This is the first texturing tutorial

Introduction

Hi everybody,

This tutorial is about using basic textures. For this and all tutorials I assume that you have some C++ experience and that the basics are clear to you. This tutorial will show a textured triangle rotating around the screen. This tutorial continues on the triangle tutorial.

Adjusting the triangle

With the source of the tutorial 2: rendering a triangle we start out by adjusting the triangle. The triangle in that tutorial only had color but now we will use texturing. First we take a look at the triangle header file:

Triangle.h

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

Here we include the well-known (If you have used the tutorial of yeldarb) graphics files. They are included in the source files. I will use this in this tutorial because for a lot of people they just want some easy way of using textures and images. In the later tutorials of texturing I will explain how to load up other file formats which will handle the whole pipeline of loading up images. This tutorial will use the graphics files.

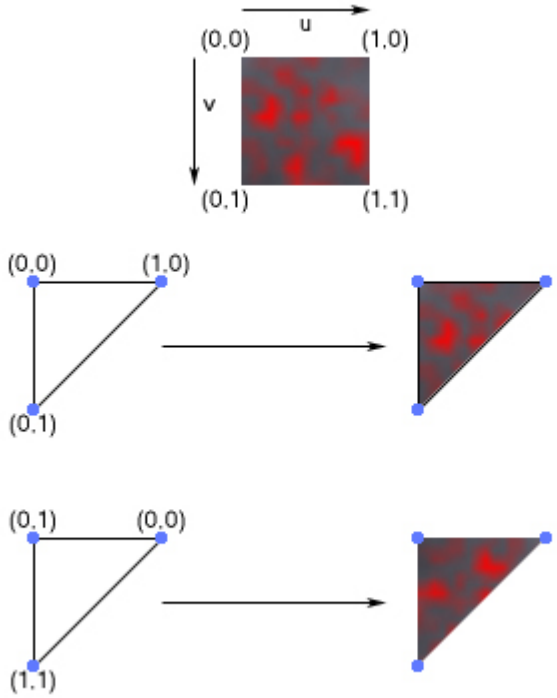
```
typedef struct { float u,v;
                 unsigned int color;
                 float x, y, z; } vertex;
```

Here we have added the u and v floats. These are the coordinates of the texture. In the image on the left you can see how this works. If it still is to complicated please read UV mapping link.

The part above in the picture shows how the U and V coordinates are defined. The above of the texture is a point with coordinate values (0,0) so it is somewhat different than a normal x-y graph. The U and V coordinates are nothing more then the x and y of the texture. As you can see you can stick a texture to a triangle using those coordinates.

The first texture is projected on the triangle just like the coordinates are on a triangle and thus the texture is in the same orientation. The second triangle however is different. As you can see I have altered the UV coordinates and the texture is projected in a rotated state.

The rest of the vertex struct is the same as in the triangle tutorial.



The last thing we have to add to the header is a pointer to our texture.

```
   Image* texture;
```

The Image* is a pointer to an Image struct define in the graphics.h file. It holds the actual pixel data as the width and height of the texture.

Triangle.cpp

```
Triangle::Triangle() {
   triangle = (vertex*)memalign(16,3 * sizeof(vertex));
   triangle[0].u = 0.0f;
   triangle[0].v = 0.0f;
   triangle[0].color = 0xffffffff;
   triangle[0].x = -4.0f;
   triangle[0].y = 2.0f;
   triangle[0].z = 10.0f;

   triangle[1].u = 0.5f;
   triangle[1].v = 1.0f;
   triangle[1].color = 0xffffffff;
   triangle[1].x = -4.0f;
   triangle[1].y = -2.0f;
   triangle[1].z = 10.0f;

   triangle[2].u = 1.0f;
   triangle[2].v = 0.0f;
   triangle[2].color = 0xffffffff;
   triangle[2].x = 4.0f;
   triangle[2].y = 2.0f;
   triangle[2].z = 10.0f;

   rot = 1.5f;

   texture = loadImage("texture.png");
}
```

Here we have our triangle constructor, as you can see I have added the UV coordinates. For this tutorial I have set the rot variable to 1.5f so that the triangle starts becoming visual more early than in the triangle tutorial.

The last line is more interesting. That line calls the loadImage function (graphics file) which loads up the PNG file (passed as a parameter) and returns a pointer to the loaded image. We save that pointer into the texture variable. Now we have loaded up the vertices and the texture.

```
void Triangle::Render() {

   rot += 0.01f;

   sceGumMatrixMode(GU_MODEL);
   sceGumLoadIdentity();
   sceGumRotateY(rot);
```

We rotate our object, we set up the matrix and then the texture stuff begins.

```
   sceGuTexMode(GU_PSM_8888, 0 ,0 ,0);
```

This function sets the texture mode. The first parameter sets the texture format. In this case we use a 8888 color setting. We can also use these: (from the SDK documentation.)

- GU_PSM_5650 - Hicolor, 16-bit
- GU_PSM_5551 - Hicolor, 16-bit
- GU_PSM_4444 - Hicolor, 16-bit
- GU_PSM_8888 - Truecolor, 32-bit
- GU_PSM_T4 - Indexed, 4-bit (2 pixels per byte)
- GU_PSM_T8 - Indexed, 8-bit

The second parameter specifies the number of mipmaps we will use. I will explain what that is in the third texturing tutorial. The third is an unknown parameter just use 0 for this one. The last parameter specifies if the texture needs to be swizzled. This will also be explained in the third tutorial in the texturing series. Now we have set the texture mode.

```
   sceGuTexFunc(GU_TFX_REPLACE, GU_TCC_RGB);
```

This function specifies how the textures are applied. In our case we tell it to replace the current color/texture and use RGB (instead of ARGB). See parameter info for complete explaination of the possible modes.

```
   sceGuTexFilter(GU_LINEAR, GU_LINEAR);
```

This function sets the texturefilter. The first parameter specifies the minimizing filter and the second specifies the maximizing filter. What Exactly does this mean? It means that when you have a 2 by 2 texture and you have a large triangle on which you project it, the PSP needs to calculate the color values between the texels(texture coordinates) because the triangle is larger that only those 4 pixels of the texture. This is also known sometimes as smoothing. The following types can be chosen:

- GU_NEAREST
- GU_LINEAR
- GU_NEAREST_MIPMAP_NEAREST
- GU_LINEAR_MIPMAP_NEAREST
- GU_NEAREST_MIPMAP_LINEAR
- GU_LINEAR_MIPMAP_LINEAR

The last four can only be used with mipmapping. If you have compiled the source try changing these values and see what happens.

```
   sceGuTexScale(1.0f, 1.0f);
```

This line lets us specify what the texturescale is. Try using this with 1.0f and 1.0f because almost everything uses that. For example when

you load an object file it almost always has UV coordinates ranging from 0 to 1.

```
   sceGuTexImage(0, texture->textureWidth, texture->textureHeight, texture->textureWidth,
(void*)texture->data);
```

Here we actually use the texture to be used on the triangle(s). The first parameter is the mipmaplevel. Set this to 0 for now. I will explain this in the third tutorial in the texturing series. The second and the third are the texture width and height. The fourth is the bufferwidth and the last is a pointer to the actual texture data.

```
   sceGumDrawArray(GU_TRIANGLES, GU_TEXTURE_32BITF | GU_COLOR_8888 | GU_VERTEX_32BITF |
GU_TRANSFORM_3D, 3, 0, triangle);

}
```

The drawarray function has been extended with the GU_TEXTURE_32BITF value. This tells the PSP that we will use UV coordinates in a 32bit float format. In our case we use 0.0f to 1.0f.

Now we are done with our triangle. Lets make some changes to the graphics initiation function otherwise the code will not work.

Graphics object

The only thing we need to do to get it to work is to tell the PSP that we will be using textures instead of colors.

```
   sceGuEnable(GU_TEXTURE_2D);
```

I like to put this line in the init function of the graphics object because I almost always use textures and no colors. The only time I use colors is with lines and 2D triangles (for screenfading for example) and to do that I turn off the texturing in those functions.

We now have a working texture code. Yo ucan texture almost every primitive so go on and mess around some.

MakeFile

The makefile for tutorial 4a

```
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 -lpspgum -lpspgu -lpng -lz -lstdc++ -lm -lpsppower

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

Compile and what happens ?! The triangle has the texture projected on it. (Do not forget to include the texture.png to the EBOOT.PBP)

Source files

Download the source files.