Imageon™ 238x Rooms Demo

IMAGEON 3D






Table of Contents

1. Introduction

ATI's Imageon 238x graphics processor accelerates nearly all of the features in the OpenGL ES 1.1 Extension Pack specification. In addition, the Imageon 238x supports several extensions that improve on the performance and rendering capabilities of core OpenGL ES 1.1+. The Rooms demo was designed to showcase some of the key visual techniques enabled by the Imageon 238x. This whitepaper describes the technical details behind the techniques used in each demo.

2. Lighting

The Lighting demo was designed to showcase the per-pixel lighting capabilities of the Imageon 238x. There are a variety of techniques used in this demo:

- Bumpmapping
- Projective spotlights
- Per-pixel specular cubemap
- Shadows

2.1 Bumpmapping

The Imageon 238x supports the DOT3 blend operation which allows the diffuse lighting equation (N.L) to be evaluated at each pixel. The Lighting demo uses two types of normal maps: tangent-space and world-space.

The walls in the Lighting demo use tangent-space normal maps. At each vertex, the vector from the light center to the vertex is computed. The application stores a 3x3 tangent matrix at each vertex that is pre-computed in a preprocessor. The 3x3 tangent matrix is composed of a Tangent (*T),* Binormal (*B),* and Normal (*N*):

$$\begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \\ N.x & N.y & N.z \end{bmatrix}$$

For more on how to compute the tangent-space for a vertex, please see:

http://www.ati.com/developer/sdk/RadeonSDK/Html/Tutorials/RadeonBumpMap.html

The tangent-space light vector is then placed in the primary color channel. The primary color has a range of [0, 1] so the light vector *(LV')* is scaled and biased as follows:

$$LV' = LV * 0.5 + 0.5$$

The per-pixel DOT3 equation in OpenGL ES will perform the reverse bias and scale to get the value in the range [-1, 1].

Finally, the trashcan object is drawn with a world-space normal map. In a world-space normal map, the texture stores the actual world-space normal at each texel. The advantage to using this type of map is that the light vector does not need to be transformed into tangent space at each vertex as it can be sent in as one color to be applied at all pixels. The necessary restriction in order to use a world-space normal map is that the object never moves. The choice of which technique to use is based on the composition of the scene and the performance/memory trade-off for each technique.

Figure 2.1 shows the result of the DOT3 per-pixel bumpmapping.

Figure 2.1 – DOT3 Bumpmapping

2.2 Projective Spotlights

Figure 2.2 – Projective Spotlights

In order to achieve per-pixel spotlight falloff, a projective texture approach is used. The spotlight effect is approximated with a 2D texture representing the light falloff (*Figure 2.3)*:

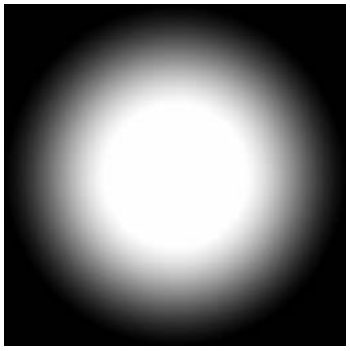


Figure 2.3 – Spotlight Falloff Texture

The spotlight falloff texture is applied to each surface using projective texturing and NEAREST filtering. For each vertex, the (x, y, z) position is passed in as a texture coordinate. The texture matrix is then used to transform the position into projective light-space. The texture matrix (*M)* is composed of:

$$M = Bias * LP * LV$$

LV=ModelView matrix from light

LP = Projection matrix from light

Bias = Bias matrix to get coordinates from [-1, 1] into [0,1]

The *LV* and *LP* matrices are calculated by creating a camera that uses the light frustum as a viewing frustum. The field-of-view for the light frustum determines how wide the spotlight cone angle is. The *Bias* matrix simply biases and scales the coordinates into the range [0, 1].

Unfortunately, although this projective texture approach will produce the spotlight effect, it also has an artifact known as back projection as highlighted in *Figure 2.4*. The back projection is due to the coordinates going negative behind the near frustum of the light.

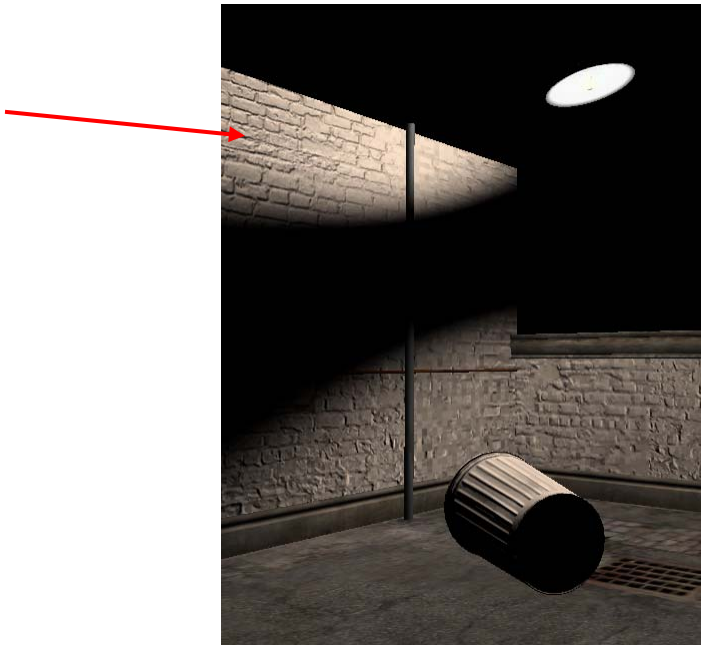


Figure 2.4 – Spotlight Back Projection

Traditionally, the back projection is resolved by using a 1D texture with a black texture border and OBJECT_LINEAR texture coordinate generation to fetch a value of black behind the light plane and white in front of it. Unfortunately, OpenGL ES 1.1+ does not have texture border color or OBJECT_LINEAR texture coordinate generation. However, the problem can still be resolved in a similar manner.

In the demo, a 256x1 texture is created that contains white at texel location (1.0, 0.0) and black everywhere else. On the second texture unit, the (x, y, z) position is passed in as a texture coordinate. Another texture matrix is used that contains the plane equation of the light in world-space:

$$\begin{bmatrix} P.x & 0 & 0 & 0 \\ P.y & 0 & 0 & 0 \\ P.z & 0 & 0 & 0 \\ P.w & 0 & 0 & 1 \end{bmatrix}$$

P = Plane equation for Light near-plane

This texture matrix causes the evaluation of the distance to plane equation:

$$Dist = P.x * V.x + P.y * V.y + P.Z * V.z + P.w * 1.0$$

The result of this equation is then used as the texture coordinate to fetch into the 256x1 texture. When a pixel is behind the light near plane, it gets a value of black from the texture. When a pixel is in front of the light plane, it gets a value of white. This value is multiplied by the spotlight falloff and then multiplied with the framebuffer using blending to get the final spotlight effect.

2.3 Per-pixel Specular Cubemap

In order to get per-pixel specular highlights on the trashcan, a specular cubemap is used to approximate the specular lighting equation. The specular cubemap simply contains a circular highlight in one of the cubemap faces. When drawing the trashcan, REFLECTION_MAP texture coordinate generation is used to generate a reflection vector in eye-space. That reflection vector is rotated by a texture matrix that computes the rotation of the reflection vector into the specular cubemap space (results in *Figure 2.5)*.

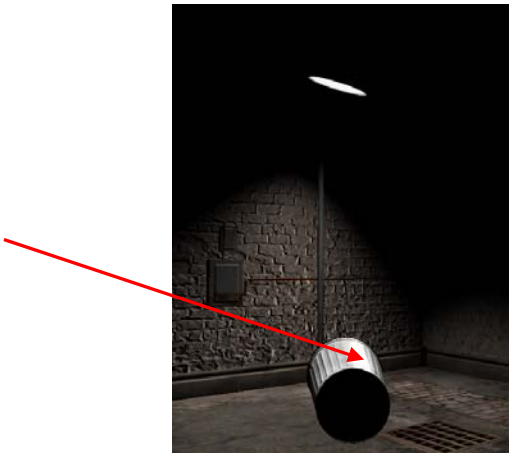


Figure 2.5 – Per-pixel specular cubemap

2.4 Shadows

Finally, in order to draw shadows on the floor and wall, a stencil shadow technique is used. The geometry is “squashed” onto the plane of the floor and wall using a matrix based on the light direction. The shadow is drawn by using the stencil buffer to only write each pixel once. The shadow geometry is drawn on top of the shadow plane using alpha blending. Once a pixel is written, it also writes a value to the stencil buffer telling it not to write that pixel again. Additionally, the stencil buffer is used to test which regions to draw the shadow. The shadows are shown below (*Figure 2.6*).

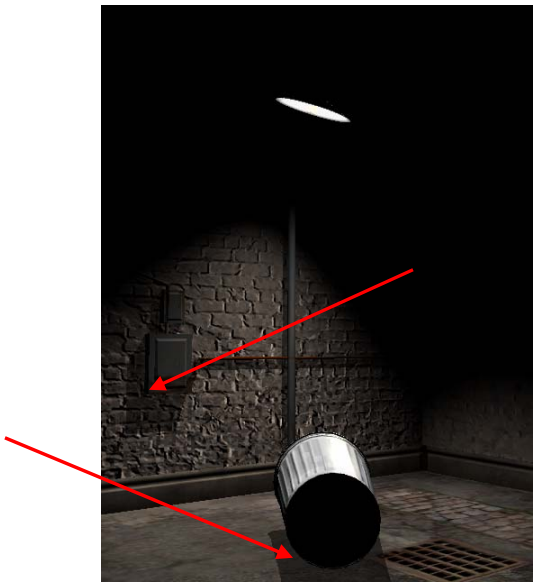
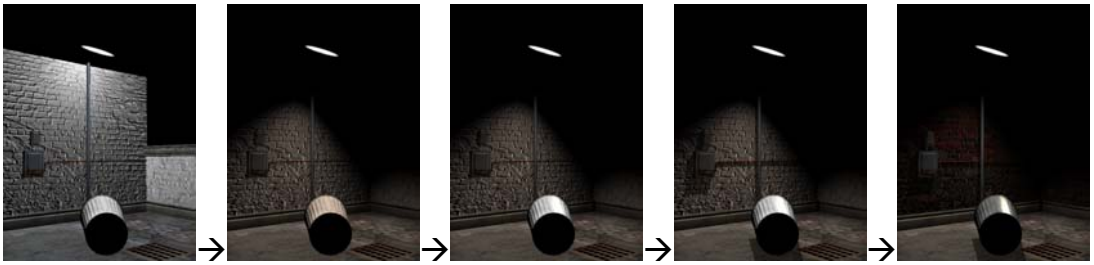


Figure 2.6 – Shadows

2.5 Final Rendering

The following figure (*Figure 2.7*) shows each of the components that go into rendering the final Lighting demo image.



Bumpmapping * Spotlight + Specular + Shadows * Base = Final

Figure 2.7 – Components of Final Frame

3. Water

The Water demo was designed to showcase a variety of reflection and refraction techniques made possible by the Imageon 238x. The Water demo uses the following techniques:

- Refraction mapping
- Dynamic planar reflections
- Cubemaps
- Projective texturing
- Framebuffer objects
- Lightmaps

3.1 Refraction Mapping

The approach to water rendering in this demo was taken from "*Refraction Mapping for Liquids in Containers*" by Alex Vlachos and Jason Mitchell (Game Programming Gems). The water is simulated as a lattice of vertices. In order to increase performance, the simulation used in the Water demo simply uses sine and cosine waves to animate the water. The refraction technique used is exactly as presented in Game Programming Gems. This technique uses a 2D texture and dynamically generates refraction texture coordinates based on the water position and view vector. The refraction texture map is shown in *Figure 3.1*.

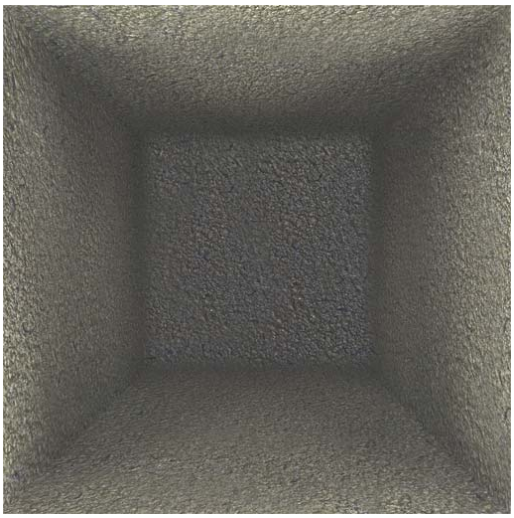


Figure 3.1 – Refraction Texture Map

3.2 Cubemaps

The reflection on the water pool is generated using a static cubemap. The static cubemap is pre-generated by placing a camera at the center of the scene and rendering each cube face with an FOV of 90 degrees and an aspect ratio of 1.0 (*Figure 3.2)*. The normals of the water surface are updated dynamically each frame and REFLECTION_MAP texture coordinate generation is used to generate cubemap lookup coordinates.

There is one trick that is required to do proper cubemap reflections with a moving camera. The REFLECTION_MAP texture coordinate generation technique generates a reflection vector in eye-space. However, the cubemap for the world is pre-generated in world-space. The reflection vector must be transformed from eye-space to world-space in order to get the correct cubemap lookup. This transformation is done on the GPU using the texture matrix. The texture matrix is loaded with the upper 3x3 of the inverse transpose of the modelview matrix.

Figure 3.2 – Static Cubemap for Reflection

3.3 Dynamic Planar Reflections

In addition to the main pool of water in the center, the floor is also mapped with puddles that have dynamic planar reflections. A framebuffer object is used to render an off-screen texture that contains the reflection map. This map is generated by flipping the camera about the floor plane and then switching the cull direction. The reflections are mapped onto geometry with projective texturing. The (x, y, z) position is passed in as a texture coordinate, just as in *Section 2.2 – Projective Spotlights*. The texture matrix is then used to transform the position into projective view-space using the following matrix:

$$M = Bias * CP * CV$$

CV = ModelView matrix from the camera

CP = Projection matrix from camera

Bias = Bias matrix to get coordinates from [-1, 1] into [0,1]

This matrix uses the current modelview and projection matrices for the camera along with a bias matrix to generate the projective texture coordinate.

In order to mask reflections for the puddles, the alpha channel of the basemap on the floor contains a [0, 1] value that specifies whether a region is a puddle. This alpha value is multiplied by the reflection in order to produce puddles in only particular regions (Figure 3.2).

Figure 3.2 – Planar Dynamic Reflections

4. Postprocessing

The Postprocessing demo uses framebuffer objects to perform light bloom. The concept of the demo was to simulate the look of HDR using fixed-function postprocessing.

4.1 Light Bloom

The basic postprocessing technique is described in the link below:

http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/Direct3D/RadeonLightGlare.html

The idea is to render the lit part of the scene in white and all obscuring objects in black to an off screen texture. This image is then down sampled multiple times to create several blurry versions of the texture. In the Postprocessing demo, the scene is rendered to a 256x256 texture then down sampled to 128x128, 64x64, 32x32, and 16x16. The four down sampled images are then combined with different weights to produce the final glare texture (*Figure 4.1*). The glare texture is then drawn on a full-screen quad and blended with the scene.

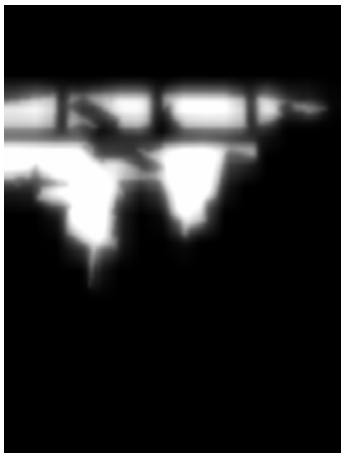


Figure 4.1 – Light Glare Texture

5. Reflect

The Reflect demo was designed to showcase the use of framebuffer objects and dynamic cubemap rendering to produce complex dynamic reflections.

5.1 Dynamic Cubemap

The Reflect demo creates six 256x256 framebuffer objects, one for each cubemap face. Each frame, the six faces of the cubemap are rendered from the center position of the ball. The objects are bounding-box culled against the view frustum for each face of the cubemap. For performance, the artists authored a "proxy" version of the scene that contains a lower polygon count than the original model that is rendered into the reflection. The ball and train cars are also vertex lit with a directional GL light that animates throughout the scene.

The cubemap is rendered on the ball using the same technique as in *Section 3.2*. The inverse upper 3x3 of the modelview matrix is used to transform the reflection vector from eye-space to world-space.

6. Particles

The purpose of the Particles demo was to showcase the use of point-sprites to accelerate the rendering of screen-aligned billboards in a particle system.

6.1 Particle System

The particle system used in this demo has many parameters that can be tuned to achieve a variety of effects. The fire used in this demo is accomplished by creating a particle

emitter than emits particles about a 2D random radius from a center point. Each particle is created with a random size clamped to a particular range. The particle size decreases based on the lifetime of the particle to another size randomly chosen at particle creation time. Finally, the color of the particle also decreases over time so that the particles appear to fade at the top of the fire.

OpenGL ES 1.1+ provides the ability to specify an array that has a point size for each sprite. This was not possible in traditional OpenGL where only one point size could be used for a glDrawElements or glDrawArrays call. The particle system dynamically updates this point size array every frame based on the particle lifetime and properties.

6.2 Point Parameters

In order to use point sprites for rendering a particle system, the distance attenuation that is provided by glPointParameters is used. The distance attenuation allows the application to set the constants a, b, and c, in the following equation:

$$atten(d) = \frac{1}{a + b * d + c * d^2}$$

d = eye-space distance to viewer

The Particles demo sets up the point parameters equation as follows:

$$a = scale * scale$$

$$b = 0$$

$$c = scale * 0.15$$

The *scale* argument is computed as a ratio of the resolution to the base VGA resolution. This scaling is necessary in order to achieve particle sizes that are relative to the screen resolution. The final point size is computed in terms of screen pixels so it is important to scale the point sprites by resolution change. The scale factor in c provides another scaling based on the square of the distance of the point to the camera.

The demo contains a rendering mode where the points are drawn with software aligned quads instead of point sprites. The distance attenuation equation allows the point sprites to nearly match identically to what the software aligned quads produce.

6.3 Lightmap attenuation

One final trick that was used to increase the realism of the scene is to globally attenuate the lightmap by a smooth random noise value. This simulated the look of flicker that is present from a fire. The random noise value is based on a lookup table that was generated as a texture by the artists.

7. Skinning

The Skinning demo uses the Matrix Palette feature of the Imageon 238x to perform vertex skinning. It also uses several lighting techniques in order to achieve dynamic lighting.

7.1 Matrix Palette

The hand used in the skinning demo has 20 bones and each vertex has up to 4 bones influencing it. There are a total of 5766 vertices in the model. Using matrix palettes, the entire bone set is uploaded each frame based on the current time in the key frame animation. Each vertex contains both a set of weights and set of bone indices that describe how skinning should be calculated.

7.2 Lighting the Skinned Model

There are two techniques used to compute lighting on the model. The first is simply per-vertex diffuse using GL lighting. The second technique is per-pixel specular as described in *Section 2.3*. The one addition to the specular technique is that there is also a gloss channel stored in the alpha of the hand texture map that gets multiplied by the specular highlight. The texture environment is configured to perform this computation in a single pass:

Unit 0: RGB = Basemap.RGB * Diffuse.RGB
Alpha = Specular.A * Base.A (Gloss)

Unit 1: RGB = Previous.rgb + Previous.a (Base * Diffuse + Specular)

The use of the gloss channel allows the specular highlights to vary across the surface of the hand and provides a more realistic look.

WP 0501 Rooms Demo 1.0

ATI

ati.com