Lighting tutorial

Summary

This is tutorial shows how to use shaders for lighting with auto-generated normal maps.
NB: If you want to see how to use orx while using C++ for your game, please refer to the localization tutorial.

As we are **NOT** using the default executable anymore for this tutorial, the tutorial code will be directly built into the executable and not into an external library.

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy, animations, cameras & viewports, sounds & musics, FXs, physics, scrolling. C++ localization and spawner & shader.

This tutorial shows how to generate normal maps and use shaders for pixel-based lighting effects.
It's only one of the many possibilities of lighting you can achieve with shaders.

The code simply deals with an array of lights and allow to change some of their properties such as position or radius.
The whole object lighting is done in the fragment shader defined in 12_Lighting.ini.

For performance sake, the normap maps are computed for each object's texture the first time the object is loaded.
This computation is made on the CPU but it could have been done on the GPU using viewports that would have textures as render target, instead of the screen.
Then all the objects would be rendered separately once with a shader which would only compute the normal maps.
This technique would improve “loading/init” performances but requires more code to be written.

A more efficient way would be to batch the normal map creation: loading all the texture at once and creating the associated normal maps in one pass.
We chose to do it on objects creations instead so as to keep this tutorial modular and allow new objects to be added in config by users without any additional knowledge on how the textures will be processed at runtime by the code.

Please note that the lighting shader is a very basic one, far from any realistic lighting, and has been kept simple so as to provide a good base for newcomers.

Details

Let's begin with a quick look to our `main` function.

```
int main(int argc, char **argv)
{
  orx_Execute(argc, argv, Init, Run, Exit);
```

```
  return EXIT_SUCCESS;
}
```

Nothing new here, we only execute orx using the helper `orx_Execute` function, providing three callbacks: `Init()`, `Run()` and `Exit()`.

Let's now have a glimpse to our `Init()` function.

```
orxSTATUS orxFASTCALL Init()
{
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  orxEvent_AddHandler(orxEVENT_TYPE_SHADER, EventHandler);
  orxEvent_AddHandler(orxEVENT_TYPE_TEXTURE, EventHandler);

  pstTextureTable = orxHashTable_Create(16, orxHASHTABLE_KU32_FLAG_NONE,
orxMEMORY_TYPE_MAIN);
  pstViewport = orxViewport_CreateFromConfig("Viewport");
  pstScene = orxObject_CreateFromConfig("Scene");

  ClearLights();

  return eResult;
}
```

`EventHandler()` will listen for shader and object events.
There we'll populate shader parameters at runtime and create normal maps for new created object if the corresponding normal map isn't already available.

A hashtable is then created for storing the normal maps.
Our traditional viewport/scene couple is also created and that's all we need to create.

`ClearLights()` is a very straightforward function that will clear all our light data.

As mentioned above, `EventHandler()` will listen to both shader and object events. Let's have a look at it:

```
orxSTATUS orxFASTCALL EventHandler(const orxEVENT *_pstEvent)
{
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  if((_pstEvent->eType == orxEVENT_TYPE_SHADER) && (_pstEvent->eID ==
orxSHADER_EVENT_SET_PARAM))
  {
    orxSHADER_EVENT_PAYLOAD *pstPayload;
    pstPayload = (orxSHADER_EVENT_PAYLOAD *)_pstEvent->pstPayload;
```

```
    if(pstPayload->s32ParamIndex <= (orxS32)u32LightIndex)
    {
      if(!orxString_Compare(pstPayload->zParamName, "UseBumpMap"))
      {
orxConfig_PushSection(orxObject_GetName(orxOBJECT(_pstEvent->hSender)));
        pstPayload->fValue = (orxConfig_GetBool("UseBumpMap") != orxFALSE) ?
orxFLOAT_1 : orxFLOAT_0;
        orxConfig_PopSection();
      }
      else if(!orxString_Compare(pstPayload->zParamName, "vSize"))
      {
        orxObject_GetSize(orxOBJECT(_pstEvent->hSender),
&(pstPayload->vValue));
      }
      else if(!orxString_Compare(pstPayload->zParamName, "avLightColor"))
      {
        orxVector_Copy(&(pstPayload->vValue),
&(astLightList[pstPayload->s32ParamIndex].stColor.vRGB));
      }
      else if(!orxString_Compare(pstPayload->zParamName, "afLightAlpha"))
      {
        pstPayload->fValue =
astLightList[pstPayload->s32ParamIndex].stColor.fAlpha;
      }
      else if(!orxString_Compare(pstPayload->zParamName, "avLightPos"))
      {
        orxVector_Copy(&(pstPayload->vValue),
&(astLightList[pstPayload->s32ParamIndex].vPosition));
      }
      else if(!orxString_Compare(pstPayload->zParamName, "afLightRadius"))
      {
        pstPayload->fValue =
astLightList[pstPayload->s32ParamIndex].fRadius;
      }
      else if(!orxString_Compare(pstPayload->zParamName, "NormalMap"))
      {
        pstPayload->pstValue = (orxTEXTURE
*)orxHashTable_Get(pstTextureTable,
orxString_ToCRC(orxTexture_GetName(pstPayload->pstValue)));
      }
    }
  }
```

When a `orxSHADER_EVENT_SET_PARAM` is received, we check the parameter name and we'll fill its content based on our stored light info.
Note that for arrays of parameters we use the array index to fill the right slot.
When the `NormalMap` parameter is requested, we'll try to find a precomputed normal map stored in our `pstTextureTable`.

```
  else if((_pstEvent->eType == orxEVENT_TYPE_TEXTURE) && (_pstEvent->eID ==
orxTEXTURE_EVENT_LOAD))
  {
    CreateNormalMap(orxTEXTURE(_pstEvent->hSender));
  }

  return eResult;
}
```

When a `orxTEXTURE_EVENT_LOAD` event is caught, we'll create a normal map for the concerned texture.
This means that a normal map for a given texture is created by calling `CreateNormalMap()` the first time a texture is loaded.
Let's have a look to that function more closely.

```
void CreateNormalMap(const orxTEXTURE *_pstTexture)
{
  const orxSTRING zName;

  zName = orxTexture_GetName(_pstTexture);
  if(zName && zName != orxSTRING_EMPTY)
  {
    orxU32 u32CRC;

    u32CRC = orxString_ToCRC(zName);
```

We're using the CRC of the texture name as a key for our normal map table. If it's not already used we need to create the associated normal map.

```
    if(!orxHashTable_Get(pstTextureTable, u32CRC))
    {
      orxFLOAT    fWidth, fHeight;
      orxU32      u32BufferSize;
      orxBITMAP  *pstBitmap, *pstNMBitmap;
      orxTEXTURE *pstNMTexture;
      orxU8      *pu8SrcBuffer, *pu8DstBuffer;
      orxCHAR     acNMName[256];

      pstBitmap = orxTexture_GetBitmap(_pstTexture);
      orxDisplay_GetBitmapSize(pstBitmap, &fWidth, &fHeight);
      u32BufferSize = (orxU32)(fWidth * fHeight) * sizeof(orxRGBA);
```

We now have the actual bitmap used by that texture.

```
      pu8SrcBuffer = orxMemory_Allocate(u32BufferSize,
orxMEMORY_TYPE_VIDEO);
```

```
      pu8DstBuffer = orxMemory_Allocate(u32BufferSize,
orxMEMORY_TYPE_VIDEO);

      orxDisplay_GetBitmapData(pstBitmap, pu8SrcBuffer, u32BufferSize);
      ComputeGreyImage(pu8SrcBuffer, u32BufferSize);
```

We got the actual pixels and turned them into a black&white image by calling `ComputeGreyImage` [1)].

```
      ComputeNormalMap(pu8SrcBuffer, pu8DstBuffer, (orxS32)fWidth,
(orxS32)fHeight);
```

Based on that B&W image we generated the actual normal map calling `ComputeNormalMap()`. We'll come back to this process later.

```
      pstNMBitmap = orxDisplay_CreateBitmap((orxU32)fWidth,
(orxU32)fHeight);
      orxDisplay_SetBitmapData(pstNMBitmap, pu8DstBuffer, u32BufferSize);

      orxMemory_Free(pu8SrcBuffer);
      orxMemory_Free(pu8DstBuffer);

      orxString_NPrint(acNMName, 256, "NM_%s", zName);

      pstNMTexture = orxTexture_Create();
      orxTexture_LinkBitmap(pstNMTexture, pstNMBitmap, acNMName);

      orxHashTable_Add(pstTextureTable, u32CRC, pstNMTexture);
    }
  }
}
```

We then created a new texture with the prefix `NM_` and filled it with our normap map image.
Finally we stored that texture in the table using the CRC of the original texture's name as a key.

Let's now go to the actual normal map creation process.

```
void ComputeNormalMap(const orxU8 *_pu8SrcBuffer, orxU8 *_pu8DstBuffer,
orxS32 _s32Width, orxS32 _s32Height)
{
  orxS32 i, j;

  for(i = ; i < _s32Height; i++)
  {
    for(j = ; j < _s32Width; j++)
    {
      orxS32          s32Index, s32Left, s32Right, s32Up, s32Down;
```

```
      orxFLOAT        fLeft, fRight, fUp, fDown;
      orxCOLOR        stNormal;
      orxU32          u32Pixel;
      const orxVECTOR vHalf = {orx2F(0.5f), orx2F(0.5f), orx2F(0.5f)};

      /* Gets pixel's index */
      s32Index = (i * _s32Width + j) * sizeof(orxRGBA);

      /* Gets neighbour indices */
      s32Left   = (i * _s32Width + orxMAX(j - 1, )) * sizeof(orxRGBA);
      s32Right  = (i * _s32Width + orxMIN(j + 1, _s32Width - 1)) *
sizeof(orxRGBA);
      s32Up     = (orxMAX(i - 1, ) * _s32Width + j) * sizeof(orxRGBA);
      s32Down   = (orxMIN(i + 1, _s32Height - 1) * _s32Width + j) *
sizeof(orxRGBA);

      /* Gets their normalized values */
      fLeft   = _pu8SrcBuffer[s32Left] * orxCOLOR_NORMALIZER;
      fRight  = _pu8SrcBuffer[s32Right] * orxCOLOR_NORMALIZER;
      fUp     = _pu8SrcBuffer[s32Up] * orxCOLOR_NORMALIZER;
      fDown   = _pu8SrcBuffer[s32Down] * orxCOLOR_NORMALIZER;

      /* Gets normal as color */
      orxVector_Add(&stNormal.vRGB, orxVector_Mulf(&stNormal.vRGB,
orxVector_Set(&stNormal.vRGB, (fLeft - fRight), fDown - fUp, orx2F(0.5f)),
orx2F(0.5f)), &vHalf);
      stNormal.fAlpha = orxFLOAT_1;

      /* Gets pixel value */
      u32Pixel = orxColor_ToRGBA(&stNormal);

      /* Stores it */
      _pu8DstBuffer[s32Index]     = orxRGBA_R(u32Pixel);
      _pu8DstBuffer[s32Index + 1] = orxRGBA_G(u32Pixel);
      _pu8DstBuffer[s32Index + 2] = orxRGBA_B(u32Pixel);
      _pu8DstBuffer[s32Index + 3] = orxRGBA_A(u32Pixel);
    }
  }
}
```

In this function, for every pixels, we look at its direct vertical and horizontal neighbors.
We then use the difference in their shade of greys to determine a normal for that pixel.
Finally we make sure every component is in the range [0, 1] instead of [-1, 1] by dividing its value by two and adding 0.5.
The reverse operation will then be done in the shader to “unpack” the normal.
The last step is to store that pixel that now contains our normal information in our image.

That's all for the source code. We'll now have a look at the config part.

We'll actually bypass all the usual object, graphic, input and spawner declarations as there's nothing new here compared to the previous tutorials.

Instead, let's focus on the shader code that's stored there.

```
[LightShader]
Code = "
```

First we define the shader's code:

```
vec2 GetLightVector(int _iIndex)
{
  return vec2((avLightPos[_iIndex].x - gl_FragCoord.x) / fScreenSize, 1.0 -
(avLightPos[_iIndex].y + gl_FragCoord.y) / fScreenSize);
}
```

We're simply getting the normalized vector from the current pixel to one of the lights here.

```
vec3 GetNormal()
{
  vec3        vNormal;
  const vec3  vHalf = vec3(0.5);

  vNormal = texture2D(NormalMap, gl_TexCoord[].xy).rgb;
  vNormal = 2.0 * (vNormal - vHalf);

  return vNormal;
}
```

That function “unpacks” the normal so that each component is now in the range [-1, 1].

```
vec4 GetLightValue(int _iIndex, vec3 _vNormal)
{
  float fIntensity, fBump;
  vec4  vValue;

  vec2 vLight = GetLightVector(_iIndex);

  fIntensity = clamp(1.0 - (1.0 / (afLightRadius[_iIndex] *
afLightRadius[_iIndex])) * length(vLight), 0.0, 1.0);

  if(UseBumpMap != 0.0)
  {
    fBump = dot(normalize(vec3(vLight, 0.1)), _vNormal);
  }
  else
  {
    fBump = 1.0;
```

```
  }

  vValue = fIntensity * vec4(fBump * avLightColor[_iIndex],
afLightAlpha[_iIndex]);

  return vValue;
}
```

This function gets the light contribution from a light. If bump mapping is active for that pixel, the actual lighting is modulated according to the pixel's normal.

```
void main()
{
  vec4  vColor, vPixel;
  vec3  vNormal;
  int   i;
  const int iLightNumber = 10;

  vColor = vec4(0.0, 0.0, 0.0, 0.0);

  vPixel = texture2D(Texture, gl_TexCoord[].xy);

  if(UseBumpMap != 0.0)
  {
    vNormal = GetNormal();
  }
  else
  {
    vNormal = vec3(0.0);
  }

  for(i = ; i < iLightNumber; i++)
  {
    vColor += GetLightValue(i, vNormal);
  }

  vColor.rgb += vAmbient;

  gl_FragColor.rgb  = vPixel.rgb * vColor.rgb;
  gl_FragColor.a    = vPixel.a - vColor.a;
}
"
```

Here we simply accumulate all the ligth contributions for a pixel and output the final color value.

Now comes the list of parameters for this shader. Their type [2)] is defined by the type of initial value we provide for them.

```
ParamList       = Texture # NormalMap # fScreenSize # vAmbient # avLightPos
# afLightRadius # avLightColor # afLightAlpha # UseBumpMap # vSize
fScreenSize     = @Display.ScreenHeight
vAmbient        = (0.05, 0.05, 0.05)
avLightPos      = (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0)
# (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0)
afLightRadius   = 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0
avLightColor    = (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0)
# (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0) # (0, 0, 0)
afLightAlpha    = 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0 # 0.0
UseBumpMap      = 1.0
vSize           = (0, 0, 0)
```

This is the list of all the parameters used by the shader. As you can see, values for arrays are provided through a config list.

Resources

Source code: 12_Lighting.c

Config file: 12_Lighting.ini

[1] which does a very simple conversion if you look at its code

[2] vector3, float or texture