3D PROGRAMMING, DV1568: PROJECT

Joakim Ståhle-Nilsson
Blekinge Tekniska Högskola

2020ht, lp2

The purpose of this assignment is to further develop your 3D programming skills and get acquainted with several useful techniques in different areas of rendering:

- *Implementing rendering related techniques in a larger self-designed project*
- *Get experience with more advanced usage of the D3D11 API*
- *Rendering more complex scenes with different types of objects and techniques applied*

Preparation 1. Read the lab instructions

Read the *whole* document containing instructions for what needs to be done for the project.

End of preparation 1.

Preparation 2. Plagiarism and cooperation

The project is to be done either individually or in groups of two students. Working in groups of two is recommended but not enforced. While discussion and such is encouraged in the course, your implementation should be done by you/your group.

End of preparation 2.

Preparation 3. Preparative materials

Look through the material on the canvas page regarding techniques that you decide to implement and use the sources that you find useful. Other sources are acceptable as well, just make sure that they are trustworthy.

End of preparation 3.

1 Project description

In this project you will create a larger and more complex rendering application that utilizes several different established techniques related to rendering and 3D programming. There is no exact requirements of how the scene has to look and it can be adapted based on your needs and preferences assuming all requirements (both general and technique specific) are fulfilled. In section 2 you will find a list of rendering techniques divided into different sub areas. To pass the project you have to correctly implement techniques from each of the different sub areas (each have their own number of required techniques that need to be implemented) and be able to orally present both the theory behind the techniques as well as the implementation itself.

1.1 General requirements

- The program **must** be written in C++ and use D3D11 for the rendering and communication with the GPU.
- Names should be descriptive, in English, and the style of the code should be consistent throughout the whole implementation.
- Variables should only be global if there is a very good reason for it.
- Remember to use const and references for parameters when appropriate.
- The code should be able to compile using either Visual Studio 2017/2019, or by using g++. If g++ is used then a makefile should be provided that can compile the program. Visual Studio is recommended but not enforced.
- The `C++11`/`C++14`/`C++17` standards should be used.
- You may use the following external libraries, but you **must** then turn in a complete project that can be compiled without having to make any significant changes or that requires any external files.
  - DirectXTK
  - ASSIMP
  - stb
  - SDL2/SFML/GLFW (only for window creation/management/event handling)
- The debug layer of Direct3D11 must be active and the finished project should not generate any significant problems detected by the debug layer.
- The scene(s) **must** be rendered using a perspective camera that can move around in the scene using the keyboard.
  - It is allowed but not required to also use the mouse for camera control
- Lighting calculations **must** be performed for all the rendered objects in the scene using an established algorithm/methodology (for example phong) unless there is a very good reason not to.
- While there is no specific performance level needed/required, the different implementations must all be reasonable in terms of how they work and perform. Your program must not require a top of the line computer to run at an interactive frame rate.

2 Techniques

In this section you will find a list of sub areas related to rendering, and in them a list of techniques. For each sub area there will be a number in red that indicates how many techniques in that specific area that you have to implement for a passing grade. *Core techniques* for example require you to implement 1 technique, while *geometry* requires 2 techniques to be implemented. You need to implement the specified number of techniques for **each** of the six different sub areas.

You also need to be able to demonstrate that your techniques work. It is not enough to just say that they work. For tips about how this can be done most easily, see section 3.4.

2.1 Core Techniques

For this area you need to implement **1 (one)** technique.

2.1.1 Deffered Rendering

- **Must** be used for all objects in the scene except if there exist good reasons to not.
- Does **not** have to use any lighting optimization (tiled, light volumes, ...) but may do so.
- You do **not** have to minimize the information stored in the g-buffers (for example storing only depth and then recreating it in lighting pass), but it is encouraged to do so if possible.
- At least one light **must** be used for shading calculations in the secondary lighting pass.

2.1.2 Skeletal Animation

- At least one object **must** correctly perform animations with an appropriate mesh.
- The mesh and the animations may be loaded from file using ASSIMP if possible.
- The animation **must** be able to be performed any number of times, either by playing continuously, or by being able to be activated by the user at command.

2.2 Geometry

For this area you need to implement **2 (two)** techniques.

2.2.1 Model Format Parsing

- The whole process of opening the file, reading the data, and parsing it into a usable format, **must** be implemented by hand.
- You may **not** use ASSIMP or other libraries to help with the parsing process. Things like vectors from the C++ standard library for the purpose of storing data, or file streams for opening and reading from files are however OK.
- Only one format needs to be supported, but more are allowed.
- You may choose the format yourself as long as it is an established format like for example the obj format.
- At least one object in the scene **must** use a reasonably complex mesh loaded by your manual implementation.
- Some basic materials **must** be read from file manually as well and used when rendering.
- You do **not** have to support meshes formated into different separated parts, but you may do so.

2.2.2 Displacement Mapping using Tessellation Hardware

- At least one object with a relatively coarse mesh in the scene **must** have the technique applied to it.
- The effect **must** be clearly visible.
- The tessellation level can be fixed (hard-coded).
- It **must** be applied using the tessellation hardware (hull and domain shader) every frame.

2.2.3 Walkable Height-map Terrain Rendering

- The height-map data **must** be stored in, and read from, a file. It may **not** be hardcoded in the implementation.
- When the user traverses the terrain then their height **must** change based on the height of the terrain at the point the user is standing.
- The height values **must** be at least per triangle/vertex, but preferably interpolated between vertices in the current triangle or similarly.
- The terrain map can be fully regular (same size of triangles everywhere).
- The number of vertices horizontally and vertically can be the same as the texture but be careful with the size of the texture.
- You can also use a tiling texture and repeat the displacements in the terrain.

2.2.4 Level of Detail using Tessellation Hardware

- At least one reasonably geometrically-complex object in the scene **must** have the technique applied to it.
- At least three levels **must** be used.
- You may select the level based on a reasonable metric of your choice, such as for example distance to the camera, or, you may have it setup so that you can manually during runtime set the level used.
- If the technique is applied to terrain there does **not** have to be "perfect" alignment between patches.

2.2.5 Morph-based Vertex Animation

- At least one object **must** correctly perform animations with an appropriate mesh.
- The technique can be any form of morphing between keyframes for a set of vertices.
- The meshes may be loaded from file using ASSIMP if possible.
- The animation **must** be able to be performed any number of times, either by playing continuously, or by being able to be activated by the user at command.
- The key part is having a set of keyframes, along a time line, and being able to specify a time between 2 keyframes and get the vertices linearly interpolated. This can be done in the CPU every frame or store the keyframes in a buffer in the GPU and interpolate in the vertex or geometry shader (preferred).

2.3 Texturing and Lighting

For this area you need to implement **1 (one)** technique.

2.3.1 Normal Mapping

- The normal map **must** be correctly applied to at least one object. A TBN matrix must be built and used to adjust the normal on the GPU.
- If the object the technique is applied to is a flat surface (such as a floor or wall) then there must be multiple instances of the object in the scene with different orientations.
- The normals from the map **must** be used correctly when shading the object(s).
- An appropriate diffuse texture (or similar in other lighting models) **must** be used so that the effect is easily visible.

2.3.2 Screen Space Ambient Occlusion (SSAO)

- All objects in the rendered scene **must** be taken into calculation for the technique except if there exist good reasons to not do so.
- You **must** dynamically calculate the effect based on the current frame, each frame. You may **not** simply apply pre calculated texture data.
- The "shadowing effect" **must** be reasonably smooth.
- There **must** be several different parts of the scene in which the effect is clearly visible.

2.3.3 Blend Mapping

- Two or more textures **must** be applied on a single object/mesh and blended together based on some factors of your choice.

2.4 Projection Techniques

For this area you need to implement **1 (one)** technique.

2.4.1 Shadow Mapping

- At least one light **must** cast shadows on at least one larger object (for example the terrain). All other objects in the scene should be able to shadow the larger object(s).
- The shadow **must** be calculated each frame, and there **must** be at least one object in the scene that shadow the other object(s) that is dynamically being transformed in some way so that the shadows change each frame.
- The shadow does **not** have to be cast in a omnidirectional fashion from the light, but the area it casts to should be of reasonable size.

2.4.2 Dynamic Cubic Environment Mapping

- The map **must** be correctly updated each frame.
- You may choose how to apply the map yourself as long as it is in a reasonable/established way, such as for example to generate reflections on an object.
- It **must** be easy to see that the map is dynamically updated each frame and that it is omnidirectional.

2.4.3 Dynamic Paraboloid Mapping

- The map **must** be correctly updated each frame.
- You may choose how to apply the map yourself as long as it is in a reasonable/established way, such as for example to generate reflections on an object.
- It **must** be easy to see that the map is dynamically updated each frame and that it is omnidirectional.

2.5 Acceleration Techniques

For this area you need to implement **1 (one)** technique.

2.5.1 View Frustum Culling against a Quadtree

- The quadtree **must** contain all objects in the scene, and/or a very large and dividable object such as terrain.
- The quadtree **must** support at least a depth of four levels. It should however **not** be hardcoded to only work for exactly four levels.
- You do **not** have to update the quadtree each frame. It is enough to generate it once at startup, but it may **not** be stored in file and read.
- The quadtree can have an equal depth in all branches (regular), or be dynamic based on the contained objects (sparse). Either way it **must** support at least a depth of four.
- A view frustum must be used when checking against the quadtree itself.
- The culling **must** be performed every frame against all objects that the quadtree deems might be visible for the view frustum by checking the frustum against the objects bounding volume.
- You may choose yourself what type of bounding volume is used for the objects that are being culled.

2.5.2 Portal Culling

- At least one portal **must** exist in the scene and must cull against all applicable objects in the scene, and/or a very large and dividable object such as terrain.
- The portal **must** function correctly in both directions.

2.5.3 Back Face Culling using Geometry Shader

- You **must** have disabled the culling performed by D3D11 by explicitly creating and using a rasterizer state that uses `D3D11_CULL_NONE`.
- The culling **must** be performed each frame for at least one object in the scene.

2.6 Other Techniques

For this area you need to implement **3 (three)** techniques.

2.6.1 Particle System with Billboarded Particles

- Particles **must** be updated, billboarded, and rendered each frame.
- You may choose yourself what type of particle effect you want to create (rain, fire, snow, ...).
- Your particles **must** move in some way apart from the billboarding effect itself.
- The effect does **not** have to look physically correct or especially realistic.
- The system **must** work indefinitely. Rain can for example not stop falling after a set amount of time. Reuse "old" particles or create new ones.
- Particles may be updated on either the CPU or the GPU. If they are updated on the GPU however this task will count as **2** for the sake of reaching the number of required tasks for this sub area.

2.6.2 Gaussian Filter in Compute Shader

- The filter does **not** have to be generated by the program if so desired. It can be pre-calculated and used. This however does **not** mean that you do not have to be able to explain how they are generated.
- You may choose yourself how the filter is applied in the scene, but the effect must be easily visible when applied.
- You may choose yourself if you perform all of the blurring in one pass or if you divide it into two passes.

2.6.3 Bilateral Filter in Compute Shader

- You may choose the filter yourself, but it **must** be an established bilateral filter.
- The filter does **not** have to be generated by the program if so desired. It can be pre-calculated and used if applicable. This however does **not** mean that you do not have to be able to explain how they are generated.
- You may choose yourself how the filter is applied in the scene, but the effect must be easily visible when applied.

2.6.4 Picking using the Mouse

- The picking **must** be relatively precise.
- There **must** be some sort of noticeable effect of the picking. You may choose yourself what this is however.
- You may have the picking be continuous so that as soon as the mouse touches an object that can be interacted with something happens, or it can be performed only when a mouse button is pushed. The choice is yours.

2.6.5 Screen-Space Antialiasing

- You may choose the algorithm/methodology yourself, but it **must** be an established one.
  - SSAA is too simple and is thus not an acceptable algorithm/methodology.
  - MSAA is only allowed if implemented in conjunction with deferred rendering. The defferedly rendered/shaded parts of the scene need to have MSAA applied along with everything else in the scene in this case. Without deferred rendering MSAA is too simple and is therefore not considered an acceptable algorithm/methodology.
- The whole screen **must** be taken into consideration by the algorithm/methodology.

2.6.6 Water-effect

- The effect **must** be based on either vertex manipulation (CPU or GPU) or texture animations.
- If vertex manipulations are used and they are updated on the GPU this task will count as **2** for the sake of reaching the number of required tasks for this sub area.
- You may choose yourself if it is pure water that you render or some other liquid with similar properties.
- The effect does **not** have to be physically correct or very realistic
- You do **not** have to care how it looks in terms of refraction and caustics, or collisions with other objects in the water.

2.6.7 Glow-effect

- The effect **must** be limited to areas so that there are parts of the scene/an object that is glowing and not absolutely everything.
- You may (while taking into consideration the point above) choose how and where the effect is applied in the scene.
- The effect must be clearly visible when applied.

2.6.8 Make your application into a small game

- The game should be minimal, but **must** be playable.
- You can re-run the application to restart the game, you do not have to implement a feature in the project itself for restarting.
- The game **must** have some basic goal, and it **must** in some way involve the 3D scene that you have built (picking up objects, shooting and destroying targets, ...).
- The game **must** have some form of "win" condition, or some type of scoring system that serves as an end goal of some sort. You may also have some form of "game over" condition if you so desire.

3 Things to consider

This section contains a number of things that should be taken into consideration, as well as some tips. Read through it **carefully** so that you do not miss any relevant information.

3.1 Be prepared to put time into the project

This is the largest practical examination (8 hp). Historically many students put much time into it out of both interest and need. Do **not** underestimate the time to finish it. Start as soon as possible, and use your time efficiently.

3.2 Plan a design/structure for your code

While this will not be the largest project you will work on during your years of studying, it will most likely be the largest you have worked on so far by a fair marginal. Thus it is very beneficial if you take some time in the beginning to plan how you are going to design/structure everything. Making large changes late in the implementation is hard, time consuming, and frustrating. Things that might be worth considering are:

- How do you manage the different D3D11 resources such as buffers and textures? Does every object have it's own storage, or are they stored in a more centralized fashion?
- Does each object render itself, or are objects with the same type of pipeline collected and rendered externally?
- Different types of objects will probably need different types and amounts of resources to be set for a draw call. They will also most likely use different pipeline setups (some for example might need the tessellation stages, others the geometry shader). How is this all managed?

With all that said, it is also important to not overdo the design process or spend too much time or effort on it. The important part is to have some kind of plan that you try and follow so there is coherency in both the code and the flow of the program. This will make it easier for everyone involved.

3.3 If you work in pairs, plan how to do it efficiently

When you are working together with someone on a larger project it is important to consider how you manage the code that both of you write. Unless you are planning to code everything together (a perfectly fine methodology for this assignment if so desired) then you need some way to merge together your implementations. While not required in any way, tools like git and github are highly recommended for this purpose. However you manage the code, do not leave the merging until the last day. Make sure to do it regularly. Partially to not have to deal with major/massive merging conflicts at the end, but also to make sure that your implementations work together as you develop them.

The section about planning a design/structure for your code is probably even more important when you are working with other people. If you both follow the same type of design you are much likelier to create systems that work well with each other. Merging together two completely different systems will take time, and you risk running into problems related to how you handle for example resources or the rendering that may require major reworks of some systems.

Also do not forget to make sure that you and your partner understand both the theory and implementation of all the techniques implemented in the project. It is not enough to have a small speech ready about a technique that you have not implemented. You have to understand it and be prepared to answer questions about both the theory behind/regarding the technique, as well as the implementation details. Of course the person that implemented a specific technique will most likely know it best, but that does not mean that the other person does not have to know it as well.

3.4 Make sure you can show that your techniques work

As was mentioned earlier it is important to be able to demonstrate that your techniques actually work. It is not enough to just say that they work and show some code. Unless there is a good reason to, showing that your techniques work should not require a change in the actual code and recompilation of the project, but should instead be possible to handle during runtime. It might be beneficial to consider before/as you implement the technique how you can do it with that specific technique. Some techniques speak for themselves if they work or not as long as you can visually inspect them. Animations and rendered meshes that have been parsed from a file manually for example can rather easily be visually inspected. Below is a list of common approaches that can be useful for demonstrating the effects and applications of techniques. You may of course come up with your own ways, as the important part is that you are able to show that the technique is applied and works, not how you show it.

3.4.1 Toggling it on and off

Some techniques are easy to see the effect of if you can toggle it on and off. They make a clear difference when they are applied compared to when they are not. Adding some way to toggling them on and off with for example the push of a button is thus effective. There exist a good library for generating graphical user interfaces often used for these kinds of things known as *Dear ImGui* that can be useful to look into. Some (but not all) techniques where this is suitable are:

- Normal mapping
- SSAO
- Glow-effect

3.4.2 Rendering from a secondary camera

Certain techniques are not directly visible from the perspective of the camera, but if we could see the scene from another point of view we could see an effect. In those situations it might be beneficial to have a secondary camera that can be used for those situations. The primary camera should still be used in relevant calculations that care about it, but we render using the second one as our view matrix. It may also be beneficial to make sure that the primary camera has some type of indicator of where it is (such as a mesh that translates and rotates along with it). Some (but not all) techniques where this is suitable are:

- View Frustum Culling against a Quadtree
- Portal Culling
- Particle System with Billboarded Particles

3.4.3 Specialized methodologies

For some techniques the easiest thing to do is to do something very unique for them. For example deferred rendering shows no real visual difference when you toggle it on or off, and the rendered result is not per say affected by using a secondary camera. What you can do tho is you can render all the different g-buffers that the geometry write to, preferably simultaneously on different parts of a quad or using different viewports. Some (but not all) techniques where this is suitable/necessary are:

- Deferred Rendering
  - Render the different g-buffers to different parts of a quad or using different viewports
- Blend Mapping
  - Make it possible to tune the blend factor/methodology manually if blending is used

3.5 Resources for your scene

You are free to choose whatever resources (meshes, textures, maps, ...) that you desire and use them in your scene assuming some conditions are met.

- You **must** have the legal right to use the resource. You may **not** use resources that you either do not own or do not have the right to use for this purpose.
- The resources used **must** be appropriate. Do not use anything containing or alluding to sensitive, harmful, or inappropriate, material or content. Overall, use common sense, and do not include anything that you would not be ready to show in public.

3.6 We examine the technical aspects

It is not important to create a “cool” or “beautiful” scene. It is the technical aspects that are examined not the aesthetics. The different objects in your scene does not have to match. If an ancient Roman soldier is piloting a spaceship racing through a magically glowing castle is a scene that you think shows the techniques you have chosen well then that is absolutely acceptable. We also do not examine how "pretty" your meshes/textures are. What matters is that they fit the techniques you are using them for and make it clear that they work. It is of course also acceptable to try and make the scene “beautiful” or “cool” if that is what you desire, but it is in no way required.

4 Submission

Your submission should be a .zip file containing **all** the implementation files (header and source files), along with any other file that is necessary for compilation (for example a makefile if g++ is needed). If any of the allowed external libraries are used then make sure that all files needed for them are included in the zip and placed in the correct directories.

If you have worked on the assignment in a group, then both students need to hand in the code on the Canvas page of the assignment. In addition, the name of who you have been working with needs to be clearly stated in a comment on the Canvas page.

5 Assessment

After you have submitted your code to the Canvas page you will have to orally present your implementation for a teacher. This will be done individually (even if you have worked in a group) to make sure that everyone that is assessed has an understanding of what have been implemented and how it works. It is thus **not** enough to quickly/shortly explain certain techniques and not be able to answer questions about it just because that you were not the person implementing it. You need to understand every technique, not just say whatever your group partner told you to say about it.

This presentation will be handled through either Zoom or the discord server by using voice chat and screen sharing. The presentations will be handled at a separate date shortly after the submission deadline that you turned in for. It will thus **not** be handled in a lab session as the other practical assignments were. The presentation should be thorough enough so that the person presented to can get a clear understanding of how the program works, without getting into every detail possible. No PowerPoint or similar tool is needed, as the code, resulting program, and discussion, should be enough. The teacher may also ask the student to go into more detail about, or ask questions regarding, certain parts of the implementation and/or techniques.

Once the oral presentation has been passed the teacher will put a comment on the submission in Canvas to mark that the student has passed the oral presentation. After that a teacher will at some point perform a short review of the code. If nothing needs to be fixed after the review then the assignment will be graded as passed. There might however also be need for some fixing based on what was found in the review. If anything needs to be fixed then the assignment will be marked as Ux. The student then needs to perform changes based on the received feedback and then resubmit, at which point the code will be looked at again. Only one resubmission chance will be given at max, so it is therefore of utmost importance that everything problematic indicated by the feedback is fixed.