Snow Tracker

An OpenGL project that simulates a dynamic trace in the snow.

The project has been made for the course Real-time Graphics Programming by Davide Gadia at Università degli Studi di Milano

Authors:

- Federico Maglione
- Matteo Sangalli

Contents

1 Overview

SnowTracker is an OpenGL application that aims to simulate the trace in the snow made by a car controlled by the player. The user can drive on an infinite terrain full of snow placed in a snowy environment.

Beside the main application, the development was focused also in developing various utility classes that can also be used in other OpenGL projects.

SnowTracker is a demo made to apply some theoretical concepts about 3D graphics. The user has no predefined goal.

1.1 Features

The main features implemented in the project are the following:

- Visualization of a trace in the snow created by the movement of the car controlled by the user. The trace is made by editing the position of the vertices of the terrain at runtime.
- Navigation of an infinite terrain.
- Driving a car by using throttle, brake and steering.
- 3 types of cameras: orbit (third-person view), follow (first-person view) and free.
- Generation of the terrain mesh at startup.
- Generation of the height map at startup using Perlin Noise. The height map is used to make the terrain irregular.
- Loading of the 3D models and textures.
- Visualization of snow particles falling from the sky through a particle system.
- Visualization of a skybox.
- UI for editing many parameters of the application.
- Regeneration of the trace as time passes (only for non-infinite plane).

2 Implementation

2.1 Project structure

This section explains the general organization of the project in terms of tools used, files' tree and code structure.

2.1.1 Development environment

The application has been written in C++ using OpenGL 4.1. It has been built using CMake on Windows. It has been developed using Visual Studio Code as main IDE.

There are several types of external and internal dependencies for the application to function properly.

External libraries:

- **imgui**: management of the GUI. Used to make a menu that allows to configure the environment.
- **glad**: interface for OpenGL API calls
- **glfw**: window creation, input handling and event management
- **glm**: mathematics
- **perlin_noise**: Perlin noise implementation
- **stb_image**: texture loading

Internal libraries

- **shader**: Shader implementation for the rendering pipeline.
- **model**: Provides features for managing models and mesh.
- **camera**: Features for camera management.
- **transform**: Features for handling transforms.
- **mesh** creator: Features for basic mesh generation.
- **particle**: Particle system.
- **trace**: Features for managing the trace on the terrain.
- **terrain**: Features for creating and managing terrain.
- **material**: Features for generating and applying materials to meshes.
- **light**: System for managing directional lights.

2.1.2 Project files' organization

The project is organized as follows:

- **assets**: contains all project assets
    - **models**: contains all models
    - **shaders**: contains all shaders (vertex and fragment)
    - **textures**: contains all textures
        - **skyboxes**: subfolder only for skyboxes
- **build**: contains all compiled files and the final .exe
- **include**: contains all libraries (external and internal)
    - assimp
    - glad
    - glfw
    - glm
    - imgui
    - perlin_noise
    - std_image
    - **utils**: contains all internal libraries

- **libs**: contains all files needed by included libraries
- **src**: contains main.cpp

The root folder contains various files such as Makefiles, Readme and .gitignore.

2.1.3 General code structure

The core of the application is written inside the main.cpp file. It is structured in some sections:

- **Includes**: this section is responsible of including all the libraries used by the main application
- **Variables setup**: this section is responsible of declaring all variables that will be used by the main application. These variables include general settings and variables that must be accessed by the main and other functions.
- **Application setup**: this section is responsible of initializing OpenGL and creating all objects that are part of the 3D scene. In this section textures, models and shader are loaded. The scene is created by placing every object (camera, terrain, player, ecc.) in the 3D space.
- **Main loop**: this section is responsible of updating every object based on player's input and rendering the scene.
- **Application dispose**: this section is responsible of freeing the memory before closing the application.

A big part of the application resides in the internal libraries. In this way, the main.cpp file is lighter and easier to understand and update.

2.1.4 How to import, compile and run

To import the project , from Visual studio -> File -> Open Folder -> SnowTracker Folder

As the examples in the class course, from the top search field in visual studio code, start the task :

Run task -> Build and Run

The project already has been compiled, you can find it in the build directory (build/main.exe)

2.2 Main features

This section explains the main features of the project. This involves the generation and visualization of the terrain, the handling of the trace left from the player and the movement of the car.

2.2.1 Terrain

The terrain describes the environment of the simulation. It is structured as a grid of chunks, where each chunk is a piece of terrain. Chunks can be edited and are rendered individually.

The terrain is obtained by using two classes:

- *Chunk*: stores a reference to a *Mesh* that represents the geometry of the chunk (explained in section Mesh), the size of the chunk and a pointer to a *Transform* object for information about the chunk's position, rotation and scaling (explained in section Transform).
- *Terrain*: stores a reference to each chunk and provides the methods to handle them.

2.2.1.1 Terrain creation

A *Terrain* is created by specifying the following parameters to create the grid of chunks:

- *rows* and *cols*: the number of rows and columns of the grid
- *chunkMesh*: the mesh assigned to every chunk. It contains data about vertices and faces. It's implemented as a *Mesh* object
- *chunkMeshSize:* The size of the mesh in world values

The constructor of the *Terrain* class takes these parameters to build the chunks and place them around the origin (0, 0, 0). The position of each chunk is calculated as an offset from the position of the central one.

2.2.1.2 The illusion of an infinite plane

Creating a terrain that appears infinite to the player could be done by increasing the number of chunks created. This solution is not efficient in terms of resources stored in memory and computed during rendering.

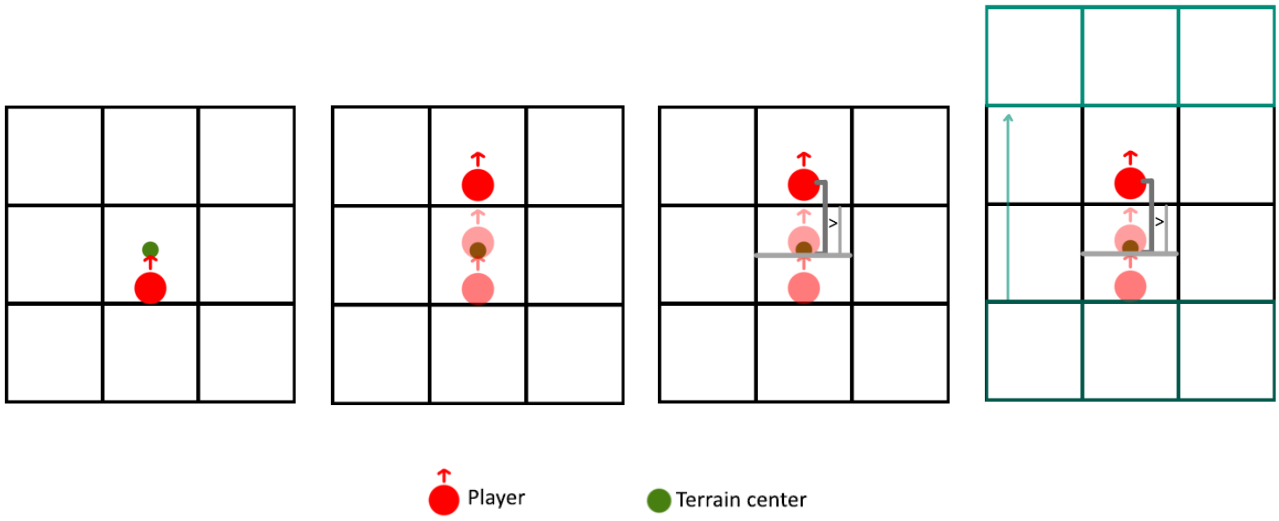
A better approach is to use only a few chunks centered around the player and update their position as the player moves.

The terrain is updated as follows:

1. The difference between the player's position and the position of the central chunk in the grid is calculated.
2. If this difference exceeds half the chunk size, on the x-axis or z-axis (positive or negative), the outermost chunk relative to the player moves in that direction. So, every time the player moves in each direction and exceeds the middle of the current chunk, chunks outside the player's field of view are moved in front of him.

By doing so, instead of generating new chunks every time the player moves, existing chunks are reused, working only with their transforms.

The following images explain this concept:



2.2.1.3 Terrain rendering

The terrain is rendered by rendering each chunk individually.

The base mesh assigned to each chunk is a flat plane. The irregularity in the elevation is obtained by an height map to move vertices of the mesh on the Y axis.

The height map is an image with 1 channel that stores values between 0 and 1. A value of 1 means that the vertex has to be moved by the maximum offset, a value of 0 means that the vertex has to stay in its original position. The height map can be read from a file or generated on-the-fly.

The new position of each vertex is computed during rendering in the vertex shader. Displacement mapping permits to edit the shape of the chunk simply by updating the vertices position by reading the value on the height map. Because the position of each vertex could change, the normal vector of each vertex must be recalculated to be compliant with the new position.

The color of each pixel of the chunk is calculated in the fragment shader based on the new position of the vertices and by using the Blinn-Phong lighting equation.

2.2.1.4 Shaders

In the project there are three files related to the rendering of the terrain: plane.vs, basic.vs and basic.fs.

- The two shaders basic.vs and basic.fs implement a generic “basic” shader that can be used for a variety of purposes. It takes a mesh, a directional light and other various optional parameters to render objects using the Blinn-Phong lighting equation.
- The plane.vs is a vertex shader based on the basic.vs that supports the rendering of the *Trace* based on a trace map accessible to the shader.

Each chunk of the terrain is rendered using the following shaders: plane.vs and basic.fs.

2.2.1.4.1 Structures

In both shaders, some common structures are used to organize parameters:

- *Vertex*: represents data for each vertex in the object. It contains information like the position of the fragment in the world (*fragPos*), the texture coordinates (*texCoords*), position details in tangent space (*tangentViewPos*, *tangentFragPos*), the Tangent-Bitnormal-Normal matrix (TBN), and the normal vector.

- *Material*: describes properties of the object's surface. It contains details on the color of the object (*diffuseColor*, *specularColor*), texture maps for these properties (*diffuseMap*, *specularMap*), shininess of the material, tiling information, and details about normal and height mapping.
- *Scene*: contains information about the scene like ambient light
- *Light*: describes the properties of a directional light source. It has information about the direction from which the light is coming and its color.

2.2.1.4.2 Vertex shader (plane.vs)

Function to Recalculate Normals:

The calculation of normal using a height map is a method to determine the new normals after applying a displacement mapping function. Rather than relying solely on the provided normals (which might represent only a smooth surface), this function uses the variations in intensity in a height map to determine how light should interact with the surface at a given point.

Main Function:

The heart of the vertex shader:

1. *Height Mapping Management*: If height Mapping is enabled, the shader modifies the y-position of the vertex based on the information contained in the height map and the trace map.
2. *Normal Calculation*: If height mapping is enabled, a new normal is calculated based on the height map and the trace map. In other words, the new normals are recalculated from the combination of the height map and trace map, which together identify the correct position of the vertices on the y-axis.
3. *Normal Mapping Management*: Normal mapping is another technique for simulating detail. Instead of deforming the geometry (as with height mapping), normal mapping creates an illusion by modifying the normals in the tangent space of the object to which the texture is applied, making it behave as if there were details on the surface. The shader prepares a TBN (Tangent-Binormal-Normal) matrix that transforms the normals into tangent space.
4. *Output Value Assignmen*t: Once all vertex details have been processed, the relevant information is assigned to the `vertex` output structure. This data is then used by the fragment shader to calculate the final appearance of each pixel in the object.
5. *Final Vertex Placement*: Lastly, the world position of the vertex is transformed into a screen position. This is done by multiplying the vertex position by the model, view, and projection matrices.

This vertex shader is designed to simulate complex details on the surface of an object through techniques such as height mapping and normal mapping. Height mapping effectively shifts vertices to simulate detail, while normal mapping alters the interaction between light and the surface without changing the actual geometry. The combination of these techniques can produce very realistic visual results without the overhead of complex geometries.

2.2.1.4.3 Fragment Shader (basic.fs)

Functions:

- *calculateDiffuse()*: Computes the diffuse reflection of light on the surface. The intensity of the diffuse reflection is dependent on the angle between the light's direction and the surface's normal. It also factors in the light's color, the material's diffuse color, and the color from the diffuse texture map at the given texture coordinates.
- *calculateSpecular():* Calculates the specular reflection. Specular reflection represents the bright, mirror-like reflections on a surface. The intensity depends on the angle between the viewer's

direction, the light direction, and the surface normal. The computed specular reflection considers the light's color, material's shininess, specular color, and the color from the specular texture map.

Main Function:

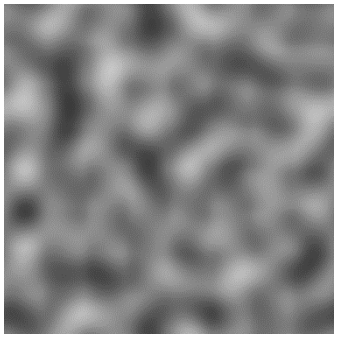
The heart of the fragment shader:

1. *Normal Mapping*: If normal mapping is enabled (material.normalMapping is true), the shader computes the normal from the normal texture map. The texture's RGB values are then converted from a [0, 1] range to a [-1, 1] range. The direction of the light and the view are also transformed into tangent space using the provided TBN matrix. Otherwise, the default normals and directions in the world space are used.
2. *Ambient Light Calculation*: Ambient light represents non-directional light that lights up everything in a scene to a certain degree. This is independent of light direction or object orientation.
3. *Diffuse & Specular Light Calculation*: Using the previously defined functions, it computes the diffuse and specular contributions based on the given light, normals, and viewer directions.
4. Result Composition: All the light components (ambient, diffuse, specular) are summed up to get the final color of the fragment.
5. *Output*: The calculated color is assigned to the output variable fragColor, which will be used by the OpenGL pipeline for further operations like blending before the fragment is drawn to the screen.

This fragment shader is designed to compute the color of a pixel (fragment) on a 3D object in a scene, taking into consideration the properties of the material it's made of, the lights in the scene, and the viewer's position. It offers features like normal mapping, which simulates surface details without modifying the actual geometry.

2.2.1.5 Height map generation using Perlin noise

In this application the height map is generated at startup using Perlin noise.



Perlin noise is a type of gradient noise. It assigns a numerical value to each point of a n-dimensional space.

The function *GenerateHeightMapFromPerlinNoise()* in main.cpp is responsible to generate the height map, with a predefined resolution, by using the function *noise2D_01(…)* of the perlin_noise.hpp library.

The resulting texture is binded to an OpenGL texture.

2.2.2 Trace

The main scope of the project is to visualize a trace left from the movement of a subject.

The idea to obtain this effect is simple: the path travelled by the subject is stored on a texture and it is used during rendering to "dig" the trace on a surface by moving its vertices.

2.2.2.1 Storing the trace

Storing the trace on a texture is like painting on a canvas. When the subject moves, its position controls a brush that "paints" the current position of the subject. This position is relative to a predefined area that defines the "canvas". If it's outside this area, the brush has no effect.

The trace texture is a texture with a single channel in which each pixel can assume a value in [0, 1]. If read as a greyscale image, 0 is black and 1 is white.

The trace texture has a resolution of 512x512 that permits to have a good result without being too heavy to update. The brush texture has a resolution of 256x256 that permits to the brush to be rescaled even for big subjects without losing quality. Resolution of both textures can be easily updated because all subsequent processing can adapt to different resolutions.

The trace texture is initialized as a black texture (all pixels at 0). When a brush paints the trace texture on a specific position, the pixels around this position become brighter (their value is increased). The actual value of each pixel depends on the brush texture and it cannot be decreased.

This behavior is obtained by the definition of three classes, all included in the *trace.h* internal library:

- *TraceBrush*: represents the brush used to "paint" the path. It is defined by a greyscale texture and a size (in 3D space units).
- *TraceArea*: represents a 2D area of the 3D space that can be "painted". It is defined by position and size (in 3D space units).
- *Trace*: stores the data of a greyscale texture that represents the current status of the path. The texture can be painted by the method *DrawWithBrush(TraceArea *area, TraceBrush *brush, glm::vec3 brushPosition)* that "paints" the texture with the texture of the *brush* in the *brushPosition* relative to the *traceArea*. The size of the *brush* and the *traceArea* permits to reason in terms of world units instead of texture resolution.

2.2.2.2 Visualizing the trace

As explained in the terrain section, the surface is a mesh rendered as an irregular shape by using an height-map. The visualization of the trace is performed by using the trace texture to rescale the standard height-map applied on this surface. The result is that in correspondence of the trace, the surface is flattened. This is performed by a custom vertex shader.

The vertex shader basic.vs implements standard height mapping by using 3 parameters:

- *heightMapping*: a boolean value that enables the use of height mapping.
- *heightWeight*: a float value that represents the maximum amount of translation of each vertex.

- *heightMap*: the default height-map of the surface. Each vertex is moved along Y axis of an amount defined by multiplying the *heightWeight* value with the value of the pixel in the height-map.

The vertex shader plane.vs uses one more parameter to view the trace:

- *traceMap*: the texture that contains the trace left from the subject.

The final amount of translation of each vertex is calculated by multiplying the value of the pixel in the *heightMap*, the value of the pixel in the *traceMap* and the *heightWeight*.

2.2.2.3 Trace regeneration

The *Trace* class provides the attribute *regenerationSpeed* and the method *Regenerate(float regenerationAmount)* that permits to decrease the value of each pixel of the trace texture by a value *regenerationAmount*. This can be used to gradually reset the trace texture simulating some falling snow that makes the trace disappear.

This method can be computationally heavy if called at each frame for textures with large resolution because it needs to edit each pixel of the texture.

2.2.3 Car movement

In this project, the player can control a car that moves on a flat surface. A realistic movement system for the car based on real physics is not part of the project so the car movement has been implemented using a simple simulated physics.

On each frame, the car updates the value of each input (throttle, brake, steering angle), computes the new value of speed and finally updates its transform accordingly.

This behavior is implemented in the *Car* class contained in the *car.h* internal library.

The general settings of a car can be set by editing the following attributes:

- *maxSpeed*: the max speed of the car used to clamp the speed value in [0, maxSpeed]. Expressed in m/s.
- *accelerationLinear*: based on the throttle value, the car accelerates by a fraction of this value. The acceleration of the car is linear. Expressed in m/s.
- *brakeFrictionLinear*: based on the brake value, the car decelerates by a fraction of this value. The brake friction is linear. Expressed in m/s.
- *generalFrictionLinear*: simulates a generic friction applied to the car by many factors like air, road, etc. Car decelerates constantly by this value. The general friction is linear. Expressed in m/s.
- *steeringTime*: the time to go from zero to full steering. Expressed in seconds.
- *antiSteeringTime*: the time to go from full steering to zero. Expressed in seconds.

The status of the car is defined by the following attributes:

- *steeringAngle*: the current angle of steering. Value is in [-90, +90] where -90 means “fully steer to the left” and +90 the opposite
- *throttle*: the current amount of pressure of the throttle. Value is in [0, 1].
- *brake*: the current amount of pressure of the brake pedal. Value is in [0, 1].
- *speed*: the current speed of the car. Value is in [0, maxSpeed].
- *transform*: the current transform (mainly position and rotation) of the car.

2.2.3.1 Updating input values

The values of the inputs are updated at each frame in the *Update(…)* method.

The car is controlled by pressing WASD keys where W is the throttle pedal, S is the brake pedal and A and D permit to rotate the steering wheel left and right respectively.

- Throttle input is simple. If the throttle key is pressed, *throttle* value is set to 1. When the key is released, *throttle* value is set to 0.
- The brake input works the same way as the throttle input.
- The steering input undergo some processing to let the player control the car more precisely and realistically. Instead of setting *steeringAngle* suddenly to -90, 0 or +90, the value is gradually updated while the player is holding (or not) the steering key. When the player presses the button to steer to the left (or right), the value gradually reaches the value -90 (or +90). When the player releases both the steering keys, the value gradually decreases to zero. The speed of change of the *steeringAngle* value is managed by the attributes *steeringTime* and *antiSteeringTime*.

2.2.3.2 Computing speed

The speed of the car is updated at each frame in the *Update(…)* method.

It is based on the current speed, the current input values and the acceleration and frictions attributes. It is computed as follows:

```
float throttleSpeedAmount = (this->throttle * this->accelerationLinear) * deltaTime;
float brakeSpeedAmount = (this->brake * this->brakeFrictionLinear) * deltaTime;
float generalFrictionSpeedAmount = this->generalFrictionLinear * deltaTime;

this->speed = glm::clamp(this->speed + throttleSpeedAmount - brakeSpeedAmount -
generalFrictionSpeedAmount, 0.0f, this->maxSpeed);
```

Note: *deltaTime* is the amount of time passed from the last frame to the current frame.

2.2.3.3 Updating transform

The transform of the car (position and rotation) is updated at each frame in the *Update(…)* method.

If the car is moving (*speed* > 0), the car transform has to be updated:

- The position is updated by moving the car forward by a *speed* amount. The car forward direction is obtained by the *transform* attribute.
- The rotation is updated by rotating the car by a *-steeringAngle* amount on the local up axis of the car. The local up axis is obtained by the *transform* attribute. The value of *steeringAngle* is flipped because rotating by a positive value means rotating counterclockwise.

2.3 Secondary features

This section explains some features that has been developed to make the application more complete. These features were not necessary to demonstrate the correct visualization of the trace on the terrain but they make the application more enjoyable and introduce many general components that can be used also in other projects.

2.3.1 Camera

The camera has been designed as an object that holds information necessary to compute the view matrix and the projection matrix. It stores its position and its orientation in the 3D space and values about FOV, aspect ratio, near plane and far plane.

The camera can be used in 3 different modes:

- Free: the camera can be freely moved in the 3D space using arrow keys and it can be rotated using the mouse.
- Follow: the position of the camera is linked to the position of another object, eventually with an offset. It can be rotated using the mouse. If the player is set as the object to follow, this mode can be used to achieve a first-person view.
- Orbit: the position of the camera is linked to the position of another object and it can be rotated around the object using the mouse. The camera always looks to the object set as target. If the player is set as the object to follow, this mode can be used to achieve a third-person view.

In every mode, the mouse scroll wheel permits to zoom in and out by changing the field-of-view (FOV).

The camera mode can be changed by pressing the C key.

The camera is implemented by the *Camera* class in the camera.h internal library.

2.3.2 Particle system

The particle system is responsible for generating particle effects in the scene. In this application, a particle system is used to simulate the effect of a snowfall.

2.3.2.1 Particle system class

The particle system consists of various properties that allow you to change the behavior of the particles:

- spawn rate
- lifetime
- alpha time
- particle To Spawn
- speed
- gravity

All particles are stored in a vector of *Particle.*

A *Particle* is a struct that stores the following attributes:

- Position
- Velocity
- Color
- Size
- Life: the remaining life of the particle
- Distance: the distance of the particle from the camera

The particle system class consists of two main functions:

- *Update*: a method called every frame that is responsible for generating particles based on the spawn rate, updating the properties of each particle (position, color, size, distance from the camera, etc). In addition, particles are sorted by their distance from the camera, to ensure that particles closer to the camera will be drawn first. This allows for correct handling of transparency.
- *FirstUnusedParticle*: a method that determines the first "dead" (when life goes under 0) particle in the particle vector that can be reused again.

2.3.2.2 Rendering

2.3.2.2.1 Application stage

The shader program to be used for the particles is first activated. The projection and view matrices are set

The code then sets the diffuse map, which is the basic texture used on the particles.

After setting up the necessary variables and textures, the code updates the state of the particle system. This involves updating the position and velocity of the particles based on the elapsed time, creating new particles, or removing old ones.

2.3.2.2.2 Vertex shader

This vertex shader is designed for a 3D particle system that implements a technique called "spherical billboarding". This technique is often used in 3D graphics to make an object always face the observer. It's particularly useful in particle systems, where individual particles are represented by a 2D image.

Particles are created from a centered square, which can be identified as the emitter of the system. Each particle has a specific position and a specific size. These two properties are calculated, as previously described, in the application stage.

The shader takes the positions of the particles and maps them into the camera's view using the view and projection matrices.

Spherical billboarding is implemented by modifying the position of each particle based on its original position, its size, and its relative position to the camera. In this way, each particle will always be facing the camera, no matter from which angle it is viewed.

Finally, the output of the shader that includes the color of each particle and the UV coordinates for texturing are sent to the fragment shader to determine the final appearance of each particle.

2.3.2.2.3 Fragment shader

The task of the fragment shader is to calculate the final color of each pixel that makes up the rendered particles on the screen. This is done using information passed from the vertex shader and uniform variables.

The color of each particle and the UV coordinates for texturing are received as input from the vertex shader. This information is used together with a diffuse texture.

The final color of the fragment is then calculated by multiplying the color of the particle by the color of the pixel taken from the diffuse texture at the specified UV coordinates.

2.3.3 Skybox

The skybox used is based on a cube map representing a texture that contains 6 individual textures, one for each side of the cube. This class has been taken from lab lectures from Davide Gadia.

A cube map is created by using the function *LoadTextureCube*. This function creates a new cubemap texture object, binds it, and then loads six image textures onto it, representing each face of a cube.

The loading of the six image textures is performed by the function *LoadTextureCubeSide*. This function loads an individual image file from the disk and assigns it to one side of the cube map.

2.3.3.1 Rendering

2.3.3.1.1 Vertex Shader

Texture Coordinate Calculation: Here, the vertex position (aPos) is directly used as the 3D texture coordinates (interp_UVW). This is applied when rendering skyboxes or environment maps where each vertex of the cube corresponds directly to a position in the cube map.

Position Transformation: The vertex's position is calculated by several transformations to go from its local model space (aPos) to clip space (pos). This transformation chain is achieved by multiplying the vertex position by the model, view, and projection matrices in that order.

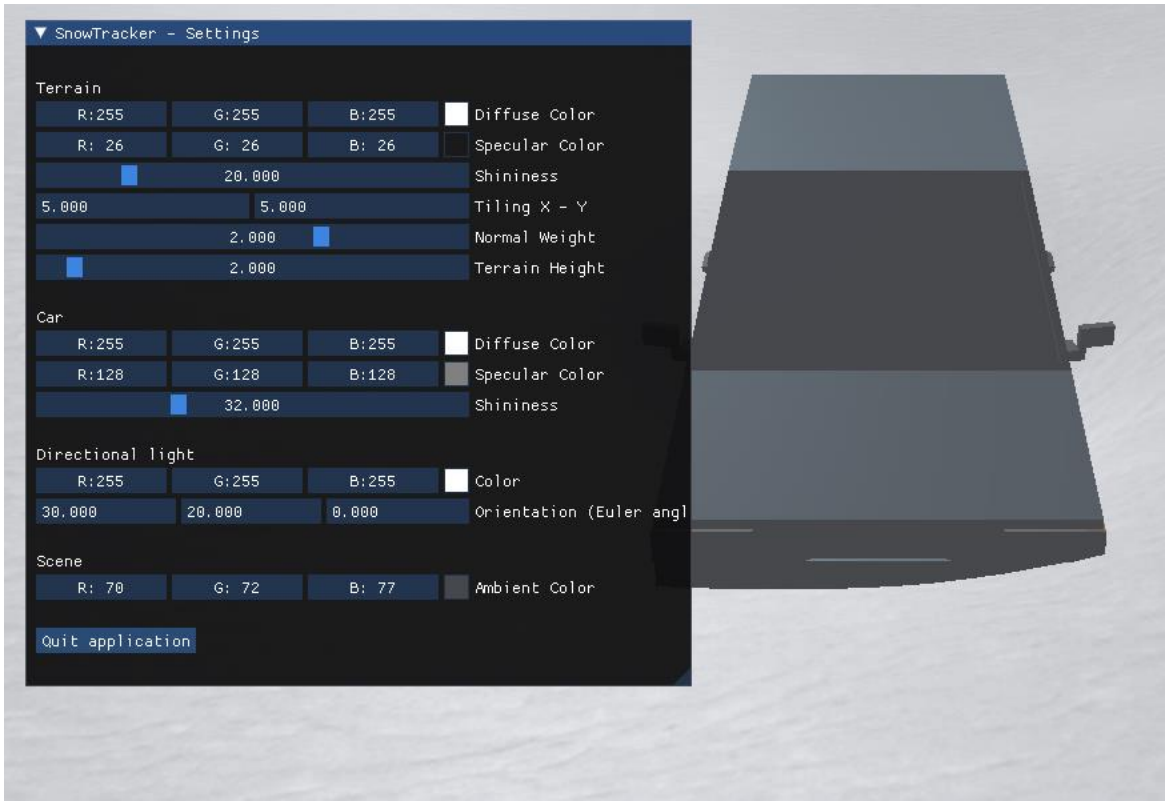
Depth Trick: This is the cooler part of this shader. The line gl_Position = pos.xyww; takes the transformed x and y coordinates of the position (pos.xy) and the w component for both z and w (pos.ww). This is a trick to ensure that when the vertex undergoes perspective division later in the pipeline (where each component of gl_Position gets divided by its w component), the z value (depth) of the vertex will always be 1.0. The purpose of this is to ensure that the skybox or environment map is always rendered "behind" all other geometry in the scene. By setting the depth to the maximum value, other objects will always pass the depth test in comparison and be rendered in front of the environment map.

2.3.3.1.2 Fragment Shader:

This fragment shader is defined to render an environment map (or skybox) using a cube map. The shader samples a cube map using the 3D texture coordinates provided by the vertex shader and outputs the corresponding color. This results in a projection of the environment onto the background of the rendered scene. The shader is very simple, focusing on displaying the environment without any additional effects or manipulations.

2.3.4 GUI Configuration

The GUI has been implemented by using the imgui plugin.



By pressing ESC key, it is possible to access the GUI to configure the following parameters:

Terrain:

- Diffuse Color
- Specular Color
- Shininess
- Tiling
- Normal Weight
- Height map Weight

Car:

- Diffuse Color
- Specular Color
- Shininess

Directional Light:

- Color
- Orientation

Scene:

- Ambient Color

2.4 Utilities

This section explains some utility classes and libraries used in the application.

The classes *Transform*, *DirectionalLight* and *Material* has been created for this project but try to be as general as possible to be used also in other OpenGL projects.

The classes *Mesh*, *Model* and *Shader* has been taken from lab lectures from Davide Gadia and are based on the relative classes used by learnopengl.com.

The libraries Mesh Creator and Texture Loader contains some utility functions that are specific to this project.

2.4.1 Transform

The *Transform* class is a utility class that stores data about position, orientation and scale of an object. Data is expressed in the local reference system of the object. The *Transform* class is defined in transform.h internal library.

It provides useful methods to:

- move, rotate and scale an object in the 3D space
- query location, orientation and scale of an object in the 3D space
- get derived data of an object such as the local model matrix and the orientation vectors (up, right and forward)

To use these features, an object that needs to be placed in a specific location of the 3D space requires only to have a *Transform* attribute. Other objects can interact with its *Transform* attribute to edit or query data.

2.4.2 Directional Light

The *DirectionalLight* struct represents a directional light source in the 3D scene. This struct is really simple thanks to the use of the *Transform* class. The *DirectionalLight* struct is defined in light.h internal library.

A *DirectionalLight* object contains the following attributes:

- *transform*: the *Transform* of the light. Since it is a directional light, rotation is the only useful data to set, position and scale are ignored.
- *color*: the color of the light. It is stored as a glm::vec3.

To use the directional light, it's enough to define a DirectionalLight object, set its attributes and pass its forward vector and its color to a shader that supports directional lights. In this application, the shader basic.fs supports directional lights by setting the attributes of the *Light* struct.

2.4.3 Material

The *Material* class is used to store data that needs to be passed to shaders for rendering specific objects. It also contains a method to easily set data to the shader. The *Material* class is defined in material.h internal library.

The *Material* class is an abstract class that needs to be subclassed for each shader created. Each subclass has to specify all the parameters that could be passed to the shader relative to the material. In addition, each subclass has to override the method *virtual void ApplyToShader(Shader shader)* that assigns all the parameters to the shader.

In the material.h there are 3 subclasses of *Material*, each one relative to one shader:

- *BasicMaterial*: contains data to be used with the "basic" shader (basic.vs and basic.fs)
- *ParticleMaterial*: contains data to be used with the "particle" shader (particle.vs and particle.fs)

- *SkyboxMaterial*: contains data to be used with the “skybox” shader (skybox.vs and skybox.fs)

To use a *Material*, it’s enough to define a *Material* object by using one of the defined subclasses, setting the attributes and, during rendering, calling the method *ApplyToShader* with the target shader as parameter.

2.4.4 Mesh

The *Mesh* class is used to setup all data required by OpenGL to store and render a mesh. This class has been taken from lab lectures from Davide Gadia and it’s based on the *Mesh* class used by learnopengl.com.

A Mesh object is created by passing two arrays:

- Array of vertices: each entry is a *Vertex* object. A *Vertex* object contains data about vertex position and optionally contains data about normal, texture coordinates (UV), tangent and bitangent.
- Array of indices: each entry is an index used to reference a Vertex object in the array of vertices. Three subsequent indices define the vertices that make a face of the mesh.

At mesh creation, all buffer objects required by OpenGL are created:

- Vertex Buffer Object (VBO): a buffer that stores data about vertices
- Element Buffer Object (EBO): a buffer that stores data about indices
- Vertex Array Object (VAO): a buffer that helps to "manage" VBO and its inner structure. It stores pointers to the different vertex attributes stored in the VBO. When we need to render an object, we can just bind the corresponding VAO, and all the needed calls to set up the binding between vertex attributes and memory positions in the VBO are automatically configured.

Once the *Mesh* object has been setup, the mesh can be rendered using the *Draw* method that binds the VAO and calls the procedure to draw the faces.

2.4.5 Model

The *Model* class is used to load 3D models using the Assimp library and to store the data in a *Mesh* object, ready to be rendered by OpenGL. This class has been taken from lab lectures from Davide Gadia and it’s based on the *Model* class used by learnopengl.com.

A *Model* object is created by passing a path to a 3D model. The 3D model is loaded by Assimp and the data is stored in an array of *Mesh* objects.

Like a *Mesh* object, a *Model* can be rendered by using the *Draw* method. It simply calls the *Draw* method for every *Mesh* in the array.

2.4.6 Shader

The *Shader* class is used to read the code of shaders from file, compile it and provide methods to set uniforms. This class has been taken from lab lectures from Davide Gadia and it’s based on the *Shader* class used by learnopengl.com.

A *Shader* object is created by passing the path of the two files that contains the code of the vertex shader and the fragment shader. The code of both files is then compiled and linked in a single OpenGL Shader Program.

When an object has to be rendered, a Shader has to be “activated” by using the method *use()*.

A Shader object provides methods to easily set many types of uniform by specifying the uniform id and the data to pass. Here are some examples of methods for various types:

- `void setBool(const std::string &name, bool value) const`
- `void setInt(const std::string &name, int value) const`

- `void setFloat(const std::string &name, float value) const`
- `void setVec2(const std::string &name, const glm::vec2 &value) const`
- `void setVec3(const std::string &name, const glm::vec3 &value) const`

At the end of the program, is a good practice to delete every OpenGL Shader Program by calling the method *deleteProgram()* on the *Shader* object.

2.4.7 Mesh creator

The internal library mesh_creator.h provides some functions to create *Mesh* objects with basic shapes. This is mainly used for testing purposes.

These functions are:

- *Mesh CreateCube()*: creates a cube of a unit size. Each vertex only contains data about position.
- *Mesh CreateFlatPlane(int cols, int rows, float cellSize)*: creates a flat plane as a grid of vertices with the specified number of columns and rows. The size of each cell of the grid is specified by the *cellSize* parameter. Each vertex contains data about position, normal, tangent and bitangent.
- *Mesh CreatePlane(int cols, int rows, float cellSize, float *heightMapData, glm::vec2 heightMapDimensions, float height)*: same as the function before but each vertex is moved along Y axis based on a height-map passed as argument. Simulates the behavior of an height-map but at application stage.

2.4.8 Texture loader

The internal library texture_loader.h provides some functions to load textures from storage using functions contained in std_image.h. Optionally, textures are binded to OpenGL textures. The library includes a struct *TextureData* to manage main data of a texture: width, height, channels, pointer to pixel data.

The functions are:

- *Glint LoadTexture(const char *path)*: load a texture from a file, binds it to an OpenGL texture and returns its ID.
- *TextureData LoadTextureWithoutOpenGL(const char *path, int req_comp = STBI_rgb)*: load a texture from a file and returns a *TextureData* object. The parameter *req_comp* allows to specify how many channels of the image have to be considered.
- *Glint LoadTextureCube(std::string path)*: load a cubemap from a file, binds it to an OpenGL texture and returns its ID.

3 Conclusions

3.1 Results

The application meets the goals that have been planned at the start of the development:

- The player can control a vehicle on a surface full of snow and leaving a trace by flattening the snow.
- The surface is irregular based on a height-map generated through Perlin noise.
- The shape of the trace (the "brush") can be changed simply by updating a texture.
- The snowfall and the skybox contribute to make the environment more polished.

In addition, the application achieves some extra goals, such as:

- Most of the classes are general and are not only tailored to this application. They can be used with little or no updates in other OpenGL projects.
- The general design of the application permits to add or remove features easily.
- The *Camera* class provides many modes and can be easily integrated in an OpenGL application.
- The *Transform* class, even if is not complete, provides the fundamental methods to easily manage the presence of every object in the 3D space.
- The GUI permits to easily change settings while the application is running.

3.2 Limits

The application has some known limits in terms of features and performance. Some aspects are not "perfect" because they are not part of the main focus of the application.

The application known limits are the following:

- **Trace repetition among the infinite plane**: while driving in the infinite plane, the user can see some traces in the snow even if he hasn't been there. This is due to the repetition of the chunks of the terrain. A solution can be resetting the trace every time a chunk is loaded. This solves a problem but if the player comes back to a previous visited chunk, the trace that he left before is lost. Another solution can be centering the trace map on the player and updating it accordingly as it moves. This solves the problem and optimize the use of the memory but the trace left some meters before is lost again.
- **High number of vertices for the terrain mesh**: to have a trace with good quality, the terrain requires a high density of vertices. Even if the terrain is divided in chunks, it's not efficient to have an high number of vertices for areas away from the camera. A solution is to implement a tessellation shader to increase the number of vertices only for an area near the camera and prefer a lower density for further areas.
- **Regeneration of trace only works with a non-infinite plane**: while "drawing" the trace only requires updating a single texture only when the player moves, the regeneration of the trace requires updating all the pixels of every trace map on every frame. With the infinite plane, this implies updating multiple trace maps at high resolution instead of just one. A solution can be reducing the trace map resolution, reducing also the trace quality. Another solution can be use only a trace map centered on the player like explained before.