

# OpenGL Project

## Heightfield (Terrain) Generator

David Orlando de la Fuente Garza, A00817582

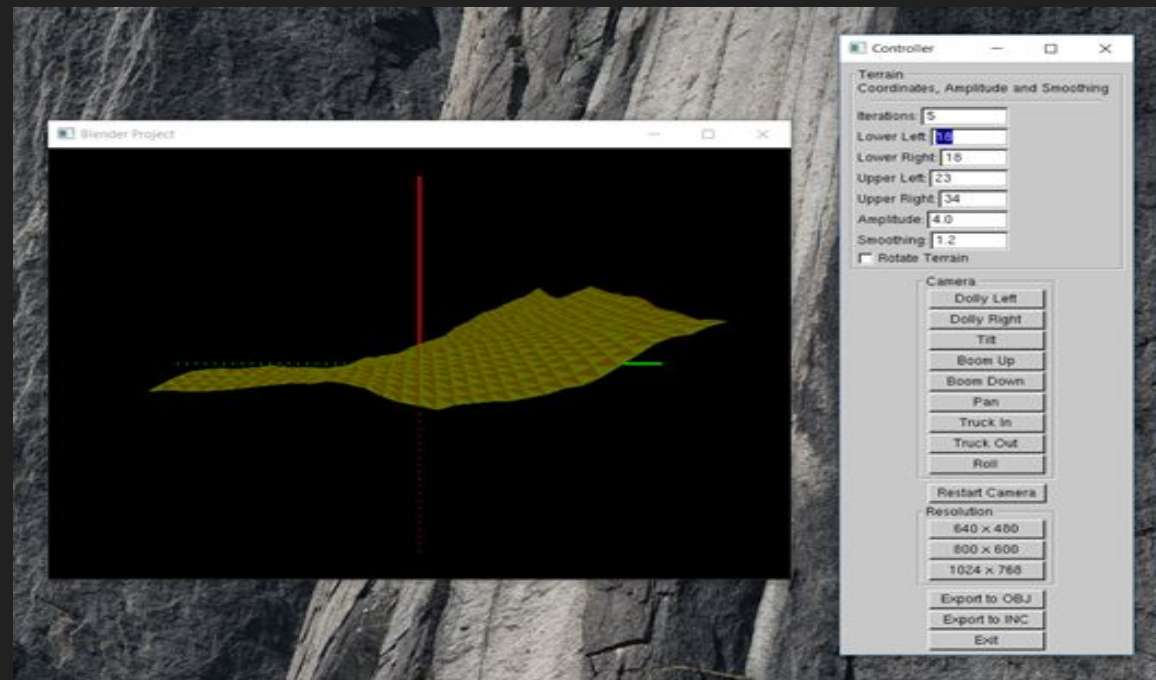
José María Flores Escalera - A01153458

Diego Adolfo José Villa - A00815260

# Introduction

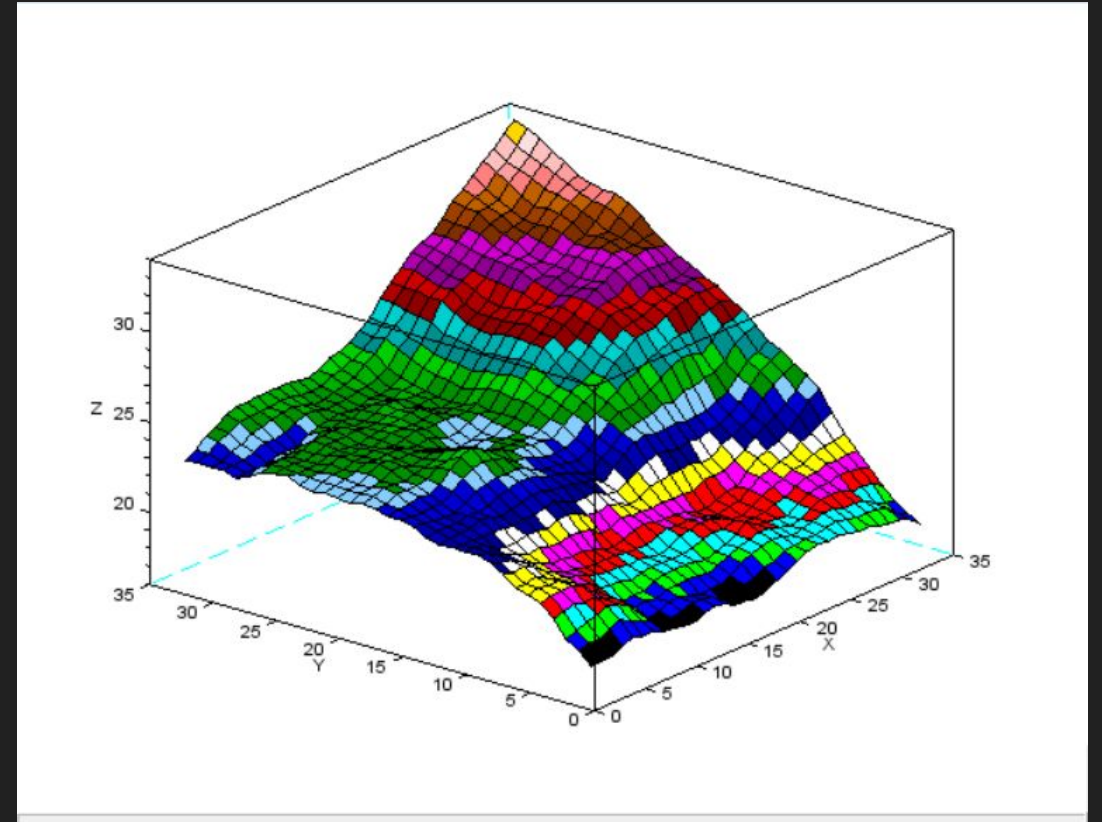
# Introduction: Our application

Our application is a **terrain generator**, which shows a mathematically generated terrain and **includes an UI** so the user can make modifications and create new surfaces.



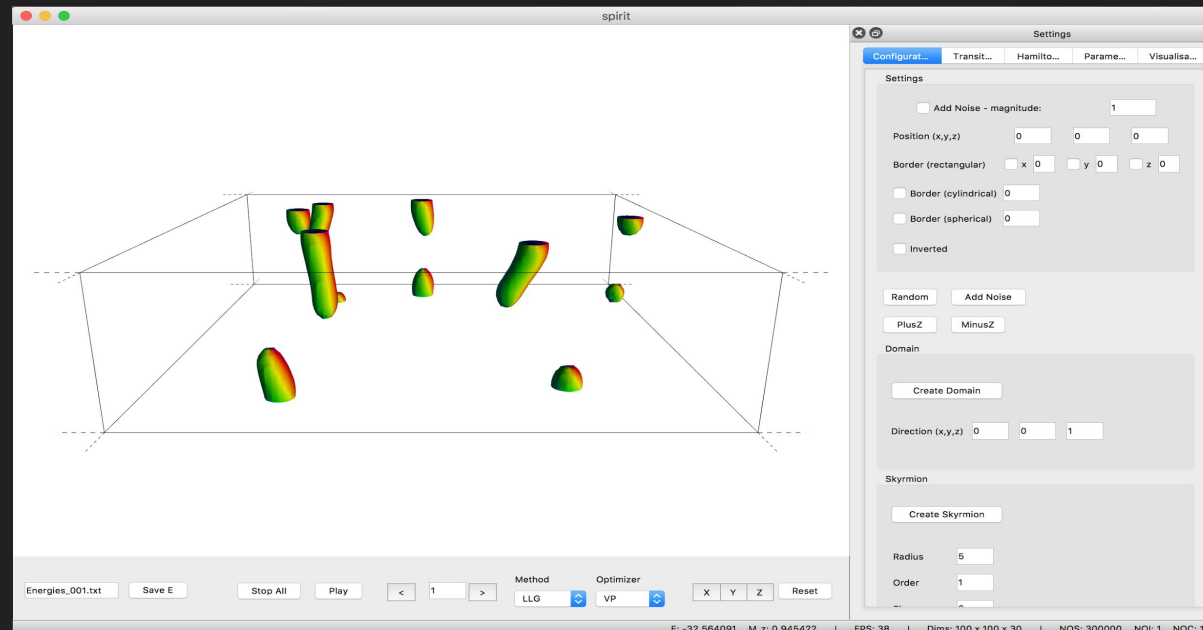
# Introduction: Background

The bases of this project were the knowledge we have in OpenGL and the **midpoint interpolation** method we learned previously on the semester.



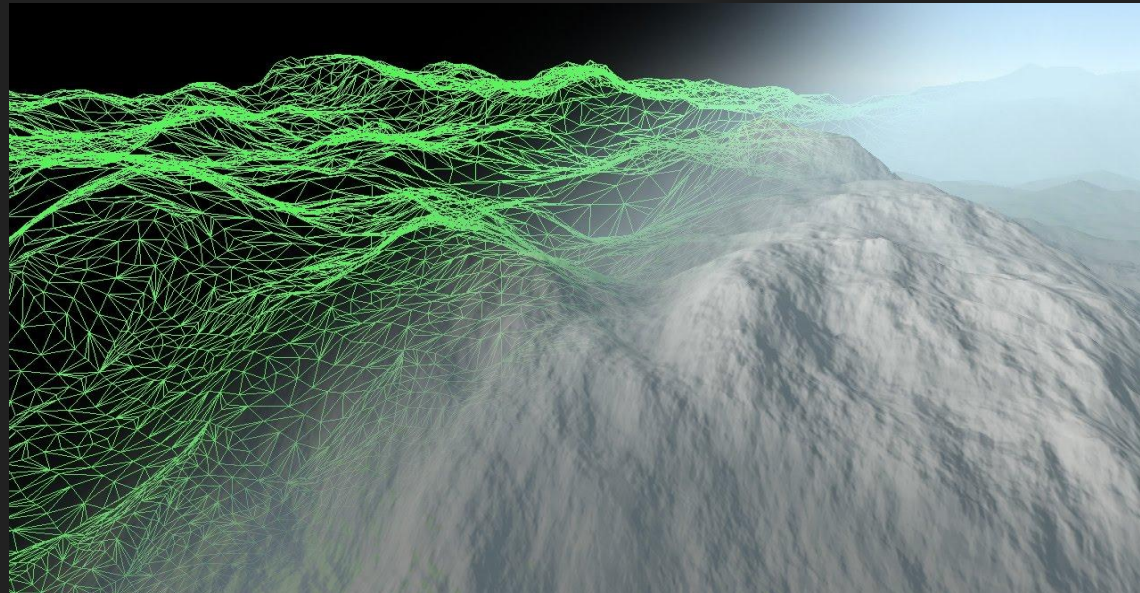
# Introduction: Design Style

We aimed to create an application similar to **scientific simulations**, which the model always being shown and with a menu for modifying its properties.



# Introduction: Inspiration

With terrain generation, **mountains** always serve as source of inspiration because of their mix of pattern and randomness. Translating mountains into graphics is an interesting task.



# Modelling

# Concept: Mathematical Model

In order to bring our application to life, our core mathematical method to generate terrain was the midpoint displacement algorithm, described as:

- At its basis, we have a line created between two points.
- The midpoint of this line is then displaced by some random amount in a vertical direction.
- The midpoints of these two new line segments are then displaced by a random amount in the vertical direction.
- We repeat until you have reached desired level of detail.



# Concept: Mathematical Model

There are a few things to take into account in order to implement this method such as:

- The height map must be a square with sides of length  $2^N + 1$ , so that each sub-square has an exact middle point.
- We calculate a random variable  $r$  that goes between -1 and 1 for the vertical displacement.
- We use an amplitude  $a$  and a smoothing  $h$  parameters to make the result of our calculation more realistic to a normal terrain.
- We also have variables  $k$  and  $k'$  that are elements from  $\{1, 3, \dots, 2^{(N-n)} - 1\}$  and  $m$  is an element from  $\{0, 2, \dots, 2^{(N-n)}\}$  where  $n < N$ .

# Concept: Mathematical Model

With a height map of:

$$T = \begin{bmatrix} T_{0,0} & \dots & T_{0,2^N} \\ \vdots & \ddots & \vdots \\ T_{2^N,0} & \dots & T_{2^N,2^N} \end{bmatrix}$$

We calculate the values for columns as:

$$T_{k \cdot 2^n, j \cdot 2^n} = \frac{T_{(k+1) \cdot 2^n, j \cdot 2^n} + T_{(k-1) \cdot 2^n, j \cdot 2^n}}{2} + A \cdot R \cdot 2^{-H \cdot n}$$

Values for rows as:

$$T_{j \cdot 2^n, k \cdot 2^n} = \frac{T_{j \cdot 2^n, (k+1) \cdot 2^n} + T_{j \cdot 2^n, (k-1) \cdot 2^n}}{2} + A \cdot R \cdot 2^{-H \cdot n}$$

And middle values as:

$$T_{k \cdot 2^n, k' \cdot 2^n} = \left(\frac{1}{4}\right) T_{(k-1) \cdot 2^n, (k'-1) \cdot 2^n} + \left(\frac{1}{4}\right) T_{(k-1) \cdot 2^n, (k'+1) \cdot 2^n} + \left(\frac{1}{4}\right) T_{(k+1) \cdot 2^n, (k'-1) \cdot 2^n} + \left(\frac{1}{4}\right) T_{(k+1) \cdot 2^n, (k'+1) \cdot 2^n} + A \cdot R \cdot 2^{-H \cdot n}$$

# Concept: Example of Model

The following example creates a matrix representation of the terrain. For simplicity this example leaves out the random factor of point generation.

- We start with  $n$  as a value of 1 so we get a simple square of size  $2^N + 1 = 3$  with four starting values on each corner:

|   |  |    |
|---|--|----|
| 3 |  | 10 |
|   |  |    |
| 5 |  | 2  |

# Concept: Example of Model

- We calculate the values for both rows and columns the same way described before as it's the middle point between the values:

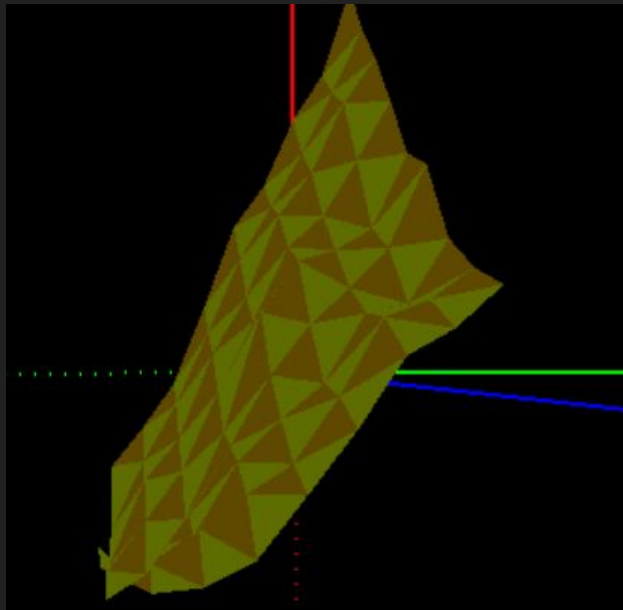
|   |     |    |
|---|-----|----|
| 3 | 6.5 | 10 |
| 4 |     | 6  |
| 5 | 3.5 | 2  |

- We calculate the middle value as an average of the other 4 calculated values as:

|   |     |    |
|---|-----|----|
| 3 | 6.5 | 10 |
| 4 | 5   | 6  |
| 5 | 3.5 | 2  |

# Concept: Example of Model

- This gives us our final result. For bigger N values and squares, we use the divide and conquer paradigm to divide the total square in smaller ones using the same concept just reviewed.



```
18 17.2548 17.0413 16.6327 15.3838 15.7487 15.5456 16.2009 18
17.0995 17.3087 17.2725 16.9285 15.9875 17.062 17.9376 18.3391 19.2399
17.1916 18.0076 17.5331 17.7383 17.3729 18.4048 19.1438 20.0735 20.6575
17.6687 18.6877 19.1811 19.3821 19.5867 20.55 21.6245 22.2817 23.1641
18.8895 19.3137 20.1335 21.1015 22.7158 22.522 23.3055 24.5854 25.8248
20.143 20.7145 20.9456 22.35 24.0856 24.6163 25.7655 26.2427 27.1978
21.5833 22.4025 22.567 23.2478 24.3658 25.5151 27.0487 27.9689 29.4508
22.125 22.7853 22.7807 24.2666 24.9552 27.1151 29.1966 29.9806 30.9881
23 23.4877 24.3162 26.403 27.0076 28.8624 30.6169 32.0613 34
```

# OpenGL: Mountain model

Given a matrix of heights we built the model of the mountain using the primitive

***GL\_TRIANGLE\_STRIP***.

For every square of the matrix we display two primitives.

```
for(int i=0; i < terrain.size() - 1; i++)
{
    for(int j=0; j < terrain.size() - 1; j++)
    {
        glBegin(GL_TRIANGLE_STRIP);
        glColor3f(float(95.0/255), float(108.0/255), 0.0f);
        glVertex3f(i+1, j, terrain[i+1][j]);
        glVertex3f(i, j+1, terrain[i][j+1]);
        glVertex3f(i, j, terrain[i][j]);
        glEnd();

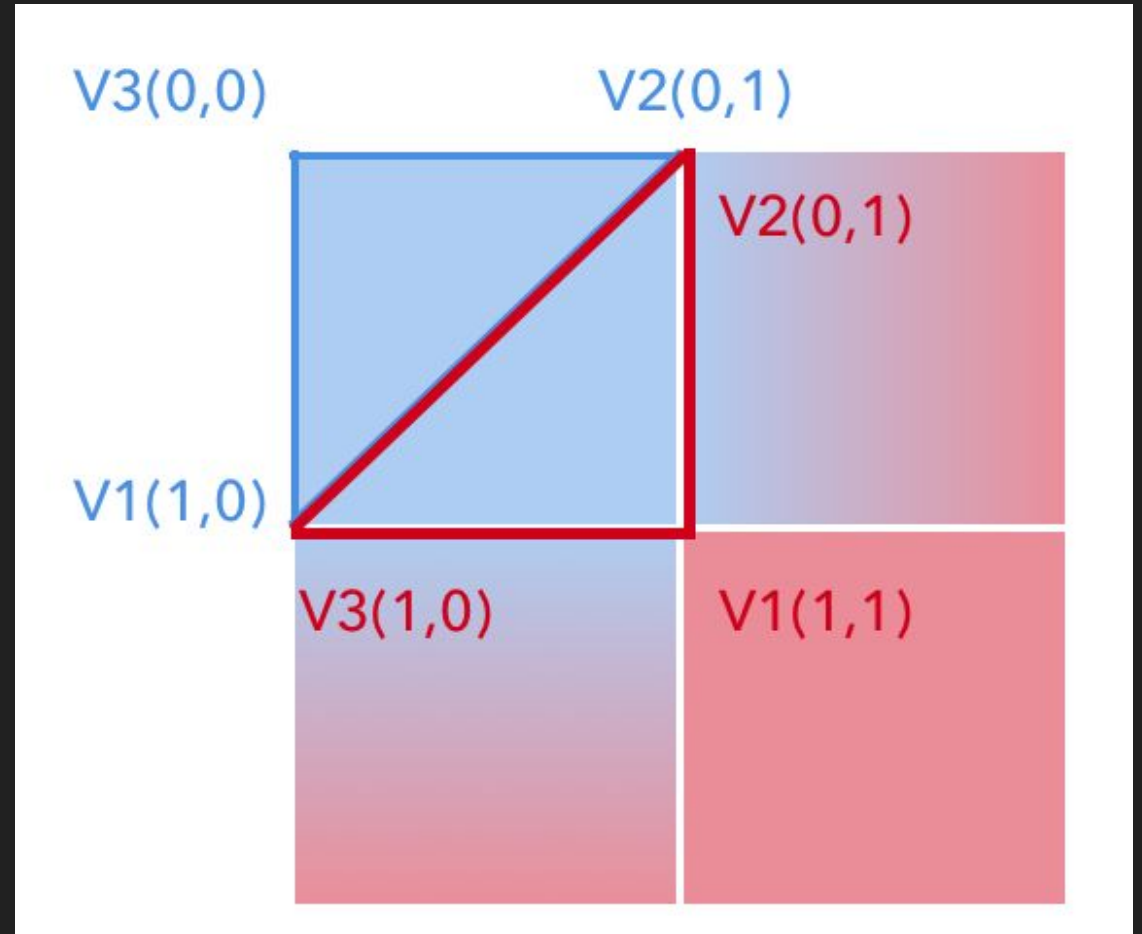
        glBegin(GL_TRIANGLE_STRIP);
        glColor3f(float(95.0/255), float(72.0/255), 0.0f);
        glVertex3f(i+1, j+1, terrain[i+1][j+1]);
        glVertex3f(i, j+1, terrain[i][j+1]);
        glVertex3f(i+1, j, terrain[i+1][j]);
        glEnd();
    }
}
```

# OpenGL: Mountain model

We first draw the blue triangle and then the red triangle.

The order of the vertex is as shown in the image.

We used the X, Y position of the vertex to get a height, this value corresponds to the Z value of each vertex.



# Fractals: Modifying the terrain

After generating the midpoint displacement terrain matrix and displaying it on screen with primitives, we can modify values for the next calculations:

- $N$ , also known as the value that gives the size of the matrix and the number of iterations of the midpoint displacement algorithm across the matrix.



# Fractals: Modifying terrain (Cont.)

The following parameters were also are calculated based on modifications:

- The lower left coordinate of the matrix.
- The lower right coordinate of the matrix.
- The upper left coordinate of the matrix.
- The upper right coordinate of the matrix.
- The amplitude value of the terrain.
- The smoothing value of the terrain.

# Fractals: Fractal Manipulation

We also implemented a menu at the beginning of the application for the user to provide initial values (not necessarily in the corners).

```
Quieres ingresar puntos para generar el terreno? (1 para Si, 2 para No)
1
Ingresa el numero de iteraciones para el terreno: 5

Ingresa la cantidad de puntos iniciales: 3

A continuacion se pediran los valores de los puntos iniciales
Los valores X y Y de cada punto deben ser enteros, el valor Z puede ser flotante

Punto #1
Ingresa los valores X Y Z del punto: 13 10 8
Punto #2
Ingresa los valores X Y Z del punto: 9 12 3
Punto #3
Ingresa los valores X Y Z del punto: 1 3 6
Se ha generado el terreno con los valores dados
```

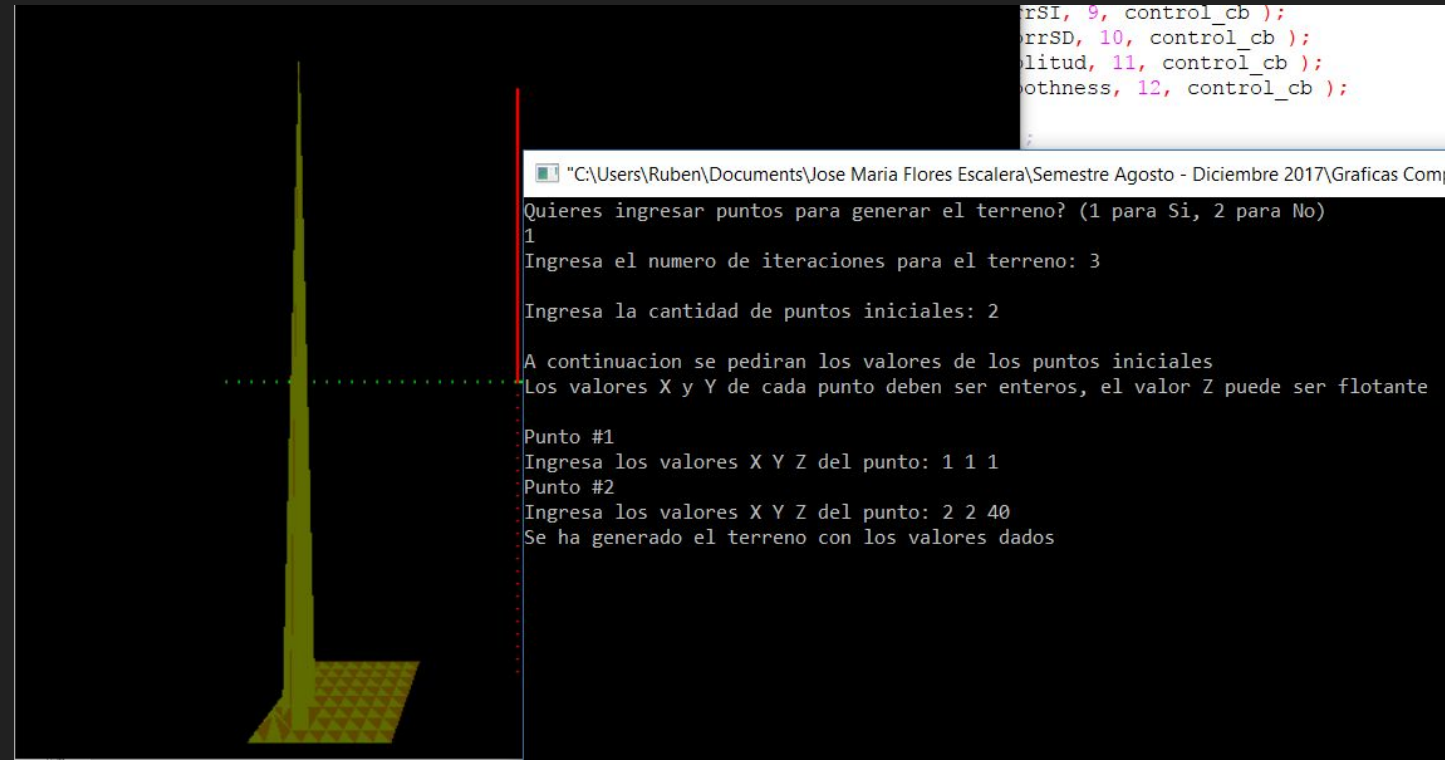
# Fractals: Fractal Manipulation

In the case the user gives initial values, those values are setted in a matrix full of 0's (which will be the new terrain).

During terrain generation, we check if a point is different than 0. If its different its mean that it was previously set by the user and there is no need to recalculate.

# Fractals: Fractal Manipulation

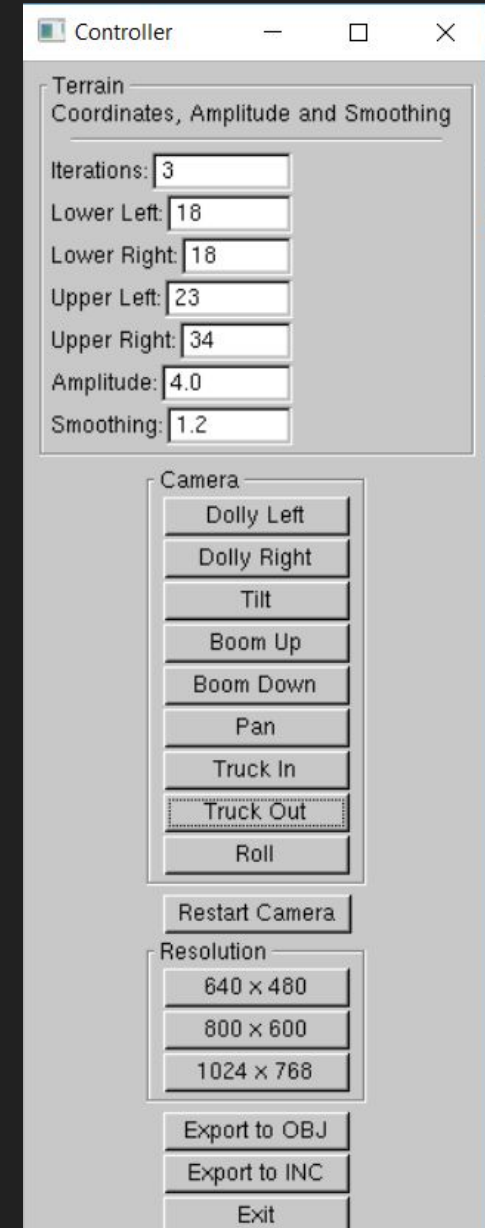
This allows the user to modify how the final terrain will look, like the following:



# UI Navigation

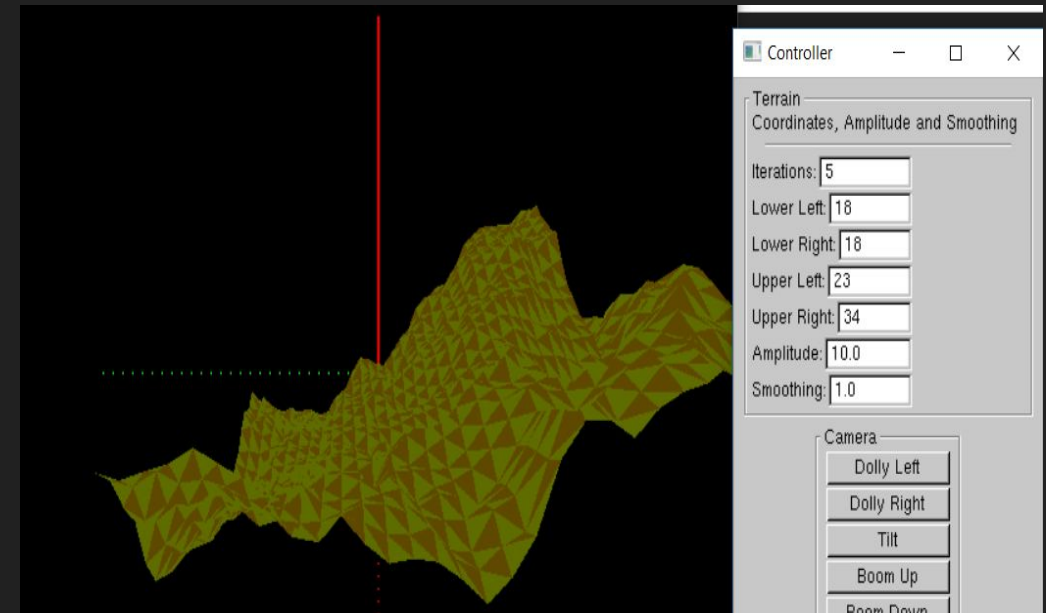
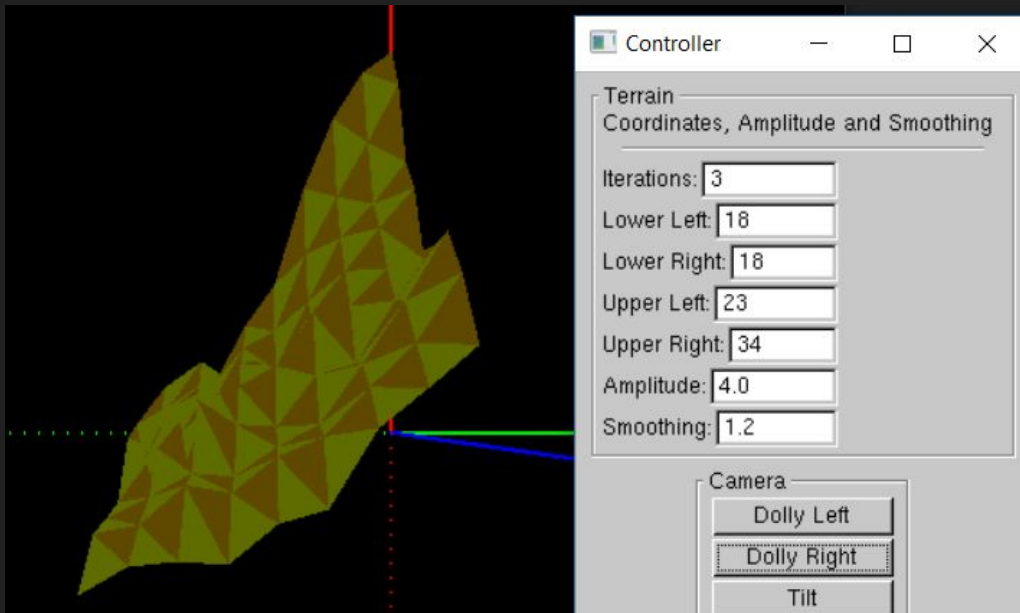
The menu is composed of textboxes and buttons which give the user the following options:

- Modify number of iterations, values of corners, amplitude, etc.
- Move camera and zoom in/out.
- Export to .obj and .inc format



# UI Navigation

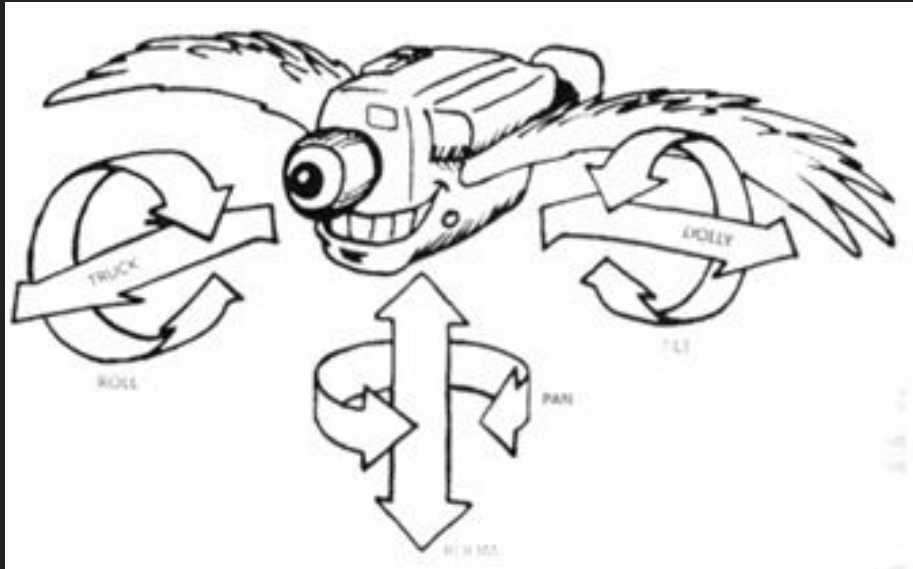
We can modify parameters and the new terrain will be generated right away. Increasing the number of iteration will increase the number of polygons that form the terrain.



# Coloring, illumination and cameras

# Camera

Our application uses only one camera to show our whole scene. This camera is capable of moving on the 9 directions learned on class.





# Lights

No lighting sources were used within the scene as it was determined that they weren't needed since the visualization didn't benefit from them. This was discussed with the teacher and we reached a common agreement on it.

# Camera placement

The camera is placed just over the x and y axis drawn on the scene with green and red colours so it could see the whole generated terrain if its size is between 3 and 65.

If the size is over those values, we would need to move the camera in order to visualize it completely.

# Camera placement

The camera can be moved with both keyboard and UI buttons. If we want to move it with keyboard keys, we use:

- **Dolly right/left:**  $\leq$ ,  $\geq$
- **Bottom Up/Down:**  $\wedge$ ,  $\vee$
- **Truck in/out:** Page up, Page down
- **Camera restart:** "O"



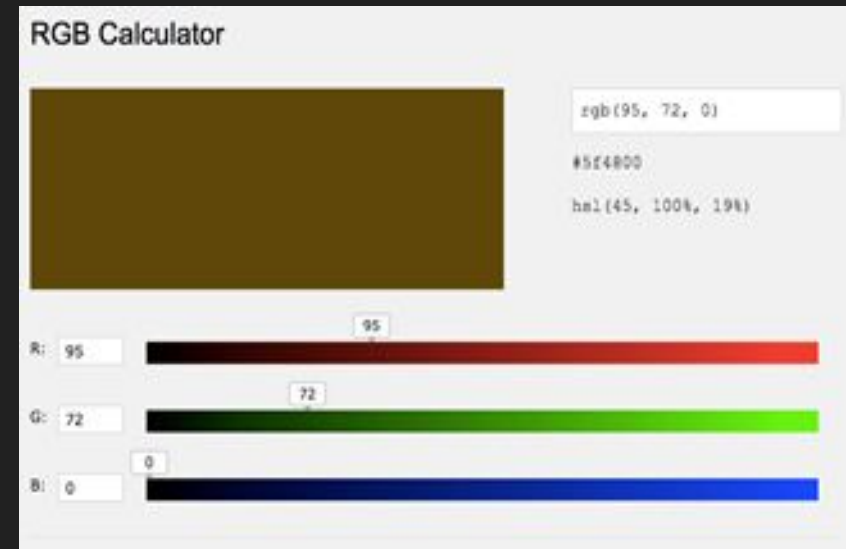
# Colour Pallet

We decided to use a combination of green and brown tones in order to give our terrain similar looks to those of real mountains.



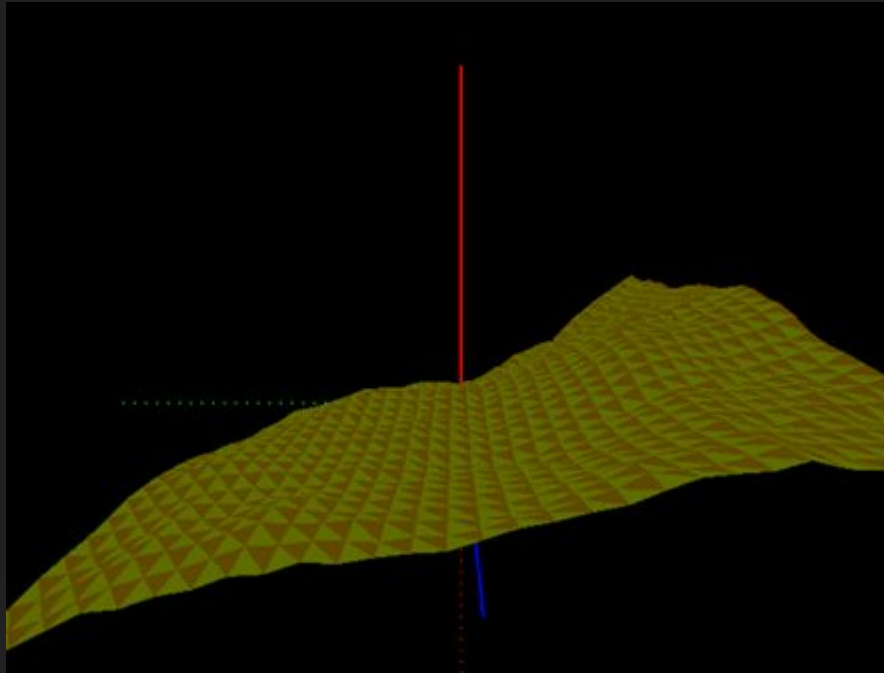
# Colouring

We used a basic two colour scheme to represent our terrain so we could see the depth within the scene and the vertical displacement of each value within the heightmap:



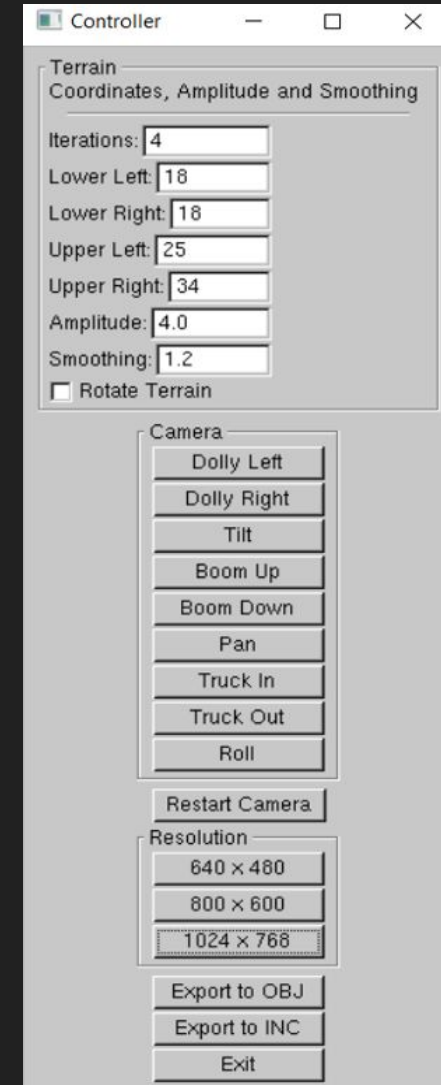
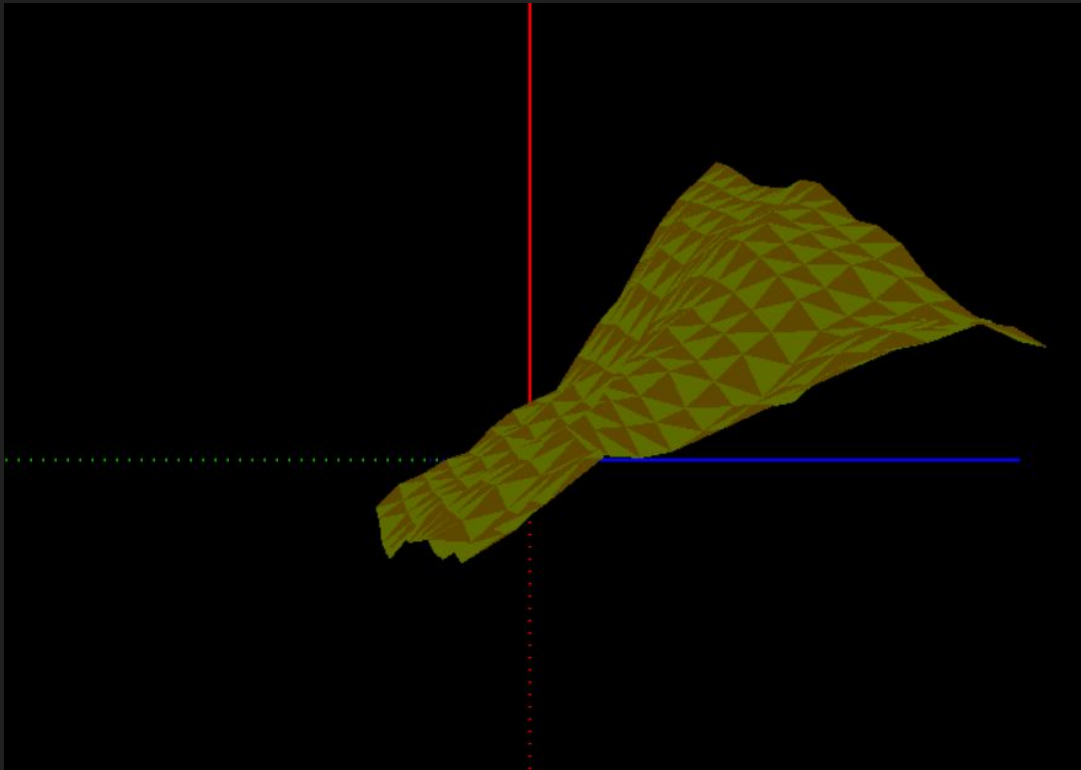
# Results of colouring

After painting polygons with the use of both colors in a patterned way, the following look was achieved:



# Rendering

# How we constructed our application





# How we constructed our application

- To build the menu we use the library **Glui32**.
- To build the mountain model we use the library **FreeGlut**.
- With the menu we can:
  - Change the number of polygons of the mountain.
  - Give initial values for the corners of the matrix.
  - Change the value for the amplitude and smoothing.
  - Move the camera and change the dimension of the window.
  - Export the mountain model to a .obj (Blender) and .inc (POV-Ray).
- On the command line we can:
  - Enter values for specific points of the matrix.

# What basic OpenGL objects were used

We used:

- ***GL\_TRIANGLE\_STRIP*** to build the model of the mountain.
- ***GL\_LINES*** to build the solid axes.
- ***GL\_LINE\_STIPPLE*** to build the stipple axes.

Each component is constructed with only one type of primitive figure.

```
glBegin(GL_TRIANGLE_STRIP);
glColor3f(float(95.0/255), float(108.0/255), 0.0f);
glVertex3f(i+1, j, terrain[i+1][j]);
glVertex3f(i, j+1, terrain[i][j+1]);
glVertex3f(i, j, terrain[i][j]);
glEnd();
```

```
glBegin(GL_LINES);
glColor3f (0.0, 1.0, 0.0); // Verde para el eje X
glVertex3f(0,0,0);
glVertex3f(10,0,0);
```

```
// Lineas punteadas para conocer los lados del mapa de ejes
glEnable(GL_LINE_STIPPLE);
glLineStipple(1, 0x0101); // Patron de lineas punteadas
glBegin(GL_LINES);
glColor3f (0.0, 1.0, 0.0); // Verde para el eje X
glVertex3f(-10,0,0);
glVertex3f(0,0,0);
```

# Storage

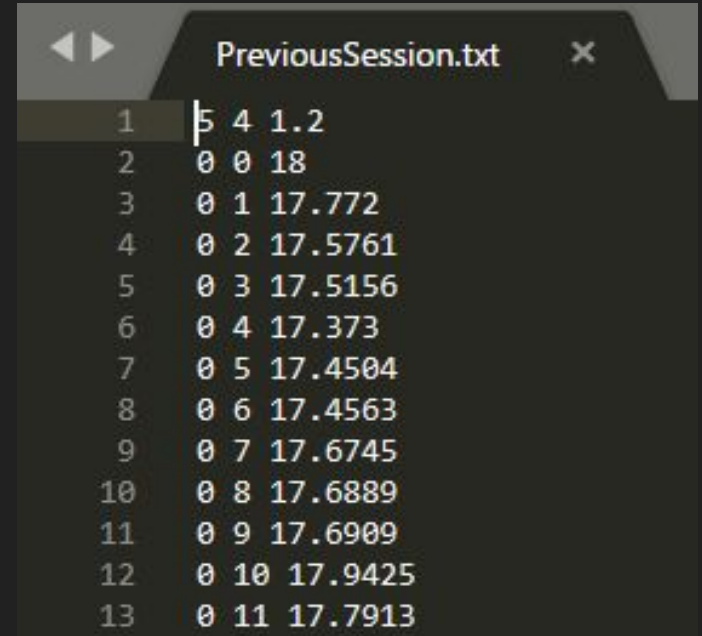
# Storage: Restore session in OpenGL

We stored the location of the vertices of our terrain using the following format:

**[Iterations] [Amplitude] [Smoothness]**

Followed by the list of coordinates:

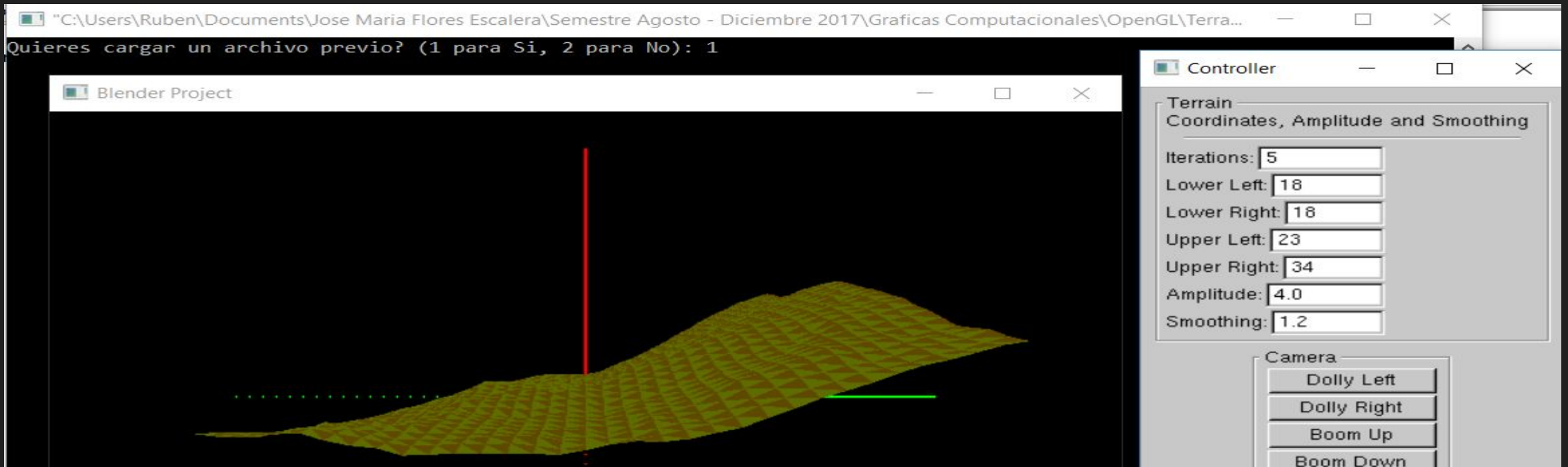
**[X Position] [Y Position] [Z Position]**



```
1 4 1.2
2 0 0 18
3 0 1 17.772
4 0 2 17.5761
5 0 3 17.5156
6 0 4 17.373
7 0 5 17.4504
8 0 6 17.4563
9 0 7 17.6745
10 0 8 17.6889
11 0 9 17.6909
12 0 10 17.9425
13 0 11 17.7913
```

# Storage: Restore session in OpenGL

This allowed us to restore session in OpenGL with an option to load the file at the beginning of execution”

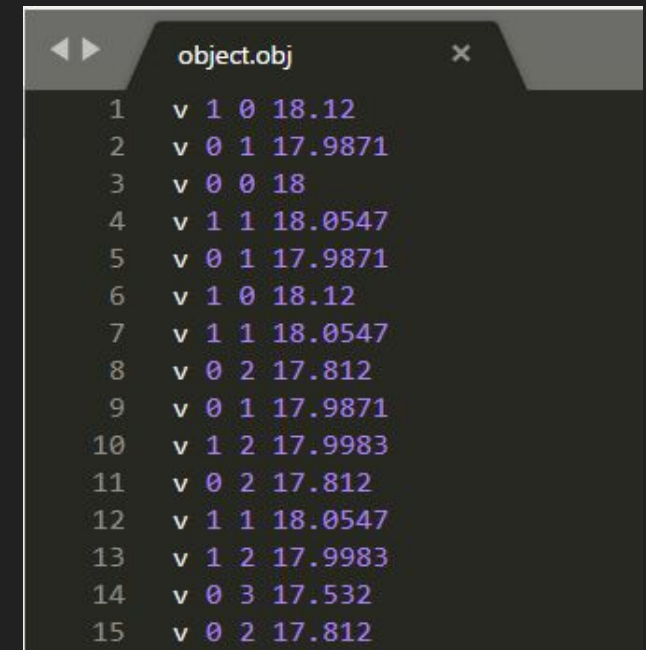


# Storage: OBJ Format (Vertices)

For storage, we used an **Wavefront OBJ** format so it could be properly imported into **Blender** and **POV Ray**.

We stored the location of the vertices of our terrain using the following format:

**V** [X Position] [Y Position] [Z Position]

A screenshot of a text editor window titled 'object.obj'. The window displays a list of 15 vertices in OBJ format. Each line consists of a line number, the letter 'v', and three floating-point numbers representing the X, Y, and Z coordinates. The coordinates are: 1: 1 0 18.12, 2: 0 1 17.9871, 3: 0 0 18, 4: 1 1 18.0547, 5: 0 1 17.9871, 6: 1 0 18.12, 7: 1 1 18.0547, 8: 0 2 17.812, 9: 0 1 17.9871, 10: 1 2 17.9983, 11: 0 2 17.812, 12: 1 1 18.0547, 13: 1 2 17.9983, 14: 0 3 17.532, 15: 0 2 17.812.

```
1 v 1 0 18.12
2 v 0 1 17.9871
3 v 0 0 18
4 v 1 1 18.0547
5 v 0 1 17.9871
6 v 1 0 18.12
7 v 1 1 18.0547
8 v 0 2 17.812
9 v 0 1 17.9871
10 v 1 2 17.9983
11 v 0 2 17.812
12 v 1 1 18.0547
13 v 1 2 17.9983
14 v 0 3 17.532
15 v 0 2 17.812
```

# Storage: Code for storing vertices

Such format was achieved by printing the values into the .obj in this way:

```
void createObject() {
    ofstream outfile;
    ofstream outfile2;
    outfile.open("object.obj");

    int k = 0;

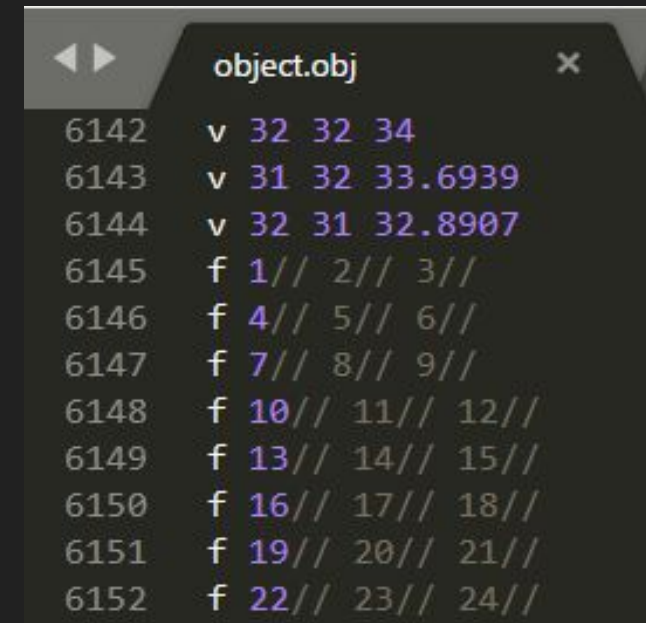
    for(int i=0; i < globalTerrain.size() - 1; i++)
    {
        for(int j=0; j < globalTerrain.size() - 1; j++)
        {
            outfile << "v " << float(i + 1) << " " << float(j) << " " << globalTerrain[i + 1][j] << endl;
            outfile << "v " << float(i) << " " << float(j + 1) << " " << globalTerrain[i][j + 1] << endl;
            outfile << "v " << float(i) << " " << float(j) << " " << globalTerrain[i][j] << endl;

            outfile << "v " << float(i + 1) << " " << float(j + 1) << " " << globalTerrain[i + 1][j + 1] << endl;
            outfile << "v " << float(i) << " " << float(j + 1) << " " << globalTerrain[i][j + 1] << endl;
            outfile << "v " << float(i + 1) << " " << float(j) << " " << globalTerrain[i + 1][j] << endl;
        }
    }
}
```

# Storage: OBJ Format (Faces)

Once we had the coordinates of vertices, then we could **create faces using those vertices**.

These faces would map to the vertices defined on the first part of the .obj file, using **3 vertices** to create a **face**.



A screenshot of a text editor window titled "object.obj" showing a list of vertex and face definitions. The window has a dark background and a light-colored title bar. The content is as follows:

| Line Number | Command | Values         |
|-------------|---------|----------------|
| 6142        | v       | 32 32 34       |
| 6143        | v       | 31 32 33.6939  |
| 6144        | v       | 32 31 32.8907  |
| 6145        | f       | 1// 2// 3//    |
| 6146        | f       | 4// 5// 6//    |
| 6147        | f       | 7// 8// 9//    |
| 6148        | f       | 10// 11// 12// |
| 6149        | f       | 13// 14// 15// |
| 6150        | f       | 16// 17// 18// |
| 6151        | f       | 19// 20// 21// |
| 6152        | f       | 22// 23// 24// |



# Storage: Code for storing faces

Such format was achieved by printing the number of the face in the following way:

```
outfile << "f";

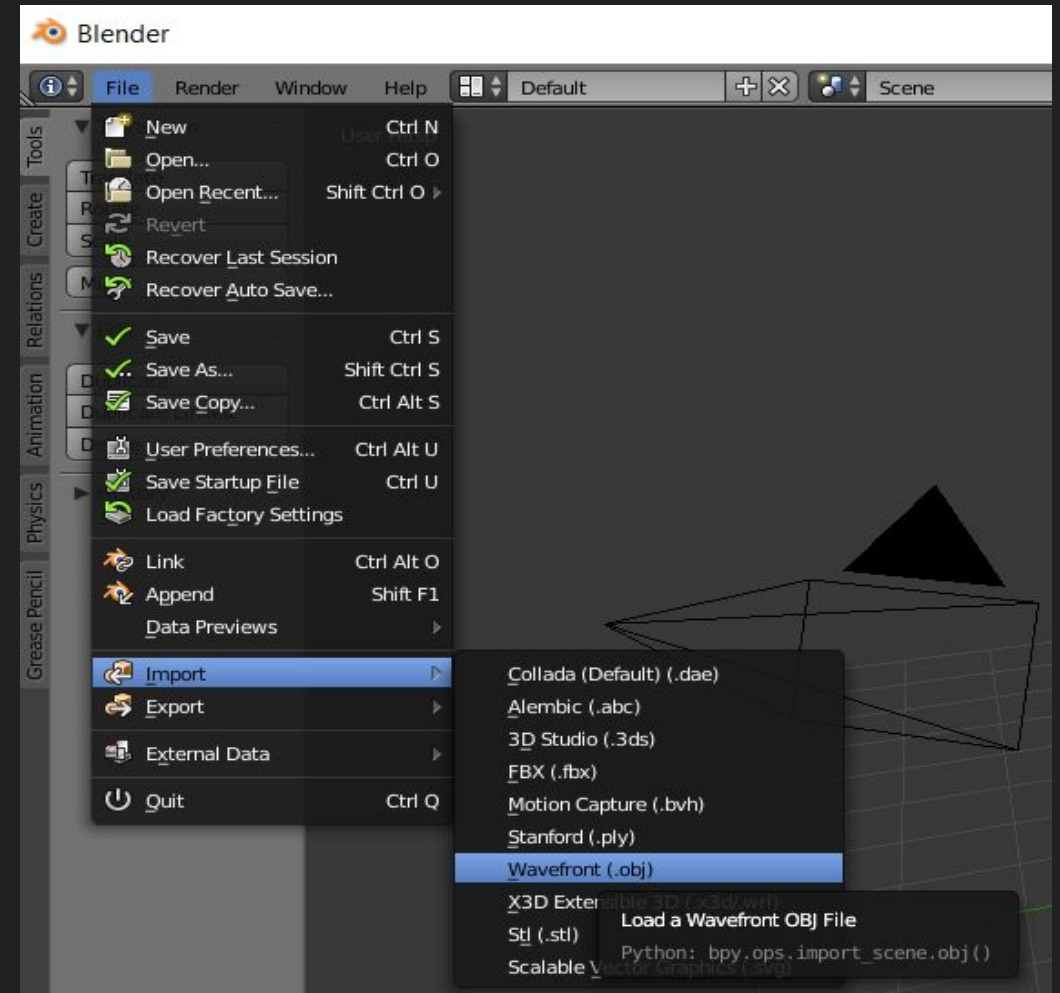
int j = 1;
for(int i = 1; i <= k; i++)
{
    if(i % 3 == 0)
    {
        outfile << " " << i << "/" << endl;

        if (i < k)
        {
            outfile << "f";
        }
    }
    else
    {
        outfile << " " << i << "/";
    }
}
```

# Storage: Blender Importing

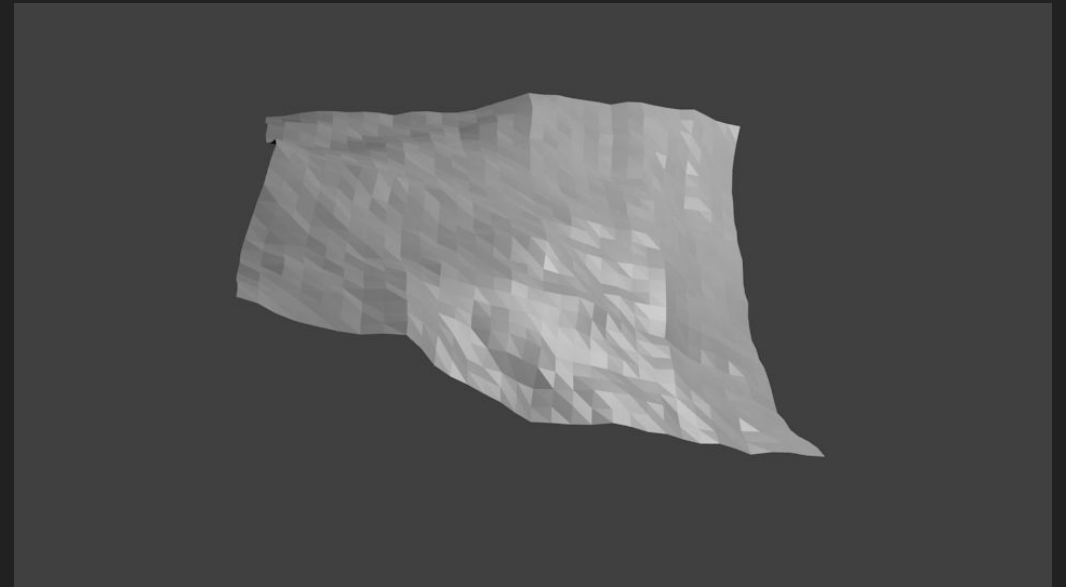
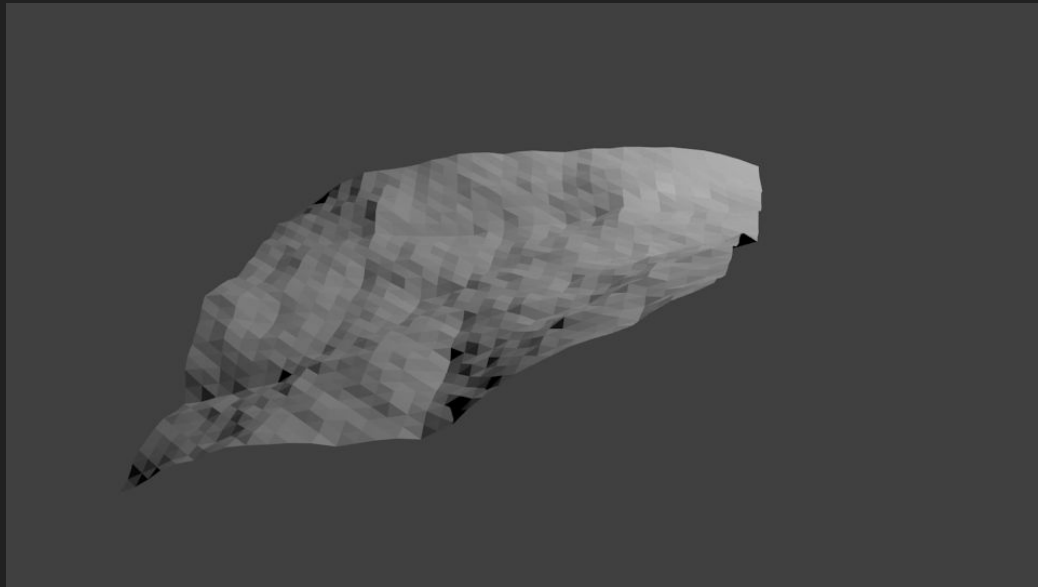
For importing the model into Blender, it was as simple as using **Blender Importing option**.

This would allow us to directly use the previously created **.obj file**.



# Storage: Blender Importing

The following results were obtained after importing to **Blender**:



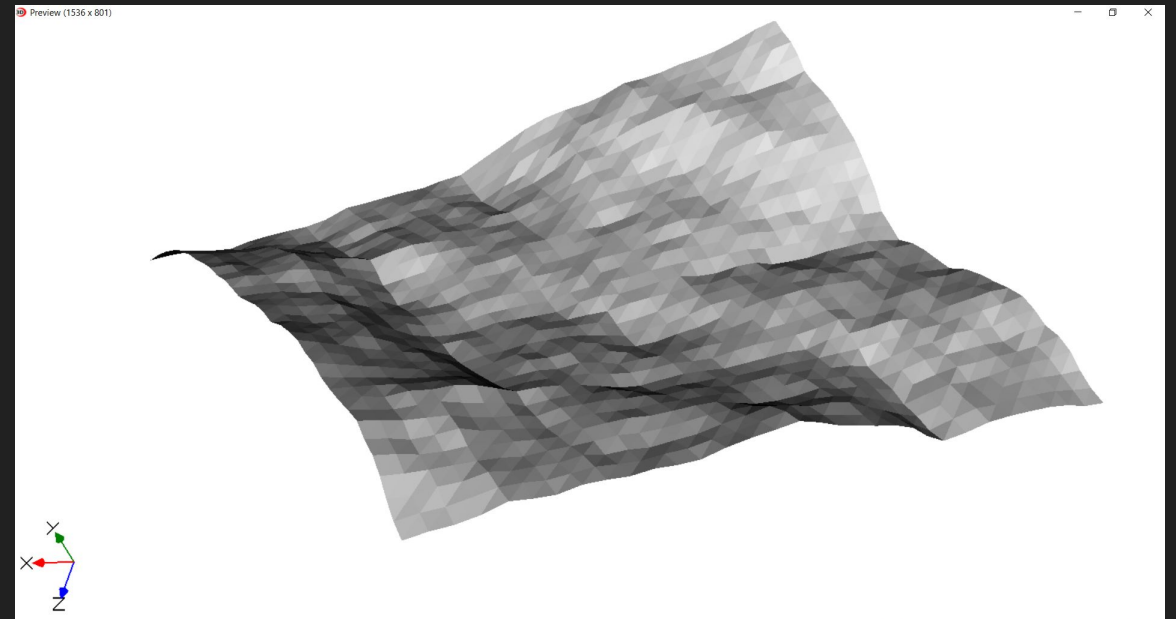
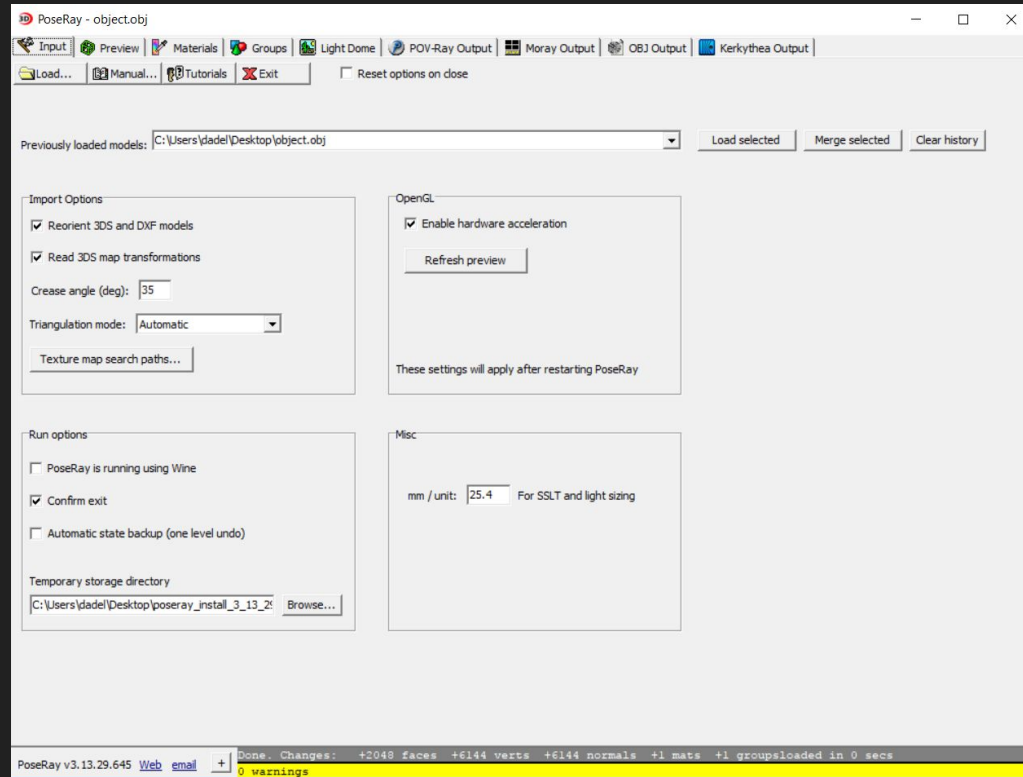
# Storage: POV-Ray Importing

For importing into POV Ray, we used two options:

- **Using** the utility **Poseray**, which converts .obj files into POV Ray Scenes
- **Creating** an **.inc file** which would contain objects in POV Ray created using the information of the **.obj file**, and could be **imported as a library**.

# Storage: POV-Ray Importing

**PoseRay** allowed us to create the following model:



# Storage: POV-Ray Importing

In order to create the **.inc file**, we had to modify our code:

```
void createObject() {
    ofstream blender;
    ofstream povray;
    blender.open("object.obj");
    povray.open("object.inc");

    povray << "#declare Mountain = mesh {" << endl;

    int k = 0;
    for(int i=0; i < globalTerrain.size() - 1; i++)
    {
        for(int j=0; j < globalTerrain.size() - 1; j++)
        {
            povray << "triangle {";
            povray << "<" << float(i+1) << ", " << float(j) << ", " << globalTerrain[i+1][j] << ">";
            povray << "<" << float(i) << ", " << float(j+1) << ", " << globalTerrain[i][j+1] << ">";
            povray << "<" << float(i) << ", " << float(j) << ", " << globalTerrain[i][j] << ">";
            povray << "}" << endl;

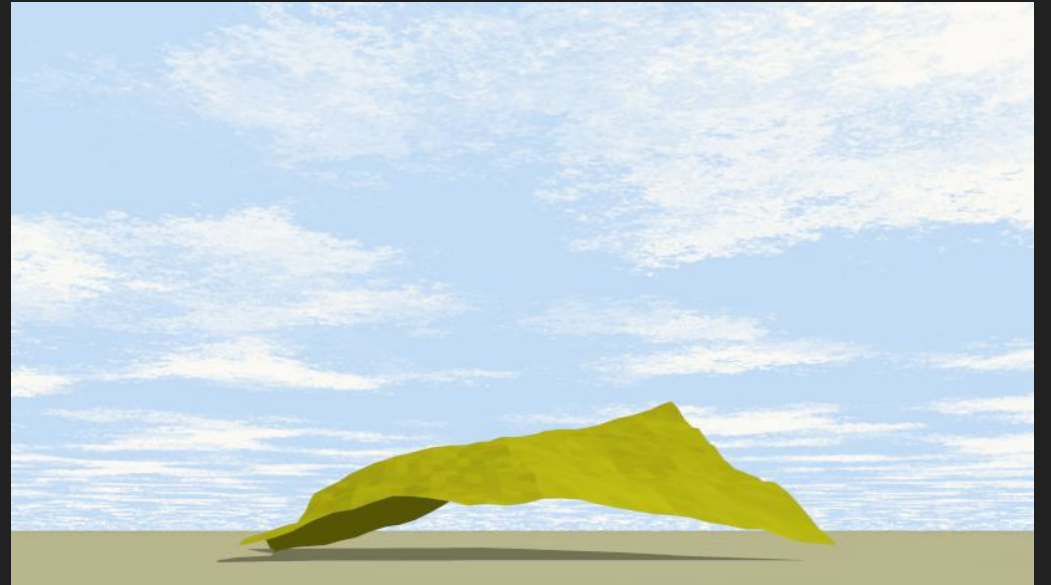
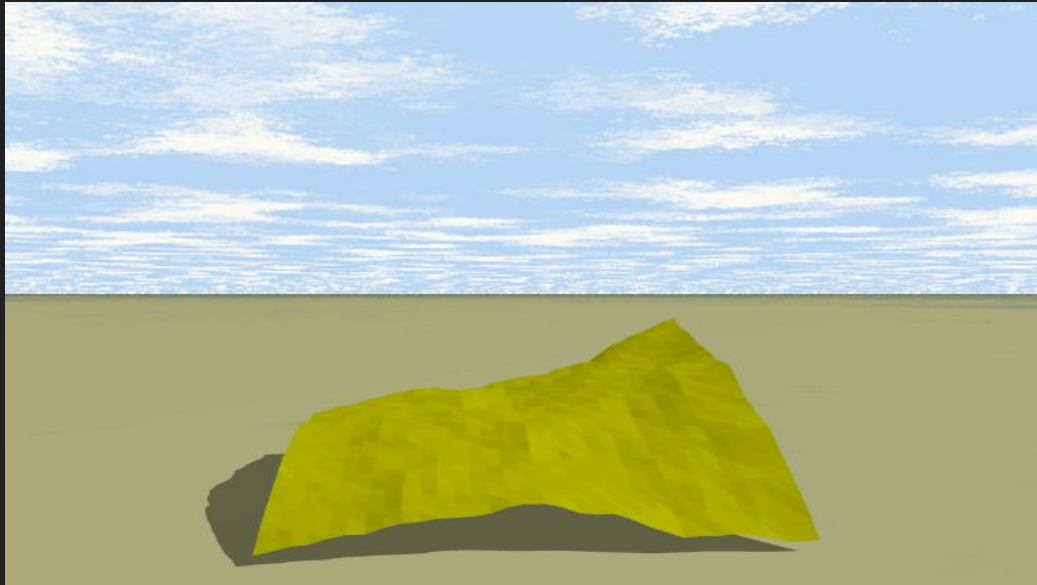
            povray << "triangle {";
            povray << "<" << float(i+1) << ", " << float(j+1) << ", " << globalTerrain[i+1][j+1] << ">";
            povray << "<" << float(i) << ", " << float(j+1) << ", " << globalTerrain[i][j+1] << ">";
            povray << "<" << float(i+1) << ", " << float(j) << ", " << globalTerrain[i+1][j] << ">";
            povray << "}" << endl;

            k += 6;
        }
    }

    povray << "}" << endl;
    povray.close();
}
```

# Storage: POV-Ray Importing

The following **results** were obtained after using the generated **.inc** file:



# Conclusions: Challenges

Some of the challenges we faced while making the project were:

- **Translating Scilab code into C++ code:** Scilab does not require type definition, variables are dynamically created, indexing starts at 1.
- **Integrating UI:** Correctly setting up library references of **glui32** and correctly linking buttons' actions and text fields to variables.
- **Creating .obj file:** We had troubles finding a way to represent our model, however, having all vertices already stored in a vector did help.



# Conclusions: Technical Lessons

With the realization of this project, we came to a better understanding of aspects such as:

- **How to manage changes in looping environments:** We had to use **global variables and booleans to coordinate changes** to what the program should display, so it didn't show a different terrain every couple of frames.
- **UI is not trivial:** Integrating UI in code was harder than we thought, and surely is not something to be taken lightly.

# Conclusions: Project thoughts

We as a team also had a better appreciation of some more general aspects:

- **Graphics have potential:** Computer Graphics are a way to transmit ideas in a variety of unique ways, and code representation may be harder but it give the creator more freedom.
- **Storage does matter:** We now are more aware of the text representation behind complex objects. Being able to import to other programs is always an excellent option and something to consider.

# What we will always remember

*Computer graphics allow us to share ideas in ways we didn't even thought were possible. The only limitations to what you may create are your imagination, and your graphics card.*

- Team Q

Thank you.