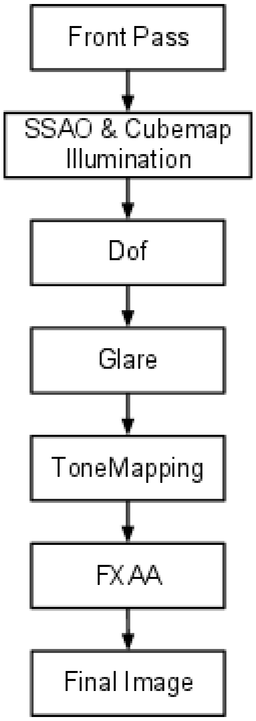
MRE - Multipass Render Engine

How it works

Introduction

Hello evvvverybody! It’s dottore here,I’m back after a some months of silence.
I’m happy to introduce you to this Multipass Render Engine, which is the result of the collaboration between me, m4d, vux and unc.
I’ll try my best to explain how the engine does work, and how to customize it.

Structure



Front Pass:

This first pass generates 3 textures at the same time, using an intrinsic directx function called MRT (multiple render targets)(http://en.wikipedia.org/wiki/Multiple_Render_Targets). MRT allows you to write up to 4 textures (render targets) in a single draw call (per mesh subset)
Some algorithms like SSAO or Dof need more information than the pixel color alone.
for our purpose, we’ll render 3 textures:

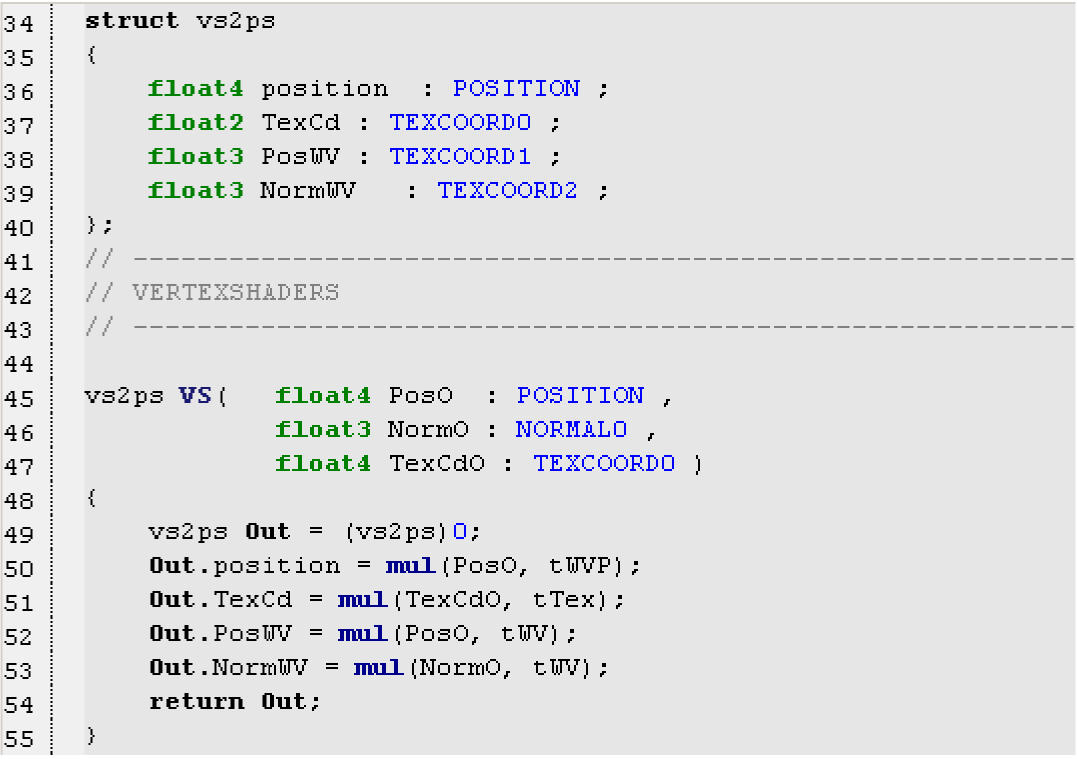
- **Color Texture (Color Buffer)**: This is color of the object, generated using common shading techniques

like Constant, PhongPoint, PhongDirectional, …
This render engine uses HDR (http://en.wikipedia.org/wiki/High_dynamic_range_imaging) ; we need float color data, so you can store numbers not only in the 0-1 space. We use A16B16G16R16F texture format.
Note: Transparency is not allowed in this engine, so you can’t apply alpha textures to the objects.

- **Position Texture (Position Buffer)**: This texture will contain position information for each pixel (XYZ = RGB). These position values are in Camera Space (after transforming them with tWV).
Position can be negative, so this render target (this texture) will be in float 32 bit format (A32B32G32R32F).

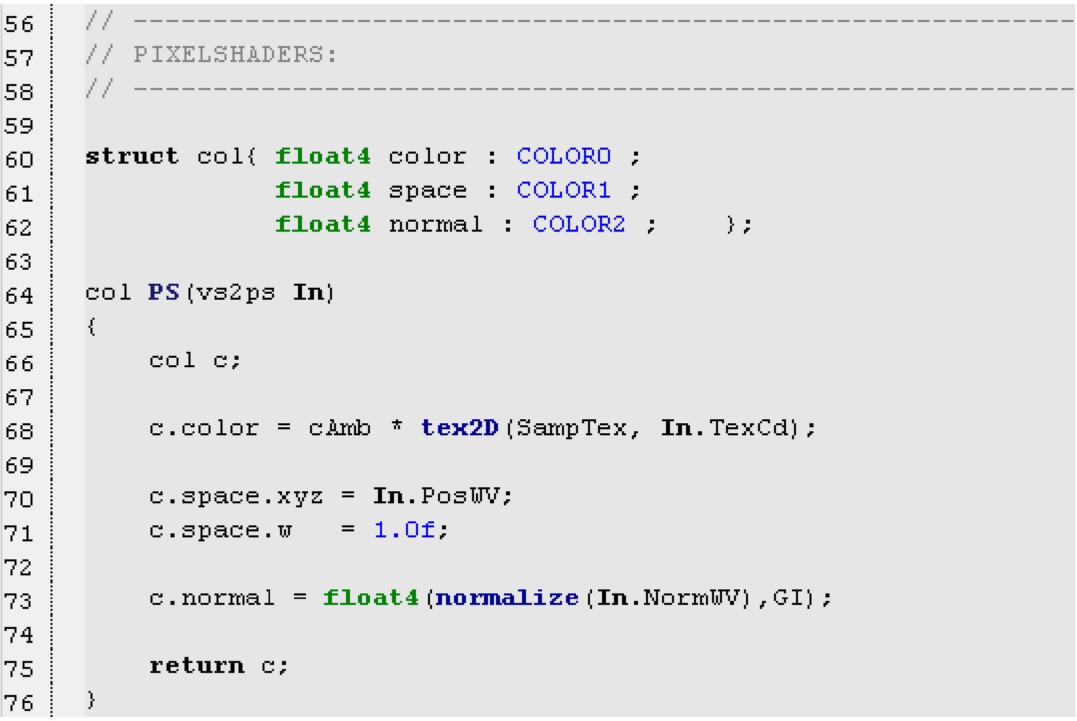
- **Normal Texture (Normal Buffer)**: Here we have normal information for each pixel.
Normal components can be negative, so this render target (this texture) will be in float 32 bit format (A16B16G16R16F).

So how do we write a MRT shader?
It’s not so different compared to a single output shader.
As an example, here is Constant (MRT):



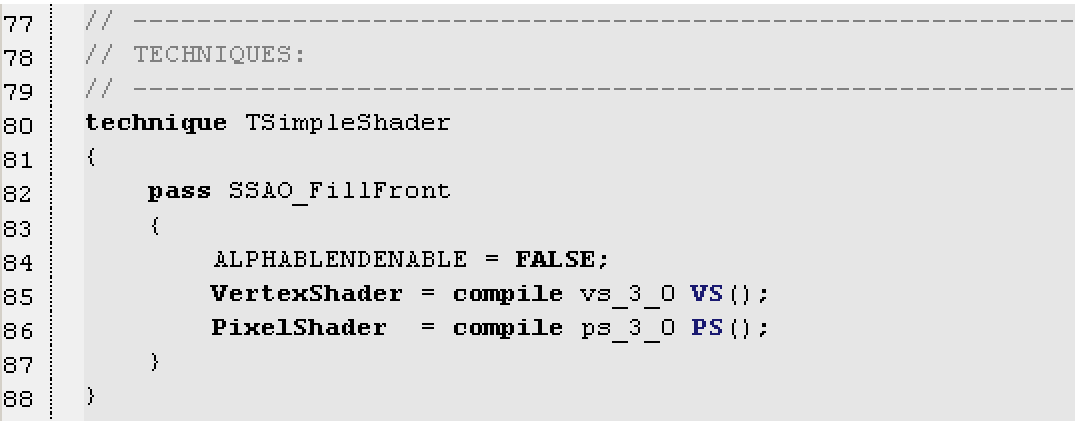
```
34  struct vs2ps
35  {
36      float4 position   : POSITION ;
37      float2 TexCd : TEXCOORD0 ;
38      float3 PosWV : TEXCOORD1 ;
39      float3 NormWV   : TEXCOORD2 ;
40  };
41  // -----------------------------------------------------------
42  // VERTEXSHADERS
43  // -----------------------------------------------------------
44
45  vs2ps VS(   float4 Pos0  : POSITION ,
46              float3 Norm0 : NORMAL0 ,
47              float4 TexCd0 : TEXCOORD0 )
48  {
49      vs2ps Out = (vs2ps)0;
50      Out.position = mul(Pos0, tWVP);
51      Out.TexCd = mul(TexCd0, tTex);
52      Out.PosWV = mul(Pos0, tWV);
53      Out.NormWV = mul(Norm0, tWV);
54      return Out;
55  }
```

Here in the vertex shader we must output all the information needed by the pixel shader for MRT rendering.
As usual we pass vertex position in Screen space (line 36 and 50) and Texture coordinates needed to apply a texture onto the object (line 37 and 51);
Then we need to output vertex position in camera space for the Position Buffer (line 38 and 52) and Normal information in camera space for the Normal Buffer (line 39 and 53).



```
56  // ----------------------------------------------------------
57  // PIXELSHADERS:
58  // ----------------------------------------------------------
59
60  struct col{ float4 color : COLOR0 ;
61              float4 space : COLOR1 ;
62              float4 normal : COLOR2 ;     };
63
64  col PS(vs2ps In)
65  {
66      col c;
67
68      c.color = cAmb * tex2D(SampTex, In.TexCd);
69
70      c.space.xyz = In.PosWV;
71      c.space.w   = 1.0f;
72
73      c.normal = float4(normalize(In.NormWV),GI);
74
75      return c;
76  }
```

Here we find the most evident difference with a non MRT shader.
This pixel shader must output colors into three different Render Targets.
In the struct part (line 60 to 62) we declare the outputs of the PixelShader function.
inside the pixel shader we define how to render into each buffer:
- color: it’s a common constant shading, combining given color with texture color.
- position: pixel position in camera space
- normal: pixel normals, additionally, we use the spare alpha channel to store GI (global illumination), which is useful to switch off SSAO and Cubemap on specific objects.



```
77  // ----------------------------------------------------------
78  // TECHNIQUES:
79  // ----------------------------------------------------------
80  technique TSimpleShader
81  {
82      pass SSAO_FillFront
83      {
84          ALPHABLENDENABLE = FALSE;
85          VertexShader = compile vs_3_0 VS();
86          PixelShader  = compile ps_3_0 PS();
87      }
88  }
```

Note: since engine doesn’t support transparent objects, we’ll disable alpha blend in technique (line 84).
Now that you know what a shader needs to work with MRT. You can try to port almost any existing material shader you can find in the Contribution vvvv.org page (Or from other sources).
For this release i have built a few fundamentals first pass shaders: Constant, PhongPoint and PhongDirectional, each one in two variants, with and without shadows (“..._MRT_VSM” and “..._MRT”).

It would be nice to collect a bunch of MRT shaders for the next releases of this engine. Please contribute!

VSM Shadows

VSM is “Variance Shadow Map”. Have a look here to have a better understanding about this approach:
http://www.punkuser.net/vsm/vsm_paper.pdf
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch08.html
Basically the geometry is rendered from the light point of view and a ShadowMap is generated.
Inside the module called “VSM ShadowMap” you will find the renderer of the shadowmap.
Shadowmap is then blurred (there are 2 techniques: gaussian or box) to smooth the shadow edges.
The final smooth shadow map is taken by each front shader that will use it to find shadowed pixels.

SSAO and Cubemap Illumination:

SSAO: Screen Space Ambient Occlusion (http://en.wikipedia.org/wiki/Screen_Space_Ambient_Occlusion)
Hope that SSAO is nothing new for you; it’s been a long time the great joreg and our friend m4d ported it into vvvv (http://vvvv.org/contribution/interleaved-ssao-with-cubemap-lighting).
It requires color, position and normal buffers.
In the same module we integrated Cubemap Illumination, a nice trick to emulate global illumination by sampling an environment texture (cubemap: http://en.wikipedia.org/wiki/Cube_mapping) according to normals of the object.

Fog

(you find this pass in customized version of the engine)
Fog is using the same technique a per the fixed function counterpart (exp pixel fog).
To calculate distance fog, we just compute the distance from the viewer position.
3 techniques are provided:
Simple Exponential : fog strength is exponentially based on distance from camera
Exponential + Height : same as above, but a second option is provided to limit fog height.
Exponential + Sun : same as first with sun color/strength extra parameters.

Dof (Depth of field)

It’s the same module I contributed last year (http://vvvv.org/contribution/dof), modified to fit into this engine pipeline (it doesn’t use depthmap anymore, since it retrieves depth from Position buffer).
It requires color and position buffers.

Glare

Thanks to unc and his fast and furious programming skills we implemented a new Glare effect in the engine.
It’s really nice and smooth :)
what’s glare effect:
http://en.wikipedia.org/wiki/Glare_%28vision%29

Tone Mapping

http://en.wikipedia.org/wiki/Tone_mapping
Inside this module there are many parts:

- **Tone Mapping Setting Values**: here we define how the original HDR image is mapped to the 8bit range.
  The tone mapper algorithm is taken from here: http://filmicgames.com/archives/75;
  is the same algorithm used for Uncharted 2 video game.
- **Luminance Adaptation**: this little engine evaluates luminance, downsample the image and take the average luma value using a pipet node. Average luminance is then used for Automatic luminance adaptation in the ToneMapper algorithm.
- **Color Correction**: a simple color transform that allow you to adjust some color parameters (hue, saturation, value, rgb levels, …)
- **Vignette**: the famous and overused postFX (http://en.wikipedia.org/wiki/Vignetting) with some control parameters.

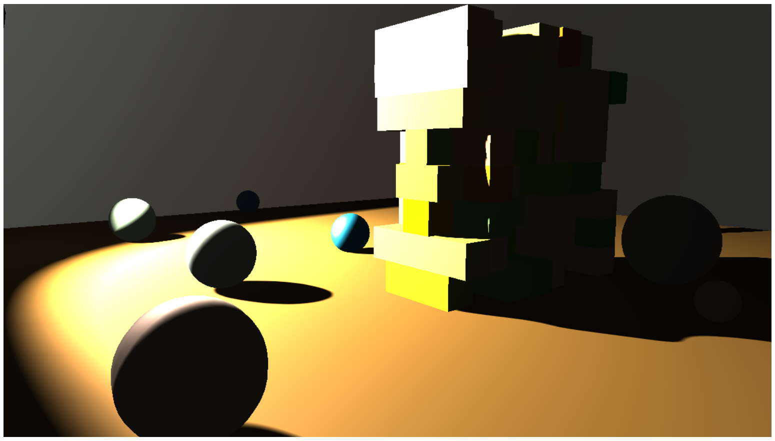
FXAA

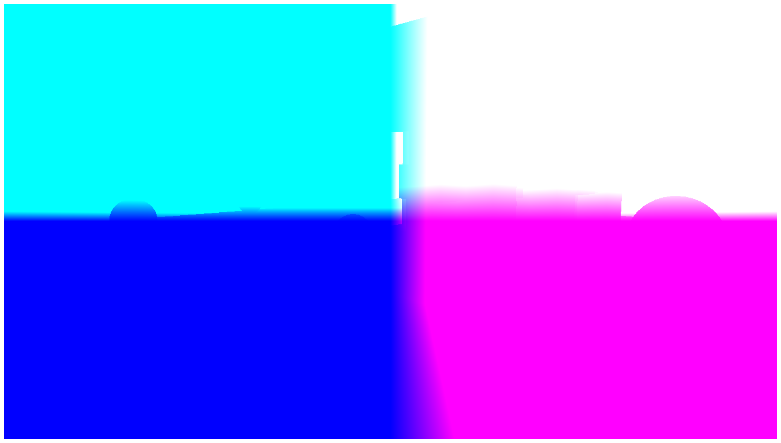
Just to have an idea:
http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
Our Implementation of the fxaa is exactly the same you find in the addon pack: **FXAA (DX9.Texture Filter)** module.
The HLSL code was kindly provided to me by the great Matt Swoboda (http://directtovideo.wordpress.com/), literally a living legend in the game dev world. Tnx Matt :)
I created a custom FXAA for this engine because a layer output was needed instead a texture output, allowing to save a useless extra pass.

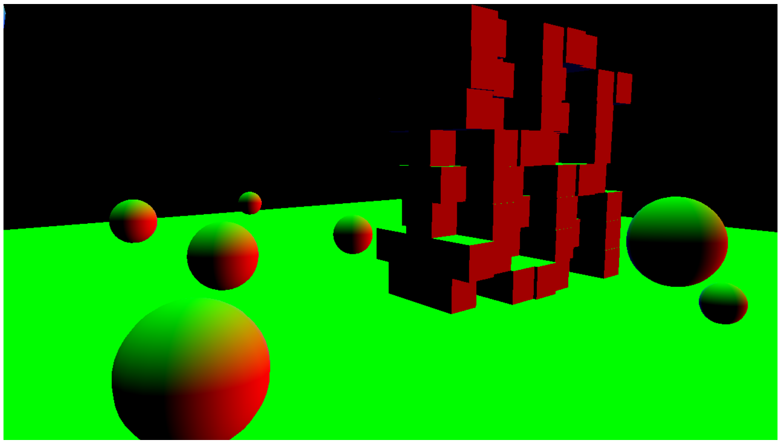
Summary:

Step by step visualization of all the buffers and passes up to the final image:

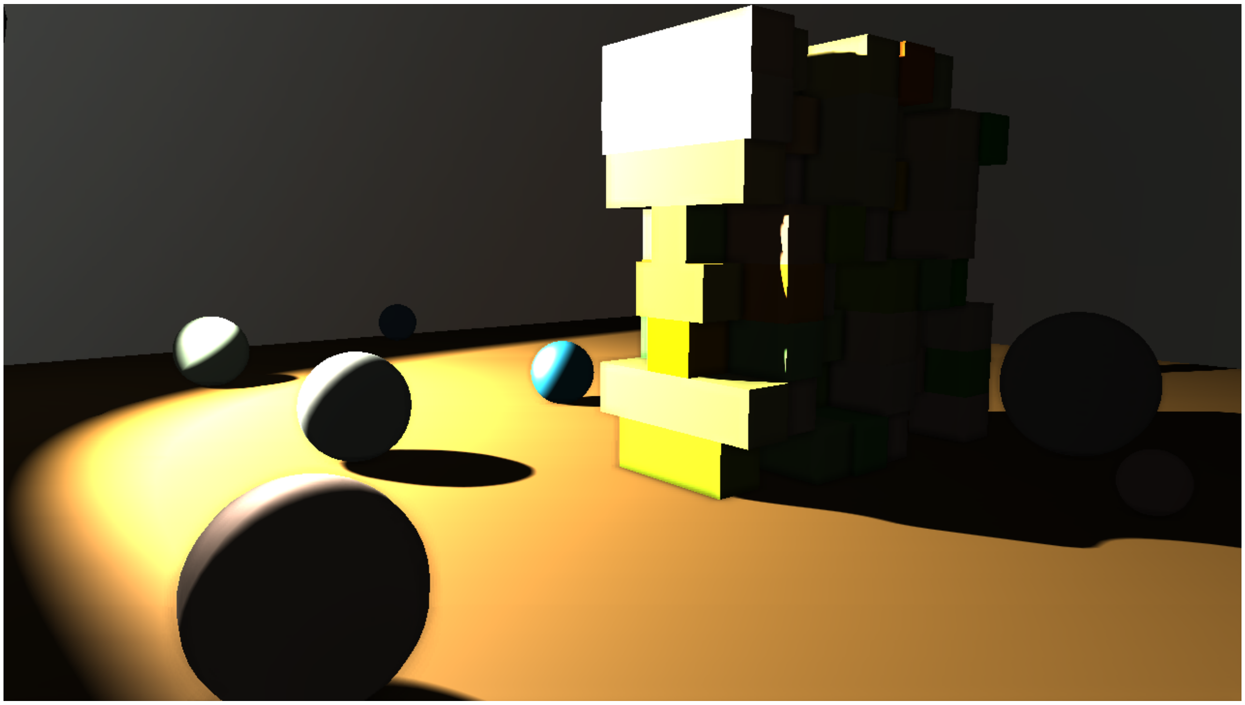
1) MRT: hdr color buffer, space buffer, normal buffer. how they look like:

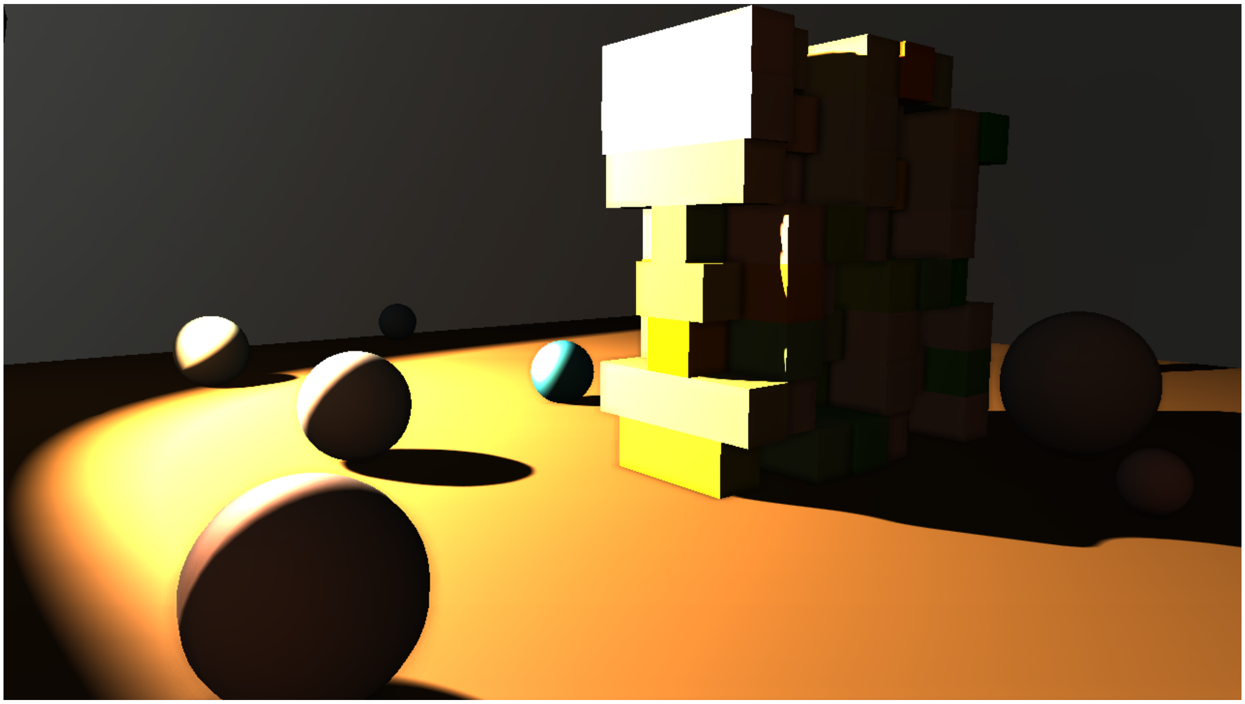




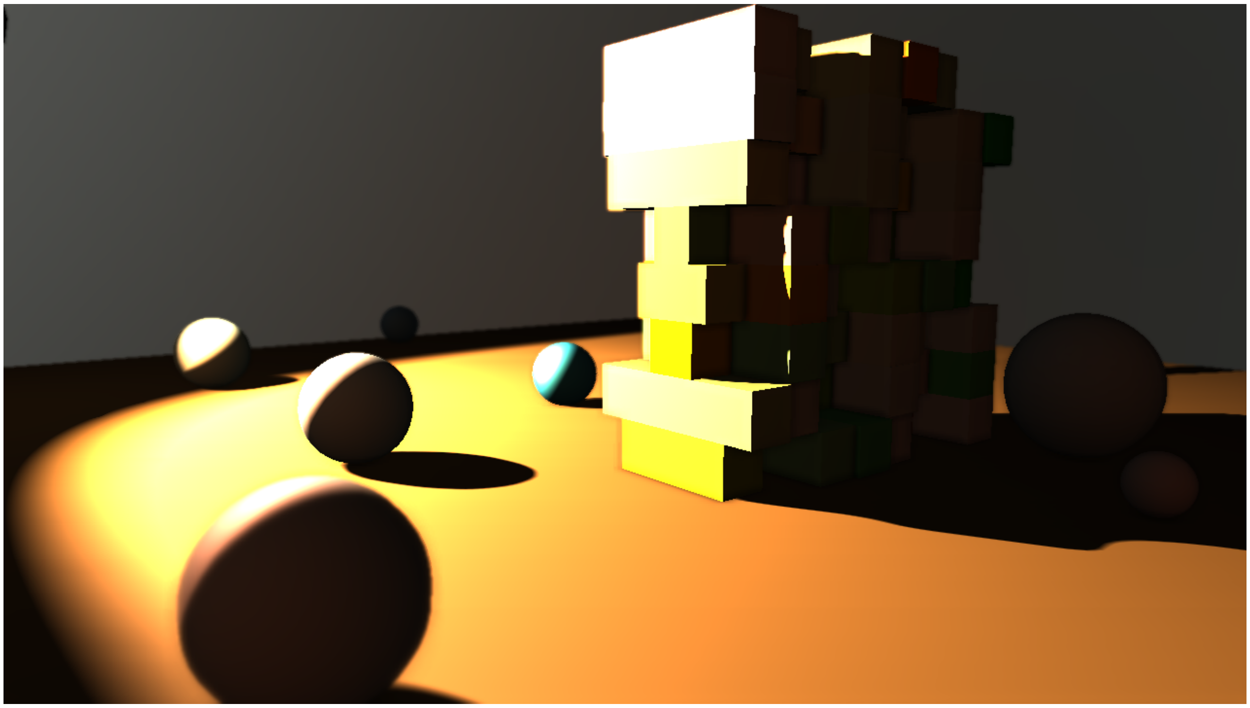


2) SSAO then Cubemap Illumination:

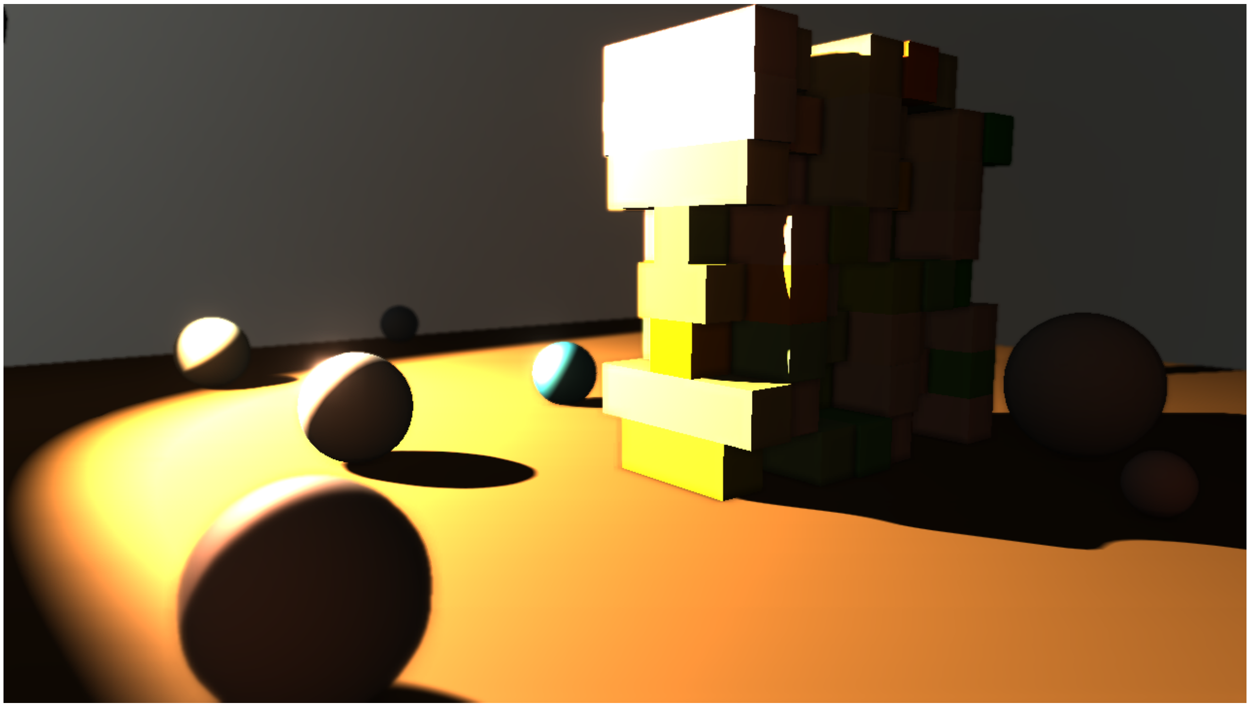




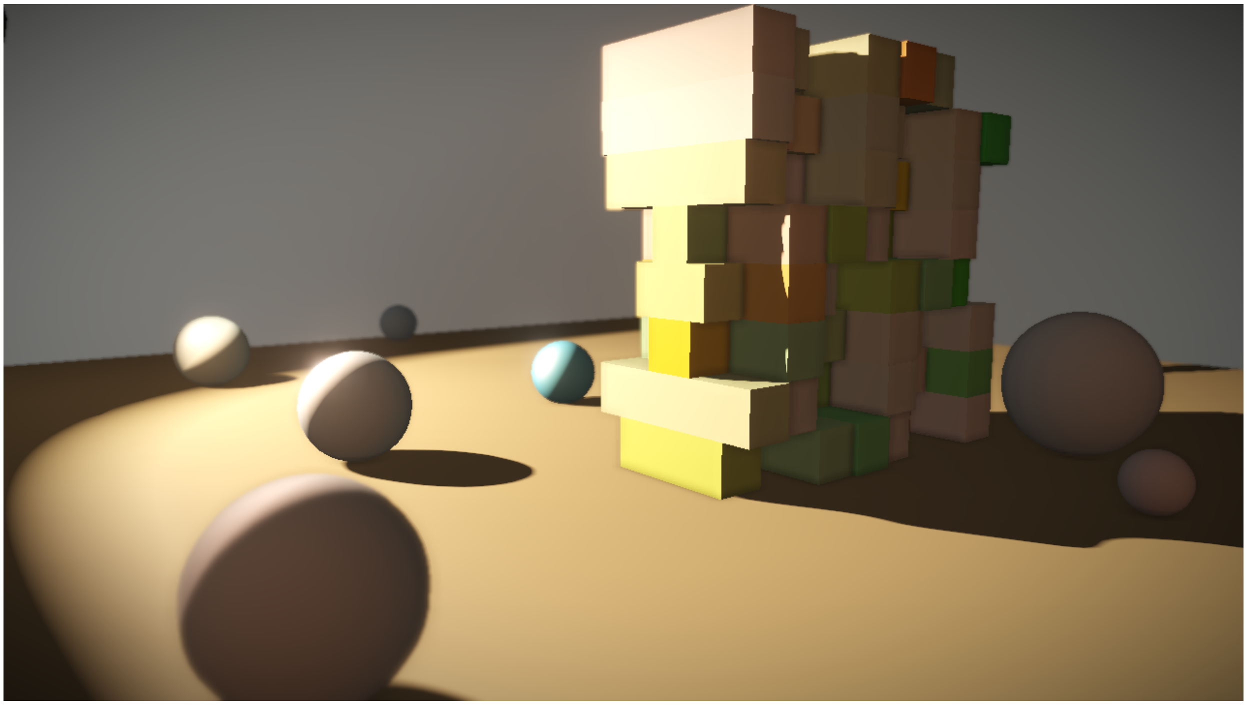
3) Dof:



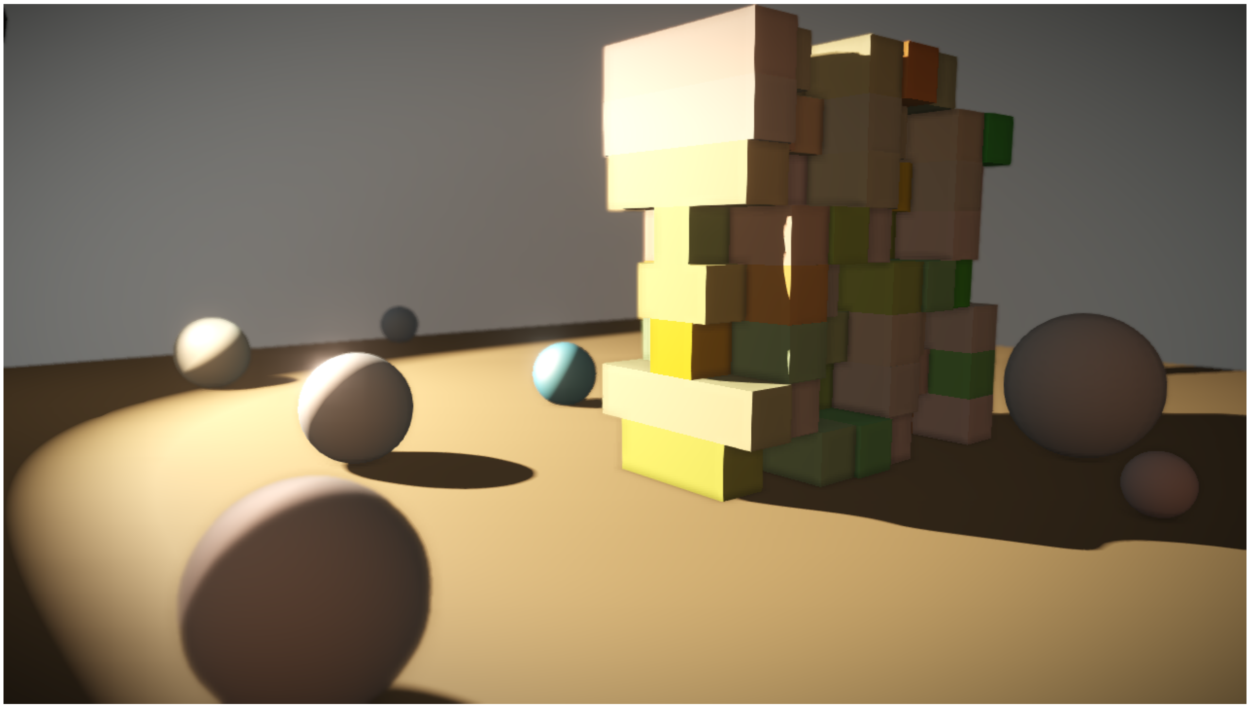
4) Glare:



5) ToneMapping and Vignette:



5) Final image after applying the last FXAA pass:



Thanks:

to devvvvs for being devvvvs.
to vux, m4d and unc whom intensively helped me during the last few days to build up this engine. ...was a little rush but we finished it in time for Patchakucha in Milan (28/01/2012). ;)

...je suis tres fatigué..

...wow this is also the first guide i complete... ;P

Ciao

Natan