⋮ INFO

CUSHY CODE

Various stories about Coding for Web and Video Games

TRANSFORMATIONS BETWEEN LOCAL, WORLD AND CAMERA SPACES IN UNITY3D SHADERS

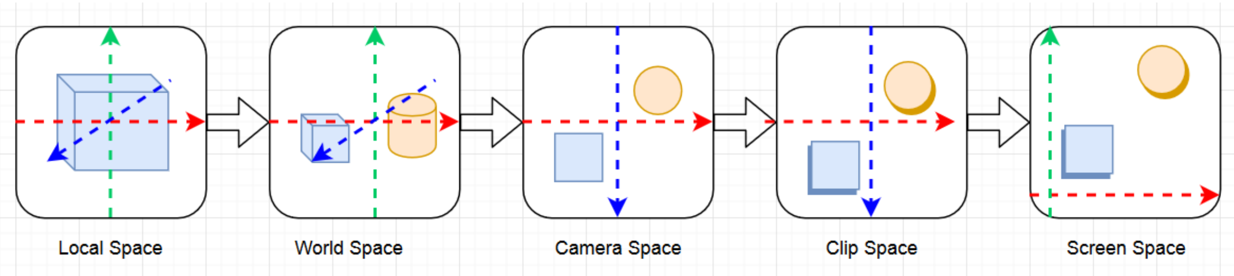
NOVEMBER 27, 2016
CUSHY
NO COMMENTS

Local, World and Camera spaces are one of the key concepts of Unity3D (or any other game engine, actually) rendering process. The knowledge about these coordinate systems and their relationships is an essential part of a successful shader development.

To decide how, where and what scene items should be rendered on the screen, Unity should transform items coordinates from Local (or Model) space to World space, than from World to Camera (or Eye) Space, and only that from Camera to Projection space. So, there are actually a total of 5 different coordinate systems that are of importance to us:

- Local space (or Object space)
- World space
- View space (or Eye space)
- Clip space
- Screen space

The following image represents the process of coordinate transformation from one to the next space.



The chain of matrix transformations is used to provide these changes. If you want to learn more about these matrices you can check this article, written by Marco Alamia, or here on learnopengl.com.

But let's talk about Unity. You do not need to worry about describing transformation when you use standard shaders and materials or even when you create your own surface shader (in main cases, of course). Unity does all the "dirty work" for you. However, the understanding of the rendering pipeline allows you to create a lot of interesting visual effects and design various animated and camera-position-related shaders.

Transformation matrices in Unity3d

As you probably know, the engine provides a bunch of built-in global variables for your shaders, including current object's transformation matrices. There are 9 float4x4 matrices provided by the current version of the Unity Engine (v5.4):

Name	Value
UNITY_MATRIX_MVP	Current model * view * projection matrix.
UNITY_MATRIX_MV	Current model * view matrix.
UNITY_MATRIX_V	Current view matrix.
UNITY_MATRIX_P	Current projection matrix.
UNITY_MATRIX_VP	Current view * projection matrix.
UNITY_MATRIX_T_MV	Transpose of model * view matrix.
UNITY_MATRIX_IT_MV	Inverse transpose of model * view matrix.
_Object2World	Current model matrix.
_World2Object	Inverse of current world matrix.

Switch between Spaces in a shader

All these built-in matrices allow you to transform point and/or direction coordinates between different coordinate systems. For simple shaders, you need only UNITY_MATRIX_MVP matrix to transform each vertex coordinates from Local direct to Clip space. However, you can also easily transform from Local to World space and back from world to local space using _Object2World and _World2Object.

You can also easily transform from Local space to Camera (Eye) space with the UNITY_MATRIX_MV matrix. But what to do if you need to apply camera-space-related transformation to vertices in your shader? Where you can get the UNITY_MATRIX_VM matrix?

First of all, as you can remember, that UNITY_MATRIX_T_MV is not the inverse of UNITY_MATRIX_MV. It means, that in general UNITY_MATRIX_T_MV * UNITY_MATRIX_MV != Matrix.Identity. However, you still have a way to transform coordinates from Camera to Local space. The trick is in the order of the multiplication a matrix and a vector. It's very important to understand, how do matrices multiply and that mul(matrix, vector) != mul(vector, matrix). After remembering what is the transpose and the inverse of a matrix, it becomes obvious, that the UNITY_MATRIX_IT_MV matrix is the key to our transformation!

It's time to summarize information about transformations between different spaces in shaders using provided matrices:

From	To	How to transform
Local Space	World Space	mul(_Object2World, currentPos)
Local Space	Camera Space	mul(UNITY_MATRIX_MV, currentPos)
Local Space	Clip Space	mul(UNITY_MATRIX_MVP, currentPos)
World Space	Local Space	mul(_World2Object, currentPos)
World Space	Camera Space	mul(UNITY_MATRIX_V, currentPos)
World Space	Clip Space	mul(UNITY_MATRIX_VP, currentPos)
Camera Space	Local Space	mul(currentPos, UNITY_MATRIX_IT_MV)
Camera Space	Clip Space	mul(UNITY_MATRIX_P, currentPos)

Please, note, that these operations are correct for vertex coordinates and don't relevant to normal transformations.

Also, we do not talk a lot about transformations to actual Screen coordinates, because it's a good theme for a separate article.

Screen space colorizer

Now we can use all this information to create a shader with some interesting effect. For the first example, let's create a shader that will change the color of an object based on its rotation related to the screen. There is a live demo of a possible variant of that kind of shaders:

Unity WebGL Player | 101DS-1

101DS-1

To achive this effect we have to specify a base direction (in the Eye space coordinates), and transform it to the object's Local Space and use it's normalized version as current color.

```
Shader "101DummyShaders/1/ScreenSpaceColor"
{
    Properties
    {
        _Direction("Screen Space Direction (x, y, z)", Vector) = (1, 0, 0
        _Glossiness("Smoothness", Range(0,1)) = 0.5
        _Metallic("Metallic", Range(0,1)) = 0.0
    }

    SubShader
    {
```

```
        Tags { "RenderType" = "Opaque" }
        LOD 200

        CGPROGRAM

        #pragma surface surf Standard vertex:vert fullforwardshadows
        #pragma target 3.0

        struct Input
        {
           fixed3 dirColor;
        };

        half _Glossiness;
        half _Metallic;
        fixed3 _Direction;

        void vert(inout appdata_full v, out Input o)
        {
           fixed3 localSpaceDir = mul(_Direction, (float3x3)UNITY_MATRIX_
           UNITY_INITIALIZE_OUTPUT(Input, o);
           o.dirColor = normalize(localSpaceDir);
        }

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
           o.Albedo = IN.dirColor;
           o.Metallic = _Metallic;
           o.Smoothness = _Glossiness;
           o.Alpha = 1;
        }
        ENDCG
        }
        FallBack "Diffuse"
  }
```

Quite interesting effect, isn't it? For example, with some small modifications this shader can be easily used in some fist-person shooter to highlight enemies' weak spots.

But what about something more advanced such as vertex deformation? It seems to be obvious to use the same technique to achieve view-related scaling, for example. But there is a trap that I'm going to describe in my next article.

CATEGORIES
#101 DUMMY SHADERS #KNOWLEDGE BASE #SHADER #UNITY

TAGS

#GAMEDEV #LIVE DEMO #SHADERS #TUTORIAL #UNITY3D

LEAVE A REPLY

YOUR EMAIL ADDRESS WILL NOT BE PUBLISHED. REQUIRED FIELDS ARE MARKED *

COMMENT

NAME *

EMAIL *

WEBSITE

POST COMMENT

PREVIOUS

LOWPOLY DISAPPEARANCE :: DEMO #2

NEXT

CALLIGRAPHY :: UNCIAL PRACTICE SHEETS

BACK TO TOP