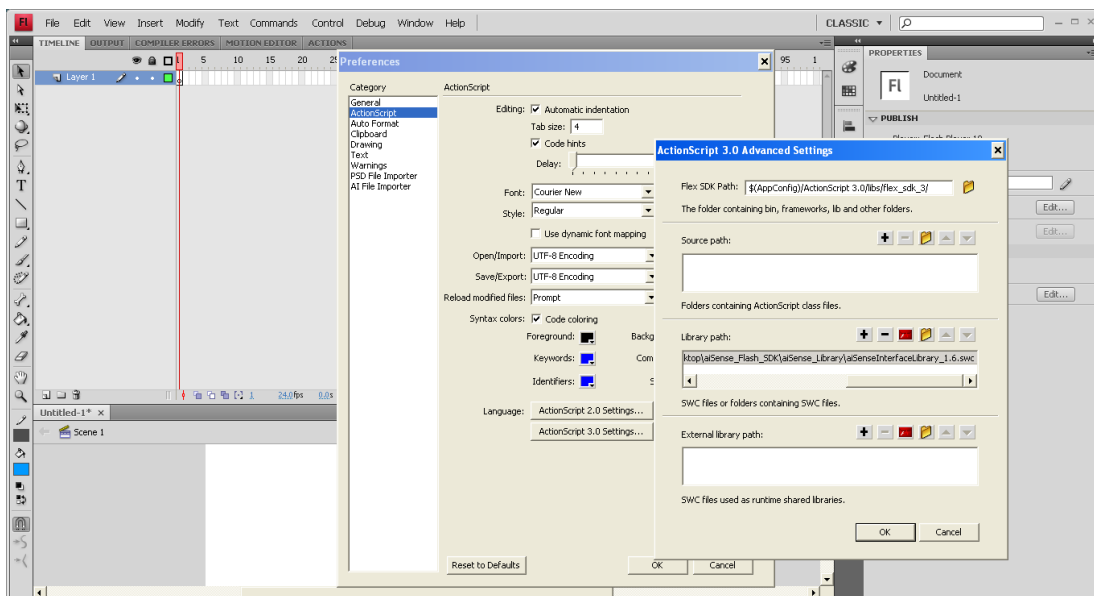


Figure 1.4: Setting the application-level aiSense library path

1.5 Organization of this document

This guide is organized according to the features of the aiSense architecture in the following sections:

- Section 2 deals with the connection to aiSense server so as to get access to all needed information.
- Section 3 introduces a first sight about the use of the aiSense Flash interface for obtaining basic presence and gender detection.
- Section 4 further develops the previous concepts and adds others to build Flash graphic objects following the trajectory of detected people faces.
- Section 5 introduces motion region features.
- Section 6 addresses advanced examples for showing other possibilities than can be developed by using the aiSense Flash interface.

2 Connection to the aiSense server

According to the architecture presented in the previous chapter, the first stage for any interactive multimedia content is starting the communication with aiSense server, which is the provider of biometric patterns from the image stream of the camera sent by the aiSense interface. In this chapter we address how to establish the connection to aiSense server in order to provide it with the image stream from Flash and obtain data back from the aiSense computer vision engine.

Thus, for any application, we shall begin by using the *aiSenseObject* class (by instance or inheritance). This class represents the entity which manages the detections returned by the aiSense server. To start the connection, the *aiSenseObject* instance must be added to the stage (Flash scenario). The *aiSenseObject* will automatically establish the communication with aiSense server (Figure 2.1).

```
public class Main extends MovieClip{  
  
    public function Main(){  
        // This class gets the face detection events thrown by aiSense server.  
        // It inherits from aiSenseObject class.  
        var manager:CBallManager = new CBallManager();  
        addChild( manager );  
    }  
}
```

Figure 2.1: Connection to aiSense server is established when the *aiSenseObject* instance is added to the stage

In the previous code, the *CBallManager* object inherits from the *aiSenseObject* class. Hence, we just need to add this object to the stage to establish the connection. The following figure shows the application state when a run-time exception is thrown.

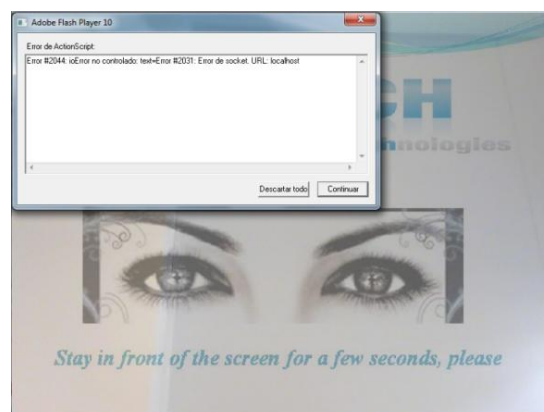


Figure 2.2: Run-time exception window

Generally, there are four possible reasons for obtaining a connection error:

- **Denied access:** The user refuses to allow the SWF file accessing to his camera. So the aiSense interface will throw a *StatusEvent* exception.
- **Input/Output (I/O) error:** When an I/O internal error is produced, it causes the failure of an internal send or receive operation. The interface throws an *IOErrorEvent* exception.
- **No camera:** If the user does not have a camera installed, the interface will throw an *ErrorEvent* exception.
- **Server is off:** The aiSense server is not running. The interface throws an *IOErrorEvent* exception.

For more information about these run-time exceptions, see the HTML documentation of the interface or the documentation from Adobe.

Furthermore, the *aiSenseObject* class provides a **connection** property. This property is used to set some connection parameters, which are:

- Setting the camera capture mode: resolution and frame rate. The `setCameraMode()` method of the *aiSenseObject* is used to set these parameters.

Note that not all resolutions are supported by all cameras. If the requested resolution is not supported by the current camera, it will default back to the most similar resolution available, or to a basic default resolution established by the camera itself.

- Enabling/Disabling face detection by calling the `setFaceDetection()` method of *aiSenseObject*. `true` value passed as parameter will enable face detection, `false` will disable it.
- Enabling/Disabling the mirror effect by using the **mirror** property of *aiSenseObject*. `true` value will enable mirror and `false` will disable it.
- Closing connection and reconnecting to aiSense server, using the `close()` and `reconnect()` methods of *aiSenseObject*.

Although the HTML documentation (*HTML_Reference* folder) describes these methods in detail, the following section shows a usage example (Figure 3.3). The rest of the tutorials consider that the connection step is already done for all the examples.

3 Basic presence and gender detection

aiSense interface contains a set of classes and properties to build interactive systems from scratch. This section describes the most basic features: presence and gender detection. Other relevant aspects such as specific properties and the inclusion of video are also discussed.

Presence and gender detection is based on face detection. Hence, we know that someone is in front of the camera when we obtain his/her face detected from aiSense server. The *aiSenseObject* and *aiSubject* classes are used in order to achieve this functionality. The *aiSubject* class contains all properties corresponding to a face detected and the color of the upper outfit of the person. Besides, it is managed by *aiSenseObject* through *aiSubjectEvent*-type events, which are dispatched into the event flow whenever subject detection - face detection - occurs.

In general, the recommended steps in order to use aiSense classes to obtain the information of detected faces are:

- 1) Listening to face detection events.
- 2) Setting up the *aiSenseObject* class.
- 3) Obtaining the subject list from *aiSubjectEvent*.
- 4) Using the *aiSubject* properties to access information about detections.

The example followed in this chapter is called *BallGender_Presence* and is composed by three files:

- Main.as: Contains the entry point of the application.
- CBallManager.as: Contains the *CBallManager* class and inherits from the *aiSenseObject* class in order to get detections and to create balls with a matched text.
- CBallGender.as: *CBallGender* class implements a small ball with a matched text label which has properties to change its state (position, color, etc).

The `main` function was shown in Figure 2.1 as an example of connecting to aiSense server. This application creates a ball with a matched text label when someone is in front of the camera and places it in the image position of the detected face. Besides, it detects the person's gender, changing its color and writing the gender value (male or female) in the text label.

The following picture shows an output of the example we are explaining in the following sections.



Figure 3.1: Output of *BallGender_Presence* example

This example is developed incrementally to describe the concepts of presence and gender detection in the aiSense system.

3.1 Presence detection

The basic scenario to build an interactive application through aiSense is detecting when someone is in front of the system. This feature will often be the starting point in order to run the interactive application.

The following is a step-by-step description of the general process mentioned above and is applied to the *BallGender_Presence* example.

1) Listening to face detections.

aiSenseObject objects, once they are connected to aiSense server, notify a set of events for informing that they have got detections. These events are divided into two kinds depending on the detection type: subject (face) or region detection. This example focuses in the first, so we shall get *aiSubjectEvent.SUBJECT_DETECTION* event types.

In order to receive event notifications, our manager class must listen to this kind of event by adding an event listener at the beginning. In the following example, the event listener is placed in the constructor function.

```
public function CBallManager(){  
    // Listening to aiSense event for face detection.
```

```

// detectPresence() method manages all data sent by aiSense server.
addEventListener( aiSubjectEvent.SUBJECT_DETECTION, detectPresence);

// The initial image is loaded when the manager is added to the root (stage).
addEventListener( Event.ADDED TO STAGE, init);

this.name = "managerBall";
m_aBall = new Array();
m_Loader = new Loader();
m_bInitialState = true;

// This timer is activated when there is no subject (face) for X ms.
m_RestartTimer = new Timer(TIME_FORINITIALSTATE);

// The timer calls toInitialState() method in order to re-load the initial image
m_RestartTimer.addEventListener( TimerEvent.TIMER, toInitialState);
}

```

Figure 3.2: *CBallManager* constructor

2) Setting up the aiSenseObject class.

Besides starting the connection, it may be required to set some initial parameters. This setting is achieved through the **connection** property of the *aiSenseObject* class. The **connection** property is a reference to the internal connection class and has public methods and properties to change settings such as the resolution of images obtained from the camera (`setCameraMode()` method), the mirror effect (**mirror** property), enable face detection (`setFaceDetection()` method) and so on. In this case we set the camera mode and the mirror effect since face detection is enabled by default. The camera resolution is set to be the same as the stage size.

```

private function init( evt:Event):void {

    ////////////////////////////////////////////////////////////////////
    //// NOTE THAT THE CONNECTION PROPERTY IS AVAILABLE WHEN THE
    //// AISENSE OBJECT IS ADDED TO THE STAGE
    ////////////////////////////////////////////////////////////////////

    // Sets the camera capture mode to the native mode that
    // best meets the specified requirements.
    this.connection.setCameraMode( stage.stageWidth, stage.stageHeight, 15);

    // Sets mirror effect
    this.connection.mirror = true;

    loadImage();
    m_bInitialState = true;
}

```

Figure 3.3: `init()` method of *CBallManager*

3) Obtaining the subject list from aiSubjectEvent.

Now, we need to implement the function which is listening to the *aiSubjectEvent* events. This method is activated when an *aiSubjectEvent.SUBJECT_DETECTION* is notified from the *aiSenseObject* class.

As a result, we asynchronously obtain *aiSubjectEvent* objects. These objects have a list of detected faces which are encapsulated in *aiSubject* objects. Then, we use the `getSubjects()` method to access the list of subjects (faces).

```
public function detectPresence(e:aiSubjectEvent):void{

    var subjects:Array = e.getSubjects();
    if( m_bInitialState )
        removeInitialState();

    m_RestartTimer.stop();

    //////////////////////////////////////
    // REMOVING BALLS WITHOUT SUBJECTS MATCHED
    //////////////////////////////////////
    for ( var i:uint = 0; i < m_aBall.length; i++){

        var subjectForBall:aiSubject = m_aBall[i].subject;
        // If there is no ball matched with subject but it exists on the scene, then remove it.
        if ( !isObjectAlive( subjectForBall.id, subjects ) ) {

            if( m_aBall[i].parent != null){
                this.removeChild(m_aBall[i]);
                // splice() changes the array's length
                m_aBall.splice(i, 1);
                // Decreases the counter because the array's length is changed.
                i--;
            }
        }
    }
    //////////////////////////////////////
    // CREATING BALLS
    //////////////////////////////////////
    for ( i = 0; i < subjects.length; i++){

        // Drawing the ball if there is none for a detected subject.
        if ( !isObjectAlive(subjects[i].id, m_aBall) ) {

            var ball:CBallGender = new CBallGender(subjects[i]);

            this.addChild(ball);
            m_aBall.push(ball);

        }
    }
    m_RestartTimer.start();
}
```

Figure 3.4: `detectPresence()` method of *CBallManager*

The rest of the previous code creates a *CBallGender* object, which is just a ball with a text over it, by giving the detected face to it. Moreover, the `id` property is used by the `isObjectAlive()` method in order to manage the set of objects by comparing the `id` of new detected faces with the previous ones. We deal with this property in the section 3.3 *Other considerations*.

4) Using the aiSubject properties to access detections.

CBallGender is a class which draws a ball and a text label on it. This class uses an *aiSubject* object in order to change its position and to place the ball on top of the detected face's image. The following code shows the constructor function of *CBallGender* and the `init()` method to start its tasks.

```
public function CBallGender( subject:aiSubject ){

    m_Subject = subject;

    // init() method is activated when a CBallGender instance
    // is added to the stage.
    addEventListener( Event.ADDED TO STAGE, init);

    // Draws a circle in the Sprite
    this.graphics.beginFill(0xFF0000);
    this.graphics.drawCircle(25, 25, 70);
    this.graphics.endFill();

    // Text field
    m_TextField = new TextField();
    m_TextField.border = true;
}

private function init( evt:Event):void {
    // The ball has a text for showing the person's gender.
    loadTextField();
    // Dimension and position are fixed in every moment.
    setDimension(m_Subject);
    setPosition(m_Subject);
}
}
```

Figure 3.5: *CBallGender* constructor

The ball dimensions are fixed, so we focus on its position. We have to access the *aiSubject* properties in order to obtain the position of the detected face. Also, note that these properties are normalized in a range between 0 and 1, therefore, we need to multiply them by the stage size.

This is the code to access the **x** and **y** properties of the *aiSubject* object:

```
public function setPosition( subject:aiSubject):void {

    this.x = (subject.x + subject.width/2 ) * stage.stageWidth;
    this.y = (subject.y + subject.height/2) * stage.stageHeight;

    // m_TextField's coordinates relative to the local coordinates of its
    // parent DisplayObjectContainer (CBallGender).
    m_TextField.x = -70;
    m_TextField.y = -40 - m_TextField.height;
    m_TextField.setTextFormat( new TextFormat("Times New Roman",50, 0x000000,
true,null,null,null,null, TextFormatAlign.CENTER) );
}
}
```

Figure 3.6: *setPosition()* method of *CBallGender*

Position properties are also normalized between 0 and 1, and represent the top-left corner of the bounding box that contains the detected face.

3.2 Gender detection

In this section we continue developing the example previously described and add another behavior to the *CBallGender* object: its color changes depending on the detected person's **gender**. This property has two possible values: *male* and *female*.

Consequently, we place the `setColor()` method in the `init()` function, as shown in the following code:

```
private function init( evt:Event):void {
    // The ball has a text for showing the subject's gender.
    loadTextField();
    //Color depends on the m_Subject's gender.
    setColor(m_Subject);
    // Dimension and position is fixed in every moment.
    setDimension(m_Subject);
    setPosition(m_Subject);
}
```

Figure 3.7: `init()` method of *CBallGender*

The `setColor()` method changes the ball color depending on the detected person's **gender**. If the person is *male*, the color is changed to blue and if the person is *female* the color changes to pink.

```
public function setColor( subject:aiSubject ):void{
    // Changes the complete container's color
    // (so both ball and text color will be changed).
    var newColorTransform:ColorTransform = this.transform.colorTransform;
    if( subject.gender == "male"){
        newColorTransform.color = 0x87CEEB; // blue
        m_TextField.text = "Male";
    }
    else{
        newColorTransform.color = 0xFFC0CB; // pink
        m_TextField.text = "Female";
    }
    this.transform.colorTransform = newColorTransform;
}
```

Figure 3.8: `setColor()` method of *CBallGender*

3.3 Other considerations

3.3.1 Id property

aiSense server assigns an **id** to every detected face. This **id** is a one-based index and is kept in the system until the face is lost (it is not detected by the server, either because the person stopped looking towards the camera for a certain time, or because the person went out of the camera field of view). Also note that, after a face is lost, when the server detects again the same face, a new **id** is assigned, which may be different that the last **id** assigned to the same person before.

We already pointed out the `isObjectAlive()` method of the `CBallManager` class in section 3.1 to manage a set of detected faces by using the `id` property (see figure 3.4). Now, we show how this function works internally.

This method checks if there is an object in the array which has the given `id`. Specifically, the function is used in two cases:

- to check if there is already an object (`CBallGender`) which has a detected face (`aiSubject`) with the same `id` as any of the faces returned by aiSense server.
- to verify if there is any new detected face without a matching `CBallGender` object.

The first case is used to remove `CBallGender` objects which do not have a subject with updated `id`, that is, we check whether the ball keeps the same detected face as it had in previous face detections or not. The second case creates a new `CBallGender` object for any new detected face if this one does not have any matching `CBallGender`.

Therefore, we just create instances of `CBallGender` when there is a detected face with a new `id` which we had not before, and we also remove balls for subjects whose `id` does not appear.

The following listing shows the code that implements both cases.

```
private function isObjectAlive( id:uint, objects:Array ):Boolean {  
    var objectItem:*;  
    for each( objectItem in objects) {  
        if ( (objectItem is CBallGender) && objectItem.subject != null &&  
            objectItem.subject.id == id )  
            return true;  
    }  
    for each( objectItem in objects) {  
        if ( (objectItem is aiSubject) && objectItem.id == id )  
            return true;  
    }  
    return false;  
}
```

Figure 3.9: `isObjectAlive()` method of `CBallGender`

3.3.2 Showing a background video

Many interactive systems require showing the video captured by the camera as the background, so we can overlay graphical elements and the user can interact with them having a clear reference of what parts of the image correspond to his/her movements.

aiSense interface provides a method to set webcam video as background. This method belongs to the `aiSenseObject` class and is ready to be used whenever necessary. The following code shows its usage.

```
private function setBackgroundVideo( bkVideo:Video):void {  
    if( bkVideo == null ){  
        bkVideo = this.createVideo( stage.stageWidth, stage.stageHeight);  
        bkVideo.alpha = 0.5;  
        this.addChild(bkVideo);  
    }  
}
```

Figure 3.10: setBackgroundVideo() method of *CBallManager*

It is also possible to create a video as usual in Flash [6], obtaining the camera object which is capturing video data. The camera object is accessible through the aiSense interface by using the camera property of the *aiClient* class, which is also accessible through the **connection** property of the *aiSenseObject* class.

4 Presence detection in real time

In the previous example we did not update the position of the Flash object by using the detected faces. We just detected people presence and made specific actions (changing colors, setting a position) for that first event.

In this chapter we continue developing the same example incrementally: we are updating in real time all properties we are going to use. Sections 4.1 and 4.2 describe this feature and how to scale a Flash object by using the **depth** property. These changes belong to the *BallGender_RealTimePresence* example which has the same file structure as *BallGender_Presence* example since it is its extension. Thus, we maintain the `main` function, *CBallManager* and *CBallGender* classes.

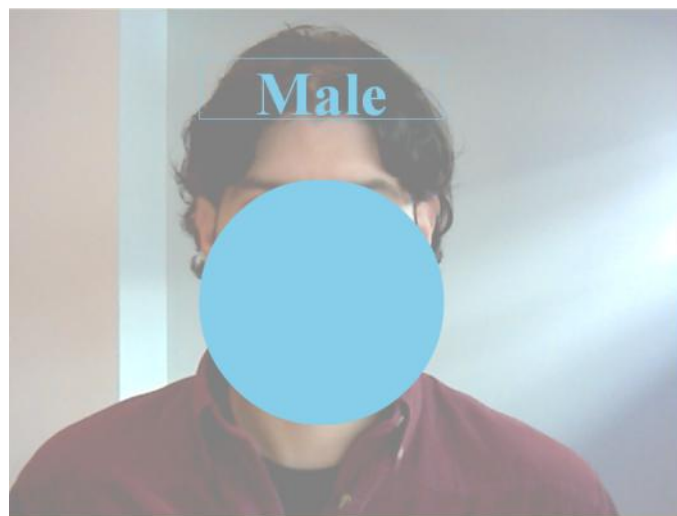


Figure 4.1: Output of *BallGender_RealTimePresence* example

On the other hand, the section 4.3 introduces other features of aiSense which are used with the aim of increasing and completing the Flash interactivities. These aspects are related to the *BallGender_ShirtColor* example which is adapted from the *BallGender_RealTimePresence* and has the same three classes but with different names since they represent more general concepts:

- `Main.as`: Entry point of the application.
- `CManager.as`: *CManager* class inherits from the *aiSenseObject* and instantiates a *CGraphicContainer* object for each person detected.
- `CGraphicContainer.as`: Equivalent to *CBallGender* but adds a square which is placed in the position of the detected person's shirt. It also changes the square color depending on the color of the shirt that is wearing the detected person.

In this last example of the section, besides having a ball with a text label shown when someone is in front of the camera, a square is drawn and placed on top of the person's shirt image. It

changes its color depending on the shirt color. Besides, the application plays a car horn sound if the detected person is not looking frontally at the camera.

The result is shown in the following picture.

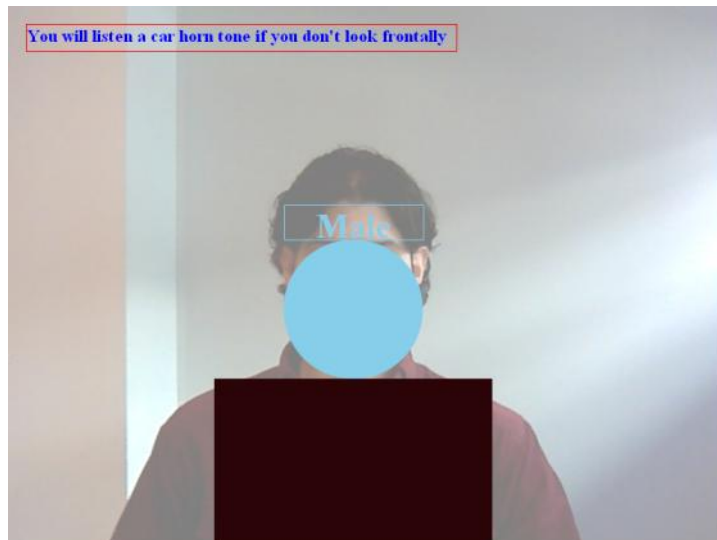


Figure 4.2: Output of *BallGender_ShirtColor*

4.1 Updating properties in real time

Following the same code structure of chapter 3, we add an `update()` method in the *CBallGender* class so that we can send the suitable subject in every detection.

The code for this method is shown in the following picture. Note that the `setDimension()` method will be explained later in section 4.2.

```
public function update( subject:aiSubject ):void {
    setColor(subject);
    setDimension(subject);
    setPosition(subject);
}
```

Figure 4.3: `update()` method of *CBallGender*

This function is called by the `detectPresence()` method of *CBallManager* (Figure 3.5). Therefore, only one single line needs to be added:

```
else m_aBall[i].update(subjects[i]);
```

, as shown in the following listing.

```

////////////////////////////////////
// CREATING BALLS
////////////////////////////////////
for ( i = 0; i < subjects.length; i++){

    // Drawing the ball if there is none for a detected subject.
    if ( !isObjectAlive(subjects[i].id, m_aBall) ) {

        var ball:CBallGender = new CBallGender(subjects[i]);

        this.addChild(ball);
        m_aBall.push(ball);
    }
    else m_aBall[i].update( subjects[i] );
}

```

Figure 4.4: Adding the `update()` method into `CBallManager`

You may find the complete code in the folder called *BallGender_RealTimePresence*, the same name as the example it contains.

4.2 Scaling objects

It is possible to update our `CBallGender` class with the detection obtained through the manager (`CBallManager` object). Also, we have seen how to modify its position and color depending on the information of the detected people.

The following code uses the **depth** property of the subject to scale the `CBallGender` object, which contains both the ball color and a text label.

```

public function setDimension( subject:aiSubject):void {

    // The container's size is the sum of ball and text sizes, so both are changed.

    // Depth property should be always multiplied by the stage's height;
    m_Ball.width = subject.depth * stage.stageHeight;
    m_Ball.height = subject.depth * stage.stageHeight;
    m_TextField.width = m_Ball.width;
    m_TextField.height = m_Ball.height/4;
}

```

Figure 4.5: `setDimension()` method of `CBallGender`

The **depth** property indicates the distance between camera and face. It is normalized in a range between 0 and 1, where 0 means the furthest distance to the camera, and 1 means the closest distance. This property has to be always multiplied by the height of the stage due to internal calculations.

As a result, the `CBallGender` object can decrease or increase its size if the detected subject is approaching or walking away. The following picture shows these states:



Figure 4.6: Scaling objects through the depth property

4.3 Other properties

As mentioned previously, the code of this example is related to the example called *BallGender_ShirtColor*. This example is derived from *BallGender_RealTimePresence* and *BallGender_Presence*.

In this example, the *CGraphicContainer* has a ball, a text field and a square. These objects are shown in the screen when the manager gets a presence detection. Its constructor is shown in the following code.

```
public function CGraphicContainer( subject:aiSubject, sound:Sound ){

    // Person detected
    m Subject = subject;

    // init() method is activated when a CGraphicContainer's instance
    // is added to the stage.
    addEventListener( Event.ADDED_TO_STAGE, init);

    // Ball
    m Ball = new Shape();

    // Text field
    m TextField = new TextField();
    m TextField.border = true;

    // Square
    m_Square = new Shape();

    //Sound
    m Sound = sound;

}
```

Figure 4.7: *CGraphicContainer* constructor

The main change is that the constructor has an additional parameter. This parameter is a *Sound* object which is used for introducing the **frontal** property later in section 4.3.2.

Moreover, *CGraphicContainer* checks, besides the typical features such as color or dimensions, if the detected subject is looking frontally at the camera. This behavior, implemented by the `checkFrontalView()` method, is added to the `update()` method in the following code.

```
public function update( subject:aiSubject ):void {
    setColor(subject);
    setDimension(subject);
    setPosition(subject);
    // Checks if the subject is looking frontally at the camera.
    checkFrontalView(subject);
}
```

Figure 4.8: `update()` method of *CGraphicContainer*

Also, the `setColor()` method is extended in order to use another property of the *aiSubject* class: **rgb**. This property is discussed in the following section.

4.3.1 Rgb property

The **rgb** property represents the color of the upper outfit that the detected person is wearing. It is an array with three RGB - red, green and blue - values which are in a range between 0 and 255. It is a property of the *aiSubject* class and is used by the example to change the square color.

```
public function setColor( subject:aiSubject ):void{
    var newColorTransform:ColorTransform = m_Ball.transform.colorTransform;

    if( subject.gender == "male"){
        newColorTransform.color = 0x87CEEB; // blue
        m_TextField.text = "Male";
    }
    else{
        newColorTransform.color = 0xFFC0CB; // Dark pink
        m_TextField.text = "Female";
    }
    // The same color for ball and text
    m_Ball.transform.colorTransform = newColorTransform;
    m_TextField.transform.colorTransform = newColorTransform;
    // Color for the square from the person's shirt.
    var rgbColor:Array = subject.rgb;
    m_Square.transform.colorTransform = new ColorTransform(0, 0, 0, 1, rgbColor[0],
    rgbColor[1], rgbColor[2], 0);
}
```

Figure 4.9: `setColor()` method of *CGraphicContainer*

4.3.2 Frontal property

The **frontal** property indicates whether the detected person is looking frontally at the camera or not. Its value is `true` if the person looks frontally and `false` otherwise.

Consequently, we check this *Boolean* property to play the sound if the person is not looking frontally at the camera.

```
public function checkFrontalView( subject:aiSubject):void {  
    if ( !subject.frontal ) m_Sound.play();  
}
```

Figure 4.10: checkFrontalView() method of *CGraphicContainer*

Another use of this property is implemented in the following sections, where a menu is disabled when the user is not looking frontally at the camera (Figure 5.3).

5 Motion regions

So far we are able to produce changes in multimedia content depending on people presence and their physical traits (gender, head, etc). In this chapter, we present another mechanism to extend the interactivity in multimedia systems through the aiSense interface: motion regions.

aiSense Regions (or aiRegions) provide an interactive mechanism to capture motions made in the scene seen by the camera. Specifically, this concept lets us define in our Flash application areas where we want to check whether there is motion or not.

For instance, we may be interested in building a button and checking if this button is pressed. This functionality can be achieved by defining a motion region in Flash and connecting it to the aiSense computer vision modules.

This chapter describes how to manage aiSense regions in Flash through a couple of examples. The first one illustrates a basic use of the *aiRegion* class and the second one describes advanced concepts related to multiple regions.

aiRegion implements a region defined by the user and it is managed by the *aiSenseObject* class. It must be initialized from a *DisplayObject* since it gets its properties (such as x, y, width, height...) and sends them to aiSense server. This *DisplayObject* is kept as a reference in the **graphicObject** property of *aiRegion*. In this way, this property gives *aiRegion* a set of characteristics such as height, width and position. As a result, once an *aiRegion* has a **graphicObject**, its properties can be accessed through the corresponding *aiRegion* properties - **width, height, x** and **y** -, which map the values of the **graphicObject**. In this chapter we analyze these concepts in detail.

Generally speaking, we may separate the region functionalities into two main parts: *definition* and *management of regions*. The former is a set of consecutive steps to begin working with regions whereas the second contains tasks in order to modify its settings in both the aiSense interface and server. Hence, the scheme is the following:

1) Definition of regions

- 1.1) Adding an event listener for region detections.
- 1.2) Creating a display object in the Flash design interface or script code.
- 1.3) Adding the display object to the Flash display list.
- 1.4) Creating an aiSense region from the display object.
- 1.5) Registering the aiSense region in the interface.
- 1.6) Sending aiRegion settings to aiSense server.

2) Management of regions

- 2.1) Obtaining regions from every event (just regions with motion).

2.2) Removing a region that was defined previously.

2.3) Getting the list of defined aiSense regions.

2.4) Modifying an aiSense region.

It is very important to note in the previous scheme that updating aiSense server by sending *aiRegion* settings is required for any region change (modification, definition, etc) made through the interface. Therefore, **1.6) Sending aiRegion setting to aiSense server** task is also required in **2.2)**, **2.3)** and **2.4)**.

The example following in section 5.1 is called *FlyingRegion* (Figure 5.1) and has the following file structure:

- Main.as: The entry point of the application.
- CManager.as: *CManager* class obtains information about detections which are given by aiSense server. As it is derived from *aiSenseObject*, it manages a dynamic motion region when a face is detected.
- CRegion.as: *CRegion* class is a dynamic motion region. This class inherits from *aiRegion* and has methods for updating itself.

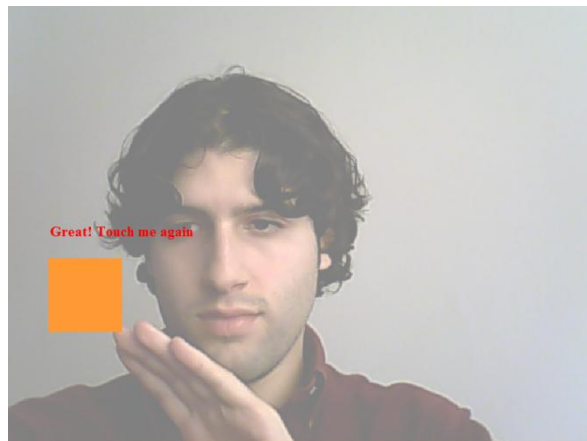


Figure 5.1: Output of *FlyingRegion* example

This application combines presence and motion detection: when a person is detected, the system creates a square object and defines a motion region on top of it. If the person causes motion within the region, the square is automatically placed in another position of the stage, that is, the region is dynamically created in the new position. If the person leaves the scene or there is no motion detected in the region for a few seconds, the system goes back to its initial state.

The second example is called *InteractiveMenu* (Figure 5.2). It is used in section 5.2 for describing the use of multiple motion regions. Moreover, it introduces the creation of static

regions, that is, regions whose position remains fixed throughout the application execution. It has the following file structure:

- Main.as: The entry point of the application.
- CManager.as: *CManager* class derives from *aiSenseObject* and starts or finishes the application on the basis of presence detection. Besides, it manages a set of static regions to control when a button is pressed.
- CHorizontalMenu.as: *CHorizontalMenu* class contains a set of options with a default animation.
- CFileLoader.as: *CFileLoader* class loads images (JPG, PNG, GIF) or SWF files from a set of paths and keeps them in memory.



Figure 5.2: Pressing an option in the *InteractiveMenu* example

A menu is shown when the application detects a person. The menu has the following features:

- It has three options.
- It is possible to highlight the options.
- If the user presses an option which has already been highlighted, it will get back into its normal state.
- If the user does not look frontally at the camera, the menu is disabled (Figure 5.3).
- If the menu is disabled the options remain in their last state, that is, if they were off, they will remain off and if they were on, they will remain on.



Figure 5.3: Disabled menu in the *InteractiveMenu* example

5.1 A basic dynamic region

As mentioned above, we focus on the *FlyingRegion* example. Thus, we analyze it in detail following the described scheme at the beginning of the chapter.

5.1.1 Definition of regions

This section describes the first module: **1) Definitions of regions**. In the following example the *CManager* class is in charge of aiSense regions. We will address each step by analyzing the code of this entity.

First of all, the constructor function implements the task **1.1) Adding an event listener for region detections**. The code is shown in the following listing.

```
public function CManager() {
    // Stars the game and defines a region when someone comes close
    addEventListener( aiSubjectEvent.SUBJECT_DETECTION, detectPresence);

    // Listens for every motion produced in the defined region
    addEventListener( aiRegionEvent.REGION_DETECTION, detectMotion);

    // The initial image is loaded when the manager is added to the root (stage).
    addEventListener( Event.ADDED_TO_STAGE, init);

    m_Player = null;
    m_bGameStarted = false;
    m_Timer = new Timer( TIME_FORINITIALSTATE );
    m_Loader = new Loader();
    m_bInitialState = true;
}
```

```
// Stops the game when there are no motions for X ms. and re-loads the initial state.
    m_Timer.addListener( TimerEvent.TIMER, stopGame);
}
```

Figure 5.4: *CManager* constructor of the *FlyingRegion* example

As the reader may observe, the *REGION_DETECTION* event type of the *aiRegionEvent* object is used for calling the *detectMotion()* method. In this way, the *CManager* class will be prepared to capture region events when the interface notifies them. However, region events will not be notified until the *aiRegion* objects are defined. *aiSense* server will begin detecting motions when it receives region definitions from the *aiSense* interface. Therefore, the example leverages presence detection through the *detectPresence()* method to define the *aiRegions* and start motion detection at *aiSense* server.

The absence of motions is also controlled by a timer: when there are no motions detected for a predefined number of seconds, (in the example defined by *TIME_FORINITIALSTATE*), the timer throws an event which runs the *stopGame()* method to remove the region from the interface and erase it graphically from the stage.

The following code shows the use of presence detection in this example.

```
////////////////////////////////////
////// PRESENCE DETECTION: The game starts when someone is in front of the camera.
//// This method creates and removes regions.
////////////////////////////////////
public function detectPresence( evt:aiSubjectEvent ):void {

    var subjects:Array = evt.getSubjects();
    //////////////////////////////////////
    //// 1. Taking the closest person (to the camera)
    //////////////////////////////////////
    var closestSubject:aiSubject = peopleDepthFilter( subjects );

    //////////////////////////////////////
    //// 2. Assigning player and change it if another one is closer.
    //////////////////////////////////////
    if ( m_Player == null ) m_Player = closestSubject;
    else
        if ( closestSubject.id != m_Player.id ) {
            // player swapping
            m_Player = closestSubject;
            // A new game will begin when the player has been changed.
            m_bGameStarted = false;

            //////////////////////////////////////
            //////////// REMOVING A REGION
            //////////////////////////////////////
            // Removes the displayObject from stage, NOT aiRegion.
            stage.removeChild( definedRegions[0].graphicObject);
            // Removes aiRegion from our list of defined regions
            removeRegion(definedRegions[0]);
            // Sends changes to aiSense Server.
            refreshServer();
            //////////////////////////////////////
        }
        else m_Player = closestSubject;

    //////////////////////////////////////
    //// 3. Starting the game by creating a region.
    //////////////////////////////////////
    if ( !m_bGameStarted ){
```

```

////////////////////////////////////
////////// CREATING A REGION
////////////////////////////////////

// Instantiate a DisplayObject
var square:Sprite = createSquare(100, 100);
// Put it into the display list before adding it to our region list
stage.addChild( square );
// Create the aiSense region from the displayObject
// Add the region to our list
addRegion( new CRegion( square ) );
// Send changes to aiSense Server
refreshServer();
////////////////////////////////////

m bGameStarted = true;
m Timer.start();
}
}

```

Figure 5.5: detectPresence() method of CManager

The previous code can be divided into two main logical parts. The first one deals with the case when the detected person either leaves the scenario or is replaced by another detected person. The replacement may be produced by either occlusion (one new detected person covers the previous one) or distance (a new detected person is closer to the camera than the previous one). Region removal is necessary for both cases and we will discuss this topic in the next section after describing the creation and definition of regions.

The second part of the detectPresence() method starts the application by creating both a graphic region in the Flash stage and a motion region defined on top of this graphic object. This part is related to the task **1) Definitions of regions**, so continuing the general process we have:

1.2) Creating a display object in the Flash design interface or script code.

We instantiate a display object by using the createSquare() method. This object will be referenced by aiRegion (graphicObject property of aiRegion).

1.3) Adding the display object to the Flash display list.

In this case, we add it as a stage child.

1.4) Creating aiSense region from the display object.

We instantiate an aiRegion object, or a class derived from it, by passing the created display object as reference.

1.5) Registering the aiSense region in the interface.

The aiSense interface internally maintains a list of defined regions. It also has a public method called addRegion() used to add an aiRegion instance to the internal list.

1.6) Sending aiRegion settings to aiSense server.

This step is crucial to make our applications work properly. Once we have defined or changed our motion regions, we need to communicate these changes to the aiSense

server by using the public method of *aiSenseObject* called `refreshServer()`. Otherwise, the server will not be able to detect any motion in the defined regions, as it will not have any information about them.

5.1.2 Management of regions

The management of regions involves a set of tasks which have the aim of both providing the use of dynamic regions and establishing a robust communication between the server and the interface.

The following listing shows the `detectMotion()` method which, as we pointed out before, is executed when an *aiRegion* event is thrown.

```
public function detectMotion( evt:aiRegionEvent ):void {

    m_Timer.stop();
    var regions:Array = evt.getRegions();
    // Removes the initial state when there are motions.
    if( m_bInitialState )
        removeInitialState();
        ////////////////////////////////////////////////////////////////////
        // Put a threshold in order to avoid small motions
        ////////////////////////////////////////////////////////////////////
    if ( regions.length > 0 && regions[0].motionLevel > MOTION_THRESHOLD ) {

        // Show the new position on the console.
        trace("REGION MOVED: x:", regions[0].x, " y:", regions[0].y );
        ////////////////////////////////////////////////////////////////////
        // The region changes its position randomly
        ////////////////////////////////////////////////////////////////////
        definedRegions[0].update();
        // Send changes to aiSense Server
        // WARNING ---> If this call is skipped, the server will not be updated!
        refreshServer();

    }

    m_Timer.start();
}
}
```

Figure 5.6: `detectMotion()` method of *CManager*

First of all, the method implements the **2.1) Obtaining regions from every event** task. The `getRegion()` method of the *aiRegionEvent* class is used to obtain a list of the regions which have motions detected by aiSense server. This list only contains the defined regions where any motion has been detected, that is, if there is no motion in a defined region then this region will not appear in this list. To access defined regions without motion detected, the **definedRegions** property of the *aiSenseObject* class must be used. This is discussed in the task called **2.3) Getting the list of defined aiSense regions.**

Second, the code modifies the region if the detected motion was relevant by checking its **motionLevel** property. This property indicates the motion volume within the *aiRegion*. The minimum value is 0 and the maximum one depends on the size of the defined region, computed as $255 * \text{region.width} * \text{region.height}$, where **width** and **height** properties are in image pixels. The maximum width and height expressed in image pixels are

320 and 240, respectively, which is the image size managed by the aiSense server. Therefore, it may be used to discriminate a set of regions which has registered motion, that is, regions which have larger **motionLevel** are more significant than others with smaller **motionLevel**.

The rest of management tasks are shown in the following listing.

```
public function stopGame( evt:TimerEvent):void {

    m_bGameStarted = false;
    m_Player = null;
    m_Timer.stop();
    ///////////////////////////////////////////////////////////////////
    // REMOVING A REGION
    ///////////////////////////////////////////////////////////////////
    // definedRegions property contains all defined regions
    if( definedRegions.length > 0){
        stage.removeChild( definedRegions[0].graphicObject);
        removeRegion(definedRegions[0]);
        refreshServer();
    }
    ///////////////////////////////////////////////////////////////////

    ///////////////////////////////////////////////////////////////////
    /// SETTING INITIAL STATE
    ///////////////////////////////////////////////////////////////////
    this.addChild(m Loader);
    m bInitialState = true;
}
}
```

Figure 5.7: stopGame() method of CManager

The stopGame() method removes the defined region and loads the initial background image. Therefore, it implements the step **2.2) Removing a region which was defined previously**. First of all, we remove the graphical object from the stage. This object is contained in the aiRegion entity so we reference it through the **graphicObject** property.

Then, we remove the aiRegion from the interface by calling the removeRegion() method of aiSenseObject class. Finally, all changes are communicated to the server through the refreshServer() method.

In this task it is important to take into account that removing an aiRegion from the interface does not remove the corresponding graphical object from the stage.

The **2.3) Getting the list of defined aiSense regions** task was already applied in the previous code since we used the **definedRegions** property which is the list of all defined regions.

Finally, the task called **2.4) Modifying an aiSense region** was implemented in the listing in Figure 5.6, where we called the update() method (Figure 5.9) of the CRegion class, which derives from aiRegion. This method changes the position of both the aiRegion and its embedded graphical object.

The listing of the CRegion constructor follows.

```

public function CRegion( graphicRegion:* ) {

    //////////////////////////////////////
    //// Calling the parent class constructor, that is, the aiRegion class.
    //////////////////////////////////////
    super(graphicRegion);

    this.x = 300;
    this.y = 400;
    m_Text = new TextField();
    m_Text.text = "New Game. Touch me!";
    m_Text.width = 400;
    m_Text.setTextFormat(getTextFormat());
    m_Text.x = this.x;
    m_Text.y = this.y - m_Text.height / 2;

    // Adding the text to the stage
    graphicRegion.stage.addChild(m_Text);
    // Removing the text when graphicRegion is removed from the stage.
    // If the text was a graphicRegion's child, this method would not be necessary
    // since it would be removed when the graphicRegion was removed.
    graphicRegion.addEventListener(Event.REMOVED FROM STAGE, removeText );
}

```

Figure 5.8: *CRegion* constructor in the *FlyingRegion* example

Due to the fact that *CRegion* derives from *aiRegion*, we call the parent constructor through the `super()` statement since every *aiRegion* object must be initialized with the graphic object. Furthermore, `x` and `y` properties are initialized. These properties belong to the *aiRegion* and change automatically the position of the contained graphic object. Besides, these coordinates, in contrast to subject positions, are in pixels and indicate the top-left corner of the region.

This modification is applied randomly in real time by using the `update()` method explained before.

```

public function update():void {

    this.x = randomNumber(0, this.graphicObject.stage.stageWidth-this.width);
    this.y = randomNumber(0, this.graphicObject.stage.stageHeight - this.height);
    updateText();
}

```

Figure 5.9: `update()` method of *CRegion* class

5.2 Using multiple motion regions

The example shown in the previous section only managed one motion region. The access to this element, to modify or remove it, is trivial because we know in every moment that there is just one element, and only one, to do all the functionalities commented. For instance, we could access the region through the `definedRegions` property with a zero index, as in Figure 5.7, since we know what the index is and assume that there is just one region.

But, what if there are multiple regions?

In this section we address this question through a more advanced example. As mentioned at the beginning of the chapter, the example is called *InteractiveMenu*. Moreover, this example will help us understand the **graphicObject** property of the *aiRegion* class suitably and it also introduces a new feature not seen so far: creating static regions.

The main idea is that a menu with three options (three buttons) is created and we need to define three *aiRegions*, one on top of each option, to detect when each option is selected (“touched”) by the user. Obviously, the user has to make some motion, generally, with the hand. Furthermore, it is necessary to identify every region so they can be removed or modified.

5.2.1 Identifying an aiRegion

The following listing shows a method of the *CManager* (`manager` instance) class, which is called when the `manager` starts the connection to the server.

```
private function initMenu():void {
    m_Menu = new CHorizontalMenu( "menu1", "option1", "option2", "option3");
    var images:Array = new Array( m_FileLoader.getImage(BUTTON_IMAGE),
    m_FileLoader.getImage(BUTTON_IMAGE), m_FileLoader.getImage(BUTTON_IMAGE));
    m_Menu.initOptions( images );
}
```

Figure 5.10: `initMenu()` method of *CManager*

As we may see, it creates a menu and provides its option names. Internally, the menu creates three button options (*Sprite* objects) and names them with the option name provided by the manager (Figure 5.11). These options, which are display objects, will be used as graphical objects to define the motion regions.

```
public function initOptions( buttonImages:Array ):void {
    for ( var i:uint = 0; i < m_aOptionName.length; i++){
        initOption(m_aOptionName[i], buttonImages[i] );
        m_aTween[m_aOptionName[i]] = null;
    }
}

private function initOption(optionName:String, buttonImg:Bitmap):void {
    var option:Sprite = createGraphic(optionName, buttonImg, OPTION_WIDTH, OPTION_HEIGHT);
    option.name = optionName;
    addChild(option);
}
```

Figure 5.11: `initOptions()` method of *CManager*

Hence, we indicate the instance name of these options through the name property of the *Sprite* class to reference them from *aiRegion* when we want to manage the motion regions.

This mechanism is a way to reference an *aiRegion* through the instance name of its **graphicObject** property. Besides, it is important to note that this approach places the identification in the display object which is contained by the *aiRegion* object. Thus, we would always check the instance name of the **graphicObject** property to know the region id.

Another mechanism is to use the **id** property of the *aiRegion* class. This property is a zero-based index set automatically by the aiSense interface. Although it is simpler since it is provided by the interface, it is not as intuitive as the first mechanism because we would have to take into account the order of *aiRegion* creation to know what the region is in every moment. Moreover, note that, in the case that a region is removed, the interface assigns to a new defined region the first free id starting at 0. For instance, for three regions the state before removing a region is 0, 1, 2 and after removing the region 1 and creating another one the state remains 0, 1, 2. Therefore, the logical order does not correspond to the order of creation.

In this section we use the first mechanism by using the **graphicObject** property.

5.2.2 Creating/Removing multiple regions

The following code shows how to define or remove regions.

```
private function defineRegionsForMenu( evt:Event ):void {
    var menu:CHorizontalMenu = evt.currentTarget as CHorizontalMenu;
    menu.removeEventListener(Event.CHANGE, defineRegionsForMenu );
    for ( var i:uint = 0; i < menu.numChildren; i++) {
        var region:aiRegion = new aiRegion( menu.getChildAt(i));
        addRegion(region);
    }
    // Do not forget to send all changes to aiSense server!
    refreshServer();
}

private function removeRegionsForMenu( menu:CHorizontalMenu):uint {
    var counter:uint = 0;
    for ( var i:uint = 0; i < menu.numChildren; i++){
        for each( var regionDefined:aiRegion in definedRegions) {
            if ( regionDefined.graphicObject.name == menu.getChildAt(i).name ) {
                removeRegion( regionDefined );
                counter++;
            }
        }
    }
    if( counter == menu.numChildren )
        refreshServer();
    return counter;
}
```

Figure 5.12: `defineRegionsForMenu()` and `removeRegionsForMenu()` methods of *CManager*

In the previous code, we create an *aiRegion* for each menu option through the `defineRegionsForMenu()` method, so the graphical object for every motion region is an option button. This method is called when the menu animation finishes.

The `removeRegionsForMenu()` method introduces how to find an *aiRegion* by using the instance name of its **graphicObject**. Furthermore, it emphasizes the idea that communicate all changes must be communicated to aiSense server.

For more information, see the source code of the *InteractiveMenu* example.

6 Advanced example

Following the examples of the previous chapter, presence and motion detections are now combined to produce a more complex application.

In this chapter we are developing an application which uses face detection to define and place dynamically an *aiRegion* with the aim of detecting motions made by the head (headers). Specifically, the program starts when a person is detected and draws a menu with two football player faces. The detected person may choose any option and the football player face is placed on the picture of the person's face. Also, the dimensions of the football player picture are fitted according to the dimensions of the detected face (Figure 6.1). If the person makes a header (bottom-up motion with their head), the ball is thrown to the top of the application window (Figure 6.2). If the detected person leaves the scene, the application goes back to its initial state.



Figure 6.1: Output of *FootBallHeading* example

The example has the following structure:

- *Main.as*: The entry point of the application.
- *CManager.as*: *CManager* class inherits from the *aiSenseObject* and starts or finishes the application on the basis of presence detection. Besides, it contains a menu with two options and detects movements made by the head.
- *CFileLoader.as*: *CFileLoader* class loads images (jpg, png, gif) or SWF files from a set of paths and keeps them in memory.



Figure 6.2: Output of *FootballHeading* example

6.1 Creating an interactive menu

The *CManager* class has the suitable methods to detect presence and motions. As mentioned above, the application starts when a face is detected and creates the menu. Also, it has to define the *aiRegions* to run motion detection. The listing in Figure 6.3 shows presence detection. Its logical order is:

1. Creating the menu the first time a face is detected.
2. Checking if the defined regions keep the face detected for the current frame.
3. Updating or creating the picture of the football player face in the detected person.
4. Sending updated information to aiSense server.

```
public function detectPresence(e:aiSubjectEvent):void{
var subjects:Array = e.getSubjects();

    if( m_bInitialState ){
        removeInitialState();
        createMenu();
        this.stage.addChild(m_WarningText);
    }

    m_RestartTimer.stop();
    ////////////////////////////////////////////////////////////////////
    // 1. Checking every object and remove in each case
    ////////////////////////////////////////////////////////////////////
    // The aiSenseObject's definedRegions property maintains all regions defined.
    for ( var i:uint = 0; i < this.definedRegions.length; i++) {

        if ( this.definedRegions[i] is CRegion) {

            var subjectForRegion:aiSubject = this.definedRegions[i].subject;

            // If there is no region matched with subject but it exists on the
            // scene, then remove region.
        }
    }
}
```

```

        if ( !isSubjectAlive( subjectForRegion.id, subjects ) ) {

            if ( this.definedRegions[i].graphicObject.parent != null) {
                stage.removeChild(this.definedRegions[i].graphicObject);
                this.removeRegion(this.definedRegions[i]);
            }
        }
    }
}
// 2. Drawing the object if it is necessary
// 2.1 Drawing it if there is no region for a detected subject.
for ( i = 0; i < subjects.length; i++) {

    if ( !isSubjectAlive(subjects[i].id, this.definedRegions)) {

        var face:Sprite = createFace(m_sCurrentFace, subjects[i].width, subjects[i].height);
        stage.addChild(face);

        var region:CRegion = new CRegion( face, m_FileLoader.getImage(BALLIMAGE_NAME) );
        addRegion( region );
    }

    for ( var j:uint = 0; j < this.definedRegions.length; j++) {

        // 2.2 If region does not still have a matched
        // subject (so it's a new region), then update it.
        if ( this.definedRegions[j] is CRegion && this.definedRegions[j].subject == null )
        {
            this.definedRegions[j].update( subjects[i], null);
        }
        else // 2.3 If region had a matched subject previously (so it is not a new region)
        if ( this.definedRegions[j] is CRegion && this.definedRegions[j].subject != null &&
            this.definedRegions[j].subject.id == subjects[i].id ) {

            // 2.3.1 If the user changed the player's face,
            // the region has to change it as well
            if( this.definedRegions[j].graphicObject.name != m_sCurrentFace ){

                var face:Sprite = createFace(m_sCurrentFace, subjects[i].width,
                    subjects[i].height);
                stage.removeChild( this.definedRegions[j].graphicObject );
                stage.addChild( face );
                this.definedRegions[j].update( subjects[i], face);

                // 2.3.2 If user did not change the face, just update the region.
                }else this.definedRegions[j].update( subjects[i], null);
            }
        }
    }
}
// 3. Send updated regions to aiSense server
refreshServer();

m RestartTimer.start();
}

```

Figure 6.3: detectPresence() method in the *InteractiveMenu* example

In the previous code the *CRegion* is created when the face is detected; this class derives from *aiRegion* and is dynamic, but in contrast to the *CRegion* example of Figure 5.8, it has the information about the detected face in order to be placed in its position every moment. The constructor function is shown in the following code.

```

public function CRegion( userRegion:*, ball:Bitmap ) {
    super( userRegion );
    m_Subject = null;
    m_bAnimationRunning = false;

    // This object contains the ball image as a child of itself.
    // So, we can manage as a common displayObject
    m_Ball = ball;
}

```

```

userRegion.stage.addChild(m Ball);
m_MyAnimation = null;

userRegion.addEventListener( Event.REMOVED FROM STAGE, removeBall);
}

```

Figure 6.4: *CRegion* constructor in the *InteractiveMenu* example

In addition, the updated method of the *CRegion* class to change its state is shown in the following listing, which changes the dimensions and positions for both the ball and the face picture.

```

public function update( subject:aiSubject, userRegion:* ):void {

    m_Subject = subject;
    if( userRegion != null ){

        this.graphicObject = userRegion;
        this.graphicObject.stage.addChild(m Ball);

        // the listener must be added when the
        // object reference is changed.
        userRegion.addEventListener( Event.REMOVED_FROM_STAGE, removeBall);
    }
    changeDimension( subject );
    changePosition( subject );

    if ( !m_bAnimationRunning ) {

        changeBallDimension();
        changeBallPosition();
    }
}

public function get subject():aiSubject { return m Subject; }

public function get isAnimationRunning():Boolean { return m_bAnimationRunning; }

public function changeDimension( subject:aiSubject ):void{

    this.width = subject.width * this.graphicObject.stage.stageWidth;
    this.height = subject.height * this.graphicObject.stage.stageHeight;
}

public function changePosition( subject:aiSubject ):void {

    this.x = subject.x * this.graphicObject.stage.stageWidth;
    this.y = subject.y * this.graphicObject.stage.stageHeight;
}

public function changeBallDimension():void {
    m Ball.width = this.width;
    m Ball.height = this.height;
}

public function changeBallPosition():void {

    m Ball.x = this.x;
    m_Ball.y = this.y - this.height;
}
}

```

Figure 6.5: update() method of *CRegion* class

According to Figure 6.3, the method to create the menu is called once after the initial state is changed. You can see the implementation in Figure 6.6. We create an *aiRegion* for each menu option in order to detect the chosen option when the detected person makes a motion in it.

```
private function createMenu():void{

    // Creating graphic options
    m_aMenu[0].x = 50;
    m_aMenu[0].y = stage.stageHeight - 120;
    m_aMenu[0].name = ROONEY_FACE;
    m_aMenu[0].addChild(m_FileLoader.getImage(ROONEYIMAGE_NAME));
    m_aMenu[0].width = 100; m_aMenu[0].height = 100;

    m_aMenu[1].x = stage.stageWidth - 130;
    m_aMenu[1].y = stage.stageHeight - 120;
    m_aMenu[1].name = LAMPARD_FACE;
    m_aMenu[1].addChild(m_FileLoader.getImage(LAMPARDIMAGE_NAME));
    m_aMenu[1].width = 100; m_aMenu[1].height = 100;

    stage.addChild(m_aMenu[0]);
    stage.addChild(m_aMenu[1]);

    //////////////////////////////////////
    //// Creating regions for menu
    //////////////////////////////////////
    var region1:aiRegion = new aiRegion(m_aMenu[0]);
    var region2:aiRegion = new aiRegion(m_aMenu[1]);
    this.addRegion(region1);
    this.addRegion(region2);

    this.refreshServer();
    //////////////////////////////////////
}
}
```

Figure 6.6: createMenu() method of *CManager* class

Furthermore, when there is no detection and the application goes back to its initial state, the menu is removed by using the `removeMenu()` method (Figure 6.7). Like the `createMenu()` method, we must remove both the graphic element from the screen and the defined region from the interface.

```
private function removeMenu():void {

    //////////////////////////////////////
    //// Removing regions from menu
    //////////////////////////////////////
    for ( var i:int = 0; i < this.definedRegions.length; i++){

        if ( this.definedRegions[i] is aiRegion )
            this.removeRegion(this.definedRegions[i]);
    }

    // Removing graphic options from visualization list
    stage.removeChild(m_aMenu[0]);
    stage.removeChild(m_aMenu[1]);
}
}
```

Figure 6.7: removeMenu() method of *CManager* class

6.2 Velocity property

Once the menu is created and the regions are defined, we focus on the feature we pointed out before: if the person makes a bottom-up motion with their head, the ball is thrown to the top of the application window (Figure 6.2). Therefore, besides of detecting motions and discriminating them by motion volume, the direction and orientation of motion have to be checked as well.

This need drives us to check another property of the *aiRegion* class: the **velocity**. This parameter means the direction vector which provides both the motion direction and orientation. Its components (x and y) are normalized in a range between -1 and 1.

The `detectMotion()` method presented in the following picture uses this property together with the **motionLevel** one in order to obtain the suitable motion to execute the animation of the ball.

```
public function detectMotion(e:aiRegionEvent):void {
    var regionsWithMotion:* = e.getRegions();
    var regionChanged:Boolean = false;
    for each( var region:* in regionsWithMotion) {
        /*
        * There are two kind of regions:
        * - CRegion: Customized class for tracking the face's motion.
        * - Menu region (aiRegion): Chooses the current face image.
        */
        if ( region is CRegion ) {
            /*
            * Change this threshold (5000 and HEAD_SENSIBILITY) for getting a
            * more/less sensitive motion region.
            * Note that the motion level depends on the person's distance to the
            * camera (depth property).
            * Velocity property defines the motion direction.
            */
            if ( region.motionLevel > 5000 && region.velocity.y < 0 &&
                region.velocity.y < HEAD_SENSIBILITY && !region.isAnimationRunning ) {
                region.loadAnimation();
            }
        }
        else {
            // if it is the other type of region (menu)
            if( region.graphicObject.name == ROONEY_FACE && region.motionLevel > 5000 )
                m_sCurrentFace = ROONEY_FACE;
            else if( region.graphicObject.name == LAMPARD_FACE &&
                region.motionLevel > 5000 )
                m_sCurrentFace = LAMPARD_FACE;
        }
    }
}
```

Figure 6.8: `detectMotion()` method of *CManager*

The rest of the code can be found in the *FootBallHeading* folder.

References

- [1.] AITECH, URL: www.aitech.es
- [2.] AS3 Online programming guide, Adobe, URL:
livedocs.adobe.com/flash/9.0/main/wwhelp/wwhimpl/js/html/wwhelp.htm
- [3.] GreenSock library, URL: www.greensock.com
- [4.] Adobe Creative Suite, Adobe, URL: www.adobe.com/uk/products/creativesuite
- [5.] FlashDevelop, URL: www.flashdevelop.org
- [6.] Creating a video in AS3, Adobe, URL:
livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/media/Video.html

Contact

Tel +34 93 586 89 72

Skype aitech.es

e-mail info@aitch.es

web www.aitch.es

Edifici Eureka - Campus de la UAB - 08193
Bellaterra (Barcelona)

Spain

Copyright © of Artificial Intelligence Technologies S.L. and all its providers. All rights reserved. Artificial Intelligence Technologies S.L. and all its mentioned products are registered trademarks property of Artificial Intelligence Technologies. Other mentioned products and companies in this document are trademarks belonging to its corresponding owners.