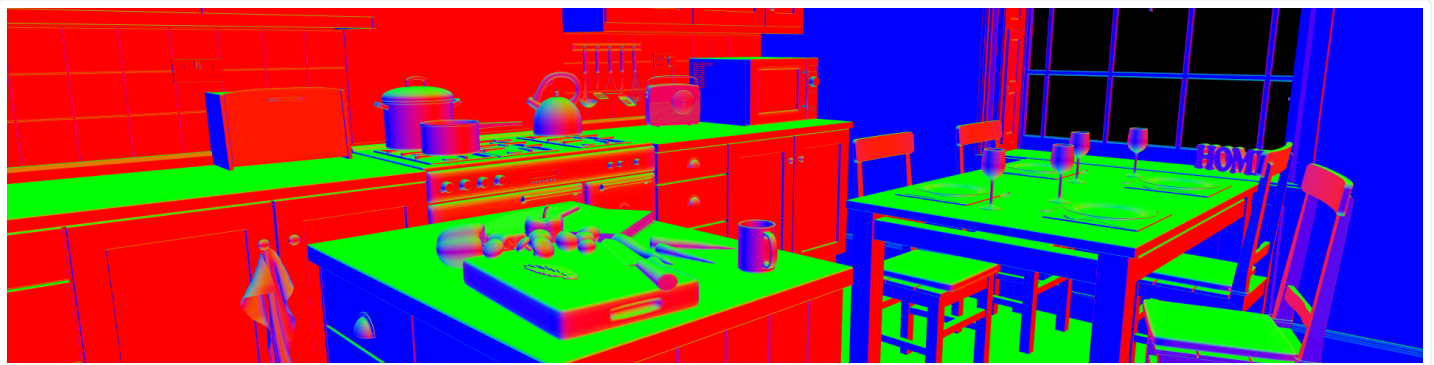


ECSE 446/546: Realistic/Advanced Image Synthesis Assignment 1: Setup & Basic Render

Due Friday, September 21st, 2018 at 11:59pm EST on [myCourses](#)
10% (ECSE 446 / COMP 598) or 5% (ECSE 546)



TinyRender is a minimalistic rendering engine written in C++11 and it runs on Linux, macOS, and Windows. The code we provide lays the foundations for the homework assignments in ECSE 446/546 and COMP 598 and is tailored to the course content. As such, it is both a real-time (online) and physically-based (offline) rendering framework.

While *TinyRender* provides a significant amount of scaffolding to simplify the development of full-fledged renderers, the code you will start with doesn't actually generate any impressive output: it loads a scene and either runs an OpenGL shader in the online case or saves an image as an OpenEXR file in the offline case — any significant rendering code is missing, and so the output consists of only black pixels. Over the course of the semester, your task will be to extend this system into a relatively complete renderer. The programming assignments will guide you incrementally through this process.

Contents

- 1 Offline rendering (50 pts)
 - 1.1 Rendering loop (35 pts)
 - 1.2 Surface normal integrator (15 pts)
- 2 Real-time rendering (50 pts)
 - 2.1 Rendering loop (30 pts)
 - 2.2 Surface normal shader (20 pts)

This first assignment handout is especially detailed/verbose, including compilation instructions for our *TinyRender* codebase as well as “getting started” guide. Despite its length, the tasks you will actually have to complete here are modest.

Core features

Our TinyRender codebase provides many features that would be tedious to implement from scratch, including:

- A [TOML](#)-based scene file parser and loader
- Basic linear algebra code for point/vector/normal/ray/bounding boxes
- A pseudorandom number generator
- Support for saving output as [OpenEXR](#) files
- An OpenGL library for real-time rendering ([SDL2](#))
- A [loader](#) for Wavefront OBJ 3D geometry files
- An optimized [bounding volume hierarchy builder](#)
- Ray-triangle intersection code

References

You may find the following general references useful, especially as the course progresses:

- Matt Pharr, Wenzel Jakob, Greg Humphreys, and Morgan Kaufmann (2016). [Physically Based Rendering, Second Edition: From Theory To Implementation](#).
- Tomas Akenine-Moller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, and Sebastien Hillaire (2018). [Real-Time Rendering, Fourth Edition](#).

Permissible reference sources

Feel free to consult additional references when completing the homework assignments, but remember to cite them in an attached read me file.

When asked to implement feature X , we request that you don't rely (or even search for) existing implementations in other renderers. You will likely not learn much by doing so, and we will obviously be on the lookout for this. Feel free to refer to PBRT, which gives a lot of implementation details, instead. If in doubt about this rule or any of the references you intend to use, ask the TA's.

Instructions

There will be five (5) programming assignments this semester for ECSE 446 / COMP 598 and an additional project for ECSE 546, each of which will help you progressively build a fully-functioning renderer. Each assignment has an offline (physically-based) and real-time (interactive) part. The assignments are to be completed individually, although you are open to consult your colleagues and the instructor/TA's. Each assignment has a strict deadline and submission instructions, both outlined in its handout. The codebase for each homework will be released along with the handout.

It is your responsibility to convince us that you have implemented the assignments correctly. You will sometimes have to reuse parts of your code across assignments. We strongly urge students to start working on the assignments as early as possible. Building your own advanced renderer is often very rewarding: use all the resources at your disposal, and don't forget to have fun during the process!

Building the codebase

TinyRender has very few dependencies that require manual installation. It relies on [GLEW](#), [SDL2](#), and [Boost](#) (Mac-only). Additionally, the [CMake](#) build system is required to compile and link TinyRender. We **strongly suggest** the use of an IDE such as [CLion](#) (Linux / macOS) or [Visual Studio 2017](#) (Windows 10) to do the assignments. In fact, we only provide building instructions for these two IDEs: use another IDE at your own risk. If you've never used these IDEs before, make sure to look up tutorials online on how to get setup. Most importantly, you will need to use their debugging tool: **do not neglect the utility of good debug tools**.

Be careful that all the directories in your path where you extracted the source code do not contain any *spaces* or *special characters*. If you do, the code won't build properly.

Linux / macOS

Installing dependencies on Linux

Installing the binary dependencies on GNU is straightforward, assuming you have a working [gcc compiler](#) which comes with most modern Linux distributions such as Ubuntu.

```
sudo apt-get update
sudo apt-get install libglew-dev libsdl2-dev cmake
```

Installing dependencies on macOS

Begin by installing a reasonably up-to-date version of [XCode](#) (≥ 8.0) along with the command-line tools. To check if the full XCode package is already installed, open a terminal and write:

```
xcode-select -p
```

If you see:

```
/Applications/Xcode.app/Contents/Developer
```

you have everything set up. Next, install [Homebrew](#):

```
/usr/bin/ruby -e
"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Then install the required packages:

```
brew install boost glew sdl2 cmake
```

You can verify that all packages were installed correctly by typing `brew list`.

Setting up TinyRender in CLion

Open CLion, click *Open...* and select the `CMakeLists.txt` file at the root of the TinyRender source code directory for the assignment. Select *Open as Project* when prompted and let CLion load the project. CMake will ensure that you have all the dependencies installed to build TinyRender. If no errors are found, you should see `[Finished]` at the bottom of the CMake console, preceded by a list of successful checks.

To build TinyRender, simply do *Run* → *Build*. This will compile and link your program in *Debug* mode. To create a *Release* build, navigate to *File* → *Settings (Preferences on macOS)*. In the tab *Build, Execution, Deployment*, *CMake*, click the + icon. You should now see *Debug* and *Release* under *Profiles* (see figure below).

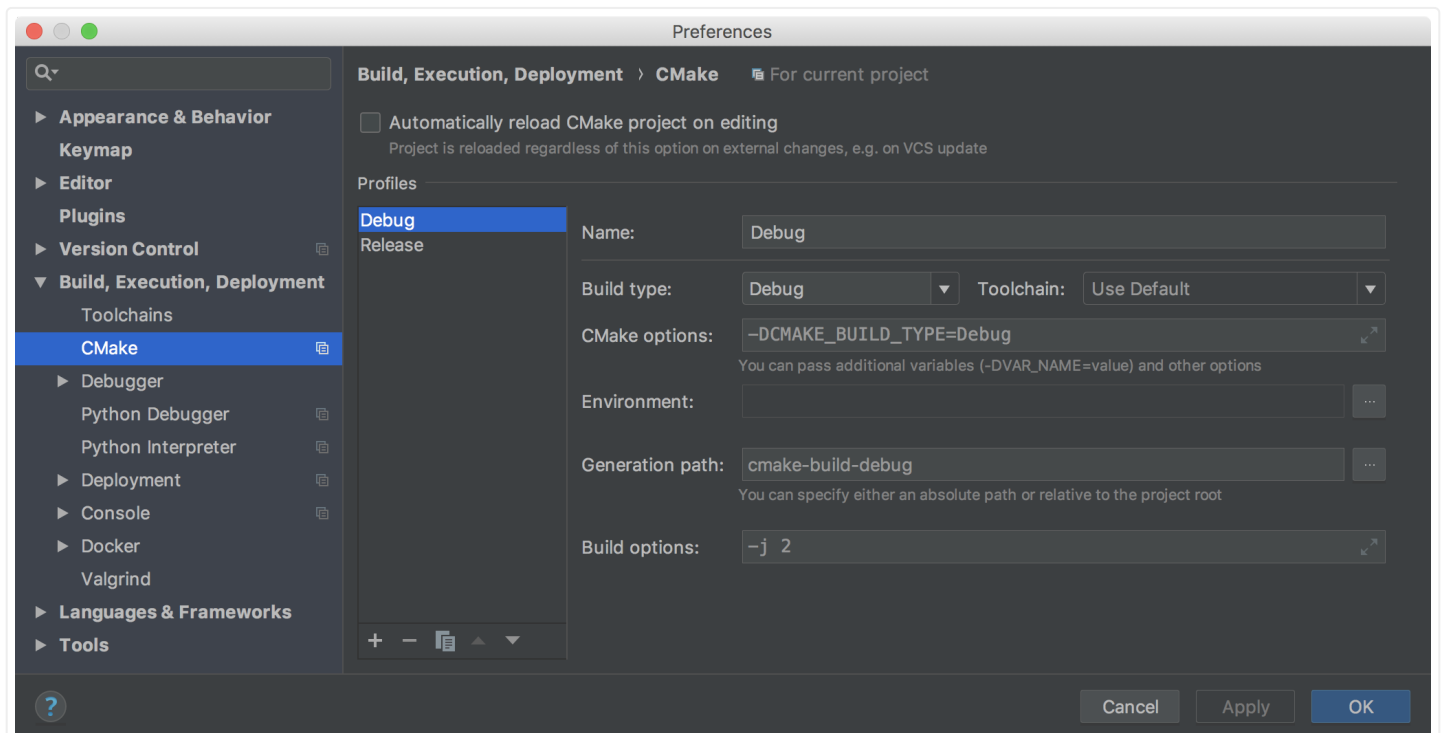



Figure 1: Adding the Release mode to a CMake build in CLion.

You can select your build type under the top-right corner dropdown in CLion and click the ↓ button to launch the build.

Windows

Installing / modifying Visual Studio

To compile and run TinyRender on Windows 10, first install a copy of [Visual Studio 2017](#) (Community edition will do). During the installation setup, you will be asked which workloads you wish to install. Select *Universal Windows Platform development* and *Desktop development in C++*. This should automatically add the latest Windows 10 SDK (10.0.17134.12) to the installation, which you can confirm on the right-hand side *Summary* panel. Once you're done, hit *Install*. Note that Visual Studio is a pretty heavy software and as such will require 10GB+ of disk space.

If you already have VS2017 installed, launch *Visual Studio Installer* from Start  and then click *Modify*. Make sure that you have the correct workloads installed along with the latest Windows 10 SDK. Click *Modify* in the bottom-right corner and let the installation finish.

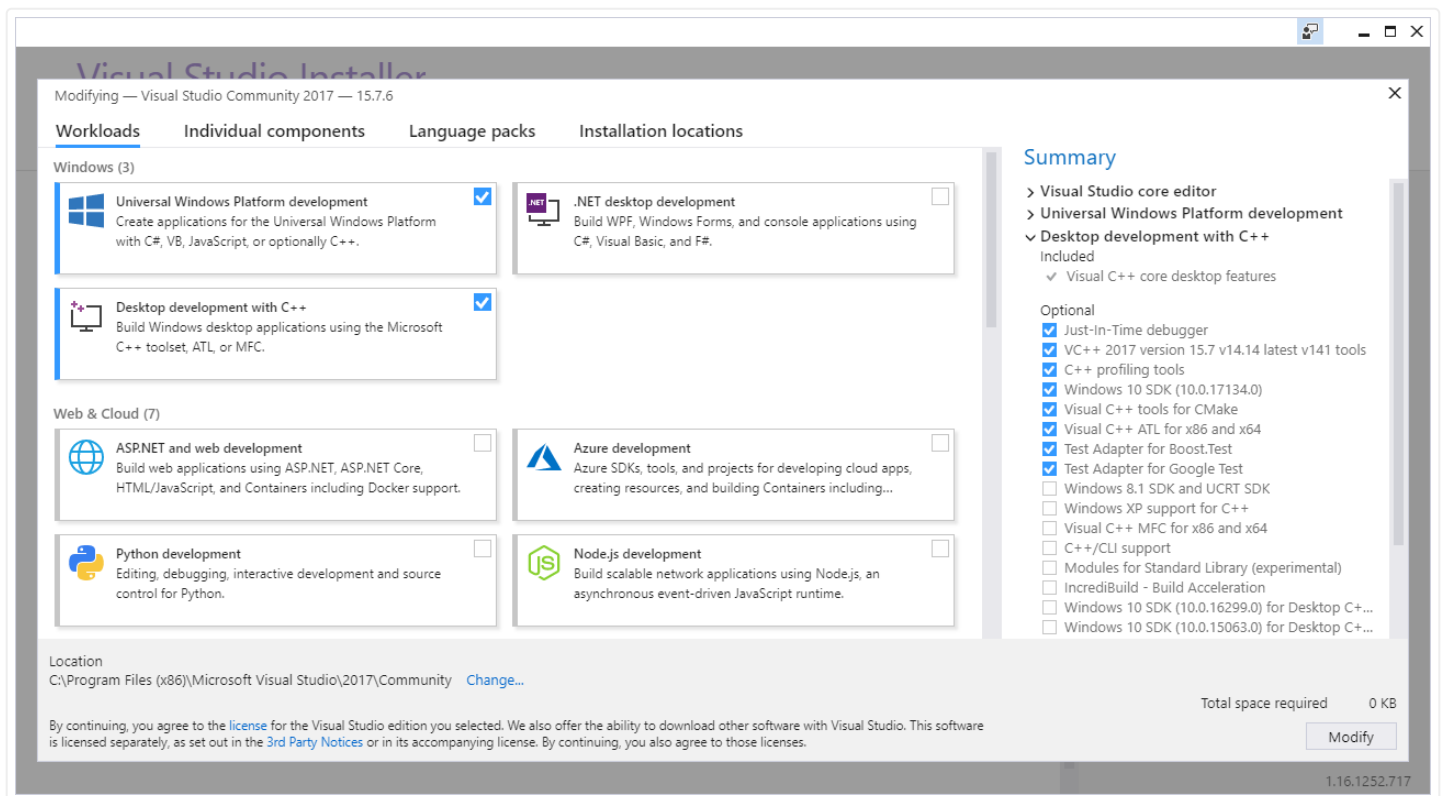


Figure 2: Installing / modifying Visual Studio.

Setting up TinyRender in Visual Studio

We provide a solution (.sln) file which you can directly open to create a TinyRender project in Visual Studio. To compile TinyRender, select your build mode at the top-left (*Release* or *Debug*), make sure *Solution*

Platforms is set to x86 and click ► *Local Windows Debugger* to launch the build.

A note on Debug / Release modes

If you are not familiar with these two types of builds, just know that *Release* is compiled with a set of standard optimization options whereas *Debug* has no optimizations and includes a debugging iterator. You should always render your final scenes in *Release* mode to accelerate the process. Developing in *Debug* mode allows you to make use of the debugging tools with the help of more verbose error messages. This ability comes at a significant speed reduction, however.

High-level overview of the codebase

The TinyRender repository consists of the codebase files and a few header-only dependency libraries that are briefly explained below.

Renderer structure

Directory / file	Description
externals/	External dependency libraries (see below)
src/	Directory containing the main C++ source code
data/	Example scenes and test datasets to validate your implementation
CMakeLists.txt	CMake build file that specifies how to compile and link TinyRender

Main source code

Students only have to deal with the files in the `src` directory (and `data` for the scenes). The table below briefly presents the content of each file and directory.

Directory / file	Description	Examples
bsdfs/	BRDFs directory	Diffuse, mirror, Phong
integrators/	Integrators directory	Ambient occlusion
shaders/	Interactive shaders directory	Diffuse with point light, SSAO
renderpasses/	Interactive render pass	Normal, direct
core/	Renderer core directory	
accel.h	Ray-scene intersection acceleration structure	
camera.h	WASD camera for real-time rendering	
core.h	C-like structures for all objects	Ray, camera, intersection, emitter
integrator.cpp	Integrator declaration file	
integrator.h	Integrator header file	
math.h	Rendering-specific mathematics	Sample warping functions
platform.h	Cross-platform type definitions and constants	ϵ , μ for numerical errors
renderer.cpp	Renderer abstraction and structure implementations	
renderer.h	Renderer header file	
renderpass.cpp	Real-time rendering pipeline declaration file	
renderpass.h	Real-time rendering pipeline header file	
utils.h	Image saving methods	
main.cpp	TinyRender's main program	

Item marked in [color](#) are the ones you will modify throughout the semester; you won't have to modify the rest.

You are not expected to understand every single line of code in these files but you should be familiar with the overall structure of the renderer. Additional template files such as BRDFs, integrators and shaders will be provided for every assignment in their respective directories. This will allow you to spend more time focusing on the algorithms and less on the actual software engineering of the system.

External libraries

Directory / file	Description
glm/	Mathematics library for graphics
bvh.h	Bounding volume hierarchy builder
cpptoml.h	TOML file parser
tinyobj.h	Wavefront OBJ mesh loader
tinyexr.h	High dynamic range image format library

Let's have a brief overview of the most important dependencies.

OpenGL Mathematics (GLM)

When developing any kind of graphics-related software, it's important to be familiar with core mathematics support libraries responsible for basic linear algebra types. TinyRender relies on [GLM](#) for this purpose, but we don't expect you to understand the inner workings of this library.

The main subset of types that you will most likely use are:

GLM type	TinyRender typedef	Object
glm::fvec2	v2f	2D float vector
glm::fvec3	v3f	3D float vector
glm::fvec4	v4f	4D float vector
glm::mat4	mat4f	4×4 float matrix

where the number in the type indicates the dimension and the `f` stands for `float`. Whenever you need to perform an operation on a vector or matrix, you will have to call `glm::` with the corresponding operator. Below are some operations that you may eventually need:

Operation	Function
Dot product between vectors u and v	<code>glm::dot(u,v)</code>
Cross product between vectors u and v	<code>glm::cross(u,v)</code>
Component-wise absolute value of vector v	<code>glm::abs(v)</code>
Normalization of vector v	<code>glm::normalize(v)</code>
Euclidean norm of vector v	<code>glm::l2Norm(v)</code> or <code>glm::length2(v)</code>
Distance between points p and q	<code>glm::distance(p,q)</code>
Reflection of vector v on surface with normal n	<code>glm::reflect(v,n)</code>

For more information, you can consult [GLM's official documentation](#) or set up autocomplete in your IDE.

Other mathematical operators can be called via the [Standard C++ Library](#) (`std`). For instance, absolute value $|x|$ can be obtained by calling `std::abs(x)` and exponentiation x^n can be done with `std::pow(x,n)`.

OpenEXR

[OpenEXR](#) is a standardized file format for storing high dynamic range images. It was originally developed by [Industrial Light and Magic](#) and is now widely used in the movie industry and for rendering. The directory `externals/tinyexr` contains a header-only implementation of this standard. You will not be using this library directly.

A word of caution: various tools for visualizing OpenEXR images exist, but not all really do what one would expect. The softwares listed below work correctly, but Preview on macOS for instance tonemaps these files in an awkward and unclear way.

- [tev](#) by Thomas Moller
- [HDRView](#) by Wojciech Jarosz
- [HDRITools](#) by Edgar Velazquez-Armendariz
- [Adobe Photoshop](#) with [OpenEXR plugin](#)

Simple DirectMedia Layer (SDL2)

[SDL2](#) is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. In TinyRender, SDL2 is used for the real-time rendering parts of the assignments. You will not have to directly interact with this framework: most of the OpenGL code will remain hidden from you. However, you will have to write the online rendering loop together with vertex and fragment shaders for most assignments, including this first assignment.

TinyRender pipeline

At a very high level, your renderer takes as input a scene description file and parses all the information necessary to render the scene. This includes the position of the mesh triangles, camera settings, type of rendering techniques used, and so on. Once parsed, this data is passed to the renderer which evaluates either an integrator or a shader to compute the resulting image. At the end of this rendering loop, an image is output and saved to files.

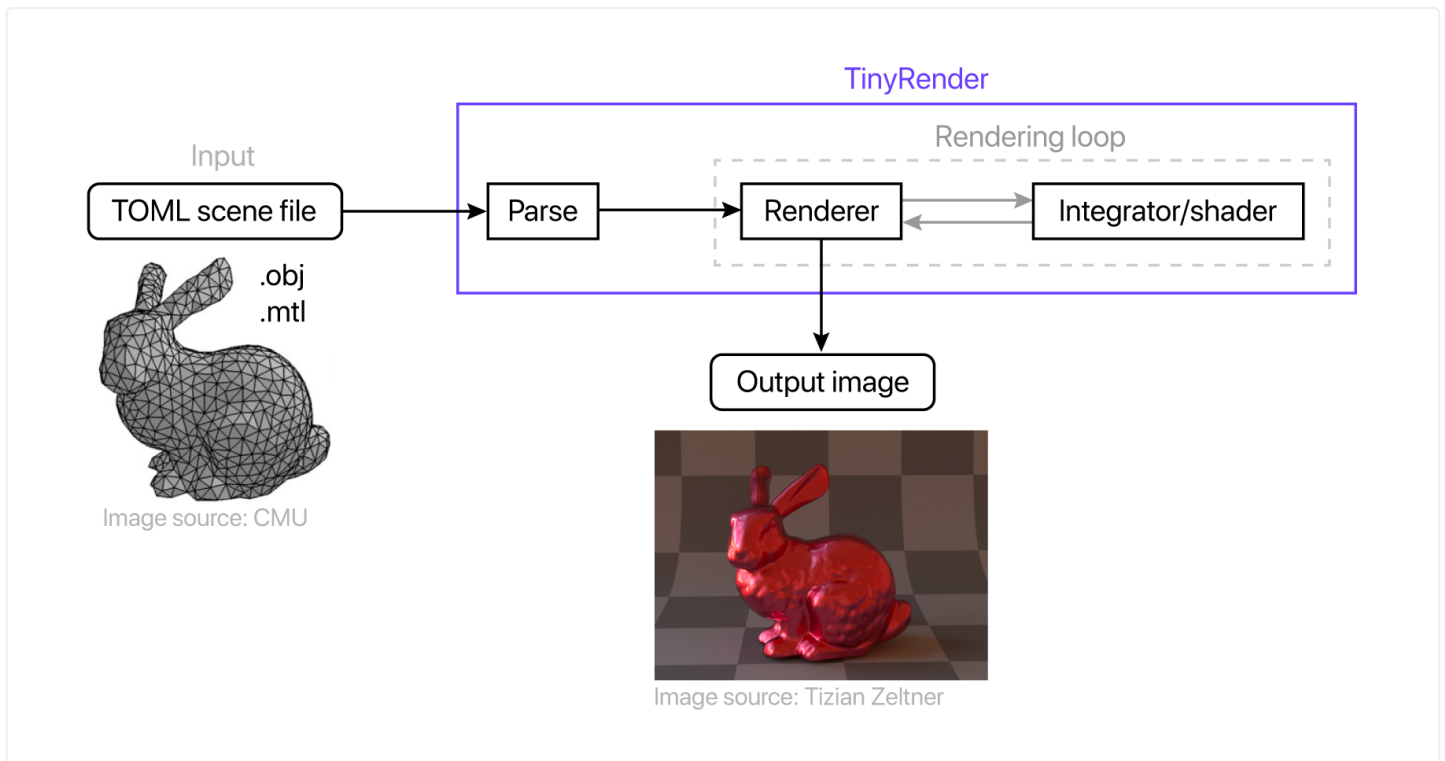


Figure 3: *TinyRender rendering pipeline.*

Scene file format and parsing

Each assignment will require you to render multiple scenes to demonstrate a proper implementation of the tasks. These scenes are given with the following tree structure:

```

data
  scene_1
    mesh
      mesh_1.obj          // Meshes
      mesh_1.mtl          // Materials (required)
    tinyrender
      task_1_offline.toml  // Offline scene
      task_2_realtime.toml // Real-time scene
  scene_2
    ...
  config.json             // Configuration file for submission (see script read
me)

```

You are free to inspect the mesh in a 3D modeling software suite, such as [Blender](#) or [Autodesk Maya](#). Both softwares are available for free but you will need to use your McGill email address if you wish to use Maya.

The .mtl file is a [material library file](#). This file format was originally designed for real-time graphics but tinyobj.h parses this information and assigns a material to each mesh in the scene. While old and inelegant, this format is convenient for many reasons. In particular, it allows TinyRender to use a single material file for both the offline and real-time subsystems. These files are necessary and very rigid syntax-wise, hence you should *not* alter them and their corresponding OBJ parent file unless you know what you're doing.

The TOML scene description file contains all the necessary information to render a scene. Below is an example of an offline task file:

```
# Offline rendering scene

[input]
objfile = "../mesh/mesh.obj"

[camera]
eye = [0.0, 1.0, -4.0]
at = [0.0, 0.0, 0.0]
up = [0.0, 1.0, 0.0]
fov = 30.0

[film]
width = 1920
height = 1080

[renderer]
realtime = false
type = "ao"
spp = 16
# Parameter = value (optional, depends on your integrator)
```

This file is given as the main input parameter to your renderer. The codebase already parses this information for you and stores the information in scene.config. It starts by reading the OBJ meshes and their corresponding material file (which needs to have the same name). TinyRender then creates a pinhole camera at point eye = (0, 1, -4) looking at the position at (in world coordinates). The up-vector of the camera is the y-axis, in this example. The field of view (FoV) is set to 30° and the image plane is 1920×1080 pixels (full HD). Finally, the integrator used to compute pixel intensities is ambient occlusion (ao) and TinyRender is being told to sample 16 rays through each pixel (SPP, sample per pixel, see [Part 1.1](#).)

A real-time rendering TOML scene file has a similar structure, except the renderer uses shaders and a different render loop (see [Part 1.2](#)). The only option to provide for real-time scenes is the shader type. In the example below, the renderer will use screen space ambient occlusion (SSAO) to render an interactive scene, something you will have to implement in Assignment 3.

```
# Real-time rendering scene
# ...

[renderer]
realtime = true
type = "ssao"
# Parameter = value (optional, depends on your shaders)
```

Technically, you won't have to change anything in the TOML scene files. Both offline and real-time scenes to render will be provided for you and marked appropriately. If you wish to play around by moving the camera or increasing the number of samples, you can substitute custom values in the corresponding fields.

Rendering a scene

CLion

To render a scene, you need to provide a TOML scene file as a program argument to TinyRender. In CLion, this is done by clicking on *Run* → *Edit Configurations...* and passing the scene file path in *Program arguments*.

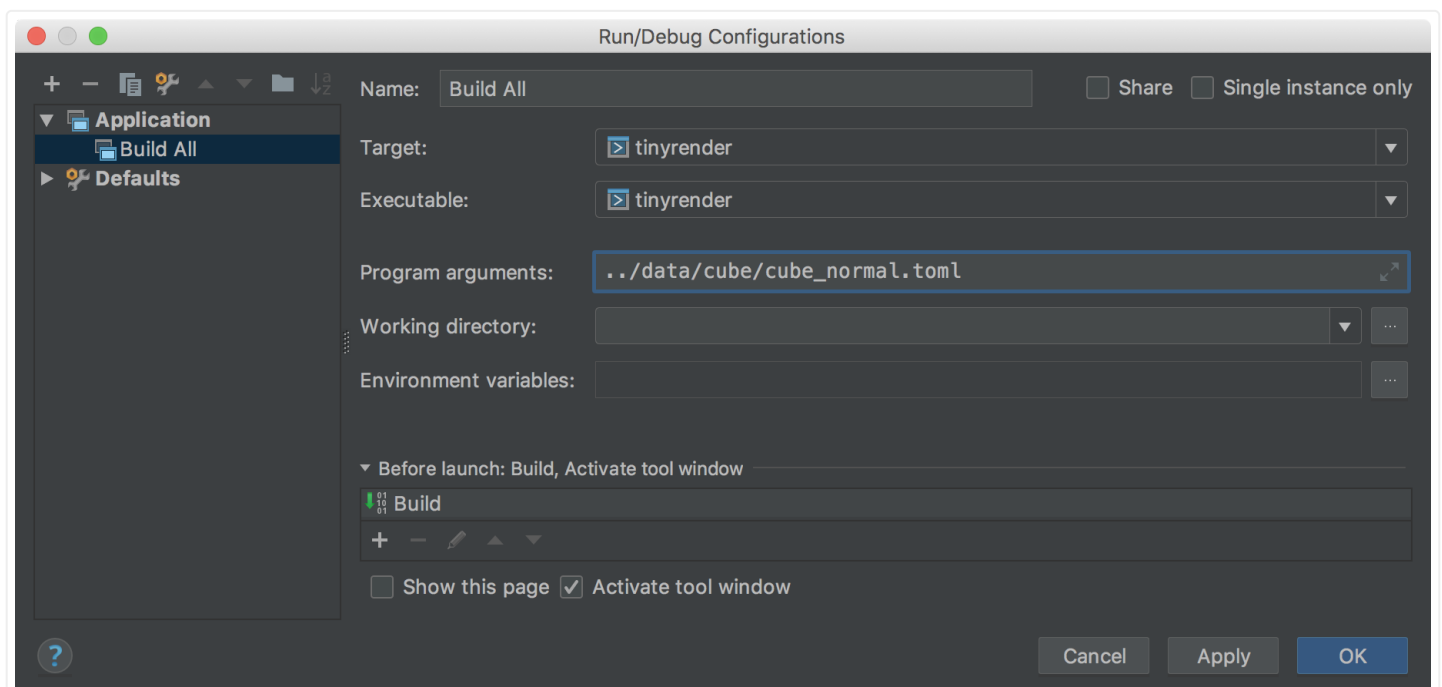


Figure 4: Passing a TOML scene file as an argument to TinyRender in CLion.

Visual Studio

In Visual Studio, go to *Project* → *Properties* (Alt + Enter). Navigate to *Configuration* → *Debugging* and enter the scene path as *Command Arguments*. The variable `$(SolutionDir)` is the path where your solution file is. You can ensure that the scene argument is shared across build modes by selecting *All Configurations* in the top-left *Configuration* dropdown.

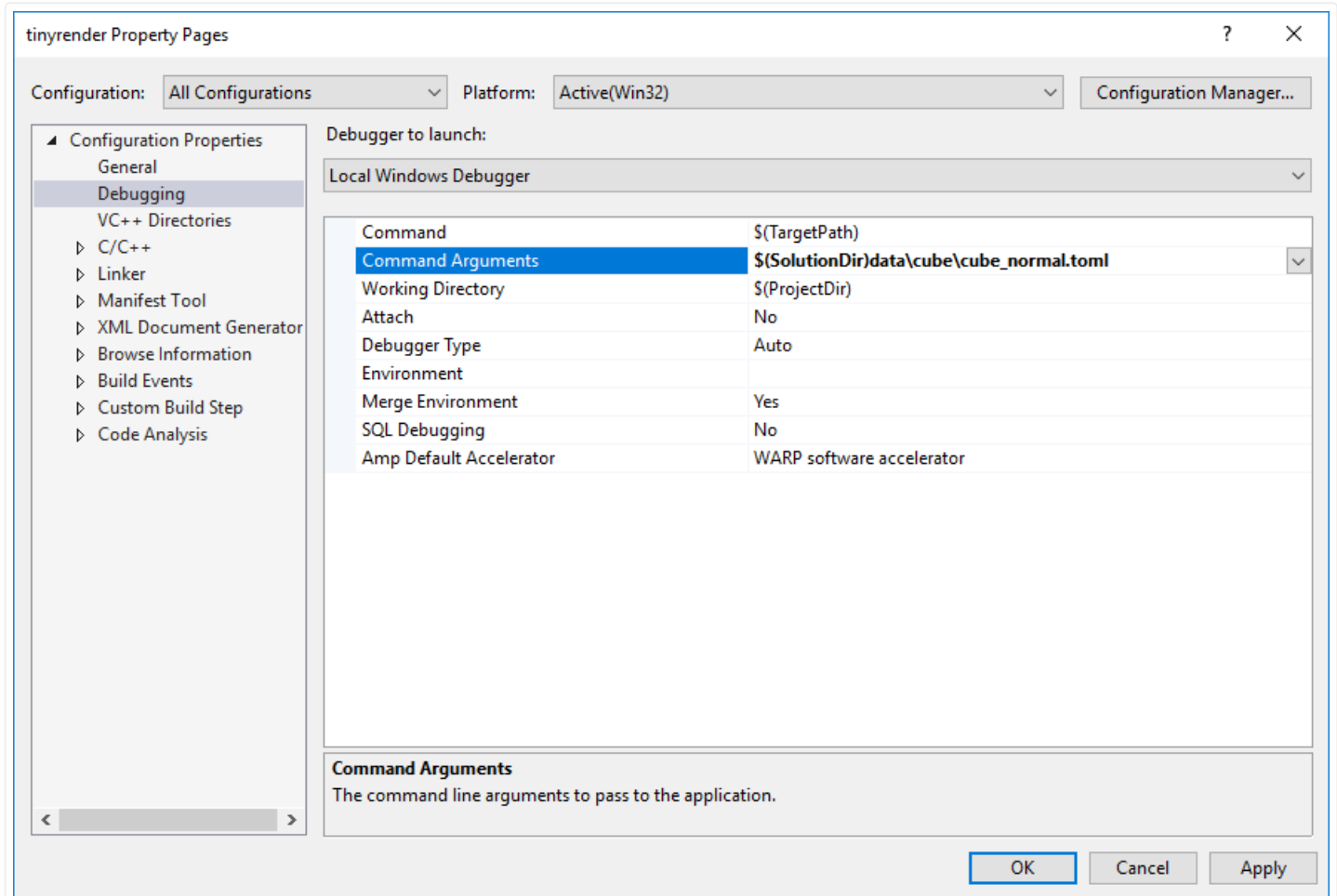


Figure 5: Passing a TOML scene file as an argument to TinyRender in Visual Studio.

1 Offline rendering (50 pts)

1.1 Rendering loop (35 pts)

Your first task is to implement the offline rendering loop. The function `Renderer::render()` in `src/renderer.cpp` is currently empty: your goal is to implement it by looping over all pixels on the image plane and computing their color. At a high-level, this is done as follows:

1. Calculate the camera perspective, the camera-to-world transformation matrix and the aspect ratio.
2. Loop over all pixels on the image plane.

3. Generate a ray through each pixel and splat their contribution onto the image plane.

The last step can be broken down into five substeps:

1. Retrieve the *center* of pixel (x, y) .
2. Build a ray with origin at the camera's center with direction passing through the pixel center.
3. Apply a transformation to map this ray to world coordinates (by taking FoV and aspect ratio into account).
4. Call the integrator on your newly constructed ray, passing in the scene's random number sampler.
5. Get the radiance contribution for the pixel and output to the image buffer.

Each step is detailed in the tutorial slides for A1 available on myCourses; make sure to read these carefully.

If done properly, you should now obtain a solid green .exr output when rendering any TOML scene in the data directory along with the following (similar) message in the console:

```
Found 1 shape
Mesh 0: cube [12 primitives | Diffuse]
BVH built in 1.3e-05s
Saved EXR image to ../data/cube/cube_normal.exr

Process finished with exit code 0
```

Note, of course, that the converse is **not** true: seeing a green image does not imply that your implementation is correct. The next part of this assignment will validate your code by rendering an actual scene.

1.2 Surface normal integrator (15 pts)

In TinyRender, offline rendering algorithms are referred to as integrators because they generally solve a numerical integration problem. The second offline task of this assignment is to modify your normal integrator to display the surface normals of a mesh. To do so, you will have to modify `NormalIntegrator::render()`, which currently returns the color green regardless of the ray intersection:

Normal integrator (`integrators/normal.h`)

```

struct NormalIntegrator : Integrator {
    explicit NormalIntegrator(const Scene &scene) : Integrator(scene) { }

    v3f render(const Ray &ray, Sampler &sampler) const override {
        // TODO: Implement this
        return v3f(0.f, 1.f, 0.f);
    }
};

```

This function is called earlier by your rendering loop, after TinyRender generates eye rays through each pixel (Part 1.1). Each one of these ray calls the `render()` function of the integrator instantiated from the scene file. The ray parameter passed to `render()` is a camera ray through a given pixel; in your `NormalIntegrator`, you will intersect the scene with this ray to find the closest surface to the ray origin, along its direction. In the case of camera rays (i.e., the rays passed to `render()`), this intersection corresponds to the closest visible surface from the viewer. If an intersection is found, simply return the component-wise *absolute value* of the shading normal at the intersection point, as a color. Have a look at the `AcceleratorBVH` structure in `src/accel.h` to see how trace rays against the scene geometry.

To run your renderer and (hopefully) see the result of your work, invoke `tinyrender` on the files `data/cube/tinyrender/cube_offline.toml` and `data/bunny/tinyrender/bunny_offline.toml` to get the images below.

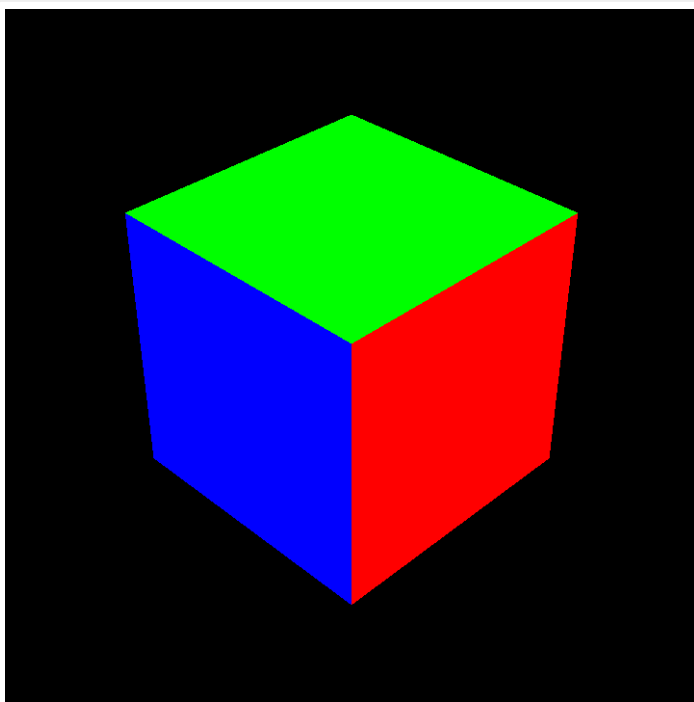


Figure 6: Cube scene with no background.

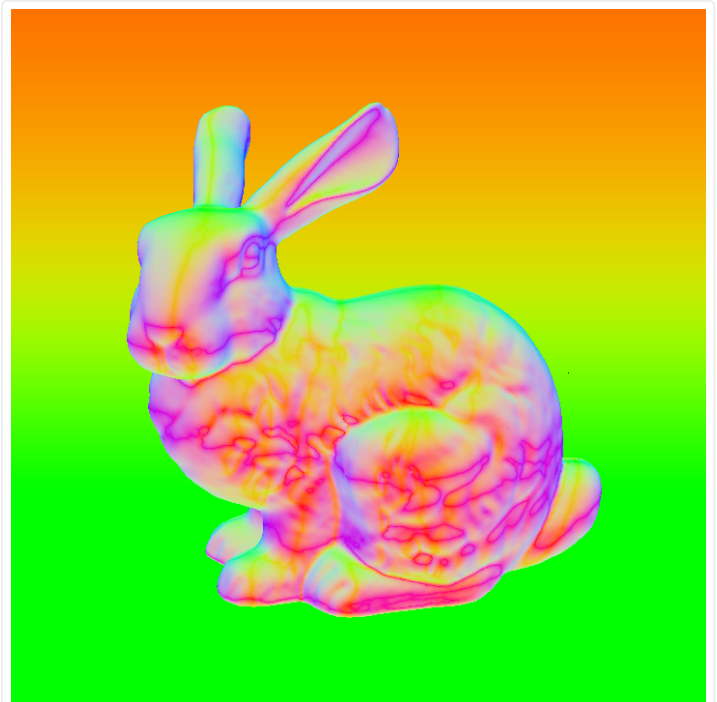


Figure 7: Stanford bunny scene with a *studio backdrop* as background.

2 Real-time rendering (50 pts)

2.1 Rendering loop (30 pts)

Your second task is to implement an online rendering loop in the function `Renderer::render()` within `src/renderer.cpp`. You must render the scene interactively, meaning fast enough that movement is fluid. In practice we render at least 60 FPS (frames per second).

To do this, create an infinite loop in which you:

1. Detect and handle the quit event to close the window when you click on the X button (see [SDL documentation](#)).
2. Call the render function using `renderpass->render()`.
3. Output the rendered image into the GUI window using `SDL_GL_SwapWindow(renderpass->window)`.

The file `src/renderpasses/normal.h` implements the different stages of the render pass for your shaders. In particular, it compiles the shaders and links them, and creates GL uniforms for the camera settings. One part is missing however: the loop that actually draws the objects in the scene:

```
for (auto &object : objects) {  
    // TODO: Implement this  
}
```

Implement this missing part along with the real-time rendering loop before moving to the second part.

2.2 Surface normal shader (20 pts)

In TinyRender, real-time rendering algorithms are implemented as shaders written in [GLSL](#). GLSL is a programming language very similar to C.

A scene is composed of many different objects. One VBO & VAO is assigned per object. The VBO & VAO are filled for you and you don't have to worry about that in this course.

A VBO ([Vertex Buffer Object](#)) is a buffer in OpenGL (using GPU's memory) to store vertex attributes like the position, color of every vertex of each triangle of each object in your scene. The position of each vertex is in *object space*. A VAO ([Vertex Array Object](#)) can be seen as an extension of a VBO, it defines how the data is stored in the VBO. One VBO is assigned to each VAO.

Binding a VBO is the way to pass vertex attributes to a vertex shader. Those will be then available in the corresponding variable in the shader.


```
layout(location = 0) in vec3 position;  
layout(location = 1) in vec3 normal;
```

You can get the ID assigned to the VBO & VAO of each object in the GLObject.

As with the offline version, complete the vertex shader `shaders/normal.vs` and the fragment shader `shaders/normal.fs` to display the surface normals of a mesh. Most of the heavy lifting necessary to run shaders should be done at this point, you only have to worry about the actual rendering algorithm to shade every pixel.

Vertex shader (`shaders/normal.vs`)

```
uniform mat4 model;  
uniform mat4 view;  
uniform mat4 projection;  
uniform mat4 normalMat;  
  
layout(location = 0) in vec3 position;  
layout(location = 1) in vec3 normal;  
out vec3 vNormal;  
  
void main() {  
    // TODO: Implement this  
}
```

Fragment shader (`shaders/normal.fs`)

```
in vec3 vNormal;  
out vec3 color;  
  
void main() {  
    // TODO: Implement this  
}
```

To run your renderer and (hopefully) see the result of your work, invoke `tinyrender` on the files `data/cube/tinyrender/cube_realtime.toml` and `data/bunny/tinyrender/bunny_realtime.toml`. The resulting images should be exactly the same as the one generated by the offline renderer. Note that if your shader has errors, the renderer won't start and the errors will appear in the console.

OpenGL documentation is available [here](http://www.khronos.org/opengl/).

Bonus: Uniform sampling of pixels (20 pts)

Most physically-based renderers normally sample multiple rays through each pixel and average their contribution, which performs antialiasing. This means that rays are not constrained to pass through pixel centers anymore: they can go through any point in the square pixel. Modify your working rendering loop to support multiple samples per pixel:

1. For all samples, do:
2. Uniformly sample a position on the pixel area.
3. Generate eye rays that pass through these new points and output the *average* radiance contribution.

The function `sampler.next()` invokes the pseudorandom number generator and returns a random value in the interval (0, 1). Further note that it is important that the overall contribution is an average: details will be covered when Monte Carlo methods are introduced later in the course. To test your new implementation, render the cube scene (after having done the following task) with a value of 16 spp and compare your result with the provided ground truth image.

What to submit

Submission script

Download the [assignment submission script](#) and read the `README.md` file to familiarize yourself with the program. Throughout the course, you will reuse this script to submit all your rendered images. This script allows the graders to directly compare your offline output with theirs. The output of the script is a single file `a1_id.html` that is self-contained: all rendered images get inlined inside the HTML in base64.

Once again, if you have troubles running the script, please email the TA's. **Do not wait until the deadline to test the submission script: test beforehand, and often!**

Files to hand in

Render all the scenes in the data directory. When you're done, submit your submission script output file `a1_id.html` and all your code in a `.zip` or `.tar` archive file. Include your raw `.exr` files in separated folders (see structure below).

```
a1_first_last.zip
  a1_id.html
  src/
  offline/
    sphere_normal_offline.exr
    dragon_normal_offline.exr
  realtime/
    sphere_normal_realtime.exr
    dragon_normal_realtime.exr
```

Make sure your code compiles and runs for both online and offline parts before submitting! *You will obtain a score of zero if your submission does not follow this exact structure.* You can use the [tree](#) command to verify this structure.

formatted by [Markdeep 1.04](#) 