Features

Freely movable camera

The camera is implemented via the scene graph: A „Transformation“ node is attached to the scene‘s root, to which the camera is attached in turn. By setting the transformation-node‘s matrices the camera can be moved around.

The exact transformation is determined by a character controller, which handles physics interactions, translation and rotation and implements a simple closed-loop control to ensure movement works as expected.

Moving objects

Moving objects are implemented in a very similar fashion to the camera: To move an object it is simply stuffed into a transformation node, whose transform is then altered.

Another way to move objects is to apply physics to them. In this case the model is set as child of a physics node, thus applying all transformations of that node to the child as well.

Hierarchical animation is implemented in puppets that walk the corridors: Their head is parented to the body, thus making them move in union.

Texture Mapping

Texture mapping is implemented in the usual fashion: All meshes have UV coordinates stored per vertex, which are then interpolated per fragment and used to sample the material’s textures.

Simple lighting and materials

Our lighting was rewritten from scratch to support dynamic, programmable lights. As we use a deferred renderer all light sources simply draw a quad over the screen and then apply their lighting calculations. While our materials are still capable of using Unreal Engine’s physically based shading model the lights currently don’t, effectively rendering PBR disabled for the moment.

Controls

The player can move with E (Forward), S (Left), D (Backward) and F (Right). The player can jump by pressing Space The camera view direction can be controlled by moving the mouse. The A Key turns on the flashlight, assuming it has charge left.

Basic Gameplay

The goal is to navigate a randomly generated dungeon before the time expires. The end is marked with a red light.

To implement the required 3D component our dungeon is made up of multiple stories connected by simple elevators.

Frustum Culling

Frustum culling is implemented by first calculating the camera’s orientation in world space. This is done by simply applying its transformation matrix to a vector (0, 0, 1) and subtracting the camera’s position from the result. This orientation vector is then used to detect objects which are outside of the viewing area and thus culled.

Additional Features

- physically based Character movement (the character is part of the physics simulation)
- resource loading framework for easy loading of assets and automatic memory management
- deferred rendering
- full scene graph

Additional Libraries and Tools

We are currently making use of these external libraries:

- bullet physics - http://bulletphysics.org/
- SDL - https://libsdl.org/
- GLEW - http://glew.sourceforge.net/
- Substance Designer - https://www.allegorithmic.com/products/substance-designer

In addtition to the game itself we also make use of the following libraries and tools, *which were also written by us*:

- pyrformat, for resource loading
- RUtil, generic utility functions
- RLib, for abstracting away low level actions, including OpenGL and I/O
- RGui, for UI drawing and I/O
- REng, the actual game engine
- assetmake, for automatically converting game assets into resources, as understood by the engine
- fontconv, for converting fonts to distance fields
- FrostScript, as shading language (used by some, but not all shaders)