Lunacy

CGUE SS2015

Ulrik Schremser (0728034)
Patrick Mayr (1226745)

June 22, 2015

1 Remarks

Please, do not call 911!

2 Implementation

The game engine is implemented as a component based game engine. That means, that every `GameObject` in our scene is able to store components, which will handle different things that the `GameObject` needs to do.

1. A subtype of `InputComponent` will handle the user input respectively other input (e.g. an autorotation of a cube arround its center)
2. A subtype of `PhysicsComponent` will handle the physics

If a `GameObject` does not need all components, you can simply set them to a `nullptr` (e.g. a static light source will be represented as `GameObject(nullptr, nullptr, Model)`, because there is no need to move or apply physic to it).

The rendering is done by using a subtype of `RenderSytem`, which will use one or more subtypes of `RenderPass` to render the scene and do the post processing.

2.1 Freely movable camera

As our game is first-person, the player not only interacts with the character (represented as an invisible sphere), but also with the camera itself, which would represent the players head: Turning left and right is accomplished by rotating the character around her height-axis, Turning up and down is implemented as rotating the camera in model space.

For better control we clamped the possible up and downward movement to ± 90 degrees (i.e. looking straight downwards and looking straight upwards). Please also note that when Luna uses her skill "Gravity Swap" the camera turns together with the player. The result is a gameplay that is consistent between moving on the ground and moving on the ceiling.

2.2 Moving objects

1. Player:
   `Input`, which gets all user input, is attached to `PlayerInputComponent`, which processes the user input further (e.g. adjusting `time_delta` or gravity direction). `PhysicsComponent`, so `Bullet`, will do all the calculations and returns a new model matrix for the `Model` player.
2. Other objects:
   a) `InputComponent`:
   It is possible to attach an `InputComponent` with a predefined movement (e.g. an autorotation of a cube arround its center). Even hierarchical transformations are possible, if other `GameObject` are attached as children.

b) `PhysicsComponent`:
If you attach a `PhysicsComponent` to a `GameObject`, `Bullet` will do all the calculations and returns a new model matrix for the `Model`.

2.3 Texture Mapping

Every `Gameobject` stores the `Model` which got imported from a `.dae-file` by using `Assimp`. `Model` is the representation of an imported model and holds e.g. the mesh data in `Mesh` for a regular game object. `Mesh` stores the textures and its vertex data using `Texture` respectively `Vertex`.

`DefaultRenderSystem` respectively its passes, which are subtypes of `RenderPass`, will finally pass the texture and vertex data to the shader to do the texture mapping. The vertex shader pass the UV data straight through to the fragment shader, which will get the color data from the texture per fragment using the `GLSL` function `texture(Texture, UV)`.

2.4 Simple lighting and materials

See Illumination and Texturing [4]

2.5 Controls

You control Luna by using the typical first-pony-shooter controls.

Key	Effect
W	move forward
S	move backward
A	move left
D	move right
1	use gravity swap
2	use slow motion
SPACE	jump
LEFT SHIFT	run
R	restart game (only available on lose/win screen)
ESC	quit game
MOUSE	camera control
F1	Help
F2	Frame Time [on/off]
F3	Wire Frame [on/off]
F4	Textur-Sampling-Quality [Nearest Neighbor/Bilinear]
F5	Mip Mapping-Quality [Off/Nearest Neighbor/Linear]
F6	Catch mouse [on/off]
F7	-
F8	Viewfrustum-Culling [on/off]
F9	Transparency [on/off]

2.6 Gameplay

The goal of the demo scene we prepared is to get to the exit without walking into traps or getting yourself hurt by sudden earthquakes (which are currently indicated by a countdown timer and invisible triggers. In addition to normal movement (WASD), jumping (spacebar) and running (left shift modifier pressed while walking) there are two special abilities which might come in handy to beat the game:

1. Gravity Swap (Key 1): Luna swaps her own gravity: She is now able to walk on the ceiling.

2. Slow Motion (Key 2): Luna uses her magic powers to extremely slow down time.

Some challenges might also require you to use things in your environment (e.g. moving objects).

2.7 Effects

2.7.1 Glow

1. `RenderPass1` with its shaders `pass1.vert` and `pass1.frag` is creating the glow source texture and renders the scene in another color texture.

2. `RenderPass3` with its shaders `pass3.vert` and `pass3.frag` filters the glow source texture vertically using a 1x64 gauss kernel.

3. `RenderPass4` with its shaders `pass4.vert` and `pass4.frag` filters the glow source texture horizontally using a 64x1 gauss kernel.

4. `RenderPass5` with its shaders `pass5.vert` and `pass5.frag` is finally blending the color texture with the blurred glow source texture scaled by a glow intensity factor. Futhermore we using a max function on this effect to avoid that the objects center is brighter than the border. Finally we spread the excess of the color before clamping it.

2.7.2 Deferred Shading

1. `RenderPass1` with its shaders `pass1.vert` and `pass1.frag` renders all necessary informations for the illumination (position, diffuse color and normals of the model) to different textures.

2. `RenderPass2` with its shaders `pass2.vert` and `pass2.frag` uses the geometry of the light source (sphere for a point light) to define which objects (fragments of an object) will be illuminated by this specific light source. With this approach we do not have to do unnecessary calculations of hidden points (depth test) or for points that are far away from the light source (out of range of the light source geometry means minimal influence to the result).
   The fragment shader then simply applies the phong illumination model using the data of the actual light source and the textures, which were genereted in the previous pass.

2.7.3 Shadow Maps (with PCF) + Omni-Directional

For shadow mapping, we used a cubemap (which can be found in `CubeMapFBO`) and the approach as described in RTR-Slides Omni-directional Shadows: The geometry is duplicated and transformed by the corresponding view-projection matrix for each of the 6 sides of the cubemap in the geometry shader (see `pass0.geom`). This way, only one instead of the usual 6 passes is needed. Furthermore as to improve the performance, we decided to implement a culling phase, where all objects lying beyond the light-sphere are not considered as they can't cast shadows anyway. This was done in `RenderPass0`.

2.7.4 References

1. **Glow:** GPUGems Chapter 21; CGUE-Slides Bloom & Glow
2. **Deferred Shading:** Neuro Productions; OGLdev Tutorial 35
3. **Shadow Maps (with PCF) + Omni-Directional:** RTR-Slides Omni-directional Shadows; GPUGems Chapter 12; CGUE-Slides Shadow Mapping

2.8 Animated Objects

To apply a hierarchical animation to a `GameObject` you have to implement and assign a subtype of `InputComponent` (e.g. `AutorotateEverythingToParentInputComponent`). `AutorotateEverythingToParentInputComponent` will rotate every child recursively around its parent on a given axis. The rotation axis has to be encoded in the model name in the `.dae-file` (e.g. „rot-z“). Subtypes of `Level` will then set the correct rotation axis while assigning the models to the `GameObject`.

2.9 View Frustum Culling

For View Frustum Culling, all Meshes get a precalculated bounding box assigned during setup. The culling of meshes takes place in `RenderPass1` for regular scene objects and in `RenderPass2` for all light objects. All objects are called in clip space, i.e. the bounding boxes get transformed by the MVP-matrix and then the clipping happens in homogenous coordniates. If an object lies completely outside of one of the frustum planes it gets culled. The mode of operation can e.g. be checked by pressing `F2`, which displays besides other metrics also the number of triangles `tr` as well as the number of lights drawn `li`.

2.10 Experimenting with OpenGL (Transparency, Texture Sampling, MipMapping Quality)

`Settings` handle all the setting changes, which are activated by the user by pressing one of the function keys from `F1` to `F9` (`F7` has no function applied). Therefore `Settings` has the constructor `Settings(Input * const input, GLFWwindow * const window, int window_width, int window_height)` to handle the input and change some settings of the `GLFWwindow`. `Settings` is simply setting flags according to the user input. In other parts of the program `Settings` can be accessed by using the `extern` keyword.

Example:
If the user has pressed `F3` the `wireframe`-flag in `Settings` will be set. The `draw`-method of `DefaultRenderSystem` will then check for every call, if the `wireframe`-

flag is set, by using `extern lunacy::settings::Settings* game_settings` defined in `RenderSystem`. Depending on that state it will enable or disable the wireframe mode.

3 Features

- single player
- various magic skills
- make use of physics
- challenging puzzles and obstacles
- glow effect
- collision detection
- more than one hundred light sources per scene
- omnidirectional shadows

4 Illumination and Texturing

Our game is illuminated by a multiude of small light sources hidden in glowing mushrooms, stones or orbs as well as an ambient light source. The lighting itself is applied by the means of deferred shading system implementing a simple Phong shading. The ambient light originates from one light source that stays and moves according to the location of the camera. For the technical details please refer to the Section where we describe the Deferred Shading in our game [2.7.2].

This illumination model will be applied for every `Gameobject` that should be rendered, therefore

1. the player with its camera is stored separately to the other `Gameobject`
2. light sources have stored their light geometry different (it is part of `Light` and not of `Model`) then the other mesh data of regular `Gameobject`
3. invisible triggers encode this in the model name (`trigger-invisible`)

so that they will not be part of the rendering.

Every `Gameobject` stores the `Model` which got imported from a `.dae-file` by using `Assimp`. `Model` is the representation of an imported model and holds e.g. the mesh data in `Mesh` for a regular game object. `Mesh` stores the textures and its vertex data using `Texture` respectively `Vertex`.

If the importet model is a light source, there are no mesh data to store, but light data. This is done by using `Light` as a container. All data that are necessary for the rendering come together in `Rendersystem`, which will pass the data to the subtypes of `RenderPass` with its vertex, geometry (optional) and fragment shader `passX.vert, passX.geom, passX.frag` (X means shader 0 to 5) , which will do the actual shading.

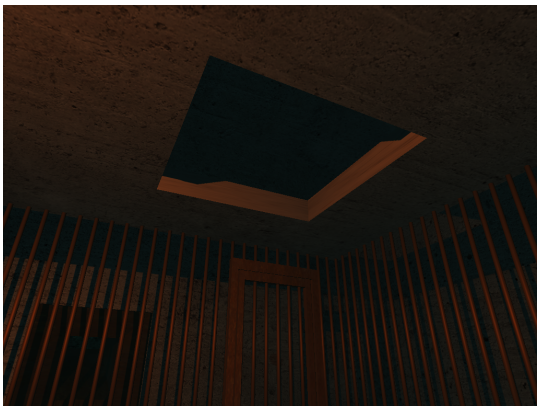
5 Game Walkthrough

(a) Use „Gravity Swap“ to escape the cell through the hole in the ceiling

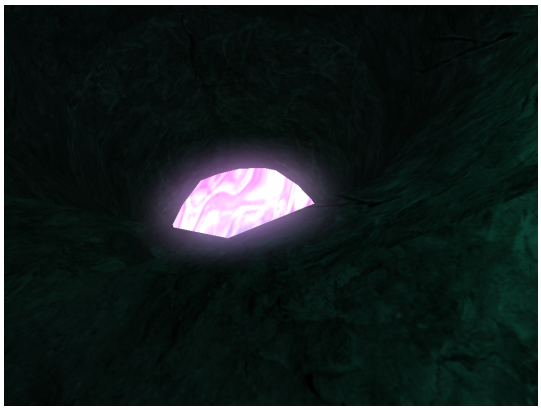
(b) Use „Gravity Swap“ to walk on the ceiling to cross the hole filled with acid

(c) Use „Gravity Swap“ to walk on the ceiling to reach the entry to tunnel at the side

(d) Throw the rocks in the hole filled with acid to cross it



(a) 1^{st} obstacle



(b) 2^{nd} obstacle

(c) 3^{rd} obstacle

(d) 4^{th} obstacle

6 Additional Libraries

1. **Object-loader:** Assimp(v.3.1.1)
2. **Physics:** Bullet (v.2.82)
3. **Image-loader:** DevIL (v.1.7.8)
4. **Window-handling:** GLFW (v.3.1)
5. **Logging:** spdlog (commit #11d4ae7)
6. **OpenGL-loader:** GLEW (v.1.12.0)
7. **Math:** glm (v.0.9.6.3)

7 Modeling

1. **Modeling Tools:** Blender
2. **Models:** Discord
   other models are self-made