

## corrid00r - Task 2

### Implementation:

#### HUD:

With the help of <https://learnopengl.com/In-Practice/Text-Rendering> tutorial, we managed to build in the Heads-Up-Display. For this we used an external library called “free-type”. You find the declaration in line 258 to 310 of the file “Main.cpp” and the corresponding method RenderText() from line 1585 to 1637. We used the same method for our dialogs.

#### Light Mapping:

For light-mapping we followed the tutorial provided by the CG-UE-Team. The first thing we did, is to set up our entire level in *Blender*. We put all the furniture and lights we wanted, textured the walls and floor and followed the instruction of the tutorial to bake the texture. After that we used the “Assimp”-model-loader library to load the previously created objects along with the uv-coordinates and finally loaded our baked texture.

A loaded object is stored in a SceneObject-object. You can find the implementation in the file SceneObject.h.

#### Animation & Hierarchical Animation using Physics engine:

For the integration of the physics engine Nvidia PhysX, we also followed the provided tutorial of the CG-UE-Team. For our moving objects we used Nvidia PhysX dynamic actors. You can see the implementation of for example one of our “ballOfDeath”s from line 718 till 725 in “Main.cpp”. The Geometry being drawn is a simple instance of the already provided class “Geometry.cpp” from the ECG framework.

Another moving object in our game is the character controller which is used for collision detection with the dynamic (balls) and static actors (walls) in the scene.

Hierarchical animations are also used for our dynamic objects (balls) since they move together.

Furthermore, we used PhysX trigger-shapes to implement the picking up of notes.

#### Cell-shading + Contours:

We used our experiences from the ECG-Lab course (old modus, 2016W) to implement cel shading. The implementation itself can be found in the corresponding shader-files (cel.vert, cel.frag). The basic approach is to take the color read from a normal texture and convert it to one of  $n$  discrete levels. In our case  $n = 4$  and can be adjusted via a uniform set in the game loop.

The rendering of the backfaces is done directly in the draw()-method of the Geometry-class. It uses another shader (silhouette.vert, silhouette.frag) which enlarges the back faces which normally would not be drawn along their normals before drawing the geometry itself.

Examples for cel-shaded objects are the balls of death in the first room (first door left), where you can see the back-face-rendering pretty well and JAVA, the coffemaker.

### **Video-Textures :**

For the Video-Textures-Effect we used *ffmpeg* and the library *libav* on which it is based. First, the video is read and we extract information about its length and FPS to calculate the amount of total frames. (Lines 754-789 in Main.cpp) This value is then used to update the extracted still images as textures so that the video is rendered in real-time. The frames are extracted and put into a `std::vector` for later use. You can see the video textures in action in room two (second door to the left). The used video can be replaced with any video in “/assets/videos/”, as long as you call it “never.mp4”, though the prepared geometries and textures may produce black frames around the image for certain resolutions or aspect ratios.

We tried to find online-tutorials for this effect but since the library has changed massively in the past few months, we were unable to do so and finally ended up using only bits from *StackOverflow* here and there. The rest we figured out by trial-and-error.

### **Procedural-Textures :**

Procedural textures are based almost entirely on the perlin noise function which is described in Computergraphics VO. You can find the implementation of the shaders that generate the marble textures in `procMarble.frag`. `ProcMarble.vert` is a standard vertex shader that does nothing of particular interest. You can see the results of the marble shader in room 4 (third room to the right). There, marble columns decorate the left and right walls. The marble texture can also be found in room 6 (first room on the right) where one table and the vase are shaded by it. The inner walls in room 5 (second on the right) also use this texture. The perlin noise function we used in the shader was inspired by Stefan Gustavsons implementation for OpenGL you can find at: <https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83>

### **Particle System:**

For our particle-system we used the provided tutorial by the CG-UE-team, which can be found in the slides “Particle Systems with Compute & Geometry Shader”. For additional help, we also used the linked tutorial “<http://www.geeks3d.com/20140815/particle-billboarding-with-the-geometry-shader-gsl/>”. Our main-implementation of the system can be found in “`particle.comp`”, “`particle.frag`”, “`particle.geom`”, “`particle.vert`” and in “`Particle.h`”. You can see the particle-system once by not skipping the start dialog and a second time by opening the third door on the left side to room three. One last time, they can be seen when finishing the game and not skipping the dialogue at the end.

### **Complex 3D Models from Files:**

As mentioned before we used the “Assimp”-model-loader to load our complex objects. For integrating the library, we followed the tutorial described in “<https://learnopengl.com/Model-Loading/Assimp>”.

For each model we load we created a `SceneObject` and each `SceneObject` is divided into several “Geometries”. For this we wrote our own class “`SceneObject.h`” and used the provided class “`Geometry.cpp`”, which we just adjusted for our needs.

The models for the elevator, the stairs, the pillars and the furniture are from the website <http://www.sweethome3d.com> and are available under the free art licence. The rest of the level was done by us with the modelling-software *Blender*.

The “ballofdeaths” are simple balls with the texture of the sun provided by the ECG-framework and the doors are simple cubes also textured with the provided wood-texture. The coffee-maker was also done by us by taking several pictures of different angles of a coffee-

maker and processing it with a software called “recap Photo”, which creates 3D-objects along with the texture and UV-coordinates we directly imported to a SceneObject.

### **Controls:**

In order to move our player we implemented the method `movePlayer()` in “Main.cpp”. Originally, the method only moved the camera, which is why we pass a reference of it in the methods signature. Since we want framerate-independent movement, we also pass the calculated `deltatime` to the method. Since the keys to move the player are implemented via polling, we call the method in our render-loop and then simply check what direction the player wants to move in. In our case, W, A, S, D and the arrow keys can be used interchangeably to move the player. By pressing the space-key the player is also able to jump. Mostly for debugging reasons, we also implemented a “boost” function. You can tell the character to move three times as fast as usual by pressing the left-shift-key once. To disable the boost, simply press the same key again.

The camera is controlled by moving the mouse to look around. We do not use zoom, strafing or any other camera-effects in our implementation.

### **Frustum Culling:**

We used a tutorial hosted at <http://www.lighthouse3d.com/tutorials/view-frustum-culling/> to implement frustum culling. So far, culling works, but is buggy. By pressing F2, you get statistics printed in the upper left corner of the screen. One of them is the number of geometries being rendered in each iteration of the game loop (Objects). If you press F8 to toggle the view frustum culling, you can see that number drop. Depending on where you are looking, it drops by one or two or many objects. Since we loaded objects like walls and ceiling as one single object, these will be drawn almost always.

View frustum culling is one effect we will work on till the final game event, since it does not work perfectly yet. To see an error produced by faulty culling, walk towards the end of the corridor and look down. You should be able to see through the floor. We did not figure out how to fix this behaviour but are still working on it.

### **Frame Rate:**

As with the objects being drawn, you also get a FPS count when pressing F2. This is calculated every iteration of the loop by dividing  $1/\text{deltaTime}$ , where `deltaTime` is the time since the last frame. (Implementation: Line 1146, Main.cpp). So the FPS number you see printed in the corner is really just an estimate but is still more accurate than actually summing up the amount of frames rendered in one second. Since we cap our framerate at 60 FPS (to use Nvidia PhysX optimally), this should always be 59 or 60. The same goes for the time since the last frame (`frametime`) which is also printed when pressing F2. It should be between 16 and 17ms.

### **Wire Frame:**

Wireframe mode is activated by pressing F3 and simply uses OpenGL functionality to only render vertices and edges. It can be used to find more complex models like the furniture used in room 6 (first room to the right) or the elevator in room 5 (second room to the right).

### **Illumination:**

For our game we have three different types of illumination: Ambient, point light and a directional light. Each object is textured using the implemented texture-shader. Also, each object is assigned a material and the normals are calculated.

The next thing we have is a “flashlight”, a point light moving with the character. The rest of the level is illuminated with pre-baked textures we created with blender, where we placed point lights throughout the scene.

### **“Features” :**

In this section we will talk about the features of our game. In the next section we will provide a detailed walkthrough through the game.

We provide help-dialogs with the coffee-maker called “JAVA”. These dialogs can be skipped by pressing “Enter”.

One of the levels has stairs which lead to an elevator and demonstrates the 3d-gameplay.

Also, as mentioned previously, you can “run” by pressing the left-shift-key once and walk slow by pressing the same button again.

### **Gameplay (Step-by-step) :**

The game starts with the introduction of the coffee-maker called “JAVA”. As mentioned before you can skip all dialogs by pressing “Enter”. “JAVA” tells the player to collect all the notes which are scattered throughout the level. Although there is no particular order for you to collect the notes, we will still refer to “first note”, “second note” etc. You can open each wooden-door by running into it and waiting for it to open. The last door made by bricks opens automatically once you have collected all six notes.

You can find the **first note** on the desk of the first room on the right side and you can pick it up by also simply running into it.

The **second note** is in the room of the first door of the left side. The trick to get through this level is by going through one of the brick walls at the very beginning of the room. Which wall you chose depends on your preferences. Once you are on the other side of the wall you go to the end of the room and again go through the brick-wall until you see the “ballofdeaths” again. What you also can see now is the note which you can pick up by, again, running into it. Apply the same trick to get out of the room or run into one of the balls to get to the position you started. You won’t lose any progress when you get “killed”.

The **third note** is in the second room on the right side. For this note you have to get upstairs to the elevator. Go a bit further into the elevator if nothing happens. The goal is to fall through a “hole”. Once you fell you can see the note. After you grabbed it a dialog with “JAVA” starts, which you can skip by pressing “Enter”. You will get teleported to the position you started the game, whether you skip the dialog or not.

The **fourth note** is in the second room on the left side. You simply go into the room and grab the note. You will see our video-texture projected on the walls.

The **fifth note** is in the third room on the left side. By opening the door, you can see our particle-system. You can go through the particles and you will see the fifth note.

The **last note** is in the third room on the right side. You go in the room and grab the note which is on the wall in front of you. After you found the last note, you get teleported to the position you started and “JAVA” starts with her “Ending”-dialogue. You can also skip this one if you want to. In this dialog “Java” tells you, that you got her last certificates for her bachelor-degree. If you wait for it to end you once again see our particle-system. The door opens and the game ends.