|  |
| --- |
|  |
| Project Report |
|  |
| **3D-Programming, DV1541, 2015/2016** |

[Välj datum]

Written by : Martin Clementson and David Wigeluis

Technical Artist

Year 2, 2015/2016

Project Report

3D-Programming, DV1541, 2015/2016

# Comments and Considerations

The following project is a 3D Application that was done using C++ programming language.

We utilized the Direct X 11 API by Microsoft to achieve the final result.

The libraries we have used in this projects is:

* DirectXMath
  + This was used to be able to utilize the mathematical functions needed for our various operations such as matrix multiplication and vector operations.
* Algorithm
  + This was added to compliment DirectXMath with more functions such as MIN and MAX.
* Diput
  + This library enabled us to create our custom input handler.
* Vector
  + This library was used to utilize the standard template “Vector” that we used as our primary container.
* Fstream
  + This library was used to open and read from files. We used it for operations such as parsing our “.OBJ” and “.MD5” files, etc.
* String
  + This library was used to get the “String” type as well as the overloaded operations it provides.
* All the necessary DirectX libraries
  + DirectX has several libraries that were included for it to work properly
* DirectXToolKit
  + An unofficial library that provided the “WICtextureLoader” function that we used when loading textures and normal maps.

In our project we have implemented certain functions that can be turned on and off during runtime.

We have mapped these to hotkeys that the user can press to switch.

The keys and its functions are:

* W: This makes the camera move forwards in the direction it is looking.
* A: Makes the camera strafe to the left.
* S: This makes the camera move backwards.
* D: Makes the camera strafe to the right.
* The camera can be rotated using either the mouse or the arrows on the keyboard.
* Left Shift: Toggles the mouse visibility.
  + The camera cannot be rotated with the mouse when it’s in the visible mode (but the keyboard arrows is still functional).
* TAB: This toggles the mini map and the surrounding edge overlay on and off.
* T: When the user is above the terrain, pressing T toggles walking on top of the terrain.
* G: Toggles the post processing render pass. In this case Gaussian blur filtering is used in the post process.

# Core Techniques

## Skeletal Animation

# Geometry Techniques

## Parsing and rendering of .OBJ files

The concept of rendering a mesh from an “.OBJ” file is to assemble it from its own instructions. Usually achieved by reading the file using libraries such as “fstream.h” and assembling it within the code, then parsing the mesh through the graphical pipeline.

We achieved this by first creating a class called “OBJHandler” which main objective is, as the name suggests. To handle the reading of the “.OBJ” file. Our constructor uses the “fstream.h” library to read the file; we read the file line by line and store the read line into a string variable, which we check using different conditions. We store all of the read variables into std::vectors (seeing, as we do not know the size of the variables) and create them using a different function. After the loop has read the “.MTL” file name, we send it to a function that reads all the information in the “.MTL” file and indexes the texture file names (because some meshes use the same material). An “.OBJ” file uses groups to clarify which object belongs to which; to solve this we implemented a parent/child system so that we can manipulate the position of the mesh without it breaking.

In the next function, the “Create” function. We parse all the information gathered from the “.OBJ” file and create the object/objects. The function first controls whether it is the first mesh/only mesh or if it is a mesh-part relative to the original mesh. We then redefine the indexing of the “.OBJ” file to fit our parsing. Seeing as the in the “.OBJ” format, the indexing never resets and with our parent/child solution, we need to reset the index for each mesh for it to properly function with an index buffer.

## Height map

# Texturing and Lighting

## Normal mapping

# Projection Techniques

## Dynamic cubic environment mapping

A dynamic cube map is a “cubic texture” that keeps updating every frame, hence the name “dynamic”.

Dynamic cube mapping could be used to make dynamic reflections. The dynamic aspect is easily demonstrated by having animated objects in the scene, and see them move in the reflection.

Steps to this method:

1. Create six cameras, everyone looks down its own world axis (+x,-x, +y, -y, +z, -z ).

2. Position the cameras at the point where the reflective object is.

3. Render the scene for every camera but do not render the reflective object. Render to one texture each.

4. Put the textures together as a cube map.

5. Use the cube map as a texture for the reflective object.

In this project, we used this technique on one single object. This allowed for a higher resolution environment map. Dynamic cube mapping is heavily demanding which means that the more things that uses this technique the lower resolution you would have on the cube map. We opted to use a single cube map with a resolution of 1024x1024 to get a good quality reflection that represents the technique in a visually satisfactory way.

A class named “DynamicCubeMap” was created for this technique. The purpose of the class was mainly to contain the six cameras, their render target views, and the shader resource view for the cube map. It also contained necessary things such as a custom viewport and a depth stencil view

# Acceleration Techniques

## View frustum culling against a quad tree

## Back face culling using Geometry Shader

# Other Techniques

## Gaussian filter using a Compute shader