Insomnia

Markus Lipp (0125260/932)
Stefan Reinalter (0225790/932)
Students of Computer Science
Vienna University of Technology
markus-lipp@gmx.at
tivolo@cg.tuwien.ac.at

1 Introduction

Insomnia is a graphics demo programmed for the lecture *Real-Time Rendering* in winter term 2005/2006. This document describes various aspects of Insomnia. It is structured the following way: Section 2 will deal with known technical problems and solutions. Section 3 describes user controls available during the demo. Section 4 provides system requirements. Section 5 describes how to configure Insomnia for your needs. Section 6 provides a detailed description of all effects. Finally, Section 7 provides a list off the effects used in the different demo parts, followed by references.

2 Known Problems

If your monitor does not support 85hz (for example a TFT monitor) the refresh rate of the demo has to be lowered. Look at section 5 for instructions.

3 Controls

The following keys are available in all scenes except intro:

f: Freefly mode, directly control camera

p: Pause animations of camera and objects

When in freefly mode, the following controls are available

w,a,s,d: Movement

left mousebutton + drag: Look around

All asteroid scenes feature exposure adjustments:

up, down: Change exposure

left, right: Change gain

Asteroid scenes 2-6 additionally have the following *really* nice feature:

right mousebutton + drag: Change daytime

You can jump between demo scenes *intro, cathedral* and *statue* using F1 to F3. When you press F4 the first *asteroid scene* is loaded. Now it is possible to jump from asteroid scene 1 to 7 using F1-F7. Due to internal reloading it is not possible to jump from asteroid scenes backward to scenes 1-3. There is also a known bug: When you jump from asteroid scene 1 or 7 to scene 5 (*raytraced fun*) the water appears too dark.

If you are really curious on how a realtime-raytraced caustic texture looks like, you can press F12 in the raytraced water scene (F5) to see this texture.

4 System Requirements

Graphics Card: Geforce6600 or later. Note that this demo does not work on current (January 2006) ATI cards, as they are not shader model 3.0 compatible. Vertex texture fetch is required for the raytraced water. Any CPU above 1600MHz

should do. At least 512MB of RAM are recommended. The demo was only tested on WindowsXP.

5 Configuration

You can change file `config.cfg` to suit your needs. However you should be very careful, as there is no sanity check on those settings. Ridiculous settings may even damage your computer. When the file is not present, or is not well formed, standard settings (1024x768,85Hz,fullscreen) are used.

line 1: Horizontal resolution

line 2: Vertical resolution

line 3: Refresh rate

line 4: 1=fullscreen, 0=windowed

6 Effects

6.1 Ambient Occlusion

By precomputing the amount of incident light coming from the hemisphere when looking in the direction of a vertex's normal, shading appears to be much more realistic. The ambient occlusion preprocessor uses the GPU to calculate the integral of several rays shot from the hemisphere towards the model. By using occlusion queries we can quickly determine if a ray is blocked or not.

6.2 Precomputed Radiance Transfer

As Precomputed Radiance Transfer is a rather large subject on its own, we can't go into details about the implementation. However, the PRT implementation in this demo is based on Stefan Reinalter's Bachelor's Thesis about PRT. Mathematical explanations as well as implementation details can be found in his thesis.

6.3 Penumbra-Based Soft Shadows

Basically, the soft-shadows are implemented by using a Percentage Closer Filter with a variable filter kernel. In addition, the occluder for every pixel is searched and an approximation of the penumbra size is computed. Using this algorithm, area lights can be simulated because their varying penumbra will make the shadows appear softer the bigger the light source is. Taking into account the penumbra size, the soft-shadows algorithm blurs the shadows by averaging several shadow map lookups based on the current penumbra.

6.4 True Object-Based Motion Blur

Our motion blur is based on the actual objects' velocity. Every frame, the velocity of every pixel in post-perspective space gets written into a floating-point render target. Because we need the velocity of the current frame and the velocity of the last frame, two render targets are pingponged before rendering. In the fragment shader for the motion blur, the image gets blurred according to the velocity by taking multiple taps in the according direction. Additionally, objects are stretched based on their velocity by interpolating between the last frame's position and the current position in the vertex shader. This is an addition to the original algorithm and is taken from GDC 2003.

6.5 Normal Mapping

We implemented Tangent Space Normal Mapping. The normals used for lighting are perturbed by looking up normals from a texture.

6.6 Parallax Mapping

This effect is used in addition to Normal Mapping and makes surfaces appear to have height - it's a kind of faked displacement mapping. Our implementation is based on the original paper by Terry Welsh.

6.7 GPU-Fire

This effect is actually a GPU-implementation of an old, popular DOS-demo effect. The font gets rendered into a frame buffer where every pixel of the font gets a random alpha value between 200 and 255. The fragment shader then averages four pixels each, and subtracts a predefined value for every pixel so that the flames are getting "colder". The averaging is slightly offset to make the flames rise to the top of the screen. The last instruction in the pixel shader looks up a RGB-value for the current color index stored in the alpha channel of the render target. By using a predefined firepalette the impression of flames can be faked.

6.8 Bump-Reflection Mapping

This effect allows perturbed reflections using a cubemap, and is described in the Cg Manual [2]. Therefore a transformation from tangent space to world space, as well as a eye to vertex vector in world space are supplied by the vertex shader to the fragment shader. In the fragment shader a tangent space normalmap is fetched. The fetched normal is transformed to world space using the

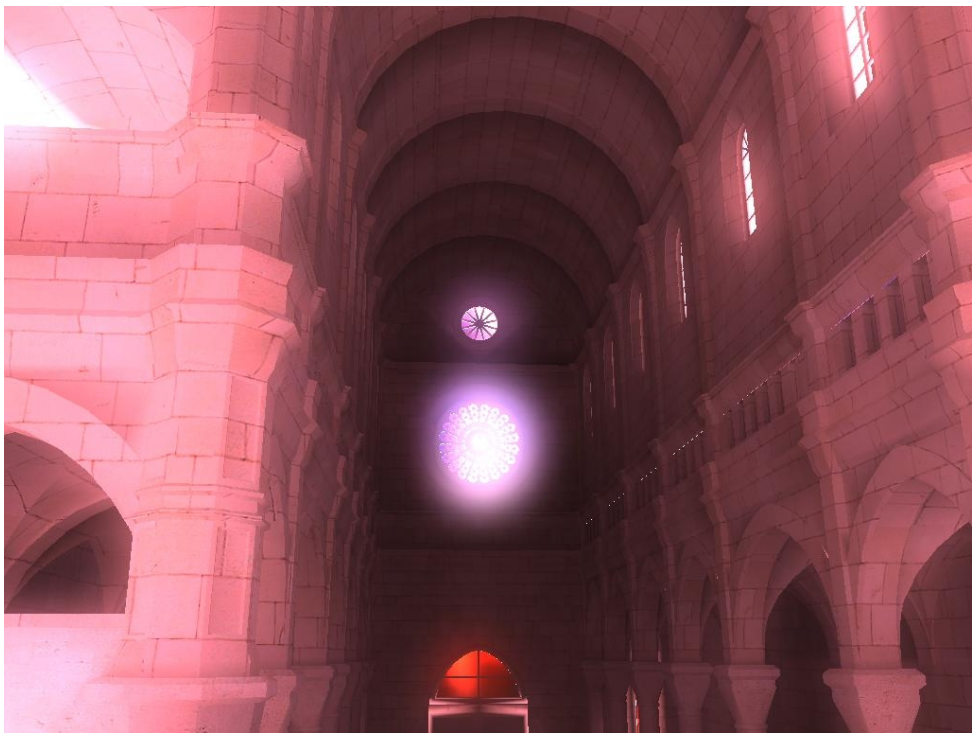

Figure 1: Precomputed Radiance Transfer in combination with High-Dynamic Range Rendering.

supplied transformation. This normal and the eye-vector are then used to calculate a reflection vector and fetch the cubemap.

Some extensions to the method described in the Cg Manual [2] where implemented: At first, the effect is explicitly programmed using standard library functions instead of `reflect_eye_dp3x3`. This allows the following flexibility: There are two cubemaps fetched instead of one, a "chrome" cubemap providing an ambient term, and a high dynamic range cubemap. The first cubemap is necessary to provide the metallic look. Another extension is to calculate the luminance of the high dynamic range cubemap, this luminance is then multiplied with the metal color. This allows to account for the property of metal to only reflect in its own color.

Figure 7 shows all these effects in action.

6.9 Water

Some effects need to be combined in order to create a realistic looking water. This section describes the effects that are implemented in Insomnia.

6.9.1 Water Normal Vectors

The first and most basic concept is to calculate normal vectors for every pixel on the water-surface that create visually convincing perturbations. We have chosen the method proposed in [3]. Therefore one normal map is fetched multiple times with different texture coordinates. The used texture-coordinates are moved by a small amount in every frame. All fetched normals are then combined. This simple method creates normals for water-surfaces without visible repetitions.

6.9.2 Fresnel Terms

Fresnel terms describe the percentage of light refracted and reflected on material borders depending on the refractive indices and the incident angle of the light. Refer to [9] for a definition of the terms. To make those terms applicable to real-time graphics, a lookup texture is precalculated. This texture allows to determine the coefficients with a simple texture fetch, using the incident angle of the light as texture coordinate. Another optimization implemented in our demo is to pull out the calculation of the incident angle into the texture, as described in [9]: Now the texture is fetched with the dot-product of the incident vector and the normal as texture coordinate. The texture contains the arcus cosinus function to calculate the incident angle from the dot-product.

6.9.3 Planar Reflections, Fake Perturbations

Planar Reflections are achieved by rendering the scene mirrored around the water plane [4] into a

Figure 2: Note the indirect lighting achieved with Precomputed Radiance Transfer.

texture. This texture is now fetched when rendering the water surface using projective texturing. Fake perturbation of the reflection is obtained by offsetting the calculated texture coordinates with an arbitrary scaled version of the water surface normal Vector.

As mentioned in [3] one important step during perturbation is masking to prevent artifacts. Therefore heuristics have to be used to decide whether an perturbed texture coordinate is valid. The following heuristic is implemented in our demo: During rendering of the mirrored reflection map, all areas that are under the water plane are marked with negative alpha values. Those areas must not be used for reflections, and are thus masked out.

6.9.4 Fake Refractions

This effect works very similar to fake perturbed reflections. However this time the original, non-mirrored, scene is used as refraction map. Therefore the scene is rendered normally at first. Then the offscreenbuffer is copied to the refraction texture. This texture is then used during water-rendering. To get fake refractions, the texture coordinates are perturbed similar to fake reflections.

Again, masking is important. This time we have to mask out objects that are in front of the water plane, as seen from the camera. This is achieved by comparing the distance to the water plane with the distance stored in the alpha-channel of the refraction-texture. If the first distance is larger, no perturbation is used. Additionally the texture coordinates are clamped to the border.

Figure 8 shows all effects described thus far in action.

6.9.5 Raytraced Refractions

For even more convincing refractions realtime raytracing is used. Please refer to the accompaining paper `"Phys_Refractions_shader_documentation.pdf"` by Markus Lipp for a description. Note that this paper was not specifically written for this demo, it was written in summer-term 2005 for the lecture "Virtual Reality".

Additionally to the method described in the mentioned paper, one iteration of sphere tracing [6] was added to the shader: This allows for a much faster convergence.

6.9.6 Raytraced Caustics

The main idea is the same as for raytraced refractions. The idea was extended by Markus Lipp in the following way: The water plane is rendered in an orthogonal view from above. For every pixel of the water plane, the pixel shader now calculates the refracted ray direction from the lightsource. This ray is now traced to the ground. When the

Figure 3: Soft Shadows, Normal Mapping and Parallax Mapping on the floor.

ground is hit, the position of the hit is returned by the fragment shader, thus generating a texture containing hits of light rays.

The position texture is now used in an additional pass. Therefore a displaylist containing as many quads as there are pixels in the position texture is rendered. Every quad contains texture coordinates pointing to distinct coordinates in the positions texture. A vertex shader now fetches the corresponding position for every quad using vertex texture fetches. Now the shader moves the quad to the position of the ray hit. Thus the quad is rendered at the position of the hit. Now a texture containing a gauss-bell is laid on every quad. When we now use blending during rendering we get an approximation to photon mapping, and thus get raytraced caustics in real time.

This caustic texture is now also fetched during the rendering of the water surface using raytraced refractions.

Figure 9 shows both raytraced caustics and refractions in action.

6.9.7 Raytraced Underwater Shadows

This is not really a effect on its own. It is a sideproduct of raytraced caustics: As we trace the rays we automatically get shadows under water for regions not visible for refracted light rays.

6.10 High Dynamic Range Rendering

6.10.1 Float Precision Everywhere: Textures, Effects, Buffers

When talking about High Dynamic Range Rendering (HDR) it is important to realize that this effect is not restricted on postprocessing-effects like bloom and tonemapping. In fact, those effects only make sense, when the image presented in them actually contains a high dynamic range scene. To achieve this high dynamic range in the rendered scene, multiple measurements where taken in Insomnia for the Asteroid scenes: At first, a high dynamic range cubemap containing stars was designed. Spherical harmonics coefficients where extracted from this cubemap. This allows a high dynamic range rendering of objects, when additionally PRT is employed. Further some objects, like lava, have a high dynamic range texture applied to them.

However, this is still not sufficient: Every material-effect and postprocessing effect (for the asteroid-scenes) creates high precision output from high precision input. This requires floating point offscreen buffers to be used thorough the pipeline. Only by providing high precision thorough the pipeline, bloom and tonemapping make sense. Let us now look at those two effects.

Figure 4: Per-Pixel Lighting with colored lightsources.

6.10.2 Bloom

The bloom effect itself is straightforward to implement. At first the current offscreen-buffer is downsampled by rendering it on a full-screen quad into a low resolution buffer. Then we implemented a two-pass bloom: In the first pass, the picture is convoluted with a vertical gaussian blur kernel, in the second pass the output from the first pass is convoluted with a horizontal blur kernel. The blur kernel itself is a fragment shader with coefficients and texel offsets compiled as literal constants. It consists of a simple loop fetching the current position with texel offsets and then multiplying them with the coefficients.

After the blurring, the blurred low-resolution texture is added to the high-resolution offscreen buffer. It is important to note that neither the values of the low-resolution nor the values of the high-resolution textures are clamped in any way: The blurred texture is *not* clamped to contain only bright parts. This is not necessary, as real high dynamic input images are used, therefore the blurring automatically gets more pronounced for bright areas near to dark areas.

6.10.3 Exponential Tonemapping

Tonemapping is the process of creating a low-dynamic range image for display-devices from a high dynamic image. Refer to [1] for a good overview. We have chosen exponential tonemapping for its simplicity. Therefore, after the combination of the high-resolution scene with the blurred low-resolution scene, the luminance l_s is calculated. According to the exponential tonemapping formula a tonemapped luminance l_t is calculated. By multiplying the high dynamic range color with l_t/l_s we get our tonemapped color. We did not implement time-dependency or automatic exposure adjustment.

6.11 Depth of Field

Our implementation follows an proposal by ATI [5]. We have chosen this method, as it allows full flexibility for specifying a depth-¿blurriness function via a texture map, without additional overhead for the unsharp/sharp/unsharp scenario. Further no leaking artifacts occur, as every filter tap is checked whether it would contribute to leaking.

6.12 Polygonal Volumetric Fog

This effect is described in [8]. Our implementation uses the path for shader model 2.0 cards, as proposed in [8]. To gain more control on the look of the fog, a small addition was implemented: The lookup texture also contains opacity values in the alpha channel. This value is used to linearly interpolate between the fog color and the scene color. Thus we can not only control the color of the fog, but also the opacity using one lookup texture.

To generate the waves of the fog seen in the

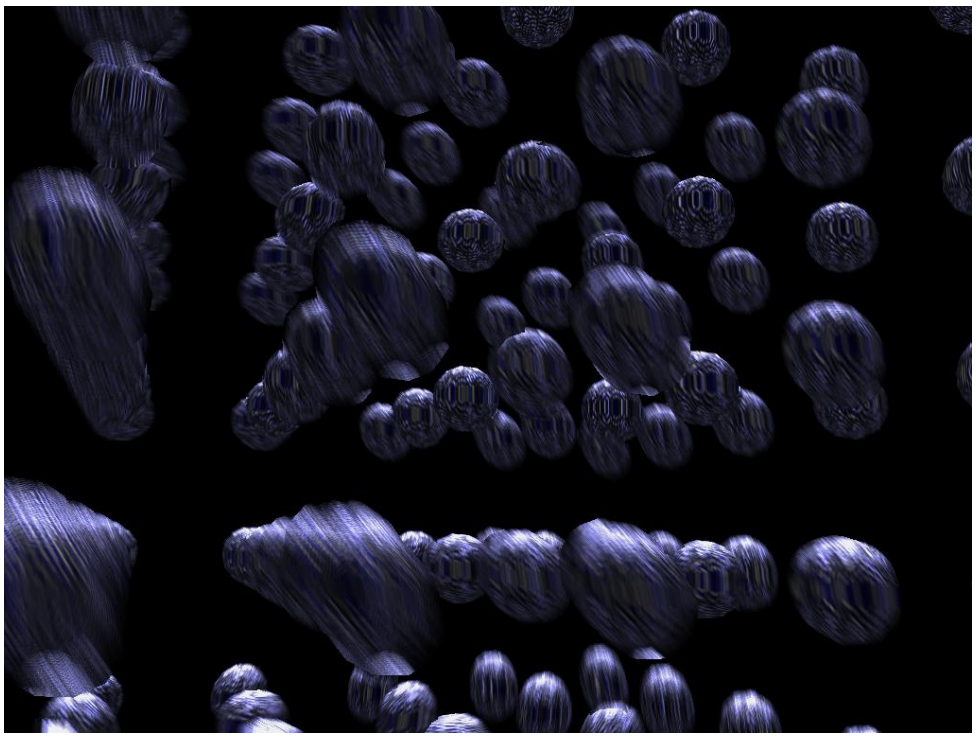

Figure 5: Object-Based Motion Blur. Note that every object gets a different amount of blurring.

demo, multiple sinus waves are overlaid in the vertex-shader to offset the vertices along their normal vectors.

Figure 12 shows our implementation of polygonal volumetric fog in action.

6.13 Heat Haze

This effect is described in [7]. However one major difference was implemented by Markus Lipp into our implementation: The amount of perturbation depends on the way-length a ray travels through a special heat haze mesh. The main idea for this empirical perturbation calculation is that the amount of perturbation is proportional to the way-length a ray travels through medium with another refraction index. The way-length is calculated using the same method as in polygonal volumetric fog.

This perturbation estimation method has major advantages compared to the approach for determining perturbations in the original presentation: At first, it is possible to enter and leave heat-haze zones with smooth transitions. Further it is possible to define arbitrary heat-haze meshes without worrying about intersections with the scene geometry, as these intersections are automatically dealt with using the polygonal volumetric fog algorithm. This creates a high flexibility for artists when defining heat-haze zones.

7 Effects Index

This section provides a mapping from scenes to used effects.

- Intro
  - True Object-Based Motion Blur
  - Faked Fire
- Cathedral
  - Precomputed Radiance Transfer
  - Normal Mapping
  - Parallax Mapping
  - High Dynamic Range Rendering: Bloom, Exponential Tonemapping
- Statue
  - Penumbra-Based Soft Shadows
  - Normal Mapping
  - Parallax Mapping
- Asteroid Scene 1 High Hopes
  - Precomputed Radiance Transfer (Asteroid)
  - Bump-Reflection Mapping (Satellite)
  - High Dynamic Range Rendering: Bloom, Exponential Tonemapping
- Asteroid Scene 2 Cold Welcome

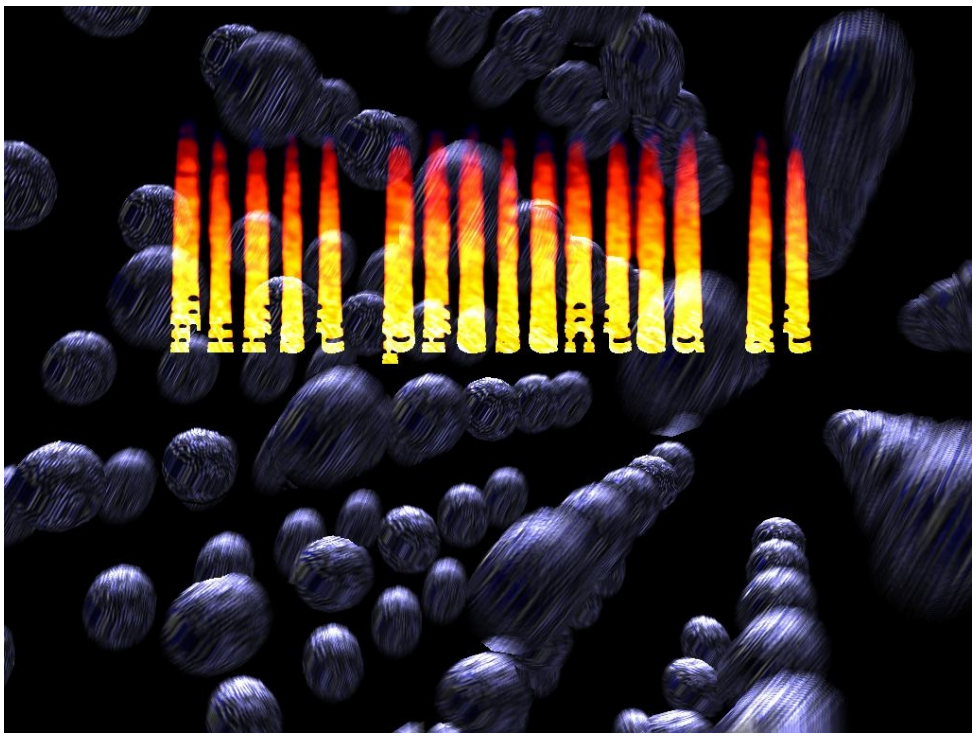

Figure 6: Fire simulated on the GPU.

  - Polygonal Volumetric Fog
  - Precomputed Radiance Transfer (Asteroid)
  - Normal Mapping (Asteroid)
  - Parallax Mapping (Asteroid)
  - Bump-Reflection Mapping (Satellite)
  - High Dynamic Range Rendering: Exponential Tonemapping
- Asteroid Scene 3 Isolated
  - Depth Of Field
  - Precomputed Radiance Transfer (Asteroid)
  - Normal Mapping (Asteroid)
  - Parallax Mapping (Asteroid)
  - Bump-Reflection Mapping (Satellite)
  - High Dynamic Range Rendering: Exponential Tonemapping
- Asteroid Scene 4 Pure Joy
  - Water
  - Fresnel Terms
  - Planar Reflections
  - Fake Refractions
  - Precomputed Radiance Transfer (Asteroid)
  - Normal Mapping (Asteroid)
  - Parallax Mapping (Asteroid)
  - Bump-Reflection Mapping (Satellite)
  - High Dynamic Range Rendering: Exponential Tonemapping
- Asteroid Scene 5 Raytraced Fun
  - Water
  - Fresnel Terms
  - Planar Reflections
  - Raytraced Refractions
  - Raytraced Caustics
  - Raytraced Underwater Shadows
  - Precomputed Radiance Transfer (Asteroid)
  - Normal Mapping (Asteroid)
  - Parallax Mapping (Asteroid)
  - High Dynamic Range Rendering: Exponential Tonemapping
- Asteroid Scene 6 Hot Parting
  - Heat Haze
  - Precomputed Radiance Transfer (Asteroid)
  - Normal Mapping (Asteroid)
  - Parallax Mapping (Asteroid)
  - Bump-Reflection Mapping (Satellite)

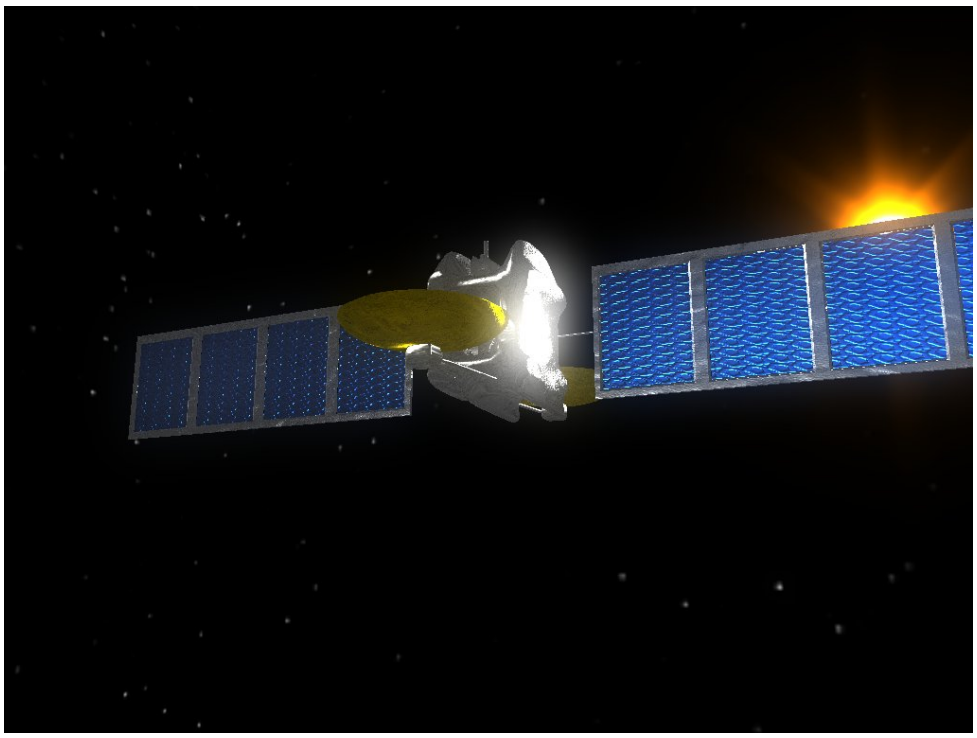

Figure 7: Bump-Reflection Mapping on the satellite. Note that the sun is reflected on the metal body, while the solar collectors reflect the blue nebula behind the camera. Additionally the metal only reflects in its own color, while the plastic solar collectors retain some of the color information from the cubemap.

  - High Dynamic Range Rendering: Bloom, Exponential Tonemapping

- Credits
  - Precomputed Radiance Transfer (Asteroid)
  - High Dynamic Range Rendering: Bloom, Exponential Tonemapping

8 External Assets

Some of the art asset was taken from free sources. However, it may be possible that some parts of the art asset are from a non-free sources by accident. If you find such material belonging to you, please let us know and we will remove it from the demo.

Here is an index of external materials:

Satelite: The base mesh is from baumgarten enterprises, `http://www.baument.com/archives.html`, modified and textured by Markus Lipp.

Lave textures: From Thorsten Willert, `http://www.the3dstudio.com/`, it was modified and normal-mapped by Markus Lipp.

The Heat-Haze perturbation texture is taken out of a presentation from ATI technologies.

Sand: From a free source from the web, we do not remeber where it originated.

The HDR textures in cathedeal demo are from Paul Debevec, `debevec.org`.

References

[1] Alessandro Artusi. *Real Time Tone Mapping*. PhD thesis, TU Vienna University of Technology, April 2004.

[2] NVidia Corporation. *Cg Toolkit - Users Manual*, chapter Bump Reflection Mapping, pages 196–199. September 2005.

[3] Tiago Sousa (Crytek). *GPU Gems 2*, chapter Generic Refraction Simulatio. March 2005.

[4] M. Pauline Baker Donald Hearn. *Computer graphics (2nd ed.): C version*. Prentice-Hall, Inc., 1996.

[5] John Isidoro Guennadi Riguer, Natalya Tatarchuk. *Shaderx2: Shader Programming Tips & Tricks*, chapter Real-Time Depth of Field Simulation. Wordware Publishing, 2003.

[6] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

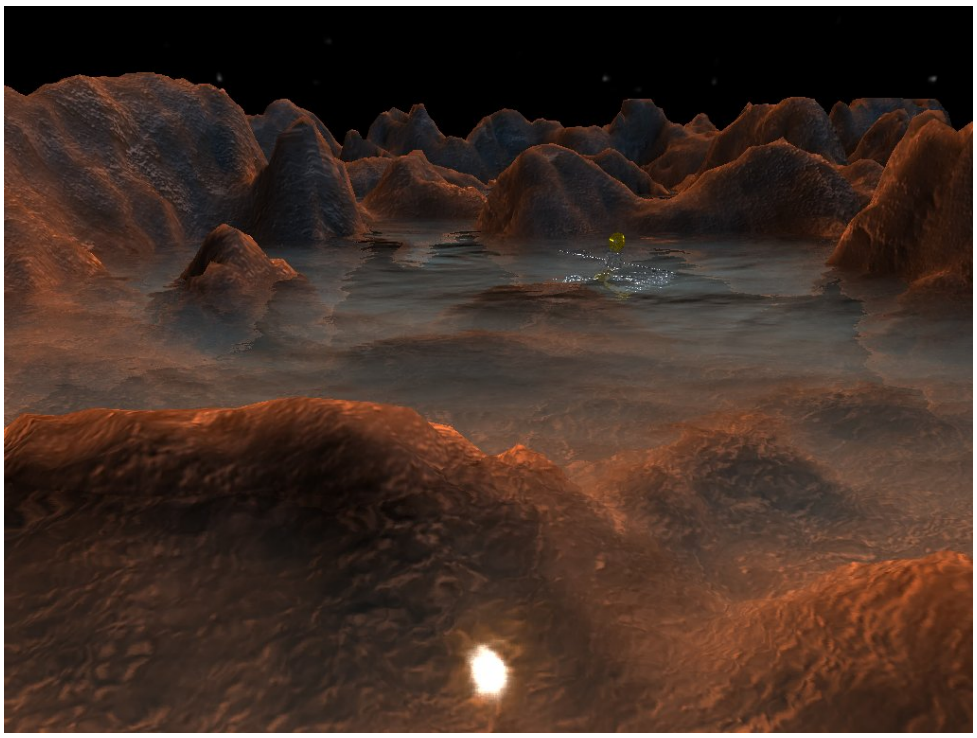

Figure 8: Water scene showing fresnel terms, perturbed planar reflections and fake refractions. Note that due to the fresnel terms the reflections far away are more pronounced as the incident angle is smaller.

[7] Chris Oat. Real-time 3d scene post-processing, heat and haze effects. Technical report, ATI Research, 2004.

[8] NVidia SDK. Fog polygon volumes. Technical report, NVIDIA, Santa Clara, CA, 2004.

[9] Matthias Wloka. Fresnel reflection. Technical report, NVIDIA, Santa Clara, CA, 2004.

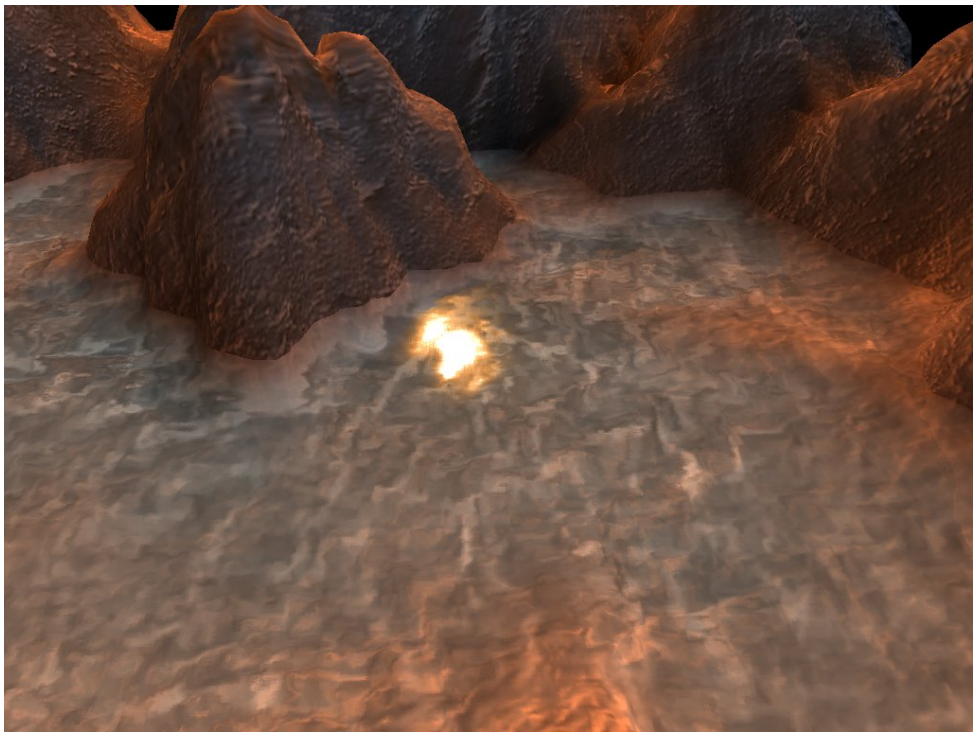

Figure 9: Water scene showing fresnel terms, perturbed planar reflections, raytraced refractions and raytraced caustics.

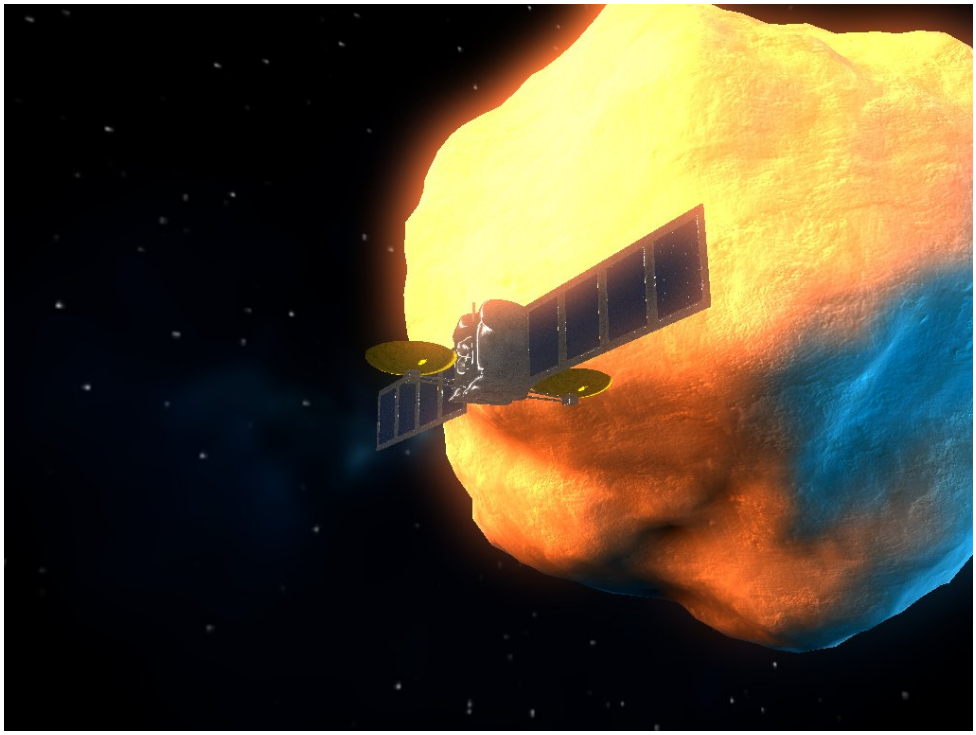

Figure 10: Scene showing bloom and exponential tonemapping. Note that the high dynamic range of the scene is achieved using a floating-point cubemap, high precision reflections on the satellite and PRT on the asteroid.

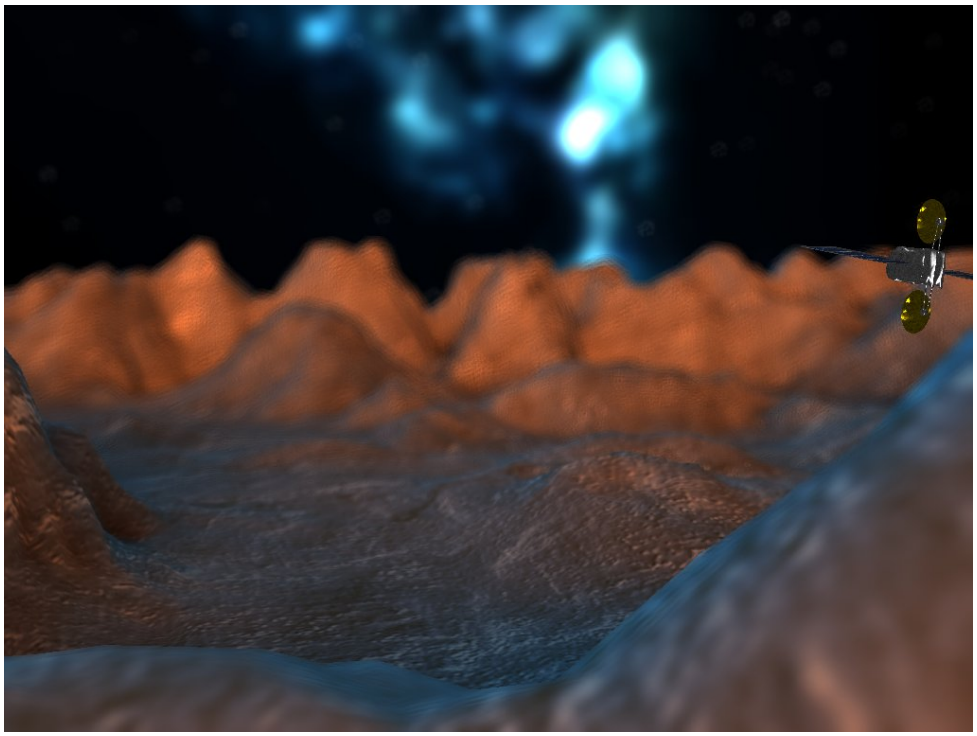

Figure 11: Scene showing Depth of Field. Note that both near and far away objects get blurred, only a small depth area is in focus. Additionally there is no leaking from the sharp satellite into the blurred background.

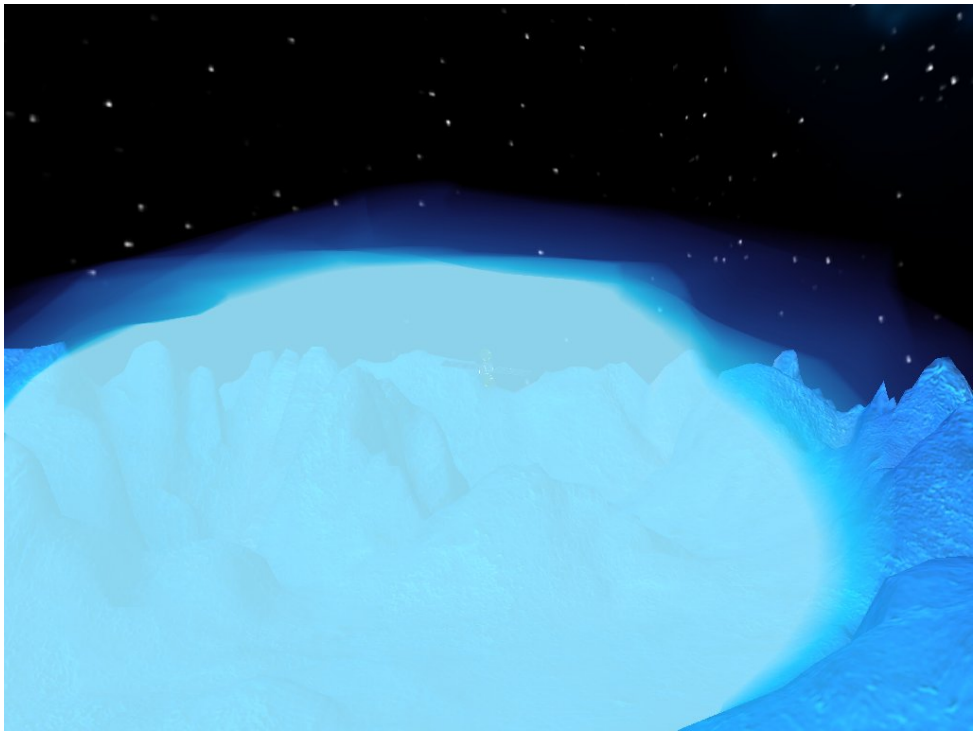

Figure 12: Scene showing Volumetric Fog.

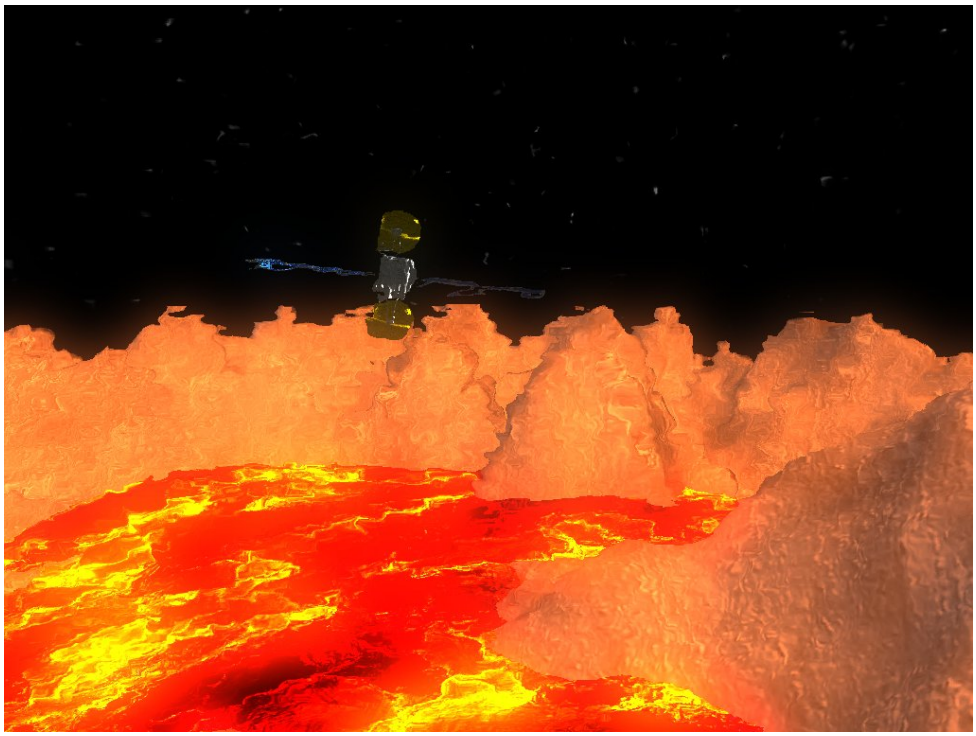

Figure 13: Scene showing Heat Haze. Note that our algorithm allows very fine control of the heat haze: Areas over lava have high perturbations, while the rock in the lower right corner has almost no perturbation.