

Real-Time Rendering: RandomSpaceScene

(Tested on Ghost AMD Graphicscard)

Submission 2

186.140, winter term 2016

Harald Scheidl (0725084)

Thomas Pinetz (1227026)

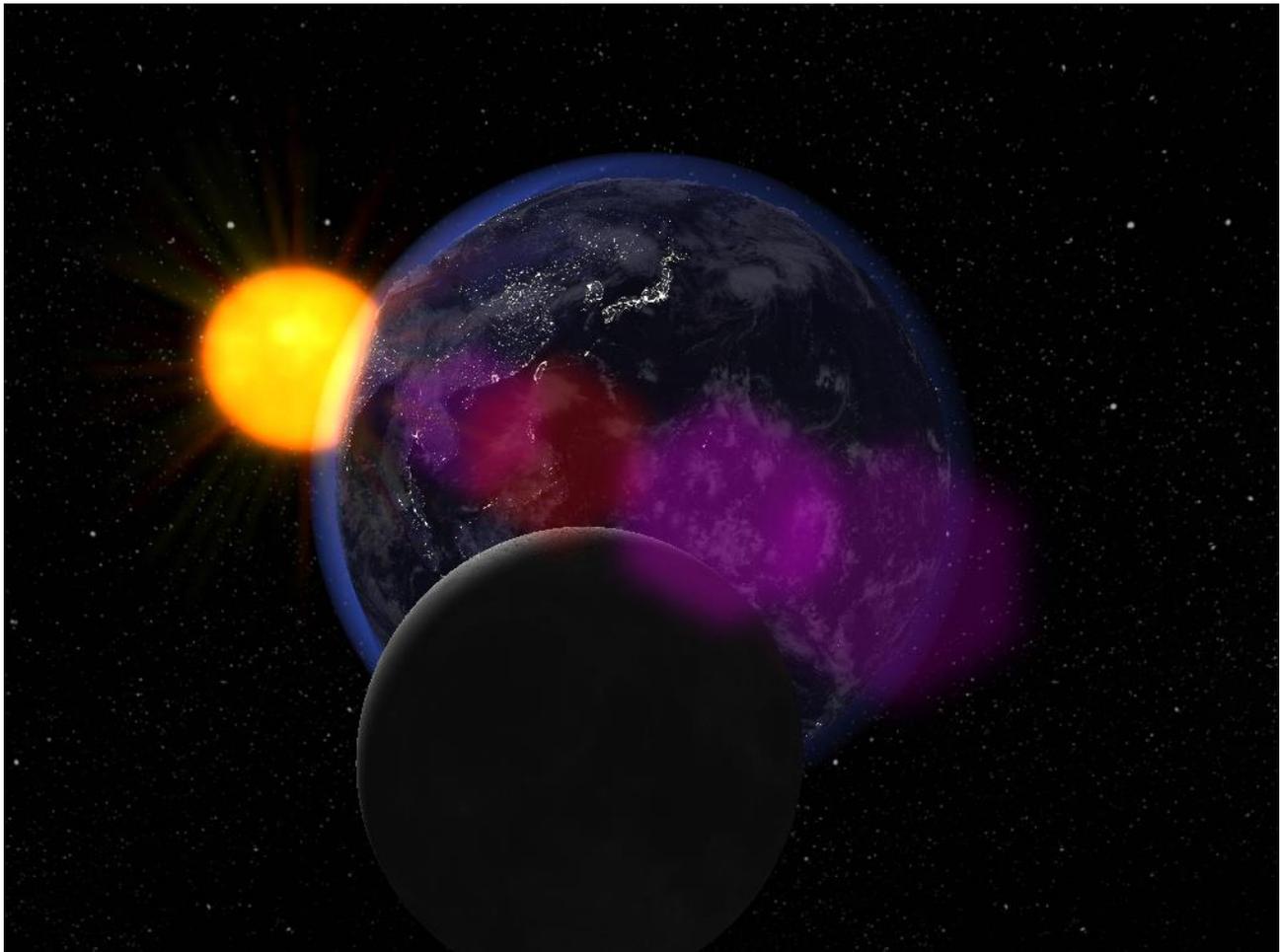


Fig.: screenshot from our demo

Effects (after feedback from submission 0)

1. Atmosphere: the earth is surrounded by an atmosphere [ATM]
2. Displacement mapping: mountains on the earth are shown as 3d objects [DM{1,2,3,4}]
3. Bloom: the bright light causes a blurred region around the sun [WRIGHT]
4. Lens flare: the sun causes a lens flare effect on the screen [LF]

Controls

Select object: e (earth), s (sun), m (moon)

Restart animation: a

camera up and down: up/down arrow keys

zoom: keypad + and -

pause: space key

Configuration

see file RandomSpaceScene.ini

[MAIN]

Fullscreen=TRUE # render into full screen window?

Width=1024 # in window mode: width in pixel

Height=768 # in window mode: height in pixel

GLDebug=TRUE # show OpenGL debug info in console (Decreases FPS!)

SensitivityZoom=0.1 # tweak sensitivity of zoom command

SensitivityUpDown=0.01 # tweak sensitivity of up and down command

WaitForVSync=TRUE # wait for VSync of monitor to refresh (Avoids tearing. Decreases FPS!)

Effects: Where to find them

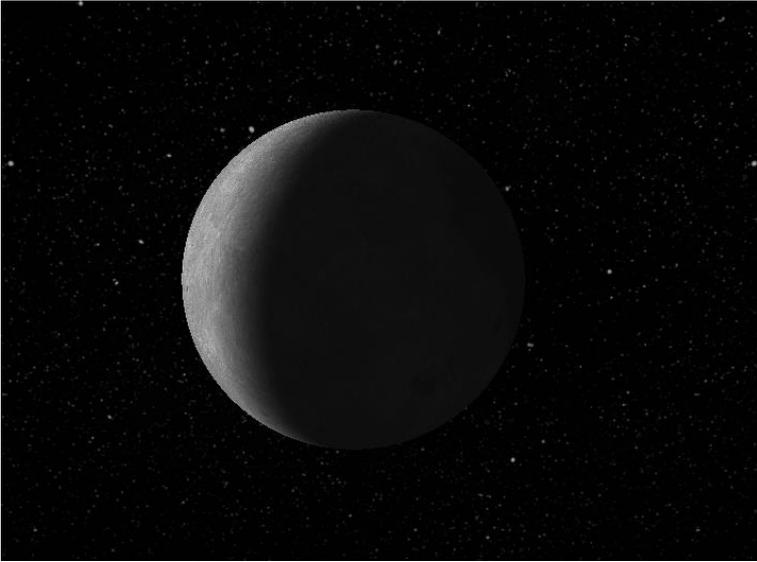


Fig.: Basic lighting and texturing

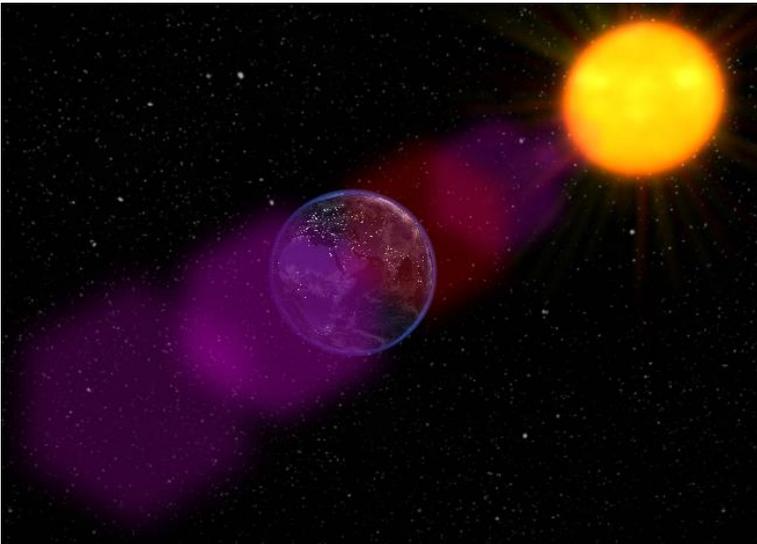


Fig.: Lens flare



Fig.: Bloom



Fig.: Atmosphere

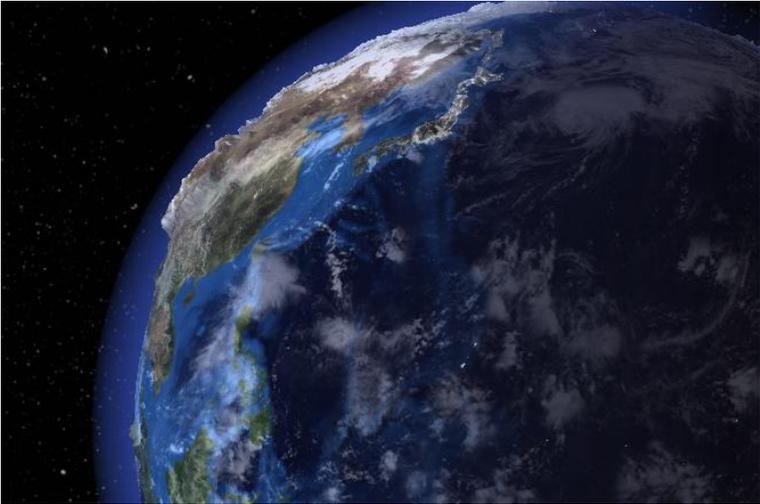


Fig.: Displacement Mapping

Effects: Implementation

Lens flare

This is a screenspace effect. For the bloom effect, we have a framebuffer-object (FBO) in which only the sun is rendered. This FBO is also used as a lookup for the lens flare effect: the position of the sun on the screen is calculated and the FBO is then sampled at this position to check if the sun is visible or if some other object is in front of the sun (e.g. the earth).

If the sun is visible, the lens flare effect is enabled: using the direction from the sun to the screen center, we blend multiple lens flare textures (see figure below for an example of such a texture) into the final scene.

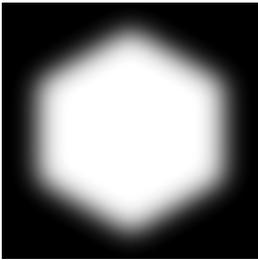


Fig.: one of the textures for lens flare

Bloom

This effect occurs around bright light sources. In our scene, this effect is used for the sun. The sun is rendered into a FBO. This FBO is then convoluted with a Gauss kernel and rendered into the next FBO. This can be regarded as a pipeline where each output of one stage is the input for the next stage. Finally, all FBOs are blended into the final scene.

To speed-up the convolution, we separate it into x and y direction. Additionally the FBOs are getting smaller and smaller when moving along the pipeline. Stage 2 and stage 7 are shown below. The reason for the distorted look of stage 7 is the already discussed separation into x and y direction of the convolution, such that in each second stage we get this distorted look.



Fig.: Stage 2, width is 512px

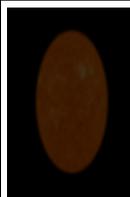


Fig.: Stage 7, width is 64px

Atmosphere

The basic idea of this atmosphere algorithm is to calculate the width of the atmosphere at each pixel. To do this, the depth value (z-value) of the front side and the back side of the atmosphere and the front side of the earth are rendered into different textures of a FBO. This can be done by rendering a sphere (see figure, blue object) with different settings for back-side-culling and different sizes (the atmosphere is a factor of 1.1 bigger than the earth).

Then, the width of the atmosphere is simply the difference between the distance to the front side and the distance to the back side (to keep things easy, we use the depth value before homogenization of the coordinates). The back side of the atmosphere is either the atmosphere or the earth, depending on which one is closer to the camera.

Finally, we render the atmosphere by rendering a sphere around the earth and blend the atmosphere colour (which depends on the atmosphere width as explained above) into the scene.

To also take the shadow of the earth into account, we render a shadow object too. Therefore, a cylinder is rendered (see figure, red object). We then calculate the width of the atmosphere at each pixel by taking the width of the lit atmosphere and subtracting the width of the shadowed atmosphere. The width of the shadowed atmosphere is simply the width of the shadow object clipped by the atmosphere and earth depth values.

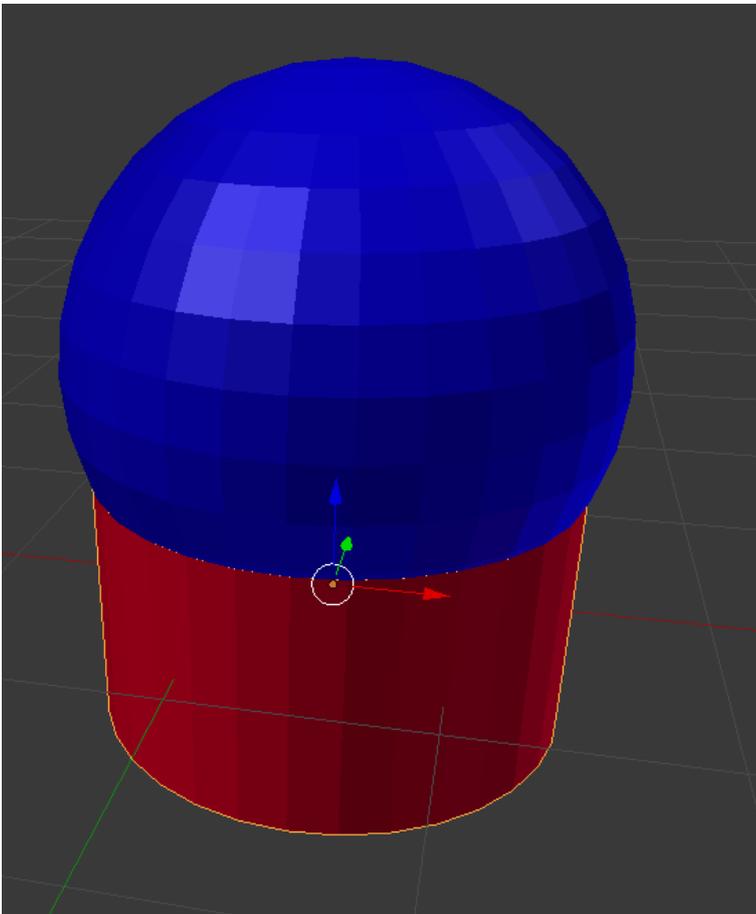


Fig.: Sketch of the shadow object (red) and the atmosphere object (blue)



Fig.: atmosphere on dark (violet) side and on bright side of the earth (yellow)

Technically, all depth values are stored in FBOs. The depth values are taken before applying the homogenization, therefore there is no need to linearise the depth-values as shown in the original paper.

It is important to use FBOs with high resolution textures attached, e.g. the usual 256 colour textures just don't work. We therefore use 32bit float textures which work fine for the fine-grained depth-value arithmetic.

Displacement Mapping using Tessellation

Displacement Mapping in its original Form [DM1] uses a texture to change the geometry of any object. In our case we want to change the geometry of our earth to reflect changes in the height of the surface. To archive this a common simplification is to use the surface normal as a direction for displacement. This gives us the equation: $p' = p + d(p)n$, where p is any point on the surface, $d(p)$ is the value of the displacement map at this point and n is the surface normal. For our purposes a unit sphere was used as base shape. Therefore, the normal are the points itself and we can therefore rewrite the equation into: $p' = p * (1 + d(p))$. In our code we added a scaling variable α to archive visually pleasing results. The final equation is: $p' = p * (1 + \alpha * d(p))$.

The problem with applying just this equation in the vertex shader is that a lot of vertices are required to reduce the impact of artefacts like aliasing. In many cases those vertices do not exist and nowadays tessellation is used to create those vertices on the fly. Tessellation is split into three parts, namely the Tessellation Control Shader, Tessellator and the Tessellation Evaluation Shader. The Tessellation Control Shader controls the amount of subdivision of a face. [DM2]

The impact of the subdivision can be seen in the following figure:



Inner/Outer Division 1



Inner/Outer Division 4

Fig.: Subdivisions. For this screenshot atmosphere was moved a bit away from earth to better see tessellation.

As can be seen in the figure above the subdivisions help create more vertices and therefore create a more fluid model than with less subdivisions. However, more vertices have to be processed with

higher subdivisions and it therefore costs computing power to create higher subdivisions. We used two uniform variables to control the inner subdivisions and the outer subdivisions. Then the Tessellator creates new vertices, which are then poured into the evaluation shader. The Tessellation Evaluation Shader gets barycentric coordinates to determine on which vertex we are in a given face. Then the displacement is calculated and applied to the newly generated vertex. This new vertex is then passed on to the fragment shader, which does the same calculations as without displacement mapping, like lighting calculations and texture mapping. For the syntax of creation and applying of tessellation [DM3] was used.

References

[ATM] Josth, Radovan. "Real time atmosphere rendering for the space simulators." *Proceedings of the 9th Central European seminar on computer graphics, CESC G*. Vol. 2005. 2005.

[LF] Akenine-Möller, Tomas, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[WRIGHT] Sellers, Graham, Richard S. Wright Jr, and Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley, 2013.

[DM1] Cook, Robert L., Loren Carpenter, and Edwin Catmull. "The Reyes image rendering architecture." *ACM SIGGRAPH Computer Graphics*. Vol. 21. No. 4. ACM, 1987.

[DM2] The Little Grasshopper. Graphics Programming Tips. 29.12.2016, accessed at: <http://prideout.net/blog/?p=48>.

[DM3] OGL. Modern OpenGL Tutorials.
<http://ogldev.atspace.co.uk/www/tutorial30/tutorial30.html>

[DM4] GPU gems 2, Per-Pixel Displacement Mapping with Distance Functions, accessed on 26.12.2016, accessed at: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter08.html