3D Programming, DV1542: Project

main authors: Joakim Ståhle Nilsson, Marcus Holmberg, David Lyhed Danielsson
(2024 version modified and adapted by Valeria Garro)
Blekinge Tekniska Högskola

Kursavveckling 2026-2027

The purpose of this project is to further delve into the area of rasterized rendering, the D3D11 API, as well as more advanced rendering related algorithms and techniques:

- *More advanced and complex usage of the D3D11 API and graphics pipeline*
- *Algorithms and techniques that are useful in rendering*
- *Designing and implementing a larger coding project*

Point 1. Read the project instructions

Read this document in its *entirety*.

Point 2. Plagiarism and cooperation

The project is to be done either individually or in groups of two students. While discussion and such is encouraged in the course, your implementation should be done by you/your group. Any act of plagiarism will be reported and issued in the BTH Disciplinary Committee https://www.bth.se/english/student/during-your-studies/rules-and-ethics.

Point 3. Preparative materials

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

Contents

1 General description

For this project, you are to implement several different rendering related techniques in one coherent project. You are allowed to work alone or in groups of two students (pairs). For those who work in pairs, there will be some additional requirements listed below.

Apart from following the requirements in this document, you are overall free to choose your own design both in terms of code structure, and scene structure. Part of the challenge will be to be able to design and implement a larger project that has to support the different techniques and requirements listed in this document.

1.1 General requirements

- 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 must be able to be compiled using only Visual Studio 2022 (no external libraries should be needed to compile/build/run your project, e.g. no premake, to avoid technical issues during the grading process).
- The `C++11` (or later) standard should be used.
- No external libraries (unless mentioned as allowed in this document) should be used or needed in your implementation.
- The standard library may be used freely.
- Your implementation must not generate D3D11 Errors (check the D3D11 ERRORs in the Output log in Visual Studio!).
- Your implementation must not generate memory leaks or have unreleased COM-objects at the end of execution (check the D3D11 and DXGI WARNINGs in the Output log in Visual Studio!).
- Your implementation must be designed in a way so that adding/removing objects is possible without having to do major changes to the code.
  - This means that hardcoding for example the number of objects taken into consideration for an algorithm/technique must be avoided.
  - Your project does not have to support dynamically adding/removing objects during runtime (even if it would be optimal for it to do so), and it is allowed to make minor changes to the code and recompile the project for the changes to take place.
- It must be possible to move the virtual camera around in the rendered scene using standard first person `WASD` (and eventually more for height manipulation) movement.
  - It is allowed (but not required) to use the mouse to orient the camera.
- Each rendered object in your scene must be textured and have **Blinn-Phong reflection model** lighting calculated per pixel. The textures must include patterns, i.e. they must not be 'single color' textures.
- You **must** calculate a per-frame delta value based on the time it takes to render the frame and use that to update your scene and camera. This is to make sure that for example rotations are done at the same speed regardless of computer.
- You **must** show that your techniques work, see Section 5.4 for details.

- You **must** add a README file (.txt, .doc, or .pdf format) to the submission explaining briefly how you showed each techniques, how your input key for the camera movement are configured, if you have toggle options which key triggers which features, and so on.
- Your lighting setup **must** be brighter enough for all techniques to be visible while running the program.
- You **must** use common sense while reading the requirements of this assignment. If you find a loophole in the text and you will use it to make a simpler but worthless implementation of a technique (e.g. super tiny particles only slightly moving, or a dynamic reflecting object located far away from other objects, or a light positioned in a way that does not cast any actual shadow in the scene, a tessellation without visible displacement, ...) it will be counted as a **conceptual error** and as **incorrect implementation**. Check with a teacher if you are unsure.
- Unprofessional language (e.g., swearing, imprecations even if mild) in any text, including comments and names of the variables is not tolerated in any case.

1.2 Allowed external libraries

This section contains a list of the external libraries that you are allowed to include and use in your project. If any external library is used, you need to submit necessary files when turning in your project, and it needs to be configured correctly to use those files. No manipulation of file structures, paths, or similar things should be required for the assessing teacher.

Each item in the list below is clickable to direct you to the web page of the library.

- stb_image.
- Dear ImGui.
- DirectX Toolkit.
- SDL (**Only allowed for window handling**).
- SFML (**Only allowed for window handling**).

2 Required techniques

The following is a list of techniques that you are required to implement in your project regardless of whether you are working alone or in pairs.

2.1 Deferred rendering

You are to implement a basic version of deferred rendering that uses a geometry pass that writes necessary information to G-buffers, and a compute shader driven light pass to calculate lighting. It is **not** required to use light culling techniques or compressing stored components, but it is allowed.

2.1.1 Requirements for deferred rendering

- The geometry pass should render all objects in the scene unless there are legitimate reasons to not include them in the deferred rendering (such as transparent objects).
- At least the following data must be stored in G-buffers
  - Position (unless reconstructing with depth buffer)
  - Normal
  - Ambient component
  - Diffuse component

  - Specular component
  - Shininess exponent

- The light pass should be done using a **compute shader** that for each pixel/texel calculates lighting using each light (unless light culling is implemented in which case only relevant lights should be used).

2.2 Shadow mapping

You have to implement shadow mapping for at least 4 unique lights in your rendered scene. You can choose how many of the lights are directional lights or spot lights, but at least one of each **must** be used. It is up to you how you position and orient the lights, but the effects of applying shadows using them must be clearly visible.

2.2.1 Requirements for shadow mapping

- All shadow maps **must** be stored in Texture2DArray resource(s), where each array element is a shadow map belonging to one of the lights that cast shadows.
  - You can either store all shadow maps in one single Texture2DArray or have two Texture2DArray resources; one for spotlight(s) and one for directional light(s).
- All information regarding each shadow casting light, such as world position and the view/projection matrices, **must** be stored in StructuredBuffer resource(s), where each array element contains information about a single light that cast shadows.
  - You can either store all light information in one single StructuredBuffer or have two StructuredBuffer resources; one for spotlight(s) and one for directional light(s).
  - For the depth/shadow pass (the first pass of this technique) it is acceptable to have a ConstantBuffer resource containing the index of the currently active light.

  Tip: Design this so that each light has the same index in both the Texture2DArray and StructuredBuffer resources.
- There must be at least 4 unique lights in the scene that cast shadows.
  - You are free to have more lights in the scene that do not cast shadows.
- The lights that cast shadows should be either directional lights or spot lights.
  - At least one of each type (directional/spot) needs to present and used in the scene for shadow mapping.
- There **must** be a simple object (e.g., a small sphere) located in the scene in the same position of each spot light, so it will be easy to see the location of the spot lights.
- The geometry pass in the deferred rendering technique **must not** perform any shadow calculations.

2.3 Level of detail using tessellation

You have to implement a version of level of detail that uses tessellation to increase the complexity of objects in the scene when closer to the camera/eye viewing the scene. The tessellation should be done on an object level so that you do not have to deal with problems related to "cracks" in the tessellated geometry.

2.3.1 Requirements for LOD using tessellation

- The implementation must use tessellation to increase the complexity of geometry dynamically
- The tessellation amount should be dynamically determined on the GPU during pipeline execution based on the distance between the object being rendered, and the eye/camera viewing it.

- Some kind of displacement must be done on the tessellated geometry.
  - Examples of this can be Phong tessellation and displacement mapping.

2.4 OBJ-parser

An OBJ-parser is a file parser that can read desired geometrical data from *.obj files, as well as desired material data from related *.mtl files. The data that will be loaded by the OBJ-parser should then be used to render objects in your scene.
You have two options here: either implement it yourself, or use an OBJ-parser that you find online and include in the project BUT in this case you must be able to explain the parser code and how OBJ/MTL works during the project presentation.
In case you decide to implement your own:

2.4.1 Requirements for OBJ-parser

- The parser needs to be able to parse *at least* the following geometrical data from *.obj files.
  - Position
  - Normal
  - UV (texture coordinate)
  - Submesh partitioning of the geometrical data for an arbitrary amount of submeshes
  - Material used for a specific mesh/submesh
- The parser needs to be able to parse *at least* the following textures from .mtl files.
  - Diffuse map (map_Kd)
  - Specular map (map_Ks)
  - Ambient map (map_Ka)
  - Specular exponent (Ns)
- Default textures for diffuse, specular, and ambient maps **must** be used if an .obj file does not contain a material.
- A vertex buffer is generated for each mesh, i.e. submeshes within a mesh share the same vertex buffer, and should not contain any duplicated elements.
- Index buffer(s) can be generated for each submesh, but should be generated once per mesh and be shared between submeshes. A submesh only needs to know from which index it starts in the index buffer and how many indices that submesh is using (see DrawIndexed and DrawIndexedInstanced).
  - In the OBJ file format, a submesh is often referred to as a *vertex group*.
  - There can be multiple ways to indicate when a new vertex group starts in an OBJ file, for example, a row starting with *g*, *o*, or *usemtl*. It is acceptable to support only one of these keywords for this purpose.
- Different materials may be applied on different submeshes within the same mesh. Some submeshes can be sharing the same material.
- When rendering a mesh, the vertex buffer can be set once, but each submesh can be renderered seperately using `DrawIndexed` or `DrawIndexedInstanced`.
- You can assume that the mesh data are triangulated.

2.5 Dynamic cube environment mapping

You have to implement dynamically updated cube environment mapping, also know as cube mapping. The environment map should be used to create a reflective surface in the scene.

2.5.1 Requirements for cube environment mapping

- The shape of the reflective object **must** not be a cube/cuboid, e.g. it could be a sphere.
- The cube map textures **must** be stored in a TextureCube resource.
- The cube map must be updated each frame.
- Some object(s) in the scene **must** move and be visible on the reflection, to visualize the real-time updating.
- Objects rendered in the reflections **must** be shaded as any other object in the scene, complete with light calculations.
- Particles and shadows are **not** required to be rendered in the reflections.
- Culling is **not** required when generating reflections.

2.6 Frustum culling using a quadtree/octree

You have to implement a culling technique that determines which objects in the scene have a possibility to be seen by the camera frustum, and use this information to only send these objects to the graphical pipeline for rendering. This is an *optimization technique* that is used to filter out objects that cannot definitely be seen by the camera. Each rendered object comes with some overhead just to render it; even if it is not seen in the final image.

2.6.1 Requirements for frustum culling using a quadtree/octree

- You choose whether to implement a quadtree or an octree as the tree structure.
- The tree structure **must** contain a reasonable number of objects in the scene (more than 20 objects).
- The tree structure's branches do not have to be dynamic. It is ok if all branches have the same depth level.
- The tree structure **must** support a depth level of at least three (3).
- Object information **must** be stored in leaf nodes, not in parent nodes.
- Only non-moving objects (i.e. static objects) need to be culled using the tree.
- A frustum has to be used for the culling against the tree structure.
- The collision library DirectXCollision (used by DirectXMath) is allowed to be used.

2.7 GPU-based billboarded particle system

You have to implement particles that always face the player and are updated on the GPU. The particles should be passed to the GPU as a collection of points. Each point should then generate a quad that always face the player - a technique knows as billboarding.

Particles that have completed their life cycle can be reused. If all particles are always reused, the number of particles in the scene will remain constant, which in turn simplifies the implementation.

2.7.1 Requirements for GPU-based billboarded particle system

- The geometry shader must be utilized to create the billboarded quads.
- The maximum number of particles can be configured at compile-time.
- The minimum number of particles is 100.
- The particle positions **must** be updated (with a new different position) on the GPU by using compute shader.
- The color of the particles can also be updated using the compute shader.
- The number of particles that are updated per thread should be "reasonable". Use 1-32 particles per thread as a starting number, and modify as seen fit.
- The number of particles per thread can be a constant given at compile-time.
- The size of the quads should allow the viewer to easily check if the billboarding effect is correct.
- There is no specific requirement for the behavior of the particles, only that it should be meaningful in terms of particle systems. Some ideas are: fire, smoke, fog, snow, and rain.

3 Additional requirements (when working in pairs)

If you are working in a pair, then in addition to the techniques listed above, you must also implement the 2 techniques listed below into your project.

3.1 Normal mapping

You have to implement normal mapping as presented in the slides. Not all objects in the scene are required to have normal mapping enabled.

3.1.1 Requirements for normal mapping

- At least one object must have normal mapping enabled.
- If the object to which you apply normal mapping is flat, then it needs to rotate to display that the TBN-matrix application is correct.

3.2 Parallax occlusion mapping

You have to implement the parallax occlusion mapping as presented in the slides. Not all objects in the scene are required to have parallax occlusion mapping enabled.

3.2.1 Requirements for parallax occlusion mapping

- At least one object must have parallax occlusion mapping enabled.
- If the object to which you apply parallax occlusion mapping is flat, then it needs to rotate to display the effect.

4 Assessment and Submission

The assessment of this assignment is divided into two steps. Note that if you have worked together with someone on the assignment, you **must** perform both assessment steps **individually**. This is to make sure that every assessed student knows what has been done and how it works.

4.1 Step 1: Canvas submission

Once you have completed the project, you can submit your Visual Studio project/solution on the course's Canvas page. Remember to check if your implementation follows the requirements included in the assignment description.

Your submission on the Canvas page should be a .zip file containing a **cleaned** project in which you have implemented your code. In the zip file you must also include the README file indicated in Section1.1.

If you have worked on the project in pairs, then **both** students need to submit the code on the Canvas page of the project assignment. In addition, the name of who you have been working with **must** be clearly stated in a comment on the Canvas page.

IMPORTANT: Check the canvas page of the assignment for how to clean the project before submission.

4.2 Step 2: Oral presentation

After you submitted the code on Canvas, you can sign up for the presentation of your project. The presentations will be done to a teacher during pre-planned times few days after the deadline. The dates and times will be available on the Canvas course calendar after the deadline.

During the presentation, the teacher will ask you to explain how your implementation works and ask questions about it. It is important to understand the details of what you have implemented. Explanations such as "I followed the book" or similar are not acceptable, as you must have an understanding of all the work you have done. The presentation should be around 1 hour and 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 and discussion should be enough.

If the oral presentation is deemed to be insufficient (the student shows no understanding of what s/he has done in the code), the student will have to present again. It is up to the teacher to determine if the problem was small enough that a new presentation can be done after few days (only small clarifications needed), or if the student needs to take more time to be able to sufficiently present during the next deadline session. Look at Table 1 for more details.

After you have submitted your implementation, the teacher will perform an inspection of the implementation and potentially give some feedback on it. Should no problems be found with the implementation the assignment will be marked as passed (**grade G**).

If something is found during code inspection that is deemed problematic, then feedback about the problem will be provided through the submission page as a comment and the student will be able to submit a fixed version of the implementation for approximately two weeks after getting the feedback (**grade UX**).

Once the resubmission has been done, the teacher will inspect the new implementation. If the problems noted in the feedback are fixed and no new problems are introduced, the teacher will mark the submission as passed on the submission page on Canvas. If no resubmission is done or the resubmission either still contains noted problems or has introduced new ones, then the assignment will be marked as not passed (**grade U**).

If the implementation does not fulfill all the requirements and/or it presents several mistakes, the assignment will be marked as Fail (**grade U**). Table 1 shows a recap of the assessment criteria explained above.

§ : Conceptual errors are defined as mistakes that show lack of understanding of the theory explained during the lectures and how to implement those concepts in Direct3D11.

*** : IMPORTANT! in order to pass the project assignment (grade G on Ladok) the student must receive a G on both criteria (Implementation AND Presentation).**

5 Important things to consider and additional requirements

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. Section 5.4 is very important since one of the requirements is to demonstrate visually that your techniques work.

Table 1: Assessment Criteria Matrix for this assignment

Criteria*	G Passed/Sufficient	UX Insufficient (minor rework required)	U Fail
Implementation	The implementation includes all the requirements presented in Sections 1.1 and 2 (and Section 3 for pairs). The code is correct or it presents few small errors that are **not** *conceptual errors*§.	The implementation includes all the requirements presented in Sections 1.1 and 2 (and Section 3 for pairs). There are only few small errors and/or only one or two conceptual errors.	The implementation is missing one or more of the requirements presented in Sections 1.1 and 2 (and Section 3 for pairs) and/or it includes several errors, also conceptual.
Presentation	The student shows a good and complete understanding of the code s/he implemented. No conceptual errors related to the theory applied in the code.	The student shows a good understanding of the code s/he implemented. Only one or two conceptual error or few small inaccuracies.	The student does not show a good understanding of the code s/he implemented. Several conceptual errors or small inaccuracies.

5.1 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 necessity. Do **not** underestimate the time to finish it. Start as soon as possible, and use your time efficiently.

5.2 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. 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 its 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 as a batch?
- 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?
- Aim to be able to easily change the number of objects in the scene without much effort. All objects in the scene may be stored in a general container (e.g. `std::vector`) and not be stored as separate member variables in a *Scene class*. Make the update and render functionalities depend on any amount of objects, for example, iterate over the generic container, instead of you manually calling update and render for each objects added to the scene.

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.

As extra material in Canvas you can find some header files that you are allowed to use (not mandatory) that can give you an idea on how to structure your code (AbstractionHeaders.zip).

5.3 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 the website 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.

5.4 You MUST show that your techniques work

As mentioned earlier, it is a requirement 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. 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.

5.4.1 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 toggle them on and off with for example the push of a button is thus effective. There exists a good library for generating graphical user interfaces often used for these kinds of things known as *Dear ImGui* that can be useful. One technique where this could be suitable is level of detail using tessellation, to show wireframe rendering.

5.4.2 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. 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 necessarily all) techniques where this could be suitable are:

- View frustum culling against a quadtree/octree
- Particle system with billboarded particles

5.4.3 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. One technique where this could be suitable is:

- Deferred Rendering: render the different G-buffers to different parts of a quad or using different viewports

5.5 Resources for your scene

You are free to choose whichever resources (meshes, textures, maps, etc.) 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.

5.6 We examine the technical aspects

It is not important to create a “cool” or “beautiful” scene. The technical aspects are examined, not the aesthetics ones. 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 that you make it clear that they work. Trying to make the scene “beautiful” or “cool” is, of course, also acceptable if that is what you desire.