3D Project Report

DV1568 & DV1542

Björn Johansson
Lowe Raivio
Victor Falkengaard Itzel

Workflow

Two of the group members developed on Windows 10 with Visual Studio 2017 (MSVC) in x86, while one member developed on Debian Sid with Sublime Text 3 (Clang++7 & Makefile) in x64. We feel like this gave us various benefits during the development cycle; with the obvious benefit being that the code's cross-platform compatibility was actively tested and another big benefit being that certain bugs that were a consequence of undefined behaviour were sometimes easier to detect since they'd manifest themselves differently for us. It also allowed us to check for memory leaks thoroughly by running both CRTDBG and ValGrind.

Libraries

We picked GLEW as the OpenGL extension wrangler;
GLM as our maths library;
GLFW as our window, OpenGL context, and input library;
Assimp as our asset (model) loading library;
Dear ImGui as our debug GUI library;
STB for bitmap loading;
and STL & STD for standard functionalities.
We also had Better-Enums for a while but dropped it due to complications, and we had planned on using fmt for string formatting and range-v3 as an STL alternative.

Demo scene

The scene consists of a tessellated floor plane, a tessellated funky monkey, an army of non-tessellated monkeys, a snowy particle system, a beatific collection of various colourful point lights, a shadowcasting directional sunlight (lightsource 9), as well as some debug components.

The demo also provides both a debug GUI that is toggled with F12, as well as various key and mouse bindings to facilitate interactions with the scene.

Directional movement can be done with the WASD and CTRL/SPACE keys, and free flight movement can be toggled with F1. The middle mouse button can also be held to rotate the view even without the free flight mode. A GUI window contains a slider to change the movement speed.

The key P triggers a funky monkey dance party.

The F2 key toggles between wireframe and face rendering.

Render mode can be switched with the following keys:
F3: full shading (default; composite of the channels)
F4: albedo channel
F5: normal channel (viewed in object space)
F6: specular channel
F7: position channel
F8: emission channel
F9: lighting mode (composite shading sans the albedo)
F10: ID channel (colour attachment used by mousepicking)
F11: displacement channel (colour attachment used by displacement tessellation)

ESC: used to exit the demo when you're feeling lazy

The debug GUI offers a window with sliders for the displacement factor and tessellation level percent.

There's also a window for each lightsource that provides some means of modifying their various attributes. 0xFF00FF

There's also a window listing all the current model instances and exposing access to their data.

Project structure

SceneManager	- provides an interface to instantiate scene-specific components
Viewport	- encapsulates the user's viewport (camera), including window information
Light	- encapsulates various lighting data and lightsource types
Shadowcaster	- wraps a lightsource to add shadow-casting functionality
ParticleSystem	- encapsulates things related to particles and their usage
AssetManager	- provides some functionality for loading various assets
Mesh	- encapsulates meshes, their functionality, and relevant data
Model	- encapsulates models, their functionality, and relevant data
ModelInstance	- wraps a model to avoid duplicating data on the CPU when instantiating a model
Texture	- encapsulates a bitmap and some relevant data
TextureSet	- collection of textures of each type; default generated as 1x1 textures if not provided
ShaderManager	- encapsulates functionality related to loading shaders and compiling shader programs
Shader	- encapsulates a shader and its related functions
ShaderProgram	- encapsulates a shader program and its related functions
Transform	- a composite class that encompasses both scaling, rotation, and translation transforms

Miscellaneous files:

defs.h; a header file that gathers most of our major definitions, using aliases, as well as some include directives in one place.

debug.h/cpp; gathers most of the debug functionality in one place.

Config.h/cpp; gathers most of the global scope configurations in one place.

Shader Programs

Default Geometry Pass Shader Program
Tessellated Geometry Pass Shader Program (shares the fragment shader with the default geometry pass
Lighting Pass Shader Program
Particle System Shader Program
Shadow Depth Shader Program

Implemented Techniques

Particle system

Computation options: GPU-computed particle logic or CPU-side logic.

Pros of CPU alternative: easier to for the engine developer and engine users to exchange custom particle logic.

Pros of GPU: more suited for highly parallelizable code.

We made the constructor accept a lambda wrapping the logic and a set of textures (albedo, specular, normal, emission) for the particle billboards. The lambda gets free access to particle data and can modify the particle values (lifetime, position, colour, scale factor, etc).

Implementation options:

Using instancing (decent compatibility & efficient; our current choice).
Using geometry shaders (limited compatibility).
A VBO with all the particles (most compatible).

Issues to be aware of:

Transparency; the easiest workaround (which we picked) is to sort the particles back-to-front and draw them in that order since due to their billboard nature the only cases where overlapping is an issue is when the distance of two billboards is identical and they overlap within their planes, which could lead to minor Z-fighting.

However, they can still intersect in very noticeable manners with other geometry (from the regular geometry pass). One potential way of dealing with this is to blend the current particle fragment based off of its proximity to the depth buffer at the same pixel.

Animated and "multi-textured" particle systems can be realized by implementing a sheet of textures--a texture atlas. That way the frame number or local texture ID can easily be converted into a sub-rectangle within the atlas.

Another way of achieving multiple textures is to just create one particle system for each texture variant and then put them in the same location.

Limitations:

The maximum particle count (per system) is hard-coded due to the need to pre-allocate the memory when loading it. However, this can be circumvented in a somewhat hacky way by simple creating more particle systems in the same location with the same textures and logic.

A current limitation is that we had to add the following check in the particle system fragment shader to by-pass a minor issue:

```
"if (g_albedo.w < 0.525) discard;"
```

This is because we use the alpha channel in certain texture's colour attachments to encode additional data (such as specular intensity for the specular textures). But this issue is easy to fix by simply changing the shaders to use the value of the RGB parts of the texture both for colour and intensity.

To handle the particles' lifetime tracking we just use std::partition() on the particle data with a unary predicate lambda that compares whether any given particle has expired, then the returned index becomes the new particle count.

Reflections:

We're considering implementing a geometry shader-based solution just to benchmark and compare the two to see which one performs the best.

Front-to-back rendering

In order to accomplish this acceleration all that is needed is to maintain a sorted order of the model instances that is based off of their distance from the viewport's camera position.

Depending on one's needs, a quad tree or octree would be an ideal data structure for storing them. Then it's just a question of traversing the tree based off of the camera position. Even for 3D games, many games have a 2D layout (that is, having the X and Z axes be the longest, while the Y axis is short). The easiest way to achieve this, however, is to just store references to the instances in a vector and then use STD's sort algorithm with a custom lambda that compares two instances' distances from the viewport, with the latter being passed via the lambda capture.

One small optimization that we did is to have our SceneManager class hand our model instances a callback function that they call every time they get transformed. If a change happens, a flag is set that frame to sort the instances before iterating them for drawing. Of course, this will also need to be called if the viewport changes its position as well. The drawback with this naïve vector method in our case is that we keep all instances in one vector. When it comes to our particle system, however, we don't use front-to-back rendering but rather back-to-front rendering in order to handle transparency in a non-complicated manner.

Backface culling

Compare the triangle normal (in the geometry shader) to the camera's facing direction to determine whether it is facing towards or away from the beholder. Using the backface culling built into OpenGL is both easier and gives better results.

Deferred Shading

We created a G-buffer with the following colour attachment channels:
- Albedo (RGBA colour; with the default being 0x808080FF)
- Specular (specular colour encoded as RGB and intensity encoded as A; with the default being 0x80808080)
- Normal (tangent space RGB; with the default 0x8080FF being neutral)
- Emission (lights baked into RGB; basically baked lights, with the default 0x000000 being neutral)
- Position (R for X, G for Y, B for Z; with the default being 0x000000)
- Displacement (bidirectional factor encoded as luma; with the default 0x808080 being neutral)
- Instance IDs (encoded as non-interpolated RGBA colours, with the default 0x00000000 implying NONE)

We use the albedo, specular channels, and normal for Phong diffuse shading and Blinn-Phong specular shading.
The emission channel was added at our own accord because we were interested in trying it out and since the implementation of it is trivial. The data in this texture colour attachment just gets added during shading as if it were the light impact from some lightsource.
The position channel is used for lighting and occlusion computations to produce the direction the fragment is viewed at.
The displacement channel is used for displacement tessellation.
The instance ID channel is used for mousepicking.

Tessellation

We use both a evaluation shader and a control shader to handle tessellation, our control shader takes patches with 3 vertices and only passes the input patch to the evaluation shader adding no extra control points.

It is also being used to set the tessellation levels based on the distance from camera to a point between each side of our patch.
A function is used to set the tessellation level, when the distance is far enough the level will decrease with quadratic falloff in order to get a smooth transition between displaced and non-displaced areas.

A uniform called tess_percent is used to let us change the overall tessellation level in our application, it takes a value from -1 to 1 (floats), if we set this value to -1 it will remove all tessellation and setting it to 1 will tessellate everything within a fixed range.

Displacement mapping is done with the evaluation shader, simply by sampling a tessellation map using our interpolated UV-coordinates. We can change the amount of displacement using a uniform, in the future we will also include a specified factor to each model instance, we need to do this since tessellating models with a different amount of vertices will affect how much impact the factor has, currently all tessellated models have the same factor.

Since we most likely do not want to tessellate all models, a boolean value in our ModelInstance class is used that specifies whether the normal geometry shader program or the tessellation shader program should be used. Since all tessellation shader programs need patches as input, we also use the boolean value to decide if we should draw patches or vertices with the glDraw functions.

Shadow mapping

We create a shadow map using a custom shader program that only handles a depth buffer, this is done before we perform our deferred rendering's geometry pass. Instead of implementing PCF we've decided to use a slightly higher resolution of 2048x2048 for our depth map texture, as the magnifying filter linear interpolation is used to get a smoother shape. However, PCF would have allowed the shadows to cover a larger area without us having to change the shadow map resolution.

For our demo we've also decided that we don't want to light the areas around what's being affected by the light in order to get a more accurate picture of what is being lit and what part is going to cast shadows. While in a game scenario it probably won't behave this way in the most common scenarious. However, since it could still occur for a select few cases it might be wise to implement this as an option per shadowcaster instance.

Our solution is so far only possible with directional lights, but we have designed it with support for other light types in mind. To create a light that casts a shadow we first have to create an instance of our Light class, then we can create an instance of our ShadowCaster class which takes a shared pointer to a Light as its constructor parameter. Based on that light we then create a light matrix that we use to generate our shadow map.

In order to use this shadow map we first have to register it to our SceneManager using a function called set_shadowcasting, we do this since the SceneManager will hold all models that eventually might cast a shadow. The SceneManager controls how the shadow map is created and under what condition it is set to be updated as well as holds the shader program for the aforementioned shadow map.

There are two conditions that may cause a shadow map to update on any given frame. The first condition is if one of the model instances have been transformed and the second condition is if the shadowcasting light has changed. This way it will at most update once per frame. This is done by letting our SceneManager know that either of them has changed, with the use of a callback function that they were given by the SceneManager during instantiation; this trigger will then change a boolean value that gets checked on every frame. If it has changed, we will update the shadow map and then flip the boolean value again so that we don't have to update it until next time something related to the shadowcasting changes. In order to more effectively see how the light changes a ShadowcasterDebug class was introduced, it simply marks the area our shadowcaster has affected by placing different coloured monkeys at the ends of the light's frustum.

Normal mapping

The texture is loaded with the help of Assimp which also calculates the normal, tangent and bitangents from the mesh. These are then sent as uniforms to the shaders. In the shaders we combine the vertices into a TBN matrix. When we do our normal sampling from the normal texture, in the fragment shader of the geometry pass. We then multiply it by our TBN matrix.

Mouse picking

We chose to implement a texture that colour codes all the objects that we want to be able to do picking on. So, our SceneManager gives every ModelInstance a distinct ID (number) starting from 1. These numbers are then bitshifted so that we get a vec4 with 8 bits of our 32-bit number in each of the four spaces. This vec4 is then sent up as a uniform to the fragment shader of the geometry pass. After this we then read our texture from the correct G-buffer color attachment and get our RGBA colour. This vec4 is then bitshifted the opposite way and added together into a 32-bit integer which is our unique ID for a model instance. Then this is used by the SceneManager to search through our vector of model instances to get the correct model and its pointer. In our demo, the pointer is then used to rotate the picked model instance to demonstrate that it works.

Memory Leaks

We ran both Valgrind and CRTDBG to check for memory leaks, and had various potential leaks; but to the best of their knowledge they originated in some of the libraries that we use (Assimp, STB, potentially GLFW). We ran the demo for an extended period of time and noticed that the resource footprint did not climb but rather subsided and plateaued after the assets had been loaded and instantiated. Also, throughout the project we made use of RAII (Resource Acquisition Is Initialization) to encapsulate the lifetimes of data instances with value semantic and automatic resource management.

Reflections

We all found the project to be very educational and fun, and we would like to continue expanding on it on our free time to implement more techniques to further our knowledge and expand our minds. Ideally, in the context of front-to-back rendering we'd like to have a separate container for instances within proximity. The predicate for this could either be a simple function that implements a distance threshold to determine eligibility; or if a level-of-detail system is used, the LoD could also factor in. Then it also depends on what kind of game the engine is to be used for. For a game like Minecraft, the octree approach would be ideal; for a real time strategy game, a quad tree or cell grid system (where each cell contains a set of instances) would be good.