CSC 305 Assignment 2 – Real-time Renderer

Due Sunday February 26th, 2017

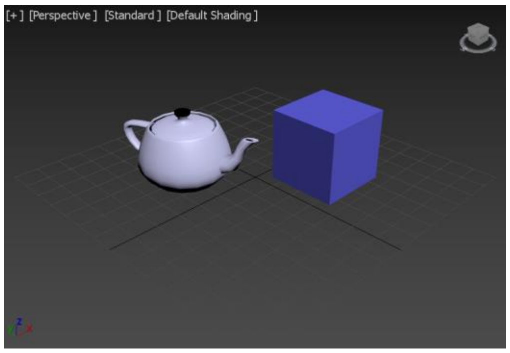


Figure 1: A view of a 3D scene in Autodesk 3ds Max 2017, using some basic shading

Figure 2: An advanced lighting technique demonstrated on the Crytek Sponza scene (by "Synce")

Overview

In the first assignment, you were asked to render a scene using CPU ray tracing. In contrast, this assignment asks you to render a scene using GPU rendering techniques. Like the first assignment, there are standard and advanced requirements.

Note: GPU code tends to be hard to make cross-platform. You may work on your own computer, and your code doesn't have to work on the lab computers. In other words, "works for me" is a valid excuse, and in-fact is generally encouraged. (Don't let the man keep you down, break the rules.)

Submission

For your submission, please submit your code and a short report explaining what you did. For each bullet point in the requirements, or for any other problem you thought was interesting, add a paragraph to your report that explains:

- "What": What is the problem?
- "How": How will you fix it?
- "Why": Why is this problem useful to fix? Why is your solution a good choice?

Use screenshots to demonstrate the features you implemented, and refer to them in your report.

DirectX?

The supplied code uses OpenGL, but you're welcomed to try doing the assignment with DirectX 11 or 12. This is probably more useful to learn if you want to work as a rendering programmer in game development, but the learning curve is steep and the online resources often aren't as good as OpenGL. If you choose the option, you have a lot of work ahead of yourself, and you should ask me (Nicolas) lots of questions, because otherwise you probably won't be able to finish satisfactorily.

Understanding the Supplied Code

The architecture of the program is mainly separated into 3 classes: Scene, Simulation, and Renderer.

Scene

“Scene” is like a database, or the “model” in “model-view-controller”. It contains the data for your scene, but doesn’t define how it needs to be rendered.

Each of the “tables” of the database are made using packed_freelist, a custom C++ container that allows insertions/deletions in random order while keeping the elements inside contiguous in memory. Instead of pointers, you use IDs to refer to objects inside it. The implementation of packed_freelist is roughly explained here: http://bitsquid.blogspot.ca/2011/09/managing-decoupling-part-4-id-lookup.html

Scene also contains some utility functions for loading 3D meshes and materials, and for adding instances of meshes into the scene.

Simulation

“Simulation” is where you can put code that updates the “Scene” based on the logic your app.

Simulation::Init() is called at the start of the program, so you can initialize your scene.

Simulation::HandleEvent(SDL_Event ev) is called so you can handle OS events like mouse movement or keyboard presses. In general, you want to look at “ev.type” to know the type of the event, then access it differently based on the type. See https://wiki.libsdl.org/SDL_Event#reading

Simulation::Update(float deltaTime) is called once at every frame of the renderer. It lets you move every object forward in your scene by the small amount of time that passed since the last update. deltaTime is stored in seconds.

Renderer

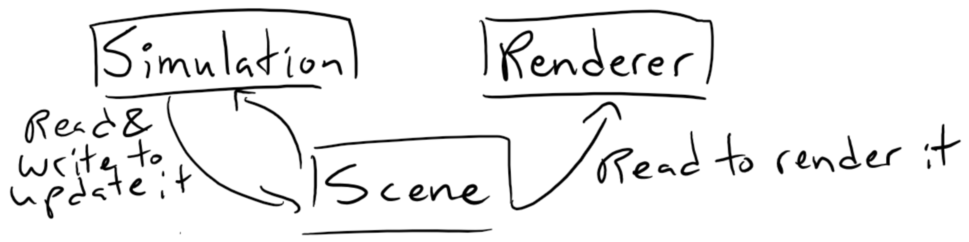
“Renderer” is where you can put code that reads and display the “Scene”.

Renderer::Init() is called at the start of the program, so you can create any resources up front.

Renderer::Resize(int width, int height) is called once after Init(), then called every time the window is resized. You can use it to create any resources that depend on the window’s size, like for example to create a texture you can render to, which fills the whole window.

Renderer::Render() is called at each frame to render the Scene. This function should read the Scene’s data, and render it using OpenGL.

The Big Picture



Understanding the Supplied Code (continued)

I wrote a long blog post about the design of the renderer in the supplied code. It's a little bit rant-like, but you might find it useful: https://nlguillemot.wordpress.com/2016/11/18/opengl-renderer-design/

Shader Live-Editing

The supplied code supports live-editing of shaders. That means that if you edit and save a shader's source code while the program is running, the shader will automatically update itself in the program. **Therefore, you don't need to restart the program to test changes in your shaders**. This is implemented using my ShaderSet class.

I wrote a blog post that explains how ShaderSet works here, which you might find helpful: https://nlguillemot.wordpress.com/2016/07/28/glsl-shader-live-reloading/

Calling OpenGL Functions

To call OpenGL functions, you can include "opengl.h". This will give you access to all functions and constants defined by the Core OpenGL spec. If you don't find a function in this file, that's because it's either: (1) deprecated and shouldn't be used, or (2) a vendor-specific extension, which you probably don't need for this project.

Automatic OpenGL Error Checking

The supplied code automatically checks every OpenGL function call for errors when you're in a debug build. If your computer supports the GL_ARB_debug_output OpenGL feature, you'll get detailed error messages when you misuse OpenGL.

If your computer doesn't support GL_ARB_debug_output (like if you're using OSX), then it'll fall back to glGetError(), which gives you an error code that you have to compare to the documentation.

In both cases, you can find the offending line of code by going up the call stack in a debugger.

Loading Images

Loading images is done using stb_image. There is some example code using it in scene.cpp. There is already code to load the images you need for the standard assignment requirements, so you should only have to load additional images if you're doing the advanced requirements.

Loading Meshes

Loading meshes is done using tinyobjloader. There is some example code using it in scene.cpp. There is already code to load the meshes you need for the standard assignment requirements, so you should only have to load additional meshes if you're doing the advanced requirements.

Interfacing with the OS with SDL2

If you want to access OS features in a platform-independent way, you can use SDL2 for that purpose. SDL2 is the library used in the code to create a window and read user inputs. You can find the documentation here: https://wiki.libsdl.org/FrontPage

Adding GUI elements with ImGui

A GUI system comes hooked up with the supplied code. It uses the "immediate mode GUI" paradigm, which is probably different from what you've seen before. This allows you to easily add GUI controls to functions like Simulation::Update() and Renderer::Render(). Look at imgui_demo.cpp for example code, and see the “Example GUI Window” code in simulation.cpp.

You can find more information on ImGui’s GitHub page: https://github.com/ocornut/imgui

The principles of immediate-mode GUIs are explained by Casey here: https://mollyrocket.com/861

Camera Controls

The supplied code comes with a camera view. You can activate camera controls by holding the right mouse button. From there, you can move the mouse to change the camera’s look direction, and you can use the WASD keys to move the camera position. The code that updates the camera can be found in simulation.cpp.

Supported OpenGL Versions

The supplied code is designed for OpenGL 4.1, because that’s the version supported on OSX (and Apple has no plans to update it. **Ever.**)

If your computer supports a more modern version of OpenGL, you can take advantage of it by modifying the code in main.cpp that selects the OpenGL version to use. You can also use a newer version of GLSL by modifying the SetVersion(“410”) statement inside renderer.cpp.

If your computer only supports older versions of OpenGL, you can modify main.cpp to downgrade the OpenGL version all the way down to version 3.3, and you can set the GLSL version to 330. It should still work, in theory.

If your computer only supports OpenGL 3.2, it’s still possible to workaround the problems. Go see Nicolas and we’ll figure it out.

If your computer only supports a version of OpenGL that is older than that, you are officially among what game developers refer to as “The Damned” ☺. Consider using the lab computers, or ask Nicolas to help you figure it out.

Standard Requirements (70%)

- In the vertex shader, transform vertices to clip space using the ModelViewProjection matrix.
  - See the scene.vert shader.
  - After doing this, you should see the silhouettes of the objects.
- In the pixel shader. implement Phong shading for a point light placed at the camera's position.
  - Use the object's diffuse map texture for the diffuse color.
  - The diffuse map is already hooked up the scene.frag shader.
- Implement a hierarchy of transformations (a "scene graph") to place objects relative to others.
  - Extend "struct Transform" in scene.h to have a "ParentID"
  - From then, each Transform is considered relative to its parent: Transforms[ParentID].
  - ParentID == -1 represents a root node.
  - At each frame, compute the absolute transform of each transform (traverse its parents.)
  - Use the absolute transform instead of the relative transform in your rendering.
  - Show that this works by rendering an object that orbits around another.
- Implement a directional shadow map (from the sun.)
  - Create a separate FBO and depth texture to render a shadow map.
  - Make sure the depth comparison mode is set on the texture. (see glTexParameter docs)
  - Render the scene into the shadow map in a depth-only pass.
  - Bind the shadow map as a texture to your scene fragment shader.
  - Sample the shadow map with sampler2DShadow using the output of the light matrix.
  - Incorporate the shadowing into your lighting computation
  - There are good shadow map tutorials on the web. Check them out.

Advanced Requirements (30%)

- Render the Sponza scene (see http://www.crytek.com/cryengine/cryengine3/downloads)
  - Implement its diffuse maps, specular maps, bump maps, and alpha masks.
  - You must extend the scene loading code in scene.h/.cpp to handle these materials.
- Implement a skybox.
  - Create a cubemap texture where the 6 faces correspond to the skybox faces.
    - You can grab a cubemap from the web.
  - Add a skybox rendering pass after rendering the scene.
  - Make sure to use depth testing to avoid rendering the skybox where it isn't necessary.

Bonus Requirements (10%)

- Render multiple lights using light volumes using a deferred shading pass:
  - Add GBuffer textures to the scene FBO that stores normals and material properties.
  - Write to the GBuffer textures from the fragment shader of the first scene pass.
  - This should give you a texture that stores the normal and material at each pixel.
  - Create an FBO for the light pass that renders light volumes onto the backbuffer.
  - Create new shaders lights.vert and lights.frag to render light volumes.
  - lights.vert mainly does the typical MVP transformation.
  - lights.frag reads the depth buffer texture to reconstruct position, reads normals and material properties from the GBuffer, computes lighting from light volume with these inputs, and blends the light's color into the backbuffer.
  - Call this shader with meshes that represent the volume of the light (eg. a sphere mesh).

Additional Resources

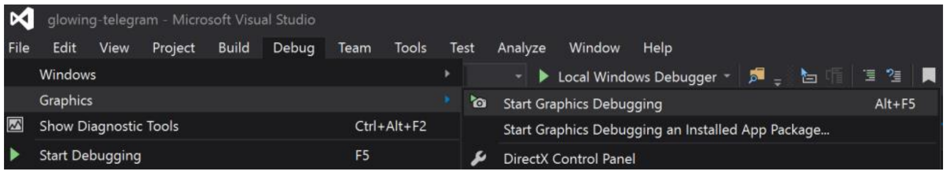
- Use the OpenGL Wiki to learn more about OpenGL: https://www.opengl.org/wiki
- If you have OpenGL bugs, read this first: https://www.opengl.org/wiki/Common_Mistakes
- You can search for OpenGL functions documentation using http://docs.gl/
- There are plenty of good tutorials and guides about OpenGL on the web, so look out for them.

OpenGL and DirectX Debuggers

Programming with a GPU can be like working with a black box. It's hard to see what's happening inside. If you get a blank screen, nonsense results, or a GPU driver crash, it can be hard to find what caused it. To make your life easier, consider using a debugger to investigate the GPU step-by-step to find bugs.

The following debuggers are recommended:

- RenderDoc (Windows, OpenGL/DirectX)
  - https://renderdoc.org/builds
- Visual Studio Graphics Diagnostics (Windows, DirectX, built-in Visual Studio 2015)


glowing-telegram - Microsoft Visual Studio
File
Edit
View
Project
Build
Debug
Team
Tools
Test
Analyze
Window
Help
Windows
Graphics
Show Diagnostic Tools Ctrl+Alt+F2
Start Debugging F5
Local Windows Debugger
Start Graphics Debugging Alt+F5
Start Graphics Debugging an Installed App Package...
DirectX Control Panel

- NVIDIA Nsight (Windows, OpenGL/DirectX, NVIDIA Only, Visual Studio plugin)
  - https://developer.nvidia.com/nvidia-nsight-visual-studio-edition
- AMD CodeXL (Windows, Linux, OpenGL/DirectX, AMD Only, Visual Studio plugin)
  - http://gpuopen.com/compute-product/codexl/
- Intel GPA (Windows/DirectX or Linux/OpenGL, Intel Only)
  - https://software.intel.com/en-us/gpa
- PIX for Windows (Windows/DirectX 12)
  - https://blogs.msdn.microsoft.com/pix/2017/01/17/introducing-pix-on-windows-beta/

There are precisely 0 (zero, zip, zilch) OpenGL debuggers on Mac, so I would seriously recommend working on this project on Windows (definite 1st choice) or Linux (2nd choice), so you can take advantage of these OpenGL/DirectX debuggers. Otherwise, you'll probably spend a long time looking at a black screen wondering what went wrong.