This is the third PSP tutorial

Introduction

Hi everybody,

This tutorial is about rendering a triangle. For this and all tutorials I assume that you have some C++ experience and that the basics are clear to you. This tutorial will finally show some graphics on screen. It will show a colorized triangle on screen.

The implementation

Well lets get right to it. If we want to render a triangle we need.... a triangle! so lets take a look at the triangle class we will create in order to use it.

Triangle.h

```
#ifndef TRIANGLE_H_
#define TRIANGLE_H_

#include <psptypes.h>

typedef struct { unsigned int color;
      float x, y, z; } vertex; // vertex to render

class Triangle {

protected:
   vertex* triangle;
   float rot;
   ScePspFMatrix4 world;

public:
   Triangle();
   ~Triangle();
   void Render();
};
#endif
```

Once again we start with our headerguard. The struct is used to store vertex data which we can pass to the render functions. We will look into this when we arrive at the .cpp code of this class. I have created a float called rot which we will use to rotate the triangle during the execution of the game. The world variable is a matrix. If you do not know what a matrix is, google it :) We need this matrix to create a transform for the triangle. The render function will be used to render the triangle. This class design is pretty much self explainatory.

Lets take a look at the .cpp file which does all the work.

Triangle.cpp

```
#include "Triangle.h"
#include <pspgu.h>
#include <pspgum.h>
#include "malloc.h"
```

We need these headers, the malloc is used for the memalign function below:

```
Triangle::Triangle() {
   triangle = (vertex*)memalign(16,3 * sizeof(vertex));
   triangle[0].color = 0xff00ff00;
   triangle[0].x = -4.0f;
   triangle[0].y = 2.0f;
   triangle[0].z = 10.0f;
   triangle[1].color = 0xff00ff00;
   triangle[1].x = 0.0f;
   triangle[1].y = -2.0f;
   triangle[1].z = 10.0f;
   triangle[2].color = 0xff00ff00;
   triangle[2].x = 4.0f;
   triangle[2].y = 2.0f;
   triangle[2].z = 10.0f;
   rot = 0.0f;
}

Triangle::~Triangle() {
   delete(triangle);
}
```

The first line reserves 3 times the memory needed for 1 vertex. We need three because a triangle is create with three vertices. So after that we create the three vertices in counter clockwise order. (Look at the coordinates) We also set the rot float to zero so that the rotation of the triangle starts with an angle of 0.0f

```
void Triangle::Render() {
   rot += 0.05f;
   sceGumMatrixMode(GU_MODEL);
   sceGumLoadIdentity();
   sceGumRotateY(rot);

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

This is the render function. We first do some matrix calculations to get the model(in this case a triangle) in a certain orientation on a certain position. The next tutorial will go more into the world of matrices and the sort but for now understand that those three lines position the object in the 3D space somewhere. The final line renders the triangle to the screen. Lets take a closer look to that function.

The sceGumDrawArray function is part of the Gum family. It is used to render a primitive. The first parameter of the function states which primitive is to be rendered. In this case we use GU_TRIANGLES but there are more (from the SDK):

- GU_POINTS - Single pixel points (1 vertex per primitive)
- GU_LINES - Single pixel lines (2 vertices per primitive)
- GU_LINE_STRIP - Single pixel line-strip (2 vertices for the first primitive, 1 for every following)
- GU_TRIANGLES - Filled triangles (3 vertices per primitive)
- GU_TRIANGLE_STRIP - Filled triangles-strip (3 vertices for the first primitive, 1 for every following)
- GU_TRIANGLE_FAN - Filled triangle-fan (3 vertices for the first primitive, 1 for every following)
- GU_SPRITES - Filled blocks (2 vertices per primitive)

When we have specified the primitive we have to specify in the second parameter what kind of vertices we pass into the function. We have been using GU_COLOR_8888 | GU_VERTEX_32BITF | GU_TRANSFORM_3D which means that we use color and position and use the primitive in 3D space. Here is a list of possible flags (from the SDK):

- GU_TEXTURE_8BIT - 8-bit texture coordinates
- GU_TEXTURE_16BIT - 16-bit texture coordinates
- GU_TEXTURE_32BITF - 32-bit texture coordinates (float)

- GU_COLOR_5650 - 16-bit color (R5G6B5A0)
- GU_COLOR_5551 - 16-bit color (R5G5B5A1)
- GU_COLOR_4444 - 16-bit color (R4G4B4A4)
- GU_COLOR_8888 - 32-bit color (R8G8B8A8)

- GU_NORMAL_8BIT - 8-bit normals
- GU_NORMAL_16BIT - 16-bit normals
- GU_NORMAL_32BITF - 32-bit normals (float)

- GU_VERTEX_8BIT - 8-bit vertex position
- GU_VERTEX_16BIT - 16-bit vertex position
- GU_VERTEX_32BITF - 32-bit vertex position (float)

- GU_WEIGHT_8BIT - 8-bit weights
- GU_WEIGHT_16BIT - 16-bit weights
- GU_WEIGHT_32BITF - 32-bit weights (float)

- GU_INDEX_8BIT - 8-bit vertex index
- GU_INDEX_16BIT - 16-bit vertex index

- GU_WEIGHTS(n) - Number of weights (1-8)
  <
- GU_VERTICES(n) - Number of vertices (1-8)

- GU_TRANSFORM_2D - Coordinate is passed directly to the rasterizer
- GU_TRANSFORM_3D - Coordinate is transformed before passed to rasterizer

Don't be alarmed by the many possible flags. At this point we only need the color and the vertex flags. In other cases some need to use the normals in there game or textures but at this moment we do not. We will use them in future tutorials so if you do not understand it completely is does not matter. One thing to understand though is that when you specify that you will use for example GU_COLOR_4444 that you also store the color data in that format and not any other format. The same goes for 8BITF vertex data. Do not use float in that case.

The third parameter is the number of vertices passed to the function. If you create a vertex array of lets say 8 triangle please understand that the number needs to be 3*8 because every triangle has 3 vertices. The fourth is an optional pointer to an index-list which can enhance the speed but we will not cover that now.
The last parameter is the actual array of vertices that needs to be passed. In our example it is the triangle array.

Now we have created our Triangle class we only have to create an instance of it and call the render function in the renderfunction of the gameApp

GameApp

GameApp.h

```
Triangle* triangle;
ScePspFMatrix4 projection;
ScePspFMatrix4 view;
```

These lines need to be placed in the protected part of the class. The first is a pointer to our newly made Triangle class. The last two are matrices which are used for the view and the projection. The next tutorial will explain more about those matrices.

GameApp.cpp

```
// setting the projection
sceGumMatrixMode(GU_PROJECTION);
sceGumLoadIdentity();
sceGumPerspective(45.0f,16.0f/9.0f,2.0f,1000.0f);

// create the triangle
triangle = new Triangle();
```

This is the creation of the triangle, this code is to be placed in the Load function of the GameApp class. We also create a projection. This will be explained in the next tutorial. It is basicly "How we see the world". The last thing to do is put the next lines of code into the render function of the GameApp class:

```
// setting the view
sceGumMatrixMode(GU_VIEW);
sceGumLoadIdentity();

// render model
triangle->Render();
```

And now the code is done !! As you can see we also had to create a view. The next tutorial will explain more about this but for we only need a default view.

Graphics Object

I have also made some changes in the graphics object. In order to see the triangle without texture we have to delete the line:

```
sceGuEnable(GU_TEXTURE_2D);
```

This way we tell the PSP to render stuff without expecting a texture.

MakeFile

The makefile for tutorial 2

```
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 -lstdc++ -lm -lpsppower

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

Compile the code and you see a large Triangle rotating around the view on the screen. Please note the libs line in the makefile. The lpspgum needs to be in front of the lpspgu lib otherwise there will be linking errors.

Source files

Download the source files.