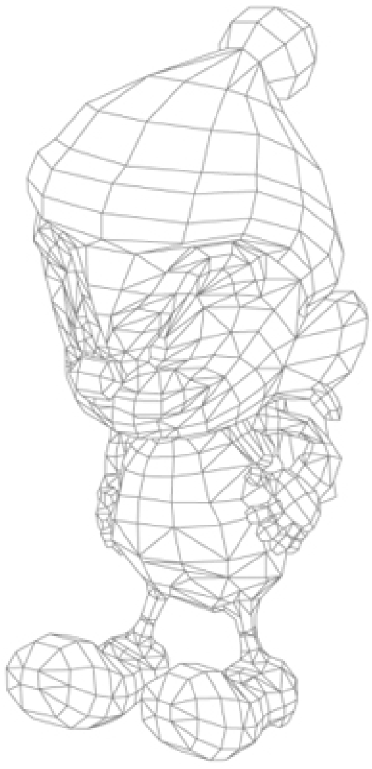
MascotCapsule V3

Tutorial Document for MIDP

English version



Ver.1.0.1

© 2007 HI CORPORATION. All Rights Reserved.

— Revision History —

November 27, 2006

- Ver.1.0 Released.

Updated Content

- Created the document.

December 26, 2007

- Ver.1.0.1 Released.

Updated Content

- Changed descriptions in the tutorial.

— Table of Contents —

Introduction

Purpose of tutorial

This tutorial provides procedures for creating MascotCapsule applications in MIDP environment. Various combinations of elements are required for 3D rendering. Descriptions will focus on procedures and relationship of each element.

As described in the next section, MIDP does not have unifying 3D graphics functionality. Therefore, it uses the functionalities provided in the profile. Please pay close attention to the relationship of these functionalities.

Knowledge of Java and MIDP is required for this tutorial.
For more information on MascotCapsule API, please refer to the programmer's manual and reference manual.

Operating environment

3D graphics API is not defined in MIDP. Therefore, the profile’s own 3D API is provided. Some profiles do not support 3D; therefore, please confirm whether 3D is supported in the intended environment.

This tutorial assumes the provided API is com.mascotcapsule. The unique 3D API from HI CORPORATION, com.mascotcapsule can be used from Java applications. It is provided as a package called com.mascotcapsule.micro3d.v3. When the API is provided with this name, sources are executable without changes.

Sample applications

The sample application consists of ten steps.
Each step contains the source code, and the resource file to be used in the source code.

The source code is **src**, which contains Java source file.
The resource file is **res**, which is 3D data to be used by MascotCapsule.

File	Explanation
*.mbac	Model data
*.mtra	Animation data
*.bmp	Image file to be used for model's texture data, or environment mapping

Note: For more details, please refer to programmer's manual provided by HI CORPORATION.

Place these files so that they accommodate the profile's development environment.

Firstly, decide the configuration of classes for the entire application.

1. **Main** class	Inherits the **MIDlet** class. This is the main class of the application.
2. **Canvas3D** class	Inherits the **Canvas** class, and performs rendering on the screen. In addition, this class implements **Runnable** interface, and provides the **run()** method.

The **Main** class generates the **Canvas3D** class object to be used as a canvas, and sets it up on the display. Also, the **Main** class creates a thread, and initiates execution.

Here, the **Canvas3D** class contains both canvas and thread functionalities.
In other words, most of the processes in the application are written in the **Canvas3D** class.

Step 1 and the subsequent sections provide explanations for this class only.

Step 1 — Displaying the 3D model, and using the texture

Step 1 simply displays the 3D character.
Please make sure that the object does not protrude from the visible range when displaying.

Step 1-1 Data member

The following data members are used here:

```
Graphics3D g3;
```

This is a graphics context to be used as a rendering target.
3D rendering will be performed onto this object.

```
Figure figure;
```

This represents a model.

```
FigureLayout layout;
```

This specifies where to place the objects on the screen when rendered.
This specifies the screen information, view coordinate transformation, projection methods, etc.

```
Texture mainTexture;
```

This retains the texture information.

```
Effect3D effect;
```

This specifies the rendering effect.
This data member enables lighting and shading functionalities.

```
AffineTrans affineTrans;
```

This represents the matrix for coordinate transformation.

```
private static Vector3D Pos = new Vector3D(0,120,500);
private static Vector3D Look = new Vector3D(0,0,-2000);
private static Vector3D Up = new Vector3D(0,4096,0);
```

This data member specifies the camera position.

When rendering the 3D object on a screen, it is important to specify from where this object is viewed.
It is a general rule of thumb to assume that there is a camera, and what can be seen from the camera will be rendered on the screen.

It is necessary to find the view coordinate in order to create 3D rendering. 3D objects exist in the world coordinate system. In terms of rendering onto the screen, however, ”how does it look from the camera” is important. Therefore, it is necessary to find out the positional relationship between the object and the one who looks at it.
This is called “view coordinate system,” and the world coordinate system must be transformed to the view coordinate system.
Vectors **Pos**, **Look**, and **Up** are used for the transformation.

Vector3D is a class that represents a three-dimensional vector.
Pos defines the camera position, **Look** defines the direction of the line of sight, and **Up** defines the up direction of the camera. Each of them is specified using the value where **4096** equals **1**. This is because floating decimal points cannot be used in this environment; therefore, the actual applied values are considered to be the setting value divided by 4096.

```
int centerX;
int centerY;
```

This is a variable to specify the center position.

```
int bgColor = 0x333377;
```

This specifies the background color.

Step 1-2 Each method and process

Each method handles the following process:

Canvas3D()

This is a constructer.

```
g3 = new Graphics3D();
```

Firstly, **Graphics3D** object is generated.
This object will be used when rendering.

```
figure = new Figure("/test_model_robo.mbac");
mainTexture = new Texture("/tex_001.bmp", true);
figure.setTexture(mainTexture);
```

Secondly, 3D data is generated.
Model data and texture data are separate; therefore, it is necessary to call the **setTexture()** method and associate them.

```
effect = new Effect3D( null, Effect3D.NORMAL_SHADING, true, null);
```

Default value is specified here, so that the effect can be changed later.

```
layout = new FigureLayout();
```

Layout value is entered when rendering.

```
viewTrans = new AffineTrans();
```

This generates the affine transformation matrix object.

```
centerX = getWidth() / 2;
centerY =getHeight() / 2;
```

This finds the center coordinate from the canvas size.

initViewParams()

This method initializes the value.
In this method, we intend to add initialization processes that are necessary for the **paint()** method.

```
viewTrans.lookAt(Pos, Look, Up);
```

This generates the matrix to be used for the transformation to the view coordinate system.
When displaying an object on the screen, a camera is assumed to be in the three-dimensional space; therefore, the camera position must be specified (the object would look differently, depending on where the camera is).

paint()

This is the method that actually performs rendering.
The **Canvas3D** class has thread functionality, and is configured to call this method at regular intervals.
This method is automatically called, and it describes what to be rendered. In MIDP environment, 2D rendering is also executed here. Especially when both 2D and 3D rendering are performed, please be careful about the procedures.

```
g.setColor(bgColor);
g.fillRect(0, 0, getWidth(), getHeight());
```

This specifies the background color of the canvas, and uniformly paints it in the same color.
Please note that rendering is performed onto **g**, instead of **g3**— this is a 2D rendering process.

```
g3.bind(g);
initViewParams();
layout.setCenter(centerX, centerY);
layout.setParallelSize(800, 800);
layout.setAffineTrans(affineTrans);
g3.renderFigure(figure, 0, 0, layout, effect);
g3.flush();
g3.release(g);
```

This is the essential portion of the code for rendering process.
Firstly, please note **Graphics3D.bind()**, and **Graphics3D.release()**.
The argument of the **paint()** method is **g**, instead of **g3**. Therefore, it can only perform 2D rendering as it is. When the **bind()** method is executed here, the rendering process performed onto **g3** will be (automatically) reflected on the screen via **g**.

In order to perform 3D rendering process, the **bind()** method is always required. In order to go back to the original state, call the **release()** method. Also, do not mix the 2D rendering process and 3D rendering process. When these processes are mixed together, objects may not be rendered correctly.
Only perform the 3D process between the **bind()** method and the **release()** method.

```
//Graphics3d g3;
paint(Graphics g){
        //Enabling 2D rendering (to g)
        g3.bind(g);
        //Enabling 3D rendering (to g3)
        g3.release(g);
        //Enabling 2D rendering (to g)
}
```

Then the value to be used in 3D rendering is prepared. Please note that this is a preparation, and not the actual rendering process.

Here, the **FigureLayout.setAffineTrans()** method is very important. This method will be used in the rendering process later, and will be necessary when rendering the object on the screen. The value is specified in the aforementioned **initViewParams()** method, and it represents the transformation to view coordinates.
The **setParallelSize()** method selects parallel projection as projection method.

Now, let's move onto the process of rendering the object.
The **renderFigure()** method is called when rendering the object; however, this method itself is not the actual rendering process. This method only specifies what to be rendered, and the result is not reflected on the screen yet.

The actual rendering process is performed when the **Graphics3D.flush()** method is called. The meaning of **flush** is same as flushing the water, and it refers to the action to let out the stored data all at once.
Only one object is used here; however, this method is useful when rendering multiple objects.

```
//Graphics3D g3;
//Figure obj1, obj2, obj3;
g3.renderFigure(obj1…);
g3.renderFigure(obj2…);
g3.renderFigure(obj3…);
g3.flush();
```

As shown above, in order to render multiple objects, **flush()** is called lastly. In general, **flush()** has to be called at the end. If **flush()** is called every time, rendering is also performed every time. Therefore, the lastly rendered object will be placed at the very front.

In the 3D rendering, however, the object closer to the viewpoint must be placed in front, and the distant object must be hidden by other objects. This is called hidden surface removal; and when **flush()** is called, rendering as well as hidden surface removal are performed.

As a result, the position of **obj1**, **obj2**, and **obj3** is verified, then rendering is performed so that the object closer to the viewpoint will be visible.

Step 2 — Moving the model

Step 2 moves the objects.
Use the dial’s **2**, **4**, **6**, and **8** keys, in order to move the object.
The following information is added to the **Canvas3D** class.

Step 2-1 Data member

Additional fields are as follows:

```
private AffineTrans modelTrans;
```

This is a transformation matrix for a model.
This data member was not necessary in the Step 1, because the model was located at the origin.

```
private int modelx = 0;
private int modely = 0;
private int modelz = 0;
public static final int MOVE_MODEL = 10;
```

MOVE_PLUS is a rate of change.
It specifies the size of movement when the key is pressed.

Step 2-2 Each method and process

keyPressed()

In order to receive the key input, event handlers are necessary.

```
protected void keyPressed(int kc) {
        switch (kc) {
        case Canvas.KEY_NUM4: // movel left
                modelx -= MOVE_MODEL;
                break;
        case Canvas.KEY_NUM6: // move right
                modelx += MOVE_MODEL;
                break;
        case Canvas.KEY_NUM2: // move up
                modely -= MOVE_MODEL;
                break;
        case Canvas.KEY_NUM8: // move down
                modely += MOVE_MODEL;
                break;
        default:
                break;
        }
}
```

initViewTrans()

```
modelTrans.setIdentity();
```

This initializes the transformation matrix for the model.
Please note what is calling this method.

paint()

```
modelTrans.m03 = modelx;
modelTrans.m13 = modely;
modelTrans.m23 = modelz;
viewTrans.mul(modelTrans);
```

This method specifies the model’s movement in the transformation matrix.
In order to translate the object in affine transformation, specify the values of x, y, and z coordinates in the **m03**, **m13**, and **m23** elements, respectively. It is necessary to calculate the products by separately creating **viewTrans** and **modelTrans**. This separate creation must be done because these are two different transformations, and then find the composite transformation matrix. Also, the product of matrices differs depending on the order of multiplication. Correct results cannot be obtained by **modelTrans.mul(viewTrans)**.

The **viewTrans**, which was found here as a result, performs transformation from the model coordinate system to the world coordinate system, as well as from the world coordinate system to the view coordinate system.
In order to render a model on the screen, the following transformation is necessary.

Screen coordinates = [Screen transformation matrix] * [View transformation matrix] * [World transformation matrix] * Model coordinates

(For the transformation on the screen, special methods such as **setPerspective()** or **setCenter()** will be used. Therefore, you do not have to pay attention to the order of multiplication in this API.)

Step 3 — Rotating the model

Step 3 adds the model's rotation functionalities.
In the previous steps, objects were only looked at from the front. In this step, however, the model can be rotated up, down, left, or right using the arrow keys.

Step 3-1 Data member

```
private AffineTrans tempTrans;
```

This is a temporary data to be used for matrix calculation.

```
private int spinx = 0;
private int spiny = 0;
public static final int SPIN_MODEL = 100;
```

SPIN_X_PLUS and **SPIN_Y_PLUS** are rotation speed.
Current rotation angles are controlled using **spinX** and **spinY**.

Step 3-2 Each method and process

keyPressed()

```
default:
        kc = getGameAction(kc);
        switch (kc) {
        case Canvas.UP: // roll up
                spinx -= SPIN_MODEL;
                break;
        case Canvas.DOWN: // roll down
                spinx += SPIN_MODEL;
                break;
        case Canvas.LEFT: // roll left
                spiny -= SPIN_MODEL;
                break;
        case Canvas.RIGHT: // roll right
                spiny += SPIN_MODEL;
                break;
        default:
                break;
        }
        while(spinx >= 4096) spinx -= 4096;
        while(spinx < 0) spinx += 4096;
        break;
```

The **getGameAction()** method obtains the direction key events.
Rotation angle is using the value where **4096 = 360 degrees**; use the value within the range of **0** to **4096**.

paint()

```
modelTrans.rotationX(spinx);
tempTrans.setIdentity();
tempTrans.rotationY(spiny);
modelTrans.mul(tempTrans);
```

This specifies the rotation in **modelTrans**.
Rotation around x-axis and rotation around y-axis are considered to be different transformations; therefore, separately calculate matrix products, then find the composite product.

Rotation transformation can be performed by directly specifying the values in the matrix; however, quite many values have to be specified.
Therefore, special API for this operation is provided.

Step 4 — Scaling

Step 4 enlarges, or reduces the model.
When the key **7** is pressed, the model is enlarged; and the key **9** reduces the model.

Step 4-1 Data member

Additional fields are as follows:

```
private int scalex = 4096;
private int scaley = 4096;
private int scalez = 4096;
public static final int SCALE_MODEL = 256;
```

Similar to rotation, **SCALE_MODEL** specifies a rate of change.
In default settings, scaling factor for each axis is **4096**, making the **1:1** scaling.

Step 4-2 Each method and process

Similarly, the following methods describe additional processes.

keyPressed()

```
case Canvas.KEY_NUM7: // zoom in
        scalex += SCALE_MODEL;
        scaley += SCALE_MODEL;
        scalez += SCALE_MODEL;
        break;
case Canvas.KEY_NUM9: // zoom out
        scalex -= SCALE_MODEL;
        scaley -= SCALE_MODEL;
        scalez -= SCALE_MODEL;
        break;
```

paint()

```
tempTrans.set(scalex, 0, 0, 0, 0, scaley, 0, 0, 0, 0, scalez, 0);
modelTrans.mul(tempTrans);
```

This specifies the value in the **AffinTrans** matrix.
In order to specify the scaling factor, provide the values in **m00**, **m11**, and **m22**, respectively.
Similar to previous steps, the composite matrix product is calculated, after finding the separate products.

Step 5 — Using the light

Step 5 specifies the settings for the light.
Settings in this step create the shadow where the light is weak or not present. In order to perform the process to shine a light on an object, you must also enable the light setting for the object.

For more details on how to specify the setting, please refer to the modeling tool's manual, and the plugin manual to be used when exporting BAC data.

In this step, the light is switched on or off, when the soft key is pressed.
The class in the Step 5 adds a declaration as a listener for receiving the command.

Step 5-1 Data member

```
Light light;
boolean lightEnabled;
```

The **Light** class manages light information. Also, a flag is needed so that it can indicate whether the light is currently enabled or not. A boolean variable manages this flag.

```
private Vector3D lightdir = new Vector3D(-3511, 731, 878);
private final int dirIntensity = 4096;
private final int ambIntensity = 2048;
```

Two types of lights can be specified: directional light, and ambient light.
Directional light requires direction and intensity. Ambient light does not have direction, and only requires intensity.

```
static final Command Light_CMD = new Command("Light", Command.EXIT, 2);
```

This is a button setting in order to turn on, or turn off the light.
This setting is specified for the soft key.

Step 5-2 Each method and process

Canvas3D

```
light = new Light(dir, dirIntensity, ambIntensity);
```

MascotCapsule V3 is capable of using one directional light, and one ambient light at the same time.
Therefore, a single **Light** object controls these lights.

```
addCommand(Light_CMD);
setCommandListener(this);
```

This is a GUI setting to turn on, or turn off the light.

commandAction()

This executes the command using the soft key.

```
public void commandAction(Command c, Displayable s) {
        if(c == Light_CMD) {
                if(lightEnabled){
                        lightEnabled = false;
                }else{
                        lightEnabled = true;
                }
        }
}
```

This is a setting to turn on, or turn off the light.

paint()

```
if(lightEnabled){
        effect.setLight(light);
}else{
        effect.setLight(null);
}
```

This is a light setting for the **Effect3D** object.
The rest of the process performs rendering the same as the previous steps.

Step 6 — Parallel projection and perspective projection

Previous steps only used parallel projection when rendering the shapes on the screen; however, parallel projection cannot represent the depth. On the other hand, perspective projection can clearly represent the sense of distance, because it changes the sizes depending on the distance. MascotCapsule is capable of using both, and Step 6 switches between these two projection methods.

Step 6-1 Data member

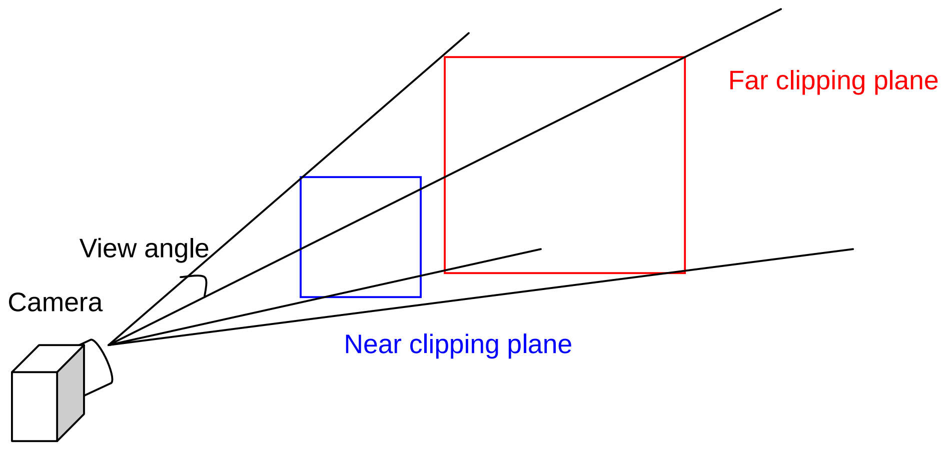
```
static final Command Perspective_CMD = new Command("Perspective",
Command.SCREEN, 1);
```

Firstly, this registers the command button that switches the projection method.

```
boolean persEnabled;
private final static int persNear = 1;
private final static int persFar = 4096;
private final static int persAngle = 682;
```

The boolean value stores the currently selected projection method.

The following three values are perspective projection’s attribute values— they are position of near clipping plane, position of far clipping plane, and view angle, respectively. Objects outside the range of these values will not be rendered.



Step 6-2 Each method and process

Canvas3D

```
addCommand(Perspective_CMD);
```

This adds the command button.

commandAction()

```
}else if(c == Perspective_CMD){
        if(persEnabled){
                persEnabled = false;
        }else{
                persEnabled = true;
        }
}
```

This switches the projection method when the button is used.

paint()

```
if(persEnabled){
        layout.setPerspective(persNear, persFar, persAngle);
}else{
        layout.setParallelSize(800, 800);
}
```

This specifies the projection method.

When the **setPerspective()** method is called, it not only specifies the perspective projection settings, but also automatically switches the projection method. On the other hand, the **setParallelSize()** method does the opposite.

Step 7 — Loading and using the animation

Previous steps only displayed the model; however, Step 7 uses animation.
Create both the animation and model using a modeling tool, and create the MTRA file based on the TRA file exported from the plugin.

Step 7-1 Data member

```
private ActionTable action;
private int frame = 0;
```

The **ActionTable** class loads the animation stored in the MTRA file. Also, multiple frames are contained in the action; therefore, this data member provides variables that store which frame is currently displayed.

Step 7-2 Each method and process

Canvas3D()

```
action = new ActionTable("/action_01.mtra");
```

This loads the MTRA file.

paint()

```
frame += action.getNumFrames(0)/10;
if( frame >= action.getNumFrames(0) ){
            frame = 0;
}
figure.setPosture(action, 0, frame);
```

The **ActionTable** object contains multiple animations (such as raising hands, walking, shaking the head, etc.).
The index number specifies which animation to be displayed; however, there is only one animation. Therefore, the index number is **0** in this case. Also, each animation consists of multiple frames.

In order to display the animation, the **Figure.setPosture()** method is used.
Programmers need to control which frame to display. Here, the frame number is 65536 multiplied by the number created during the modeling process. In this way, the status between the frames can also be displayed. The **ActionTable.getNumFrames()** method obtains how many frames are contained in the animation.

When the frame number becomes the value that is not contained in the animation data, it goes back to the beginning.
In this way, the animation will be repeated forever.

Step 8 — Using multiple animations

Step 8 increases the number of animations.
Step 7 displays a single animation by repeating it forever. In Step 8, however, the other animation starts when the key **1** is pressed.

Step 8-1 Data member

```
ActionTable action[] = new ActionTable[2];
int actNo;
```

The **ActionTable** class manages multiple actions.
A single MTRA file can actually store multiple actions; however, the MTRA file is divided here. In such cases, multiple **ActionTable** objects are also needed.

The **actNo** variable is provided in order to specify which **ActionTable** to be used.

Step 8-2 Each method and process

Canvas3D

```
action[0] = new ActionTable("/action_01.mtra");
action[1] = new ActionTable("/action_02.mtra");
```

There are two MTRA files; therefore, loading is performed as shown above.

keyPressed()

```
case Canvas.KEY_NUM1:
            actNo = 1;
            frame = 0;
            break;
```

When the key **1** is pressed, the animation **1** starts up.

paint()

```
frame += action[actNo].getNumFrames(0)/10;
if( frame >= action[actNo].getNumFrames(0) ){
            frame = 0;
            actNo = 0;
}
figure.setPosture(action[actNo], 0, frame);
```

When the animation (either **0** or **1**) finishes, the animation **0** starts up from the beginning.

Step 9 — Using multiple models

Previous steps only used a single model. Step 9 uses multiple models.

Background is rendered here. Background can be simply rendered in 2D; however, 3D model is rendered here for your information. The new model data for the background, and the texture data are required; therefore, it is necessary to load new files.

Step 9 uses a new MBAC file as a model data.
Texture data is also needed. The bitmap file, which has been used for the model, also contains the background data. Therefore, only the MBAC file is loaded here.

Step 9-1 Data member

```
Figure figureBg;
```

The only added element is **Figure**.

Step 9-2 Each method and process

Canvas3D

```
figureBg = new Figure("/test_model_haikei.mbac");
```

Same as the previous steps, the resource is loaded.

paint()

```
initViewParams();
modelTrans.m03 = 0;
modelTrans.m13 = 400;
modelTrans.m23 = -500;
viewTrans.mul(modelTrans);
layout.setAffineTrans(viewTrans);
g3.renderFigure(figureBg, 0, 0, layout, effect);
```

Same as the previous steps, the model is rendered.
Please note the difference in procedures because of multiple models. Compared with the processes before and after this code, the procedure for rendering a single object can be summarized as follows:

```
Initialization of the affine transformation matrix
Setting the value in matrix
layout.setAffineTrans()
(If there is animation)figure.setPosture()
g3.renderFigure()
```

This procedure is required for each single object.
After calling the **renderFigure()** method for all the objects, execute rendering using the **flush()** method.

Step 10 — Displaying primitives

MascotCapsule is also capable of rendering primitives.
In order to render primitives, model data creation is unnecessary. Instead, rendering is directly specified in the program. Primitives are not suitable for detailed designs; however, programmers have more freedom of control in rendering, etc.

Types of primitives are as follows: point, line, triangle, quadrilateral, and point sprite (image).
Depending on the profile, there are differences in procedures or supported primitive types. For more details, please refer to each profile’s programmer’s manual and reference manual.

Step 10-1 Data member

```
private static int command =
                Graphics3D.PRIMITVE_TRIANGLES |
                Graphics3D.PDATA_COLOR_PER_FACE |
                Graphics3D.PATTR_BLEND_HALF;
private int[] vertexcoords = {0, -300, 450, 300, -300, 450, 0, 0, 450};
private int[] normals = {0};
private int[] textureCoords = {0};
private int[] colors = {255<<16 | 255<<8 | 0};
```

In order to render primitives, you must specify the commands, vertex, normal, UV coordinate, and polygon color elements. In the first place, commands are needed. This example does not contain triangle, and no textures are applied. Instead, color polygons and semi-transparency process are specified.

The four lines below the commands are **int** arrays that specify respective data.

Step 10-2 Each method and process

paint()

```
g3.renderPrimitives(null, 0, 0, layout, effect,
            command, 1, vertexcoords, normals, textureCoords, colors);
```

The **renderPrimitives()** method is used instead of the **renderFigure()** method.
Instead of the **Figure** object, the primitive data created in the Step 10-1 will be passed.

MascotCapsule V3 Tutorial Document for MIDP

Version:	1.0.1
Publication Date:	November 27, 2006
Publisher:	HI CORPORATION Meguro Higashiyama Bldg. 5F Higashiyama 1-4-4 Meguro-ku, Tokyo 153-0043, Japan http://www.hicorp.co.jp/

– Trademarks, copyrights, and disclaimer –

- This document is protected by copyright law. Neither the whole nor any part of the information (including software and program) contained in this document may be copied, reproduced, or reprinted in any material form without the written permission from HI CORPORATION.
- MascotCapsule® is the registered trademark of HI CORPORATION in Japan. Other brand names and product names in this document are trademarks or registered trademarks of their respective owners.
- HI CORPORATION is not responsible for any problem, loss, or damages that may be caused by utilizing the materials or information contained in this document.
- The contents of this document are subject to change for revision without prior notice.

© 2007 HI CORPORATION. All Rights Reserved.