This is the sixth PSP tutorial

Introduction

Hi everybody,

This tutorial is about rendering 2D graphics on a 3D environment. For this and all tutorials I assume that you have some C++ experience and that the basics are clear to you. I also expect that you have used the previous tutorials. This tutorial will show some graphics on screen. It will show a 2D image on top of a 3D environment.

The implementation

In this tutorial we will just use a 3D environment we have previously created like tutorial 2. What we want to do is create a function which lets us draw an image onto the screen. Using the ways of other tutorials(not on this site) it often does work in a 2D environment but in a 3D environment it doesn't so we will just create it for a 3D environment and not 2D.

We will only add something to our GraphicsObject and to the gameApp. lets get on with it :)

GraphicsObject.h

```
int Render2DImageOn3D(float left, float top, const Image* texture);
```

This is the function we can call to render an 2D onto a 3D environment. We pass the left and the top coordinates and the image we want to be drawn on the screen. In this example we only specify the top and left but you can change this code in such a way that you can also specify the bottom and right in order to scale up the image. There are more variations and I will discuss some of them later in this tutorial but for now we make it as simple as it can be.

```
extern "C" {
   #include "graphics.h"
}

typedef struct { float u, v;
         float x, y, z; } vertex2d; // vertex to render
```

Here we include the graphics.h file which holds our way of loading png files. We will use such a png file for a texture in this tutorial. The graphics.h and .cpp file come from the yelarb tutorials. When you will use png textures in the future always include those files together with the framebuffer files. The vertex2d structure is used to pass to the renderer so that it knows what to do.

Now on to the code:

GraphicsObject.cpp

```
int GraphicsObject::Render2DImageOn3D(float left, float top, const Image* texture){
```

The same as above.

```
vertex2d* DisplayVertices = (vertex2d*) sceGuGetMemory(4 * sizeof(vertex2d));
```

Here we reserve 4 times the space used by a vertex2d struct. To display a picture on screen we need 4 vertices in our case. It can be done with less but for now we will use this way.

```
// we do not need to test for depth
sceGuDisable(GU_DEPTH_TEST);
```

We disable the depth test. We do not need to test it. Just use this function in order so that everything is rendered the way you want it to be.

```
// setting the texture
sceGuTexFunc(GU_TFX_REPLACE, GU_TCC_RGBA);
sceGuTexMode(GU_PSM_8888, 0 ,0 ,0);
sceGuTexImage(0, texture->textureWidth, texture->textureHeight, texture->textureWidth, (void*)
texture->data);
```

Here we set the texture. We have passed an Image pointer to this function which we can use to set the texture. The first function tells that we replace the color and that we use RGBA color. The second line tells us what texture format we use. Just use it this way for now, in a future tutorial about texturing I will explain these function in more detail. The last function sets the texture. The first parameter tells which mipmap level the texture has. Since we do not use mipmapping at this point we set it on 0. The second and third parameters tell the width and the height. The fourth parameter specifies the texture bufferwidth and the last passes the actual texture data. So at this point the texture is set.

```
// setting the 4 vertices
DisplayVertices[0].u = 0.0f;
DisplayVertices[0].v = 0.0f;
DisplayVertices[0].x = left;
DisplayVertices[0].y = top;
DisplayVertices[0].z = 0.0f;

DisplayVertices[1].u = texture->textureWidth-1;
DisplayVertices[1].v = 0.0f;
DisplayVertices[1].x = left + texture->textureWidth;
DisplayVertices[1].y = top;
DisplayVertices[1].z = 0.0f;

DisplayVertices[2].u = 0.0f;
DisplayVertices[2].v = texture->textureHeight-1;
DisplayVertices[2].x = left;
DisplayVertices[2].y = top + texture->textureHeight;
DisplayVertices[2].z = 0.0f;

DisplayVertices[3].u = texture->textureWidth-1;
DisplayVertices[3].v = texture->textureHeight-1;
DisplayVertices[3].x = left + texture->textureWidth;
DisplayVertices[3].y = top + texture->textureHeight;
DisplayVertices[3].z = 0.0f;
```

Here is where we create the vertices. As you can see both the width and heigth are used to create the UV's and the coordinates. This way we do not need to pass more info to this function. Please note that you can make the coordinates and UV's anyway you want and please play around with this, just change some values or so and see what happens.

```
// draw the trianglestrip with transform 2D
sceGuDrawArray(GU_TRIANGLE_STRIP, GU_TEXTURE_32BITF | GU_VERTEX_32BITF | GU_TRANSFORM_2D, 4, 0,
DisplayVertices);
```

Well here is the function we have allready seen in a previous tutorial. We now use the GU_TRIANGLE_STRIP instead of the GU_TRIANGLES. The rest of the code is pretty much the same only now we also pass the texture coordinates to the renderer.

```
    // enable the depthtesting again.
    sceGuEnable(GU_DEPTH_TEST);

return 0;
}
```

Well these lines are self explanatory.

Now we have our function to render a 2D image on 3D. The only thing we have to do is call that function and pass the image to it. Like:

```
    // 2d on 3d
    gfx->Render2DImageOn3D(4.0f, 4.0f, this->pic);
```

And now we have made the call :). Also loading the picture needs to be done:

```
    // create the 2d pic
    char sBuffer[250];
    sprintf(sBuffer, "texture.png");
    pic = loadImage(sBuffer);
```

This is everything we need.

MakeFile

The makefile for tutorial 0

```
TARGET = out
OBJS = $(wildcard *.cpp) $(wildcard *.c)

INCDIR =
CFLAGS = -O2 -G0 -Wall -g
CXXFLAGS = $(CFLAGS) -fno-exceptions -fno-rtti -g
ASFLAGS = $(CFLAGS)

LIBDIR =
LDFLAGS =
LIBS = -lc -g -lpng -lz -lpspgum -lpspgu -lstdc++ -lm

PSPSDK=$(shell psp-config --pspsdk-path)
include $(PSPSDK)/lib/build.mak
```

Compile and what happens ?! You see the texture shown on screen, not 3D but 2D, and behind the triangle is rotated around the screen. We have combined 2D and 3D!

Source files

Download the source files.