Developing a VR Game

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

Nico Linder* Vivian Huff† Jonathan Müller‡ Timo Gervens§ Benjamin Roth¶ Hendrik Gruss‖

Figure 1: *The living room. You can see shadows on the floor, transparency in the window and a glowing effect at a walking node in the back.*

Abstract

The main goal of our practical course was to develop a virtual reality game for the Durovis Dive by creating an iOS application. We decided to make a 3D point and click adventure playing in an atmospheric setting. In our game, tyou are trapped in an empty house and must escape. On your way you will have some scary encounters. Will you get out?

Keywords: game programming, virtual reality, iOS programming

1 General Information

When we first discussed what kind of game we wanted to create, we had to ask ourselves: How do you navigate while wearing a virtual reality headset? We came to the conclusion that the simplest way of navigating is to prevent using a controller and having your hands free. Furthermore, this keeps the illusion of being in the game alive, since you have no physical contact with your surroundings just as the character you are controlling.

So that's why we basically made a point and click adventure in 3D. Meaning, you examine your surrounding environment by simply looking around, and if there is something you can interact with in your line of sight, it starts glowing. After a short amount of time focussing on it the object performs some special action, for example opening a door or playing a sound. To move around in our game, you have to look at special waypoints, which take you through the house. With all that, it is possible to move around, solve all the riddles thrown at you and eventually escaping our house.

*nico.linder@rwth-aachen.de
†vivian.huff@rwth-aachen.de
‡jonathan.mueller@rwth-aachen.de
§timo.gervens@rwth-aachen.de
¶benjamin.roth@rwth-aachen.de
‖hendrik.gruss@rwth-aachen.de

2 Graphics

2.1 General

For an atmospheric game it's very important to provide realistic graphics that convince you of being in the game, especially for virtual reality headsets. That's why we implemented several graphical techniques, always working at the limit of our devices to get the most out of them. We will present the most interesting ones of them in the following paragraphs.

2.2 Deferred Shading

Since we wanted to use multiple lights we decided to use deferred shading. This enabled us to render way more lights than with the forward rendering technique.

Basically, when you draw a scene with forward rendering, for every object and light source pair the lighting calculation has to be done separately. This amount of calculation increases rapidly when using many lights.

With deferred shading, we first save the color, normal and specularity of each pixel in a separate buffer, then reconstruct the position of it from the depth buffer and finally do the lighting calculations on this buffer. This means for every light source there has to be done one calculation, independent from the object count.

A known bottleneck of deferred shading is that transparency is difficult to implement. So we had to draw all transparent objects with a forward renderer, too, but only affecting the nearest four point- and spotlights for the lighting calculation. We also drew the objects from back to front to create the correct transparency effect.

2.3 Shadows

We implemented real time shadows enabled per light source in our engine. This is done by rendering the scene in an additional depth pass, that we then project on the scene together with the light calculation in our deferred shader. It was the choice for deferred shading which allowed us to keep high quality performance while implementing such a sophisticated effect like real time shadows.

2.4 Glowing

To give the user a feedback if he is looking at something, objects that can be interacted with start glowing with an increasing intensity around their borders. We implemented this by first drawing an object into a black transparent buffer, setting all drawn pixels to opaque white. Then we blur the white pixels in a vertical direction and afterwards in a horizontal direction, to get a smooth transition. In the end, the object is stenciled over the buffer again, to refrain from setting the object itself to white. With all that, you get a white border around the glowing object.

2.5 Font

The user is directed through the house mainly by looking at so called sticky notes containing the text what to do next. So we had to add some text rendering. The procedure was the following: We have one texture containing a whole font. When we want to draw some text on a render target in our engine (e.g. a texture) we simply use it as an atlas and draw letter by letter as a rectangle using the right texture coordinates locating the letter in the atlas. With that we can display text wherever we want.

3 Game Logic

3.1 Engine Architecture

Some very important aspect of our practical course was to code as reusable as possible. So we built a whole engine which enabled us to create our game apart from the necessary content very easily, and can be used for other games, too.

Therefore, we decided to built it up in several independent layers. Since most of us had neither Macintosh nor iPhone devices, it was important to code platform independently. That was easily achieved, because OpenGL is using C calls, which can be used in combination with C++ and Objective-C both. All platform dependent code was coded for Windows and iOS and capsulated in system interface classes. Those interfaces can be used from our engine without knowing the specific implementation. Furthermore, after the setup for Windows or iOS is finished, all other code can be the same for both platforms in C++.

The next layer is the graphics layer. In this layer we wrote classes for shaders, materials, textures, renderers and other graphical parts, which are usable without knowing OpenGL in all detail. This made developing the game easier for the team members who were not deeply into OpenGL.

One layer higher is the game layer. Here we implemented our game, by creating an entity hierarchy, a picking, culling, moving and event system, and all the specific entities our game is consisting of, for example a door, a computer or a teddy bear, implementing the actions and animations they perform directly in those classes.

3.2 Picking

The picking system is the heart of our game logic, since everything in the game is done through it. The implementation is relatively straightforward. The model class represents a mesh together with its material to draw it. This class also calculates the bounding box of the mesh, which is used in the picking system. It simply fires a ray and checks for intersection with some of the models by using an octree, into which all the models are sorted. That makes picking very efficient. The system gives back a list of intersected models, sorted by their distance.

4 Content Tools

It's important for a game of greater size to get around with all the content that occurs. In the following sections we will explain how he got to manage all the content and how to easily work with it.

4.1 Resource Management

First of all, we had to get a system that manages our resources like sounds, textures, models and so on. We implemented a central manager to load resources from, which also releases them and uses reference counting to only load a resource exactly once instead of multiple times. All other load requests just return pointers to the originally loaded resources.

The resource manager is also responsible to dispose all classes which have to unload something, that means all classes that implement the *IDisposible* interface. They register themselves in it and when the central resource manager is disposed, it disposes all of them with it. With that, we don't forget to unload something and don't leave memory leaks.

4.2 Level Representation

There are many objects in our levels. Laying out all of them hard coded in our app is very time-consuming. Instead, we designed our whole level in Cinema4D or Blender and exported it to a special file format that describes which mesh and texture has to be used and with its specific transformation. Our game loads this file and places all objects correctly and directly puts them in the renderer.

But now we had the problem that objects cannot be changed from within our game. That's why we added a name tag in our file format describing a specific object, so every *Entity* in the game has one as a name attribute. Thus every object with a specific name tag can have additional features, for example every object with the door name tag will get the ability to open and close in the game.

5 Summary

Through all those techniques and design principles implemented in our engine, designing the whole game became an easy task and could be done very efficiently in the end.