This is PSP tutorial 17: Skybox

Introduction

Hi everybody,

This tutorial is about rendering a skybox. 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. Use the source of the triangle tutorial as a beginning and remove all triangle references in it.

SkyBoxes

Skyboxes are used a lot in games, nowadays skyboxes have evolved in skyspheres, 3D skyboxes and other ways of creating better looking skies. This tutorial will create a skybox and not the spheres or other new ways of doing them. The reason for this is because you can, with a little imagination, create the skyspheres or other methodes without much difficulty when you know the basics. The nicest way, in my opinion, is creating a 3D skybox. This is a 3D space with object and a skybox sky which in it whole is used as a skybox. When you do this you do not have to use any of the objects in the 3D skybox with collision detection etc. and it looks a lot better than the skybox I will present in this tutorial. So do experiment further after this tutorial, it will benefit your game a lot probebly.

So at this moment we stick with the old way of creating a skybox. Lets start with taking a look at a screenshot of the sample application:

As you can see the whole screen is filled with terrain. It is not as nice as it can be but you have to envision a whole game area covers almost every part and thus the bad quality is not as visible anymore. Lets take a look at the code:

SkyBox.h

```
#ifndef SKYBOX_H
#define SKYBOX_H
```

```
#include "GraphicsObject.h"
#include <psptypes.h>

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

class SkyBox {

protected:

   ScePspFMatrix4 world;

   vertex2d* boxside1;
   vertex2d* boxside2;
   vertex2d* boxside3;
   vertex2d* boxside4;
   Image* texture1;
   Image* texture2;
   Image* texture3;
   Image* texture4;
   void CreateBox();

public:
   SkyBox();
   ~SkyBox();
   void Render(ScePspFVector3 pos);
};
#endif
```

This is the header file of the skybox. We begin with the header guard and the includes. Nothing new here. If you have done more tutorials everything is known to you. I will however explain the images and the vertices. The four images are the four images around you. Our skybox consists of 4 quads around the player. I have not included the floor or the sky because for this tutorial it is not needed and when you have completed this tutorial you are able to add the floor and the top. Back to our code, the vertex2d structures are used to store the quad data. The images are the textures which will be projected on the quads around the player.

We also need a render function and a createbox function. The latter is just to keep the code readable and could also be left out and the code could be placed in the constructor of the skybox but this way it is more clear.

Lets take a look at the actual code which will be run.

SkyBox.cpp

```
// PSP includes
#include <pspgu.h>
#include <pspgum.h>

// Misc includes
#include "malloc.h"

// Own includes
#include "SkyBox.h"
```

We include the gu and gum libraries, malloc to get memory for our vertices and the header of our class.

```
SkyBox::SkyBox() {
```

```
    // loading the images
    texture1 = loadImage("land1.png");
    texture2 = loadImage("land2.png");
    texture3 = loadImage("land3.png");
    texture4 = loadImage("land4.png");

    // create the skybox geometry
    CreateBox();
}
```

In the constructor we load the four textures and call the CreateBox() function, which will create the geometry of our skybox.

```
SkyBox::~SkyBox() {

    // free images
    freeImage(texture1);
    freeImage(texture2);
    freeImage(texture3);
    freeImage(texture4);

    // free vertices
    delete(boxside1);
    delete(boxside2);
    delete(boxside3);
    delete(boxside4);
}
```

We delete everything in the memory when the skybox object is killed(destructor) so that we can use it for other parts of the game.

```
void SkyBox::CreateBox() {

    // side 1
    boxside1 = (vertex2d*)memalign(16,4 * sizeof(vertex2d));
    boxside1[0].u = 0.0f;
    boxside1[0].v = 0.0f;
    boxside1[0].x = -3.5f;
    boxside1[0].y = 3.5f;
    boxside1[0].z = -3.5f;
    boxside1[1].u = 1.0f;
    boxside1[1].v = 0.0f;
    boxside1[1].x = 3.5f;
    boxside1[1].y = 3.5f;
    boxside1[1].z = -3.5f;
    boxside1[2].u = 0.0f;
    boxside1[2].v = 1.0f;
    boxside1[2].x = -3.5f;
    boxside1[2].y = -3.5f;
    boxside1[2].z = -3.5f;
    boxside1[3].u = 1.0f;
    boxside1[3].v = 1.0f;
    boxside1[3].x = 3.5f;
    boxside1[3].y = -3.5f;
    boxside1[3].z = -3.5f;

    // side 2
    boxside2 = (vertex2d*)memalign(16,4 * sizeof(vertex2d));
```

```
	boxside2[0].u = 0.0f;
	boxside2[0].v = 0.0f;
	boxside2[0].x = 3.5f;
	boxside2[0].y = 3.5f;
	boxside2[0].z = -3.5f;
	boxside2[1].u = 1.0f;
	boxside2[1].v = 0.0f;
	boxside2[1].x = 3.5f;
	boxside2[1].y = 3.5f;
	boxside2[1].z = 3.5f;
	boxside2[2].u = 0.0f;
	boxside2[2].v = 1.0f;
	boxside2[2].x = 3.5f;
	boxside2[2].y = -3.5f;
	boxside2[2].z = -3.5f;
	boxside2[3].u = 1.0f;
	boxside2[3].v = 1.0f;
	boxside2[3].x = 3.5f;
	boxside2[3].y = -3.5f;
	boxside2[3].z = 3.5f;

	// side 3
	boxside3 = (vertex2d*)memalign(16,4 * sizeof(vertex2d));
	boxside3[0].u = 0.0f;
	boxside3[0].v = 0.0f;
	boxside3[0].x = 3.5f;
	boxside3[0].y = 3.5f;
	boxside3[0].z = 3.5f;
	boxside3[1].u = 1.0f;
	boxside3[1].v = 0.0f;
	boxside3[1].x = -3.5f;
	boxside3[1].y = 3.5f;
	boxside3[1].z = 3.5f;
	boxside3[2].u = 0.0f;
	boxside3[2].v = 1.0f;
	boxside3[2].x = 3.5f;
	boxside3[2].y = -3.5f;
	boxside3[2].z = 3.5f;
	boxside3[3].u = 1.0f;
	boxside3[3].v = 1.0f;
	boxside3[3].x = -3.5f;
	boxside3[3].y = -3.5f;
	boxside3[3].z = 3.5f;

	// side 4
	boxside4 = (vertex2d*)memalign(16,4 * sizeof(vertex2d));
	boxside4[0].u = 0.0f;
	boxside4[0].v = 0.0f;
	boxside4[0].x = -3.5f;
	boxside4[0].y = 3.5f;
	boxside4[0].z = 3.5f;
	boxside4[1].u = 1.0f;
	boxside4[1].v = 0.0f;
	boxside4[1].x = -3.5f;
	boxside4[1].y = 3.5f;
	boxside4[1].z = -3.5f;
	boxside4[2].u = 0.0f;
```

```
    boxside4[2].v = 1.0f;
    boxside4[2].x = -3.5f;
    boxside4[2].y = -3.5f;
    boxside4[2].z = 3.5f;
    boxside4[3].u = 1.0f;
    boxside4[3].v = 1.0f;
    boxside4[3].x = -3.5f;
    boxside4[3].y = -3.5f;
    boxside4[3].z = -3.5f;
}
```

Wow! this is a lot of code, but it is really simple. It is the creation of the geometry of the skyboxes. If you use model object you can also use those which results in less code but since we do not use that in this tutorial this will do. The box is modellen around the origin with a distance of 3.5. You will probebly think: "But then I can't render anything because our skybox is blocking the view. Fear not! The render code:

```
void SkyBox::Render(ScePspFVector3 pos) {

    sceGuDepthMask(GU_TRUE); // disable writes to the depth buffer

    sceGumMatrixMode(GU_MODEL);
    sceGumLoadIdentity();
    sceGumTranslate(&pos;);

    sceGuTexFunc(GU_TFX_REPLACE, GU_TCC_RGBA);
    sceGuTexMode(GU_PSM_8888, 0 ,0 ,0);
    sceGuTexFilter(GU_LINEAR,GU_LINEAR);
    sceGuTexWrap(GU_CLAMP, GU_CLAMP);

    sceGuTexImage(0, texture1->textureWidth, texture1->textureHeight, texture1->textureWidth,
          (void*)texture1->data);
    sceGumDrawArray(GU_TRIANGLE_STRIP, GU_TEXTURE_32BITF | GU_VERTEX_32BITF |
          GU_TRANSFORM_3D,4,0,boxside1);
    sceGuTexImage(0, texture2->textureWidth, texture2->textureHeight, texture2->textureWidth,
          (void*)texture2->data);
    sceGumDrawArray(GU_TRIANGLE_STRIP, GU_TEXTURE_32BITF | GU_VERTEX_32BITF |
          GU_TRANSFORM_3D,4,0,boxside2);
    sceGuTexImage(0, texture3->textureWidth, texture3->textureHeight, texture3->textureWidth,
          (void*)texture3->data);
    sceGumDrawArray(GU_TRIANGLE_STRIP, GU_TEXTURE_32BITF | GU_VERTEX_32BITF |
          GU_TRANSFORM_3D,4,0,boxside3);
    sceGuTexImage(0, texture4->textureWidth, texture4->textureHeight, texture4->textureWidth,
          (void*)texture4->data);
    sceGumDrawArray(GU_TRIANGLE_STRIP, GU_TEXTURE_32BITF | GU_VERTEX_32BITF |
          GU_TRANSFORM_3D,4,0,boxside4);

    sceGuDepthMask(GU_FALSE); // enable writes to the depth buffer
}
```

This is the render function. A position is passed to the function so that we can always position the SkyBox around the player. We start in the function by masking the depth of our object, the skybox, so that the geometry is not written in the depth buffer so no matter how far another object is rendered it will not be blocked by the skybox since it will not exist in the depth buffer. (see also: Depth buffer)

After the masking we translate the skybox and set the texture parameters. Note that we set the wrap to clamp so that the texture is clamped and not repeated. This fixes the glitch of showing the stiches between the quads.(thanks to *nexis2600*) After it, the four quads are rendered and finally we disable the masking so that other object will be written in the depth buffer.

Now we have our SkyBox class so we can use it. Lets take a look at the other code that needs to be changed.

GameApp

In the GameApp class, if you have not done this already, remove every reference to the triangle. You can let it stay and then you should see the triangle also in resulting application but the source files do not include the triangle to keep the source as simple and clean as possible. Start with adding the following code to the load function of the GameApp class:

GameApp.cpp

```
    // rotation of the camera
    rot = 0.0f;

    // load up the skybox
    sky = new SkyBox();
```

Here we set the rotation variable for our camera to 0.0 and we create a new SkyBox object.

```
void GameApp::Render() {
    // first set our position
    ScePspFVector3 pos = {0.0f, 0.0f, 0.0f};
    rot += 0.01f;

    // render information.
    sceGuStart(GU_DIRECT,list);

    // clear screen
    sceGuClearColor(0xff000000);
    sceGuClearDepth(0);
    sceGuClear(GU_COLOR_BUFFER_BIT|GU_DEPTH_BUFFER_BIT);

    // set the ambient light.
    sceGuAmbient(0xffffffff);

    // setting the view.
    sceGumMatrixMode(GU_VIEW);
    sceGumLoadIdentity();
    sceGumRotateY(rot);
    sceGumTranslate(&pos;);

    // render the skybox.
    sky->Render(pos);

    // ending rendering
    sceGuFinish();
    sceGuSync(0,0);
    sceDisplayWaitVblankStart();
    sceGuSwapBuffers();
}
```

This is the render function of the GameApp class. The first two lines create a position vector and increment the rotation variabel with 0.01. The position variable will be used to position the view and the skybox. If you create a game you can use the players position instead so that the skybox is always around the player. The other lines are already known if you have done the previous tutorial but to summary, we start a render cycle and clear the depth, set the color to black and clear the buffers. We set the ambient color to fully white so that

everything is visible. The view is rotated and translated and the skybox is rendered at the same location as the view. The last lines finish the render cycle and swaps the buffers so that the render is displayed on screen.

GameApp.h

```
#include "SkyBox.h"
```

We must ofcourse include the skybox header, otherwise we can not use it.

```
SkyBox* sky;
```

This variable is a protected variable, it is the pointer to the skybox.

```
float rot;
```

This variable is also protected.

Now save and compile the whole code. Do not forget to include the skytextures. Also try other images (png format) and read the following links to create even better skyboxes. Thats the end of this tutorial.

Links

Useful links to create better skyboxes:

- Skyboxes, a tutorial on how to use skyboxes
- SkyBoxes II, a wiki about the skyboxes

Source files

Download the source files.

Download the Skybox textures. *(include these with the EBOOT.PBP)*

MakeFile

The makefile for tutorial 17

```
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 ?! You are rotating the camera and are looking at every part of the skybox. Just like the screenshot at the beginning of this tutorial.