|  |
| --- |
|  |
| 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

The first thing to do with the skeletal animation was to parse a file containing the animation information.

We chose to parse the animation using the “.md5” file format. This was quite easy for us to do compared to a binary file type. Since we had already implemented the obj parser, we knew how to deal with the parsing of ASCII files.

The .md5 file consists of two separate file. One with the extension “.md5Anim” and one “.md5Mesh”. Which are quite self-explanatory. One file contains the information of the mesh and its skeleton. The other contains the animation in the form of key frames and the joints position in each key frame.

We split up our model into many subsets. This is mainly to make animation easier. Since each subset will have its own set of vertices, we do not have to update a vertex buffer for the whole mesh if only a small part of the mesh moved that frame. We only have to update the subsets that has been animated that frame.

The function “loadModel” handles the parsing of the “.md5mesh” file. It is similar to how we parse the “.obj” files.

This function creates the skeleton by reading the information about all the joints from the file. Then it creates the mesh itself, split into subsets.

Now that the mesh is loaded and can be rendered, the next step is to load the animation from the “.md5Anim” file. This file mostly contains information of the joints state in each key frame set.

When we load the animation data, we treat each key frame as a separate skeleton. The main concept of our animation technique is to interpolate between these skeletons each frame to get the animation effect.

We store the information in a struct called “ModelAnimation” this is mainly to be able to have multiple animations in a sorted manner. However, to demonstrate this technique we only have one instance of an animation.

The “update” function is what really handles the animation. This function is executed once each frame.

We find the frame we came from and the frame we are targeting while also finding where the current frame is between them. Then we create a temporary skeleton that we call “interpolatedSkeleton”.

This skeleton is the one that will rendered this frame. We interpolate the position and the rotation of the two skeletons using spherical interpolation. Then we update the main skeleton with the new vertices and update the subsets vertex buffers.

# 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. When the renewed indexing has been done, we calculate the tangent for the model vertices (to be used later in normal mapping) and send all the information back to the model class. In the Model class, we create a buffer for each of the objects and parse them down the graphical pipeline.

## Height map

The height map is a grayscale texture that stores information which is used to set the height of several vertices. This is usually applied to a flat plane. The texels on the height map determines how much each vertex on that plane should go up or down. The Idea is to sample one or several texels and use that value to determine the elevation of the vertex. Usually grey color is the middle ground, black represents the depth of the vertex and white represents the height.

In our project we created our plane based on the information from the height map. This ensured that we got the same amount of vertices as there are texels on the height map. This saved us from doing interpolation between many texels, if they would be “fighting” over a vertex, then we would have to interpolate the values into one. To ease our process, we created the vertices as we read the texels. The height map has a resolution of 512x512 which resulted in a big but still manageable terrain.

As we read the file, one texel at a time. We put the position of the texel in the texture as the x and z coordinates of the vertex to represent that texel. Then we sampled the texel and used as y value.

At first the whole map was loaded into a vertex buffer, but this was later changed when a quad tree was implemented. Please refer to the section of our quad tree implementation for further detail.

The creation itself was pretty straight forward. But the terrain was quite jaggy and did not really present a pleasant result. To solve this, we “smoothed” the terrain we just created. We implemented a function called “smooth” that would take the current height map and output a new smoothed one to replace it.

The algorithm for smoothing sampled the information from eight neighbour vertices and average them. We had to be careful not to sample from vertices that didn’t exist. This was a problem when you wanted to average vertices at the end of a row or column.

The camera had to be able to “walk” on the terrain. To solve this, we had a function in our Terrain class named “getYValue”. As inputs it received an x and a z value. This is essentially the coordinates of the camera when the function is called.

In our function we find where on the terrain the camera is and return the y value of that vert in the terrain

This resulted in a jaggy movement, because there were never any values to return when the camera was between several vertices. This made the camera “snap” when walking because it jumped from vertices to vertices.

To solve this, we sampled the information of the other three points that made up the quad we were standing on. Then we interpolated these points. The result was a smooth walking on our terrain.

# Texturing and Lighting

## Normal mapping

The concept of normal mapping is to give flat texture the appearance of bumps or depth when rendered. This is achieved by having a normal that is specified to each pixel instead of each vertex and then interpolated to each pixel. To get this information we sample from a texture that holds the information in RGB value. We can sample that and use it to create a normal for the pixel being processed.

The problem is that the original normal is in another space than the normal being sampled. So we need to transform the sampled normal into the same space as the original. But a problem still persists, every vertex in the object may have different positions, normals and texture coordinates which means that each triangle could be facing different directions and have other texture coordinates as other triangles in the same object. The solution is to transform the sampled normal into the space of each triangles space. This kind of space is called “Texture space” or “Tangent space”.

The first thing we do is to calculate each vertex tangent when we load our object in the “ObjHandler”.

We load a normal map in our model class and apply that as a sub resource in the pixel shader.

There is not much more to be done on the cpu side on this technique, since it is mostly handled on a pixel level.

In our pixel shader we have a function that is run if a pixel is set to process a normal map. This function takes the sampled normal, the original normal and the tangent. And produces a matrix that is used to multiply the sampled normal to change its basis to “texture space”. This matrix is usually called the TBN matrix. To Create this matrix we need the original normal, the tangent and the bitangent. The bitangent is achieved by taking the cross product of the normal and the tangent. In our function we also make sure that the tangent is completely orthogonal to the normal.

# 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.

The mentionable functions are “Init” , “buildCubeFaceCamera” and “Render”. The “Init” function is called in the Engine class. Here six cameras are created along with six render targets and a shader resource view in the form of a “texture cube”. The depth stencil and viewport is also initiated.

The “BuildCubeFaceCamera” function is called every render call. Its purpose is to position the cameras in the scene. Essentially this function would want the position of the object that is to have the resulting texture on it.

In our scene we have a sphere and we pass its position into the render function that sends it into this function. Here we update all the cameras to be centered at that position we send in and have their respective matrices updated. If there were more objects in the scene, we would have to do this for every object.

The Render function is the function called every frame from the engine. This is the only class were we send in a pointer to the whole engine object. This is because we want to access the Engine class functions “updateCamera” and “renderScene”.

The function receives a position which is intended to be the position of the model that is going to have the texture. It sets the shaderResource in the gpu to NULL before rendering. Since we can’t use the cube map as a render target and a shader resource at the same time. The reason we do this first is because it is set from the previous frame.

Then we call the “BuildCubeFaceCamera” function and set the cameras. We loop through each camera, for each iteration (i), we clear the I’th target view and depth stencil view. We set the cube map as render target and call the engine function “updateCamera” this function sends the current camera to the constant buffer that holds the view matrix. Then we render the whole scene through that camera.

After we have rendered all the six cameras we restore the viewport to the one of the player camera. We also set the render target back to the one used in the main rendering.

Lastly we generate mip maps of the cube texture and set it as a shader resource.

Custom shaders were used when rendering this. To sample the texture we needed a vector of three floats instead of the normal uv format, usually used.

The texture coordinates were set in the geometry shader. And it’s basically a vector from the object center to the vert being processed. Simpler put, it’s the local coordinates. In the pixel shader we sample the cube map texture and call it a day.

# Acceleration Techniques

## View frustum culling against a quad tree

## Back face culling using Geometry Shader

The concept of Back face culling is relatively simple. You create a vector from the Users camera position to the tested plane in the GS (Geometry Shader) and create a dot product. Then you test the angle between them to see if the tested plane’s normal is away from the cameras and if it is, it will not be rendered.

To implement this we sent the camera position into the GS, along with everything else. Now seeing as the vertices have their own normals (for smoother shading), we chose to recreate the test with a new normal. The new normal is a cross product from two vectors between the three vertices defining the triangle. That way we know that the normal is perpendicular to the triangle face. We normalize the new normal and multiply it with the inverse transposed world matrix. Then just like described above, we created a vector from the cameras position to the new triangle normal.

Now to test the normal we need to test the angle. The equation for an angle between two vectors is “cos A = v\*u/||v\*u||”, this can be simplified for simpler use. Seeing as we have normalized both the vectors in the equation, we can assume that their length is one (1\*1=1), and anything divided by one is the same. Which leaves us with the equation “cos A = v\*u”, a dot product.

We made the dot product with the negative of the “camera to triangle” vector. Because then the product will have to be over zero to be visible. After the calculations are complete we issued an “if” statement that checks the variable and if it is over zero the triangle will be rendered. This will apply to all objects/triangles that pass through the GS and the result is an increase in performance.

# Other Techniques

## Gaussian filter using a Compute shader

For our other techniques, we chose to implement a Gaussian filter with a CS (Compute Shader). To implement this we created a CS with [30][32][1] groups, [25][20][1] threads (800X600, which is our window resolution) and one kernel. We connected the CS to the back buffer and created a bool variable, controlled by the keyboard input. Because we connected the CS to the back buffer, the filter will be a post-processing filter that manipulates the output image.

Our Gaussian blur filter is a 7x7 matrix created offline. We chose to do this because it would be an unnecessary calculation to have in the program, considering that the filter is constant. The filter was created so that the result of the calculated colour will be one, which is important. Otherwise, the output image would be darker if it is below one or brighter if it is above one.

With our hard-coded filter as a constant in the CS, our kernel calculates the output image. We did this by assigning an int3 variable called “textureLocation” which will be used to traverse the surrounding pixels of the current pixel. The variable will be assigned a base value of the current [x,y] coordinate of the pixel minus (3,3,0). Then we assign a variable for the base colour called “finalColor” with the value (0.0, 0.0, 0.0, 1.0). The current thread will then enter a double for-loop in which the final colour variable will add the surrounding pixels colour value, based on the Gaussian blur filter matrix. For instance, if we’re at thread nr [3][3], it will first check the pixels colour value at [0][0], multiply that by the Gaussian blur filter value at [0][0] and finally assign it to the final colour variable. As mentioned above, using the Gaussian blur filter, the result of the variable using the filter will be one.

After we have sampled the value from the surrounding pixels, we assign the final colour variable to the output texture.