This is the first PSP tutorial

Introduction

Hi everybody,

This tutorial is about setting the environment for 3D rendering. For this and all tutorials I assume that you have some C++ experience and that the basics are clear to you. This tutorial does not show anything great on screen and is not meant for that. This tutorial creates the basic building blocks on which all other tutorials are based.

The implementation

As with C and C++ programs is expected we first begin with the main.cpp file. This file holds the main function and the usual callback function. Lets start of with the code:

main.cpp

```
#include <psptypes.h>
#include <pspmoduleinfo.h>
#include <pspthreadman.h>
#include <pspkernel.h>
#include "GraphicsObject.h"
#include <pspgu.h>
```

These are the include files. Nothing much to say about them, we need the pspgu library for the Graphical Unit which we will use to set up the 3D environment.

```
PSP_MODULE_INFO("Tutorial0", 0x0000, 1, 1);
```

Well this line gives some info about the game. The first attribute is the name of the game, the others are for versions but we will only use the naming. This is not neccesary but I always use it with these default values. I will mention this line though in tutorial m0 where we want to convert 1.5fw code to 3.xxfw code.

```
/*!
* \fn int exit_callback(int arg1, int arg2, void *common)
* \brief Exits the game.
*
* @return 0
*/
int exit_callback(int arg1, int arg2, void *common) {
   sceKernelExitGame();
   return 0;
}

/*!
* \fn int CallbackThread(SceSize args, void *argp)
```

```
* \brief callback thread.
*
* @return 0
*/
int CallbackThread(SceSize args, void *argp) {
   int cbid;
   cbid = sceKernelCreateCallback("Exit Callback", exit_callback, NULL);
   sceKernelRegisterExitCallback(cbid);
   sceKernelSleepThreadCB();
   return 0;
}

/*!
* \fn int SetupCallbacks(void)
* \brief Creates a callback thread and returns the thread ID.
*
* @return thid
*/
int SetupCallbacks(void) {
   int thid = 0;
   thid = sceKernelCreateThread("update_thread", CallbackThread, 0x11, 0xFA0, 0, 0);
   if(thid >= 0) {
      sceKernelStartThread(thid, 0, 0);
   }
   return thid;
}
```

These three functions are used to created a callback function which enables that when you press the HOME button on Sony's PSP you can exit to the XMB. Well you have probebly seen these functions on other PSP tutorials also so I won't elaborate to much about it. Just understand that this can be very handy for the user that he or she can exit the game at any moment. Just use this in every game you will make.

```
/*!
* \fn int Main(void)
* \brief The main function
*
* @return 0
*/
int main (void) {
   // setup the callbacks
   SetupCallbacks();
   // setup the environment
   GraphicsObject* gfx = GraphicsObject::Instance();
   // init the 3D environment
   gfx->Init3DGraphics();
   // shutdown
   sceGuTerm();
   sceKernelExitGame();
   return 0;
}
```

This is the main function. First we setup the callback function. After this we create a pointer to an GraphicsObject. This is the object that handles a lot of the graphics functions in the future. This tutorial we will only setup an environment. This object will be explained in more detail later on in the next chapter of this tutorial. As you can see we use instead of the new function the Instance function of the object. This is part of an design pattern called "Singleton". I will explain a bit about this in the next chapter but not in great detail so if you want to learn more, use Google :D.

When the object is created we will call the Init3DGraphics function. This is the initialization function of the 3D environment. After this we terminate the GU and the game.

The GraphicsObject will be a little blurry to most at this moment but fear not, it will be explained in the next chapter.

GraphicsObject

I will explain the Graphics object in this chapter. The GraphicsObject is the class that handles a lot of graphics functionality. Well it will in these tutorials anyway. Here is the implementation of the object and as usual with comments.

GraphicsObject.h

```
#ifndef GRAPHICSOBJECT_H_
#define GRAPHICSOBJECT_H_

class GraphicsObject {

private:
   static GraphicsObject* _instance;
protected:
   GraphicsObject(void);
   GraphicsObject(const GraphicsObject&);
   GraphicsObject& operator= (const GraphicsObject&);
public:
   static GraphicsObject* Instance(void);
   ~GraphicsObject(void);
   bool Init3DGraphics(void);
};

#endif
```

This is the GraphicsObject class at this moment. In the future we will extend this greatly but for now this will suffice. The #ifndef, #define and #endif are used to avoid including this file multiple times (). This is handy because in some header files you can have defined a struct and if the file is included multiple times then the struct will be defined multiple times resulting in an error. Use this extensively in C++ (see also: idempotency).

As some of you can see the class is a singleton. This means that there can only exist one instance of this object. This is very usefull since we only need one graphics object in our game. Using this design pattern enables us to use the graphics object in all other classes, function etc. without passing the pointer to the object everytime.
As we go further in the tutorials we will see why this is handy exactly. Understanding it is not neccesary in this tutorial but is adviced. (see also: Design Pattern Singleton)

Init3DGraphics is the function we call after we have created the object. Lets take a look at the source file:

GraphicsObject.cpp

```
#include "GraphicsObject.h"
#include "pspgu.h"

static unsigned int __attribute__((aligned(16))) list[262144];

GraphicsObject* GraphicsObject::_instance = 0;// initialize pointer
```

```
GraphicsObject* GraphicsObject::Instance (void) {
   if (_instance == 0){
      _instance = new GraphicsObject; // create sole instance
   }
   return _instance; // address of sole instance
}

GraphicsObject::GraphicsObject(void) {
}
GraphicsObject::~GraphicsObject(void) {
}
```

The list variable is used to store the information of the GU. The other functions are used for the Singleton implementation. The constructor and the destructor are empty at the moment but this will change in future tutorials.

```
bool GraphicsObject::Init3DGraphics(void) {
   // Initialize the GraphicalSystem
   sceGuInit();
   sceGuStart(GU_DIRECT,list);
   sceGuDrawBuffer(GU_PSM_8888,(void*)0,512);
   sceGuDispBuffer(480,272,(void*)0x88000,512);
   sceGuDepthBuffer((void*)0x110000,512);
   sceGuOffset(2048 - (480/2),2048 - (272/2));
   // create a viewport centered at 2048,2048 width 480 and height 272
   sceGuViewport(2048,2048,480,272);
   sceGuDepthRange(0xc350,0x2710);
   sceGuScissor(0,0,480,272);
   sceGuEnable(GU_SCISSOR_TEST);
   sceGuDepthFunc(GU_GEQUAL);
   sceGuEnable(GU_DEPTH_TEST);
   sceGuFrontFace(GU_CW);
   sceGuShadeModel(GU_SMOOTH);
   sceGuEnable(GU_CULL_FACE);
   sceGuEnable(GU_TEXTURE_2D);
   sceGuEnable(GU_CLIP_PLANES);
   sceGuEnable(GU_LIGHTING);
   sceGuEnable(GU_BLEND);
   sceGuBlendFunc(GU_ADD, GU_SRC_ALPHA, GU_ONE_MINUS_SRC_ALPHA, 0, 0);
   sceGuFinish();
   // wait untill the list has finished.
   sceGuSync(0,0);
   // turn on the display
   sceGuDisplay(GU_TRUE);
   return true;
}
```

This is the function which is really this whole tutorial but we needed all the other code to be able to launch this code in a proper manner.

We first start by initializing the GU, then we tell it to use the list. GU_DIRECT means that the GU performes rendering as the list is filled. You can also use other contexts but we will use this one. We setup the screen with the correct dimensions and the viewport. The Scissor is used to cut thing away outside the screen dimension. After this we enable a lot of functions.

We enable depthtest, this means that object are store with depth. Object fully behind another object will not be rendered. Objects partially behind another object will only be rendered partially. The ShadeModel is set to smooth, this means that Primtives are gouraud-shaded which means that all vertex-colors take effect. You can also set it to GU_FLAT which means that the primitives are flat-shaded. Well to understand all the other functions at this point just read the PSP documentation, at this point in the tutorial we do not need to understand

them and they will be explained in more detail in future tutorials. The last functions will finish the setting up and turn on the display.

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 = -lstdc++ -lc -lpspgu

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

Compile and what happens ?! It starts but then exits immediatly ?! Yes this is suppose to happen. We only create and initialize an graphics object but immediatly after it is created and initialized the game is terminated. In the next chapter we will see our first output.

Source files

Download the source files.