DM842 Computer Game Programming

Christian Skovborg: chsko16
Jonas Teglbjærg: joteg16
Jeff Gyldenbrand: jegyl16

Supervisor: Rolf Fagerberg
Southern University of Denmark

December 24, 2017

Contents

1 Abstract

This project is made in connection to the course "DM842 Computer Game Programming". It consists of observations, lessons and descriptions of the main obstacles and theories encountered while implementing OpenGL in C++. For 3D modeling, Blender[1], a free open source modelling software is used, in order to create models to use inside the program. The models are imported to the program via Assimp[3], a library that is capable of interpreting a variety of different 3D model file types, and convert them to data types that OpenGL understands. Sound is implemented through the sound engine irrKlang[1]. By utilizing these tools, to mention a few, we have managed to achieve the goal; A simple playable space game where you can fly around in 3D space.

2 Specification

2.1 Project description

The primary objective of this project is to develop a computer game in 3D space, with elements like camera movement, movement of models, applying textures and light. In addition, the project should contain at least one of the elements A.I., physics simulation or collision detection. For the project, there was free choice of game genre as long as the above requirements were met. Finally, the game should run as either an executable program or with an installer that can be run on either Windows, Mac or Linux.

2.2 Project specification

The game is compiled and made executable and ready to run. Simply run ./game in the terminal or double click the game-executable to play.

When the application starts, the game will appear in a *800x600px* size window. By pressing the *F1* key, full-screen mode can be toggled. A 3D world consisting of the following models will appear:

- a complete 3D room
- a 3D spaceship
- PC monitors
- tables
- office chairs
- posters
- a blackboard
- two TV monitors

[1]Blender is a free to use and open source 3D program, URL: www.blender.org

When all these models are loaded, you will be able to control the spaceship inside this room with the keyboard keys *W* and *S*, and you will be able to steer with the mouse. At the start of the application, the mouse is captured by the game window, but it can be released by pressing the keyboard key *F3*. Pressing the *ESC* key will completely exit the application. Sound effects and background music are turned on by default, but can be muted by pressing the *F5* key and the volume can be adjusted by the keys *F6* and *F7*.

A text-GUI with the following text will be displayed on the middle top of the screen: "Battleground IMADA" Additionally another text is displayed at the middle left side of the screen: "First objective commander: scout the room for enemies". Additionally, in the lower left corner there is a fuel counter, showing how much fuel is left.

The game has no objective other than the ability to fly a spaceship around in 3D space that may collide with another spaceship. Hence, no A.I was implemented, only simple physics and collision detection.

3 Design and Implementation

The game is built around the idea of a space battle simulation. It takes place in the IMADA terminal room at SDU. Miniature spacecrafts navigating around furniture and walls makes for a decent practice-arena, in respect to novel programming of a game environment, and allows for scaling in both complexity or size of: model-shapes, model-detail, independent agents, lighting, collisions, sound and physics.

3.1 Platform

The game runs using the modern OpenGL approach (version 3.0 and on) together with the C++ programming language, primarily for speed and flexibility. While the course and the accompanying literature mainly explain theory and principles by using the legacy version of OpenGL, the modern approach is more manageable and flexible than the legacy one. As such, the modern approach is preferred, as it is more future-proof and also easier to handle in the long run. Because of the way the modern approach is built, it comes with the requirement of defined shaders [2].

3.1.1 Shaders

Shaders are basically small programs that decide how vertices are drawn on the screen. These shaders are run directly on the graphical processing unit (GPU) and needs to be compiled within the game itself. OpenGL defines six different shaders, where the most regularly customized ones are the vertex and fragment shaders.

- Vertex shader - takes care of drawing vertices correctly, telling the GPU where each vertex goes
- Fragment shader - makes sure each vertex has the right color, light and texture

In simpler terms, to manipulate and configure where and how the vertices are drawn, we configure the vertex shader. If we want to adjust how the colors are displayed, how textures are mapped or what kind of light we want the object to reflect, we configure the fragment shader.

3.2 Setting up initial configurations

In addition to the requirement of OpenGL and graphics drivers on the computer in question, there are some additional libraries that are necessary, in order to get up and running. In particular, the game depends on the following libraries:

- assimp - capable of importing different types of 3D model formats
- freetype - makes it possible to render text with customizable fonts
- GLFW - enables the ability to create and manage context windows
- irrKlang - enables sound integration with custom sound files
- glm - mathematical library that integrates with OpenGL to access vectors and matrices amongst others
- stb - lightweight header-only library that enables texture loading
- glad - Extension loading library that enables OpenGL function calls at runtime

All these libraries help minimize workload and trivialize otherwise complicated tasks. Some are more important than others; GLFW and glad could almost be defined as requirements to make an OpenGL application. Since OpenGL is open-source, so are usually the libraries that you can use with it. Therefore, all of these libraries have substitutes, all of which have pro's and con's.

3.3 Camera and Movement control

The implemented physics are naive, meaning that they are only loosely based on the Newtonian laws. Acceleration is a constant added to the previous velocity - not taking any mass into account. Drag or dampening of rotation is also just a constant factor subtracted from the current angular velocity or speed. All the speeds are simply naively capped - meaning that the speed accelerates at a steady rate until some given point, where as in real life the acceleration is a result of many forces acting on the object - only when the forces are in equilibrium does the object stop increasing speed.
At first the implementation was made after the kinematic model described in the lecture notes, but that approach got replaced by this very simple approach. This decision was made due to the simple fact that it is simpler to adjust parameters when programming collision detection and incorporating camera.
A time component is missing though - at first the game was so simple that the timekeeping between the frames was overkill. But at this point with the models imported it will probably be noticeable. When the frame-rate decreases the speed will as well. Introducing a time component would fix this effect - and in strained cases introduce lag or jumping in the graphics instead of slowing the whole game down.

3.4 Collision detection

The collision detection itself is *only implemented on one object* as a proof of concept. The detector calculates Euclidean distances between the spaceship and all objects in the environment. This approach is chosen to give better appearance of bouncing off or sliding of objects in odd angles, when colliding. Instead of rectangular bounding boxes, the spaceship gets a "bounding sphere". fig:1 The physics invoked in the collision is describes in pseudocode:

```
1 if(Collison){
2     RelativeDirection = Obstacle(x,y,z) - Spacecraft(x,y,z);
3     // get new direction vector
4     tempVec = CrossProduct(RelativeDirection, SpacecraftDirection); // gives a
          direction perpendicular to both
5     newDirectionVec = CrossProduct( tempVec, RelativeDirection); // This vector is
          perpendicular to the temporary and the relative, which ensure a new direction
          , tangent to the obstacle with some resemblance of the initial direction.
6
7     }
```

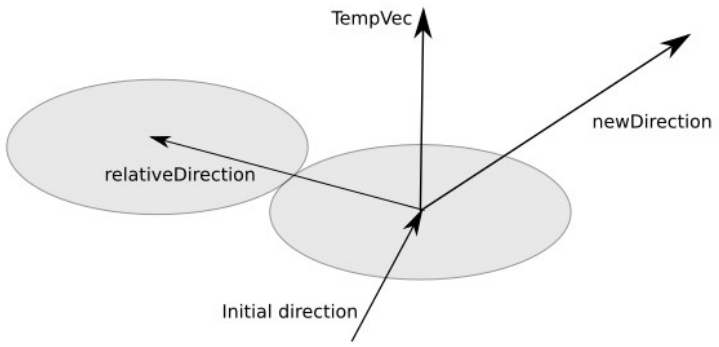


Figure 1: Illustration of direction-change

As previously explained, this solution is a rather simple one, and is subject to improvements. Below is listed some of what is considered the most important improvements to make to this solution:

- Problem: currently every object in the world is calculated immediately
  - Solution: Use large bounding boxes and only determine distance when the object first is within this bounding box
- Problem: visual improvement to add realism to the collision - current behaviour is sliding along colliding object or sporadically erratic flipping
  - Solution: add the difference between initial direction and new direction to the new direction. Normalize the result to gain a vector that reflects the angle of the initial impact - along the same plane.

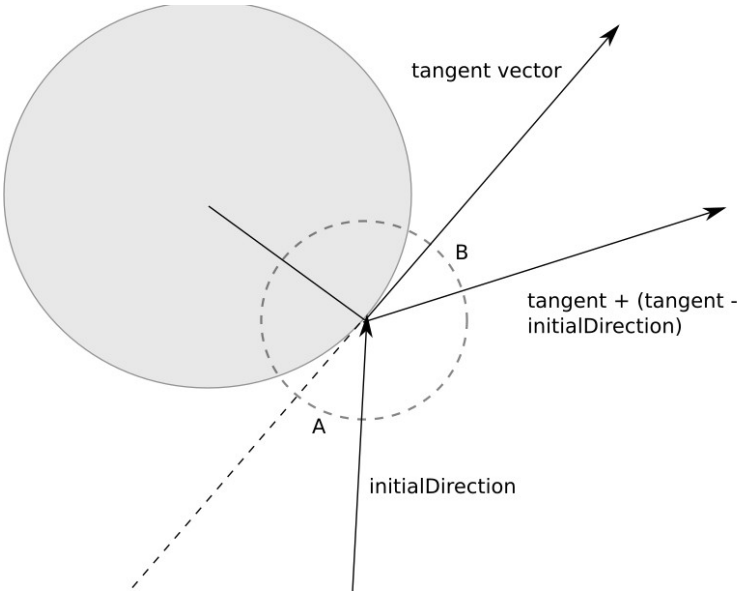


Figure 2: A = in-going angle, B out-going angle

3.5 Importing Graphic

Initially, we made simple objects directly in the main function, similar to fig:3. This is quite easy as long as we work with simple objects like lines, triangles, squares and circles, where we wont be defining too many coordinates for vertices, textures and normals. We could create a square by first creating two triangles, from coordinates in 3D spaces, and then telling which points OpenGL connects, as shown in the code below. The result would return a square on the screen:

```
1  float vertices[] = {
2  //    x      y     z
3      -0.5f, -0.5f, 0.0f,   // first point of triangle A
4       0.5f, -0.5f, 0.0f,   // second point of triangle A
5       0.0f,  0.5f, 0.0f    // third point of triangle A
6
7      -0.5f, -0.5f, 0.0f,   // first point of triangle B
8       0.5f, -0.5f, 0.0f,   // second point of triangle B
9       0.0f,  0.5f, 0.0f    // third point of triangle B
10
11 unsigned int indices[] = {
12     0, 1, 3,              // triangle A
13     1, 2, 3               // triangle B
14 };
```

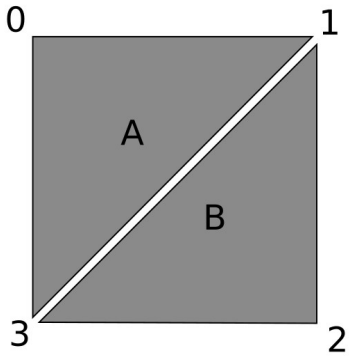


Figure 3: Triangle A = 0,1,3 and triangle B = 1,2,3

Of course, this would be resilient not to say an impossible process if you were to make more complex objects with hundreds of vertices or even thousands more. Therefore, we have used the open source program Blender that allows us to develop complex 3D objects, define colors, materials and textures, after which we can export the model to a file format known as wavefront.obj[2], which exports the positions of all the vertices, UV positions, normals and faces into an *.OBJ file. Before we examine an example of a wavefront.obj file, we will just describe overall what UV mapping, normals and faces means. We know that a vertex is a point where two or more edges meet. Normals is a vector perpendicular to a surface, or face, which is the outward facing side of a surface. UV mapping is the process of mapping a 2D image onto a 3D surface, hence applying texture to a model.

Lets go through an example of a wavefront.obj file. In fig:4 we have created an simple cube in Blender. First image shows the clean cube. Second image is our texture-file. And the third image is our cube with that texture wrapped around it.

[2]https://en.wikipedia.org/wiki/Wavefront_.obj_file

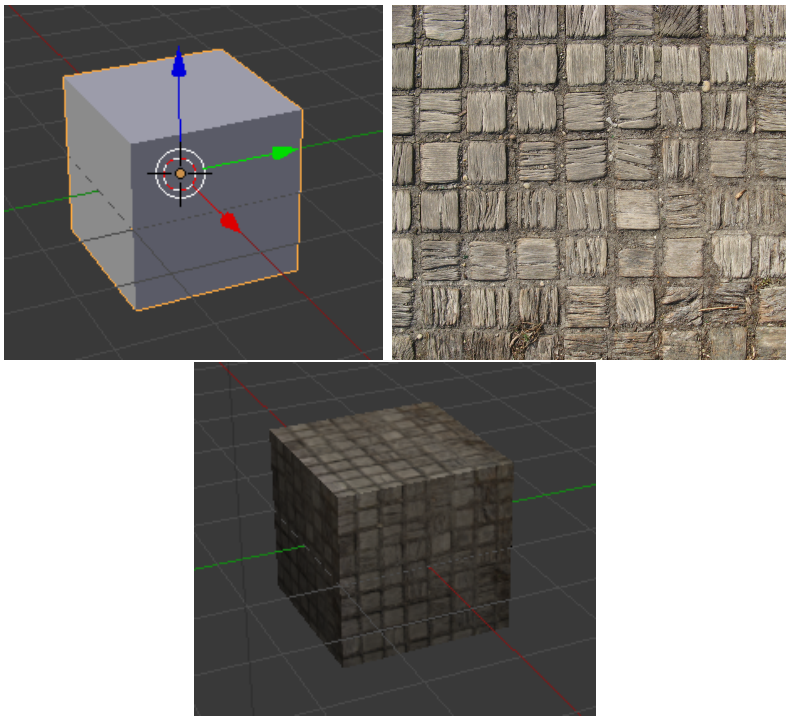
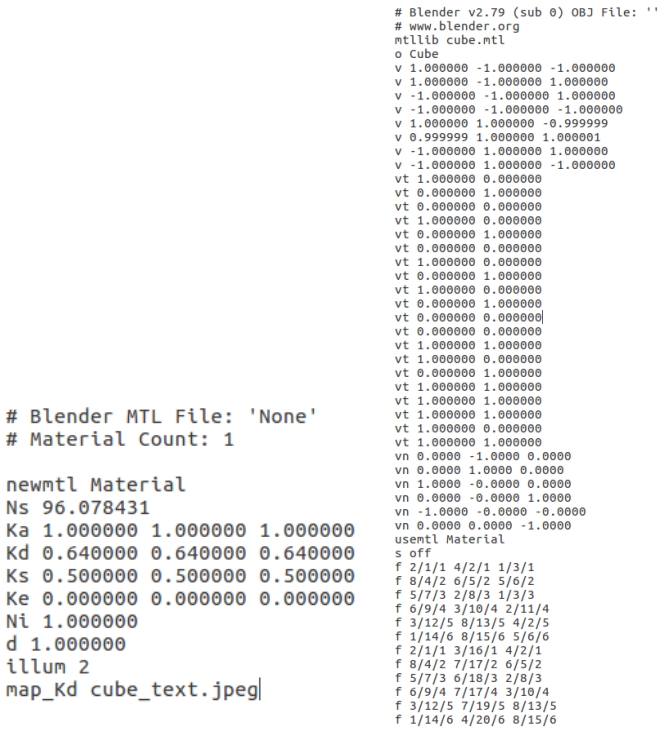


Figure 4: Cube model in blender applied texture

Now when we export it to an *wavefront.obj* file we get three files; *cube.mtl*, *cube.obj* and a local copy of our texture. In fig:5 we see two of our exported files. Lets go through the object file first. The first many lines of *v*'s are basically a list of coordinates on the models vertices. Then follows many lines of *vt*'s, these are the list of texture coordinates. *vn* is the list of the normals and finally, *f* is a list of our faces elements. This object file references to the *cube.mtl* material file, which, among other things, determines material the ambient color, diffuse color, specular color and transparancy. Lastly it holds the reference to the actual picture that is the texture file.



```
# Blender MTL File: 'None'
# Material Count: 1

newmtl Material
Ns 96.078431
Ka 1.000000 1.000000 1.000000
Kd 0.640000 0.640000 0.640000
Ks 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 2
map_Kd cube_text.jpeg
```

```
# Blender v2.79 (sub 0) OBJ File: ''
# www.blender.org
mtllib cube.mtl
o Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 0.000000 0.000000
vt 0.000000 0.000000
vt 1.000000 1.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 1.000000
vt 1.000000 1.000000
vt 1.000000 1.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
vn 1.0000 -0.0000 0.0000
vn 0.0000 -0.0000 1.0000
vn -1.0000 -0.0000 -0.0000
vn 0.0000 0.0000 -1.0000
usemtl Material
s off
f 2/1/1 4/2/1 1/3/1
f 8/4/2 6/5/2 5/6/2
f 5/7/3 2/8/3 1/3/3
f 6/9/4 3/10/4 2/11/4
f 3/12/5 8/13/5 4/2/5
f 1/14/6 8/15/6 5/6/6
f 2/1/1 3/16/1 4/2/1
f 8/4/2 7/17/2 6/5/2
f 5/7/3 6/18/3 2/8/3
f 6/9/4 7/17/4 3/10/4
f 3/12/5 7/19/5 8/13/5
f 1/14/6 4/20/6 8/15/6
```

Figure 5: cube.mtl and cube.obj

The assimp library parses the .obj file and separates meshes into nodes. Each node can contain multiple children, which again are meshes. Additionally, each node contains some information regarding that particular mesh, such as number of vertices, number of materials, colors etc. All these properties can be accessed and used by the program in order to draw the read object accurately.

Now we can use the assimp library to load in our models to the world space.

```
1 const char* modelPath1 = "../../src/assets/models/table/table01.obj";
2 Model tables(modelPath1);
```

Here we load our model of the table. We could choose to load the table independently 32 times and place them in world space. But instead we have made some nested loops that do the job for us. We can set their position in world space, rotate and scale them here.

```
1         for (unsigned int j = 0; j < 4; j++){
2             for (unsigned int i = 0; i < 4; i++){
3                 Shape tableObj(&cube, &tables, view, projection);
4                 tableObj.Move(3.0f + i * 21.0f, 5.0f, -15.0f - j * 40);
5                 tableObj.Scale(0.7f, AXIS_XZ);
6                 tableObj.Scale(1.2f, AXIS_Y);
7                 tableObj.Scale(1.3f, AXIS_X);
8                 tableObj.Rotate(180.0f, AXIS_Y);
9                 tableObj.Render();
10            }
11        }
12        for (unsigned int k = 0; k < 4; k++) {
```

```
13            for (unsigned int m = 0; m < 4; m++) {
14                Shape tableObj(&cube, &tables, view, projection);
15                tableObj.Move(3.0f + k * 21.0f, 5.0f, -5.0f - m * 40);
16                tableObj.Scale(0.7f, AXIS_XZ);
17                tableObj.Scale(1.2f, AXIS_Y);
18                tableObj.Scale(1.3f, AXIS_X);
19                tableObj.Rotate(0.0f, AXIS_Y);
20                tableObj.Render();
21            }
22        }
```

3.6 Lighting

To add some realism to the scene, we have implemented lighting. At first, this may sound simple, but implementing light is not just the mere task of creating an object that simulates light. In order for light to properly work, you need to apply modifiers to all the objects exposed to the light, and these modifiers need to be variable relative to the position of the light source. Additionally, different types of materials (wood, iron, glass) reflect light differently, and as such, implementing light becomes a complicated process. For this game, three types of lighting is implemented:

- Ambient lighting - some constant light that always lights the object
- Diffuse lighting - direct light impact a light source has on the object
- Specular lighting - highlighting of the object according to it's material

All these types can be combined in various ways to simulate real-world lighting, and in this case, the Phong[4] model has been used. The actual programming of this light simulation is done in the shaders, more specifically in the fragment shader.

3.7 Audio

Although a sound implementation was not a requirement for this project, we felt that this would be a necessary detail to create an overall impression of the game. But due to the fact that OpenGL does not provide any way to handle audio, and we do not know much on the subject of audio programming, we have used an already developed sound engine for this purpose. This sound engine is a free to use[3] software named irrKlang and consist of several libraries to handle both 2D and 3D sound. However, for the sake of simplicity, we have only implemented the sound to be handled in 2D, meaning that the player would not be able to distinguish which direction or how far away the sound is.

irrKlang supports various audio formats as .wav, .ogg, .mp3 and much more. For this project these three formats where used. In addition irrrKlang is supported on most operating systems, including windows and linux as our game supports. When drivers and header files was imported, we could simply play sounds with little code. For 2D sound we would simply define a path and whether the sound should loop or not.

[3]Free in the sense that as long as the engine is used for non-commercial purposes its free to use

```
SoundEngine->play2D(path, true);
```

If we wanted to implement sound effect in 3D we would have to specify where in world space the sound is. Then relative to the camera, we would hear the sound as an distance and direction, meaning that the further away an sound is placed in world space relative to the camera, the lower the sound would be, and vice verca.

```
SoundEngine->play3D(path, vec3df(0,0,0), true, false, true);
```

In order to give the player control over the sound, we have implemented it so that the player can turn on and off completely for all audio by pressing the F5-button, or just adjust the volume between 0.0 and 1.0 by pressing the F5- or F6 button on the keyboard respectively. To pause audio, we simply invoke the sound engine with:

```
SoundEngine->setAllSoundsPaused(true); // pauses sounds
SoundEngine->setAllSoundsPaused(false); // unpauses sounds
```

And to change volume we simply provide an arbitrary double-value as volume:

```
SoundEngine->setSoundVolume(0.1);
```

Some interesting issues occurred during playback of the audio files in the game. On linux machines there was a strange distorted sound. Sometimes during the entire playback of the sound, sometimes only the first few seconds after which it played normally. This issue did not occur on windows, which led us to believe something was wrong with the sound drivers on linux, and searched through archlinux.org and found:

"***The newer implementation of the PulseAudio sound server uses timer-based audio scheduling instead of the traditional, interrupt-driven approach. Timer-based scheduling may expose issues in some ALSA drivers.***"[4]

To prevent this issue to occur we have visit the link and follow their guide - or simply mute the game.

4 Testing and Conclusion

The testing has been done within a simple environment only containing the bare minimum. The odd thing is if the camera gets offset by some unexpected value or turned in opposite direction of what was expected - and there is no fix point, things might work as intended, but no way of knowing it. So getting a good fix point, some grid of sorts is noted as a good starting point for next project!

[4]URL: https://wiki.archlinux.org/index.php/PulseAudio/TroubleshootingGlitches.2C$skips_or_crackling$

Figure 6: Testing environment

List of known bugs:

The up-vector for the camera flips the spaceship, when executing a loop.

The collision can get stuck and slide around the periphery of the obstacle.

Collision tends to send the spacecraft upwards, regardless of impact direction

Collision is not implemented on anything but the dummy-spacecraft. (but the groundwork for further development has been laid.)

References

[1] Ambiera. *irrKlang*. URL: `https://www.ambiera.com/`.

[2] Khronos.org. *Shader*. URL: `https://www.khronos.org/opengl/wiki/Shader`.

[3] Assimp Team. *Assimp Open Asset Import Linrary*. URL: `http://www.assimp.org/`.

[4] Wikipedia. *Phong reflection model*. URL: `https://en.wikipedia.org/wiki/Phong_reflection_model`.