Project 2: Creating a World

CS 351-1, Winter 2025

Expectations

Projects in CS 351-1 are the core of the course, and you should expect to spend substantial time on each project. Projects are graded based on three component: demo, writeup, and code submission. See the Project 1 rubric for grading details. While the project rubric is strict, there are many (infinite) ways to achieve the requirements in the rubric. Creativity and effort will be rewarded with more generous grading (and will make the projects hopefully more fun), so feel free to experiment! You are always welcome to ask if you are not sure if your project design will match the rubric criterion.

Academic Integrity

As with assignments, you may freely use any source (so long as you cite any code or algorithms you directly use) and discuss problems with classmates, but you *may not* explicitly share code or exact solutions with classmates. **Note**: I will be explicitly checking for plagiarism in projects much more carefully than assignments (using MOSS and related tools), so please be careful to cite your sources. I take academic integrity violations quite seriously, and I would really rather not have to make your (and my) life miserable by having to escalate a plagiarism or cheating report.

Overview

The focus of this project is to extend your project 1 animation work by constructing a proper world. You will be adding a freely moving camera (that can rotate and "fly"), along with some randomized terrain. We will be providing a library that generates a terrain mesh using Perlin noise, which you should expect to use to help construct your world. You will need to use properties of this terrain mesh to properly place your animation in the world and navigate the scene with a camera.

By completing this project, you will have built the following:

- **A Freely Movable Camera**, which can rotate up, down, left, and right, and is able to move forward/backward relative to its facing direction.
- **An animated 3D scene**, which can be repositioned to match your terrain.
- **Generated Terrain**, which is used to create a "ground" that your animation stands on.
- **A Drivable Camera**, an option for the camera to "drive" along the terrain you generated.

As always, you may feel free to implement more than this to build an even cooler scene, but the rubric will describe the minimum requirement you should follow. There are a bunch of requirements in this rubric that might be confusing on first read, so we highly recommend asking (on Ed or in office hours) if you're not sure on a requirement or would like clarification.

Submission

You will be expected to submit **one zip folder** containing at least the following files:

- project2.pdf: a writeup description of your project1

- project2.html: the main HTML file to run your project (you can rename this file)
- project2.js: the main JS file to run your project (you can also rename this file)

Additionally, you may submit any number of additional HTML and JavaScript files. All additional files submitted must be (briefly) described in your `project2.pdf` writeup. If your code relies on the provided `lib` utility folder, please include that folder in your submission.

Making sure that your project is easy to run will help the grading team to grade your project quickly and fairly. If your project requires a different setup than opening your code in the usual local server, please let us know in the writeup summary.

Materials

We will provide the following materials:

- **project2.pdf**: this writeup
- **project2_rubric.pdf**: an exact project grading rubric
- **terrain.js**: a JavaScript implementation of perlin noise that builds terrain based on provided parameters

You may, however, use any code provided as part of in-class demos, including your own previous assignments, so long as you cite where the code originated. Note that we do not provide starter code for this assignment; by default, expect to continue from where you left off with your project 1 code OR from the project 1 starter code.

Project Guide

The following section is going to be a rough guide for how to complete the project. For specific grading details and descriptions, please refer to the rubric.

Part 1: Freely Movable Camera

Your first priority is to upgrade your perspective camera from project 1 with the ability to "fly" and rotate freely (rather than being restricted to a single axis). Note that you no longer need to maintain an orthographic camera for this project (though you are welcome to have an orthographic mode if it looks cool with your setup).

Specifically, your updated camera should be able to rotate up, down, left, and right *relative* to its current direction. You must specifically be able to rotate the camera fully "upside down" using up-and-down rotation, and be able to rotate the camera to face "backwards" of its initial orientation using left-and-right rotation. We strongly recommend using keyboard inputs or mouse movement to rotate your camera (arrow keys and WASD are popular choices, though you may want to use WASD for moving the camera relative to its rotation).

Additionally, your camera must be able to move forward and backwards *relative* to its currently facing direction. This means that a camera which you rotated to face "up" (relative to your scene) would move up when moving forward and down when moving backwards. We similarly recommend using keyboard inputs for camera movement. You may also find it helpful to include keys to move the camera left and right or up and down for debugging, but these features are optional to include in your project.

Hint: Avoid gimbal lock with quaternions. Remember that you can always convert a quaternion to a rotation matrix when you actually need to use the rotation!

Hint: Calculate a "forward vector" as needed rather than storing it directly. It is a lot easier to manage rotations and the direction you're facing if all you need to store is a single quaternion! Note that you can calculate the correct "forward direction" for a quaternion by converting that quaternion to a rotation matrix and multiplying that matrix by your fixed forward vector (`[0, 0, -1]` is a reasonable right-handed forward vector if your up direction is `[0, 1, 0]`, for reference)

Hint: Use slerp to rotate the camera (see lecture 14). By using slerp, rotation will feel more natural and be easier to debug, though you might need to think carefully about how you can slerp in each direction!

This first part will be a large bulk of the project 2 work for many students, so if you're struggling to get this working, don't panic! Please make time to ask or get help if you're not sure on the requirements or how to get started!

Part 2: An Animated 3D Scene

You already have most of this part finished from project 1! You will once again need 3 models including some form of animation, which can either be automatic or involve user interation. Note that you do not need to have an assembly (though you may include one) and you also do not need to maintain per-vertex colors (though you might find them helpful to keep around for debugging).

The only addition from project 1 is a requirement that your animation interact with your generated terrain in some way (see part 3 for details on what is meant by generated terrain). An important requirement here is that your animation should react in some way to the size/shape of your terrain, such as having a plane fly above your world or a character walking over a generated hill.

We recommend getting this interaction setup before you construct the actual terrain. Specifically, we suggest placing a simple cube with some random properties (say size and position) and having a function to place your animation based on this random property. For example, if your goal is to have a boat sailing around a generated island, then generate cubes with varying sizes and test that your boat is rotating at a wide enough angle to avoid hitting the corners of the cube.

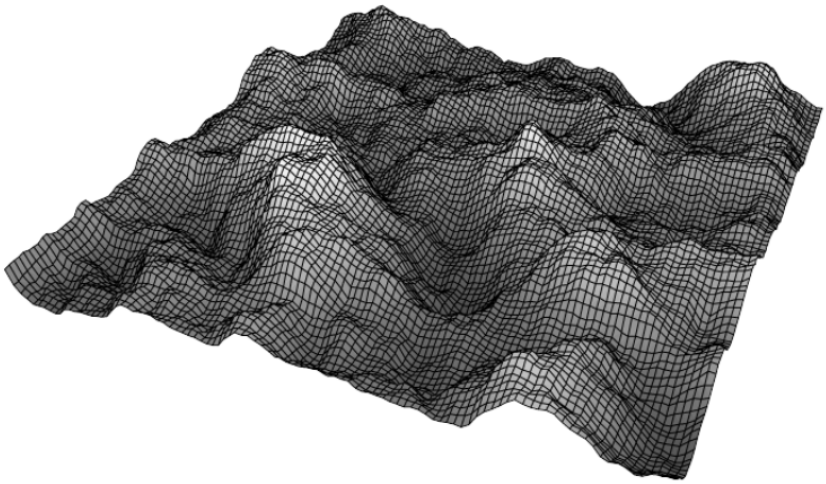
Note that you will have substantial control of what can be randomized with your terrain, so you don't need to account for edge cases where we "break" these constraints. For example, if you intend to have a plane flying above your ground, you could choose to randomize only the "roughness" of the terrain to avoid having to change the height of the plane at all (so long as the highest peak is below the plane!)

Part 3: Randomized Terrain

Now for the exciting part of this project, generating a world! For this part, you will need to generate a mesh using some sort of randomization that resembles "real ground".

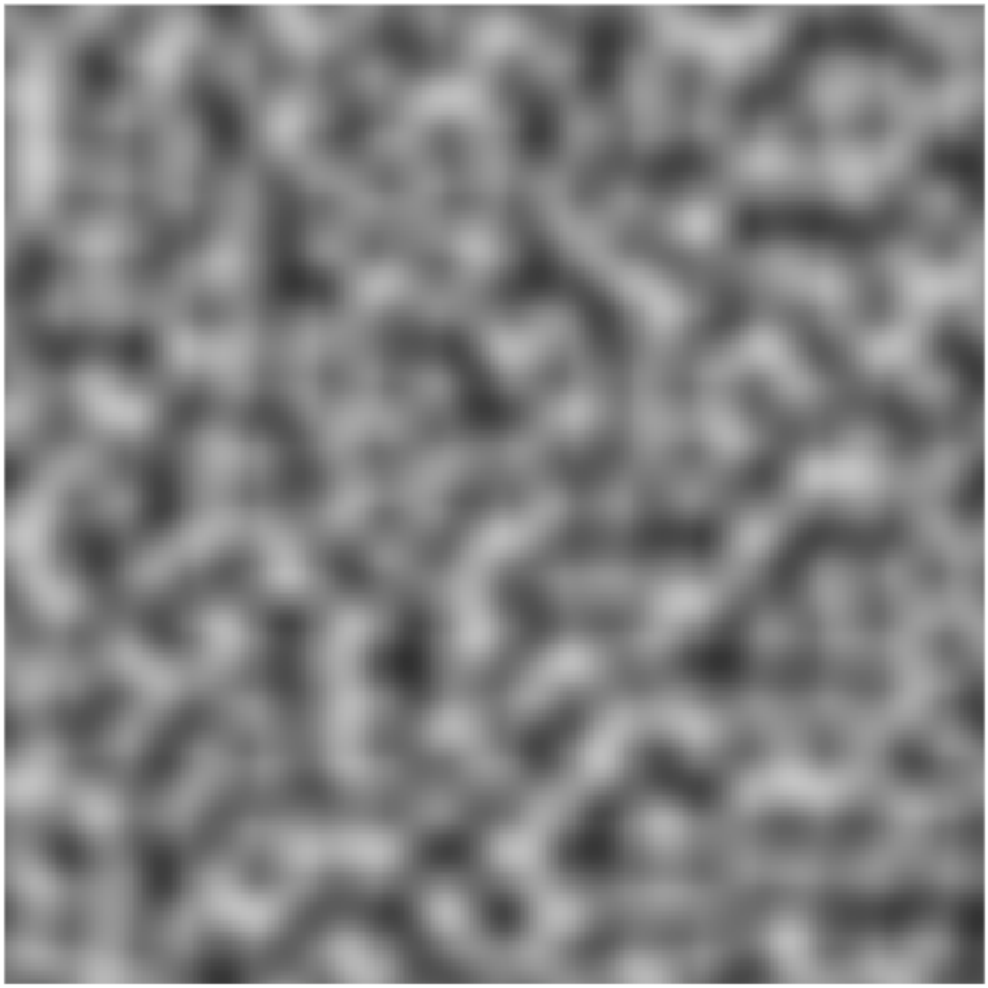
Background

We'll be providing a library for a simple form of terrain generation, which involves using a special type of noise known as Perlin noise, to generate a hilly terrain:



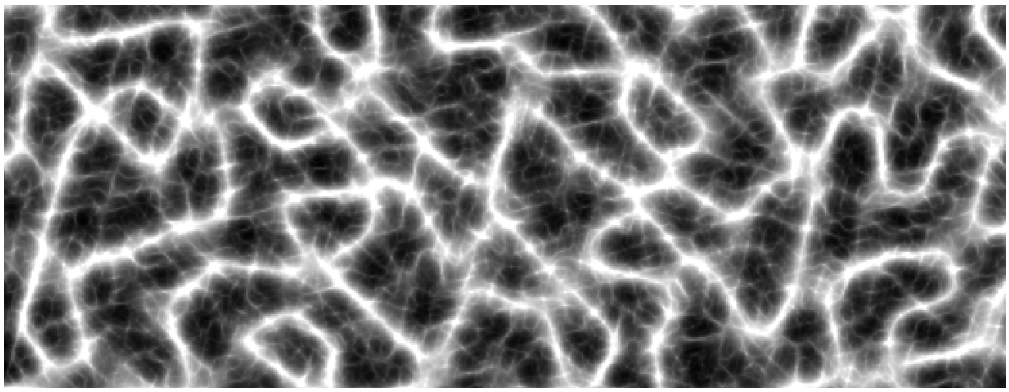
This type of terrain generation is surprisingly straightforward to implement, but is fundamental to games such as Minecraft and Terraria. Core to this terrain generation is "noise", which describes random values that vary over a collection of points (such as a 2D grid describing the elevation of ground terrain).

Perlin noise is a type of “smooth” noise, which you can think of as smooth random values. In its simplest form, Perlin noise looks like this, where each pixel corresponds to a grayscale RGB color such that white is 1 and black is 0:

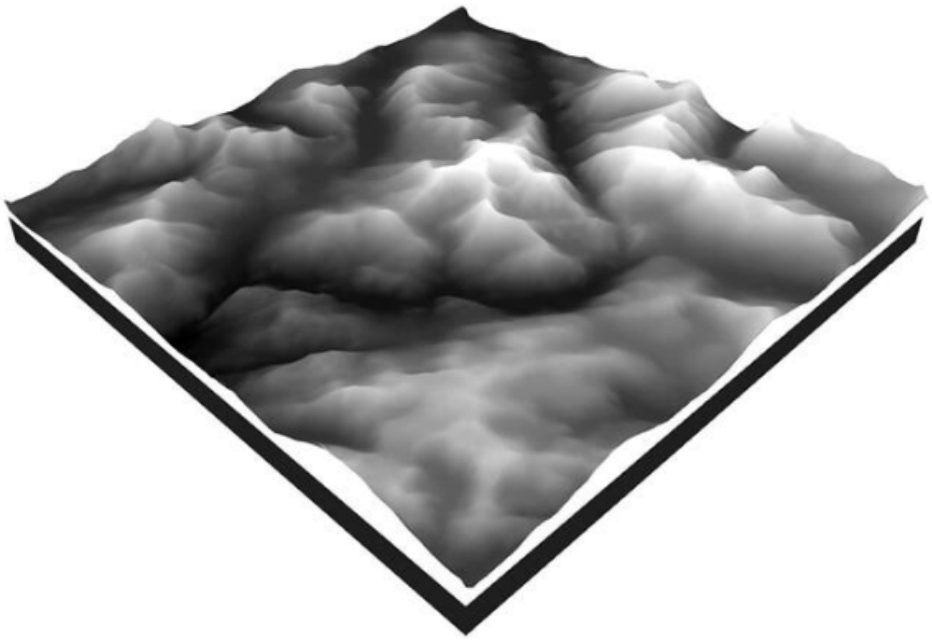


This image looks blurry precisely because the values blend "smoothly" along each line. While this looks unnatural in a basic image, this smoothness is exactly what we want for constructing ground that resembles rolling hills.

Additionally, Perlin noise has the extremely useful property that we can adjust this smoothness to our liking with some code manipulation. This great article in the book of shaders goes into some good detail on how we can manipulate the baseline noise generation algorithm provided. For example, if we select parameters that increase the "sharpness" of points, we can produce terrain resembling ridges and canyons:



Now that we have a generated image with some amount of smoothness, we need to build an actual mesh. Our terrain generator will follow the idea that we can interpret perlin noise as a heightmap, which is an image where darker values (near 0) correspond to lower areas of elevation, and lighter values (near 1) correspond to higher areas of elevation. If we were to map a generated noise map to a heightmap and display it, we might see something like this:



Connecting our image to this heightmap is, in some sense, straightforward. Given the `x` and `y` coordinates of each point of the mesh and the color `c` of each pixel, we can construct a triangle for each point triple with vertex coordinates [`x`, `y`, `c`], and that's it. A slightly trickier, but important detail, is our construction of our mesh normals (which we will need to place objects on the terrain). We simplify this problem by considering

only triangle-level normals, which can be calculated directly using the surface normal algorithm, and using correct winding order to have each triangle face "up" on the terrain.

Your task

With all of this theory out of the way, your task is somewhat straightforward. You should start by using the provided terrain generation code (modified or written on your own if you want more control) to generate a mesh and display it in your scene. This may take some debugging to get the mesh loaded onto the GPU correctly.

Note that you may need to include per-vertex colors on your terrain to have it render properly (or at all). You should consider using the elevation (or z value) of each vertex to construct per-vertex colors automatically -- for example, having green valleys and white mountain peaks is as simple as constructing a list of colors where any Z value below some threshold is green and any Z value above that threshold is white.

Once you have successfully loaded a single terrain mesh, we suggest playing around with arguments to the terrain generator to get a sense of what you might want to use for your random generation. It's much easier to understand how random generation changes with parameter inputs by trying it yourself than attempting to read the code (or read articles explaining this connection).

Once you've gotten a sense of terrain generation parameters and tested single examples, your next task is to add a way for the user to generate terrain with different parameters without reloading the page. Having a clickable button with a slider for some property (say "roughness") is a great way to implement this step. It may take some time for your terrain to load (especially for larger meshes), so don't panic if the web page freezes briefly.

Finally, your last task in this part is to hook up the user interaction from Part 2 to interact with your terrain randomness property. This interaction can be somewhat "hacky" -- it is perfectly acceptable to create a platform on the highest mountain peak that has your animation occuring on top. We only expect that your animation react to the random generation in some way, so you don't randomly have your animation "clipping" through terrain during some generation attempts.

Hint: This section of the project can be a bit overwhelming, but it's very likely not as hard as you think. If you're lost with the theory/writeup here, just try calling some functions and reading through the data they spit out. Hooking up a perlin noise image to a mesh or loading a heightmap mesh should require on the order of a few dozen lines of code, so you should consider rethinking your solution if you're writing a huge amount of logic. Similarly, hooking your animation up to the terrain does not *need* to be complicated, so consider simplifying this interaction if it's seeming overly daunting.

Part 4: Drivable Camera

For the last part of this project, you should implement a mode for your perspective camera that can "drive" along the terrain generated in Part 3. Note that you should consider this part to be somewhat optional (hence why this part is only worth 5 points), so make sure to get everything else working first.

Concretely, you must implement a mode where the camera is "locked" to the top of your ground and can be driven forward/backward and turned left/right. The camera should always remain stuck to the ground, and should maintain a ~90 degree up-down rotation angle from the current triangle you are "attached" to. To

calculate this angle, we suggest using the mesh normals discussed at the end of Part 3. Indeed, you can use similar logic to have a non-camera model "walk" or "drive" along terrain.

Hint: as with the freely rotating camera from Part 1, consider using slerp to invoke smooth rotation as you drive your camera along the ground

Hint: due to the up/down rotation requirement, you can calculate camera rotation purely from yaw and pitch, where yaw is given by user input and pitch is given by mesh normal angle. Recall that quaternions can be calculated from Euler angles.

Part 5: Writeup

For the last part of the project, you should create a write up. As with project 1, this should include:

- A descriptive title for your project (such as "Grass Waving in the Wind" or "Rainbow of Spinning Shapes")
- Your name and netid
- A short paragraph describing your project
- At least two images showing off your project
    - We recommend making at least one of these images show the result of your project after user interaction. This can help us out with grading if we can't figure out the user interaction for some reason.
- A 1 or 2 sentence summary of all the files submitted for the project
- Instructions for how to interact with your project
- If you are unsure on a rubric item, justification for how a feature in your project satisfies the requirements
    - This is totally optional, but if you're unclear on a rubric item but believe you met the requirements, adding this justification can help us see "what you were thinking"