Mishungerstanding

(or a game-dream come true by the hands of

Harald Scheidl, 0725084, 066 932

and

Tomas Weissenböck, 0800776, 033 534)

Documentation of the 2nd Submission

Brief overview

Features

The main goal of our game is to survive the hunger of the blackhole. If you get hit by an asteroid you're dead. You can shoot lasers once per second and destroy an incoming asteroid. But be aware of the suns and the black hole too!

The game also features levels (look at INI settings) where you have to fulfill certain requirements to pass each, be that surviving for X seconds or shooting Y asteroids or a combination of both. It's really easy to add additional levels based on that simple win/lose condition.

Regarding high scores: The player name is set via the INI. Also make sure to close the game via ESC, else your score is not recorded.

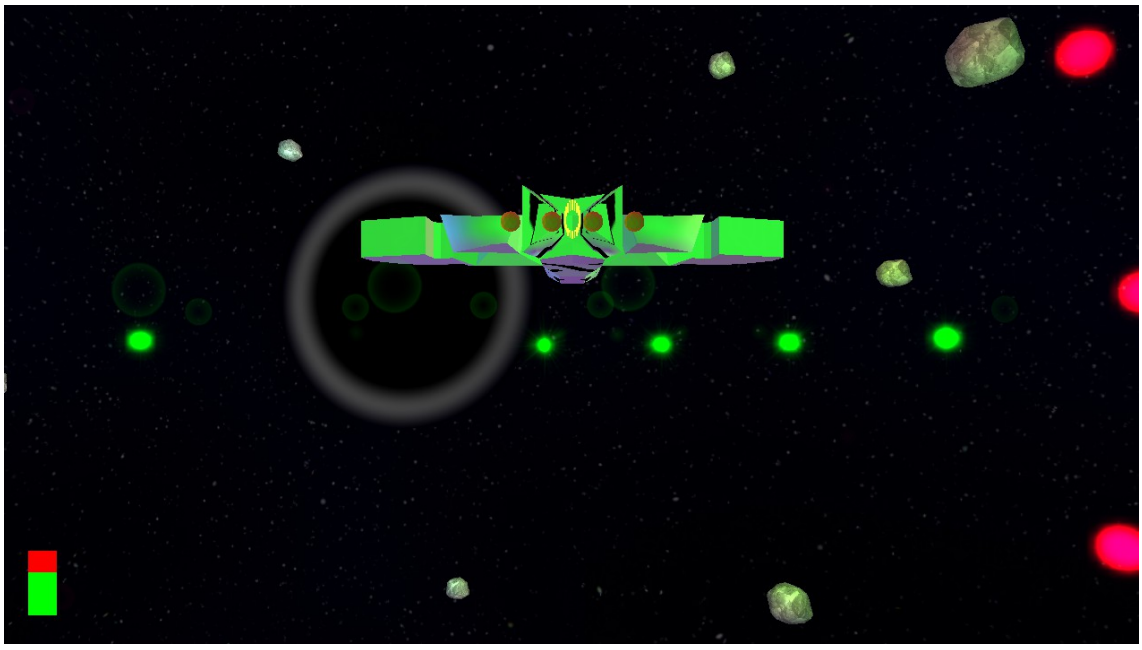

Figure: screen shot of our game

Controls

- ESC: quit game
- Mouse left/right/up/down: steer ship (yaw and pitch)
- Left/right arrow key: steer ship (roll)
- Up/down arrow key: speed up/down
- Space: shoot laser weapon
- F1: help
- F2: show FPS (filtered) and frame time (unfiltered)
- F3: fill or line renderer
- F4: texture quality (sampling mode)
- F5: mip mapping on/off and sampling mode
- F8: frustum culling on/off
- F9: transparency (GUI energy level) on/off

INI settings

A very basic INI reader is implemented, supporting data types string and int. For boolean values, we use the values 0 and 1 and interprete them as false and true. Just add the line OptionName=Value to adjust one of those values:

- Width: width of the game in windowed mode (int)
- Height: height of the game in windowed mode (int)
- DumpGBuffer: dump the GBuffer and BrightPass buffers to ./out (bool)
- MonitorPasses: log performance of each pass (bool)
- LiveTimeSec: you're dead after this time if you don't shoot an asteroid (int)
- TestAsteroids: enable test asteroids forming an axis of coordinates (bool)
- Invulnerable: enable to live forever, good for looking around (bool)
- FullScreen: show game in fullscreen mode (bool)
- LenseFlareMinVal: adjust lense flare intensity(int)
- LenseFlareMaxVal: adjust lense flare intensity (int)
- GLDebugMessages: show GL messages on stdout (bool)

- PlayerName: name for high score (string)
- DontMoveAsteroids: don't apply gravity to asteroids (bool)
- LevelsEnabled: enable game levels (bool)

Lighting

Lighting is implemented through our Deferred Pipeline [OGL] like so:

We have multiple suns as light sources all of which are point lights. There is one spot light which will be described in more detail in the Effects section.

The material properties are set as uniforms in an entity and used in the entity's fragment-shader to render a special texture to the offscreen fbo during the geometry pass.

These values are then incorporated into the actual lighting in the lighting pass.

As we use deferred shading (meaning one geometry pass and one lighting pass) there is no real way to make use of different lighting algorithms for our materials, so all of them just use the standard phong shading which should be enough for our purposes.

We also have an ambient pass at the end as the point lights alone wouldn't lit the scenery enough but we didn't really need multiple directional lights so just lighting everything in one go is certainly enough.

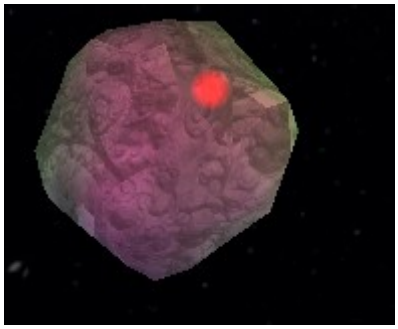

Figure: superposition of multiply light sources: green and red suns (point light), laser (spot light)

Effects

Bloom

This effect blurs bright regions. You can see bloom when looking at our suns, and the blur can be seen when a sun is partly occluded.

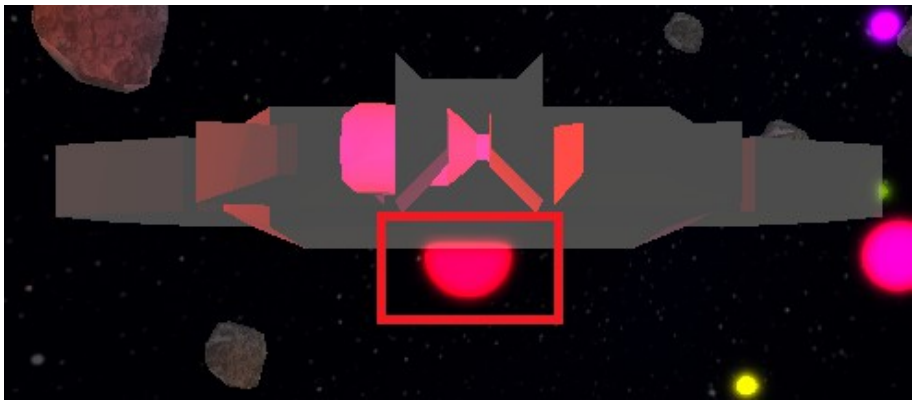

Figure: bloom effect in our game

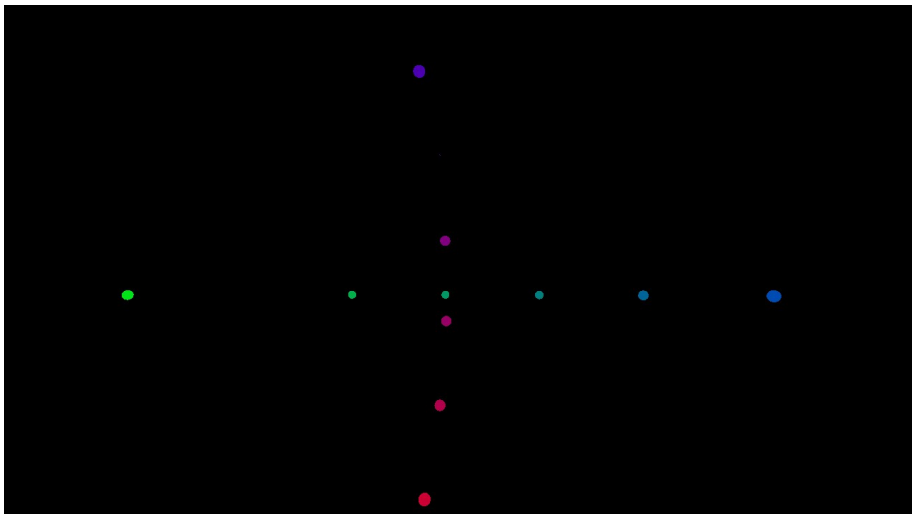

Figure: all light sources as seen from camera

We start by drawing all light sources in a buffer [Wright]. Then we continue by blurring (convolution with Gaussian kernel) this buffer several times into smaller and smaller buffers. For better performance, we separate the 2D kernel into two 1D kernels [Szeliski].

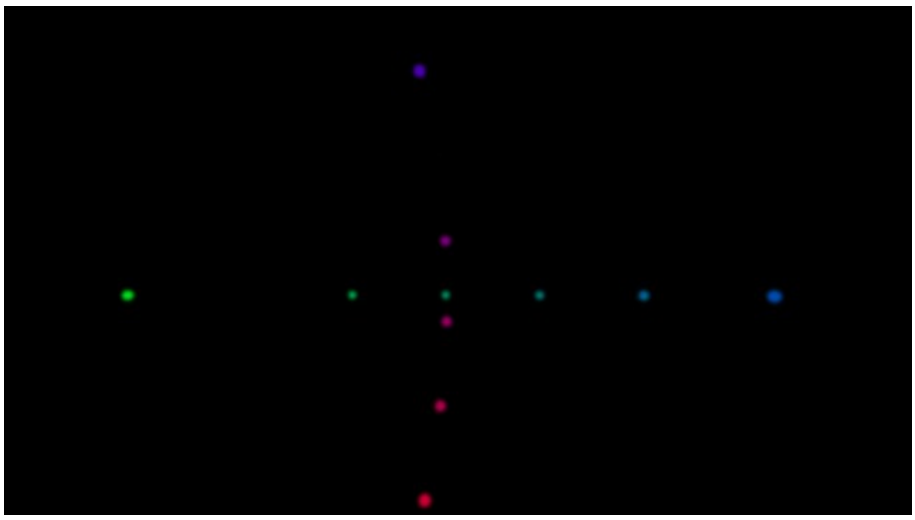

Figure: first blurred and downsampled version of light sources

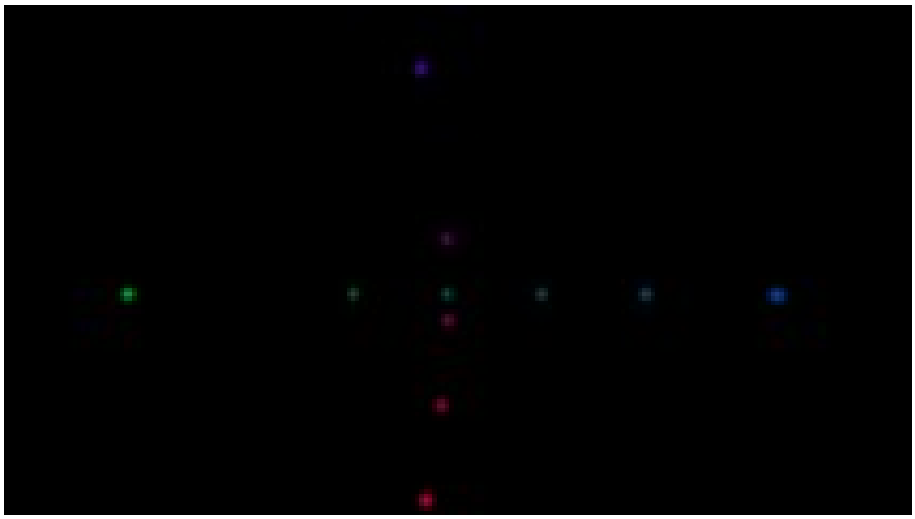

Figure: fourth blurred and downsampled version of light sources

Finally we take those blurred images and blend them into the final image.

Deferred Pipeline

To improve the performance of the light calculations, we perform deferred shading [OGL]. Instead of calculating the final image in one step, we first draw all objects into buffers, which will later on be used for the lightning.

Object information will be written into four buffers, called the "GBuffer":

- position: world position of pixel
- normal: normal vector of pixel
- parameter: some parameters needed for lightning calculation
- diffuse: the diffuse color, e.g. texture color

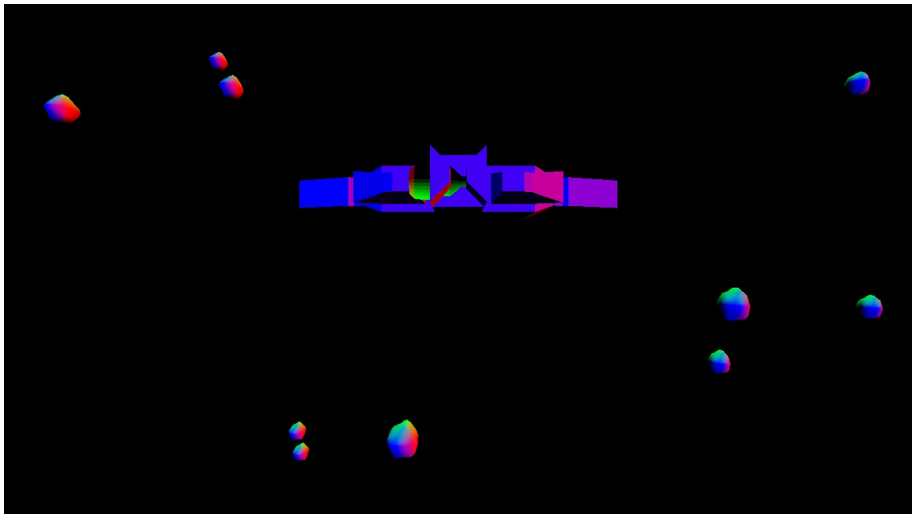

Figure: the buffer containing the normal vectors of all objects

After all geometry is drawn into those buffers, lightning calculation is done by simply taking the values from each texture (position, normal, color, ...) to calculate the final, shaded pixel value.

Lense Flare

This effects occurs when looking in the direction of the sun through a camera.

Just fly near to a sun and try to center the sun at least a bit, then you'll see the effect.

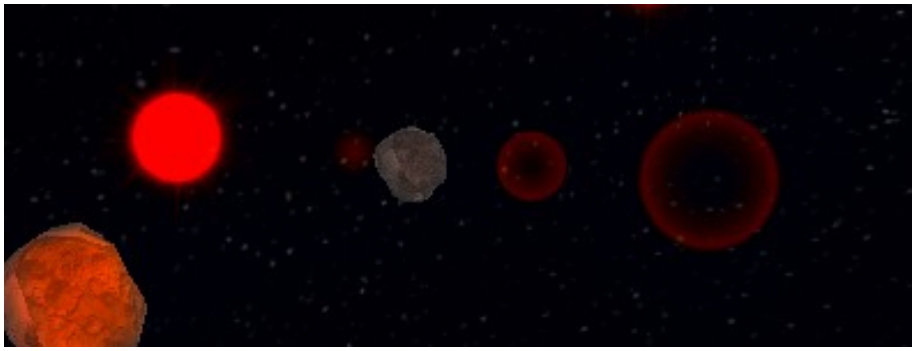

Figure: lense flare in our game

The program first looks for suns which are near the center of the window and which are near enough in z direction. If so, it checks if the sun is visible by reading one pixel from the light source texture. If it is visible, the effect will be applied.

Therefore four effect textures get blended into the final image, starting at the sun position and moving into the direction of the center of the window.

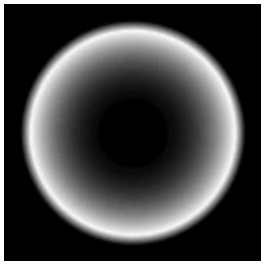

Figure: one of the effect textures

Spot Light

The laser beam can be seen if you move the ship towards an asteroid. If you can see the spot light on the asteroid you're ready to shoot.

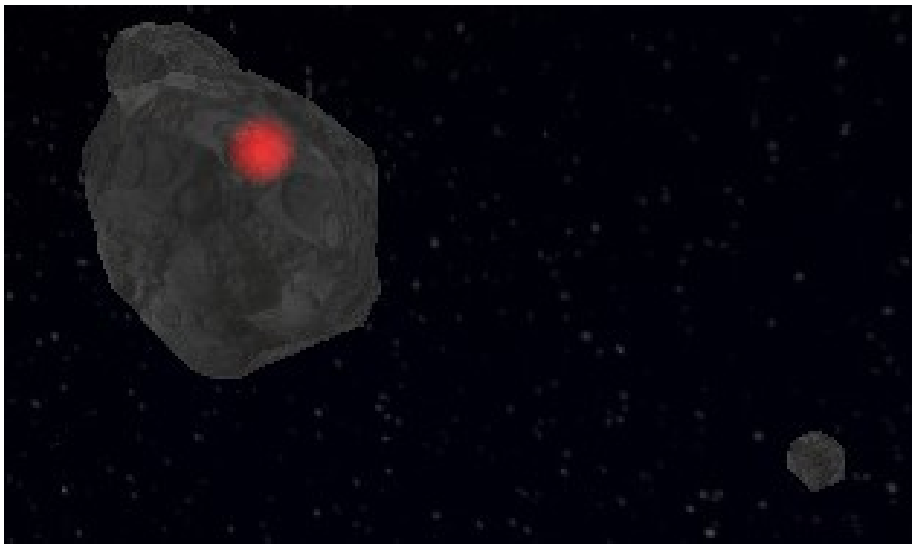

Figure: You're ready to shoot

The effect is implemented in our deferred pipeline. The difference between the light direction and the direction from the light source to the pixel is compared – if they are similar enough then the pixel will be shaded. You can see an intensity fall-off when looking at the spot light.

GPU-Particle System

We use a GPU-Particle System to give our ship's engine a nice effect in form of a tail. [OGL][MBS] The system is implemented via a TransformFeedbackBuffer. It is excluded from the Deferred Pipeline and rendered after everything else. There are two kinds of particles, generators and normal particles.

In the first step we update all the particles using the ParticleSystemUpdate-Shaders. The geometry shader is where all the magic happens. Here we emit vertices for each primitive (=particle) we want to render. There are just two constraints: If the particle we currently test (in the buffer) is a generator, we Emit it and generate additional particles of type normal. For each normal particle we check if its lifetime is already over, if *not* we render it.

In the second step we actually render the particles with the billboard-technique (so the particles are always facing the camera). There is nothing special going on here.

The velocity of the particles are dependent on the speed of the ship, if you fly faster the tail is longer, while if you stop you can just see a small circle that should depict the engine working.

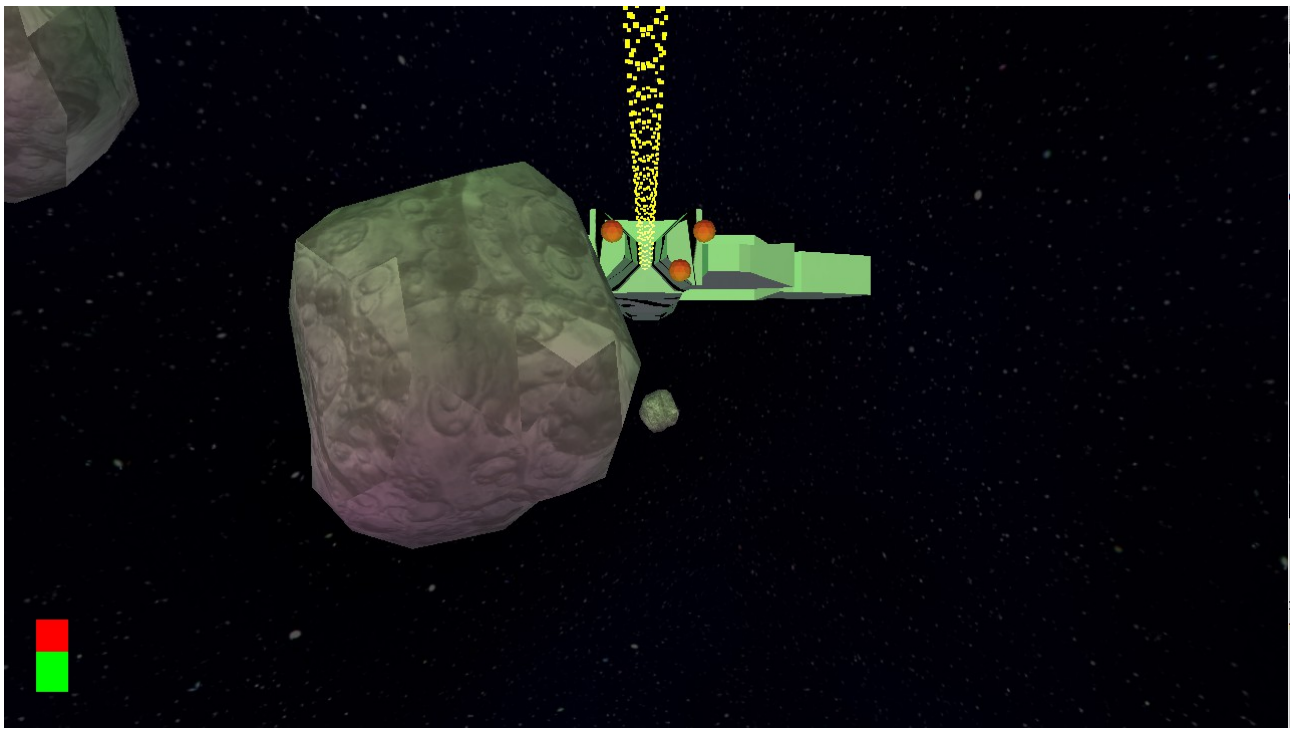

Figure: What a wonderful particle tail you have!

Physics, controlls, collision detection, sound

For **physics** we apply gravity to all asteroids, moving them towards the black hole. All objects have a data field representing the mass, of course the mass of the black hole is pretty big compared to all others.

Collision detection is done via bounding spheres (the same spheres are used for frustum culling!). We simply check if the sphere of the ship touches any other sphere (e.g. the sphere of a sun) by taking the distance between the center positions and comparing this distances to the bounding radius.

When implementing the **controlls** of the ship, we had to move to quaternions to avoid lock-in effects of usual transform matrices. The math can be looked up at [Yaldex], here we will just give a short overview of what we did:

- take the delta angles of yaw, roll and pitch to form the delta euler angles.
- calculate the delta quaternion of those euler angles
- multiply this delta quaternion with summed up quaternions of prior frames
- convert the summed up quaternion back to a matrix which can be used as usual as a rotation matrix

Via OpenAL we implement playing **sound**. As in OpenGL, buffers are created and the PCM data is uploaded to a target, which can then be played in different ways. To read PCM data from WAV files, we're using libSndFile.

Additional Requirements

General

Feedback (which option is turned on/off) is displayed in the title. So please make sure you don't start the game with fullscreen enabled, when you want to check these!

Animated Objects

You can see hierarchical animated objects when you look at the ship and the ship engine. There are two engines attached to the ship and they move corresponding to the ships speed. The hierarchical transformation is simply implemented by taking the transformation matrix of the parent object (the ship in this case) and multiplying this transform with the transformation matrix of the child.

FPS

Can be found in the window title when pressing F2. We do some low pass filtering on the FPS value to avoid “jumping” FPS values, if you are interested in the current frame time have a look at the value beside the FPS: it's the frame time in milliseconds.

Texture-Quality / Mipmapping

Both features are implemented. You can see them best on the ship itself. You can also set DontMoveAsteroids=1 to have a closer look at the asteroids.
We have used a “broken” texture so you can see that the interpolation is set to nearest neighbor or linear. If you want to check the code have a look at Texture.cpp and Controls.cpp.

Viewfrustum-Culling

Is implemented via the “geometric approach”. We start with the camera and projection settings of our game to calculate the bounding planes of the frustum. We then check for each object of our world if its bounding sphere is completely outside of the frustum. If this is the case, the object will be marked as invisible and won't be drawn in the next frame.

Considering the math, a good point to start is [Sutherland] and [Lighthouse3d].

You will see that frustum culling does not add to our fps. Why is that? Because the geometry pass, where frustum culling takes effect, is not the limiting factor in our pipeline.
You can see the effect tho:

In the INI set MonitorPasses=1. This will query the passes and write the output of that query into out/passes_time_elapsed.txt. The first column in that file represents the geometry pass. You will see that if you enable Frustum Culling the pass takes much less time to run than if you disable it!

In the console you can also see how many objects have been culled. They are simple not rendered!

Experimenting with OpenGL

For our deferred pipeline and models we used VAO and VBOs, we also used multiple FBOs to render the outcomes of our passes.

Transparency

As we use a deferred pipeline adding transparency isn't trivial. We used blend functions on multiple occasions throughout our pipeline (e.g. lense flare). Additionally if you press F9 the GUI element showing the energy level will switch from being opaque to transparent.

Libraries

Regarding libraries we use a few to make our lives easier:

Assimp: the open asset import library, is used for model loading. We used an older version and as it just worked and after hearing from problems and reading up a bit on the github issue tracker we refrained from updating to a newer version. Version used: 3.0.1270-sdk

[http://assimp.sourceforge.net/]

GLM: the OpenGL mathematics library, is the math support library we chose to use. Version used: 0.9.6.3

[http://glm.g-truc.net/]

GLFW: is used to create our window and the OpenGL context as well as receiving inputs from keyboard and mouse. Version used: 3.1

[http://www.glfw.org/]

libSndFile: getting PCM data from WAV files into memory

[http://www.mega-nerd.com/libsndfile/]

OpenAL: OpenGL like interface to sound driver

[http://kcat.strangesoft.net/openal.html]

Tools

We've used **Blender** to create our models.

[https://blender.org]

Audacity was used for modifying sound files.

[http://audacityteam.org/]

References

[Wright] Wright et.al.: OpenGL SuperBible

[Shirley] Shirley et. al.: Fundamentals of computer graphics

[Sutherland] Sutherland: Beginning Android C++ Game Programming

[Szeliski] Szeliski: Computer Vision

[Stroustrup] Stroustrup: Die C++ Programmiersprache

[Lighthouse3d] View Frustum Culling Tutorial:
http://www.lighthouse3d.com/tutorials/view-frustum-culling/

[Yaldex] Using Quaternions for flight simulators:
http://www.yaldex.com/game-programming/0131020099_ch16lev1sec3.html

[OGL] Deferred Shading:
http://ogldev.atspace.co.uk/www/tutorial35/tutorial35.html

[OGL] Particle System:

http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html

[MBS] Particle System:

http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=26