Path of Enlightenment - Submission 2

Benjamin Grössing
Georg Ursits

Implementation

The base **datastructure** of our choice, to organize all scene objects, is a **scenegraph**. To allow better organization of the rendering (eg grouping by shader program used, or sorting by certain criteria) the visitor pattern is used. This organization is vital to operate on the tree structure provided by a scene. To get a final Model matrix for a single mesh, the scenegraph is traversed recursively to calculate the transformations needed for camera and overall object movement.
Exactly this approach made us able to easily animate whole subtrees of our scene graph. One Object, e.g. one of our towers, can hold reference to certain sub- scene- nodes to manipulate their model matrices in it's update function to **hierarchically animate** the object and it's parts.
We started with **complex objects** right from the beginning of our project. As mentioned above, we organize the objects we load via assimp in a scenegraph. All objects used in our game are 100% selfproduced. A good third of our content creation (initial draft of our level, the stickfigures including the rigging) happened before the actual coding kick of. The final results of our complex objects are based on those initial drafts, that we refined along the way to the finished game.
Our game is played in **first person perspective**, therefore the typical controls via mouse and W, A, S, D are used. Inputs are read in the suggested polling fashion that glfw provides.
In order to apply **View-Frustum-Culling** to the scene, we precompute bounding boxes for each object during the import process. In the actual render function call, six plane equations, depending on the players current position and view direction are created. These plane equations are used to check the bounding boxes of each object. If all bounding box coordinates are on the same (outer) side of one specific plane, the object can be culled.
Every object in our scene is **textured** via uv mapping (main ground, walls, stairs, stickfigures, pavements, stoppers...). Currently one primary type of light sources is used, point lights. The lightning model used right now is a direct blinn phong model, which was successfully extended to a deferred shading blinn phong model. Also, there is a slight variation of point lights, called 'laser lights'. These lights emit the type of light as a point light, but uniformly along a 3-dimensional line.
Transparency is not used in our game due to the complication of a deferred shading pipeline in conjunction with non opaque objects. This is mainly the reason why we skipped the transparency on/off switch as requested for debugging. We have alpha blended objects, like our towers during the building phase, our particles or our hud.
To restrict the player movement and deliver a realistic feeling a **physics engine** is used for collision detection and application of gravity. If the player falls off the map the player's position is resetted to the initial spawn point.

Features

- physics and collision detection
- enemies follow paths
- texturing
- dynamic lighting, with multiple dynamic light sources (projectiles, laserbeams, …)
- deferred shading
- gamma correction
- HUD
- tower placement on pavements
- multiple tower types
- tower shooting animation and interactive projectiles (illuminate surroundings)
- view frustum culling
  http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-boxes/

Lightning, Texturing

Every object including the main-ground is lit by point lights. The point-lights' influence is limited via the usual attenuation calculations. All objects in range of a light source are illuminated via a blinn phong shader. The focus of our game is heavily concentrated on lightning and it's effects, that's why deferred shading was implemented to enable our engine to calculate around 50-100 light sources.
Furthermore one additional type of light was added, to provide correct calculation of the laser beams. These beams illuminate their surroundings in the fashion of a striplight.
Also, so called 'range lights' are used to preview the range of tower when building them. They use a modified attenuation formula to create a hard cut-off at the end of the range.

Effects

- Normal Mapping - 1
  http://ogldev.atspace.co.uk/www/tutorial26/tutorial26.html
- GPU-Particle System (+Transform Feedback,Instancing) - 1
  http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html
- Deferred Shading - 1
  for deferred shading the following tutorials/ references were used for inspiration:
  - http://www.codinglabs.net/tutorial_simple_def_rendering.aspx
  - http://www.neuroproductions.be/opengl/making-a-3d-game-with-opengl-deferred-shading-and-stuff/
- Bloom - 1
  implemented by following the outlined process of one of the cg talks
  https://lva.cg.tuwien.ac.at/cgue/wiki/lib/exe/fetch.php?media=students:cgue13_bloom.pdf

Implementation details

- **Animations:** Towers are animated by applying rotational transformations to specific nodes in the scene graph.
- **Path following:** Paths are defined in the world model as meshes with lines only. When loading the map, some optimizations are applied to that mesh in order to place it directly on top of the underlying surface and precalculate enemy orientations (i.e. XZ rotations).
- **Tower placement:** A bullet ray cast as well as a ghost object are used to efficiently check if a new tower would collide with any other object. Towers can only be placed on specific meshes and with a proper orientation.
- **Shader management:** Each drawable object (i.e. mesh) registers itself for subsequent draw calls. A render priority can be used to specify the order of these calls (e.g. for transparent meshes like tower drafts).
- **Blinn Phong Shading**: Normals and light directional positions are interpolated and used to determine the correct shading for each pixel based on the texture.
- **Deferred Shading**: to optimize the game-event version deferred shading was enabled to allow a high number of dynamic point lights.
- **Gamma correction**: to compensate problems of beamer or screen setups gamma correction was implemented, to adapt the game's visual appearance to the quality of displaying medium
- **HUD**: to provide a first draft of the Head Up Display basic bars to visualize certain properties of the game was added. A progressbar for the tower building was implemented as well.

Tools

Blender, was used for multiple purposes:

- cad authoring tool (content creation)
- rigging & static transformation of the model
- level editor & level composition
- action editor via emptys
- enemy path definitions

Interaction Sequence/ Wave Management:

Our Wavemanager is driven by a json configuration file (bin/poe.json) that defines among other things the types, speed, health and spawn frequency of our enemies. The enemies spawn in waves on certain ways of our map. Delays are defined in between those waves, were events, like hud messages or opening new ways could be triggered. If a way is opened the corresponding message is displayed and the symbolic barrier in front of that way vanishes. The vanishing of these barriers is one interaction sequence triggered upon defeating a wave. Another interaction is building towers. As indicated in the hud and the loading screen different

tools can be selected via the mouse wheel and the corresponding number keys. By standing on or aiming at a pavement a temporary draft of a tower (ghostly shaded) appears that indicates the position, the selected tower will be positioned at if the process of building (left clicking and holding down the mouse button) is finished. As described above a range light indicates the shooting range of each individual tower in that drafting phase.

Libraries

The only libraries used were suggested by the cg wiki:

- GLFW (http://wnww.glfw.org/)
- GLEW (http://glew.sourceforge.net/)
- DevIL (http://openil.sourceforge.net/)
- FMOD (http://www.fmod.org/)
- Bullet (http://bulletphysics.org/wordpress)
- Assimp (http://assimp.sourceforge.net/)
- glm (http://glm.g-truc.net/)
- jsoncpp (http://jsoncpp.sourceforge.et/)