TUTORIAL 4: 3DS LOADER

Home
Tutorials
The 3d engine
OpenGL and GLUT
Texture mapping
3ds File Loader
Vectors and lighting
Projects
Useful links
Store
About

INTRODUCTION

It's now time to say goodbye to our dear cube! In this lesson we will develop a routine to load 3ds objects, a very popular file format on the internet and supported by various 3d modelers. A 3d modeler allows you to create any type of object in a more intuitive and human way rather than to define by hand the coordinates of the vertices, which can become an impossible task even for simple objects just slightly more complicated than a cube. Actually, I am very reluctant to throw away the cube, such a simple and perfect figure. However, until proven otherwise, spaceships, planets, missiles and anything that has to do with a space simulator seems to be completely different from the cube.

Before starting to write the code it will be necessary to analyze the 3ds file structure. Ok, prepare your favorite programming drink and get ready...

THE 3DS FILE STRUCTURE

A 3ds file contains a series of information used to describe every detail of a 3d scene composed of one or more objects. A 3ds file contains a series of blocks called **Chunks** What is contained in these blocks? Everything necessary to describe the scene: the name of each object, the vertices coordinates, the mapping coordinates, the list of polygons, the faces colors, the animation keyframes and so on...

These chunks don't have a linear structure. This means that some chunks are dependent on others and can only be read if their relative parent chunks have been read first. It's not necessary to read all the chunks and we will only consider the most important ones here
I will base my description of the 3ds file format on the information contained in the 3dsinfo.txt file written by Jochen Wilhelmy which explains in detail the structure of all the chunks.
A chunk is composed of 4 fields:
-Identifier a hexadecimal number two bytes in length that identifies the chunk. This information immediately tells us if the chunk is useful for our purpose. If we need the chunk we can then extrapolate the scene information in it and, if necessary, any child chunks it may have. If we don't need the chunk, we jump it using the following information...
-Length of the chunk: a 4 byte number that is the sum of the chunk length and all the lengths of every contained sub-chunk.
-Chunk data: this field has a variable length and conatians all the data for the scene.

This table shows the offset (in bytes) and the length (also in bytes) of each field in a typical chunk:

Offset	Length	
0	2	Chunk identifier
2	4	Chunk length: chunk data + sub-chunks(6+n+m)
6	n	Data
6+n	m	Sub-chunks

We can see from the last line in the table exactly how some chunks are dependent on others: each child chunk is in fact contained inside the field "Sub-chunks" of the parent chunk.

The following are the most important chunks in a 3ds file. Please note the hierarchy among the various elements:

```
MAIN CHUNK 0x4D4D
   3D EDITOR CHUNK 0x3D3D
      OBJECT BLOCK 0x4000
         TRIANGULAR MESH 0x4100
            VERTICES LIST 0x4110
            FACES DESCRIPTION 0x4120
               FACES MATERIAL 0x4130
            MAPPING COORDINATES LIST 0x4140
               SMOOTHING GROUP LIST 0x4150
            LOCAL COORDINATES SYSTEM 0x4160
         LIGHT 0x4600
            SPOTLIGHT 0x4610
         CAMERA 0x4700
      MATERIAL BLOCK 0xAFFF
         MATERIAL NAME 0xA000
         AMBIENT COLOR 0xA010
         DIFFUSE COLOR 0xA020
         SPECULAR COLOR 0xA030
         TEXTURE MAP 1 0xA200
         BUMP MAP 0xA230
         REFLECTION MAP 0xA220
         [SUB CHUNKS FOR EACH MAP]
            MAPPING FILENAME 0xA300
            MAPPING PARAMETERS 0xA351
      KEYFRAMER CHUNK 0xB000
         MESH INFORMATION BLOCK 0xB002
         SPOT LIGHT INFORMATION BLOCK 0xB007
         FRAMES (START AND END) 0xB008
            OBJECT NAME 0xB010
            OBJECT PIVOT POINT 0xB013
            POSITION TRACK 0xB020
            ROTATION TRACK 0xB021
            SCALE TRACK 0xB022
            HIERARCHY POSITION 0xB030
```

As mentioned earlier, If we want to read a particular chunk we must always read its parent chunk first. Imagine the 3ds file is a tree and the chunk that we need is a leaf (and we are a little ant on the ground). In order to reach the leaf, we need to start from the trunk and cross any branches that lead to that leaf. For example, if we want to reach the chunk VERTICES LIST, we have to read the MAIN CHUNK first, then the 3D EDITOR CHUNK, the OBJECT BLOCK and finally the TRIANGULAR MESH chunk. The other chunks can safely be skipped.
Now let's prune our tree and leave only the branches we are going to use in this tutorial : "vertices", "faces", "mapping coordinates" and their relative parents:

```
MAIN CHUNK 0x4D4D
   3D EDITOR CHUNK 0x3D3D
      OBJECT BLOCK 0x4000
         TRIANGULAR MESH 0x4100
            VERTICES LIST 0x4110
            FACES DESCRIPTION 0x4120
             MAPPING COORDINATES LIST 0x4140
```

Here are the chunks described in detail :

MAIN CHUNK	
Identifier	0x4d4d
Length	0 + sub-chunks length
Chunk father	None
Sub chunks	3D EDITOR CHUNK
Data	None
3D EDITOR CHUNK	
Identifier	0x3D3D

Length	0 + sub-chunks length
Chunk father	MAIN CHUNK
Sub chunks	OBJECT BLOCK, MATERIAL BLOCK, KEYFRAMER CHUNK
Data	None
OBJECT BLOCK	
Identifier	0x4000
Length	Object name length + sub-chunks length
Chunk father	3D EDITOR CHUNK
Sub chunks	TRIANGULAR MESH, LIGHT, CAMERA
Data	Object name
TRIANGULAR MESH	
Identifier	0x4100
Length	0 + sub-chunks length
Chunk father	OBJECT BLOCK
Sub chunks	VERTICES LIST, FACES DESCRIPTION, MAPPING COORDINATES LIST
Data	None
VERTICES LIST	
Identifier	0x4110
Length	varying + sub-chunks length
Chunk father	TRIANGULAR MESH
Sub chunks	None
Data	Vertices number (unsigned short) Vertices list: x1,y1,z1,x2,y2,z2 etc. (for each vertex: 3*float)
FACES DESCRIPTION	
Identifier	0x4120
Length	varying + sub-chunks length
Chunk father	TRIANGULAR MESH
Sub chunks	FACES MATERIAL
Data	Polygons number (unsigned short) Polygons list: a1,b1,c1,a2,b2,c2 etc. (for each point: 3*unsigned short) Face flag: face options, sides visibility etc. (unsigned short)
MAPPING COORDINATES LIST	
Identifier	0x4140
Length	varying + sub-chunks length
Chunk father	TRIANGULAR MESH
Sub chunks	SMOOTHING GROUP LIST
Data	Vertices number (unsigned short) Mapping coordinates list: u1,v1,u2,v2 etc. (for each vertex: 2*float)

Now that the 3ds file format is clear enough, we are going to take a look at the code for this tutorial. What? You're completely lost? =D Let's continue anyway. The "chunks" structure will become clearer to you as you go through the lesson. After all, we are programmers and we understand C better than own chatter ;)

A SHORT BRIEFING

The steps we need to take in order to load a 3ds object and save it in the format defined by our

engine are:
-implement a "while" loop (as we did for the texture loader) that continues its execution until the end of file is reached.
-read the chunk_id and the chunk_length each iteration of the loop.
-analyze the content of the chunk_id using a switch .
-if the chunk is a section of the tree we don't need to read, we jump the whole length of that chunk by moving the file pointer to a new position which is calculated by using the length of the current chunk added to the current position. This allows us to jump any chunk we don't need as well as all contained sub-chunks. In other words: let's jump to another branch! Are you starting to feel like a monkey yet? =)
-if the chunk allows us to reach another chunk that we need, or it contains data that we need, then we read its data if needed, and then move to the next chunk.

FINALLY... CODE!

The first thing to do is to create the files that will contain the new routines.
We have used the file tutorial(n).cpp to contain the main data types of the engine in the previous tutotials. However, since our data structures are becoming bigger, we will insert the declarations of the data types in a header file that we will call tutorial4.h.

First, we increase the number of vertices and polygons that our engine is able to manage.

```
#define MAX_VERTICES 8000
#define MAX_POLYGONS 8000
```

Next, we add the field char name[20]; to the structure obj_type. This field will contain the name of the loaded object.
Lastly, we modify the name of our object variable from **obj_type cube;** to **obj_type object;** just to "highlight" the generic nature of our object.

The next file to create is 3dsloader.cpp. In this file, we insert the following routine:

```
char Load3DS (obj_type_ptr p_object, char *p_filename)
{
   int i;
   FILE *l_file;
   unsigned short l_chunk_id;
   unsigned int l_chunk_length;
   unsigned char l_char;
   unsigned short l_qty;
   unsigned short l_face_flags;
```

The Load3DS routine accepts two parameters: a pointer to the object data structure and the name of the file to open. It returns "0" if the file has not been found or "1" if the file has been found and read. There aren't too many variables to initialize: we have the usual counter **i**, a pointer to the file ***l_file** and a support variable to extrapolate byte data **l_char**.
The other variables are:
-unsigned short l_chunk_id; a 2 byte hexadecimal number that tells us the chunk's id.
-unsigned int l_chunk_length; a 4 byte number used to specify the length of the chunk.
-unsigned short l_qty; a support variable that will tell us the quantity of information to read.
-unsigned short l_face_flags; This variable holds various information regarding the current polygon (visible, not visible etc.) which the 3d editor uses to render the scene. We will only use this value to move the file pointer to the next chunk position.

So let's open the file at last!

```
   if ((l_file=fopen (p_filename, "rb"))== NULL) return 0; //Open the file
   while (ftell (l_file) < filelength (fileno (l_file))) //Loop to scan the whole
file
   {
```

The **while** loop is performed for the entire length of the file. The **ftell** function allows us to acquire the current file pointer position while **filelength** returns the length of the file.

```
fread (&l_chunk_id, 2, 1, l_file); //Read the chunk header
fread (&l_chunk_length, 4, 1, l_file); //Read the length of the chunk
```

Here, we have extrapolated the identifier and the length of the chunk and have saved them in l_chunk_id and l_chunk_length respectively .
First, we analyze the content of l_chunk_id.

```
switch (l_chunk_id)
{
   case 0x4d4d:
   break;
```

We have found the **MAIN CHUNK!** Cool! What are we going to do with it? Simple... nothing! In fact, the MAIN CHUNK has no data. However, we are interested in its sub-chunks. We have included this particular "case" statement so that the whole MAIN chunk is not jumped! Jumping the length of the MAIN CHUNK would have meant moving the file pointer to the end of the file due to the "default case" at the end of this switch statement. I will discuss this "default case" more, later in this tutorial

We take the same approach for the **3D EDITOR CHUNK**. This is the next node that we need to navigate through in order to reach the information we need. Once again, this node has no data. So let's pretend to read it =) This will bring us to the child called Object Block.

```
case 0x3d3d:
break;
```

The chunk **OBJECT BLOCK** finally has some interesting information: the name of the object. We store this data in the "name" field of the "object" structure. The while loop exits if the '\0' character is encountered or the number of characters exceeds 20. Be careful! We have just read all the data of this chunk and this has moved the file pointer to the next chunk.

```
case 0x4000:
   i=0;
   do
   {
      fread (&l_char, 1, 1, l_file);
      p_object->name[i]=l_char;
      i++;
   }while(l_char != '\0' && i<20);
break;
```

This next chunk is simply another empty node that is the parent node of the next chunks that we must read.

```
case 0x4100:
break;
```

Finally, here are the vertices! The chunk **VERTICES LIST** contains all the vertices of the model. First, we read the value "quantity" and use it to create a for loop that reads all the vertices. We then save each vertex in the corresponding field of the object structure.

```
case 0x4110:
   fread (&l_qty, sizeof (unsigned short), 1, l_file);
   p_object->vertices_qty = l_qty;
   printf("Number of vertices: %d\n",l_qty);
   for (i=0; i<l_qty; i++)
   {
      fread (&p_object->vertex[i].x, sizeof(float), 1, l_file);
      fread (&p_object->vertex[i].y, sizeof(float), 1, l_file);
      fread (&p_object->vertex[i].z, sizeof(float), 1, l_file);
   }
break;
```

The chunk **FACES DESCRIPTION** contains a list of the object's polygons. As explained in tutorial 1, the structure dealing with polygons doesn't contain coordinates, only numbers that correspond to

elements containing a list of vertices. In order to read this chunk we do exactly the same procedure we have done for the vertices chunk: first, we read the number of faces then we create a for loop to read all the faces.
Each face also has another 2 byte field, the face flags, that contains some information useful only for 3d editors (indicating visible faces and so on). We will only read it to move the file pointer to the next chunk.

```
         case 0x4120:
            fread (&l_qty, sizeof (unsigned short), 1, l_file);
            p_object->polygons_qty = l_qty;
            printf("Number of polygons: %d\n",l_qty);
            for (i=0; i<l_qty; i++)
            {
               fread (&p_object->polygon[i].a, sizeof (unsigned short), 1, l_file);
               fread (&p_object->polygon[i].b, sizeof (unsigned short), 1, l_file);
               fread (&p_object->polygon[i].c, sizeof (unsigned short), 1, l_file);
               fread (&l_face_flags, sizeof (unsigned short), 1, l_file);
            }
         break;
```

Finally, we read the **MAPPING COORDINATES LIST**. Once again, We read the quantity and use this value to set up a for loop. Each point has two coordinates, u and v do you remember? No?? What are you doing here then? ;)

```
         case 0x4140:
            fread (&l_qty, sizeof (unsigned short), 1, l_file);
            for (i=0; i<l_qty; i++)
            {
               fread (&p_object->mapcoord[i].u, sizeof (float), 1, l_file);
               fread (&p_object->mapcoord[i].v, sizeof (float), 1, l_file);
            }
         break;
```

The **default** case! This means that we are at the end of the routine. This case is simple: when we find chunks that we don't want to read, the **fseek** function moves the file pointer to the beginning of the next chunk using the chunk_length information

```
         default:
            fseek(l_file, l_chunk_length-6, SEEK_CUR);
      }
   }
```

We have finished! Very little remains to be done. We close the file and return 1.

```
   fclose (l_file); // Closes the file stream
   return (1); // Returns ok
}
```

CONCLUSIONS

The 3ds reader that we have developed here is a starting point for more complex readers. Keep in mind however that our routine can only read a 3ds file if there is only one object present and it is positioned at the center. The next tutorial (the matrices tutorial), will add the functionality needed to load other objects. This will be the fun part. We have to include other spaceships right? Otherwise we won't have anything to destroy =)
This lesson wasn't so hard was it? After all, we have already done the big work in previous lessons. We can use all the code written so far for the next tutorial, in which we will learn how to add lighting using OpenGL functions. Bye bye for now happy coders!

SOURCE CODE

To compile and execute this project you need the GLUT libraries that can be found at:www.opengl.org/developers/documentation/glut.html

Download the C/C++ source code and executable in zip format:
Windows version
Linux version (port by Panteleakis Ioannis)
SDL version (port by Afrasinei Alexandru)
MAC OS version (port by Martin Williams)

UTILITIES

Mingw32 makefile (To use the Linux version on Mingw32 for Windows - by Jose Ortega)
Printer friendly version of this tutorial (by Steve Bruce)

<< PREVIOUS TUTORIAL **NEXT TUTORIAL>>**

Latest works: Tutorial 5 - Vector and lighting released Tutorial 6 - Matrices and multi objects loading - source code released!

>>2006OCT30

After this long period of silence I am here again! I am so sorry for the lack of updates but I have faced up a very hard period in my life:
I changed girlfriend (after a long story), work (now I work on my own) and home (I went out from my family house).
Things now are going better and I have in mind very nice projects.
First of all I am developing a Cutting-Edge Php/Ajax Web Framework. I use it mainly for my real work to create Web Applications and Control Panels. Spacesimulator.net website will be completely renewed using it, this means: W3C compliant, users management, forums and so on.
Second: Zetadeck project will be completely renewed too and released under the GPL licence using your collaboration to create the definitive Space Simulator in a modular approach, exactly the way the tutorials are done.
I want to tell you also that my e-mail: info a t spacesimulator d o t net is always full of spam, everyday I receive hundred unwanted messages. To avoid that I can delete inadvertely messages from you please use this other address: damianovit a t libero d o t it If you sent me an e-mail and didn't receive any answer please send me it again to this other address.
Please stay tuned!

>>2005 APR 08

Another collaborator: François Devic has done a very good work! He has translated the (very long!) 2nd tutorial to French.

>>2005MAR2

Sorry for this last long period of silence but I had many things at work to finish, I've had not time also to answer to some of your e-mail regarding tutorials support, please consider that I receive many e-mails everyday, I do my best for give support to all but I give priority to the contributors, as I wrote into the store section.

This year I started again my studies at the university, computer science at

La Sapienza in Rome, since I interrupted them 5 years ago. I discover that studing and working in the same time is really hard. Sometimes I think to stop again making exams, I found infact that for this University is not important your background history and your past knowledge, you are only a number... it is really strange... with my tutorials I helped many people making their graduation thesis and for some reason I don't succeed helping myself.

Again I want to update you with the Zetadeck project, since about one year infact I stopped the development of this space sim but I never thought to abandon this project. Now I started again working on it.

There are two new collaborators that have translated the first tutorial in French and Dutch, they are:

Sylvain Carding - French

Maarten Heidenrath - Dutch

I've also published the original italian translation of the 3th tutorial.
All the translations can be found on the tutorial page.

>>2004NOV24

Again a new collaborator: **Jonathan Dornan** has sent the english corrected version of the tutorial 5, vectors and lighting. Thank you Jonathan!

I've also published the original italian translation of the 2nd tutorial.

>>2004OCT31

Two new collaborators of spacesimulator.net have sent the translation of the first tutorial in other languages, they are:

Frank Kramer - German

Ciro Durán - Spanish

I've inserted a new column into the tutorials section in which will be inserted all the tutorials translations. For the Italian users I have also published the original **Italian translation** of the first lesson, soon I will publish also the translations of the other tutorials.

If you want help me with the translations you must download the tutorial html page, modify it and send me the new translated html file.
My apologies if you don't like the source code of the html pages, unfortunately, because of my short time, I have used a WYSIWYG compiler. I am working for a new layout (PHP based) of this website but I think to release it not until the end of this year. You can have a look at www.spacesimulator.net/test_newsite/index.php just to see something similar to the future layout I will use.

Some updates also regarding money contributions, the total amount of the donations is near 100$ (in 13 months). This is not so much but this means that probably as soon as I have to pay the renew of this web space I will use, for the first time, the money of the contributions, thank you so much! ;)

>>2004SEP03

Good news for Mac OS coders: Martin Williams, a new collaborator of spacesimulator.net, has
successfully ported the source code of the 4th tutorial - 3ds file loader - to Mac OS X with GLUT and Xcode headers. Martin has also created an utility header with endian swapping functions for easier convertions of the tutorials code for Mac OS platform.

You can download the new porting from the tutorials section.

>>2004AUG16

After a long time of coding and cleaning I have finished to write the source code of other 4 tutorials:

Tutorial 9: .INI configuration files - Parser
Tutorial 10: SDL Framework and FPS calculation
Tutorial 11: Physics: dynamics, acceleration and force
Tutorial 12: Physics: Collision detection and collision response

You can download the Windows - EXE version from the tutorial section.
I am sure: You will be really amazed to see how much Physics adds realism to the 3d engine!

To get the source code now I invite you to make a donation or help me with the tutorials, visit the store section for more information.

>>2004MAY08

I have just inserted a test page for the new website, you can find it at: www.spacesimulator.net/test_newsite/index.php
It is programmed in PHP and HTML. Many links are not working yet but the layout is more or less the final version. I would like to know your opinion.

>>2004APR16

I want to let you know about the great risk related with the software patents. **European Commission proposed a directive that would legalize software patents**, this means that the free-software is facing a great risk. This could cause the death of this web site (and many others). I pray that it will never happen.

You can find more information in this web site: http://swpat.ffii.org.

>>2004MAR25

Finally I finished to write the tutorial 5: vectors and lighting! It is a very long and tech tutorial, exactly what are you looking for ;)
Please note that this tutorial is not yet completely English-corrected so maybe you can find some mistakes, let me know any error please!

>>2004FEB25

Wow! The new web site is going to be completed soon and has really a professional aspect. I think to release it the next month.
For the rest I thank all those people that have already sent donations through PayPal (four people in four months for a total of 20 $). Without any doubt a concrete help to support this web-site =))).
I am very happy also because this website was inserted into the opengl.org tutorials section, to celebrate this event I decided to do a nice thing:
Source code of lesson 6 (matrices and multi objects loading) RELEASED in advance!!! (download it from the tutorials index)
Now you can rate me 10/10! ;-)
Finally... have a look at this screenshot:

...finally a clean space to work in... ;)

>>2004JAN28

Important info: I've temporarily disabled the subscribe[at]spacesimulator.net address because I am receiving many viruses. To continue subscribing you can send a message to info[at]spacesimulator.net with "subscribe" on the subject line. Subscribing means that I will send you e-mails only to let you know about updates. Please consider also that I don't store the e-mail addresses into the outlook address book, a location often used by viruses, but simply into a my local directory - txt file.

>>2003DEC24

Just in time for Christmas the site is updated again. Now also the tutorial 4 is more pleasant to read, of course it was corrected by Robert Napolitano!
In the meantime Zeeshan Ali has converted for Linux the tutorial 5. I have finished also to write the source code of the tutorial 8 (fonts printing), to

download it in advance before I release the correspondent lesson please read the contribute section!
I wish you a Merry Christmas! ..and Happy Holidays-Coding! ;)

>>2003NOV23

I want to thank all those people that have sent compliments for the tutorials, you are helping me to continue my work! Regarding the many questions about the future of spacesimulator.net and Zetadeck please don't worry, I have many projects for the future, I will continue to write lessons and develop Zetadeck in spite of my not much free time.
First of all I want to renew completely the site: now the HTML code is redundant and dirty because I have used a WYSIWYG compiler, therefore the most important thing is to re-write the various pages using directly the HTML. I will insert also some dynamic elements and roll-overs, a new logo, a FAQ, maybe a forum and, where possible, translations in other languages. Regarding the tutorials, thanks to the help and the suggestions of Robert Napolitano, I will modify the structure to make the various source codes like libraries. The new structure of the lessons will be based from a central knot: base code (GLUT and SDL) and from various libraries (graphics, physics, AI and so on) that can be linked together as needed. Will be inserted also conversions of the tutorials from C to C++. Obviously with this structure I have intention to satisfy any developer of videogame that will visit this site. Any other suggestion is welcome, write me!

Old messages

TUTORIALS INDEX

Home

Tutorials

The 3d engine

OpenGL and GLUT

Texture mapping

3ds File Loader

Vectors and lighting

Projects

Useful links

Store

About

SPACESIMULATOR.NET IS PROUD TO PRESENT...

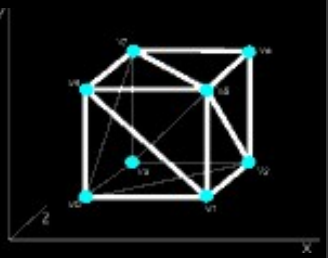
3D ENGINE TUTORIALS: FROM VERTICES DEFINITION TO SPACE FIGHTS!

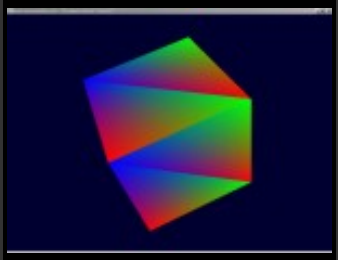
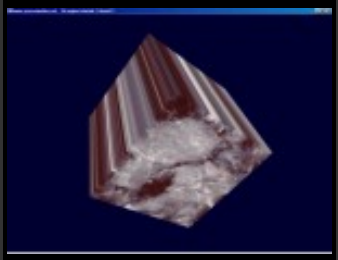
In these lessons I will explain how to construct a 3d engine in C using OpenGL. I assume that you already have a basic knowledge of C. If not, I advise you to study it before reading these tutorials (there are tons of online tutorials for C). The engine that we are going to build will be structured to account for open environments, include features suitable for space simulators (after all, we are at spacesimulator.net) and will take into account that in the future will be added sections to manage landscapes and more advanced tutorials in which I will reveal some solutions that I have implemented in my own 3D engine. However, I will proceed gradually from the most elementary concepts to the more complex ones.

More advanced tutorials are in development. I plan to release about four tutorials every year. If you want to keep yourself informed as to when I will release a new tutorial, please subscribe to the mailing list by sending a mail to subscribe[at] spacesimulator.net.

I would also appreciate it if you would tell me your opinion regarding my tutorials: how they are written, conceptual or grammar mistakes, suggestions and so on. I am only a programmer and I cannot guarantee that my work is perfect. Surely there are mistakes. I am also not responsible for any damage that may occur from using my source code and/or the files from this Web Site.

The code was developed using Visual C, but I think you won't have too many problems using other compilers. At the moment, I am looking for ports to other platforms and other languages. Please let me know if you are interested in doing it.

N.	Tutorial (English)	Translations	Source code	Screenshots	Description
1	The 3d engine	Dutch French German Italian Spanish	All platforms		Basic definitions: 3d engine, vertices and polygons. The first step to create the data structure for a 3d engine.

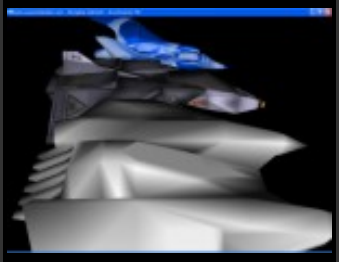
2	OpenGL and GLUT	French Italian	Windows Linux SDL MAC OS		Here we analyze the rendering pipeline, the graphic library OpenGL and the utility library GLUT. At the end of this lesson we will be able to show a rotating solid object!
3	Texture mapping		Windows Linux MAC OS		How to load an image and use it to cover an object. This technique is called texture mapping and is the most responsible for the realism of the scene.
4	3ds Loader		Windows Linux SDL MAC OS		The data structure for a complex object can't be written by hand. There are a lot of programs that help to create 3d meshes in a very quick way. In this lesson we study how to load a 3ds file, a format that is really famous on the net.

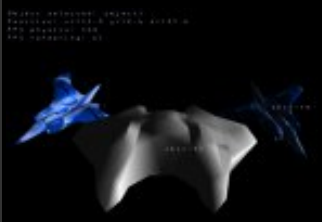
5	**Vectors, normals and OpenGL Lighting**		**Windows Linux**		Lighting is the other important thing, after texture mapping, that adds to the scene further realism. Without lights all the objects seems flat. We will study also Vectors and normals because they are indispensable to make lighting calculations.
6	**Matrices and multi-objects loading**		**Windows**		In this lesson we will improve and extend our object structure, we will be able to load some objects in our scene and manage their positions and rotations using matrices.

If you want to support this web site and have the possibility to use the **source code** of the next lessons I invite you to make a donation.

Read more info on the store section!

DONATIONS! THE BEST WAY TO KEEP ALIVE SPACESIMULATOR.NET!!!

7	**Camera and 3d freedom**		**Windows (only exe)** **Windows (source)**		Of course our biggest aim is to explore the universe by moving ourselves through it as we like, making the monitor a free moving video camera. How can we do this? Simple: with the videocamera tutorial!
8	**Print fonts using display lists**		**Windows (only exe)** **Windows (source)**		All this advanced graphic programming and then? We forgot to print simple fonts on the screen! We will do this using a bitmapped font, the best way to develop multiplatform games!
9	**.INI configuration files - Parser**		**Windows (only exe)** **Windows (source)**		Where can we store all our Universe settings? Ini files are the best choice! Very useful and portable library to read/write INI files!

10	**SDL Framework and FPS calculation**		**Windows (only exe)** **Windows (source)**		This os-platform independent framework will give us the possibility to manage all the boring stuff that a videogame needs: graphic context, sound and input.
11	**Physics: Dynamics, Acceleration and Force**		**Windows (only exe)** **Windows (source)**		Physics, the secret to give life to our spaceships. In this tutorial we will add the essential rules to our engine to simulate a real universe.
12	**Physics: Collision Detection and Collision Response**		**Windows (only exe)** **Windows (source)**		Another tutorial on Physics: this time we will learn basic collisions detection and collision response.

Home

Tutorials

The 3d engine

OpenGL and GLUT

Texture mapping

3ds File Loader

Vectors and lighting

Projects

Useful links

Store

About

TUTORIAL 1: THE 3D ENGINE

INTRODUCTION

First, it's necessary to explain the basic concept... what is a 3d engine? The answer is relatively simple: **a 3d engine is a whole of structures, functions and algorithms used to visualize, after many calculations and transformations, 3d objects on a 2d screen.**

The main sections of a 3d engine are:

1-The acquisition of the objects' data in structures.

2-The transformations to position the objects in the world.

3-Rendering the scene on the 2d screen.

This is why we are here so... let's go to work!

In this lesson, we will study how to define the main structures needed to draw a 3d object.

THE VERTICES

Suppose you have an object and want to show it on a 2d screen. In order to do this, it's necessary to obtain information about its structure. How can we do this? First, we define some key points: the vertices of the object. Every vertex is composed of three coordinates x, y, z. Every coordinate must be expressed through a FLOAT or DOUBLE variable, this is because we always need the best resolution to render the scene. This solution is used in every 3d engine but it is not free of flaws, in fact it loses precision when the numerical value is too high. This fact limits the space in which the simulation can work. In a space simulator, we should be able to move through infinite space so we will face this problem eventually. To define a vertex in C, we use a structure composed of three variables x, y, z.

```
typedef struct{
   float x,y,z;
}vertex_type;
```

It's very important to keep in mind that all of the calculations that deal with the positioning and the rotation of the object are applied to the vertices, since they are the units that make up the basic structure. The second structure, in

terms of importance, is the polygon.

THE POLYGONS

The polygons are the faces of the object and are composed from N vertices. In most 3d engines polygons are composed of 3 vertices hence we will use this rule as well.
Using our vertex_type structure, we can define a polygon structure that contains 3 vertices:

```
typedef struct{
   vertex_type a,b,c;
}polygon_type;
```

We will also declare a polygon_type array variable that will be filled with all the polygons that constitute the object.

```
#define MAX_POLYGONS 2000
polygon_type polygon[MAX_POLYGONS];
```

But... look out! Our definition assigns 3 vertices to every polygon and these vertices are not shared with the other polygons. Actually, if we reflect a while we are going to see that every polygon of an object does in fact shares its sides, and also its vertices, with other polygons. So we have made a mistake! Well, it is not really a mistake, but we have increased, considerably, the real number of vertices on the scene well beyond what is necessary. We have already said the engine will use the vertices to carry out most of its calculations so we should really find another method to define the polygons. We could create a list of vertices that will hold all of the vertices of the entire object. Then, in order to define the polygons, we will use a sort of referencing scheme to "point" to the vertices of that list. We now declare an array variable of type vertex_type that will hold MAX_VERTICES vertices.

```
#define MAX_VERTICES 2000
vertex_type vertex[MAX_VERTICES];
```

The polygon structure will not contain the vertices any more but only 3 numbers that will point to 3 elements of the vertices list. In this way more polygons can point to the same vertex. This greatly optimizes the design of the engine.

```
typedef struct{
   int a,b,c;
}polygon_type;
```

THE OBJECT

We do a little bit of cleanup here and organize the previous definitions in a

structure that we will call obj_type.

```
typedef struct{
   vertex_type vertex[MAX_VERTICES];
   polygon_type polygon[MAX_POLYGONS];
}obj_type,*obj_type_ptr;
```

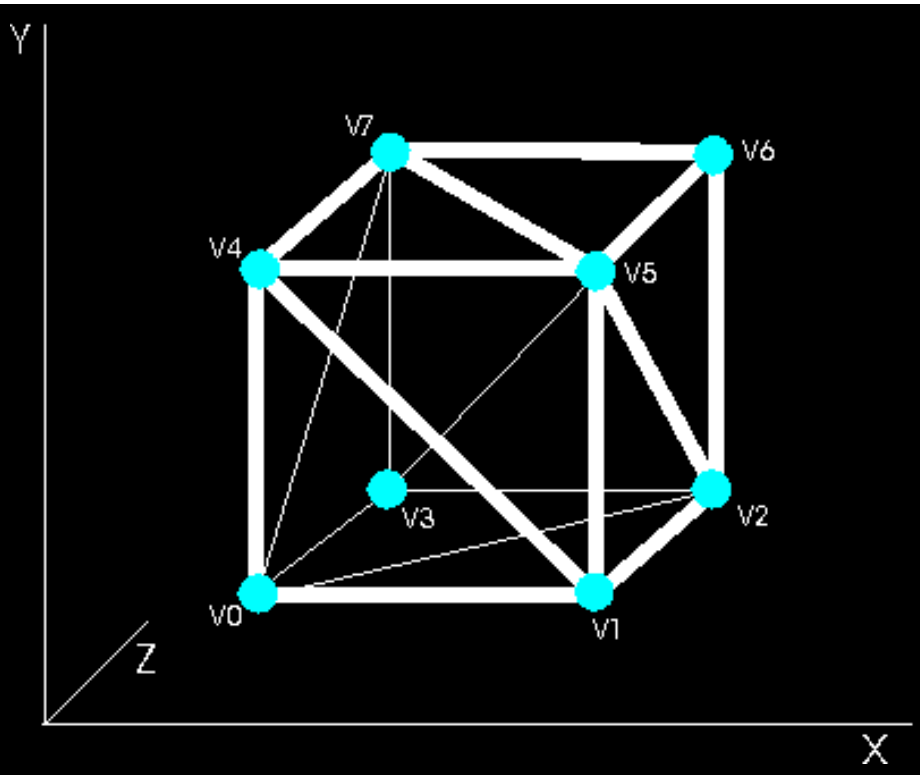
That's only a basic definition. In the future, we will add more fields that will identify the position, rotation and state of the object.

At this point we can declare the object variable and fill the vertices list:

```
obj_type obj;

obj.vertex[0].x=0;  obj.vertex[0].y=0;  obj.vertex[0].z=0; //
vertex v0
obj.vertex[1].x=1;  obj.vertex[1].y=0;  obj.vertex[1].z=0; //
vertex v1
obj.vertex[2].x=1;  obj.vertex[2].y=0;  obj.vertex[2].z=1; //
vertex v2
obj.vertex[3].x=0;  obj.vertex[3].y=0;  obj.vertex[3].z=1; //
vertex v3
obj.vertex[4].x=0;  obj.vertex[4].y=1;  obj.vertex[4].z=0; //
vertex v4
obj.vertex[5].x=1;  obj.vertex[5].y=1;  obj.vertex[5].z=0; //
vertex v5
obj.vertex[6].x=1;  obj.vertex[6].y=1;  obj.vertex[6].z=1; //
vertex v6
obj.vertex[7].x=0;  obj.vertex[7].y=1;  obj.vertex[7].z=1; //
vertex v7
```

Now the problem is on how to subdivide our cube in triangles. The answer is simple: every face of the cube is a square composed of two adjacent triangles. So our cube will be composed of 12 polygons (triangles) and 8 vertices.



The list of polygons must be filled like so:

```
obj.polygon[0].a=0;  obj.polygon[0].b=1;  obj.polygon[0].c=4;  //
polygon v0,v1,v4
obj.polygon[1].a=1;  obj.polygon[1].b=5;  obj.polygon[1].c=4;  //
polygon v1,v5,v4
obj.polygon[2].a=1;  obj.polygon[2].b=2;  obj.polygon[2].c=5;  //
polygon v1,v2,v5
obj.polygon[3].a=2;  obj.polygon[3].b=6;  obj.polygon[3].c=5;  //
polygon v2,v6,v5
obj.polygon[4].a=2;  obj.polygon[4].b=3;  obj.polygon[4].c=6;  //
polygon v2,v3,v6
obj.polygon[5].a=3;  obj.polygon[5].b=7;  obj.polygon[5].c=6;  //
polygon v3,v7,v6
obj.polygon[6].a=3;  obj.polygon[6].b=0;  obj.polygon[6].c=7;  //
polygon v3,v0,v7
obj.polygon[7].a=0;  obj.polygon[7].b=4;  obj.polygon[7].c=7;  //
polygon v0,v4,v7
obj.polygon[8].a=4;  obj.polygon[8].b=5;  obj.polygon[8].c=7;  //
polygon v4,v5,v7
obj.polygon[9].a=5;  obj.polygon[9].b=6;  obj.polygon[9].c=7;  //
polygon v5,v6,v7
obj.polygon[10].a=3; obj.polygon[10].b=2; obj.polygon[10].c=0; //
polygon v3,v2,v0
obj.polygon[11].a=2; obj.polygon[11].b=1; obj.polygon[11].c=0; //
polygon v2,v1,v0
```

You must keep in mind that in order to define the polygons properly it is necessary to always use the same clockwise or counter-clockwise direction for all the polygons on the scene. We will see in the next tutorial, how the direction is used to control weather the polygon is visible or not. So pay close attention to the way you define the polygons or many of them will not be visible. We will use the counter-clockwise method (for example the first

polygon is defined by v0,v1,v4 or v1,v4,v0 or v4,v0,v1 and not v1,v0,v4 or v0,v4,v1 or v4,v1,v0).

ANOTHER MORE ELEGANT WAY TO DEFINE OUR OBJECT

In C/C++ we can fill the obj_type structure using this more elegant way:

```
obj_type cube =
{
    {
        -10,-10, 10, //vertex v0
         10,-10, 10, //vertex v1
         10,-10,-10, //vertex v2
        -10,-10,-10, //vertex v3
        -10, 10, 10, //vertex v4
         10, 10, 10, //vertex v5
         10, 10,-10, //vertex v6
        -10, 10,-10  //vertex v7
    },
    {
        0, 1, 4, //polygon v0,v1,v4
        1, 5, 4, //polygon v1,v5,v4
        1, 2, 5, //polygon v1,v2,v5
        2, 6, 5, //polygon v2,v6,v5
        2, 3, 6, //polygon v2,v3,v6
        3, 7, 6, //polygon v3,v7,v6
        3, 0, 7, //polygon v3,v0,v7
        0, 4, 7, //polygon v0,v4,v7
        4, 5, 7, //polygon v4,v5,v7
        5, 6, 7, //polygon v5,v6,v7
        3, 2, 0, //polygon v3,v2,v0
        2, 1, 0, //polygon v2,v1,v0
    }
};
```

You can also see that we changed the vertices coordinates from 0,1 to 10,-10. This is intentional, In fact, we need to have the center of the object at 0,0 (we will understand why in the next tutorial). Our object is also 20 times bigger (just to easily manage coordinates).

CONCLUSIONS

That's all for this lesson. In the next one, we will begin to use OpenGL to show our cube on the screen. If you have any doubts or find mistakes don't hesitate to write me at info[at]spacesimulator.net

SOURCE CODE

Download the source code here: C/C++ source code

NEXT TUTORIAL>>

TUTORIAL 2: RENDERING PIPELINE, OPENGL AND GLUT

Home
Tutorials
The 3d engine
OpenGL and GLUT
Texture mapping
3ds File Loader
Vectors and lighting
Projects
Useful links
Store
About

INTRODUCTION

In the previous lesson, we constructed the first section of our rendering pipeline: how to acquire the object's data and store all the data in a structure. Today, we will complete points 2 and 3:

-Transformations to place the objects in the world (modeling and viewing transformations)
-Rendering the scene on the 2d screen (projection transformation, viewport transformation, back face culling, color and depth buffers)

Then we will introduce the graphic libraries of OpenGL and the utility libraries of GLUT. At the end of this lesson we will be able to show a rotating solid object on our screen!

TRANSFORMATIONS TO PLACE THE OBJECTS IN THE WORLD (modeling and viewing transformations)

After defining the object's structure, we must apply some transformations to all the vertices we have before we can show it on the screen. We want to create a world, or to be more exact, (being mad about space simulators) a whole universe.

The first transformation is the **MODELING TRANSFORMATION**. Since our intention is to create spaceships that, until proven otherwise, are not static objects ;-), we have to transform the object's local coordinate system (that is relative to the central position of each object) to an absolute coordinate system (that is relative to the center of the 3D universe ;-)). In other words we must translate the object by adding the current object's position in the universe (which may continue to change if there is motion) to the local vertices' coordinates.

Next is the **VIEWING TRANSFORMATION**. Our biggest aim, of course is to explore this universe by moving ourselves through it as we like, making the monitor a free moving video camera. How can we do this kind of transformation? The answer is relatively simple. In fact, you can simply consider the camera to always be at position 0,0 with no rotations. And then? Simple! We take whatever transformation we would have to apply to the camera to obtain a certain motion and we apply the opposite transformation to all the objects instead of moving the camera. Suppose for example that we wanted to move our point of view towards the object +10 points on the Z axis

and to look at the object from above (rotating on the x axis 40 degrees). What really happens in most graphic engines is that the object has made a translation of -10 on the Z axis and then a rotation of -40 degrees on the X axis. This greatly simplifies the management of the engine because the video camera will always be at the origin.

RENDER THE SCENE ON THE 2D SCREEN (back face culling, projection and viewing transformation, color and depth buffer)

The next operation to perform is the **BACK FACE CULLING**. This means we are going to exclude the triangles that are not visible, that is the faces on the back sides of the objects. We can save a lot of rendering time this way because the drawing function will draw only half of the total triangles. OpenGL will do this action for us

The next transformation is the **PROJECTION TRANSFORMATION**. At this point we need to "squash" a 3d scene onto a 2d screen. We need to simulate the Z axis because our poor monitor only has two axes X and Y. The easiest way to carry out this type of translation is to divide all the point's (x,y) coordinates relative to their Z component. The effect of this procedure is to compress the distant points so that they seem to approach the central point x = 0 and y = 0. This is exactly what happens in GL. Many of you may know this as "perspective projection".

The last transformation is the **VIEWPORT TRANSFORMATION**. All it does is convert all the points that will be used to the current viewport resolution.

A buffer is a zone of memory in which we can save some data. The OpenGL buffers are regions precisely as large as our viewport. For example, if we open a window 640 x 480 in size we allocate a buffer: 640 x 480 = 307200 pixels. That means, for 16 bit color mode: 307200*16 = 4915200 bits. That corresponds to about 614 Kbytes of video memory!

OpenGL has two main buffers: the **COLOR BUFFER** (that can be single or double) and the **DEPTH BUFFER**.

The COLOR BUFFER buffer is what we see on the screen and where the end of all the drawing operations go. After having carried out all the geometrical calculations, OpenGL begins to fill this buffer pixel for pixel, filling our triangles. If the scene is animated the COLOR BUFFER buffer is drawn and deleted every frame. The COLOR BUFFER buffer is often used in dual mode operation, called **DOUBLE BUFFER**, which is what we will use. Dual mode operation consists of displaying one buffer while the other buffer is cleaned and filled with the next frame. Once this operation is completed the buffers will be exchanged. Using this technique, the resulting animation is practically free of flicking.

Now, suppose that there are two triangles in our scene one behind the other, both visible. In this case, the order in which the triangles are drawn is very important. If we draw the triangle nearest to our point of view first and then the more distant one, the pixels of the near triangle will be covered by the far

one, creating an unpleasant effect. One technique to avoid this is to order all the visible triangles by their Z vertices. Then draw them in order, from the more distant to the nearest triangle. This technique is called "**THE PAINTER'S ALGORITHM**". We will not use this method because OpenGL supplies us with a more efficient tool: the **DEPTH BUFFER**. This buffer has the same dimensions as the COLOR BUFFER but instead of containing the pixel's colors it contains information about the depth of every pixel on the Z axis. Saving this information is very important and for a very simple reason. When we are going to draw our triangles pixel for pixel on the screen, we first do a test to see if the pixel to print is closer than the pixel already stored in the Z buffer. If it is closer then we update the depth buffer with the new value and write to the COLOR BUFFER. If it is not closer we don't consider that pixel for drawing. This produces excellent results!

We don't need to worry about coding these operations "by hand" because our great GL library will do all the calculations for us. OpenGL will also carry out all the low-level drawing operations including painting our triangles and applying colors to them as well as lighting and mapping effects.

We don't have to do anything? So,what are we doing here? Wasting our time? No! We need to supply OpenGL with all the information it needs so that it can make all those calculations, interface with the video card and perform all the low level operations using (if it is present) the 3d hardware acceleration.

FINALLY, OPENGL!

OpenGL is a library that lets us interface with the graphics hardware. It has a series of functions to draw points, lines and polygons, and it carries out all the calculations necessary for the illumination, shading and transformation of the vertices. Glut instead is a utility library used to interface OpenGL with the window system. It allows us to create a window independent of the platform used (Windows or Linux). It also handles the keyboard input.

The structure of our OpenGL program is divided into different sections:

-Init function: used to boot OpenGL and to init the modeling, viewing and projection matrices. We can also put all the init operations we require in this function.

-Resize function: called every time the user starts the program or changes the output-window resolution. This is necessary to communicate the new viewport size to OpenGL.

-Keyboard function: called every time the user presses a key.

-Drawing function: clears all the buffers (color and depth). All the modeling, viewing and projection transformations are carried out and the scene is drawn. Finally the 2 color buffers are exchanged.

-Main Cycle: an infinite loop, which calls all our functions every frame.

A typical OpenGL function looks like: **glFunctionName(GL_TYPE arguments)**. For Glut functions we have: **glutFunctionName(arguments)**. OpenGL also has custom types to help portability. These types start with the prefix "GL" and are followed by "u" (for the unsigned values) and by the type (float, int etc.). For example, we can use GLfloat or GLuint to define variable types similar to the "float" and "unsigned int" types in C.

HEADERS

The first thing to do is to include all the necessary headers: windows.h (for the windows users) and glut.h

```
#include <windows.h>
#include <GL/glut.h>
```

By including glut.h we have also indirectly included gl.h and glu.h (the OpenGL headers). It's also very important to set up the linker options in the compiler so that it includes the libraries opengl32.lib, glu32.lib and glut32.lib

Now, we must declare a function which initializes OpenGL.

INIT FUNCTION

```
void init(void)
{
   glClearColor(0.0, 0.0, 0.2, 0.0);
   glShadeModel(GL_SMOOTH);
   glViewport(0,0,screen_width,screen_height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0f,(GLfloat)screen_width/(GLfloat)
screen_height,1.0f,1000.0f);
   glEnable(GL_DEPTH_TEST);
   glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
}
```

Let's start to analyze the code:

-void glClearColor(GLfloat *red*, GLfloat *green*, GLfloat *blue*, GLfloat *alpha*); specifies the red, green, blue, and alpha values used by glClear to clear the color buffers. We use dark blue as a background color so we assign 0.2 as the blue component. The other colors are set to 0.0 and so is the alpha value (I will explain the meaning of this component in another tutorial). I forgot to tell you that in OpenGL the effective working range for the parameters is 0-1. So, we can mix all the components to create any color we want. For example, to create a yellow background we must set the red component to 0 and the green and blue component both to 1.0.
-void glShadeModel(GLenum *mode*); specifies a value representing a shading technique (the way OpenGL will fill the triangles). If we use **GL_FLAT**

for "mode" then each triangle will be drawn in **FLAT SHADING** mode and the color will be uniform (the same color for each pixel of the triangle). If we use **GL_SMOOTH** instead, OpenGL will make linear interpolations between the colors of the triangle's vertices. This technique is also called **GOURAUD SHADING**.
-void glViewport(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*); sets the dimensions of the current viewport for the viewport transformation.
-void glMatrixMode(GLenum *mode);* tells OpenGL which matrix is the current active matrix for matrix operations. What is a matrix? We have not yet spoken about matrices. This subject is too complicated for now and we will dedicate a complete tutorial to it later. You only need to know that a matrix is an object used by OpenGL to make all the geometrical transformations for now. In "mode" we can insert the value **GL_PROJECTION** to make projective transformations or **GL_MODELVIEW** to make viewing and modeling transformations. Once we specify the current active matrix, we can modify it as we like.
-void glLoadIdentity(void); resets the current active matrix (GL_PROJECTION in our case) by loading the identity matrix.
-void gluPerspective*(*GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far);* do you remember the projection transformation? This is where we put it into practice. OpenGL fills the current active matrix (GL_PROJECTION) using the values specified in gluPerspective. In "fovy" we must specify the angle of the field of view, in "aspect" the aspect ratio, in "near" and "far" the minimum and maximum distance of the clipping planes.
-void glEnable*(* GLenum *cap);* enables various capabilities. In this situation we have enabled the Z Buffer (DEPTH BUFFER).
-void glPolygonMode*(* GLenum *face*, GLenum *mode);* draws our polygons as points, lines or filled (using GL_POINT,GL_LINES or GL_FILL as a parameter).

RESIZE FUNCTION

This function is very similar to "init". It clears the buffers, redefines our viewport, and redisplays our scene.

```
void resize (int width, int height)
{
   screen_width=width;
   screen_height=height;
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glViewport(0,0,screen_width,screen_height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0f,(GLfloat)screen_width/(GLfloat)
screen_height,1.0f,1000.0f);
   glutPostRedisplay ();
}
```

Here are the functions we haven't covered:

-void glClear(GLbitfield *mask*); clears the buffers specified in "mask". We can insert more than one buffer separating them with the OR logical operator

"|". In our case we clear both the color and the depth buffer.
The call to glViewport is necessary here to redefine the new viewport with the new values stored in screen_width and screen_height.
-void glutPostRedisplay(void); This is a glut function. It calls whatever routine we insert in the glutDisplayFunc function (called in our Main function), in order to redraw the scene.

KEYBOARD FUNCTIONS

We will define two keyboard functions: one to handle the ASCII character input ("r" and "R" and a blank character ' ') and another to handle the directional keys:

```
void keyboard (unsigned char key, int x, int y)
{
   switch (key)
   {
      case ' ':
         rotation_x_increment=0;
         rotation_y_increment=0;
         rotation_z_increment=0;
      break;
      case 'r': case 'R':
         if (filling==0)
         {
            glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
            filling=1;
         }
         else
         {
            glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
            filling=0;
         }
      break;
   }
}
```

We use three variables to make our object rotate around the desired axis: rotation_x_increment, rotation_y_increment and rotation_z_increment. We can reset all these variables using the spacebar and pause our object's movement at it's current position. We can also change the drawing mode for our polygons to outlined or filled with the key "r" or "R".

```
void keyboard_s (int key, int x, int y)
{
   switch (key)
   {
      case GLUT_KEY_UP:
         rotation_x_increment = rotation_x_increment +0.005;
      break;
      case GLUT_KEY_DOWN:
         rotation_x_increment = rotation_x_increment -0.005;
```

```
      break;
      case GLUT_KEY_LEFT:
         rotation_y_increment = rotation_y_increment +0.005;
      break;
      case GLUT_KEY_RIGHT:
         rotation_y_increment = rotation_y_increment -0.005;
      break;
   }
}
```

This function is very similar to the last one but it handles the directional keys. Notice the Glut constants GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_LEFT and GLUT_KEY_RIGHT that identify their respective directions and increase or decrease our rotational values.

DISPLAY FUNCTION

Ladies and gentlemen, the function you've been waiting for: the drawing function!

```
void display(void)
{
   int l_index;
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0,0.0,-50);
   rotation_x = rotation_x + rotation_x_increment;
   rotation_y = rotation_y + rotation_y_increment;
   rotation_z = rotation_z + rotation_z_increment;
   if (rotation_x > 359) rotation_x = 0;
   if (rotation_y > 359) rotation_y = 0;
   if (rotation_z > 359) rotation_z = 0;
   glRotatef(rotation_x,1.0,0.0,0.0);
   glRotatef(rotation_y,0.0,1.0,0.0);
   glRotatef(rotation_z,0.0,0.0,1.0);
```

The first part of this function clears the color and depth buffer and applies the viewing and modeling transformations. We set the modelview matrix as the current active matrix using glMatrixMode with GL_MODELVIEW. Then, we initialize this matrix each frame with a call to glLoadIdentity.

-void glTranslatef(GLfloat *x*, GLfloat *y*, GLfloat *z*); moves our object in 3D space. This function multiplies the model matrix by a translation matrix defined using the x,y,z parameters. The letter "f" at the end of the name indicates that we are using float type values, instead of using the letter "d" which indicates double type parameters. We use glTranslate to move the object 50 points forward in this case. Do you remember the video camera and the viewing transformation? Well, we can consider this operation as a translation of -50 for our video camera. This movement is necessary because we must move a small distance away from the object so that we can see it. Once you compile this project you can try to modify the Z value just to see

how it affects the distance.
-void glRotatef(GLfloat *angle*, GLfloat *x*, GLfloat *y*, GLfloat *z*); This function multiplies the current matrix by a rotation matrix. It's used to apply rotations of *angle* degrees around the vector (x,y,z). The variables rotation_x, rotation_y and rotation_z store the rotation values of the object. We consider this transformation a model transformation because we only rotate the object and not the point of view. As you can see, the differences between modeling and viewing transformations are not so evident. However, we don't need to worry about this now. We will face this subject in the video camera tutorial.

```
   glBegin(GL_TRIANGLES);
   for (l_index=0;l_index<12;l_index++)
   {
      glColor3f(1.0,0.0,0.0);
      glVertex3f( cube.vertex[ cube.polygon[l_index].a ].x, cube.
vertex[ cube.polygon[l_index].a ].y, cube.vertex[ cube.polygon
[l_index].a ].z);
      glColor3f(0.0,1.0,0.0);
      glVertex3f( cube.vertex[ cube.polygon[l_index].b ].x, cube.
vertex[ cube.polygon[l_index].b ].y, cube.vertex[ cube.polygon
[l_index].b ].z);
      glColor3f(0.0,0.0,1.0);
      glVertex3f( cube.vertex[ cube.polygon[l_index].c ].x, cube.
vertex[ cube.polygon[l_index].c ].y, cube.vertex[ cube.polygon
[l_index].c ].z);
   }
   glEnd();
   glFlush();
   glutSwapBuffers();
}
```

The second part of the display function uses the functions **glBegin** and **glEnd**. These two commands mark the vertices that define a graphic primitive.

-void glBegin(GLenum *mode*); indicates the beginning of a vertex-data list that define a geometric primitive. In the "mode" parameter we can insert the type of primitive we are going to draw. There are ten available types (GL_TRIANGLES, GL_POYLGON, GL_LINES etc.). We use **GL_TRIANGLES** because we want to draw our cube using 12 triangles. We do this by starting a "for" loop in which we make 3 calls to glVertex3f and glColor3f.
-void glVertex3f(GLfloat *x*, GLfloat *y*, GLfloat *z*); specifies the x,y,z coordinates of a vertex. The "3f" indicates that there are three parameters of type float. If you only need to work with values x and y of type double, the correct command is: void **glVertex2d**(GLdouble *x*, GLdouble *y*). We insert our object's geometrical data in each parameter.
-void glColor3f(GLfloat *red*, GLfloat *green*, GLfloat *blue*); specifies the current active color. We define a different color (red 1,0,0 - green 0,1,0 - blue 0,0,1) for each vertex. Notice how the OpenGL shading mode GL_SMOOTH affects the final colors of the triangle. We see that the colors are interpolated from vertex to vertex so that they change gradually. That's very nice too see!
-void glEnd(void); ends our object definition.
-void glFlush(void); used to force OpenGL to draw the scene. The color and the depth buffers are filled and the scene is now visible. Finally!
-void glutSwapBuffers(void); exchanges the back buffer(where all the drawing functions work) with the front buffer (what we see in our window)

when in double buffered mode. If we are not in double buffered mode this function has no effect.

THE MAIN FUNCTION

```
int main(int argc, char **argv)
{
```

The following four functions of the glut library allow us to create our window for the graphic output.

```
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(screen_width,screen_height);
   glutInitWindowPosition(0,0);
   glutCreateWindow("www.spacesimulator.net - 3d engine
tutorials: Tutorial 2");
```

-void glutInit(&argc, argv); initializes the glut library. We need to call this before calling any other glut functions.
-void glutInitDisplayMode(unsigned int *mode*); sets the display mode. The glut constants GLUT_DOUBLE, GLUT RGB, GLUT DEPTH define the double buffered mode, the RGB color mode and depth buffer respectively.
-void glutInitWindowSize(int *width*, int *height*); and **void glutInitWindowPosition(int *x*, int *y*);** used to set the dimensions and the initial position of the output window
-int glutCreateWindow(char *name*); creates our output window.

To define the callbacks we use:

```
   glutDisplayFunc(display);
   glutIdleFunc(display);
   glutReshapeFunc (resize);
   glutKeyboardFunc (keyboard);
   glutSpecialFunc (keyboard_s);
   init();
   glutMainLoop();
}
```

-void glutDisplayFunc(void (*func) (void)); specifies the function to call when the window needs to be redisplayed, that is when there is a glutPostRedisplay call or when there is an error reported by the window system.
-void glutIdleFunc(void (*func) (void)); sets the idle callback function: the function called every time there are no window system events. This means that our idle function "display" is continuously called, unless window events are received. This will keep our animation working.
-void glutReshapeFunc(void (*func) (void)); sets the reshape callback function.
-void glutKeyboardFunc(void (*func) (void)); keyboard callback function for ASCII characters.

-void glutSpecialFunc(void (*_func_) (void)); keyboard callback function for non-ASCII characters.
-void glutMainLoop(void); starts the glut infinite loop, with event processing.

CONCLUSIONS

So? Yes, that's really all there is! I know, it was hard work but look out! You are now able to create a real 3d rotating object! That means you made your first 3d engine! In the next lesson, we will study how to do texture mapping. Now, it's time to let me know your opinion. Please tell me about any bugs you noticed. Write me at info[at]spacesimulator.net

SOURCE CODE

To compile and execute this project you need the GLUT libraries that can be found at:www.opengl.org/developers/documentation/glut.html

Download the C/C++ source code and executable in zip format:
Windows version
Linux version (port by Panteleakis Ioannis)
SDL version (port by Afrasinei Alexandru)
MAC OS version (port by Karakoussis Apostolos)

<< PREVIOUS TUTORIAL NEXT TUTORIAL>>

TUTORIAL 3: TEXTURE MAPPING

Home
Tutorials
The 3d engine
OpenGL and GLUT
Texture mapping
3ds File Loader
Vectors and lighting
Projects
Useful links
Store
About

INTRODUCTION

In the last lesson, we printed the faces of a cube using some OpenGL functions after having analyzed the rendering pipeline. Drawing polygons is the most critical step in a 3d engine because it involves almost all of the CPU processing power! Hence, I must first explain this stage of the pipeline in greater depth. I then begin with some general theory on how to implement the texture mapping technique. This theory is then put into practice using the OpenGL high-level commands.

DRAWING POLYGONS

Once upon a time there was MS DOS... When graphic engines were developed for this operating system, a lot of assembly code was required to draw a single point on the screen. In the Windows 95 Era, most developers still worked on their engine in a low-level approach since the hardware didn't allow fast applications to be created using graphics libraries. The first step was to initialize the graphic context using the interrupt 0x13 of the BIOS in order to work in high resolution (the standard VESA). Then the developer created all the basic functions to draw points, lines and so on. We are very lucky. Powerful modern PCs allows us to work under Windows or Linux without any loss of speed. Moreover, our dear OpenGL saves us days or even months of work by allowing us to fully exploit all the features of our expensive 3d video card.
We learned how to draw polygons as points, lines, or filled with colors using only a simple command (glPolygonMode) in the 2nd tutorial. I briefly explain the various methods here again:
- Draw polygon as points: the simplest and fastest approach, all the points corresponding to the polygon's vertices are printed on screen.
- Draw polygon as lines: also called Wireframe. The perimeter of the polygon is drawn connecting all the vertices through segments. A fast algorithm (Bresenham) is used To draw every segment.
- Polygon filled with color: requires more resources in comparison to the previous methods because it must print all the intemediate points to fill the polygon. Every horizontal line is drawn using the Bresenham algorithm starting from the upper vertex and linearly interpolating the edges of the polygon to find the beginning and end of every line.
- Polygon with texture mapping: texture mapping is a technique used to cover an object with an image. Each polygon of the object is assigned a small section of the image.
The procedure to fill a polygon using texture mapping is very similar to the

one used to fill it with color.

THE MAIN STEPS OF TEXTURE MAPPING

There are approximately three steps to add to our 3d engine to implement texture mapping:
- Load the texture in memory: the first thing to do is to create a function that is able to read an image file and save it in memory. We are going to use the Bitmap format since it is the most supported in the Windows environment and it doesn't include any compression algorithm which would only complicate matters right now.
- Assign a 2d point of the texture to every vertex: we must first add some fields to the structure obj_type at this point. The new variables will be used to match up each (3d) vertex with a (2d) point of the image, in order to cover the object as desired.
- Draw the polygons of the object "covering them" with sections of the texture: this phase (called the "filling" phase) is the most critical. We don't have to worry about coding this by hand since OpenGL already gives us some simple high level commands that make our life easy.
Well, enough of this chatter, let's get down to work...

LOADING AN IMAGE

First of all, let's include a new C/C++ file in our project and call it "texture.cpp". This file will contain all the routines we need to manipulate images. Those of you not having much experience at this point may be afraid because we are using more than one file to create our project. Don't panic! We are actually simplifying the job! A 3d engine, just like any other program of a certain complexity, needs to have a modular structure. So, let's create the file and begin to write the function LoadBitmap:

```
int LoadBitmap(char *filename)
{

   unsigned char *l_texture;
   int i, j=0;
   FILE *l_file;
   BITMAPFILEHEADER fileheader;
   BITMAPINFOHEADER infoheader;
   RGBTRIPLE rgb;
```

-unsigned char *l_texture; A pointer to the zone of memory where we will insert the image. Every point of the image is represented by 4 values of unsigned char (with a range of 0-255), one for each color component.
-int i, j=0; Some variables useful for iteration in this routine
-FILE * l_file; A pointer to the Bitmap file opened with the fopen function.

The next few variables are very interesting. In fact, they allow us to easily

read a Bitmap file since they are structures made for this specific purpose. **BITMAPFILEHEADER fileheader;** This is our fileheader! This structure contains information regarding the type and the size of the bmp file to load. This structure only helps us find the zone of memory where the file is stored. The **BITMAPINFOHEADER infoheader;** structure will give us very important information: the width and the height of the image.

Next, the global variable num_texture which represents the number of the loaded texture (useful for OpenGL to reference that texture). is increased. Our function will return this value.

```
   num_texture++;
```

Now we can open the file in read mode (if the file doesn't exist our function will return the value "-1"), read the fileheader and, "fseek" through the data. The pointer to the file is shifted up to the beginning of the next header. Let's go there now and read the infoheader!

```
   if( (l_file = fopen(filename, "rb"))==NULL) return (-1);
   fread(&fileheader, sizeof(fileheader), 1, l_file);
   fseek(l_file, sizeof(fileheader), SEEK_SET);
   fread(&infoheader, sizeof(infoheader), 1, l_file);
```

The fields biWidth and biHeights of the infoheader contain the width and the height of the image. These values will be used to assign the exact quantity of memory for storing the texture. In fact, the size of the image is measured by its height x, its width x and its color depth. Bitmap images have a color depth of 3 bytes. Each byte is a color component of either red, green or blue. This method of storing the image is called RGB.

```
   l_texture = (byte *) malloc(infoheader.biWidth * infoheader.
biHeight * 4);
   memset(l_texture, 0, infoheader.biWidth * infoheader.biHeight
* 4);
```

The malloc function assigs a zone of memory to the pointer variable l_texture. Since l_texture is now full of junk values, we use the function memset to clean that zone of memory, and fill l_texture with zeros.

Now... we have our zone of memory, ready and cleaned! There is only a thing to do: fill it with the image! So let's write this little algorithm:

```
   for (i=0; i < infoheader.biWidth*infoheader.biHeight; i++)
   {
      fread(&rgb, sizeof(rgb), 1, file);

      l_texture[j+0] = rgb.rgbtRed; // Red component
      l_texture[j+1] = rgb.rgbtRed; // Green component
      l_texture[j+2] = rgb.rgbtBlue; // Blue component
      l_texture[j+3] = 255; // Alpha value
      j += 4; // Go to the next position
   }
```

Many of you may already have an understanding of how an image is saved in a file. Basically, every point of the texture (refered to as TEXEL from now on)

is represented by 3 values RGB. The whole image is a vast series of these points placed side by side. When a line is complete the next point begins on the underlying line starting from the left.

Using fread our FOR loop first reads a single point of the image along with the three RGB values. The variable rgb used by this function is defined implicitly in windows.h (along with BITMAPFILEHEADER and BITMAPINFOHEADER) and is composed of 3 byte values **(rgb.rgbtRed,rgb.rgbtGreen,rgb.rgbtBlue).**

The next lines save the contents of every RGB component in l_texture increasing the pointer by one for four values. Why four and not three? We have read an image with a color depth of 3 and have created a texture that has a color depth of 4?! There is another component set to 255... What is it? This is the Alpha component! The Alpha component doesn't interest us for now. It will be very useful however when I introduce the topic of Blending where it can be used for representing the transparency level of the texture. So stop complaining (otherwise I will insert another component! ;-P) and let's continue...

```
   fclose(l_file); // Closes the file stream
```

The file can now be closed since we have finished reading the image data. Our texture is stored in l_texture ready to be used! Nice isn't it? We will communicate all our happiness to OpenGL and it will immediately reward us by giving us more work to do...;-) I will need to introduce a series of OpenGL commands that will define some parameters which are needed in order to correctly interface our "raw" texture with the OpenGL layer.

The first thing we need to do is tell OpenGL what texture number to use. The global variable num_texture holds this value and is increased for every call to the function LoadBitmap. Therefore every image we load will have one unique number. We then need to use some function calls to set some very important parameters. The overall quality of the final result will depend on the parameters used in these calls. However, keep in mind that the better the result, the more rendering time and processing power will be required, and that means a lower FPS count. Finally, we give OpenGL the pointer to the zone of memory where the image is saved.

```
   glBindTexture(GL_TEXTURE_2D, num_texture);

   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

   glTexImage2D(GL_TEXTURE_2D, 0, 4, infoheader.biWidth,
infoheader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, l_texture);
   gluBuild2DMipmaps(GL_TEXTURE_2D, 4, infoheader.biWidth,
infoheader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, l_texture);
```

-glBindTexture(GLenum *target*, GLuint *texture*); This function specifies the id of the current texture. OpenGL needs to know the number of the

current texture and the "texturing target " before doing any other operations (GL_TEXTURE_1D, for uni-dimensional texture or GL_TEXTURE_2D for a 2d texture). This command must be used in both the texture loading phase and the rendering phase.

-void glTexParameterf(GLenum *target*, GLenum *pname*, GLfloat *param*); This function sets some important parameters for the rendering of the texture. In target we must insert GL_TEXTURE_1D or GL_TEXTURE_2D (same as the last function). pname holds the name of the OpenGL parameter to modify. The parameters GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T select the behaviour of OpenGL when the coordinates of the texture go over the limits (generally 0,1). If we use the value GL_REPEAT in param (as we did) the result is a repetition of the texture starting from the beginning. For example, suppose you need to draw a floor and you need a texture that reproduces the look of tile. In this case, the tile would be repeated every time the end of the coordinate (s or t) is reached. We must plan the texture coordinates in the correct way to do this but don't worry about that particluar aspect for now. If the value GL_CLAMP is used in param the last pixel will be used to continue the mapping. This causes OpenGL to to fill the remaining area using the last pixel rather than repeat the texture.
GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER tell OpenGL how to behave when the texels have to be drawn in a greater or smaller space than their dimensions. This is due to the fact that a single texel will rarely correspond to a single pixel on the screen after the texture is applied to the object and the scene has been subjected to all the transformations (modeling, viewing etc.). If our object has been placed very far from the point of view, each pixel of the screen cooresponds to more than one texel. However, if the object is drawn very close to the point of view a single texel requires many pixels. In both cases the problem is to decide what texel to draw and how to filter the image in order to avoid mangled results. If we use the GL_NEAREST parameter, the texel nearest a given pixel will be drawn. The GL_LINEAR parameter uses a weighted average on a 2x2 array of texels sourrounding the pixel. There are other parameters that make use of something called called MipMaps (when our image needs to be smaller). However, I will not get into too many OpenGL details for now (there are special books for this). Our overall purpose is to create a 3d engine so the functions will be analysed gradually.

-void glTexEnvf(GLenum *target*, GLenum *pname*, GLfloat *param*);
Normally when texture mapping is performed only the texels are used to draw every point on the surface of our object. However, this function allows us to modulate the colors of the texture with those colors that the polygon would have without texture mapping. We must insert the parameter GL_TEXTURE_ENV in target. In the pname parameter GL_TEXTURE_ENV_MODE is used. We then must decide the way to combine the colors, valid parameters are: GL_DECAL, GL_REPLACE, GL_MODULATE AND GL_BLEND. There is no need to modulate the colors of the texture map in our example. We want to use the colors of the texels. Therefore, we use GL_REPLACE in that field.

-void glTexImage2D(GLenum *target*, GLint *level*, GLint *internalformat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **pixels*); This command is very important since it allows us to define a 2d texture. Again, GL_TEXTURE_2D is the first parameter. The level parameter is used to set the number of the texture used for different levels of detail. These textures will be created automatically, therefore, we put 0 as the value. We insert the value 4 (the RGBA format) in the internalformat parameter which specifies the internal

storage format of our image. The following two parameters denote the width and height dimensions of the texture. We don't need a texture border now so this field is 0. The parameters format and type describe the format and the type of data in which our texture has been saved in memory. We must use the values GL_RGBA and GL_BYTE in these parameters. Finally, we specify the pointer to the zone of memory where we have stored our beloved texture in *pixels

I have already spoken about the problems involved when there is not an exact correspondence between the texels and the pixels on the screen. Obvioulsly, objects could be very far from the point of view, especially in an engine like ours in which we will use "spatial"coordinates. What happens really is due to perspective projection. The apparent size of each object is very dependant on the distance it is from the point of view. This is a slight problem. It's not easy to maintain good image quality if the textures are rendered very far away, even with particular systems of filtration. A solution to this problem is the use of Mipmaps. Mipmaps are a series of textures derived from the main texture and already filtered. Each Minimap has a smaller resolution than the previous one. It starts from the basic texture with native dimensions of the power of 2 (for instance 256x256). The consequential Mipmaps will have dimensions halved: 128x128, 64x64, 32x32, etc. Depending on the distance, only the texture that best satisfies the principle that "texel size = 1" will be drawn. We can manually insert these textures, if they are ready by calling the function glTexImage for each MipMap. Each call to this function resets the parameters according to the number of Mipmaps defined and inserts the new texture size and a different zone of memory for storage every time. However, our Mipmaps have not been defined yet! In fact we won't define them by hand because there is a beautiful function in the GLU utility library that will do this job for us:
-void gluBuild2DMipmaps(GLenum *target*, GLint *internalformat*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **pixels*); notice that the parameters of this function are the same of the previous one. Only level and border are not defined here. Once declared (using the same values of glTexImage2D) the Mipmaps will be automatically created using the texture pointed to by *pixel as the starting point. Great, isn't it? Well, now we can free the zone of memory holding the image.

```
    free(l_texture);
```

Some of you may think that this procedure deletes the current texture we have loaded. Don't worry. OpenGL automatically stores the textures in its internal memory. So, we don't need to leave the old zone of memory full of values that we won't use anymore.
Lastly, the number of the texture loaded is returned to the caller.

```
    return (num_texture);
}
```

You are a little bit worn-out, aren't you? Well! That's a good sign after all we are programmers and stress is our best and faithful friend... Now let's relax a little bit with phase 2. It's not too hard to understand...

ASSIGN A 2D POINT OF THE TEXTURE TO EVERY VERTEX

With our texture already defined, we now need to modify our structure object_type by adding some fields. Our goal here is to associate each vertex to a 2d point of the image, in order to cover the object as desired.
Let's define a new type:

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

The mapcoord_type structure initializes 2 variables u and v. These are just a couple of coordinates used to identify a 2d point in a texture. Why are we calling two coordinate variables u and v rather than x and y? The reason is that the coordinates of a texture have been called u and v by convention for a long time. The main reason for this is to avoid confusion with the coordinates x, y and z used for vertices. We must assign a point of the texture to each vertex. In essence, a small triangular section of the image must be assigned to every triangle contained in our object.
Therefore, we will modify the obj_type structure to account for texture coordinates :

```
/*** The object type ***/
typedef struct {
   vertex_type vertex[MAX_VERTICES];
   polygon_type polygon[MAX_POLYGONS];
   mapcoord_type mapcoord[MAX_VERTICES];
   int id_texture;
} obj_type, *obj_type_ptr;
```

Two things have been added: an array called mapcoord, used to store the texture coordinates for each vertex and a variable named id_texture, used to hold the current texture id number (the return value of the function LoadBitmap). Now, the structure obj_type must be filled, please have a careful look at the following code:

```
obj_type cube =
{
   {
      -10, -10, 10, // vertex v0
       10, -10, 10, // vertex v1
       10, -10, -10, // vertex v2
      -10, -10, -10, // vertex v3
      -10, 10, 10, // vertex v4
       10, 10, 10, // vertex v5
       10, 10, -10, // vertex v6
      -10, 10, -10 // vertex v7
   },
   {
        0, 1, 4, // polygon v0,v1,v4
        1, 5, 4, // polygon v1,v5,v4
        1, 2, 5, // polygon v1,v2,v5
```

```
            2, 6, 5, // polygon v2,v6,v5
            2, 3, 6, // polygon v2,v3,v6
            3, 7, 6, // polygon v3,v7,v6
            3, 0, 7, // polygon v3,v0,v7
            0, 4, 7, // polygon v0,v4,v7
            4, 5, 7, // polygon v4,v5,v7
            5, 6, 7, // polygon v5,v6,v7
            3, 2, 0, // polygon v3,v2,v0
            2, 1, 0 // polygon v2,v1,v0
      },
      {
            0.0, 0.0, // mapping coordinates for vertex v0
            1.0, 0.0, // mapping coordinates for vertex v1
            1.0, 0.0, // mapping coordinates for vertex v2
            0.0, 0.0, // mapping coordinates for vertex v3
            0.0, 1.0, // mapping coordinates for vertex v4
            1.0, 1.0, // mapping coordinates for vertex v5
            1.0, 1.0, // mapping coordinates for vertex v6
            0.0, 1.0 // mapping coordinates for vertex v7
      },
      0, // identifier for the texture
};
```

Things are going well: the texture is defined and the object structure is filled. There's nothing left to do but draw our cube using texture mapping!

DRAW THE POLYGONS "COVERING THEM" WITH SECTIONS OF THE TEXTURE

The first function to be modified is "init". During the procedure of initialization we have to enable texture mapping and call the function LoadBitmap to load the texture:

```
glEnable(GL_TEXTURE_2D);
cube.id_texture=LoadBitmap("texture1.bmp");
if (cube.id_texture==-1)
{
   MessageBox(NULL,"Image file: texture1.bmp not found",
"Zetadeck",MB_OK | MB_ICONERROR);
   exit (0);
}
```

glEnable(GL_TEXTURE_2D) enables 2d texture mapping. Notice that the return value of the function LoadBitmap is an identifier used to assign the texture to the object. When we implement the ability to manage more objects in our engine we will assign a different number to each texture we use. If the function LoadBitmap has not succeeded in loading the image, a MessageBox with the error message will be shown and the program will be interrupted. Now let's modify the drawing function and remove the calls made to glColor3f because we don't need to assign colors to the vertices anymore. Instead we use the following:

```
glBindTexture(GL_TEXTURE_2D, cube.id_texture);

glBegin(GL_TRIANGLES);
for (l_index=0;l_index<12;l_index++)
{
   /*** FIRST VERTEX ***/
   glTexCoord2f( cube.mapcoord[ cube.polygon[l_index].a ].u,
                 cube.mapcoord[ cube.polygon[l_index].a ].v);
   glVertex3f( cube.vertex[ cube.polygon[l_index].a ].x,
               cube.vertex[ cube.polygon[l_index].a ].y,
               cube.vertex[ cube.polygon[l_index].a ].z);

   /*** SECOND VERTEX ***/
   glTexCoord2f( cube.mapcoord[ cube.polygon[l_index].b ].u,
                 cube.mapcoord[ cube.polygon[l_index].b ].v);
   glVertex3f( cube.vertex[ cube.polygon[l_index].b ].x,
               cube.vertex[ cube.polygon[l_index].b ].y,
               cube.vertex[ cube.polygon[l_index].b ].z);

   /*** THIRD VERTEX ***/
   glTexCoord2f( cube.mapcoord[ cube.polygon[l_index].c ].u,
                 cube.mapcoord[ cube.polygon[l_index].c ].v);
   glVertex3f( cube.vertex[ cube.polygon[l_index].c ].x,
               cube.vertex[ cube.polygon[l_index].c ].y,
               cube.vertex[ cube.polygon[l_index].c ].z);
}
glEnd();
```

void glBindTexture (GLenum target, GLuint texture); We have already looked at this function during the initialization of the texture. The difference now is that in the parameter texture is the identifier of the texture for the current object. Therefore, this function lets OpenGL know what the the active texture is.
void glTexCoord2f (GLfloat s, GLfloat t); Each call to this function defines the two coordinates u and v of the texture (what OpenGL respectively calls s and t). This function must be inserted between the two commands glBegin and glEnd, and it must be called every time there is a call to glVertex3f. This is how a coordinate of the texture is assigned to every vertex.

THE LAST THING

The only thing left to do is insert another file into our project, and call it "texture.h" Then, add these two lines of code to it:

```
extern int num_texture;
extern int LoadBitmap(char *filename);
```

You must then insert the file in the "include" section of the main cpp file:

```
#include "texture.h"
```

If you are using the command line to compile this project remember to compile the texture.cpp file or add it to the makefile.

Nothing could be easier right? =)
There are a few things to consider now about our code. If you are tired you can just skip to the conclusion. However, I must explain a little problem that you may have noticed.

A LITTLE TROUBLE...

The more observant of you may have noticed that we have only drawn the complete pattern of the texture on two faces of the cube. In fact, only the faces made up by the vertices v0,v1,v5,v4 and v3,v2,v6,v7 have the complete texture drawn. This is because the mapping coordinates are correct only for the 4 triangles that compose those 2 faces. Unfortunately, there is really nothing we can do for the remaining faces
The cause of this anomaly is due to the fact that we have tightly coupled one and only one texture coordinate to every vertex. Therefore, some faces were forced to sacrifice their texture coordinates because they share their vertices.

There are some solutions to this problem :
1- Rather than have a one to one relationship between each texture coordinate and each vertex, use the polygon type and add a texture coordinate to every point of the polygon. Now, a vertex can have more than one texture coordinate and the problem is solved. Here is an example:

```
typedef struct{
   int a,b,c;
   mapcoord_type map_a, map_b, map_c;//Every point of the polygon
has a point u,v in the texture
}polygon_type;
```

2-Add the number of vertices necessary to draw the texture on every triangle correctly. This also solves the problem, but we have made processing a lot heavier. The number of vertices will increase dramaticaly. Our cube, for example, would require 4 vertices for each face. 4 vertices x 6 faces = 24 vertices! This approach would be necessary when a certain uniformity in the object must be maintained.
So, guess which solution we're going to use? Are you thinking solution 1? Nope! I will use 2! Why have I decided to use this more complex solution? A complex figure rarely needs to have a lot of vertices added to it in order to maintain uniformity with its texture coordinates. Moreover, almost all 3d engines use solution 2. The .3ds format that we will use for loading models also works this way.
However, this whole discussion is kind of useless right now because in this tutorial we will simply draw the cube as it is for now, with its 4 ugly faces and only two of which have the texture correctly applied. We don't need to worry about implementing solution 2 by hand because the 3ds format will help us do this. In fact, 3d studio will automatically add vertices where necessary.
Why did I start talking about this? Well, simply put, because I don't think its good a idea to keep on working leaving this problem behind us. I have wasted a lot of time on these issues in the past and don't want you to stuggle needlessly with them.

CONCLUSIONS

Well, we are done! I don't know about you but I am really tired! Guess what beautiful surprise I have saved for you in the next lesson? You will no longer see that horrendous cube anymore on the screen. We are going program a model loader for objects in the .3ds format! I'm sure many of you are familiar with 3d Studio and an infinite number of objects can be found in the 3ds format (many of these are spaceships!) on the internet. But for now I will leave you with your great cube =) Have a good time!

SOURCE CODE

To compile and execute this project you need the GLUT libraries that can be found at:www.opengl.org/developers/documentation/glut.html

Download the C/C++ source code and executable in zip format:
Windows version
Linux version (port by Panteleakis Ioannis)
Mingw32 makefile (To use the Linux version on Mingw32 for Windows - by Jose Ortega)
MAC OS version (port by Karakoussis Apostolos)

<< PREVIOUS TUTORIAL **NEXT TUTORIAL>>**

TUTORIAL 5: VECTORS, NORMALS AND OPENGL LIGHTING

Home
Tutorials
The 3d engine
OpenGL and GLUT
Texture mapping
3ds File Loader
Vectors and lighting
Projects
Useful links
Store
About

INTRODUCTION

Vectors and normals... great words! Certainly, most of you would have come across these at school during either physics or geometry. But, you would never have imagined that one day you would have use these notions to create a 3d engine! Anyway, don't worry too much, you won't have to re-open your old dusty books! In this lesson we will cover vectors in detail, and we will program a library to manage them.
The first practical application of vectors deals with the calculations relating to illumination within the world... Many of you could be wondering why we need to bother with this because it's clear that the scene is already uniformly illuminated! Why we need to insert lights then?
Until now we have drawn our virtual world using a uniform, non-realistic illumination. Illuminating objects means drawing them with shiny and shaded parts, which shows more accurately what the object would look like in the real world. Disorder and chaos are the keywords you should keep in mind for your 3d engine... But please, your source code has to be ordered and clear as well =)
With this lesson we will increase the realism of our virtual world.

FUNDAMENTAL NOTIONS ON STRAIGHT LINES, VECTORS and NORMALS

Ok guys, now we have to face up to some theory. Keep in mind that all we are going to study is what is essential to perform the calculations concerning illumination. So again, take your favourite programming drink and get ready!

Do you know what straight lines and segments are? I hope so. Here is a brief explanation of the the fundamentals anyway:
- A **straight line** is an endless line, it doesn't have origin, nor an end, but it has a direction. It can be represented by the simple equation **y=ax+b**. Two straight lines that have the same direction are called **parallel straight lines**. Two straight lines that form an angle of 90 degrees are called **perpendicular straight lines**.

- If we take two points **A** and **B** belonging to a straight line, the space included between them is called a **segment**. To represent a segment we often put a hyphen over the two letters: $\overline{AB}$
- The points **A** and **B** can run in two possible ways: from **A** toward **B,** or from **B** toward **A**. A segment with an assigned orientation is called **oriented segment** and it is represented by putting an arrow over the two extreme

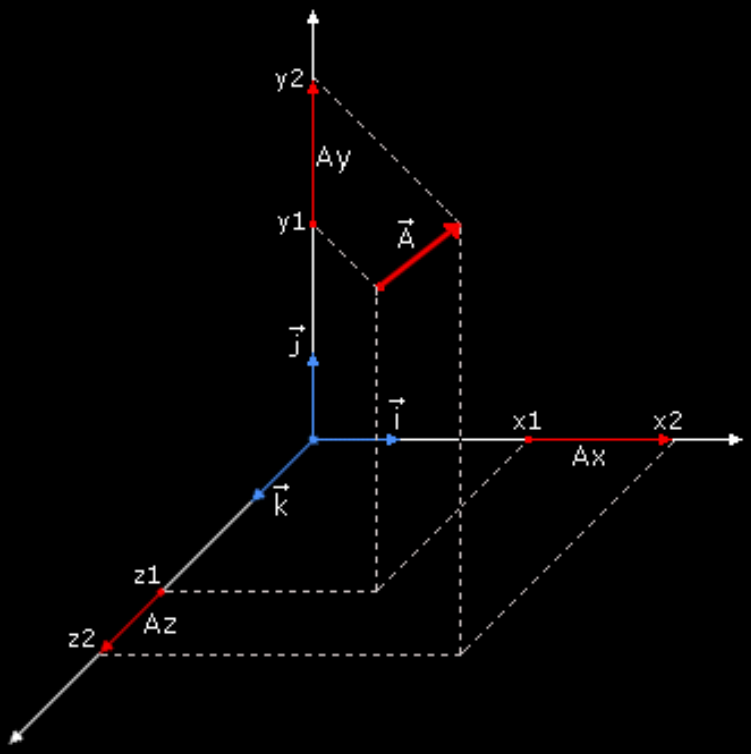
points: $\vec{AB}$.

(1) - Finally here is the definition of a vector: **A vector is a quantity that represents every line segment with the same magnitude and the same direction.**
This means that two vectors are equal if they have the same direction and the same length, regardless of whether they have the same initial points.
That was a geometric definition of a vector, and doesn't really show us the true utility of a vector.
The concept of a vector finds its origins in the field of physics. There are some magnitudes that can be measured simply using a number, these are called scalar quantities (i.e. room temperature or mass) Other magnitudes however, like speed or force, have an oriented direction (direction and angle) as well as a numerical component (that can be only positive). These magnitudes are called vectors.
Graphically a vector is represented as an arrow, this defines the direction. The length of the arrow represents the numerical component, or magnitude.
To identify a vector different notations are used. If we identify the origin of a vector as the point O and the end as the point P, algebraically we can identify the vector using its extreme points: OP.
The notation more commonly used is a capital letter with an arrow over it, example: $\vec{A}$
The numerical component, or magnitude, of a vector (its length) is represented using the absolute value sign: magnitude of $\vec{A}$ = $|\vec{A}|$

The last important definition is the Versor:
A versor is a vector with magnitude equal to 1. Often in our 3d engine we will need to transform some vectors into versors, this operation is called: normalizing a vector.
To identify a versor we will use a lower case letter with an arrow over it, example: $\vec{a}$

REPRESENTATION OF A VECTOR

Now let's begin to organize our data structures to manage these strange vectors! The best thing do when adding a particular functionality to the graphic engine is create a special library, so as to make the code as clean as possible. Therefore we add two blank files: mat_vect.cpp and the relative header mat_vect.h into our development environment. I have decided to insert the prefix "mat" just to identify these as mathematical libraries.
Please have a look at this picture:



We have split our vector $\vec{A}$ into three vectors that are parallel to the x, y and z axis . Now we can represent the vector using this notation:

(2) $\vec{A}$ = [(x2-x1),(y2-y1),(z2-z1)]

And then:

(3) $\vec{A}$ = (Ax,Ay,By)

This is called the **Cartesian Representation** of a vector.
Another way to represent a vector is:

(4) $\vec{A} = Ax\vec{i} + Ay\vec{j} + Az\vec{k}$

where $\vec{i},\vec{j},\vec{k}$ are the axis versors.
And finally here is a little bit of code, using (3) let's create our vector structure:

```
typedef struct
{
    float x,y,z;
} p3d_type, *p3d_ptr_type;
```

We have identified the vector type as a simple 3d point, why? Well, we have simplified the job because by using (1) we can consider all vectors starting

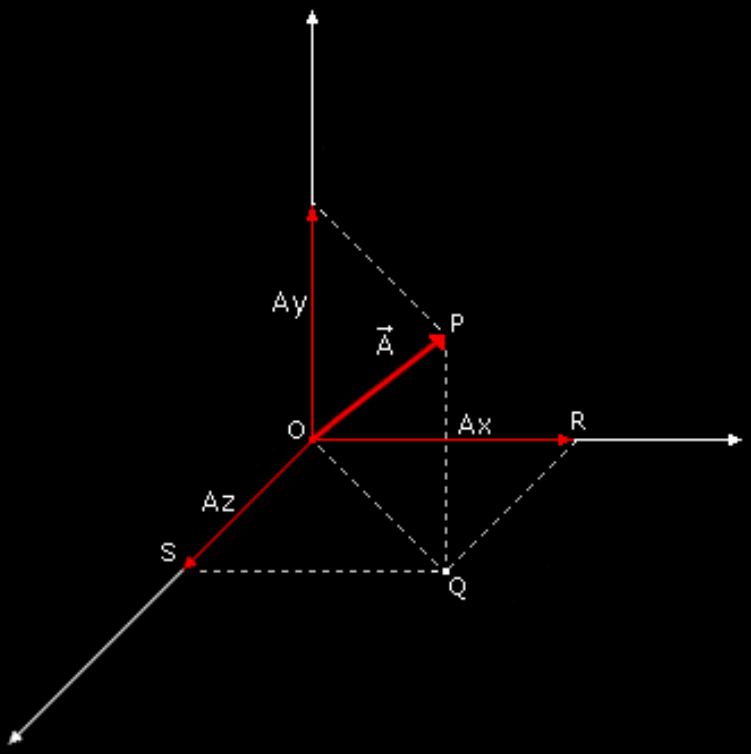
from origin. ;-)

OPERATIONS WITH VECTORS: CREATION, NORMALIZATION AND LENGTH CALCULATION

To create a vector we use formula (2), we write a function with three parameters: the start point, the end point and the final vector. All the parameters are pointers:

```
void VectCreate (p3d_ptr_type p_start, p3d_ptr_type p_end,
p3d_ptr_type p_vector)
{
    p_vector->x = p_end->x - p_start->x;
    p_vector->y = p_end->y - p_start->y;
    p_vector->z = p_end->z - p_start->z;
    VectNormalize(p_vector);
}
```

Don't worry now about the function VectNormalize, that I will explain this in a moment.
To calculate the length of a vector (the magnitude) we have to use Pythagoras' Theorem twice, and don't come here saying that you don't remember that theorem! Here is a brief recap if you have forgotten it though: if we take a right-angle triangle (a triangle with an angle of 90°) the square built on the hypotenuse is equal to the sum of the squares built on the other two sides.
Now its time to use your imagination... have a look at this picture, we have created a vector that starts from the origin of the axis:



To find the length of the vector $\vec{A}$, also called magnitude $|\vec{A}|$=OP, we apply the Pythagoras's Theorem to the triangle OPQ:
OP = Square root of (Ay*Ay + OQ*OQ)
Now Ay is known, to calculate OQ we have to apply Pythagoras's Theorem again, this time using the triangle OQR:
OQ*OQ = (Ax*Ax + Az*Az)

We fill this expression into the first equation and we find the length (magnitude) of the vector:
OP = Square root of (Ax*Ax + Ay*Ay + Az*Az)
And here is the function in C:

```
float VectLenght (p3d_ptr_type p_vector)
{
    return (float)(sqrt(p_vector->x*p_vector->x + p_vector-
>y*p_vector->y + p_vector->z*p_vector->z));
}
```

We have already said that normalizing a vector means transforming its length to 1. To do this we divide all the vector's components: Ax,Ay,Az by its length (magnitude).

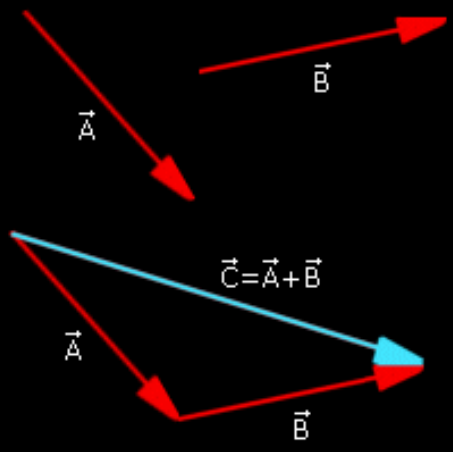
```
void VectNormalize(p3d_ptr_type p_vector)
{
    float l_lenght;

    l_lenght = VectLenght(p_vector);
```

```
    if (l_lenght==0) l_lenght=1;
    p_vector->x /= l_lenght;
    p_vector->y /= l_lenght;
    p_vector->z /= l_lenght;
}
```

OPERATIONS WITH VECTORS: SUM AND DIFFERENCE

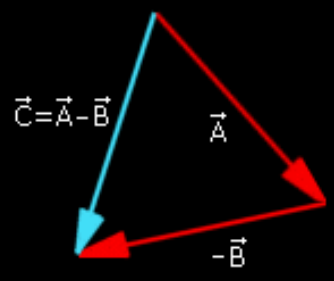
There are 4 fundamental operations that we can do with vectors: sum, difference, scalar product and vector product.
The sum of two vectors $\vec{A}$ and $\vec{B}$ is another vector $\vec{C}$, which is obtained by applying the vertex of $\vec{A}$ the origin of $\vec{B}$ and connecting with a line the origin of $\vec{A}$ to the end of $\vec{B}$.



If we represent the two vectors using the notation (3) then the sum of two vectors can be obtained also using the analytical method:

$\vec{A}+\vec{B}$ = (Ax+Bx, Ay+By, Az+Bz)

The difference of two vectors is a particular case of the sum and it is obtained applying to the vertex of $\vec{A}$ the origin of $-\vec{B}$ and connecting with a line the origin of $\vec{A}$ with the end of $-\vec{B}$.



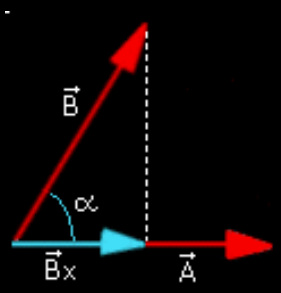
We won't create special functions for the sum and the difference of vectors because there are not useful for now.

OPERATIONS WITH THE VECTORS: SCALAR PRODUCT (OR DOT PRODUCT)

We have two ways to multiply vectors:
-the first one is called scalar product (or dot product) whose result is a simple number.
-the second one is vector product (or cross product) whose result is a vector.

The scalar product is defined as the product of the magnitudes of the vectors times the cosine of the angle between them

$\vec{A}$ dot $\vec{B}$ = $|\vec{A}|$ * $|\vec{B}|$ * cos (α)



This is a geometrical approach that doesn't fit easily in with our source code. We must find a way to express this definition in an analytical approach, so listen:

Using our last definition it is easy to reach these relationships for the axes versors:

(5) $\vec{i}$ dot $\vec{i}$ = $\vec{j}$ dot $\vec{j}$ = $\vec{k}$ dot $\vec{k}$ = 1

(6) $\vec{i}$ dot $\vec{j}$ = $\vec{j}$ dot $\vec{k}$ = $\vec{k}$ dot $\vec{i}$ = 0

Now let's define two vectors using the definition (4):

$\vec{A}$ = Ax$\vec{i}$ +Ay$\vec{j}$ +Az$\vec{k}$
$\vec{B}$ = Bx$\vec{i}$ +By$\vec{j}$ +Bz$\vec{k}$

And calculate the scalar product:

$\vec{A}$ dot $\vec{B}$ = (Ax$\vec{i}$ +Ay$\vec{j}$ +Az$\vec{k}$) dot (Bx$\vec{i}$ +By$\vec{j}$ +Bz$\vec{k}$)

Using the (5) and the (6) we can obtain the scalar product analytic formula.

$\vec{A}$ dot $\vec{B}$ = Ax*Bx + Ay*By + Az*Bz

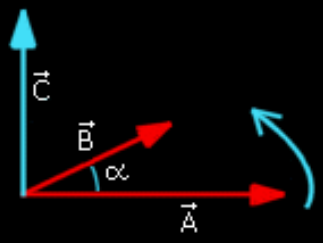
And now let's create the function VectScalarProduct:

```
float VectScalarProduct (p3d_ptr_type p_vector1,p3d_ptr_type
p_vector2)
{
    return (p_vector1->x*p_vector2->x + p_vector1->y*p_vector2->y
+ p_vector1->z*p_vector2->z);
}
```

Wow! It's just crazy!

OPERATIONS WITH THE VECTORS: VECTOR PRODUCT (OR CROSS PRODUCT)

After all these notions you will certainly wonder why we are studying all this theory... but be patient be guys, it will all come together when we get to "illumination" =)
Finally, here is the vector product...
The vector product of two vectors, $\vec{A}$ and $\vec{B}$, is defined as another vector $\vec{C}$ which is calculated in this way:
1-Its magnitude is the product of the magnitudes of vectors $\vec{A}$ and $\vec{B}$ multiplied by the angle between them, $\vec{C} = \vec{A}x\vec{B} = AB\ sen AB$.
2-Its direction is perpendicular to the plane formed by $\vec{A}$ and $\vec{B}$ and oriented so that a right-handed rotation about it carries $\vec{A}$ into $\vec{B}$ through an angle not greater than 180°.



As we have already done for the scalar product we must find the analytical approach to the vector product, so let's write some relationships:

(7) i x i = j x j = k x k = 0

(8) i x j = k j x k=i k x i = j

We now express the two vectors using definition (4):

$\vec{A} = Ax\vec{i} + Ay\vec{j} + Az\vec{k}$

$\vec{B} = Bx\vec{i} + By\vec{j} + Bz\vec{k}$

$\vec{A} \times \vec{B} = (Ax\underline{i} + Ay\underline{j} + Az\underline{k}) \times (Bx\underline{i} + By\underline{j} + Bz\underline{k})$

After a few steps using (7) and (8) we can obtain the expression of the vector product:

A x B = (AyBz-AzBy)i + (AzBx-AxBz)j + (AxBy-AyBx)k

Finally let's create the function VectDotProduct:

```
void VectDotProduct (p3d_ptr_type p_vector1,p3d_ptr_type
p_vector2,p3d_ptr_type p_vector3)
{
    p_vector3->x=(p_vector1->y * p_vector2->z) - (p_vector1->z *
p_vector2->y);
    p_vector3->y=(p_vector1->z * p_vector2->x) - (p_vector1->x *
p_vector2->z);
    p_vector3->z=(p_vector1->x * p_vector2->y) - (p_vector1->y *
p_vector2->x);
}
```

That was like an university lecture, don't you think? Now let's relax a little bit, because we are going to switch on the lights. =D

MAIN STEPS TO ILLUMINATE OUR VIRTUAL WORLD

1-First of all it is necessary to define and to activate at least one light source in the space.
2-For each polygon we need to determine the amount of light that it is able to reflect and to transmit to the eye of our observer.
3-In the rendering phase we paint the polygons according to their illumination value.
We will now analyze these various points in detail.

LIGHT POINTS DEFINITION AND ACTIVATION

To implement the illumination there are two main elements to consider: the light source and the material properties.
OpenGL considers the light constituted by 3 fundamental components: ambient, diffuse and specular.
- The ambient component is the light whose direction is impossible to be determined since it seems to come from all directions. It is mainly light that has been reflected several times. A polygon is always uniformly illuminated by the ambient environment, it doesn't matter what orientation or position it has in the space.
- The diffuse component is the light that originates from one direction, it considers the angle that the polygon has with regards to the light source. The more perpendicular the polygon is to the light ray the brighter it will be. The position of the observer is not used for this and the polygon is always uniformly illuminated.
- The specular component takes into account the degree of inclination of the polygon and also the observer's position. Specular light comes from a direction and is reflected by the polygon according to its inclination.

Return now in the main.c file... For every light point we want to implement it is necessary to specify the various components of it (ambient, diffuse, specular):

```
GLfloat light_ambient[]= { 0.1f, 0.1f, 0.1f, 0.1f };
GLfloat light_diffuse[]= { 1.0f, 1.0f, 1.0f, 0.0f };
GLfloat light_specular[]= { 1.0f, 1.0f, 1.0f, 0.0f };
```

As you can notice each component is composed by 4 float values that represent the RGB and alpha channels.
We have just defined a white light with a little ambient component and a maximum diffuse and specular components. It is exactly what happens in the deep-space!
We have also to specify the position of the light source:

```
GLfloat light_position[]= { 100.0f, 0.0f, -10.0f, 1.0f };
```

We now load the variables just created into OpenGL using the function glLightfv:

```
glLightfv (GL_LIGHT1, GL_AMBIENT, light_ambient);
glLightfv (GL_LIGHT1, GL_DIFFUSE, light_diffuse);
glLightfv (GL_LIGHT1, GL_SPECULAR, light_specular);
glLightfv (GL_LIGHT1, GL_POSITION, light_position);
```

-void glLightfv(GLenum *light*, GLenum *pname*, const GLfloat *params*); assigns values to the parameters of an individual light source. The first argument is a symbolic name for an individual light source, the argument pname specifies the parameters to modify, and can be GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION or others, the third argument is a vector that specifies what value, or values. will be assigned to the parameter.

Now let's activate the light point...

```
glEnable (GL_LIGHT1);
```

...and the OpenGL illumination:

```
glEnable (GL_LIGHTING);
```

DETERMINATION OF THE POLYGONS REFLECTION ABILITY (THE MATERIALS)

The polygons, as well as the lights, have some fundamental properties. They can have different abilities to reflect the light, we call this ability Material.
The OpenGL materials can have 4 fundamental components: ambient, diffuse, specular and emissive.
The first three material properties are exactly equal to the light properties,

while the last property, the emissive component, simulates the light originating from the object, it is very useful because it can be used to simulate bulbs, anyway it doesn't add other light sources and it is not affected by the other light sources.
We now define a material with all of these fundamental components:

```
GLfloat mat_ambient[]= { 0.2f, 0.2f, 0.2f, 0.0f };
GLfloat mat_diffuse[]= { 1.0f, 1.0f, 1.0f, 0.0f };
GLfloat mat_specular[]= { 0.2f, 0.2f, 0.2f, 0.0f };
GLfloat mat_shininess[]= { 1.0f };
```

and let's activate it through the function **glMaterialfv:**

```
glMaterialfv (GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv (GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv (GL_FRONT, GL_SHININESS, mat_shininess);
```

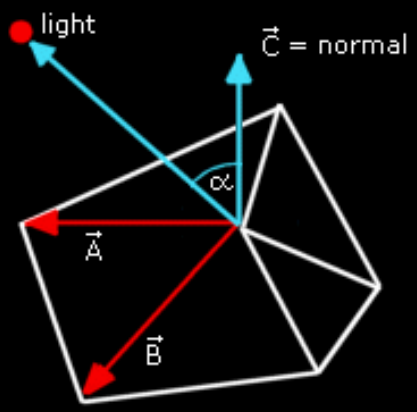
-void glMaterialfv(GLenum *face*, GLenum *pname*, const GLfloat *params*); This assigns values to the material parameters. The argument face can be GL_FRONT, GL_BACK or GL_FRONT_AND_BACK, the second argument specifies the parameters to be modified, and can be GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS, GL_EMISSION, the third argument is a vector that specifies what value or the values will be assigned to the parameter.

The material just created has good diffuse and specular components and a low ambient component, we have simulated the behaviour of a metallic material illuminated by a light source in a dark space... ;-)

DETERMINATION OF THE POLYGONS REFLECTION ABILITY (INCLINATION OF THE POLYGONS IN COMPARISON WITH THE LIGHT POINT)

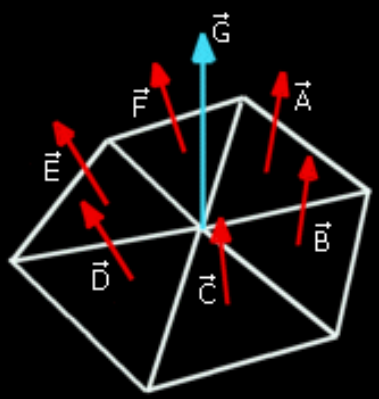
Besides the physical characteristics of the material the most important element in the illumination is the inclination degree of the polygons with regards to the light source. In fact it is clear that the more the polygon plane is perpendicular to the light "ray" more it is illuminated.
To calculate the inclination degree it is necessary, firstly, to calculate the polygon normal vector.
The normal vector of a polygon is nothing but a versor that is orthogonal to the polygon plane. To calculate the normal vector (also called only "normal") of a polygon it is enough to take the vector product of two vectors that are co-planar to the polygon (we can take for example two sides of the polygon) and then normalize the result.
It is necessary also to create another vector whose origin corresponds to the light point and the end to the origin of our normal vector, let's normalize this new vector.
At last, to calculate the illumination degree, we need only to get the scalar product of these two vectors: polygon normal dot light vector!
Nothing easier! The final value will have a value in the range 0.0 - 1.0 and

can therefore be easily used to color our polygon (it can be used for example as a factor).



If we apply this procedure on every polygon, we have illuminated the scene in a particular method called **FLAT SHADING**. Today this method is not used anymore because it doesn't represent objects in a realistic way, every polygon is given a uniform shading and the differences between the colors of adjacent polygons are much too visible.

A more realistic but more expensive technique is the GOURAUD SHADING also called SMOOTH SHADING. The main difference of this technique is that instead of using the polygons, normals are used as the vertices normals. To tell the truth a vertex's normal is a definition that doesn't make much sense! In fact we need a plane to calculate a normal vector! The correct definition could be instead "Average of the normals of the polygons adjacent to the vertex". Then to calculate a vertex's normal it is enough to make the arithmetic average of the normals of the polygons adjacent to the vertex.



Once you have calculated the vertices normals the procedure for the calculation of the illumination coefficient is the same, nevertheless this time every polygon will have three illumination coefficients, not more only one. How can we use three coefficients? Simple, when we draw the polygon we apply the illumination using a linear interpolation of the coefficient values to the extreme points of every scan line. The result is an illumination with gradual variations, much more realistic.

And now I have to confess something: we will only have to calculate the normals of the vertices of our object, OpenGL will perform the rest of the job

drawing the scene in GOURAUD SHADING! So why don't you take a rest now? ;-)

AND HERE IS THE MOST IMPORTANT FUNCTION!

In this paragraph we will program a function able to calculate all the normals of the vertices in a 3d object. We will do this calculation only once during the initialization phase, during the rendering phase we will pass this data to OpenGL.

To make the code easier to understand we need to create a new file object.c and the relative header in which to store all the object's functions.
At first we add (in our object structure, in object.h) an array to contain all the normals of its vertices:

```
vertex_type normal[MAX_VERTICES];
```

We open the file object.c and let's write...

```
void ObjCalcNormals(obj_type_ptr p_object)
{
```

Let's create some "support" variables...

```
   int i;
   p3d_type l_vect1,l_vect2,l_vect3,l_vect_b1,l_vect_b2,l_normal;
   int l_connections_qty[MAX_VERTICES];
```

...and reset all the normals of the vertices

```
   for (i=0; i<p_object->vertices_qty; i++)
   {
      p_object->normal[i].x = 0.0;
      p_object->normal[i].y = 0.0;
      p_object->normal[i].z = 0.0;
      l_connections_qty[i]=0;
   }
```

For each polygon...

```
   for (i=0; i<p_object->polygons_qty; i++)
   {
      l_vect1.x = p_object->vertex[p_object->polygon[i].a].x;
      l_vect1.y = p_object->vertex[p_object->polygon[i].a].y;
      l_vect1.z = p_object->vertex[p_object->polygon[i].a].z;
      l_vect2.x = p_object->vertex[p_object->polygon[i].b].x;
      l_vect2.y = p_object->vertex[p_object->polygon[i].b].y;
      l_vect2.z = p_object->vertex[p_object->polygon[i].b].z;
      l_vect3.x = p_object->vertex[p_object->polygon[i].c].x;
      l_vect3.y = p_object->vertex[p_object->polygon[i].c].y;
      l_vect3.z = p_object->vertex[p_object->polygon[i].c].z;
```

We create two co-planar vectors using two sides of the polygon:

```
      VectCreate (&l_vect1, &l_vect2, &l_vect_b1);
      VectCreate (&l_vect1, &l_vect3, &l_vect_b2);
```

calculate the vector product between these vectors:

```
      VectDotProduct (&l_vect_b1, &l_vect_b2, &l_normal);
```

and normalize the resulting vector, this is the polygon normal:

```
      VectNormalize (&l_normal);
```

For each vertex shared by this polygon we increase the number of connections...

```
      l_connections_qty[p_object->polygon[i].a]+=1;
      l_connections_qty[p_object->polygon[i].b]+=1;
      l_connections_qty[p_object->polygon[i].c]+=1;
```

... and add the polygon normal

```
      p_object->normal[p_object->polygon[i].a].x+=l_normal.x;
      p_object->normal[p_object->polygon[i].a].y+=l_normal.y;
      p_object->normal[p_object->polygon[i].a].z+=l_normal.z;
      p_object->normal[p_object->polygon[i].b].x+=l_normal.x;
      p_object->normal[p_object->polygon[i].b].y+=l_normal.y;
      p_object->normal[p_object->polygon[i].b].z+=l_normal.z;
      p_object->normal[p_object->polygon[i].c].x+=l_normal.x;
      p_object->normal[p_object->polygon[i].c].y+=l_normal.y;
      p_object->normal[p_object->polygon[i].c].z+=l_normal.z;
   }
```

Now, let's average the polygons normals to obtain the vertices normals!

```
  for (i=0; i<p_object->vertices_qty; i++)
   {
      if (l_connections_qty[i]>0)
      {
         p_object->normal[i].x /= l_connections_qty[i];
         p_object->normal[i].y /= l_connections_qty[i];
         p_object->normal[i].z /= l_connections_qty[i];
      }
   }
}
```

In the file object.c we insert also another function that allows to initialize every aspect of the object:

```
char ObjLoad(char *p_object_name, char *p_texture_name)
{
   if (Load3DS (&object,p_object_name)==0) return(0);
```

```
   object.id_texture=LoadBMP(p_texture_name);
   ObjCalcNormals(&object);
   return (1);
}
```

AND NOW LET'S DRAW THE ILLUMINATED WORLD

The last change to make concerns the drawing routine. We must send to OpenGL the normals of the object. The OpenGL function that does this is glNormal3f.

-void glNormal3f(GLfloat *nx*, GLfloat *ny*, GLfloat *nz*); specifies the three coordinates x,y,z of the current normal.

Here is the new drawing routine:

```
glBegin(GL_TRIANGLES);
   for (j=0;j<object.polygons_qty;j++)
   {
      //----------------- FIRST VERTEX -----------------
      //Normal coordinates of the first vertex
      glNormal3f( object.normal[ object.polygon[j].a ].x,
                  object.normal[ object.polygon[j].a ].y,
                  object.normal[ object.polygon[j].a ].z);
      // Texture coordinates of the first vertex
      glTexCoord2f( object.mapcoord[ object.polygon[j].a ].u,
                    object.mapcoord[ object.polygon[j].a ].v);
      // Coordinates of the first vertex
      glVertex3f( object.vertex[ object.polygon[j].a ].x,
                  object.vertex[ object.polygon[j].a ].y,
                  object.vertex[ object.polygon[j].a ].z);

      //----------------- SECOND VERTEX -----------------
      //Normal coordinates of the second vertex
      glNormal3f( object.normal[ object.polygon[j].b ].x,
                  object.normal[ object.polygon[j].b ].y,
                  object.normal[ object.polygon[j].b ].z);
      // Texture coordinates of the second vertex
      glTexCoord2f( object.mapcoord[ object.polygon[j].b ].u,
                    object.mapcoord[ object.polygon[j].b ].v);
      // Coordinates of the second vertex
      glVertex3f( object.vertex[ object.polygon[j].b ].x,
                  object.vertex[ object.polygon[j].b ].y,
                  object.vertex[ object.polygon[j].b ].z);

      //----------------- THIRD VERTEX -----------------
      //Normal coordinates of the third vertex
      glNormal3f( object.normal[ object.polygon[j].c ].x,
                  object.normal[ object.polygon[j].c ].y,
                  object.normal[ object.polygon[j].c ].z);
      // Texture coordinates of the third vertex
      glTexCoord2f( object.mapcoord[ object.polygon[j].c ].u,
```

```
                        object.mapcoord[ object.polygon[j].c ].v);
        // Coordinates of the Third vertex
        glVertex3f( object.vertex[ object.polygon[j].c ].x,
                    object.vertex[ object.polygon[j].c ].y,
                    object.vertex[ object.polygon[j].c ].z);
    }
glEnd();
```

CONCLUSIONS

Finally we have reached the end, and we have survived, hopefully! This was a very technical tutorial, but we have learned some very important theory. The notions studied here will certainly be useful for the next tutorial in which we will introduce matrices. Matrices will allow us to manage the position and the orientation of the objects. Are you tired of the random rotation of your spaceship? ;-)

SOURCE CODE

To compile and execute this project you need the GLUT libraries that can be found at:www.opengl.org/developers/documentation/glut.html

Download the C/C++ source code and executable in zip format: Windows version

<< PREVIOUS TUTORIAL **NEXT TUTORIAL>>**

PROJECTS

Home

Tutorials

- The 3d engine
- OpenGL and GLUT
- Texture mapping
- 3ds File Loader
- Vectors and lighting

Projects

Useful links

Store

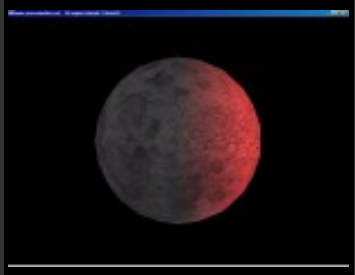
About

Subscribe to know about the latest updates

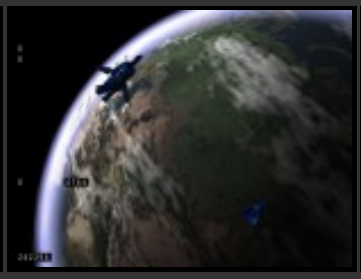
Send E-mail

Here you can find some projects done by me or other people. If you have a project in topic with this site or a space simulator in development or already done, or simply you have done something using the tutorials I can insert it on this section. Please send me the description and some screenshots with size 150x113 pixels.

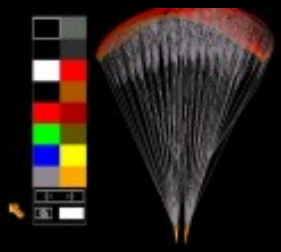
Tutorials based projects

Title	Screenshot	Description	Platform	Author
Planet shading		This project is based on tutorial 4 (3ds loader). A planet mesh with a texture of the moon are loaded. The author has developed on its own also the lighting with a red diffuse colour and smooth (gouraud) shading. Works for Windows and Linux!	Windows and Linux (Source code and exe)	Zeeshan Ali

Projects in development

Title	Screenshot	Description	Platform	Author
Project Zetadeck		Zetadeck is an innovative space simulator currently in development.	Windows and maybe Linux	Damiano Vitulli

Project Monroe		The Viewscreen Peer program is being developed as part of an ongoing project to build a full-scale simulation of the model E-10. The E-10 is a realistic but futuristic space-faring vessel.	Windows	Jason Reskin

Other projects				
Title	**Screenshot**	**Description**	**Platform**	**Author**
Wing Painter (Amiga) - Only Italian		Wing Painter is a program I developed for the Amiga 500 when I was 16 years old. It allows you to plan paragliders and traction kites in a very easy approach. Of course you need an Amiga emulator and a little bit of patience because this program is only in Italian language.	Amiga Emulators	Damiano Vitulli

USEFUL LINKS

Home

Tutorials

- The 3d engine
- OpenGL and GLUT
- Texture mapping
- 3ds File Loader
- Vectors and lighting

Projects

Useful links

Store

About

Subscribe to know about the latest updates

Send E-mail

For banners exchange please send an e-mail to info[at]spacesimulator.net	
OpenGL	
opengl.org	The official OpenGL web site, full of useful information, faqs, libraries.
Game Programming	
Gamasutra GameDev.net flipCode	Game developing on-line magazines.
NeHe	Many tutorials, resources, portings and links for OpenGL developers!!
PC GPE	PC Game Programmer's Encyclopedia: many useful information (Algorithms, file formats, MS DOS graphic programming and so on).
Wotsit	This site contains file format information on hundreds of different file types (also 3DS and BMP that we are using in the tutorials).
3DS editors	
Anim8or AC3D Milkshape	Some shareware and free powerful 3d editors to make 3ds models!
3DS models	
3d Cafe free models	Many free 3ds models to download.
Flash Fire Designs	High quality low cost models, custom 3d modeling, model conversion, texture map creation, graphics design.
Textures (Remember to convert the format in .bmp for the tutorial 3)	
JPL planetary image maps	Textures of all the planets of our solar system.
Planetary image maps	
Blue Marble	High resolution Earth maps.

Björn Jónsson Planetary image maps	Very good images of some solar system planets.
3d Cafe free textures	Many texture to download.
Music	
MAZ Sound Tools	Many programs to do music for games (trackers and sampler editors)
Mad Tracker	Really good Tracker. It is really simple to use with a good interface.
Other interesting links	
JPL Solar System Simulator	Set your point of view, target, date and time and see what you want in the solar system.
JPL Solar System Dynamics	Many useful information regarding orbital and physical parameters, ephemerides and so on.
3D Glasses	3D Glasses - American Paper Optics, Paper 3D Glasses, 3-D Glasses Manufacturer
Contributors' links (e-mail me if you want to be inserted in this section)	
Sysmedia	My real job! The new generation of microcontrollers and industrial PCs.
A.I. and 3D Mountains Simulator	ALife programs for download (also available codes in C++ / Pascal) and 3D Polish Tatra Mountains simulator. Soon available 3D artificial life simulator.

ABOUT

Home

Tutorials

- The 3d engine
- OpenGL and GLUT
- Texture mapping
- 3ds File Loader
- Vectors and lighting

Projects

Useful links

Store

About

Subscribe to know about the latest updates

Send E-mail

Some data of the author	
Name	Damiano Vitulli
Date of Birth	2 June 1977
Place of Birth	Rome, Italy
Citizenship	Italian
Languages	Italian, English
Telephone N.	+393200633150
E-Mail	info [at] spacesimulator [dot] net
Home Page	spacesimulator.net
Curriculum vitae (italian)	

Me encircled by my autobuild paraglider (1996)

Hi, my name is Damiano Vitulli, I thank you for having visited my web site and I hope that it is to your liking.
Currently my real job is the software and hardware development for data acquisition, data processing and remote control systems. I work mainly in C/C ++ and Visual Basic. I also develop managerial and graphic applications.
Since 1999 I started the development of an innovative space simulator (see Zetadeck) to which nevertheless I don't succeed in devoting enough time.
Then, because of my passion for the space simulations and in the hope of finding financings for Zetadeck, I have realized this web site. Spacesimulator.net was born as a portal for the space simulations, but the desire to create something great was so strong that I decided to write some tutorials on how to realize a 3d engine. The many compliments and the success I have found among the numerous visitors have convinced me to continue my work, despite my short free time.
Currently spacesimulator.net is looking for financings to continue the publication of the tutorials and the development of Zetadeck. Possible proposals and applications' requests can be sent to info[at]spacesimulator.net

For legal information read the disclaimer.

I thank so much all those people that these years have helped me in the publication of the tutorials and the web site, here is the list:

Name	Profession	Contact	Web Site	Section
Alexandru Afrasinei		alex-toranaga [at] home [dot] ro		SDL porting tutorial: 2,4
Apostolos Karakoussis		ktolis [at] ccf.auth [dot] gr		MAC OS porting tutorial: 2,3
Ciro Durán		cirojota [at] hotmail [dot] com		Spanish translation Tutorial 1
François Devic		francois.devic [at] free [dot] fr		French translation Tutorial 2
Frank Kramer		fk173 [at] compuserve [dot] de		German translation Tutorial 1
Ioannis Panteleakis	Student/ Technical Assistant	pioann [at] csd. auth [dot] gr		Linux porting tutorial: 2,3,4
Jonathan Dornan		jonathan_dornan [at] hotmail [dot] com		English corrections: tutorial 5
Jose Ortega	Web Designer	joorce [at] hotmail [dot] com		Mingw32 makefile tutorial: 4
Maarten Heidenrath				Dutch translation Tutorial 1
Martin Williams				MAC OS porting tutorial: 4
Rene Reiter	Student	renereiter [at] hotmail [dot] com	http:// eyeball. waidi.net	3DS model tutorial: 4
Robert Napolitano	Programmer	enapolitano1 [at] nyc.rr [dot] com		English corrections: tutorials index, tutorial 1,2,3,4
Steve Bruce		steve_bruce [at] lineone [dot] net		Printer friendly tutorial: 4
Sylvain Carding		sylvain_cardin [at] yahoo [dot] com		French translation Tutorial 1

Zeeshan Ali Khattak	Software Developer	zak [at] yahoo [dot] com		Tutorials based project: Moonshading Linux porting tutorial: 5

Please contact me for any change, the contacts are covered to prevent spam, I have no responsibility for the data inserted.

```
/*
 * ---------------- www.spacesimulator.net --------------
 *   ---- Space simulators and 3d engine tutorials ----
 *
 * Author: Damiano Vitulli <info@spacesimulator.net>
 *
 * ALL RIGHTS RESERVED
 *
 *
 * Tutorial 1: 3d engine object definition
 *
 *
 * You cannot see anything as output because we need the drawing libraries
 *
 */



#include <windows.h>

#define MAX_VERTICES 2000 // Max number of vertices (for each object)
#define MAX_POLYGONS 2000 // Max number of polygons (for each object)



/*************************************************************
 *
 * TYPES DECLARATION
 *
 *************************************************************/

/*** Our vertex type ***/
typedef struct{
   float x,y,z;
}vertex_type;

/*** The polygon (triangle), 3 numbers that aim 3 vertices ***/
typedef struct{
   int a,b,c;
}polygon_type;

/*** The object type ***/
```

```
typedef struct {
   vertex_type vertex[MAX_VERTICES];
   polygon_type polygon[MAX_POLYGONS];
} obj_type, *obj_type_ptr;



/*************************************************************
 *
 * VARIABLES DECLARATION
 *
 *************************************************************/

/*** And, finally our first object! ***/
obj_type cube =
{
   {
      -10, -10, 10,   // vertex v0
      10,  -10, 10,   // vertex v1
      10,  -10, -10,  // vertex v2
      -10, -10, -10,  // vertex v3
      -10, 10,  10,   // vertex v4
      10,  10,  10,   // vertex v5
      10,  10,  -10,  // vertex v6
      -10, 10,  -10   // vertex v7
   },
   {
      0, 1, 4,  // polygon v0,v1,v4
      1, 5, 4,  // polygon v1,v5,v4
      1, 2, 5,  // polygon v1,v2,v5
      2, 6, 5,  // polygon v2,v6,v5
      2, 3, 6,  // polygon v2,v3,v6
      3, 7, 6,  // polygon v3,v7,v6
      3, 0, 7,  // polygon v3,v0,v7
      0, 4, 7,  // polygon v0,v4,v7
      4, 5, 7,  // polygon v4,v5,v7
      5, 6, 7,  // polygon v5,v6,v7
      3, 2, 0,  // polygon v3,v2,v0
      2, 1, 0,  // polygon v2,v1,v0
   }
};
```

```
/*****************************************************************
 *
 * The main routine
 *
 *****************************************************************/

int main(int argc, char **argv)
{
   return(1);
}
```

PROJECT ZETADECK

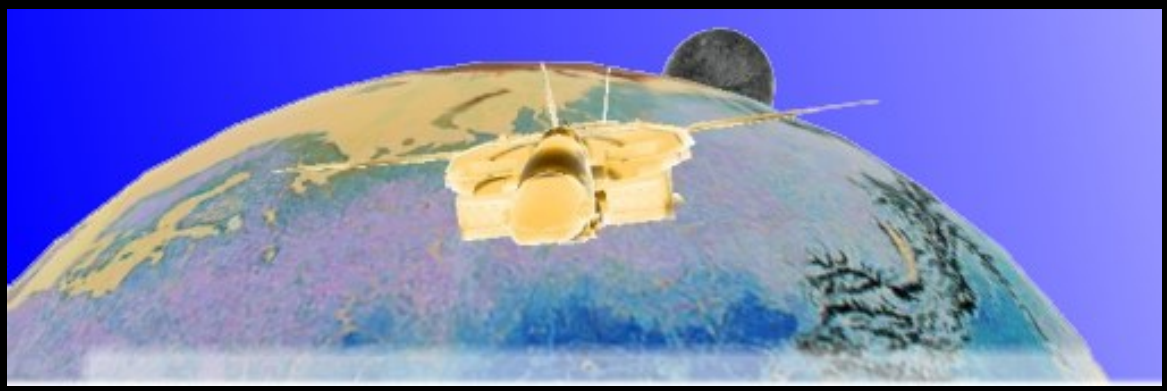
Home
Tutorials
The 3d engine
OpenGL and GLUT
Texture mapping
3ds File Loader
Vectors and lighting
Projects
Useful links
Store
About

Subscribe to know about the latest updates

Send E-mail

What is Zetadeck?
This space sim, whose 3d engine is currently in development, wants to cover the lacks that the other space simulators have.

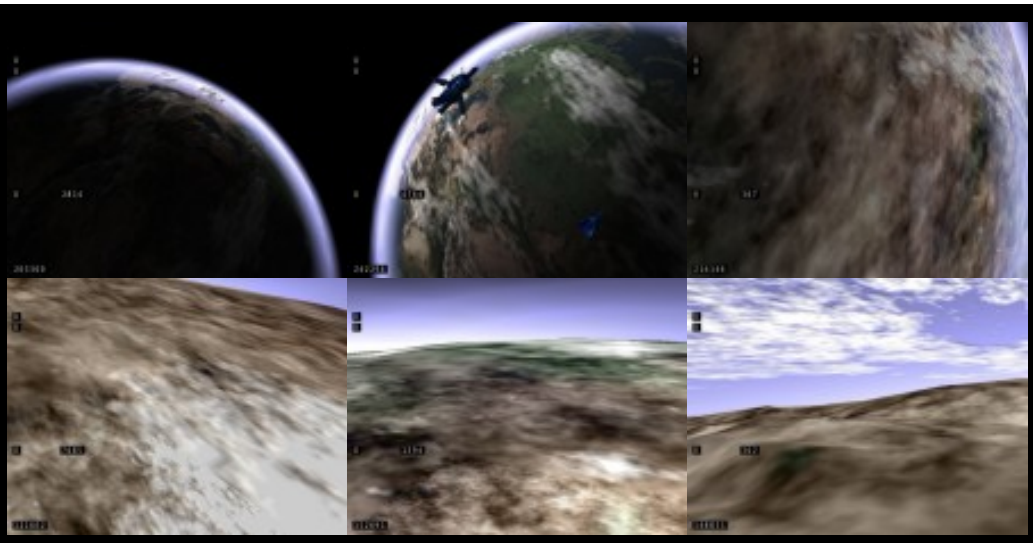
When I began to develop this space simulator I had the following objectives:
1-A particular use of the coordinate system, so that it can manage a whole universe of thousand million of km with a perfect resolution
2-Planets and stars shown big as they are in reality.
3-The possibility to explore planets and land on them. Thanks to the news features of the graphic engine it will be possible to show realistic landscapes full of planetary elements with a resolution that increases according to the proximity... Have you ever dreamt of a space simulator where you can see the life and the nature evolving in a realistic approach, see the creation of intelligent creatures?
4-The engine will also be able to manage close environments so that you can explore the interior of buildings or the inner decks of bigger spaceships.
5-Some ships will have the pecularity to be multi-transformable in robots so to increase the realism and the fun while exploring or fighting with other mechas.



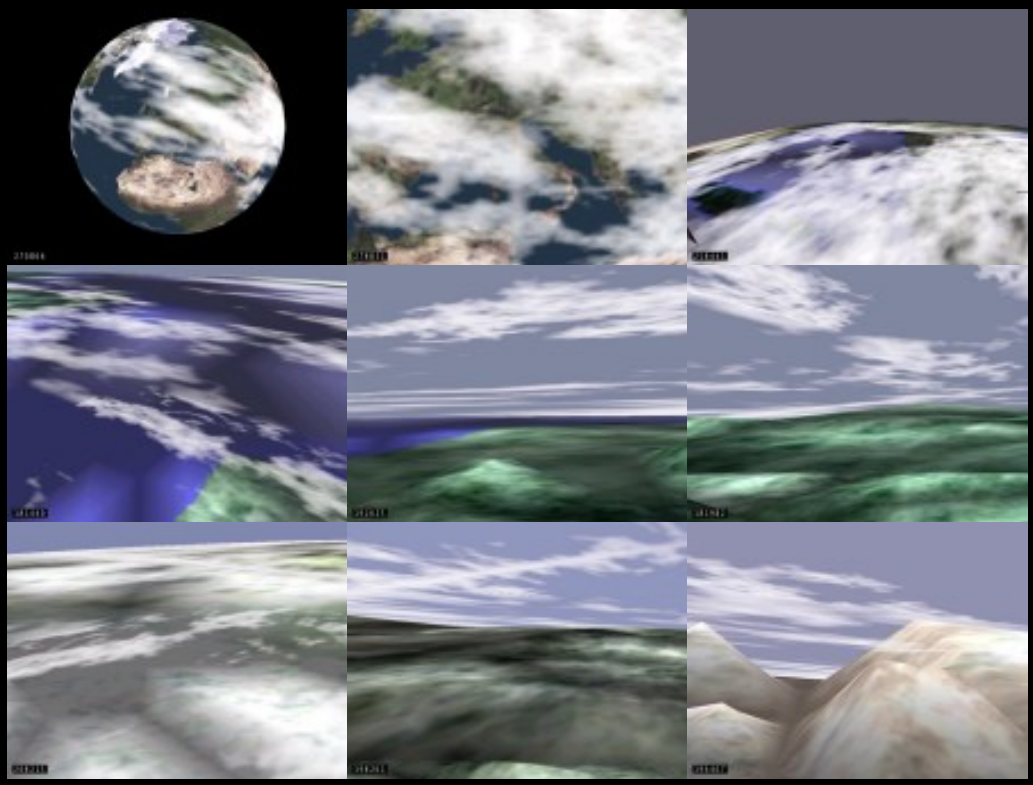
With Zetadeck the space gamers' dream comes true...

Here you can find some screenshots of Zetadeck. Please have a look at the graphic improvement in comparison to my old images. At the moment I am working to include volumetric fog, 3d clouds and Bézier "smoothed" surfaces.

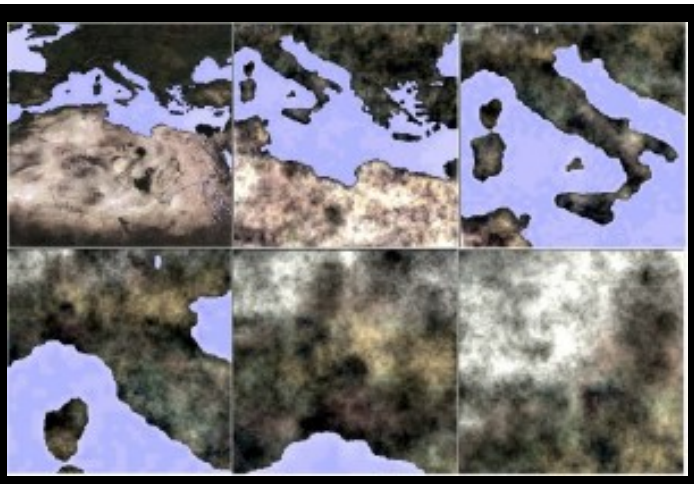
>>2002AUG2



>>2001OCT1



>>2001JUN23

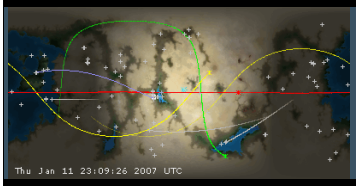


RISE

THE OLD KINGDOMS OF ONLINE GAMING ARE FALLING. TIME TO RISE!

Experience over 100,000,000 km of seamless reality in an accurately simulated extra-solar system. Use ground, air, and space vehicles and structures to discover and develop sprawling urban cities, rain swept valleys, vast uncharted deserts, and glittering alien seas. Freely climb to space to take part in physics-accurate orbital maneuvers as the economic and social frontiers expand to interstellar commerce and travel. Colonists arrive and thrive in the world that YOU are helping to establish. Ready to RISE?

CORE SITE
FORUMS
CITIZENS
UNIVERSE
SUPPORT
LINKS



Games for Windows | Get Adobe Reader | ATi | RP | Photosensitive Seizure Warning | d20 system