This is the object loading tutorial.

Introduction

Hi everybody,

this tutorial is about creating a very basic object loader. It is not meant as a universal way of loading up 3D objects but just a tutorial to show you how such a thing can be done. Use this knowledge and try to create code that reads and renders other 3D formats. For this and all tutorials I assume that you have some C++ experience and that you know the basics. This tutorial will show a 3D object rendered. This tutorial uses Tutorial 2: Rendering a triangle as a basis with all the triangle code deleted from it.

3D objects (.obj)

Before we start with the code you will need to know some bits of information about 3D object and specifically .obj files. In this tutorial I will use those .obj to handle 3D objects. The Obj format is in my opinion the simplest format to store 3D objects currently available. I do not say that it is the best or the most efficient but it is very simple. Let's take a look at a small example:

```
v 1 1 1
v 1 1 -1
v 1 -1 1
v 1 -1 -1
v -1 1 1
v -1 1 -1
v -1 -1 1
v -1 -1 -1
f 1 3 4 2
f 5 7 8 6
f 1 5 6 2
f 3 7 8 4
f 1 5 7 3
f 2 6 8 4
```

As some of you may have spotted, it is the object file of a cube. The first part creates vertices, you can tell by the 'v' at the beginning. The numbers following the 'v' are the x, y and z coordinate values. The second part of the file are the faces. It starts with an f (the first letter of face ofcourse.) followed by four numbers. These four numbers specify which vertices are used to make up the face. So if we take a look at the first face (f 1 3 4 2) we need vertices 1(v 1 1 1) 3(1 -1 1) 4(1 -1 -1) and 2(1 1 -1). The order is also important. To render such a face the vertices have to be in that order otherwise the vertices can be facing wrong or be messed up. This example uses 4 vertices to make up a face. Because the PSP does not have a quad primitive, this tutorial will be using a .obj file in which all the faces are triangulated. We can use the triangle primitive supported by the PSP to pass all the triangles in one pass to the hardware.

We ofcourse also want to use texture coordinates to use textures on our 3D object.

```
vt 1.000000 0.000000
vt 0.000000 1.000000
```

The 'vt' specifies it is a texture coordinate. The u and v values are the two numbers following the 'vt'. There are a lot of other letter

combinations in .obj files but we stick only to those we are going to use in this tutorial.

```
vn 0.000000 -1.000000 0.000000
```

The letters 'vn' are used to specify a vertex normal. In this case the normal points along the y-axis in negative direction.

We now have 3 different sets of data for our vertices. We have the coordinates(v), we have the texture coordinates(vt) and we have the vertex normals(vn). With the first example of the cube the faces only specified the vertices it was made from. Now we also need to specify the texturecoordinates and the normals. With .obj files it is done the following manner:

```
f 1/1/1 3/3/2 2/2/3
```

Now we have all three types of information about the vertices that make up a triangle.

This is the format of the .obj files. The result of this tutorial will yield the following image on Sony's PSP. Play around with the numbers and stuff and see how the application behaves.

That is all the theory about .obj files for now; let's start with the code!

GraphicsObject (Datastructures)

We start by creating some datastructures which we will need to store all the data we retrieve from the .obj file.

GraphicsObject.h

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

We need the graphics class again, or your own image loading functions, so we can use textures.

```
typedef struct { float u, v;
         unsigned int color;
         float nx, ny, nz;
         float x, y, z; } vertexfull; // vertex to render
```

This datastructure is used to store a vertex. We store the UV's, color, normals and ofcourse the actual x, y and z values of the vertex. This structure will be passed to the drawfunctions.

```
typedef struct { float u, v; } ScePspFTVector2;
```

This structure is not needed but I always feel the vector is missing in the SDK, I created my own texturecoordinate vector.

```
typedef struct { unsigned int vertices[3];
         unsigned int textc[3];
         unsigned int normals[3]; } FaceObj;
```

This structure is used to store the faces retrieved from the .obj file. It does not represent a face but it is just a structure holding the f 1/1/1 2/2/2 3/3/3 format.

```
typedef struct { unsigned int faceCount;
         vertexfull *Vertices;
         Image* texture; } ObjMeshPart;
```

The last structure in the GraphicsObject is a part of the mesh. Obj files can be made up of multiple objects. We can for example save 2 of those boxes in 1 .obj file. These objects can also have different textures and thus we save the parts separately so we can render them correctly with the correct textures.

Model class

The model class is used to load up a 3D object, store it and render it.

Model.h

```
#ifndef MODEL_H_
#define MODEL_H_

#include "GraphicsObject.h"

class Model {

protected:
   ObjMeshPart* parts;
   int partCount;
```

```
   bool Load(const char* file, const char* mtl);

public:
   Model(const char* file, const char* mtl);
   void Render();

};
#endif
```

We start with the headerguards again. We include the GraphicsObject because we want to use the datastructures described in the previous section. In the Model class itself we create a pointer "parts". This pointer points to an ObjMeshPart structure. Before we load up a .obj we do not know how many parts it will have, when we create a pointer we can create space in memory when we find out how many parts there are in the .obj file. We also want to store the number of parts in an integer. We use this in the render function, but I will explain the render function at the end of this section.

The load function needs the .obj file and a material file. The material file will not be the same as the .mtl files exported when you create .obj files. Again this is because I do not give a universal loader but only the knowledge to build one yourself. This tutorial will only create a very basic loader with a lot of limitations. More information on the material files when I explain the Load function.

The public part of our class has two function; Model and Render, the latter renders the model on the screen and the Model function calls the load function. Let's take a look at the functions:

Model.cpp

```
#include "Model.h"
#include <pspgu.h>
#include <pspgum.h>
#include "stdio.h"
#include "string.h"
#include "malloc.h"

Model::Model(const char* file, const char* mtl) {
   Load(file, mtl);
}
```

We start with the includes. The Constructor of the class calls the load function and passes the file and the mtl char pointers to that function. I like to keep the constructors as empty as possible so I have put the loading code in a separate function.

```
bool Model::Load(const char* file, const char* mtl) {

   char Textures[100][256];       // 2D array to store paths of the textures.
   char ReadBuffer[256];       // buffer use for reading files.
   char sBuffer[256];

   int textureCount = 0;       // variable to keep count of the number of textures in the model.
```

This is the beginning of our load function. Beware because this function is large and can be quite difficult. Read carefully and do not go any further when something is not clear enough, just ask it using the question form below.

The Load function takes two variables which are the paths to the object file and material file. We start by creating some variables. Since we do not know exactly how many textures we will encounter we create a 2D array of 100 texturepaths which can be 256 characters long. The readbuffer char array will be used to hold the lines we read from the files. The sBuffer is a char array used for different reason when we need some char array. The texturecount variable is used to store how many textures are present for the object.

```
   // creating a FILE pointer
   FILE *fp = NULL;
```

```
// check whether the file can be opened.
if ((fp = fopen(mtl, "rb")) == NULL) { return false; }

// loop through the lines.
while(!feof(fp)) {
   // get new line.
   fgets(ReadBuffer, 256, fp);
   // check if a texture is present.
   if (strncmp("map_Kd ", ReadBuffer, 7) == 0 ) {
      sprintf(Textures[textureCount],"%s", strtok((ReadBuffer+7),"#"));
      textureCount++;
   }
}
// close the file.
fclose(fp);
```

This part of the load function reads the material file and retrieves the paths to the textures. This particular way is not the correct way but it works. Here is what a material file could look like(So it is NOT code):

```
newmtl initialShadingGroup
illum 4
Kd 0.00 0.00 0.00
Ka 0.00 0.00 0.00
Tf 1.00 1.00 1.00
map_Kd UVtexturesCrate256.png
Ni 1.00
```

There are a lot of attributes here describing the material. In this tutorial we are only interested in the textures. The textureline always starts with map_Kd in our case. We only read out that line. Anyway back to the code, we create a File pointer and then we open it. We read the file line by line and check if it starts with map_Kd. If it does we have found a texture and save the texturefile and increment our texture counter. Afterwards we close the file.

```
// read the object file
// this is the first read. This reads how many elements there are.
// With that information we can create the data structures.
fpos_t position;
int groupCount = 0; // amount of groups in object file.
int vertexCount = 0; // amount of vertices in object file.
int textureCoordCount = 0; // amount of texture coordinates in object file.
int faceCount = 0; // amount of face in object file.
int normalCount = 0; // amount of normal in object file.
```

At this moment we have the textures from the material file, we now need to know how many vertices,texture coordinates, normals and faces are in the object file. So we first create integers to keep track of the count for each entity in the file. We set them to zero because there are none at the beginning ofcourse.

```
if ((fp = fopen(file, "rb")) == NULL) { return false; }
// we save the position so we can go back to the beginning
fgetpos (fp, &position;);
while(!feof(fp)) {
   // read the line
   fgets(ReadBuffer, 256, fp);
   // check what kind of object is given in the line.
   if (strncmp("g default", ReadBuffer, 9) == 0 ) groupCount++;
   else if (strncmp("v ", ReadBuffer, 2) == 0 ) vertexCount++;
   else if (strncmp("vt ", ReadBuffer, 3) == 0 ) textureCoordCount++;
   else if (strncmp("f ", ReadBuffer, 2) == 0 ) faceCount++;
```

```
    else if (strncmp("vn ", ReadBuffer, 3) == 0 ) normalCount++;
}
```

In the piece of code above we count every entity so that we know how many there are. We open de object file and then we save the position. Why is this? Well, we open the file, we will read through it and then we want to go back to the beginning. We have saved the position at the beginning so we can go back. The last part is where we loop through every line in the file and check if we encounter an entity at that line.

```
// prepare the arrays
ScePspFVector3 Vertices[ vertexCount ];
ScePspFTVector2 TexCoords[ textureCoordCount ];
FaceObj Faces[ faceCount ];
ScePspFVector3 Normals[ normalCount ];
// setting the number of modelgroups and get some memory for it.
parts = (ObjMeshPart*)malloc(groupCount * sizeof(ObjMeshPart));
// save the number of groups/parts
partCount = groupCount;

// go to the first line in the file.
fsetpos(fp, &position;);
```

Now we have our counts we can create array to hold all the data retrieved from the object file. We also create the mesh parts. The last line set the read back to the start position previously saved.

We have prepared all the variables and now we are going to store everything correctly so that we can use our model in our game:

```
// variables
int group = 0;
int v = 0;
int n = 0;
int t = 0;
int face = 0;
int i = 0;
int l = 0;
int Correction = 0;
int j = 0;
int k = 0;
```

we set up all our variables (not a very good order but it will suffice.) we need to create a model in memory. Note the correction variable, it is needed to get the correct vertices, faces etc. for a meshpart.

```
while(!feof(fp)) {
    fgets(ReadBuffer, 256, fp);
```

We start the loop and we loop until we have reached the end of the object file. Inside the loop we start by getting a line from the file.

```
    if (strncmp("v ", ReadBuffer, 2) == 0 ) {
        sscanf((ReadBuffer+2), "%f%f%f",&Vertices;[ v ].x, &Vertices;[ v ].y, &Vertices;[ v ].z);
        v++;
    }
```

In this if statement we check whether the line we read from the file specifies a vertex. If it is we store the vertex data in the Vertices array. (Please keep in mind that since our arrays start with 0 and the vertices in the objectfile start with 1 we have to subtract 1 but more on this when we reach the code with the faces.

```
    else if (strncmp("vn ", ReadBuffer, 3) == 0 ) {
        sscanf((ReadBuffer+3), "%f%f%f",&Normals;[ n ].x, &Normals;[ n ].y, &Normals;[ n ].z);
```

```
            n++;
        }
```

In this if statement we check whether the line we read from the file specifies a vertexnormal. We also increment the n value which is the normal counter.

```
        else if (strncmp("vt ", ReadBuffer, 3) == 0 ) {
            sscanf((ReadBuffer+3), "%f%f",&TexCoords;[ t ].u, &TexCoords;[ t ].v);
            t++;
        }
```

In this if statement we check whether the line we read from the file specifies a texture coordinate. We also increment the t value which is the texture coordinate counter.

```
        else if (strncmp("f ", ReadBuffer, 2) == 0 ) {

            char *pSplitString = NULL;
            i=0;
            pSplitString = strtok((ReadBuffer+2)," \t\n");

            do {
                sscanf((pSplitString), "%d/%d/%d",&Faces;[ face ].vertices[ i ], &Faces;[ face ].textc[ i
], &Faces;[ face ].normals[ i ]);
                Faces[ face ].textc[ i ] -= 1; // 1 down because the obj file objects start at 1 and
arrays start at 0
                Faces[ face ].vertices[ i ] -= 1;
                Faces[ face ].normals[ i ] -= 1;
                pSplitString = strtok(NULL," \t\n");
                i += 1;
            }
            while( pSplitString );

            face++;
        }
```

In this if statement we check whether the line we read from the file specifies a face. We take the string apart to store the vertices, texture coordinates and the normals of the face. We subtract 1 from the values because our arrays start at 0 and the vertices/normals/texturecoordinates start at 1.

```
        else if (strncmp("EndGroup", ReadBuffer, 8) == 0 ) {
            parts[group].Vertices = (vertexfull*)malloc((face-Correction) * 3 * sizeof(vertexfull));
            parts[group].faceCount = (face-Correction);
            l = 0;
```

If we encounter the EndGroup characterstring in the readbuffer then we know that the end of a mesh part has been reached. You are now probably wondering: "Why have I not encounter this string before?" Well I have included it to separate each part nicely and help me with my code. This "EndGroup" string is not native to the .obj format. At the end of the file I have also included a random string. When this random string is read the code does nothing ,when I leave it EndGroup then it will try to create the part twice. There are numbers of different ways to approach this and mine is surely not the best but it will work and lets me create somewhat easily readable code also for semi-beginners. So when we encounter the "EndGroup" we know that we have all the vertices and other info to construct the vertexlist for our meshpart.

```
            for (j=Correction;j             for (k=0;               parts[group].Vertices[l].u =
TexCoords[Faces[j].textc[k]].u;
                parts[group].Vertices[l].v = -(TexCoords[Faces[j].textc[k]].v);
                parts[group].Vertices[l].color = 0xffffffff;
                parts[group].Vertices[l].nx = Normals[Faces[j].normals[k]].x;
```

```
                parts[group].Vertices[l].ny = Normals[Faces[j].normals[k]].y;
                parts[group].Vertices[l].nz = Normals[Faces[j].normals[k]].z;
                parts[group].Vertices[l].x = Vertices[Faces[j].vertices[k]].x;
                parts[group].Vertices[l].y = Vertices[Faces[j].vertices[k]].y;
                parts[group].Vertices[l].z = Vertices[Faces[j].vertices[k]].z;
                l++;
            }
        }
```

The correction value is the number(index) of the last face used in the previous mesh part. So we start our loop at that point to only save the faces for this meshpart. The k for loop will be looped three times because our faces have three vertices. In that loop we store the vertex information into the vertices structure of the meshpart.

```
        // the texture for this part.
        sprintf(sBuffer, "%s",Textures[group]);
        parts[group].texture = loadImage(sBuffer);
        if (parts[group].texture == NULL) { fclose(fp); return false; }
```

The mesh part also has a texture. We retrieve the texture from the texture array constructed at the beginning of this function. If the texture does not exist or can not be loaded we return false because the loading has failed.

```
        // increment the group number.
        Correction = face;
        group++;
      }
   }
   fclose(fp);
   return true;
}
```

This is the end of the load function. We now have loaded the object file into triangles and the material files into textures. Lets take a look at the code to render the triangles. It should be familiar with you if you have done the previous tutorials.

```
void Model::Render() {
   ScePspFVector3 pos = {0.0f, 0.0f, -10.0f};

   // translate the model.
   sceGumMatrixMode(GU_MODEL);
   sceGumLoadIdentity();
   sceGumTranslate(&pos;);

   // setting of the texture environment.
   sceGuTexMode(GU_PSM_8888, 0 ,0 ,0);
   sceGuTexFunc(GU_TFX_REPLACE, GU_TCC_RGB);
   sceGuTexFilter(GU_LINEAR, GU_LINEAR);
   sceGuTexScale(1.0f, 1.0f);
   sceGuTexOffset(0.0f, 0.0f);

   int i;

   // loop through all the meshparts.
   for (i=0;i < partCount;i++) {
      sceGuTexImage(0, parts[i].texture->textureWidth, parts[i].texture->textureHeight,
parts[i].texture->textureWidth, (void*)parts[i].texture->data);
      sceGumDrawArray(GU_TRIANGLES, GU_TEXTURE_32BITF | GU_COLOR_8888 | GU_NORMAL_32BITF |
GU_VERTEX_32BITF | GU_TRANSFORM_3D, parts[i].faceCount*3, 0, parts[i].Vertices);
   }
```

```
}
```

We start the render function with setting the world matrix correct. This should be known from the previous tutorials. The second part is where we setup the texture environment. If you do not understand this ,I have explained it in the texture tutorial. The last part is also pretty well known, setting the texture and then passing the vertices to the drawarray function. The only difference now is that we do that per meshpart.

This is the end of our model class. You can extend it ofcourse in various ways, one for example is to add get and set position functionality or rotation functionality. Scaling can also be a nice feature to do. You can try to alter the functions or rewrite them to load up other object files. The sky is the limit :D

GameApp

Like with the triangle tutorial we have to alter the GameApp a bit to let it render the model.

GameApp.h

```
#include "Model.h"
```

We have to include ofcourse the Model header if we want to use it.

```
Model* model;
```

We create a pointer to a model. That's all for the GameApp header. Lets take a look at the cpp file.

GameApp.cpp

```
// load up the skybox
model = new Model("krat.obj", "krat.mtl");
```

These lines should be in the Load function. Here we create our object from the krat.obj file and the material is the krat.mtl. (krat means crate in my language)

```
// setting the view.
    sceGumMatrixMode(GU_VIEW);
    sceGumLoadIdentity();
    ScePspFVector3 pos = {-2.0f,10.0f,10.0f};
    ScePspFVector3 at = {0.0f,0.0f,-10.0f};
    ScePspFVector3 up = {0.0f,1.0f,0.0f};
    sceGumLookAt(&pos;, &at;, &up;);
```

We set up the view so we look at our model.

```
// render the skybox.
model->Render();
```

So now we render the model. Now we have created code that will load up an obj file and render it to the screen.

MakeFile

The makefile for tutorial 12

```
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 see a house rendered on screen. Use your knowledge to rotate it or expand the .obj loader. You can also use this knowledge to try to create other loaders. I have also include krat.obj and krat.mtl which will render a crate. Try it out!

Please note: The textures, object and material file need to be put in the folder with the EBOOT.PBP

Source files

Download the source files.