Developing ”Lasers”, a Virtual Reality Puzzle Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Daniel Gotzen* Johannes Groß† Lea Hiendl‡ Leon Knollmeyer§

Figure 1: *”Lasers” is a 3D puzzle game set in an abandoned and destroyed space station.*

Abstract

”Lasers” is a 3D puzzle game for Windows, Mac and Linux, developed by students in the scope of a practical course at the Chair for Computer Graphics and Multimedia (RWTH Aachen) with a focus on graphic programming and virtual reality.
The gameplay is based on a simple puzzle game idea that uses the environment to redirect a focused beam and activate triggers. This concept translates well into a first player experience with the Oculus rift, allowing for precise control of the head-mounted laser.
The objective is to repair an abandoned space station by advancing through the wrecked parts of it, posing as levels. This setting was chosen primarily for its potential in graphic design and immersion in a virtual reality game.
The resulting challenge was the programming of a graphics engine in C++ using ACGL (Aachen Computer Graphics Library, a high level abstraction of OpenGL) and other external libraries, and the creation of the game world and logic.

Keywords: game programming, puzzle, laser, space station, oculus rift, virtual reality

1 General Information

The concept of our project depended on various factors and constraints, such as playability in a first player perspective, and integrating the Oculus rift into the gameplay in a believable fashion, since immersion is one of the key factors of virtual reality. Thus we devised the idea of giving the player a tool directly controllable by his head movements, the laser, as his primary way of interacting with the environment. We wanted to restrict these interactions to the activation of triggers, and create gameplay through the introduction of mirror objects capable of redirecting the laser. The context and the meta-goal of repairing an abandoned space station then where natural consequences of the idea of lasers and mirrors, a.k.a. ”force fields”.

In the following we want to give an abstract overview of our development process and the key features we implemented in the timespan of the course.
In the first stages of the development process, our focus was mostly on the technical aspects and laying a foundation for various graphic effects. This included the integration of the Bullet physics library in our game, and the establishment of supporting structures for level loading with XML and lightning. This basis was then expanded to a full fledged graphics pipeline allowing consistent implementation of various (post-processing) effects.
The next step to make the project an actual game and not just a graphics demo, was realizing those early level design and logic ideas. This meant replacing our dummy models and textures with free and self-made resources.

2 Data Structures

We chose XML for describing our levels as it is easy to write and read, and could be extended over the time. Additionally it offers a tree structure, which we adopted for our internal data structures. A level loader function parses the file and builds an object tree to match the XML tree by using the composite design. We later dismissed storing the geometry as an attribute for every single object, because loading time and memory usage haven’t been on a reasonable level. Additionally binding and drawing every objects attached geometry and textures was a naive approach. We created the manager class GeometryHandler instead, which stores geometry, textures and shaders, so they just have to be loaded once. Moreover it creates a list of all drawable objects and sorts it so that fewest possible bind calls are invoked.

*daniel.gotzen@rwth-aachen.de
†johannes.gross1@rwth-aachen.de
‡lea.hiendl@rwth-aachen.de
§leon.knollmeyer@rwth-aachen.de

Another advantage of our data structure is the possibility of exporting our object tree to XML without loss. Placing objects by coordinates without seeing the result immediately is difficult. We made objects in the game movable with shortcuts, which makes fine tuning the level easier. After implementing a complete level exporter in an early version, we didn't use it because it deleted all comments and changed layout of the file. Instead we printed single objects as XML Nodes as debug output.

3 Physics

As our player can move freely, we had to have at least collision detection. We chose to integrate Bullet Physics. After identifying what to call and to set to create a physics environment that acts properly, we connected a Rigid Body to all our level objects. To be flexible later, we made geometric primitives, concave and convex hulls usable for collision detection. Now the physics world is implicitly created by the drawable geometry and attributes like mass and collision shape, defined in the level file.
For player controls, the Bullet Physics btKinematicCharacterController class was used at first. It enables climbing stairs and jumping, but avoids moving around other objects. Moreover it has a number of bugs, so using it was a temporary solution. Later we implemented basic controls and jumping by accelerating a player attached Rigid Body and climbing stairs with ray casts to replace it.
The laser is realized with Bullet Physics ray casts. The first one is emitted in view direction. When a collision with a mirror is detected, the function calls itself recursively with the mirrored direction. If the mirror is transparent, another ray is emitted behind it in the original direction. If a trigger is hit, it gets activated.

4 Shading & Lightning

4.1 Phong-Shading

For light we started out simple and implemented the basic Phong per Fragment Shading Model, with the three components diffuse for color and intensity, ambient to lighten up the rest of the scene and specular for highlights that move with the player's perspective. At first, we only hardcoded a directional light into the fragment shader that illuminated all the objects in the scene from a certain direction. But we wanted different types and more lights in a lightning system embedded into our XML structure and level loader. Now a light was completely defined by a pair of objects consisting of a LightSource class object containing all the lightning properties and a GeometricObject from the original level structure that defined its position and model. With that we are able to give any object light parameters and were able to create lights that moved with the geometry, such as rotating spotlights that lent the scene a little more dynamic feel.

4.2 Shadow Mapping

Naturally, after light we implemented shadows for our game. Because of its simplicity we decided on Shadow Mapping. This meant we needed an extra render pass for every light source, rendering the geometry depth-only from the view of that lightsource into a texture, a shadow map managed by the LightSource class. This texture then provided us with the means to determine whether something was lit or shadowed in the actual render pass.

4.3 Normal Mapping

Normalmapping per se was implemented very fast. The normal texture had to be integrated in loading the level and drawing the scene. In the fragment shader, the normal was read from this texture later. More complex was doing normal mapping right. The normal texture is in normal space, so besides the normal, tangent and bitangent per vertex had to be calculated while reading the obj file. We extended the ACGL VertexArrayObjectCreator by writing an inherited class to do that. Then in fragment shader, the texture-read normal is converted into this space.

5 Post-Processing Effects

Before applying postprocessing effects, multipass rendering had to be implemented. Classes for Render to Texture, Texture to Texture and Combine Two Textures passes were written. They manage framebuffers, color and depth textures and shaders internally and are created with a function pointer or input textures. They are attached to a manager class, to be drawn with one function call.

For Glow, the the scene is drawn with a special shader. It reads the fragment color from a glow texture or sets it to black if none exists. The draw function is assigned to a Render to Texture pass. Then two Texture to Texture passes apply a blur shader, so that all pixels are blurred in vertical and horizontal direction. The resulting texture is combined with the original scene draw by a Combine Two Textures pass.

Implementing FXAA as another postprocessing effect was chosen primarily because the laser showed noticeable aliasing in the low resolution Oculus Rift. The idea and the first algorithm steps were easy to find. Nvidia offers a whitepaper [Lottes January, 2011], that shows where to read pixels from the original texture and how to determine where to apply blur to smooth edges. Reduced to a simple version, without fine-tuning, the algorithm could be written with just a few lines of code. So most of our code is just a short, commented version of the existing algorithm.

6 Mirrors / Force Fields

Force fields reflecting the laser and mirroring the whole scene can be activated completely or set on semi transparency, splitting the laserbeam.
For each fully activated force field a texture is rendered during a separate renderpass into rectangle textures. The cameras position is calculated based on the actual player position and the position and rotation of the force field.
In order to avoid exponential recursion, mirrors in mirrors only show a standard force field-texture. Transparent force fields are only rendered in glowmaps. A parameter decides if this glowmap has a flickering effect.

7 Content

The main level was designed in blender, using textures, primarily handpainted in paint.net.
The audio tracks are composed in Fruity Loops Studio.
3D holograms show a miniature level-map on screens. These holograms are drawn in a seperate renderpass and added up to the framebuffer during the renderpipeline. The flickering effect is used for these screens too.

References

LOTTES, T., January, 2011. FXAA-Whitepaper. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.