By King (http://www.gizzedinc.com)

OpenGL|ES Tutorials (9): TGA / Bitmapped Text

TGA Files

In this tutorial we will load TGA Files. TGA stands for Targa Files. The importance of TGA Files is that they can hold images which contain transparent pixels. Combined with the information you learnt in Tutorial 5 (Blending ant Text), you will learn how to load TGA Files as textures, and then remove the transparent parts using Blending. Transparency can be an invaluable tool, Imagine trying to move a Character along a scene in a 2D Game. Unless your character is a complete square/rectangle, then you would need to produce hundreds of bitmap images to do the same thing a one TGA file would do.

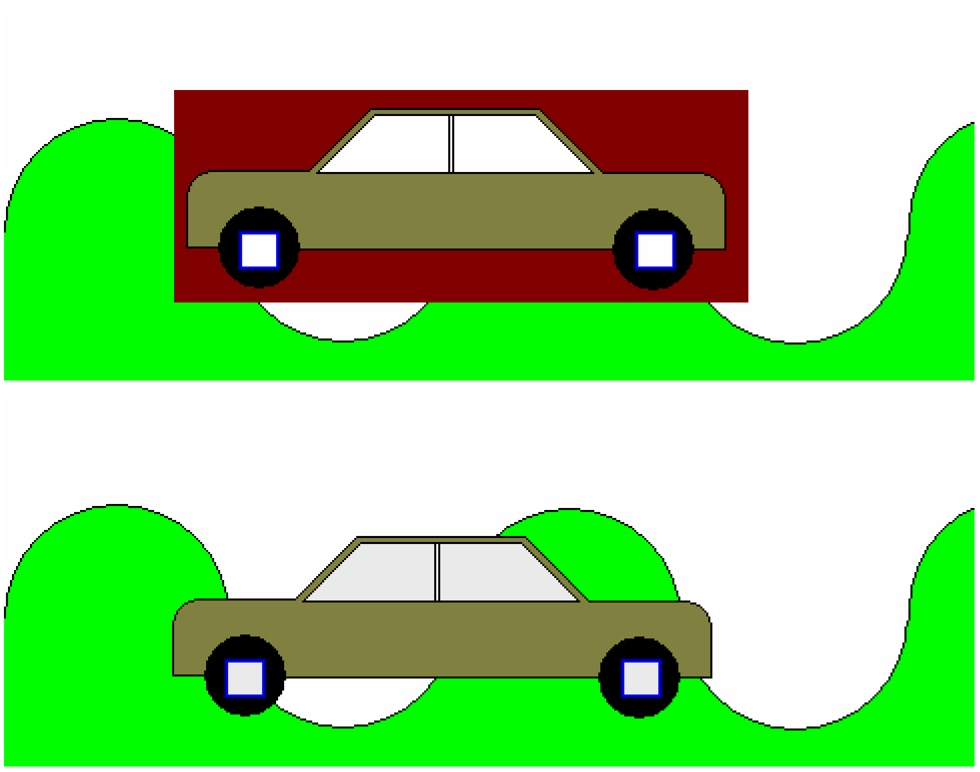


Figure 9.1 Comparison of Loading a Bitmap image, with a TGA File with an alpha channel.

It is also important to note that to define areas of a TGA File transparent, you must specify an Alpha Channel. This will be discussed about later in the tutorial.

Bitmapped Text

TGA Files is the main principle of Bitmapped Text. What we are going to do is load up a TGA File, like the one below, and then specify texture co-ordinates to select only one character.

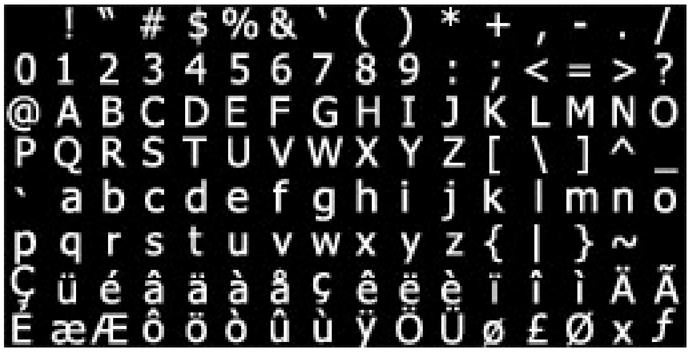

! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [\ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
Ç ü é â ä à å ç ê ë è ï î ì Ä Ã
É æ Æ ô ö ò û ù ÿ Ö Ü ø £ Ø x ƒ

Figure 9.2 The Font Blue Print, for the Font: Tahoma

Essentially, this file containing every letter is just like a Windows Font File (.ttf). Without this file you wouldn't be able to write text, just as you can't write in a Font in Microsoft Word, if you don't have the .ttf file. Furthermore, you can get more than one font file, just as you can type in more than one font in Microsoft Word.

Introduction

This tutorial will show you *how* the Bitmapped Font Class works. However, it is not necessary to know this, as you will see at the end of the tutorial. All this is wrapped up for you and included in a Header File as part of the KGSDK.

Besides the Functions.h Header File, I have also added another Header File (BitmappedText.h), which contains code specific to the Bitmapped Font Class.

File: main.cpp

Declarations

```
#include "Functions.h"      // Non-relevant drawing code in this header
```

Analysis: Again, we just have to include the Functions Header File, as it contains all declarations etc. It also links to out new Header File: BitmappedText.h as well.

Init

```
void Init()
{
   glEnable(GL_LIGHTING);              //Enable Lighting
```

```
        glEnable(GL_LIGHT0);                //Enable GL_LIGHT0
        glEnable(GL_COLOR_MATERIAL);        //Enable Colour Material
        glEnable(GL_BLEND);                 //Enable Blending
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glDisable(GL_CULL_FACE);            //Disable Back face culling

        if (!LoadTextures())
        {
             MessageBox(NULL, L"Error loading textures", L"Error", MB_OK);
        }
        else
        {
             glEnable(GL_TEXTURE_2D);
        }

        glLoadIdentity();

        GLfixed mat[4][4];

        float nearz, farz;

        farz  = 300.0f;
        nearz = 0.01f;

        memset(mat, 0, sizeof(mat));
        mat[0][0] = (int) (65536.0f * 2.4f);
        mat[1][1] = (int) (65536.0f * 3.2f);
        mat[2][2] = (int) (65536.0f * (farz / (farz - nearz)));
        mat[2][3] = (int) (65536.0f * 1.0f);
        mat[3][2] = (int) (65536.0f * ((-farz * nearz) / (farz - nearz)));

        glMatrixMode(GL_PROJECTION);
        glLoadMatrixx(&mat[0][0]);

        rot[0] = 0;
        rot[1] = 0;
        rot[2] = 0;
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 3;

    }
```

Analysis: Nothing new here, but it is important to note Blending is turned on, and the Blending Function is set to: `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`. The blending function must be set to this, or otherwise our TGA Transparency loading won't work.

Function: DrawMyText()

```
    void DrawMyText()
    {
       glRotatef(xrot,1.0f,0.0f,0.0f); //Rotate axis based on value of xrot
       glRotatef(yrot,0.0f,1.0f,0.0f); //Rotate axis based on value of yrot

       BitmappedText MyText("King Rules!!");  //Create text object

       MyText.RValue= 1.0f;                   //Red value of colour
       MyText.GValue= 1.0f;                   //Green value of colour
       MyText.BValue= 1.0f;                   //Blue value of colour
       MyText.AValue= 1.0f;                   //Alpha value of colour
       MyText.FontName = Tahoma;              //Font to use: Tahoma
       MyText.LocX = -0.5f;                   //Don't Move on X-Axis
```

```
14:       MyText.LocY = 0.0f;                     //Don't Move on Y-Axis
15:       MyText.LocZ = 0.0f;                     //Don't Move on Z-Axis
16:       MyText.SizX = 0.35f;                    //Don't Scale on X-Axis
17:       MyText.SizY = 0.35f;                    //Don't Scale on Y-Axis
18:       MyText.SizZ = 0.35f;                    //Don't Scale on Z-Axis
19:       MyText.TextureFilter = Use_GL_NEAREST; //Apply GL_NEAREST Texture
                                                 filter to font
20:
21:       MyText.FrameworkKing_DrawText();        //Draw it !
22:    }
```

Analysis: The first two lines of this function rotate the scene, as usual. Line 6 creates a BitmappedText Object (the one that is defined later in the BitmappedText.h Header File). Note that the Constructor of this class takes one parameter, which is the Text you wish to display. If need be, this can be changed later on. Lines 8 – 19 set the properties of the Text Object. Line 8 – 11 sets the colour. Line 12 sets the Font. Note that the value 'Tahoma' is defined in an enumeration in our header file, along with the .ttf file, as you will see later. You can't just write a name of font (e.g. "Times New Roman"), then expect it to work without files & definition. Lines 13 – 15 set the location of the text in 3D Space, on the X, Y and Z axis's respectively. In this example, we only move the text 0.5 to the left. Lines 16 – 18 set the size of the object. It scales the object on the X, Y and Z axis's respectively. 1.0f, means 100%, so there is no scaling. If it was 50%, the text would be half-size. I've scaled this text down to 0.35f (35% of original size). Line 19 sets the Texture Filter to be used, when rendering the Font. You will notice the value is set to Use_GL_NEAREST. This is also defined in an enumeration in our header file. Finally on Line 21, the text is drawn. In a real program you would define lines 6 – 19 in the Initialisation part of the code, unless it is absolutely necessary to create the object/set the properties on every frame.

Render

```
void Render()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   DrawMyText();           //Draw Strips onto screen

   glFinish();
   eglSwapBuffers(FrameworkGL_GetDisplay(), FrameworkGL_GetSurface());
}
```

Analysis: Our render function remains unchanged. We have only updated it so that it now calls function DrawStrips instead of function DrawCube as it did previously.

Update

```
1:     void Update()
2:     {
3:       GLfixed mat[4][4];
```

```

        memset(mat, 0, sizeof(mat));
        mat[0][0] = (int) (65536.0f *  cos(rot[0]));
        mat[0][2] = (int) (65536.0f *  sin(rot[0]));
        mat[1][1] = (int) (65536.0f *  cos(rot[1]));
        mat[1][2] = (int) (65536.0f *  sin(rot[1]));
        mat[2][0] = (int) (65536.0f * -sin(rot[0]) * cos(rot[1]));
        mat[2][2] = (int) (65536.0f *  cos(rot[0]) * cos(rot[1]));
        mat[3][2] = (int) (65536.0f * loc[2]);
        mat[3][3] = 65536;

        glMatrixMode(GL_MODELVIEW);
        glLoadMatrixx(&mat[0][0]);

        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_LEFT_SHOULDER ))
             loc[2] += 0.3f;
        // If the Left Shoulder Button has been pressed: Zoom out
        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_RIGHT_SHOULDER))
             loc[2] -= 0.3f;
        // If the Right Shoulder Button has been pressed: Zoom In
        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_DOWN     ))
             xrot -= 1.0f;
        // If the Down button (DPAD) has been pressed: Rotate the Plane downwards
        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_UP       ))
             xrot += 1.0f;
        // If the Up button (DPAD) has been pressed: Rotate the Plane upwards
        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_LEFT     ))
             yrot -= 1.0f;
        // If the Left button (DPAD) has been pressed: Rotate the Plane towards the left
        if (Framework_IsButtonPressed(FRAMEWORK_BUTTON_DPAD_RIGHT     ))
             yrot += 1.0f;
        // If the Right button (DPAD) has been pressed: Rotate the Plane towards the Right
    }
```

Analysis: No change here. Simply rotate the scene in respect to which D-Pad button has been pressed.

Main

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow )
{
   if (!Framework_Init(hInstance, hPrevInstance))
   {
       return 0;
   }

   if (!FrameworkGL_Init())
   {
       Framework_Close();
       return 0;
   }

   //Initialise the 2D Framework.
   if (!Framework2D_Init())
   {
       Framework_Close();
       return 0;
   }

```

```
        Init();

        do
        {
            //If text is displayed on the screen, dont draw the strips
            if (!TextDisplayed)
            {
            Render();
            }

            Update();

            Framework_ProcessMessages();

            if (!Framework_IsAppActive())
            {
                  while (!Framework_IsAppActive())
                  {
                        Framework_ProcessMessages();
                        Sleep(50);
                  }
            }

        } while (!Framework_IsButtonPressed(FRAMEWORK_BUTTON_HOME));

        //Clean Up
        FrameworkGL_Close();
        Framework2D_Close();
        Framework_Close();

        return 0;
    }
```

Analysis: Also no change here. Same old Main message loop.

File: Functions.h

Declarations

```
#include <windows.h>
#include <GLES\gl.h>
#include <GLES\egl.h>
#include "KGSDK\framework.h"
#include "KGSDK\framework2D.h"
#include "KGSDK\frameworkGL.h"

#pragma comment(lib, "Framework.lib")
#pragma comment(lib, "libGLES_CM.lib")
#pragma comment(lib, "dxguid.lib")
#pragma comment(lib, "ddraw.lib")

static float rot[3];
static float loc[3];

float xrot = 0.0f;
float yrot = 0.0f;

#define NUMBER_OF_COLUMNS 16.0f;     //Number of columns in the Font Blue
print
```

```
20:  #define NUMBER_OF_ROWS 8.0f;         //Number of rows in the Font Blue
print
21:
22:  const int NumberOfFonts = 2;         //Number Of Fonts
23:  char *FontFiles[] =
     //Filename of Fonts
24:  {
25:      "KGSDK_Tahoma_Font.tga",
26:      "KGSDK_Twentieth_Century_Poster_1_Font.tga"
27:  };
28:
29:  GLuint texture[NumberOfFonts];       //Storage for two textures (two
     fonts)
30:
31:  enum { Tahoma, TCP1} AvailableFonts;
     //Enumeration of the Fonts we have loaded

32:  enum {Use_GL_NEAREST, Use_GL_LINEAR} Texture_Filters;
     //Enumeration of the Texture Filters
33:
34:    struct TGAFile                     //Structure of a TGA File
35:    {
36:               unsigned char typeCode; //Format of the File
37:               short int width;        //Width of the File
38:               short int height;       //Height of the File
39:               unsigned char bpp;      //Bits per pixel
40:               GLenum format;          //Format of the File
41:               unsigned char *imageData;//Image Data
42:    };
43:
44:  #include "BitmappedText.h"
```

Analysis: Lines 1 – 11 link the necessary Header Files/Libraries. On Line 19 and 20 there are two pre-processor directives. One defines the number of columns in our font file (In this case 16), and the other defines the number of rows in our font file (In this case 8). On Line 22 a constant variable is defined to hold the number of Fonts. Line 23 is an array holding the filenames of the TGA Font Blueprint Files. Line 29 creates storage space for our textures, but it creates it upon the value of NumberOfFonts. Remember the Tahoma Value and Use_GL_NEAREST Value used in the Render section of main.cpp ? They are defined on lines 31, which also contains a TCP1 value (Twentieth Century Poster 1 Font), and line 32, which also contains a Use_GL_LINEAR Value. Unlike Bitmaps, there is no predefined structure for TGA Files. This is why we create our own on lines 34 – 42. The structure of a TGA file is discussed below in the LoadUncompressedTGA analysis.

Function: LoadUncompressedTGA

```
1:    bool LoadUncompressedTGA(char *filename, TGAFile *tgaFile)
2:    {
3:       FILE *file;                          //TGA File we're loading
4:
5:       unsigned char tmpRGB;                //Temporary variable used when
                                              converting from BGR -> RGB
6:
7:       unsigned char ucNull;                //Recycle-Bin for 'unsigned char'
                                              elements of the TGA File
8:       unsigned short usNull;
```

```

TCHAR path[256];                          //Path
char fullPath[256];                       //Path
GetModuleFileName(NULL, path, 256); //Get Current Working Directory

TCHAR *pos = wcsrchr(path, '\\');   //Find last \\ so that we can remove the last part
*(pos + 1) = '\0';                        //Add NULL Character to position found, effectively removing last part

wcstombs(fullPath, path, 256);      //Convert Wide characters to multi-byte

strcat(fullPath, filename);         //Combine path with filename

file = fopen(fullPath,"rb");        //Open File
if (!file)                                //Check validity of file
{
   MessageBox(NULL, L"Can't Find TGA", L"Error", MB_OK);
   return false;
}

fread(&ucNull,sizeof(unsigned char),1,file);   //Discard idLength variable
fread(&ucNull,sizeof(unsigned char),1,file);   //Discard colorMapType variable

fread(&tgaFile->typeCode,sizeof(unsigned char),1,file);
//Save imageType variable to TGAFile structure

//Make sure the File is an uncompressed TGA File
if (tgaFile->typeCode != 2 && tgaFile->typeCode != 3)
{
   MessageBox(NULL, L"Incorrect texture type", L"Error", MB_OK);
   fclose(file);
   return false;
}

fread(&usNull,sizeof(unsigned short),1,file);  //Discard colorMapOrigin variable
fread(&usNull,sizeof(unsigned short),1,file);  //Discard colorMapLength variable
fread(&ucNull,sizeof(unsigned char),1,file);   //Discard colorMapEntrySize variable
fread(&usNull,sizeof(unsigned short),1,file);  //Discard originX variable
fread(&usNull,sizeof(unsigned short),1,file);  //Discard originY variable

fread(&tgaFile->width, sizeof(unsigned short), 1, file);
//Save Width variable to TGAFile structure
fread(&tgaFile->height, sizeof(unsigned short), 1, file);
//Save Height variable to TGAFile structure
fread(&tgaFile->bpp, sizeof(unsigned char), 1, file);
//Save BPP(Bits Per Pixel) variable to TGAFile structure

fread(&ucNull,sizeof(unsigned char),1,file);   //Discard descriptor variable

int colorMode = tgaFile->bpp / 8;              //Convert Bits Per Pixel, to Bytes Per Pixel (Simply divide by 8)
```

```
53:
54:     if (colorMode == 3)                              //If there isn't an
                                                         Alpha Channel..
55:        tgaFile->format = GL_RGB;                     // - Set to RGB Only..
56:     else                                             // - Otherwise..
57:        tgaFile->format = GL_RGBA;                    // - Set RGBA
58:
59:     long imageSize = tgaFile->width * tgaFile->height * colorMode;
        //Work out total size of file
60:
61:     tgaFile->imageData = new unsigned char[imageSize];
        //Allocate enough memory for all of the image data
62:
63:     if (!tgaFile->imageData)      //Check if Allocation is successful
64:     {
65:        MessageBox(NULL, L"Out of Memory", L"Error", MB_OK);
66:        delete[] tgaFile->imageData;
67:        fclose(file);
68:        return false;
69:     }
70:
71:     fread(tgaFile->imageData,sizeof(unsigned char),imageSize,file);
        //Save image data to TGAFile structure
72:
73:     if (!tgaFile->imageData)      //Check if the image data was loaded
                                      successfully
74:     {
75:        MessageBox(NULL, L"Error reading TGA", L"Error", MB_OK);
76:        fclose(file);
77:        return false;
78:     }
79:
80:     for (int i = 0; i < imageSize; i += colorMode)
     //Convert Image Data from BGR -> RGB (Swap last and first values around)
81:     {
82:        tmpRGB = tgaFile->imageData[i];
83:        tgaFile->imageData[i] = tgaFile->imageData[i+2];
84:        tgaFile->imageData[i+2] = tmpRGB;
85:     }
86:
87:     fclose(file);              //Close File..
88:
89:     return true;               //Function Completed Successfully .. Return..
90:  }
```

Analysis: Similar to the LoadBitmap Function, we take a filename, and a pointer to our TGA File Structure as the parameters. Lines 3 – 26 simply calculates the fullpath of the File, and then opens it, just as we do in the LoadBitmap Function. In the declarations section we created our own structure for a TGA File, however it does not contain all the elements of TGA File. Only the relevant information is stored and the rest is discarded. We do this by creating two "Recycle Bins". The elements of a TGA File are only of type: 'unsigned char' or 'unsigned short'. The two Recycle Bins are created on lines 7 & 8, one for the 'unsigned char' elements and one for the 'unsigned short' elements. Below is a list of all the elements of a TGA File, and whether we are going to keep it or discard it.

Element	**Save or Discard ?**
`unsigned char idLength;`	Discard

`unsigned char colorMapType;`	Discard
`unsigned char imageType;`	Save
`short int colorMapOrigin;`	Discard
`short int colorMapLength;`	Discard
`unsigned char colorMapEntrySize;`	Discard
`short int originX;`	Discard
`short int originY;`	Discard
`short int width;`	Save
`short int height;`	Save
`unsigned char bits;`	Save
`unsigned char descriptor;`	Discard

After we open the file, we can start discarding/saving the information from it. The First two pieces of information are the idLength variable and the colorMapType variable. We simply read this into our ucNull (`unsigned char` recycle bin). The next element: imageType is important. It specifies the type of the image. Below is a list of all the possible values it can be.

imageType	**Description**
2	Uncompressed RGB Image
3	Uncompressed Black and White Image
10	Run-length Encoded RGB Image
11	Compressed Black and White Image

We are only dealing with Uncompressed TGA Files. Thus, imageType can only either be 2 or 3. On lines 33 – 38, we make sure the TGA file is an uncompressed TGA File, or otherwise we exit immediately. The next 3 elements: colorMapOrigin, colorMapLength, and colorMapEntrySize, are always 0. We simply discard these values on lines 40- 42. The originX, and originY values specify the lower-left co-ordinates of the image. This is always (0,0), so it is no use to us either, and is discard on lines 43, and 44. The Width, Height and BPP (Bits Per Pixel), are all information we require, and are loaded into our TGA File structure on lines 46 – 48. The last element: descriptor variable, specifies whether the image is a 24 bit (0x00) or 32 bit (0x08) image. This is also not needed and discarded on line 50. We have collected all the information we require now, and loaded it into our TGA File structure.

Line 52: Since we are dealing with images in bytes, it would be more convenient to convert our BPP variable, from Bits Per Pixel to Bytes Per Pixel. This can be done be by simply dividing by 8, as there are 8 bits in a byte.

Lines 54 – 57: This is a simply check to see if there is an alpha channel. If there is an extra alpha layer, the format variable of our TGA File structure is set to GL_RGBA (Red, Green, Blue, Alpha). Otherwise it is set to GL_RGB (Red, Green, Blue).

Line 59: This works out the total image size (in bytes). This is done by multiplying the number of pixels (width * height), by the bytes per Pixel (Bits per pixel / 8).s

Lines 61 - 69: The next step is to allocate enough memory for all of the image data. If Allocation fails, then a error message is displayed and we exit the program.

Lines 71 - 78: Simply loads the image data into our TGA File structure. The image data immediately follows the header part of the file. Again, if this fails, we display an error message and quit.

Lines 80 – 85: The final step is to convert the image data from BGR (Blue, Green, Red) to RGB (Red, Green, Blue). We do this the same was as we did with the bitmaps, by swapping the first and last values around.

Finally, we close the file on line 87, and then the function completes successfully on line 89.

Function: LoadTextures

```
bool LoadTextures()
{
     TGAFile tgaFile;                                 //TGA File

     glGenTextures(NumberOfFonts, texture);           //Generate Unique Texture
                                                      Identifiers

     //Loop through each TGA File we are going to load
     for (int i = 0; i < NumberOfFonts; i++)
     {
        if (!LoadUncompressedTGA(FontFiles[i], &tgaFile)) //Load The File..
           return false;

        glBindTexture(GL_TEXTURE_2D, texture[i]);     //Select the current
                                                      texture

        //Set Properties
        glTexImage2D(GL_TEXTURE_2D, 0, tgaFile.format, tgaFile.width,
           tgaFile.imageData);

        glTexParameterf(GL_TEXTURE_2D,             //Set Min Filter of Texture
           GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D,             //Set Max Filter of Texture
           GL_TEXTURE_MAG_FILTER, GL_LINEAR);

           delete[] tgaFile.imageData;             //Delete the Image Data
        }

26:         return true;        //Function Completed Successfully .. Return..
27:     }
```

Analysis: First we create a TGA File structure on line 3. We will use our `LoadUncompressedTGA` function later on to fill this structure with the information from the TGA File. Line 5 generates Texture Identifiers, based on the value of `NumberOfFonts`. Since we our loading more than one font, we will need to load more than texture. This is why we create a `for` loop to loop through each font (again based on the value of the variable `NumberOfFonts`). Line 9 Loads our TGA File structure with information from the TGA File. The TGA Filename is stored in an array we created earlier on called `FontFiles`. If Loading Fails, the function quits immediately. Next, we select the current texture and with `glBindTexture` and then we set the properties of the textures just as we did in our LadTextures function we were loading Bitmaps. From Line 18 – 21, we simply set the `GL_TEXTURE_MIN_FILTER` & `GL_TEXTURE_MAG_FILTER` Filters. Both of these are set to `GL_LINEAR`, but this is not particularly important as we reset these filters when drawing our text. Finally we delete the image data in the TGA File structure, ready for the next texture.

File: BitmappedText.h

Declarations

```
class BitmappedText  //The Bitmapped Font Text Class Implementation..
{
   public:
       BitmappedText(char String[]);   //Constructor
       FrameworkKing_DrawText(); //Function to draw character to screen
       int FontName;             //What Font to use ?
       float LocX;               //X Location of character in 3D Space
       float LocY;               //Y Location of character in 3D Space
       float LocZ;               //Z Location of character in 3D Space
       float SizX;               //Scale Factor on X-axis
       float SizY;               //Scale Factor on Y-axis
       float SizZ;               //Scale Factor on Z-axis
       float RValue;             //Red Value of the Color of the font
       float GValue;             //Green Value of the Color of the font
       float BValue;             //Blue Value of the Color of the font
       float AValue;             //Alpha Value of the Color of the Font
       int TextureFilter;        //What texture filter to use to render
                                 the font ?
       char Text[512];           //Text to draw to screen
   private:
       int NumberOfCharacters;   //The Number of Character in the text
       float OneCWUnit;          //The Width of one character
       float OneCHUnit;          //The Height of one character
       float XOffsetOfC;         //Which column to select our character
                                 from
       float YOffsetOfC;         //Which row to select our character
                                 from
};
```

Analysis: This is our BitmappedText Class. It consists of a custom constructor, one public member function, 13 public member variables, and 5 private member variables. Each element of the class will be discussed as we analyse this Header file.

Function: BitmappedText::BitmappedText

```
BitmappedText::BitmappedText(char String[])
{
     //Store Text in member variable..
     memcpy (Text,String,strlen(String)+1);

     //Set NumberOfCharacters to Number Of Characters in Text..
     NumberOfCharacters = strlen(Text);

     OneCWUnit = 1.0f/NUMBER_OF_COLUMNS;  //Set One Character Width Unit
     OneCHUnit = 1.0f/NUMBER_OF_ROWS;     //Set One Character Height Unit
}
```

Analysis: This is the constructor implementation of the BitmappedText Class. It takes one parameter, the text to draw. Referring back to the `DrawMyText` Function, in file main.cpp, you will see that when we created a `BitmappedText` Object, we entered "King Rules!!" as its parameter. The first thing this function does is copies the String passed in to the member variable: Text. Next it sets the member variable: `NumberOfCharacters`, to the length of Text. Finally it defines the values of `OneCWUnit` & `OneCHUnit`.

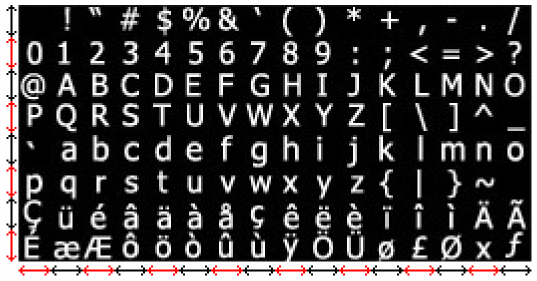


Figure 9.3 The values of `OneCWUnit` & `OneCHUnit`

In Figure 9.3 I have demonstrated for you the value `OneCWUnit` (One Character Width Unit: Across the Bottom) & `OneCHUnit` (One Character Height Unit: Across the Left-Hand Side). These values are calculated by doing 1 divided by the Number of Columns (`NUMBER_OF_COLUMNS`) for `OneCHUnit;` and 1 divided by the Number of Rows (`NUMBER_OF_ROWS`) for `OneCWUnit.`

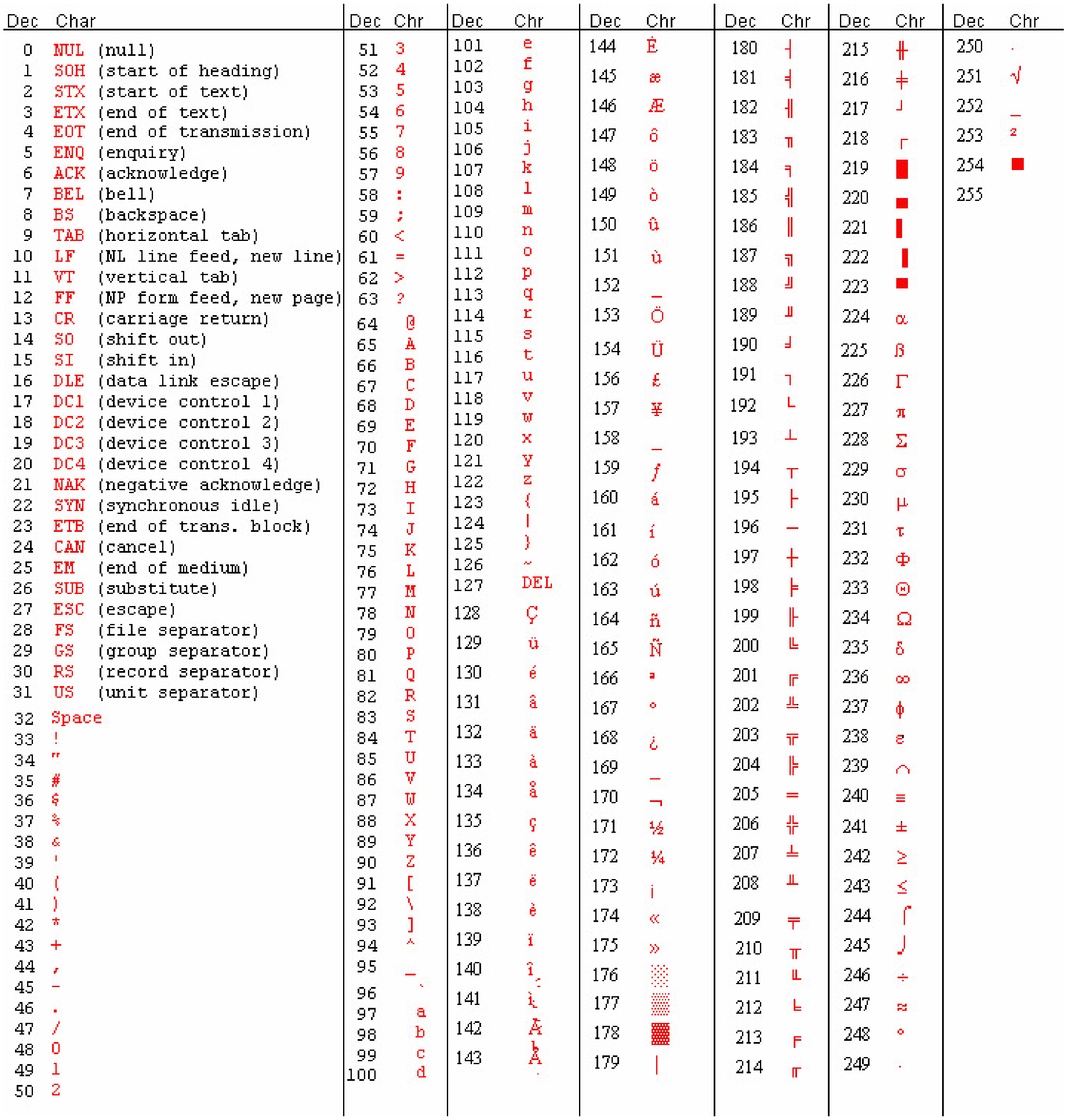
Function: BitmappedText::DrawBitmappedText

```
BitmappedText::DrawBitmappedText()
{
     glTranslatef(LocX, LocY, LocZ);           //Move to specified area..
     glScalef(SizX, SizY, SizZ);               //Scale font to desired
                                               size..

     int i;
     for (i = 0; i < NumberOfCharacters; i ++)  //Loop through each
                                                character of the text..
```

```
{

   switch (Text[i])
   {
     case 32:

        XOffsetOfC = 1.0f;
        YOffsetOfC = 8.0f;
        break;

     case 33:

        XOffsetOfC = 2.0f;
        YOffsetOfC = 8.0f;
        break;

     case 34:

        XOffsetOfC = 3.0f;
        YOffsetOfC = 8.0f;
        break;

     case 35:

        XOffsetOfC = 4.0f;
        YOffsetOfC = 8.0f;
        break;

…
     case 157:

        XOffsetOfC = 14.0f;
        YOffsetOfC = 1.0f;
        break;

     case 158:

        XOffsetOfC = 15.0f;
        YOffsetOfC = 1.0f;
        break;

     case 159:

        XOffsetOfC = 16.0f;
        YOffsetOfC = 1.0f;
        break;

     default:
        XOffsetOfC = 0.0f;
        YOffsetOfC = 0.0f;
        break;
   }

   GLfloat TexCoords[]=     //Array which determines which character to load
   {
     OneCWUnit * (XOffsetOfC - 1), OneCHUnit * (YOffsetOfC - 1), //Bottom-Left of Character
     OneCWUnit *  XOffsetOfC, OneCHUnit * (YOffsetOfC - 1), //Bottom-Right of Character
     OneCWUnit * (XOffsetOfC - 1), OneCHUnit *  YOffsetOfC, //Top-Left of Character
     OneCWUnit *  XOffsetOfC, OneCHUnit *  YOffsetOfC //Top-Right of Character
   };
```

```

        GLfloat Plane[] =       //A simple rectangle. Moves one unit (0.5)
                                to the right for each character
        {
          -0.5f + (0.5 * i), -1.0f,  0.0f,        //Bottom-Left
           0.5f + (0.5 * i), -1.0f,  0.0f,        //Bottom-Right
          -0.5f + (0.5 * i),  1.0f,  0.0f,        //Top-Left
          0.5f + (0.5 * i),  1.0f,  0.0f          //Top-Right
        };

        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);

        glTexCoordPointer(2, GL_FLOAT, 0, TexCoords);  //Point Texture
                                                       Pointer to Tex
                                                       Coords..
        glVertexPointer(3, GL_FLOAT, 0, Plane);        //Point Vertex
                                                       Pointer to the
                                                       plane..

        glBindTexture(GL_TEXTURE_2D, texture[FontName]);

        glTexParameterf(GL_TEXTURE_2D,
          GL_TEXTURE_MIN_FILTER, ((TextureFilter == Use_GL_NEAREST) ?
          GL_NEAREST : GL_LINEAR));
        glTexParameterf(GL_TEXTURE_2D,
          GL_TEXTURE_MAG_FILTER, ((TextureFilter == Use_GL_NEAREST) ?
          GL_NEAREST : GL_LINEAR));

        glColor4f(RValue, GValue, BValue, AValue);//Set Colour from Colour
                                                  Member Variables ..

        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);    //Draw it!
     }

     glTranslatef((-LocX), (-LocY), (-LocZ));     //Move back to original
                                                  location
     glScalef(1/SizX, 1/SizY, 1/SizZ);            //Scale font to desired
                                                  size
}
```

Analysis: The `DrawBitmappedText` draws the text onto the screen. First thing we do is make a call to `glPushMatrix()`, this is so that any changes we make do not affect other objects, for example if you were to draw a cube at the same time, then that would also be affected. Next, we move the text based upon the values in the member variables: `LocX`, `LocY` and `LocZ`. We also scale the font (change the font size) based upon the values in SizX, SizY and SizY. If you look back at the DrawMyText function, in main.cpp, you will see that just after we created a BitmappedText Object (MyText), we set the value of these member variables, just before drawing it onto the screen. Next we loop through each character in the text, determine which character it is, then specify the correct offsets for that character. A switch statement is used on Line 10 to determine which character it is. Every character has an unique ASCII Code. The ASCII Code of any character is an integer, as this is how we identify a character. Figure 9.4 shows the ASCII Code for every character there is. I've taken the table below from www.lookuptables.com and modified it a little.



Dec	Char
0	NUL (null)
1	SOH (start of heading)
2	STX (start of text)
3	ETX (end of text)
4	EOT (end of transmission)
5	ENQ (enquiry)
6	ACK (acknowledge)
7	BEL (bell)
8	BS (backspace)
9	TAB (horizontal tab)
10	LF (NL line feed, new line)
11	VT (vertical tab)
12	FF (NP form feed, new page)
13	CR (carriage return)
14	SO (shift out)
15	SI (shift in)
16	DLE (data link escape)
17	DC1 (device control 1)
18	DC2 (device control 2)
19	DC3 (device control 3)
20	DC4 (device control 4)
21	NAK (negative acknowledge)
22	SYN (synchronous idle)
23	ETB (end of trans. block)
24	CAN (cancel)
25	EM (end of medium)
26	SUB (substitute)
27	ESC (escape)
28	FS (file separator)
29	GS (group separator)
30	RS (record separator)
31	US (unit separator)
32	Space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL
128	Ç
129	ü
130	é
131	â
132	ä
133	à
134	å
135	ç
136	ê
137	ë
138	è
139	ï
140	î
141	ì
142	Ä
143	Å
144	É
145	æ
146	Æ
147	ô
148	ö
149	ò
150	û
151	ù
152	_
153	Ö
154	Ü
156	£
157	¥
158	_
159	ƒ
160	á
161	í
162	ó
163	ú
164	ñ
165	Ñ
166	ª
167	º
168	¿
169	_
170	¬
171	½
172	¼
173	¡
174	«
175	»
176	░
177	▒
178	▓
179	│
180	┤
181	╡
182	╢
183	╖
184	╕
185	╣
186	║
187	╗
188	╝
189	╜
190	╛
191	┐
192	└
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌
219	█
220	▄
221	▌
222	▐
223	▀
224	α
225	ß
226	Γ
227	π
228	Σ
229	σ
230	µ
231	τ
232	Φ
233	Θ
234	Ω
235	δ
236	∞
237	φ
238	ε
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	·
250	·
251	√
252	_
253	²
254	■
255	

Figure 9.4 The ASCII values of every character

So from the table above, the ASCII Code for capital ‘A’ is: 65, and the ASCII Code for the number ‘3’ is: ‘51’. You will notice that some of the characters are of no use to us. For example, characters which have an ASCII code between 0 – 32, and 160 – 255. This is why our switch statement (and our Font TGA File) starts from 32 (Space) upto 159 (ƒ). Once the character has been identified, then the X & Y offsets are set in relation to the Font TGA File. The Offsets are explained below, with the TexCoords.

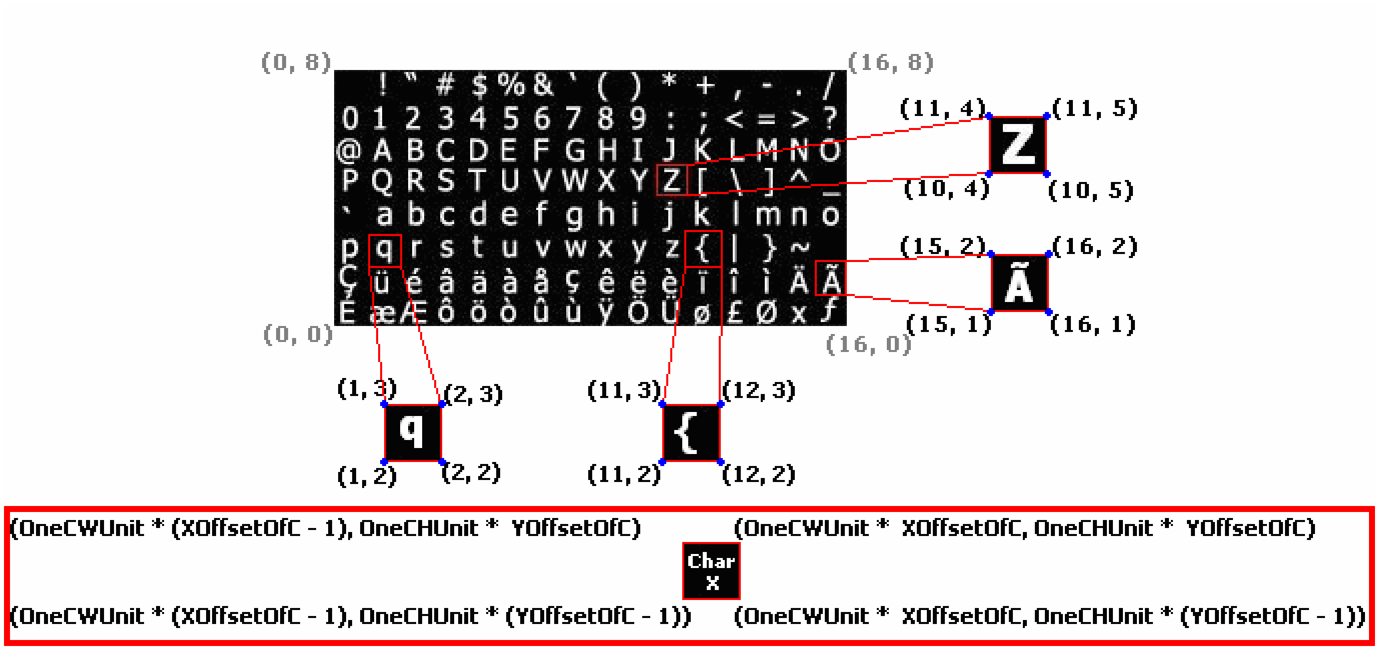

(0, 8)
(16, 8)
(0, 0)
(16, 0)
(11, 4)
(11, 5)
(10, 4)
(10, 5)
(15, 2)
(16, 2)
(15, 1)
(16, 1)
(1, 3)
(2, 3)
(1, 2)
(2, 2)
(11, 3)
(12, 3)
(11, 2)
(12, 2)
(OneCWUnit * (XOffsetOfC - 1), OneCHUnit * YOffsetOfC)
(OneCWUnit * XOffsetOfC, OneCHUnit * YOffsetOfC)
Char x
(OneCWUnit * (XOffsetOfC - 1), OneCHUnit * (YOffsetOfC - 1))
(OneCWUnit * XOffsetOfC, OneCHUnit * (YOffsetOfC - 1))

Figure 9.5 The ASCII values of every character

If you remember back to Tutorial 6 (Texture Filters), we used TexCoords to manipulate the image on the cube faces. Similarly, we do the same here to select just one character from the image above.

Ok, now to discuss the general formula: In this example we are going to select the letter 'Z'. For this character, we would set the XOffset to: 11, and the Y Offset to 5. This would give us the top-right, co-ordinate, and from this we could work out all the other co-ordinates of the character, either by subtracting OneCWUnit (to go to the left one: top-left co-ordinate), OneCHUnit (to go down one: bottom-right co-ordinate), or both (to go down and left: bottom-left co-ordinate).

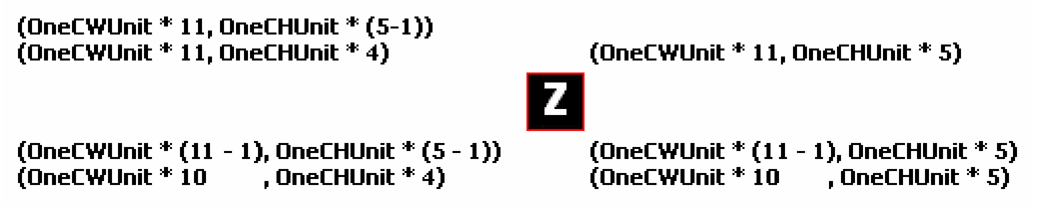

(OneCWUnit * 11, OneCHUnit * (5-1))
(OneCWUnit * 11, OneCHUnit * 4)
(OneCWUnit * 11, OneCHUnit * 5)
Z
(OneCWUnit * (11 - 1), OneCHUnit * (5 - 1))
(OneCWUnit * 10 , OneCHUnit * 4)
(OneCWUnit * (11 - 1), OneCHUnit * 5)
(OneCWUnit * 10 , OneCHUnit * 5)

Figure 9.6 Using the general formula, to calculate the texcoords, for the letter 'Z'

If you ignore all the multiplying by the OneCWUnit/OneCHUnit, you will see that you actually get the same co-ordinates as in the Figure 9.4.

So what lines 6 – 784 do is determine which character it is (base on the ASCII value), then set the Offsets. Lines 786 – 792, is simply an array for the TexCoords, which uses the General formula, and the offsets to select the character from the Font TGA File. Lines 794 – 800, is an array of vertices, which draws a rectangle. The text character will be mapped onto this. Note that, that we move 0.5f every time towards the right for every character. If we didn't do this, all the characters of the text will be on top of each other. The character is drawn on line 817. Once all the characters in the text are drawn, we 'pop' the matrix off the stack, so that the scene is now back to the original starting postion, size etc.

Output

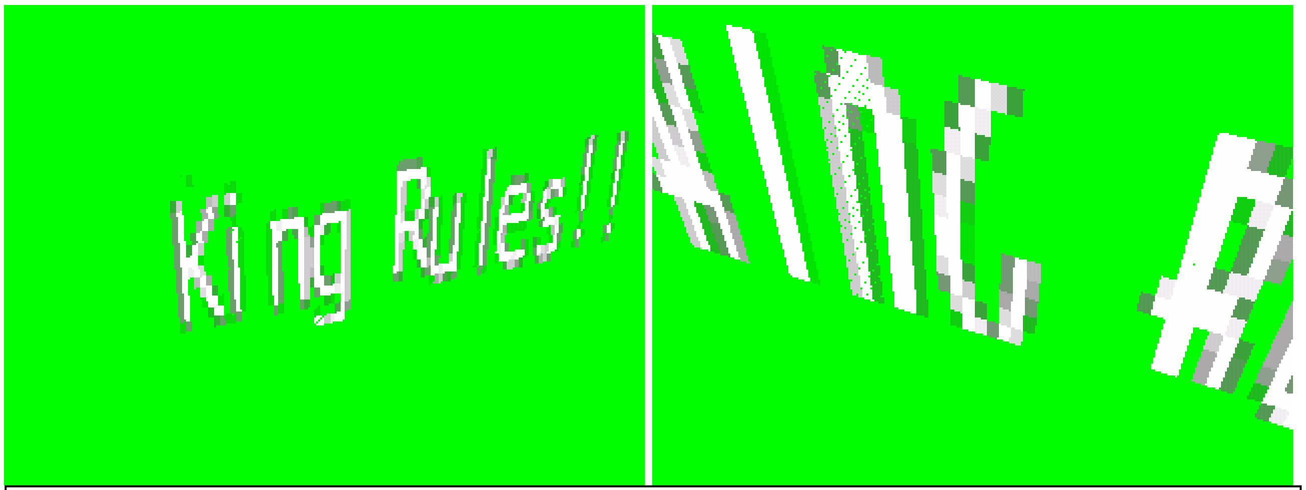


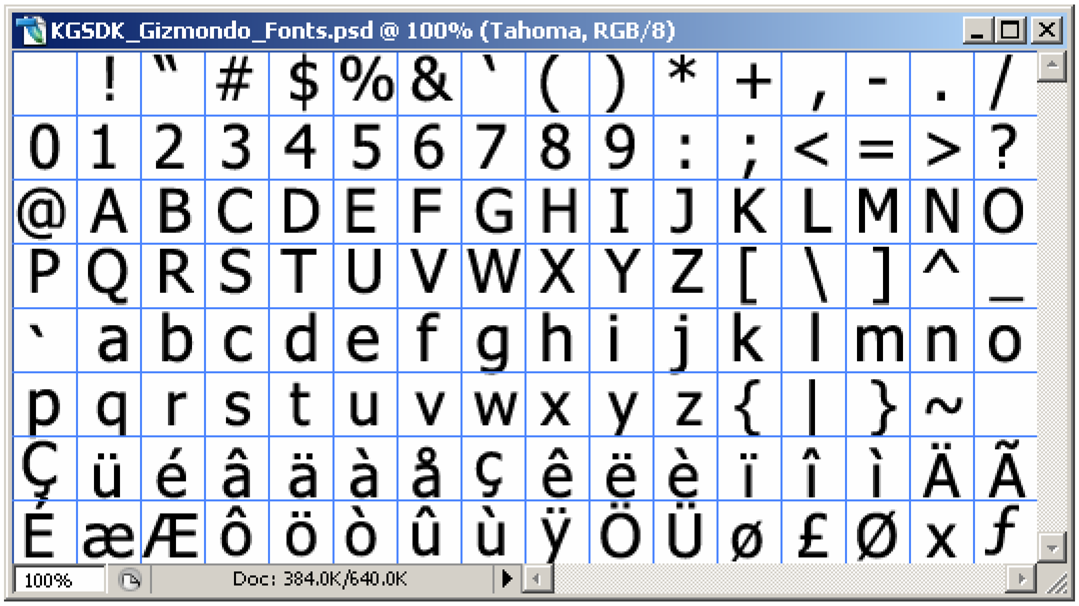
Figure 9.7 Output for Tutorial 9

As you can see, the text is very much like a normal object in 3D Space. It can be rotated, translated, texture filtered, scaled etc.

A Word about making your own Font Files

The font files must be made in a special way. Each character must occupy a cell, the same size as any other character. To do this I would recommend you use Adobe Photoshop. Photoshop has many features you can use to help you create a Font TGA File:

1.) You can use the Grid in Photoshop to draw out the cells which each character must be placed in. This is particularly useful, as the Grid isn't actually part of the image, and can also be adjusted, by going to Edit > Preferences > Guides, Grid & Slices.



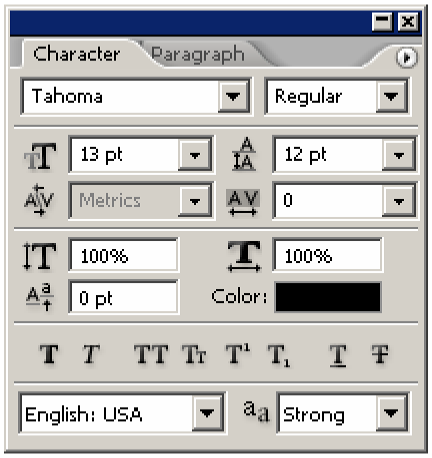
KGSDK_Gizmondo_Fonts.psd @ 100% (Tahoma, RGB/8)

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Ã
É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	x	ƒ

100% Doc: 384.0K/640.0K

Figure 9.8 Grid in Photoshop

2.) You can use the "Character" window to space each character out, so that they fit perfectly in there cells.



Character Paragraph

Tahoma Regular

13 pt 12 pt

Metrics 0

100% 100%

0 pt Color:

English: USA Strong

Figure 9.9 "Character" window in Photoshop

2.) You can add alpha channels to your image, which is essential. Without them, your TGA Files would be just like Bitmaps, with no transparency.

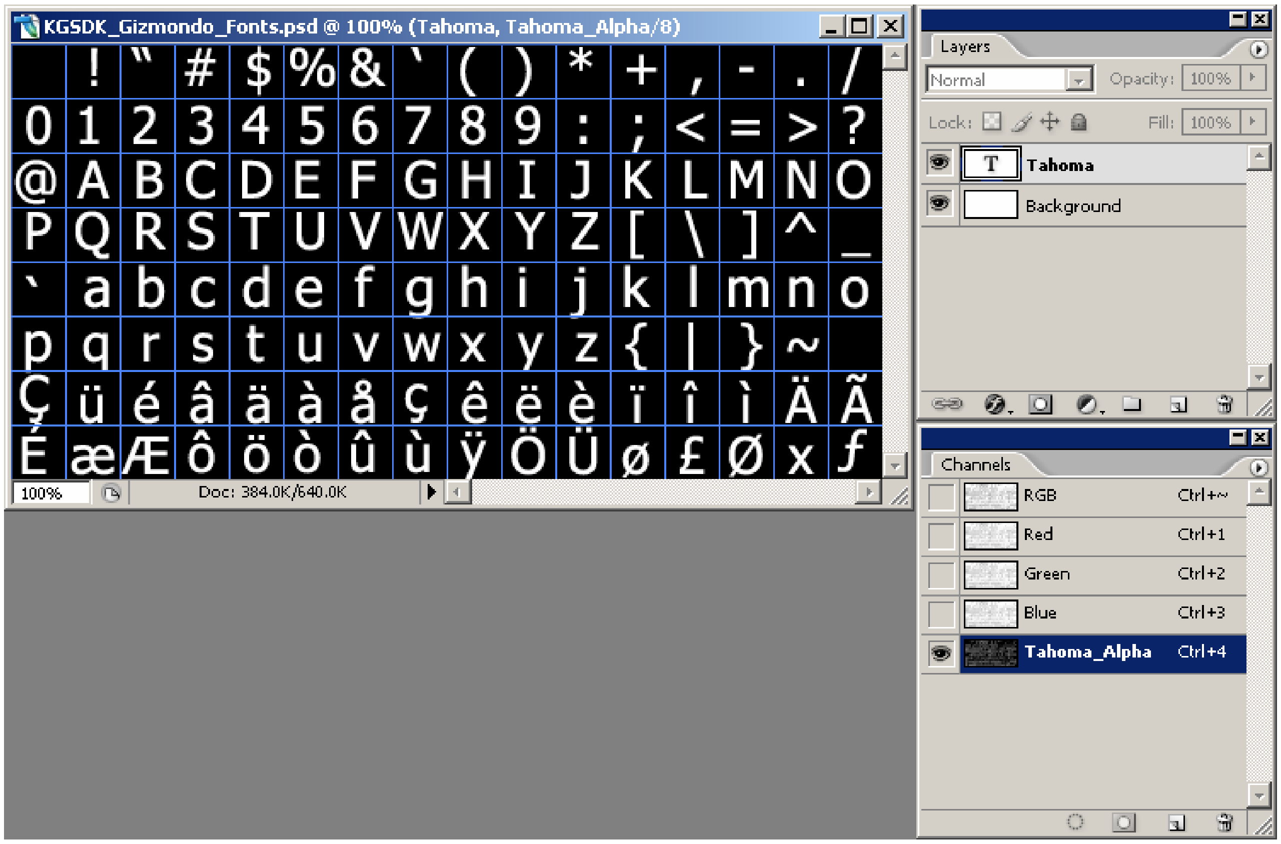

KGSDK_Gizmondo_Fonts.psd @ 100% (Tahoma, Tahoma_Alpha/8)
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [\ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
Ç ü é â ä à å ç ê ë è ï î ì Ä Ã
É æ Æ ô ö ò û ù ÿ Ö Ü ø £ Ø x ƒ
100%
Doc: 384.0K/640.0K
Layers
Normal
Opacity: 100%
Lock:
Fill: 100%
Tahoma
Background
Channels
RGB Ctrl+~
Red Ctrl+1
Green Ctrl+2
Blue Ctrl+3
Tahoma_Alpha Ctrl+4

Figure 9.10 “Character” window in Photoshop

Also don’t forget to save your file as a 24-bit (Uncompressed) TGA file. You can also scale your image up, so that it is easier for you to work in, then scale it down once you’ve finished. In the “Bin” folder, of the KGSDK, you will find two .PSD (Photoshop files) for the “Tahoma” Font, and the “Twentieth Century Poster 1” Font.

- End of OpenGL|ES Tutorials: TGA / Bitmapped Text -