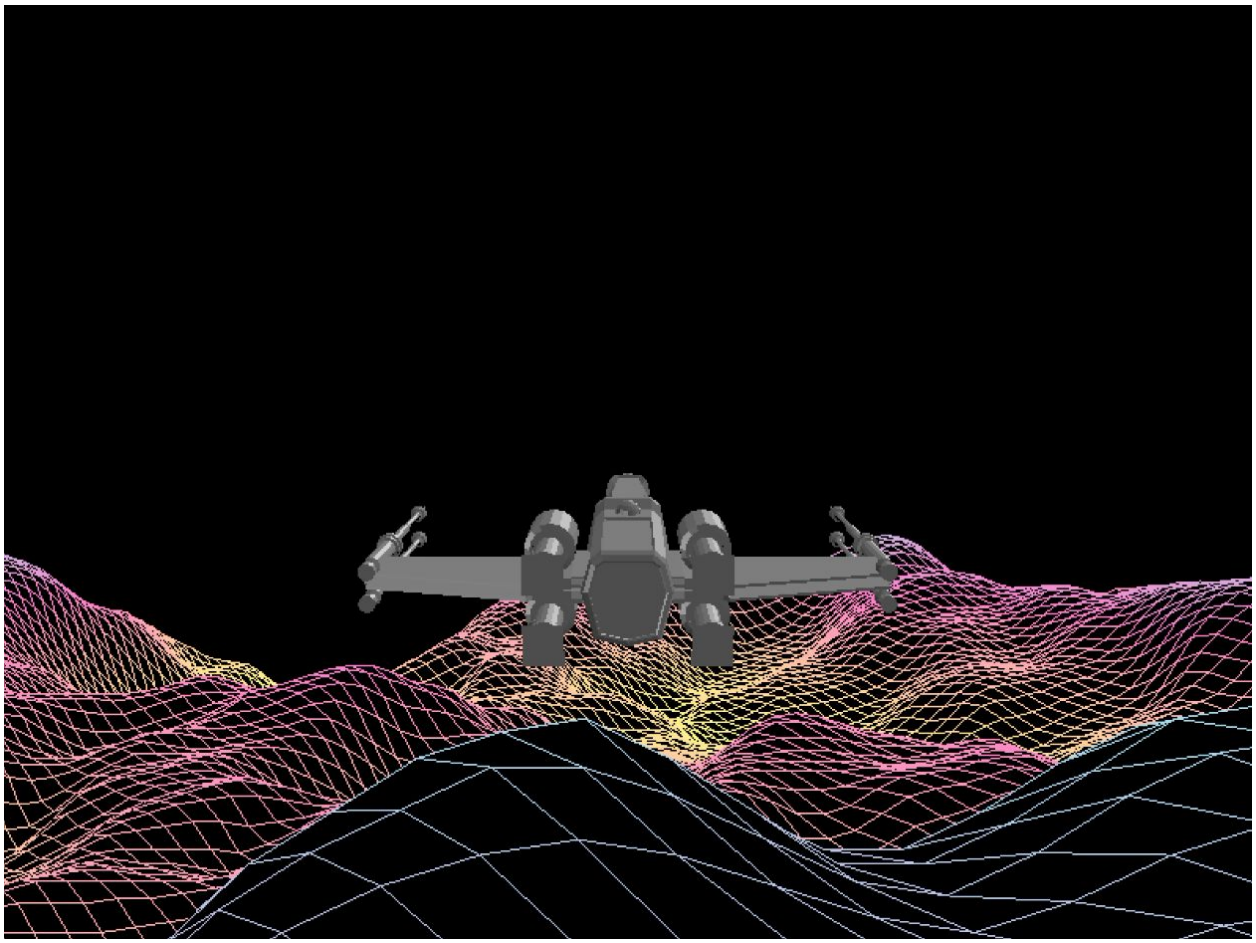
Terrain Generator

José Luis Rojas Aranda, 258177
Computer Graphics
José Ignacio Nuñez Varela



Introduction

The project consists of a procedural generated terrain, and a X-wing fighter that flies through the terrain avoiding crashing with the mountains. It uses Perlin noise to generate the terrain, which is also used in computer graphics to add realistic detail to materials, it is also widely used for generating terrains in video games because it saves a lot of modeling time. The purpose is only to show a demo of one technique used in computer graphics. The reason why I choose to do this, is that if executed right, we can get a really good-looking project and pleasing to see, because you don’t know what to expect from generator.

OpenGL

OpenGL is an API for rendering 2D and 3D graphics, is cross-platform and cross-language. Since its introduction in 1992, OpenGL has become the industry's most widely used, this interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications. OpenGL also takes advantage of graphics processing units (GPUs), to make more complex computation easier.

OBJ File Reader

Description

The most common way that a 3d model is represented is through a polygon mesh, which consists of lists of vertices and faces. These lists can be stored in different file formats, one of the most common is an OBJ file, this is due to the fact that it uses ASCII to encode the information thus making it easier to read. An OBJ reader consists of a program that is able to read this .obj file and store the information in a data structure, in this case a C++ object. Also, the OBJ file format allows us to store more than one 3d object in one file. With this information loaded in the program we can use a graphics library to draw the 3d model for example OpenGL. The OBJ reader also allow us to use and modify model that we can download from the internet that somebody else made.

Design

OBJ File Format

An OBJ file consists of 3 elements, each of these elements can be differentiated by their prefix, o for the object name, v for defining a vertex, f for defining faces. Faces are represented by the number of the vertex defined earlier starting by one, note that we can also have more than one object in the file. For example, this is the information of a cube:

- Object name:
  o Cube

- list of vertices:
  v 1.000000 -1.000000 -1.000000
  v 1.000000 -1.000000 1.000000
  v -1.000000 -1.000000 1.000000

- list of faces:
  f 1 2 3 4
  f 5 8 7 6
  f 1 5 6 2

OBJ Reader

The OBJ reader programs reads line by line of the file, and does the following:

1. Creates and empty lists of objects
2. Parses the line and separates all the elements separated by a blank space.
3. Checks the prefix of the line and if it is o, v or f. Does the following:
   a. If it is an o, add a new object to the end of the list with the corresponding name.
   b. If it is v and the object list isn't empty, add the vertex to the last object of the list.
   c. If it is f and the object list isn't empty, add the face to the last object of the list.
4. If the prefix isn't o, v or f. Continues to the next line.

Note that we need to subtract 1 to the face vertex index and if there are more than one object in the file, we need to subtract the number of vertices from the last object.

Models

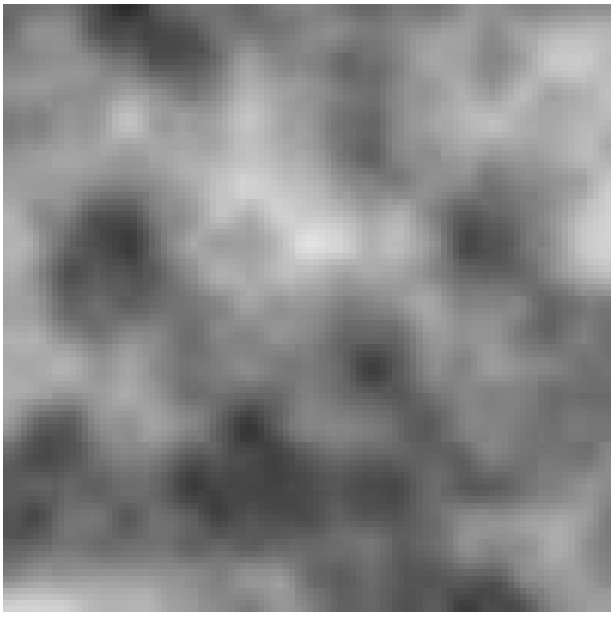
Terrain Generator

The terrain mesh is generated from a height map, which is an image generated using perlin noise. In this image the value of every pixel represents the height of the terrain at that given point.

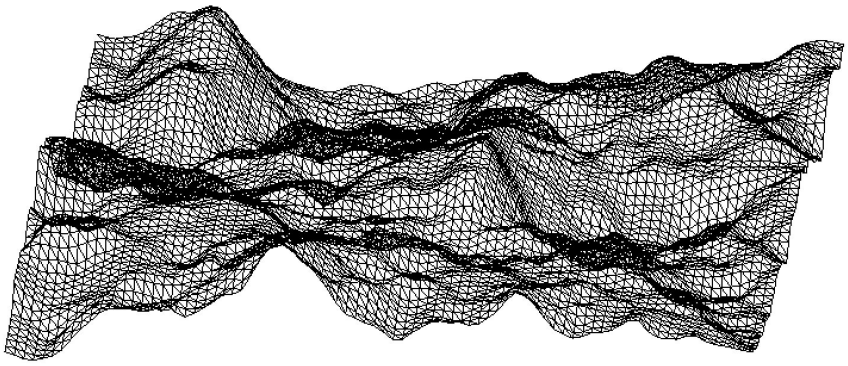
Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. An implementation typically involves three steps: grid definition with random gradient vectors, computation of the dot product between the distance-gradient vectors and interpolation between these values. The algorithm has a complexity of $O(2^n)$, where n is the number of dimensions of the gradient vector.

I am using the following implementation of Perlin noise, refactored to an object oriented fashion: https://gist.github.com/nowl/828013

The following image is a height map generated using Perlin noise:



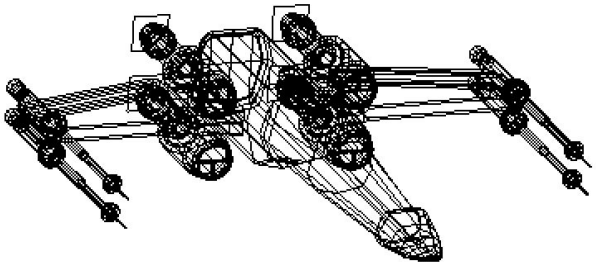
From this height map generated, we generate the mesh using triangles and use the height as the normalized gray scale value multiplied by the max height. Currently we are only considering that the terrain will be drawn only as wireframe.



Currently for the height map generation the only tweakable parameters are the size, the frequency and the depth. This gives us a lot of freedom to achieve different kinds of terrain, but there are also more ways to add more real life like appearance but isn't the purpose of this project right now.

X-Wing

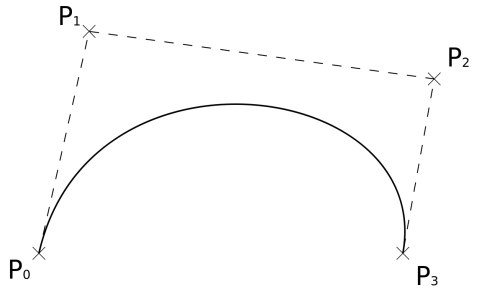
We are using a model of the X-Wing fighter from Star Wars, the model is low poly but we used blender to lower the faces count, the original model can be found here: https://sketchfab.com/3d-models/textured-x-wing-low-poly-32c242b812e549b2aa632373fd994e95



Movement

Bezier Curves

A Bezier Curve is a parametric curve that uses the Bernstein polynomials as a basis, in this case we are using a cubic polynomial because lower-degree polynomials are not so flexible, and higher-degree polynomials require more computation and are more difficult to handle. The Bezier form of the cubic polynomial curve segment indirectly specifies the endpoint tangent vector by specifying two intermediate points that are not in the curve:



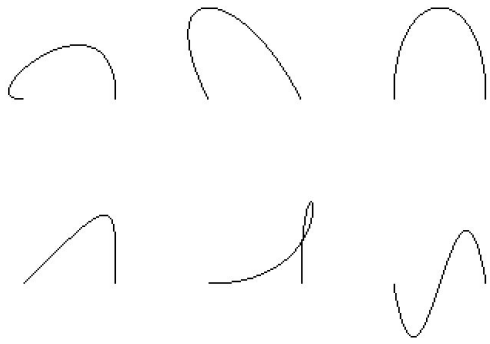
The starting and ending vectors are determined by the vectors P_0P_1 and P_2P_3 and are related to R_1 and R_4 by:

$$R_1 = Q'(0) = 3(P_1 - P_0)$$
$$R_2 = Q'(1) = 3(P_3 - P_2)$$
$$Q(t) = T \cdot M_B \cdot G_B$$

The final parametric formula is:

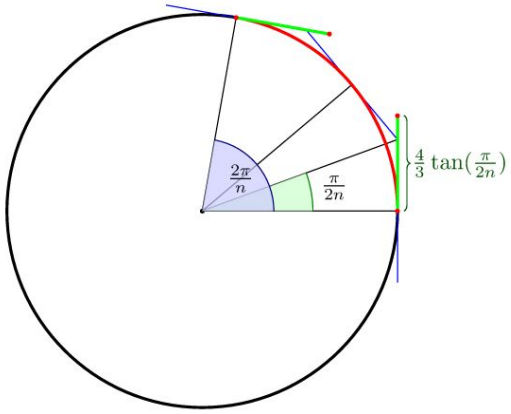
$$Q(t) = (-t^3 + 3t^2 - 3t + 1)\cdot P_0 + (3t^3 - 6t^2 + 3t)\cdot P_1 + (-3t^3 + 3t^2)\cdot P_2 + (t^3)\cdot P_3$$

The following are examples of different Bezier curves:



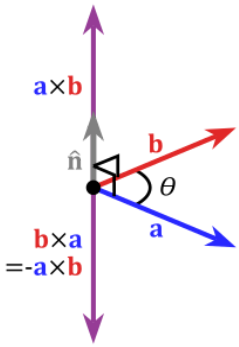
Approximating a circle with bezier curves

The movement consists of a circuit around the terrain, so this means that all the movement can't be described with a single Bézier curve we need to concatenate various curves, this is done by approximating a circle with n segments with Bézier curves, is calculated this way:



$$\theta = (2 * \pi) \,/\, segments$$
$$\alpha = (4/3) * tan(\pi/(2 * segments))$$

With this formula we obtain the magnitude of the vector from the auxiliary points of the Bézier curve, in order to calculate the actual coordinates of the points, we use the cross product in respect with the segment point and some arbitrary perpendicular vector.

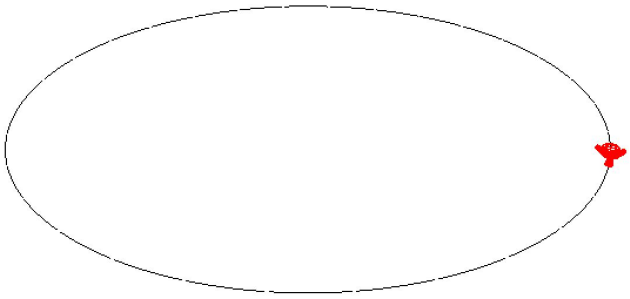


The formula looks like the following, where c and d are control points P_0 and P_1 respectively:

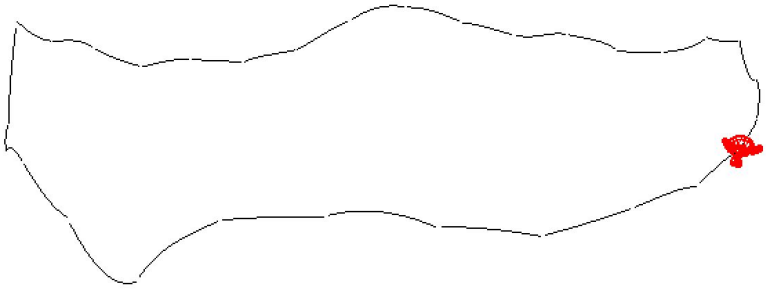
$$a = (cos\ \theta,\ 0,\ sin\ \theta)$$
$$b = (0,\ -\alpha,\ 0)$$
$$c = a * radius$$

$$d = (a \times b) + c$$

With this we can graph the obtained concatenation of Bézier curves:



Finally we can adapt the height of the points of the Bézier curve, so that the flying object moves around the map depending on the height of the terrain mesh.



Movement v2.0

Originally the X-wing moved in circles, that's why we approximate the circle using bezier curves, but we decided to make a simpler version that looks better, in which the X-wing move in a straight line and is avoiding crashing to a mountain. This is done by always maintaining a separation with respect to the ground, but we smooth this in the following way:

$$\lambda \in [1, \infty]$$
$$separation = terrainHeight(x, z)$$
$$\Delta y = (seperation - y) / \lambda$$
$$y := \Delta y$$

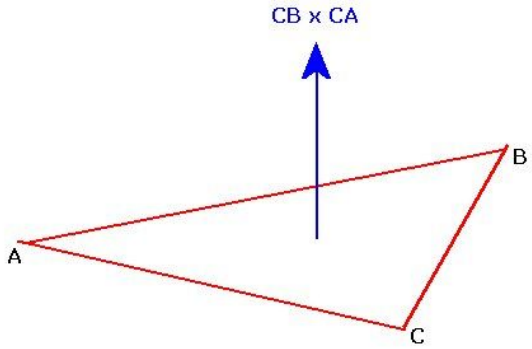
Where λ is a constant that indicates the amount of smoothing to be applied

Illumination

Normal and visible surface determination

An important thing to do, is to calculate the normal of the faces of our 3d objects. This helps us to first determine if the surface is visible to the camera, this allows us to save resources and to not see through objects, and second it serves when calculating the illumination of the surface.

To obtain the normal, we use the cross-product as shown in the following image. One important note is that the order of ABC, need to be counter clockwise if we see it from the front.



To determine if face is visible we perform the dot product with the PRP of the perspective camera, so that

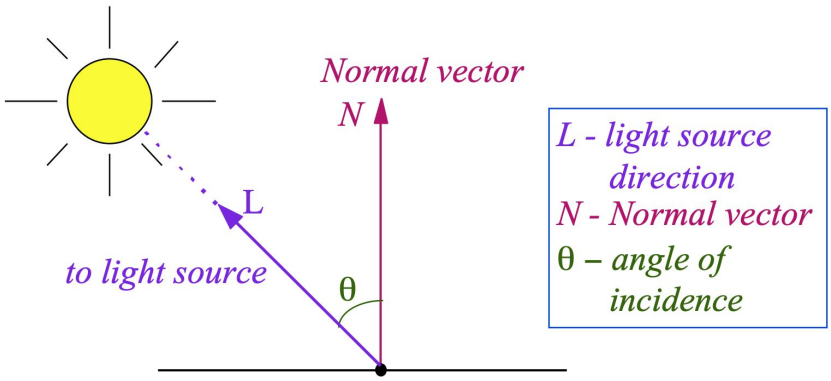
$$If:\ N \cdot PRP > 0 \ \Rightarrow\ the\ surface\ is\ visible.$$
$$If:\ N \cdot PRP == 0 \ \Rightarrow\ you\ can\ draw\ it\ if\ you\ want.$$
$$If:\ N \cdot PRP < 0 \ \Rightarrow\ the\ surface\ is\ not\ visible.$$

Painter Algorithm

One important thing to take into consideration is that the order in which we draw matters, this applies to faces inside object and between objects, because if you draw first something that is near and then something that is far away it won't look right. One simple way to solve this is to use the Painter Algorithm, which consists in ordering by distance and drawing first what is furthest away from the camera, this ensures that some object or face doesn't obstruct others and looks right.

Illumination Model

Once we have the normal, we can calculate the illumination of that face, for this we use an illumination model.

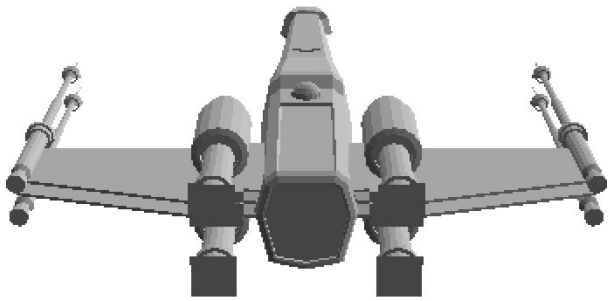


There exist various models, but we use one that is simple were we take in consideration ambien light and diffuse light:

$$I = Il * Kd * (N \cdot L) + Ia * Ka$$
$$Ka + Kd \leq 1$$

Where Ia and Ka are the intensity of ambient light and ambient reflection, Il and Kd are the intensity of incidence and diffuse reflection. It also important to make sure that N and L are unit vectors.

The following image is an example of how it looks, once applied to a 3d model:



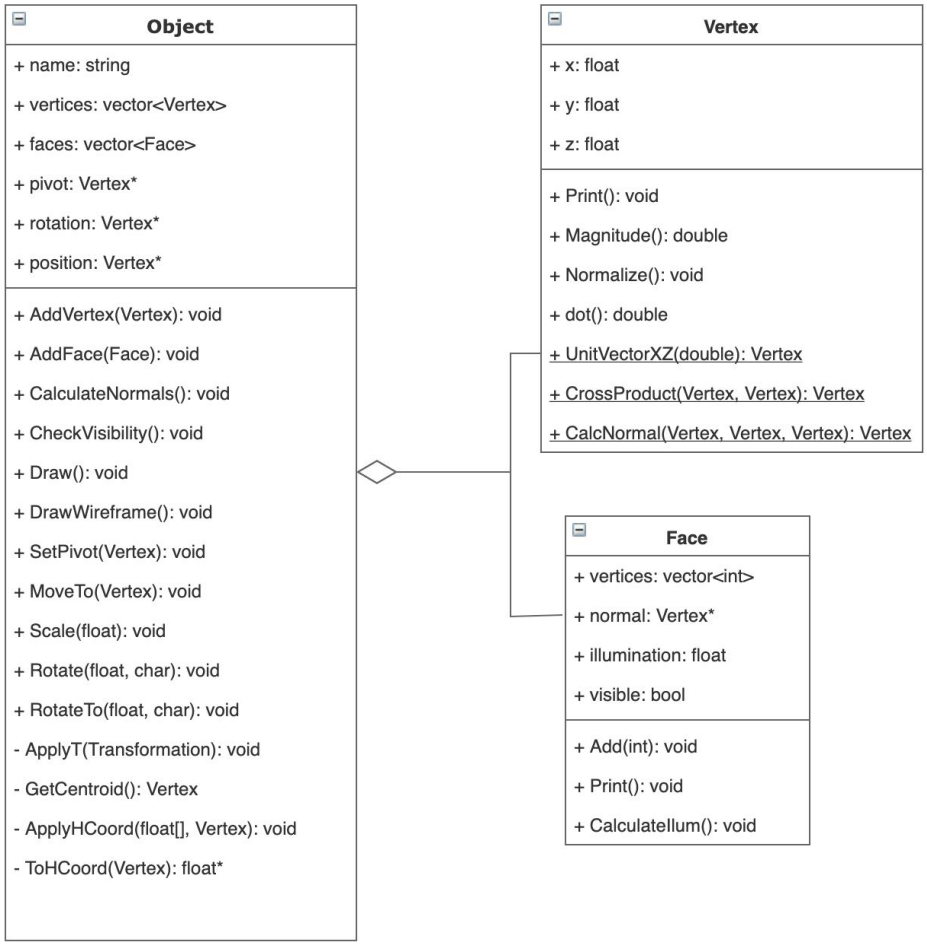
Code Structure

The project was created in C++, we an object oriented programming approach for structuring the code. The following libraries are needed:

- OpenGL
- GLUT

Objects

Objects allows us to define the information of the 3d model, we can load a model from a .obj file using the OBJReader discussed earlier, or it can be defined in runtime, the class diagram is the following:



The object class is composed mainly by two other classes, Vertex and Face. The Vertex class stores the coordinates of the vertex that composes the model, it also has methods for doing operation with them, like obtain the magnitude, normalize and static methods like cross-product, UnitVector, etc. The Face class, define the connected vertices and in what order, but it is also in charge of storing the information about the normals of the object and calculating the illumination of the face.

The object class has attributes containing the list of vertices and the list of faces, but it also has other attributes like pivot, rotation and name, this helps us to have more control over the object. It also has implemented various manipulation methods:

- MoveTo: moves the object to a certain position
- Scale: scales the object to certain amount.
- Rotate: rotates the object some amount, in an axis.
- RotateTo: rotates the object to a certain degree in an axis.
- SetPivot: sets the pivot of the object at some position in space or to some vertex of the object

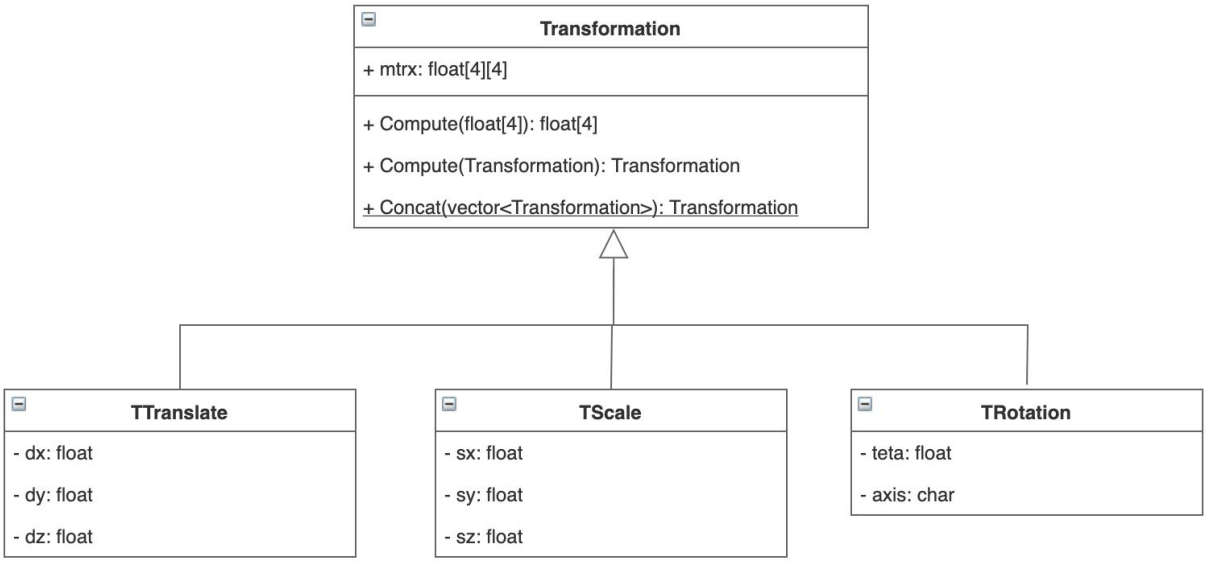
There are also methods for visualizing the object:

- CalculateNormals: recalculates all the normals of the faces of the object, this is used for when the object moves.
- CheckVisibility: check if the face is visible to the camera, for this it uses the PRP of the camera. This helps us to save resources, and that it looks realistic.
- DrawWireframe: draws the object only using wireframe
- Draw: this method draws the object using the illumination value from each face, this gives us a more realistic look of the objects.

Note: It is also important to take in consideration that we need to manually recalculate normals and visibility, if the object or the camera moves.

Transformations

The object class is an interface for manipulating object, but behind those function we are using geometrical transformation. For this we implemented matrix transformations, that allow us to do different operations, like: translate, rotate and scale. The way it is implemented is the following:



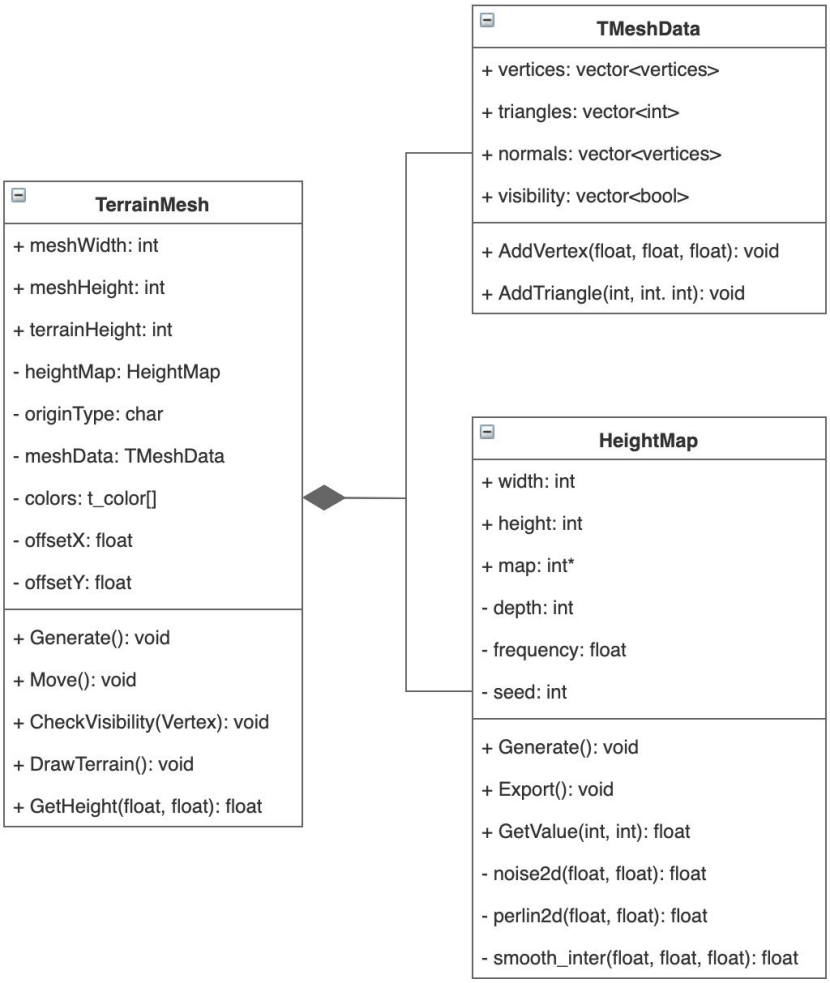
All transformation inherit from Transformation class, that consists of a 4x4 matrix, and the implementation to compute the operation. The values of the matrix depends on the transformation, for example when rotating the axis changes the position of the values inside the matrix.
The computation can be done between transformations, this allows us to concatenate them, just take in consideration that the order is correct. And at the same time we can compute it with homogeneous coordinates, that translates directly to the coordinates XYZ of the object.

This kind of implementation allows to easily extend and implement other transformations. Ej. sheer.

Terrain Mesh

We decided to use a different class to represent the terrain mesh information, instead of using the Object class. This is because the terrain its being moved constantly, for this we need to sample new points of perlin noise, but at the same time we reuse the earlier calculated values to save resources, to do this we need new methods and data structures to achieve this.

The class diagram is the following:



The TerrainMesh consists is composed of two main classes:

- HeightMap
- TMeshData (Terrain Mesh Data)

The HeightMap is the one that is generated using Perlin noise, it tells us the height of the terrain at some point, Perlin noise has various parameters such as frequency and depth, this class allow us to manipulate this values to obtain different types of terrains.

The TMeshData class, is a data structure that stores the vertices and triangles that conforms the terrain, from this data we can generate a wireframe model or a rendered model. We can also manipulate this data to make the terrain generate dynamically, this is why we don't use the object class to store the terrain mesh information.

With these two classes, we can generate and draw the terrain via the TerrainMesh class, and has methods to obtain the terrain height at some point and to move the terrain. The triangles are filled with black and we also draw wireframe, the color of the

wireframe depends on the height at that point, and we interpolate colors to have a gradient like effect. With this we get a really good looking terrain and simple to implement.

Utilities

There are also more classes that are useful but nothing out of this world, we call this utility classes, this includes:

- Timer: to get an accurate measure of elapsed time, to achieve smooth movement.
- BMP: this allows us to generate an image with the Bitmap format, we use this for visualizing the height map generation.
- t_color: this data structure serves to store color information, and also do operation like interpolation
- Basic math operations, like converting from radians to degrees and vice versa.

Problems Encountered

The most challenging part of the project was the terrain generation, there are various ways to implement it and some implementations allow for more complex mesh generation. This is why we followed the simpler implementation that only consists of frequency and depth, at first there were problems because I was getting some strange values, the best way to test if it was working was to generate a Bitmap with the height map values this was useful for debugging the Perlin noise.

As the terrain generation it is useful to create tests of different components of the code, there is also tests for terrain movement, bezier trajectory and model rendering.

One thing that never worked properly is the normals of the terrain, when joining the vertices of the mesh, we do it in triangles and in counter clockwise order and hoped that this was enough to calculate the normal correctly. But it wasn't, determining the visible surface never worked on the terrain, we compensated it by drawing the further first, and filling the triangles in black and the on top the wireframe. This causes that the nearby mountains to cover the others.

Final Comments

The project was fun and challenging to make, I believe that projects need to be challenging enough for them to be motivating to work on them. This was the case of this project, terrain generator can be a very complex software but looks very good when achieved. There are many possible improvements to the project, like implementing a more complex Perlin noise to achieve more natural looking mountains, or creating a collision system with the mountains.

References

“Definition of Bezier curves and its properties.”
http://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node12.html, 2009

Sebastian Lague, “Procedural Landmass Generation.”
https://www.youtube.com/watch?v=wbpMiKiSKm8, 2016

Flafia2, “Understanding Perlin Noise.” https://flafla2.github.io/2014/08/09/perlinnoise.html 2014

User Manual

OS Requirements:

Operating System: MacOS X or Ubuntu
Minimum RAM: 1 gb
Minimum free disk space: 1gb

Installation

The dependencies to be able to run the project are the following:

- C++ compiler
- OpenGL < 2.0
- GLUT

You can install does libraries with a package manager, or if on MacOS X are already preinstalled. Once installed those libraries download the project .zip file and decompress it.

Running

Once installed the dependencies and decompressed the .zip, open a new terminal window and cd to the project directory, once there run the following command to compile the code and generate the executable.

```
$ make
```

Then to execute the program run:

```
$./main
```

Finally, sit and enjoy the ride.