This is the fourth PSP tutorial

Introduction

Hi everybody,

This tutorial is about the three main matrices used in a game. For this and all tutorials I assume that you have some C++ experience and that the basics are clear to you. This tutorial will mainly explain the three matrices and show some code but there will be no example code or source. You should use the code from the previous tutorial 2: rendering a triangle. That code already holds a basic example of the use of matrices.

Matrices

Matrices are a very complex subject in mathematics. This tutorial will not explain the workings of a matrix but will explain how to use them and what they do to your game. Matrices on the PSP are generally 4X4 matrices. Think of them as a 2D grid of numbers with 4 numbers along the top and 4 down. Matrices are used for example to store a lot of world calculation in only one matrix which is used very low level by the PSP to alter a lot of things. We call this changes "transformations".

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrices are used in the following ways:

- Translate, rotating and scale objects in space.
- Setup projection and view
- and a lot more...

In this tutorial we will look into the projection, view and world matrices. Translating, rotating and scaling of the view and world matrices will also be discussed in more detail in this tutorial.

But first let us take a look at how a matrix is created. Here is the code:

```
ScePspFMatrix4 matrix;
```

But this is not enough. With normal values like a float or an int we declare them like that but with matrices it does not work that well. With matrices we also need to make it an identity matrix.

```
gumLoadIdentity(&matrix;);
```

Now we can use the matrix. For the rest of the tutorial we will use this but in a strange way. The PSP SDK for some odd reason needs variables without them being called. Programmers who do not know this will not get anything to work. What I am talking about is that the function sceGumMatrixMode, which will be explained in the next paragraph, needs the matrix variable but it is not documented. I am not even sure if it is correct but without the variables the code simply does not work. So when we will use the GU_PROJECTION we also have to declare a ScePspFMatrix4 called projection. The same goes for view and model with that names. Why this is? I do not know but I do know that when I do not declare those variables it does not work. If someone knows why this is, please let me know!

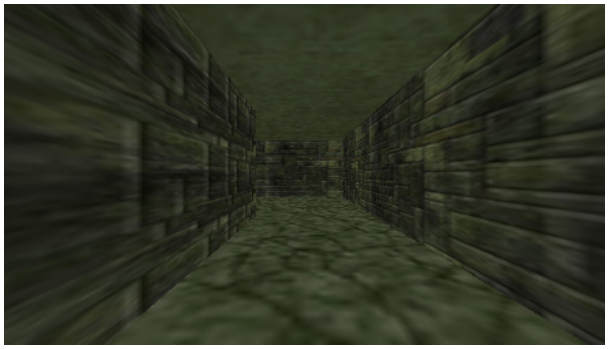
The Projection matrix

The projection matrix pretty much tells the renderer how it sees the 3D world and how it should render it. Lets take a look at the code to set up the projection matrix:

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

First we set the GU_PROJECTION mode. Matrix function calls (using gum) will affect the projection after that call. Please remember that you need to declare the projection matrix variable also. The second line makes the projection matrix an identity matrix. (Also see identity matrix) Now the last lines sets up a perspective matrix. Lets take a look at the first parameter. It specifies the angle of the view. Lets take a look at some example screenshots (from my game Elementals):

The projection with view angle of 70

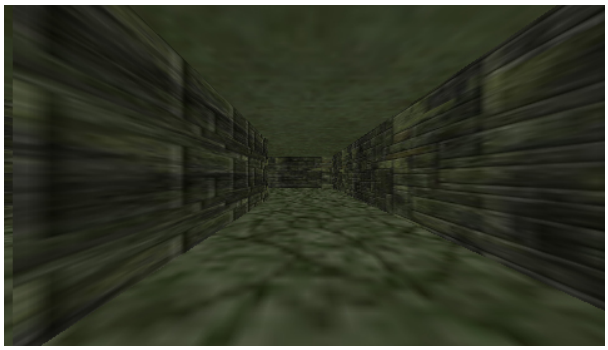


The projection with view angle of 100

You can use this for example to create nice ingame effects. Please remember that you set this matrix when you load the game, so when you will use it for a special effect remember to call the functions again but only then. Using it every frame will have an impact on your framerate.

The second parameter is the aspect ratio. Below is the effect.

Aspect ratio of 16/9



Aspects ratio 4/3

I always set it on 16/9 because I do not need a different aspect ratio.

The last two parameters specify the near and the far plane. The near and far plane are used to create a boundary to your projection. Objects behind the far plane will not be rendered and objects in front of the near plane will also not be rendered. The PSP has some strange behaviour if you do not understand how it works. If you make the near buffer for example too small the accuracy of the Z buffer will be less precise. For example:

2.1 nearplane

0.4 nearplane

As you can see some of the triangles are not even rendered. This is a very common problem. Setting the near plane further is the easiest solution but in some cases not the best and most usable solution. Also a common problem is clipping. The PSP only clips the near plane and not the sides, bottom and top plane of the projection. This causes large triangles to disappear at times you really do not want it. More on this in a future tutorial. That was the projection matrix on to the view and model matrix.

The View matrix

The view matrix is not so difficult. You can think of the viewmatrix as a camera. The camera has a position somewhere in space and a orientation. Everything you see on your PSP screen is rendered as viewed through that camera. What kind of lens and stuff is specified by the projection matrix. So what can we do with the view matrix? well for example we can translate and rotate it. Lets look at some code:

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

Well as before, we tell that we want to use the view, then we identity it.

```
// using the lookat function
ScePspFVector3 pos = {0,0,500};
ScePspFVector3 view = {0,0,0};
ScePspFVector3 up = {0,1,0};
sceGumLookAt(&pos;,&view;,&up;);
```

Here we use the look at function. We can just translate and rotate the view matrix but sometimes it is more useful to use the lookat function. The function takes three parameters. The first is the position of the camera, the second is the position we look at and the last is the up vector so that the camera knows what way up is. In this case we look at the origin from position 0,0,500 where the positive y is up. This function can be useful for example if you want to follow the player or a rocket.
The ScePspFVector3 is a data structure which will be used a lot during your PSP programming adventures :). Look at tutorial m6 to see a lot more datastructures that the PSP uses and explaination about them.

Translation

If we want the camera (or an object) to move in space we need to translate it. How do we that? We can give every vertex a new position but that is not really the fastest and best way of doing it. Using matrices is the better way. So how do we translate? Lets take a look at an example:

```
// using the sceGumTranslate function
ScePspFVector3 pos = {0,0,500};
sceGumTranslate(&pos;);
```

Now the view will be at position (0,0,500). When you use this for the Model matrix all vertices will be rendered around that position the same way as it would around the origin. However just translating is not very nice in a game. We also need rotation.

Rotation

```
// using the sceGumRotateY function
float angle = 1.5f;
sceGumRotateY(angle);
```

Here we rotate around the Y axis. Imagine the angle value is incrementing in small steps. We can give the illusion of spinning this way. Please not that when we use a model instead of a view(see next paragraph) that the model is modeled around the origin otherwise it will not spin but rotate around the origin because it always takes the origin as centerpoint of the rotation. If you have modeled it around the origin it will spin around it.

This rotate function is not the only rotation function. Here is a list of other functions you can use:

- **sceGumRotateX(float)**, same as the example only now rotation around the X axis.
- **sceGumRotateZ(float)**, same as the example only now rotation around the Z axis.
- **sceGumRotateXYZ(ScePspFVector3*)**, the combination of the three above functions. Rotation is applied in the order x, y, z. So first the rotation is done around the X axis and so forth.
- **sceGumRotateZYX(ScePspFVector3*)**, the combination of the three above functions. Rotation is applied in the order z, y, x.

Scaling will be explained in the next paragraph. On to the Model matrix.

The Model matrix

The model matrix is similar to the view matrix. Where the view matrix is used to transform the camera, the model matrix is used to transform models (objects in space). The matrix is used to transform the vertices of a model. This can be a simple triangle to a 1000 count poly object. As we could see in tutorial 2: rendering a triangle, the triangle was rotated around the camera. The model matrix was responsible for this behaviour. When using this matrix please remember that every alteration is done around the origin. (0,0,0)

```
// setting the model matrix
sceGumMatrixMode(GU_MODEL);
sceGumLoadIdentity();
```

Now the model matrix can be altered in the same way as the view matrix. You can translate and rotate just like we can with the view or we can scale it. Scaling is not so difficult and it can be used for all sorts of effects or corrections. Image a model of a human you use and a building and the building is smaller than the human. To correct we can scale the human down or the building up. Here is how the code looks:

```
// scaling
ScePspFVector3 scale = {2,2,2};
sceGumScale(&scale;);
```

Here we have scaled the object we will render using this matrix by 2 in every dimension. You can pass whatever values you like and they do not have to be the same. So the x can be 5 while the y is 2. Scaling with the value 0 is not recommended :).

When you use the world matrix, everything after the set up of the matrix will be rendered with those transformations. The beautiful part however is that you can reuse the matrix and then render something else or the same thing again in the same render loop only with different transformations. This is for example very handy if you have multiple objects that have different positions.

Lets take a look at a game example. We have a car which is at position (10, 0, 10) and we have a tree that is at position (20, 0, 20). The car is moving but the three is not, how can we solve this with only using one world matrix for those two objects? The only way is to

transform the coordinates of the vertices of the car but we do not want that. The better way of doing this is to create the objects around the origin (so we can also rotate them around an axis if we want) and then use a world matrix for the car and a world matrix for the tree. This way each object is created at (0,0,0) but can be translate (or rotated or scaled) to any position we want independantly of each other using the world matrix and reset it for the other object.

Other matrix functions

The functions mentioned above are not the only matrix functions. Here is a list of other functions and what they do:

- **sceGumMultMatrix(ScePspFMatrix4*)**, This function multiplies the passed matrix with the currently used matrix.
- **sceGumFullInverse()**, This function inverts a matrix. (MathWorld on inverted matrices)
- **sceGumFastInverse()**, This function does the same as the one above but it is only for orthogonal matrices. (see also: Orthogonal matrices.)
- **sceGumPushMatrix()**, This functions pushes the current matrix on the stack. At this moment (In my version of the SDK) there is a hardcoded limit of 32 entries per type. You can change this in the gumInternal.c and recompile it if you need.
- **sceGumPopMatrix()**, This function pops matrix from the stack.

So that's it for this tutorial. If you understand it correctly you should now be able to use the view and transform it, transform game objects (models) and set up a projection matrix. With this the world of 3D really opens up! On to the next tutorial.