M3G Chapter 6. Dynamically Textured Billboards

M3G chapter 5 uses the AnimBillboard class to display a series of GIFs as textures on a square mesh. AnimBillboard is a variant of the Billboard class from M3G chapter 4, which displays a single image.

Both AnimBillboard and Billboard stay z-axis aligned with the current camera position, and their images can have transparent elements.

This chapter describes yet another type of Billboard, DynaBoard. In common with AnimBillboard, it creates a square mesh of length size, resting on the XZ plane at the origin, that stays aligned with the camera. However, its rendered images are generated dynamically at execution time.

The DynaM3G application illustrates two uses of DynaBoard, both shown in Figure 1.

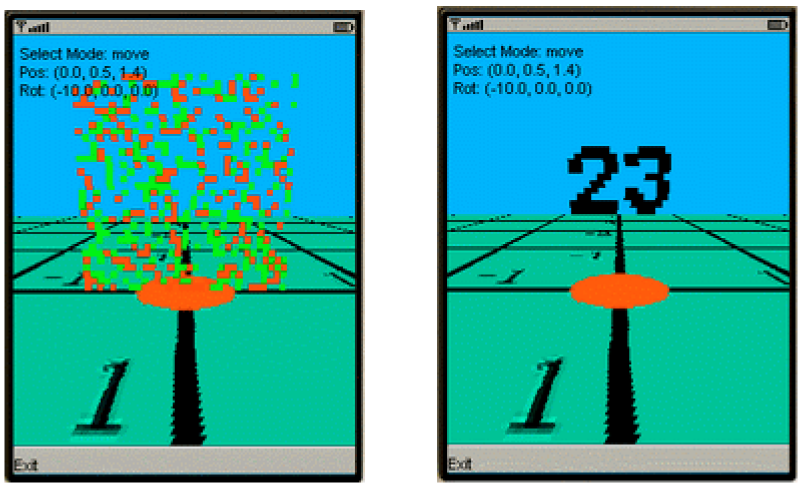

Figure 1. Two Versions of DynaM3G

The version of DynaM3G on the left of Figure 1 displays a constantly changing random patchwork of red and green squares. The second version (which requires a few lines of code be changed in DynaBoard), renders a counter, which progresses from 0 to 99, then repeats.

DynaBoard functionality is useful in situations where the nature of the rendered image can't be determined until the application is running. For example, images containing user names can't generally be created ahead of time. DynaBoard also allows an application to offer more variety, since the choice of images isn't limited to a predetermined selection. For instance, an explosion can look different each time, and error messages can be tailored to the particular scenario that caused them.

There is a downside to this flexibility: processing time must be expended on creating the images.

1. Class Diagrams for DynaM3G

Figure 2 shows the class diagrams for the DynaM3G application. The class names and public methods are shown.

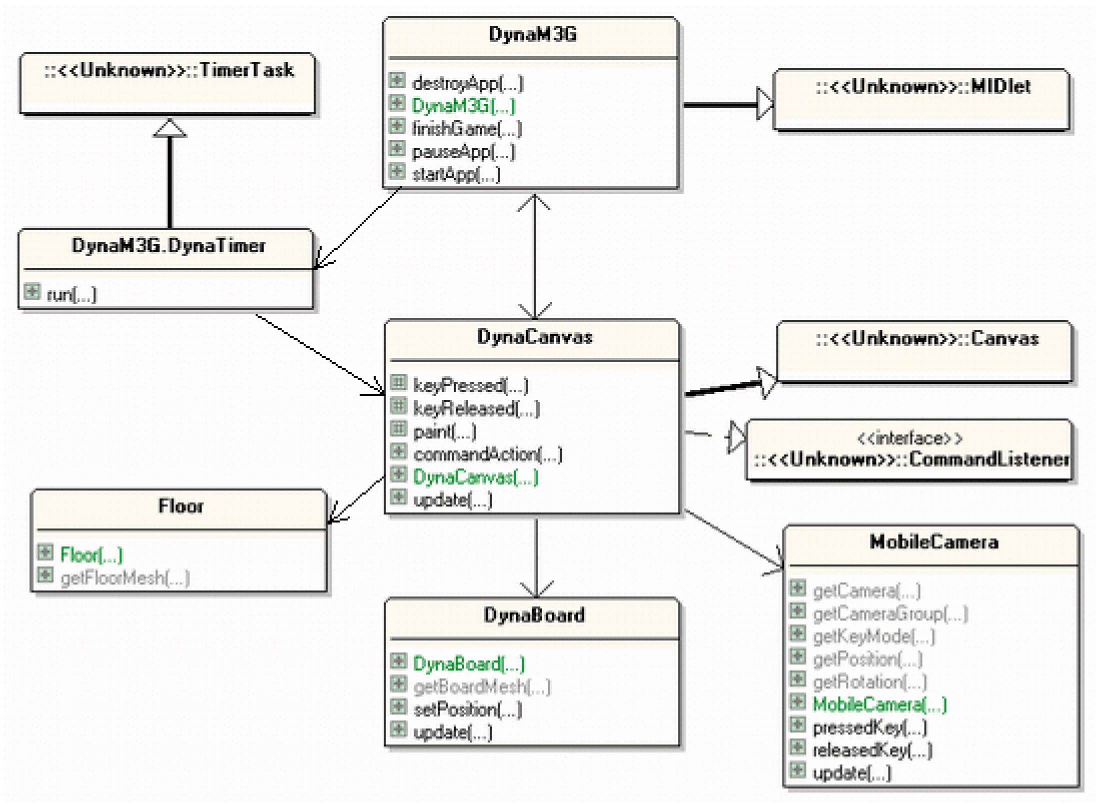

Figure 2. Class Diagrams for DynaM3G.

DynaM3G is similar to earlier examples, borrowing many classes from MorphM3G in M3G chapter 3. DynaM3G is the top-level MIDlet, and uses a TimerTask inner class (DynaTimer) to keep triggering the update() method in DynaCanvas, which updates the scene and redraws it.

The Floor class is the one used in M3G chapters 2 and 3 (the AnimM3G and MorphM3G examples): it creates a square aligned with the XZ plane, centered at (0,0), with an image wrapped over it as a texture.

MobileCamera is the mode-based camera, last seen in M3G chapter 3, which allows the user to move, rotate, and float around the scene.

2. The DynaCanvas Class

The 'canvas' class' main tasks (as usual) are to build the scene, respond to key presses, and to periodically update the scene. The scene graph constructed by DynaCanvas appears in Figure 3.

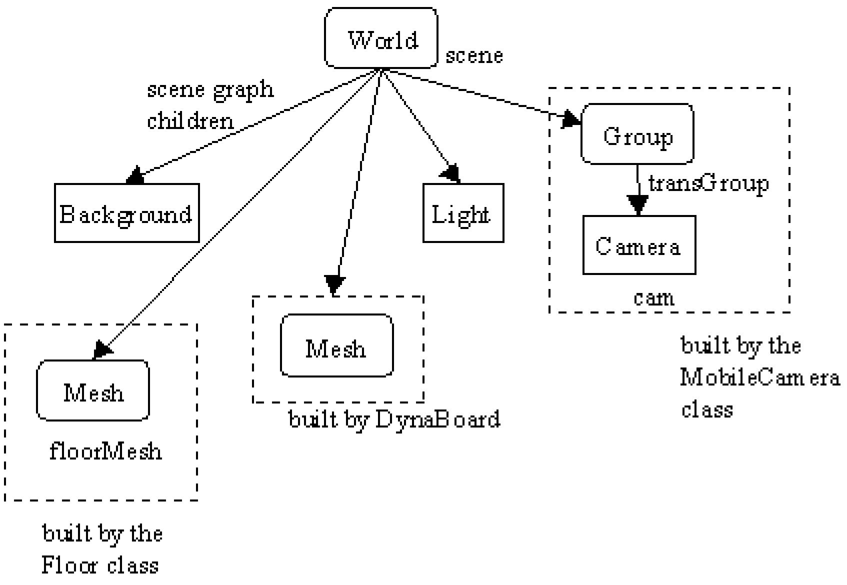

Figure 3. Scene Graph for DynaM3G.

DynaCanvas uses buildScene() to create the scene:

```
private void buildScene()
{ addCamera();
  addLights();
  addBackground();
  addFloor();

  addDynaBoard();
}
```

Most of these methods have been seen several times before. For example, in M3G chapter 3. The only new method is addDynaBoard():

```
// global variable
private DynaBoard dynaBoard;   // the dynamic billboard


private void addDynaBoard()
{
  Group camGroup = mobCam.getCameraGroup();
  dynaBoard = new DynaBoard(camGroup, 1.0f);
  scene.addChild( dynaBoard.getBoardMesh() );
}  // end of addDynaBoard()
```

The DynaBoard instance requires a reference to the mobile camera (mobCam), so it can keep itself aligned. The second argument is the size of the billboard.

DynaCanvas' update method() updates the camera's position and the billboard:

```
public void update()
// called by a timer task in DynaM3G
{
  mobCam.update();
  dynaBoard.update();   // generate a new image for the board
  repaint();
}
```

The call to update() in DynaBoard gives the object the opportunity to update/change its image.

3. The Dynamic Billboard

A DynaBoard is a square of length size, sitting at the origin on the XZ plane. The board can be positioned with setPosition(), and keeps its z-axis aligned with the camera position.

The texture wrapped over the board is dynamically created when the board is created, and updated when the board's update() method is called.

The texture is constructed at run time by drawing into an Image object, acting as an off-screen buffer. The object is converted to an Image2D instance, and used to initialize a Texture2D variable. As the Image is changed to Image2D, any white parts in the picture are set to be transparent.

I'll begin by describing the version of DynaBoard which displays a randomly changing mix of red and green squares as its texture (see Figure 1). The other version of DynaBoard, which shows a repeating counter, is explained in section 4.

DynaBoard's constructor is very similar to AnimBillboard in M3G chapter 5:

```
// globals
private final static int IM_SIZE = 32; // size of image square, im

private Group cameraGroup;   // used for camera alignment
private Mesh bbMesh;
private Appearance app;

// image related objects
private Image im;
private Graphics graphicIm;
private int[] pixels;
private byte[] imBytes;

private Random rnd;    // used to select random colours


public DynaBoard(Group camGroup, float size)
{
  cameraGroup = camGroup;
  // build the mesh
  VertexBuffer vb = makeGeometry();

  int[] indicies = {1,2,0,3};  // one quad
  int[] stripLens = {4};
  IndexBuffer ib = new TriangleStripArray(indicies, stripLens);

  // initialize global image objects
  im = Image.createImage(IM_SIZE, IM_SIZE); // holds the image
  graphicIm = im.getGraphics();            // graphics context for im
  pixels = new int[IM_SIZE * IM_SIZE];     // holds pixels from im
  mBytes = new byte[4 * pixels.length];
                            // holds bytes taken from the pixels

  rnd = new Random();

  // set the board's initial appearance
  app = makeAppearance( dynamicTexture() );    // random colours

  bbMesh = new Mesh(vb, ib, app);

  // scale and position the mesh
  float scale2 = size * 0.5f;   // to reduce 2 by 2 image to 1 by 1
  bbMesh.scale(scale2, scale2, scale2);
  bbMesh.setTranslation(0, scale2, 0);

  bbMesh.setAlignment(cameraGroup, Node.Z_AXIS, null, Node.NONE);
             // only use z-axis alignment with the camera
}  // end of DynaBoard()
```

makeGeometry() builds a square centered at the origin on the XZ plane with sides of 2 units (which explains the additional scale factor of 0.5 later on in the constructor).

makeAppearance() produces a texture-based appearance, which only shows the opaque parts of the texture supplied by dynamicTexture():

```
private Appearance makeAppearance(Texture2D tex)
// The appearance comes from the transparent texture in tex.
{
  app = new Appearance();    // no material

  // Only display the opaque parts of the texture
  CompositingMode compMode = new CompositingMode();
  compMode.setBlending(CompositingMode.ALPHA);
  app.setCompositingMode(compMode);

  app.setTexture(0, tex);

  return app;
}  // end of makeAppearance()
```

3.1. Updating the Board

An update to the board entails two tasks: making sure the board is aligned with the current camera position, and generating a new texture. These are carried out by DynaBoard's update() method:

```
public void update()
{
  bbMesh.align(cameraGroup);   // update alignment
  app.setTexture(0, dynamicTexture());    // new random colours
}
```

The new texture is created by calling dynamicTexture() again.

3.2. Making a Dynamic Texture

dynamicTexture() creates the board's texture in a series of steps outlined in Figure 4.

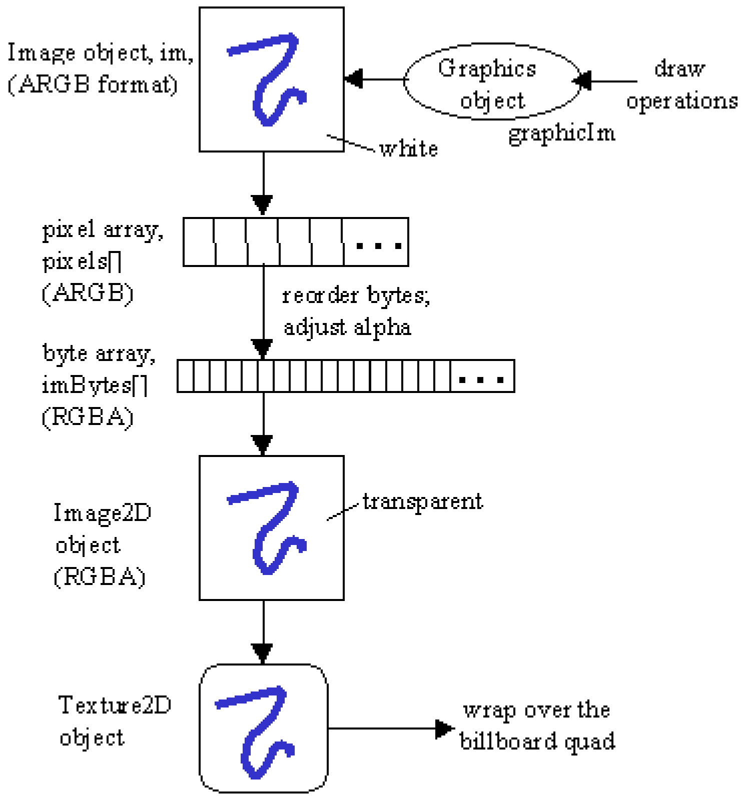

Figure 4. Creating a Texture Dynamically.

Many of the data structures mentioned in Figure 4 are globally defined, to avoid the numerous temporary objects that would be created otherwise. These data structures are initialized in the DynaBoard() constructor:

```
// globals
private Image im;
private Graphics graphicIm;
private int[] pixels;
private byte[] imBytes;


im = Image.createImage(IM_SIZE, IM_SIZE); // holds the image
graphicIm = im.getGraphics();            // graphics context for im
pixels = new int[IM_SIZE * IM_SIZE];     // holds pixels from im
```

```
imBytes = new byte[4 * pixels.length];
                              // holds bytes taken from the pixels
```

The Image object, im, is used as a 'scratch pad' (off-screen buffer) for the image used by the texture; it's created via draw operations applied to the graphicIm graphics context. The image's pixels are extracted as an array of integers, then manipulated and stored in the imBytes[] byte array. This array is used to create an Image2D object, which is employed to initialize a Texture2D instance.

These stages are packaged into three methods called by dynamicTexture():

```
private Texture2D dynamicTexture()
// create a texture from a random mix of colours
{
  makeRandomImage();    // initialise global Image object, im
  Image2D im2D = convertImage(im);      // Image --> Image2D
  Texture2D tex = createTexture(im2D);  // Image2D --> Texture2D
  return tex;
}
```

3.3. Drawing a Random Image

The Image object has dimensions (IM_SIZE, IM_SIZE), a fact used extensively in makeRandomImage(). The image's background is turned white, and red and green squares are randomly placed over the top.

```
private void makeRandomImage()
{
  // a white background
  graphicIm.setColor(0xFFFFFF);  // white
  graphicIm.fillRect(0, 0, IM_SIZE, IM_SIZE);

  int numPixels = IM_SIZE * IM_SIZE;

  // draw a series of randomly placed red squares
  graphicIm.setColor(0xFF0000);  // red
  for (int i=0; i < numPixels/4; i++)
    graphicIm.fillRect( rnd.nextInt(IM_SIZE), rnd.nextInt(IM_SIZE),
                              1, 1);

  // draw a series of randomly placed green squares
  graphicIm.setColor(0x00FF00);  // green
  for (int i=0; i < numPixels/4; i++)
    graphicIm.fillRect( rnd.nextInt(IM_SIZE), rnd.nextInt(IM_SIZE),
                              1, 1);
} // end of makeRandomImage()
```

The Image object is accessed indirectly via its graphics context, graphicIm.

White has a special meaning: when the Image object is transformed into an Image2D instance, any white elements will become transparent. As a consequence of setting the background to white, it will 'disappear' when the DynaBoard object is rendered.

3.4. Converting an Image to Image2D

A M3G texture requires its source image to be an Image2D object, so the constructed Image object must be converted. This requires Image's ARGB pixel format to be reordered to the RGBA format supported by Image2D.

The RGB components (R=red, G=green, B=blue) will be moved across to Image2D without change, but the alpha (A) value requires slightly more work. If the combined RGB colour for a pixel is white, then the alpha will be switched 'full on' so the pixel becomes transparent, otherwise its value will be set to 0 so the pixel is opaque.

As Figure 4 indicates, the translation of Image to Image2D uses two intermediate steps, first the pixels are extracted from the Image object, then copied (with some modifications) to a byte array, which is employed to fill the Image2D instance.

convertImage() shows these steps in details:

```
private Image2D convertImage(Image im)
{
  int alphaVal, redVal, greenVal, blueVal;

  // extracts ARGB pixel data from im into pixels[]
  im.getRGB(pixels, 0, IM_SIZE, 0, 0, IM_SIZE, IM_SIZE);

  // store pixels in byte array, imBytes[]
  for(int i=0; i < pixels.length; i++){
    // extract RGB components from the current pixel as integers
    redVal = (pixels[i]>>16)&255;
    greenVal = (pixels[i]>>8)&255;
    blueVal = pixels[i]&255;

    // set alpha to be transparent if colour is white
    alphaVal = 255;   // opaque by default
    if((redVal==255) && (greenVal==255) && (blueVal==255))  // white?
      alphaVal = 0; // fully transparent

    // store components in RGBA order in byte array
    imBytes[i*4] = (byte)redVal;
    imBytes[(i*4)+1] = (byte)greenVal;
    imBytes[(i*4)+2] = (byte)blueVal;
    imBytes[(i*4)+3] = (byte)alphaVal;
  }

  // use the byte array to create Image2D object
  Image2D im2D = new Image2D(Image2D.RGBA, IM_SIZE,IM_SIZE, imBytes);
  return im2D;
}  // end of convertImage()
```

The main work of convertImage() is carried out by its loop, which examines each integer (pixel) in pixels[], and stores four bytes in imBytes[]. A single pass through the loop is illustrated by Figure 5.

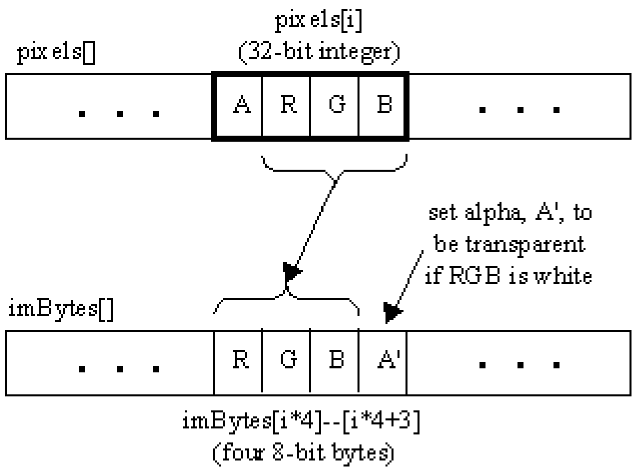

Figure 5. Integer to Bytes Conversion.

The RGB components of the pixel are extracted as integers:

```
redVal = (pixels[i]>>16)&255;
greenVal = (pixels[i]>>8)&255;
blueVal = pixels[i]&255;
```

Each integer has a value between 0 and 255, corresponding to its byte value in the pixel. The integers can be easily tested, and readily converted back to bytes when added to imBytes[].

3.5. Making a Texture

The aim is to wrap the texture over the entire billboard quad, and have its colour values replace any inherent in the mesh:

```
private Texture2D createTexture(Image2D im2D)
{
  Texture2D tex = new Texture2D(im2D);
  tex.setFiltering(Texture2D.FILTER_NEAREST,
                             Texture2D.FILTER_NEAREST);
  tex.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
  tex.setBlending(Texture2D.FUNC_REPLACE);

  return tex;
}
```

Although the texture has transparent elements, it's still necessary to switch on alpha compositing in the shape's Appearance node for those transparency parts to 'disappear' on screen. This is done by makeAppearance() described above.

4. The Counter Version of DynaBoard

The right hand image in Figure 1 shows the other version of DynaBoard: a counter that iterates from 0 to 99, then repeats.

The changes to DynaBoard are fairly small. The counter is represented by a global integer, as is the font used for drawing the counter into the Image.

```
// globals
private final static int MAX_COUNTER = 100;
private int counter;
private Font font;
```

The counter and font are initialized in the constructor:

```
// in the DynaBoard constructor
counter = 0;
font = Font.getFont(Font.FONT_STATIC_TEXT,
                    Font.STYLE_BOLD, Font.SIZE_LARGE);
```

dynamicTexture() is still called in two places: once in the constructor when the billboard is first created, and also in update() when the texture is regenerated. counter is passed to dynamicTexture() as a String.

```
// in the DynaBoard's constructor
app = makeAppearance( dynamicTexture(""+counter));


public void update()
{
  bbMesh.align(cameraGroup);   // update alignment

  if (counter == MAX_COUNTER)
    counter = 0;    // reset the counter
  app.setTexture(0, dynamicTexture("" + counter));
  counter++;
}
```

dynamicTexture() calls makeStringImage() to draw the string onto the image. The conversion steps from Image to Texture2D (convertImage(), createTexture()) are unchanged from the first version of DynaBoard:

```
private Texture2D dynamicTexture(String str)
// example 2: create a texture using a string
{
  makeStringImage(str);     // initialise global Image object, im
  Image2D im2D = convertImage(im);      // Image --> Image2D
  Texture2D tex = createTexture(im2D);  // Image2D --> Texture2D
  return tex;
}
```

makeStringImage() draws the string in the middle of the image using a large bold black font.

```
private void makeStringImage(String str)
{
  // a white background
  graphicIm.setColor(0xFFFFFF);  // white
  graphicIm.fillRect(0, 0, IM_SIZE, IM_SIZE);

  // set the font to be black, large, and bold
  graphicIm.setColor(0x000000);  // black
  graphicIm.setFont(font);

  // draw the string in the center of the image
  int strWidth = font.stringWidth(str);
  int strHeight = font.getHeight();
  int xOff = (IM_SIZE - strWidth)/2;
  int yOff = (IM_SIZE - strHeight)/2;
  graphicIm.drawString(str, xOff, yOff, Graphics.TOP|Graphics.LEFT);
}  // end of makeStringImage()
```

Once again, I made the background white so it will disappear when the board is rendered.

makeStringImage() can be called with any string, not just the counter value.

5. Creating a Single Dynamic Texture

The two examples of DynaBoard shown here keep changing the board's image, which means that dynamicTexture() is called repeatedly.

In fact, there are many situations where a dynamic texture only needs to be created once. For example, the original motivation for this work was the need by my students to create "name labels" for users joining a networked game. The names aren't known until run time, so predefined images can't be employed. However, once a label has been generated, there's no need to change it.

This "create once, use forever" requirement is easily met by only calling dynamicTexture() once, in the constructor of DynaBoard. The subsequent calls in update() can be removed.

This change improves the performance of the application, since there's no repeated image generation. It also allows the coding style to be modified.

DynaBoard uses many global variables related to image creation. For instance, the counter version of DynaBoard employs:

```
// globals
// image related
private Image im;            // for holding the dynamic image
private Graphics graphicIm;  // the graphics context for im
private int[] pixels;        // holds pixels extracted from im
private byte[] imBytes;      // holds bytes extracted from pixels[]

// drawing related
private int counter;
private Font font;
```

These variables are global due to the repeated calls to makeStringImage() and convertImage() in dynamicTexture() (once for each screen update). Making these variables local to the methods would lead to the creation of numerous temporary objects that would need garbage collecting frequently. The impact on running time could be considerable.

However, if dynamicTexture() is only going to be called once, then it's better to move these variables into their methods, both for understandability, and so they can be garbage collected easily when their usefulness is over.