**Table of Content**

**TASK 1: Generating Height Maps**

1. Introduction to MuJoCo
2. Procedural Generation and Perlin Noise
3. Key Functions Used for Terrain Generation
4. Key Concepts
5. Visualizing the Terrain
6. Results Without Perlin Noise
7. Perlin Noise and Its Characteristics
8. Results With Perlin Noise
9. Larger Heat Maps
10. Code for Generating Multiple Terrains

**TASK 2: Creating Objects Using Blender**

1. Introduction to Blender
2. Key Features of Blender
3. Basic Transformations in Blender
4. Object Creation:
   * Apple
   * Acorn
   * Chestnut
   * Leaf
   * Single Stone
   * Multiple Stones
5. Integration with MuJoCo
6. Creating and Adding Objects to a MuJoCo Environment
   * Exporting Blender Object as an .obj File
   * Creating Height Map Terrain in MuJoCo
7. Positioning Objects on a Dynamic Terrain and creating complete Environment

**TASK 1**

**Generating Height Maps**

MuJoCo (Multi-Joint dynamics with Contact) is a physics engine designed for accurate and efficient simulation of continuous, high-dimensional, and contact-rich systems. It is widely used in robotics and biomechanics research, especially for environments requiring physics-based simulation of complex object interactions. However, configuring MuJoCo for advanced setups—such as procedurally generated terrains with intricate features—can be challenging due to its limited built-in functionalities for procedural complexity.

In this report, we tackle these challenges by demonstrating how to set up a procedurally generated terrain in MuJoCo using Perlin noise for complex, natural-looking 3D landscapes. The tutorial-style approach provides a step-by-step guide for configuring the MuJoCo environment to handle more complex scenes and achieve realistic terrain generation.

Procedural generation allows for the creation of randomized, natural-looking terrains by leveraging mathematical algorithms. Procedural island generation is a powerful tool that lets you create unique environments, making it a great skill for game development and visualizations. In this report, we will use heightmaps and Perlin Noise to generate a island with varying biomes. Key components such as noise functions, octaves, and functions used will be explained. Experiment with different seeds, octaves, and color schemes to create endless island variations!

Heightmap: A heightmap is a grayscale image where each pixel represents the elevation of the terrain. Light pixels indicate high elevation (e.g., mountains), and dark pixels indicate low elevation (e.g., water).

Noise Function: A mathematical function generating random values with smooth transitions, simulating natural variations. Common types include Simplex Noise and Perlin Noise, both popular for terrain generation. We have use Perlin noise for generating our maps.

Octave: A layer of noise that increases the detail of a heightmap. Each octave has a different frequency and amplitude.

Seed: An initial value for random generation functions that ensures repeatable outputs.

The following are the functions used in perlin noise as well as in Island function for generating the terrain.

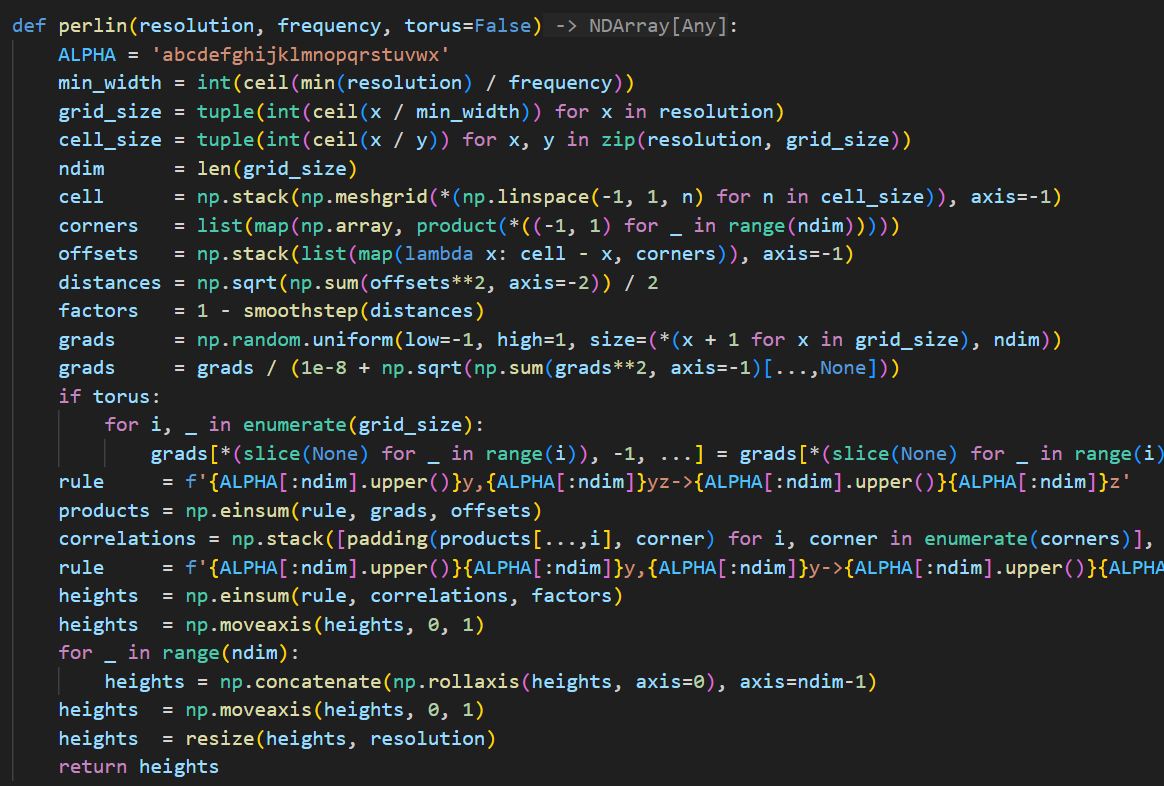
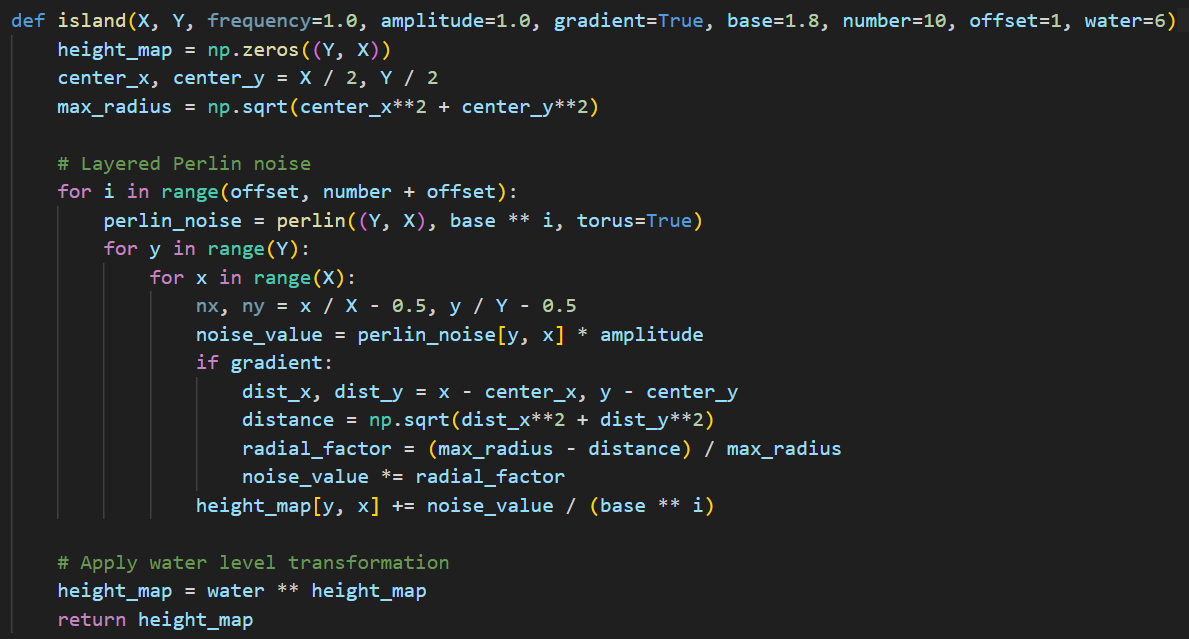
**Functions**

* smoothstep(x): Applies a smoothstep function to an array x, which is a common function used in graphics to interpolate values smoothly.
* padding(x, axis): Adjusts the padding of an array x along specified axes.
* resize(img, shape): Resizes an image img to a new shape using interpolation.
* perlin(resolution, frequency, torus=False): Generates Perlin noise for a given resolution and frequency. It can also generate toroidal noise if torus is set to True.
* island(X, Y, frequency=1.0, amplitude=1.0, gradient=True, base=1.8, number=10, offset=1, water=6): Creates an island-like terrain using layered Perlin noise. It applies a radial gradient to simulate an island shape.
* generate\_terrain(resolution, base, number, offset, water, use\_island=False): Generates terrain using either the island function or layered Perlin noise.
* visualize\_with\_blueprints(height\_map): Visualizes the terrain using the blueprints library.
* save\_terrain\_image(height\_map, filename='perlin\_terrain12.png'): Saves the generated terrain as an image file.

Key Points

* Perlin Noise: A gradient noise function used to create natural-looking textures and terrains.
* Layered Noise: Multiple layers of noise are combined to create more complex and detailed textures.
* Radial Gradient: Used in the island function to simulate the shape of an island by reducing noise values towards the edges.
* Visualization: The blueprints library is used to visualize the generated terrain, which is not a standard library and might require additional setup.

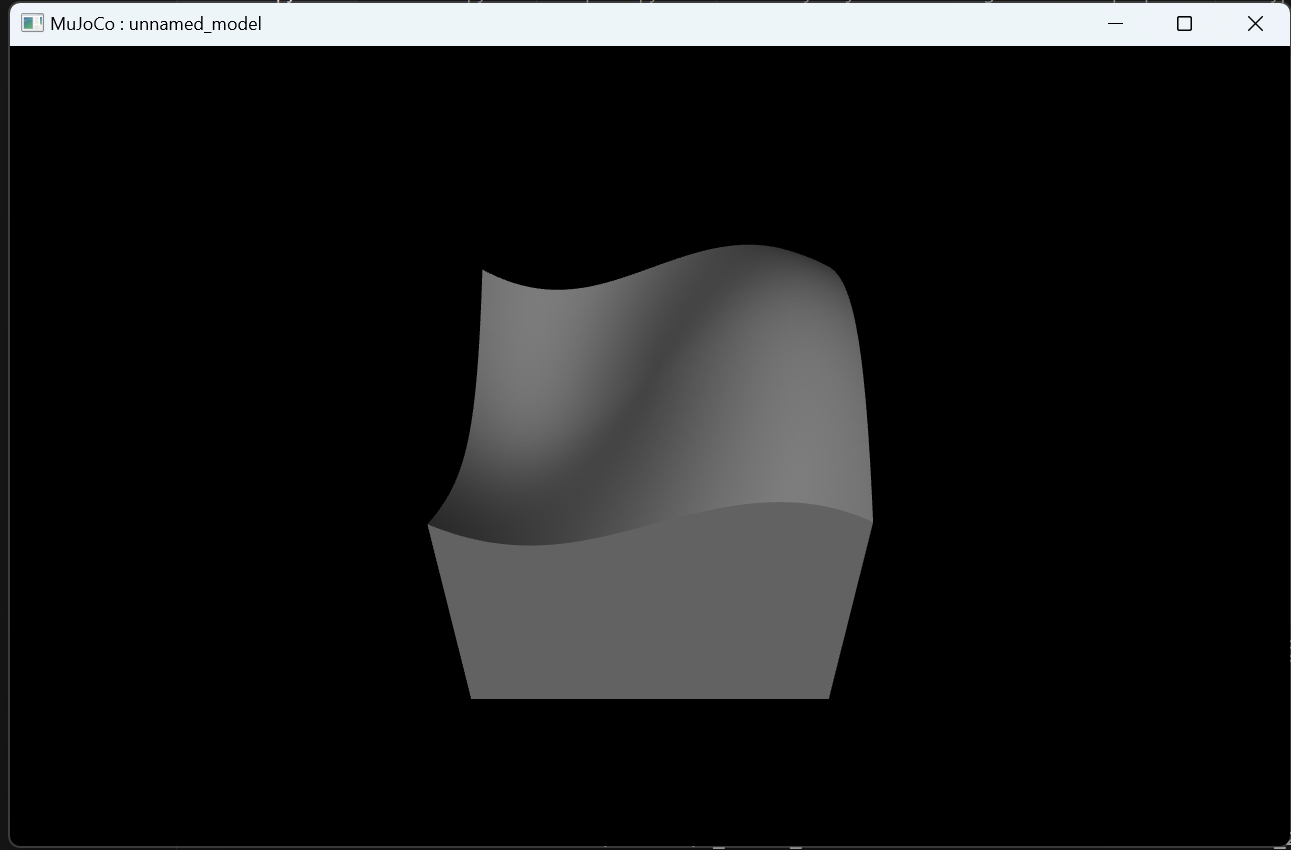
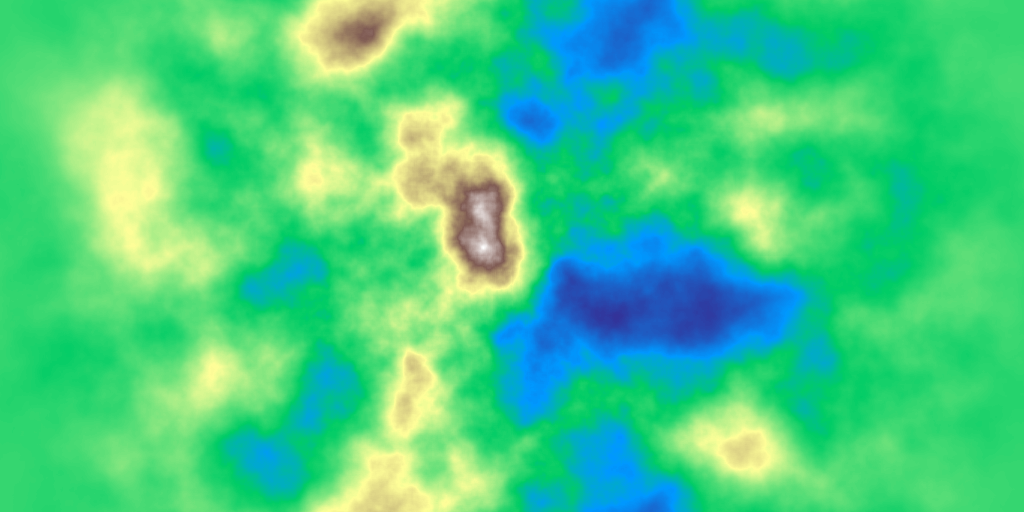
Figure 1 and 2 depicts the implementation of perlin noise and the Island function which generates the output which is shown in figure 3 and 4.

**Fig 1 Perlin Noise Function Implementation Fig 2 Island Function Implementation**

**Output :**

This is when perlin noise is not introduced. A simple terrain is produced by just running Task1.py which only includes island function. Below figure 3 and 4 is the output without perlin noise.

**** 

**Fig 3 Simple Terrain Without Perlin Noise Fig 4 Heat Map Without Perlin Noise**

Perlin noise is another type of noise function commonly used in procedural generation, particularly for generating terrain. Unlike completely random noise, Perlin noise produces smoother, more natural transitions between values, giving it a more organic look suitable for terrains, clouds, and textures.

**Characteristics of Perlin Noise**

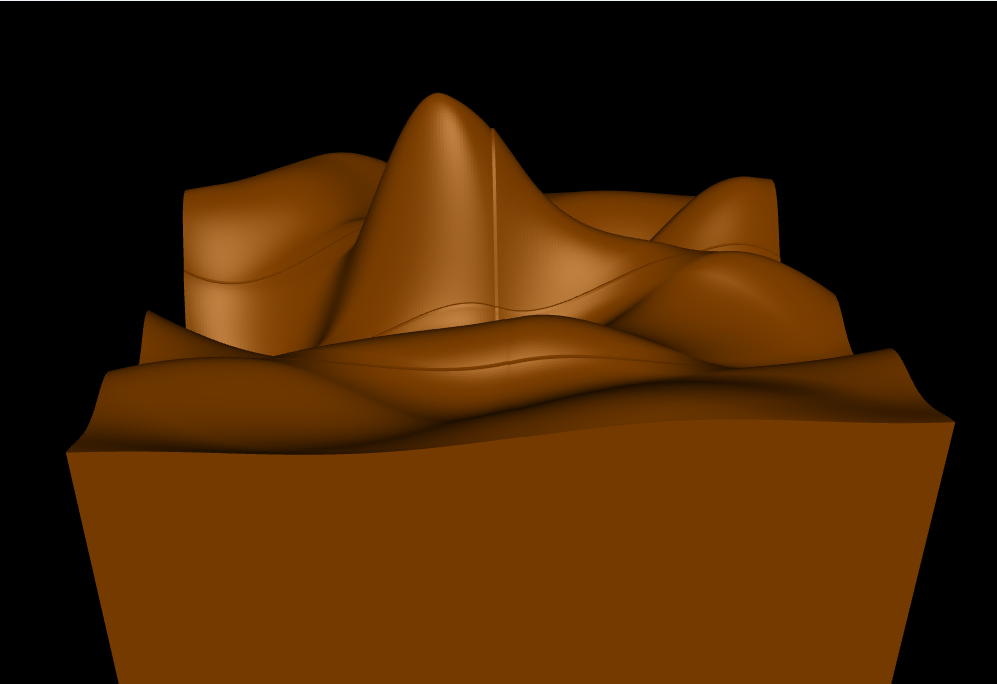
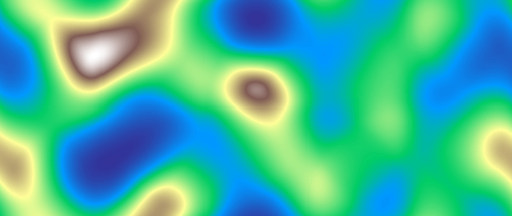
Smooth Transitions: Perlin noise produces gradual transitions between values, creating a smooth, flowing effect rather than abrupt changes.

Grid-Based Interpolation: It is generated by interpolating between points on a grid, which gives it a natural, non-repetitive pattern with a "wavy" effect.

Multiple Octaves for Detail: By combining multiple layers (octaves) of Perlin noise at different scales and frequencies, we can achieve more complex and realistic textures.

Gradient Noise: Perlin noise uses gradients (not just value points) at grid intersections, which makes it look smoother and less grid-like.

This is when perlin noise is introduced to give more resemblance with actual terrain. You can reproduce it by running Task11.py. The output shown below in figure 5 and respective heatmap is in figure 6.

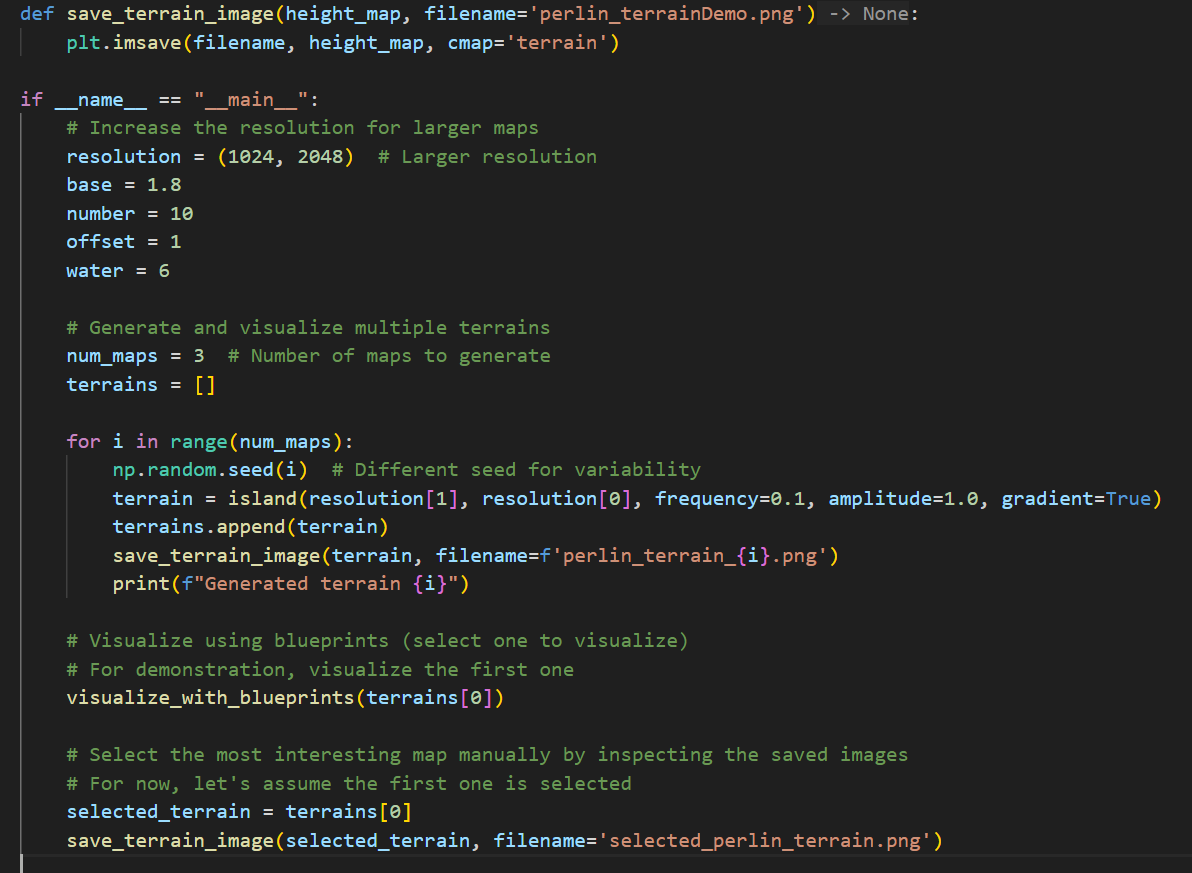
** **

**Fig 5 Terrain with Perlin Noise Fig 6 Heat Map With Perlin Noise**

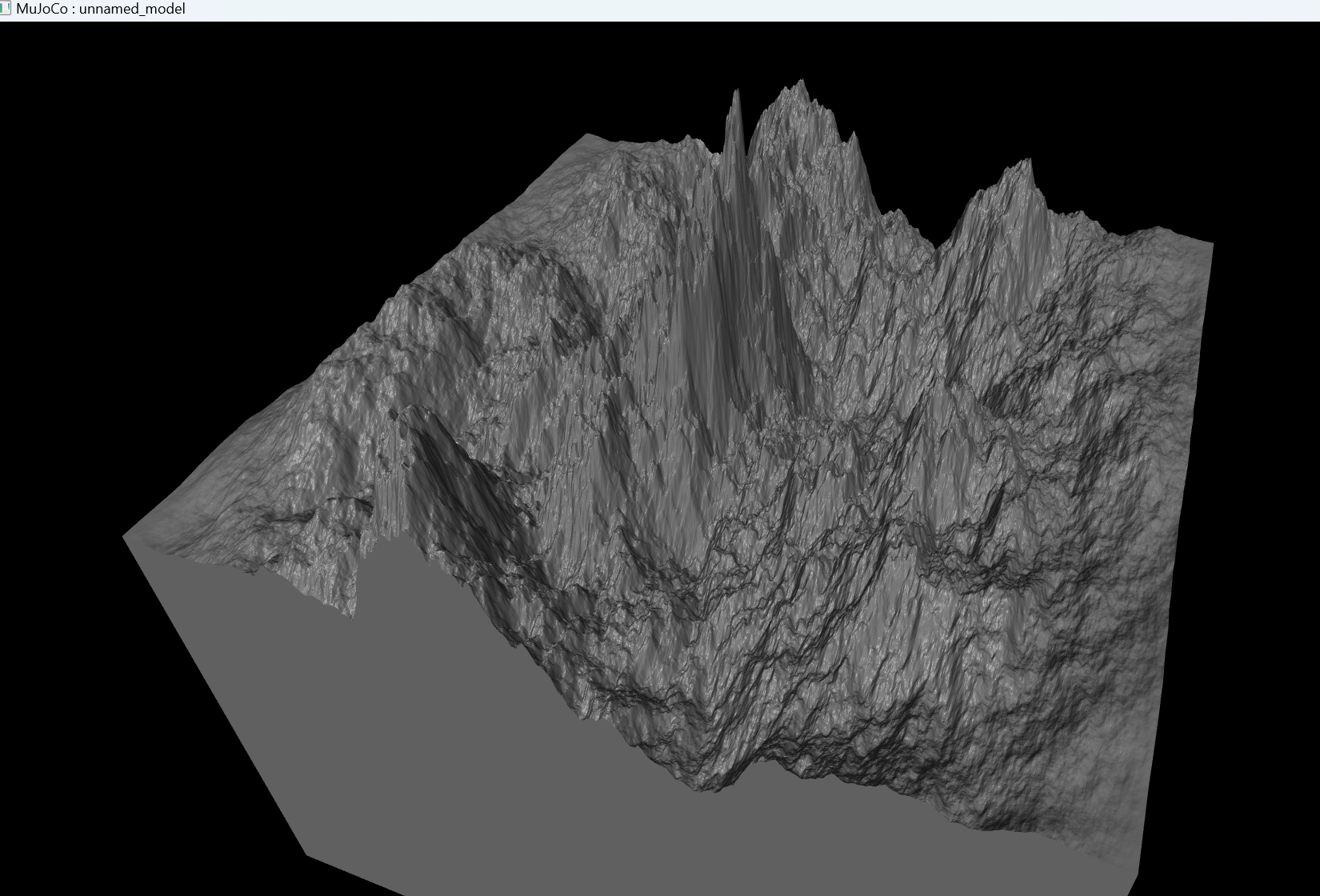
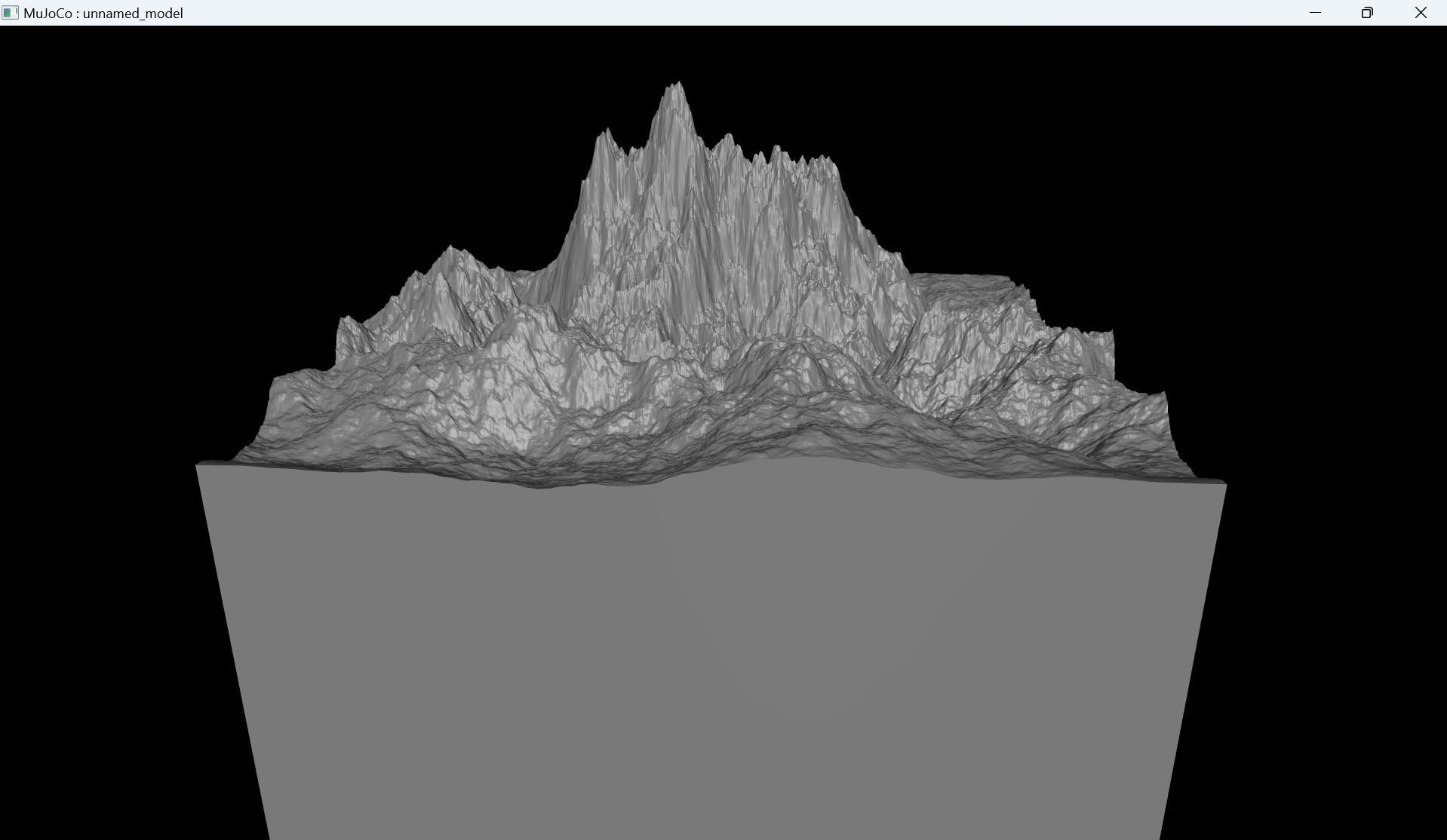
**Larger heat Maps**

For Larger Maps below code is modified for generating multiple larger heat maps and visualising the first one. For creating more you just need to change the loop number and resolution for increasing/decreasing the size of maps in python file **TaskDemo2.py**. But for generating only single image you can use **TaskDemo.py** As seen in the image, rugged and complex terrain features require multiple octaves (e.g., 6–8). Adding octaves results in more small-scale features like sharp peaks and intricate valleys. High-octave Perlin noise adds rough, mountainous features as shown in the Figure 8 and 9, with large cliffs and steep inclines. Figure 10 and 11 shows higher resolution.

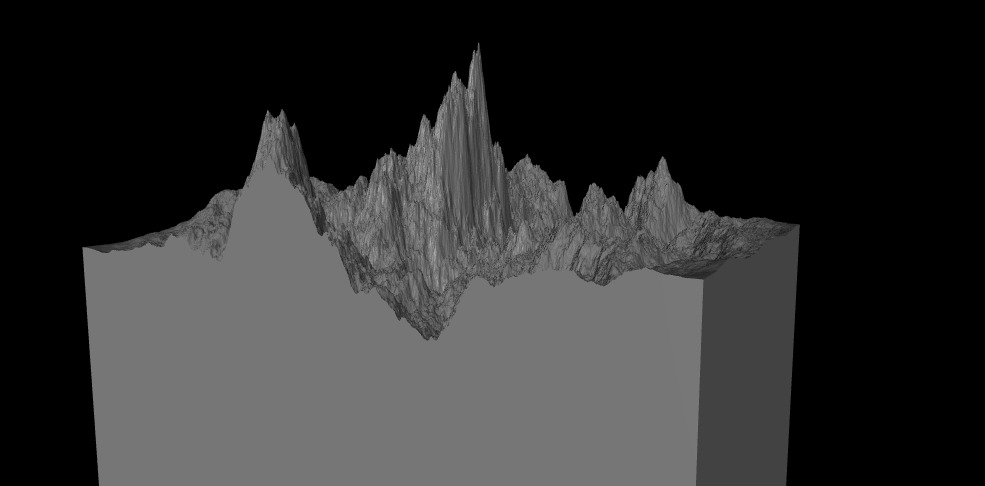
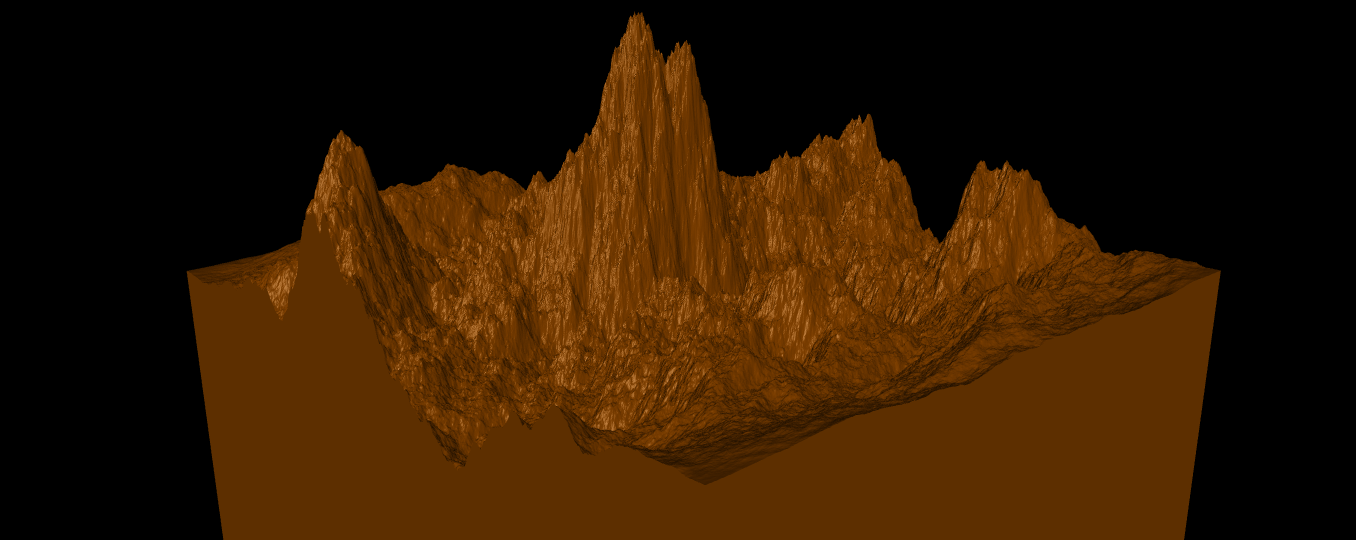
Below in Fig 7 is the snapshot of the code generating the multiple terrain and saving their heatmaps in png format.

****

**Fig 7 Code for generating Multiple terrain**

**Fig 8 Terrain with Mountains Fig 9 Terrain with Mountains**

** **

**Fig 10 Resolution [2048 X 4096] Fig 11 Resolution [2048 X 4096]**

**TASK 2**

**Creating objects using blender**

**Introduction to Blender**

Blender is a free and open-source 3D creation suite that supports the entirety of the 3D pipeline, including modeling, rigging, animation, simulation, rendering, compositing, and motion tracking, as well as video editing and game creation. It is widely used by professionals and hobbyists alike for creating animated films, visual effects, art, 3D printed models, and interactive 3D applications.

***Key Features of Blender***

1. Modeling: Blender offers a wide range of modeling tools, including sculpting, retopology, and curve-based modeling. It supports both polygonal and NURBS modeling.
2. Animation: The software provides a comprehensive set of animation tools, including a character animation pose editor, non-linear animation (NLA) for independent movements, and a sound synchronization feature.
3. Rendering: Blender includes powerful rendering engines like Cycles and Eevee. Cycles is a path-tracer engine known for its realistic results, while Eevee is a real-time engine suitable for quick previews and animations.
4. Simulation: It supports various simulations such as fluid, smoke, hair, cloth, and rigid body dynamics.
5. Video Editing: Blender's Video Sequence Editor (VSE) allows for basic video editing tasks like cutting, splicing, and color grading.
6. Scripting: Python scripting is integrated into Blender, allowing for custom tool creation and automation of tasks.

#### **Basic Transformations:**

In Blender, transformations refer to the manipulation of an object's location, rotation, and scale. These are fundamental operations in 3D modeling and animation.

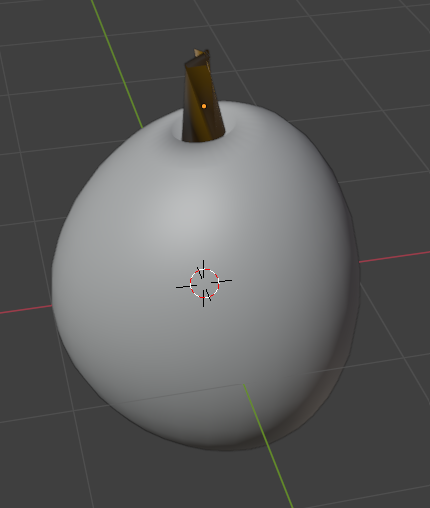
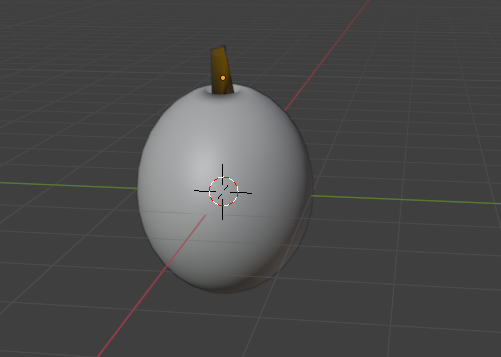
* Location: Moving an object along the X, Y, or Z axis.
* Rotation: Rotating an object around its pivot point.
* Scale: Changing the size of an object along one or more axes.

***How to Perform Basic Transformations****:*

* Using the Properties Panel: Access the Transform section in the Object Properties tab to manually input values for location, rotation, and scale.
* Using Keyboard Shortcuts:
  + Press 'G' to grab and move an object.
  + Press 'R' to rotate an object.
  + Press 'S' to scale an object.
  + Constrain transformations to a specific axis by pressing 'X', 'Y', or 'Z' after initiating the transformation.

**Apple:**

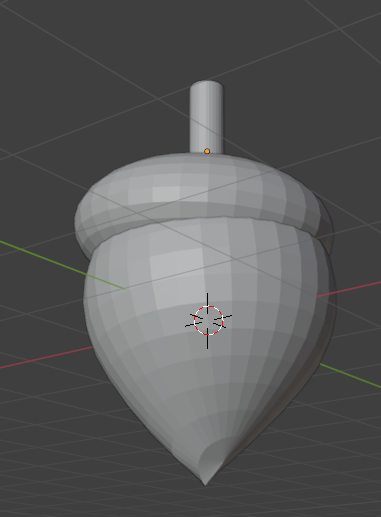
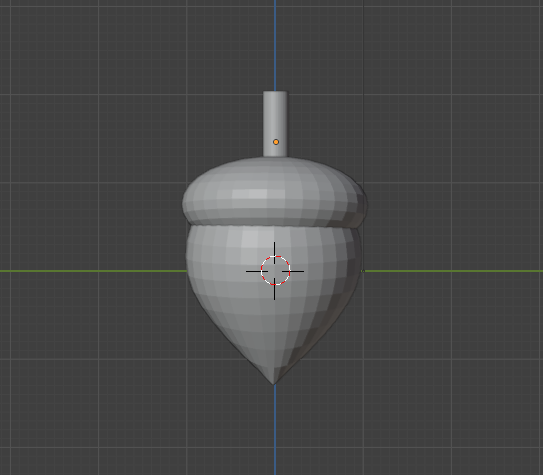
* Insert a Sphere: Use Shift + A and choose "Mesh" > "UV Sphere" to place a sphere in your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Choose Vertices: Highlight the top and bottom vertices of the sphere.
* Activate Proportional Editing: Press O to turn on Proportional Editing.
* Adjust Z-axis Scale: Use S followed by Z to scale the sphere along the Z-axis, forming the basic shape of an apple.
* Add a Cylinder: Press Shift + A and select "Mesh" > "Cylinder" to add a cylinder for the stem.
* Scale and Position: Scale the cylinder down and position it at the top of the apple. Use S, Z to adjust the thickness and G to move it.

****

**Fig 12 Apple**

**Acorn:**

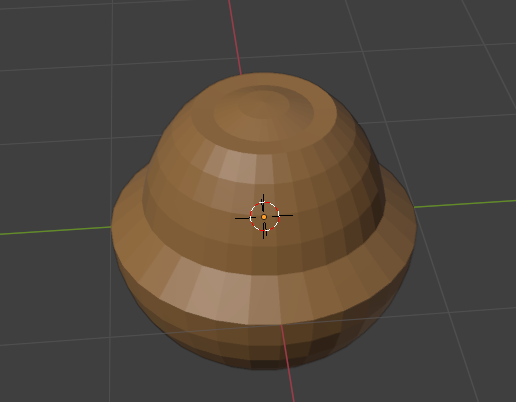
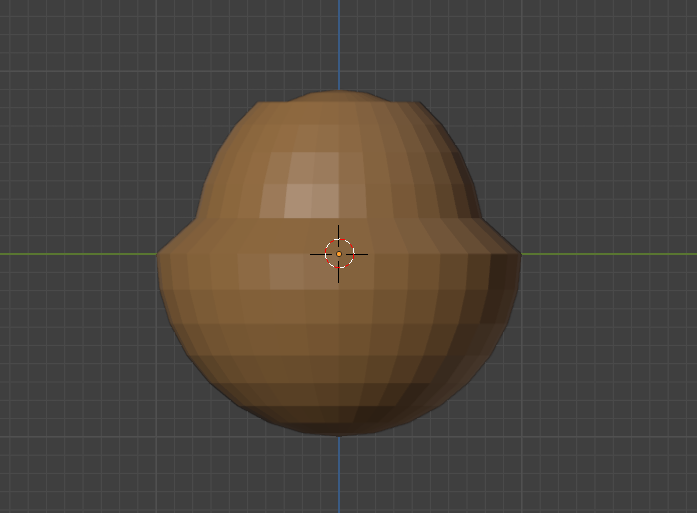
* Insert a Sphere: Use Shift + A and choose "Mesh" > "UV Sphere" to add a sphere to your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Select Vertices: Highlight the top vertices of the sphere to shape the acorn's cap.
* Activate Proportional Editing: Press O to enable Proportional Editing.
* Adjust Z-axis Scale: Use S followed by Z to scale the sphere along the Z-axis, forming the elongated body of the acorn.
* Add Acorn Cap: Duplicate the top portion of the sphere to create the acorn's cap. Scale it slightly larger than the body.
* Create the Stem: Add a small cylinder using Shift + A and position it at the top of the acorn cap.
* Scale and Position: Scale the cylinder down and adjust its position to resemble the acorn's stem.
* Refine Shape: Use loop cuts (Ctrl + R) and Proportional Editing to refine the stem's shape, making it appear more natural.

****

**Fig 13 Acorn**

**Chestnut:**

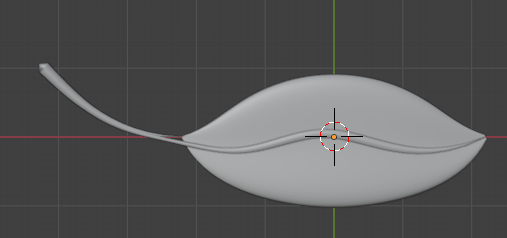
* Insert a Sphere: Use Shift + A and select "Mesh" > "UV Sphere" to add a sphere to your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Select Vertices: Choose the top vertices of the sphere to shape the chestnut's rounded top.
* Activate Proportional Editing: Press O to enable Proportional Editing.
* Adjust Z-axis Scale: Use S followed by Z to scale the sphere along the Z-axis, forming the slightly flattened body of the chestnut.
* Smooth the Surface: Right-click on the sphere and select "Shade Smooth" to smooth out the surface.
* Modify Shape: Use Proportional Editing to adjust the shape, creating the characteristic bulges of a chestnut.
* Add Details: Use loop cuts (Ctrl + R) to add more geometry and refine the chestnut's shape, focusing on the natural asymmetry.
* Texture and Paint: Assign a material and enter Texture Paint mode to add realistic color and texture variations to the chestnut's surface.

****

**Fig 14 Chestnut**

**Leaf:**

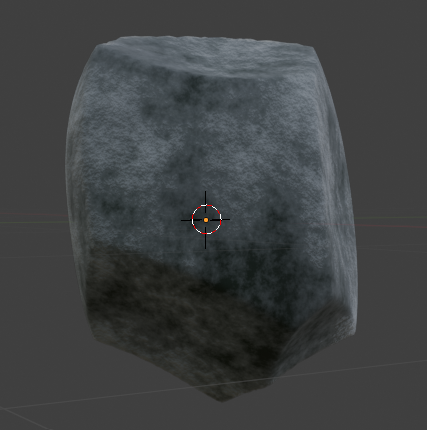
* Insert a Plane: Use Shift + A and select "Mesh" > "Plane" to add a plane to your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Subdivide the Plane: Right-click on the plane and select "Subdivide" to increase the geometry, allowing for detailed shaping.
* Activate Proportional Editing: Press O to enable Proportional Editing.
* Shape the Leaf: Select vertices and use G to grab and move them, forming the basic outline of a leaf. Focus on creating a symmetrical shape with a pointed tip and a wider base.
* Add Veins: Use loop cuts (Ctrl + R) to add additional geometry for the leaf veins. Adjust these vertices to create a natural vein pattern.
* Smooth the Surface: Right-click on the leaf and choose "Shade Smooth" to smooth out the surface.
* Add Texture: Assign a material and use a texture map to add realistic leaf textures, focusing on color variations and vein details.
* Bend the Leaf: Use Proportional Editing to slightly bend the leaf, giving it a more natural, organic appearance.
* Final Adjustments: Make any final tweaks to the shape and texture to ensure the leaf looks realistic and detailed.

****

**Fig 15 Leaf**

**Single Stone :**

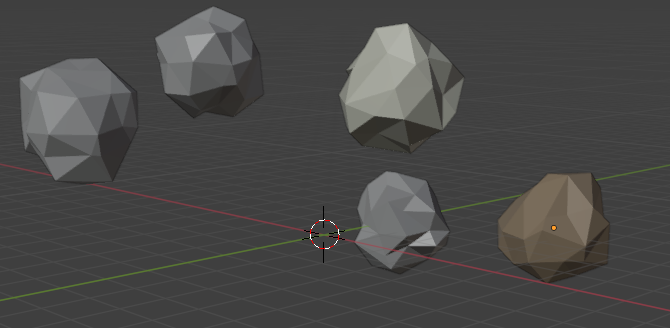
* Insert a Cube/Sphere: Use Shift + A and select "Mesh" > "Cube/sphere" to add a cube/sphere to your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Subdivide the Cube: Right-click on the cube and select "Subdivide" to increase the geometry, allowing for more detailed shaping.
* Activate Proportional Editing: Press O to enable Proportional Editing.
* Randomize Shape: Select vertices randomly and use G to grab and move them slightly, creating an irregular stone shape.
* Smooth the Surface: Right-click on the stone and choose "Shade Smooth" to smooth out the surface.
* Refine Shape: Use additional loop cuts (Ctrl + R) and Proportional Editing to further refine the stone's natural, uneven appearance.
* Add Texture: Assign a material and use a texture map to add realistic stone textures, focusing on color variations and surface details.
* Final Adjustments: Make any final tweaks to the shape and texture to ensure the stone looks natural and realistic.

****

**Fig 16 Single Stone**

**Multiple stones:**

* Insert a Cube/Sphere: Use Shift + A and select "Mesh" > "Cube/sphere" to add a cube/sphere to your scene.
* Switch to Edit Mode: Press Tab to enter Edit Mode.
* Subdivide the Cube: Right-click on the cube and select "Subdivide" to increase the geometry for detailed shaping.
* Activate Proportional Editing: Press O to enable Proportional Editing.
* Randomize Shape: Select vertices randomly and use G to grab and move them slightly, creating an irregular stone shape.
* Smooth the Surface: Right-click on the stone and choose "Shade Smooth" to smooth out the surface.
* Duplicate Stones: Return to Object Mode by pressing Tab, then use Shift + D to duplicate the stone multiple times.
* Vary Each Stone: For each duplicate, enter Edit Mode and slightly adjust the vertices to ensure each stone has a unique shape.
* Position Stones: Use G to move and position each stone within the frame, creating a natural arrangement.
* Add Texture: Assign a material and use a texture map to add realistic stone textures, ensuring variations in color and surface details for each stone.
* Final Adjustments: Make any final tweaks to the shapes and textures to ensure the stones look natural and cohesive within the scene.

****

**Fig 17 Multiple Stones**

**Integration with Mujoco**

**1) Exporting a Blender Object as an .obj File and Loading It in a MuJoCo Python Script**

In this tutorial, we'll use an apple as an example object to guide you through creating the 3D model in Blender, exporting it as an .obj file, and loading it into a MuJoCo environment using a Python script.

**Step 1: Create an Apple Object in Blender**

In the tutorial above, we've demonstrated how to create an apple in Blender. Now, assuming the apple has already been created, we'll proceed to the next step.

**Step 2: Export the Apple Object as an .obj File**

To export your apple object, first go to the top menu and click on **File**. From the dropdown menu, navigate to **Export** and select **Wavefront (.obj)**. In the export window that appears, choose the directory where you'd like to save the .obj file. Name the file **apple.obj**. Make sure to check the options for **Selection Only** and **Apply Modifiers** (if necessary), so that only the apple model and its transformations are exported. Once everything is set, click **Export OBJ** at the top right to complete the export.

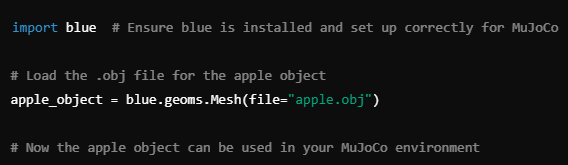
**Step 3: Place the apple.obj File in the Same Directory as Your Python Script**

Once you have exported the **apple.obj** file from Blender, move or save the file into the same directory where your Python script is located for Mujoco. This will make it easier for the script to find and load the file.

**Step 4: Load the apple.obj File in a MuJoCo Python Script**

Now, let's load the exported apple model into MuJoCo using a Python script.

1. Open your Python environment (such as PyCharm, VSCode, or Jupyter Notebook).
2. Use the following Python code to load your apple object using MuJoCo's blue module:
3. Code Example:

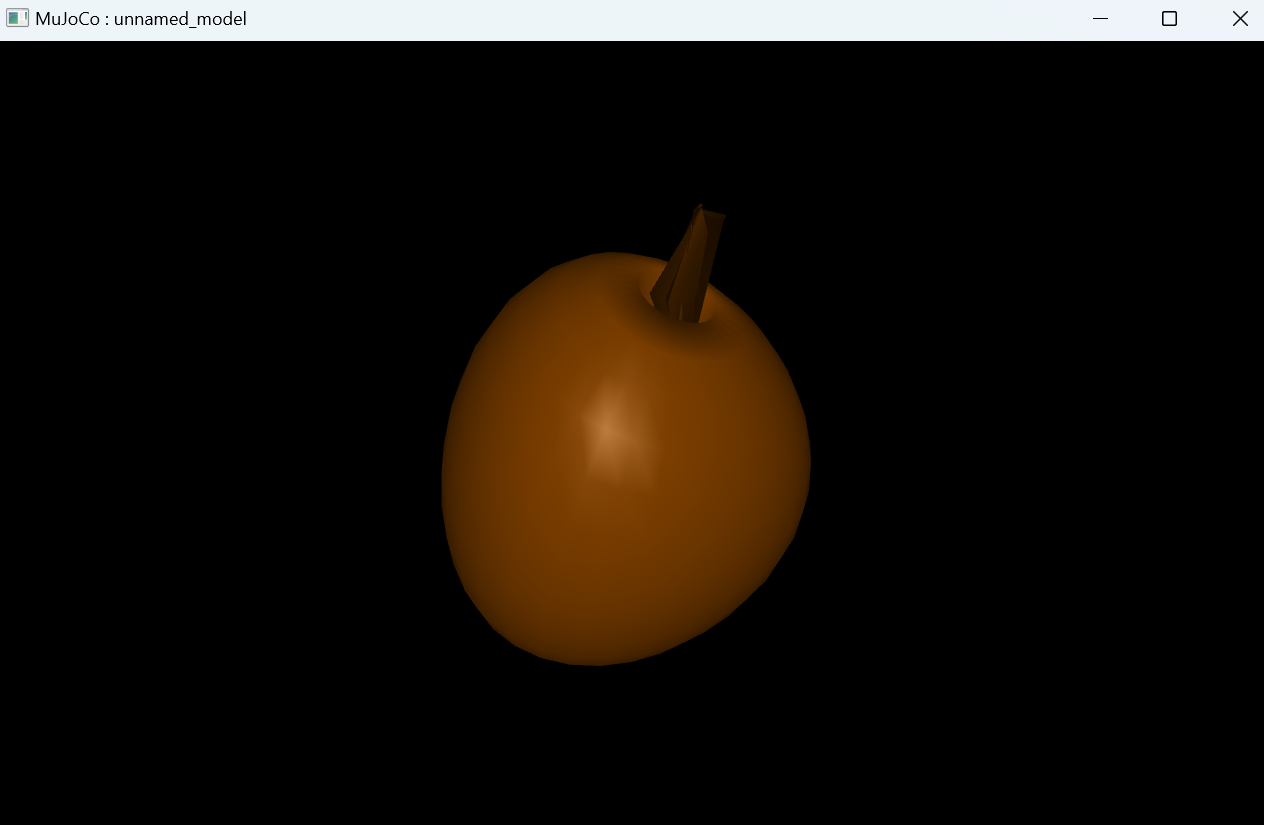
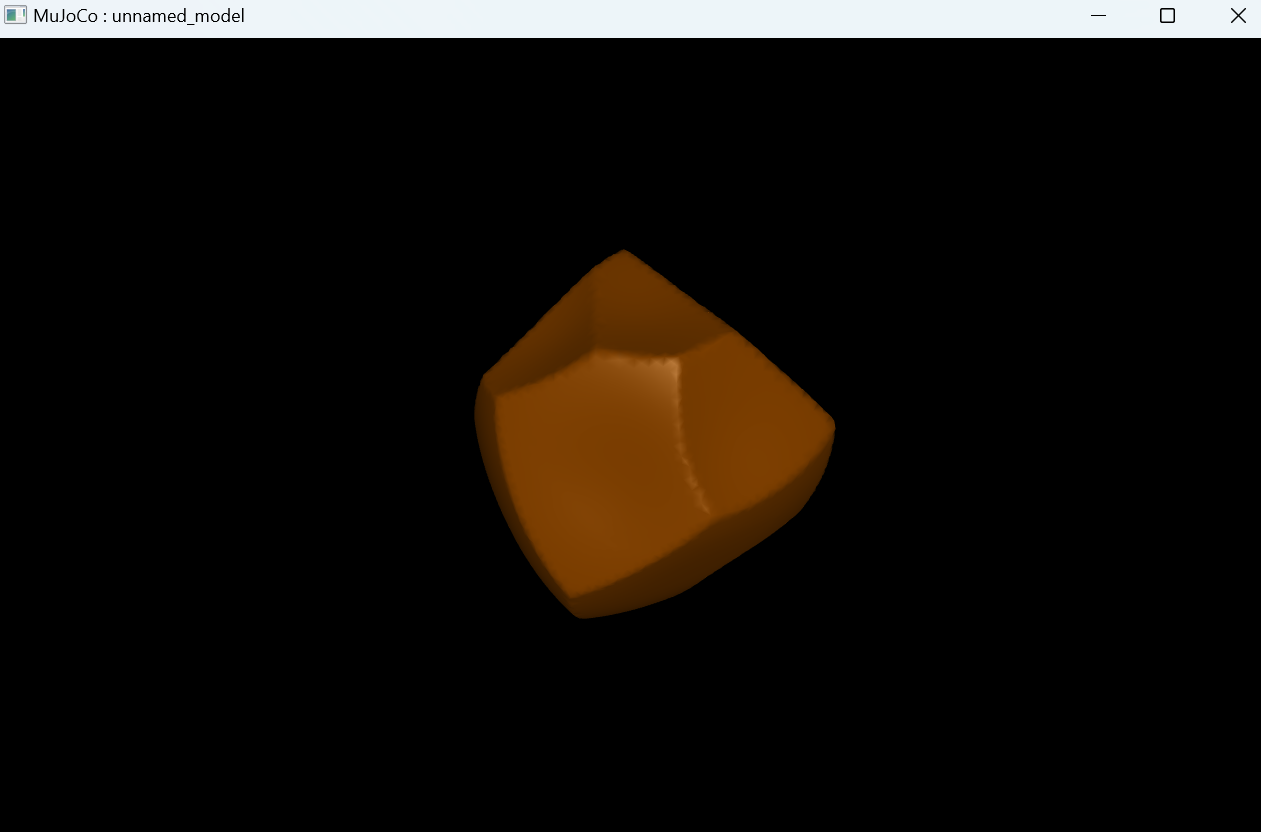
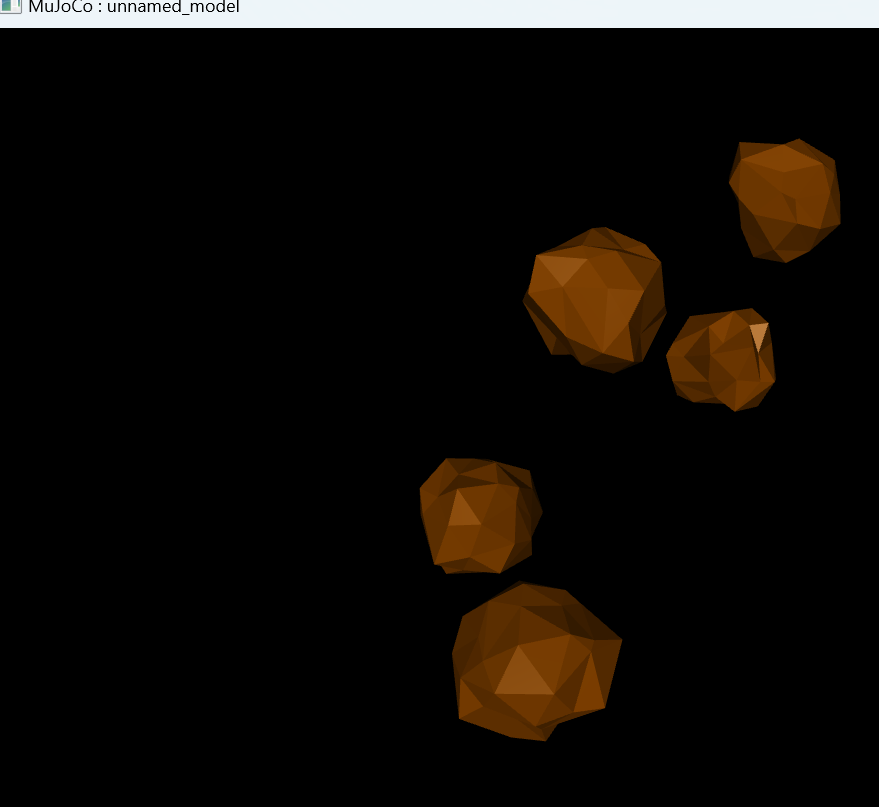


**Fig 18 Code implementation**

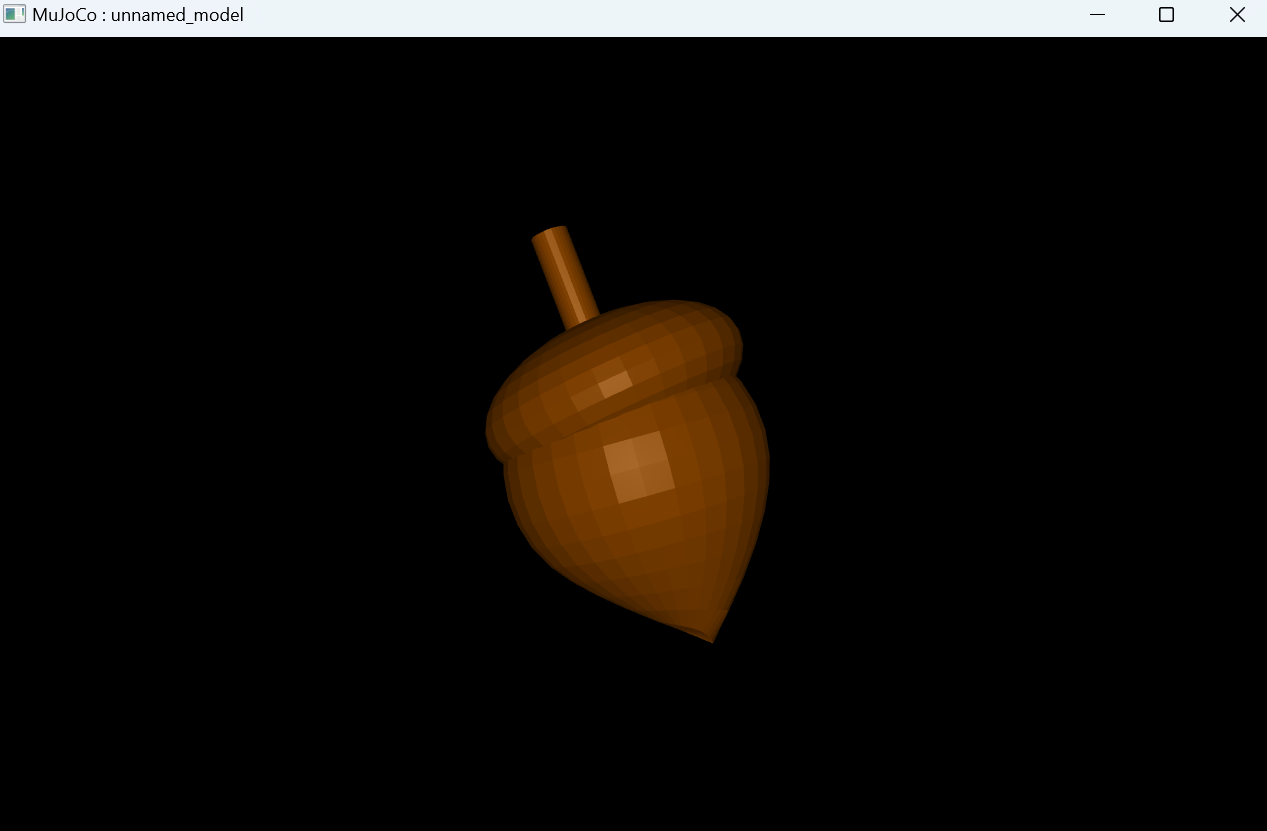
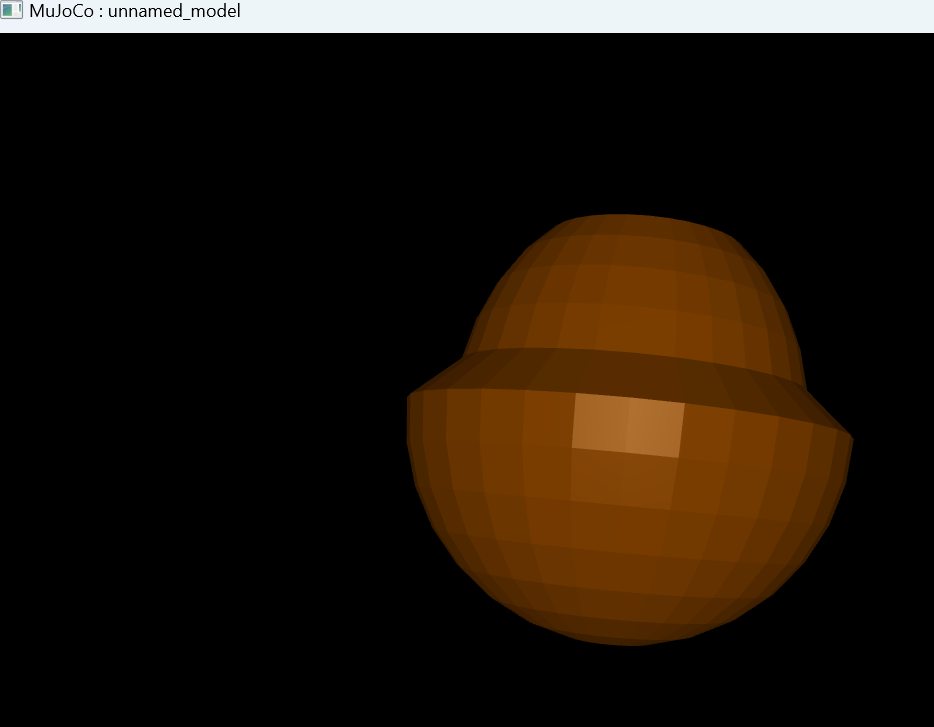
**Step 5: Run the MuJoCo Environment**

Now that your **apple.obj** file is successfully loaded into the MuJoCo environment, you can simulate or manipulate the apple model according to your needs in the MuJoCo environment. For example, you can apply physics to the apple, move it around, or interact with other objects.

Below figures from 19 to 22 are the images for different objects created in blender and simulated in Mujoco Environment.

**  **

**Fig 19 Apple in Mujoco Environment Fig 20 Stone in Mujoco Environment Fig 21 Multiple Stones in Mujoco Environment**

** **

**Fig 22 Acorn in Mujoco Environment Fig 23 Chestnut in Mujoco Environment**

**2)Creating a Height Map Terrain and Adding an Object in a MuJoCo Environment Using Python**

In this tutorial, we'll walk through how to create a simple height map (terrain) in a MuJoCo environment using the blue library. We'll define the terrain using a height map, attach it to the world, and view the result in the MuJoCo simulation.

**Step 1: Create the Height Map**

We generate the height map using the island function as mentioned in above tutorial, which creates terrain features like mountains or islands. We specify parameters like frequency, amplitude, and whether to include a gradient effect.

Parameters:

* **frequency=0.1**: This controls the "bumpiness" of the terrain. A lower value means fewer hills and smoother terrain, while a higher value makes the terrain more rugged.
* **amplitude=1.0**: This defines the height of the terrain features. A higher value results in taller peaks and deeper valleys.
* **gradient=True**: This adds a gradient effect, giving the terrain a smooth transition from low to high points.

The resulting height\_map contains the elevation data that will be used to shape the terrain.

**Step 2: Initialize the World**

We create a new blue.World() object, which represents the entire simulation environment in MuJoCo. This world will serve as the container for all objects, including the terrain and any additional objects (like the apple or stones) you wish to add later.

Code Example:



**Fig 24 Code implementation**

**Step 3: Create and Attach the Height Field (Terrain)**

To add the height map to the world, we create an HField (height field) geometry using the previously generated height\_map. We also set the color of the terrain to "white" for visibility.

Here:

* **terrain=height\_map**: Specifies the height map that shapes the terrain.
* **color="white"**: Sets the terrain color to white. You can change this to any other color as needed.



**Fig 25 Code implementation**

Next, we attach the height field (terrain) to the world.



**Fig 26 Code implementation**

**Step 4: View the World**

Finally, you can view the entire scene by calling world.view(). This launches the MuJoCo viewer, allowing you to interact with and visualize the terrain and any attached objects in 3D.



**Fig 27 Code implementation**

**Positioning Objects on a Dynamic Terrain in 3D**

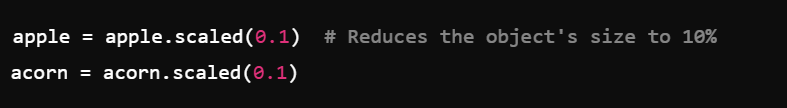
In this tutorial, we'll explore how objects, like an apple or an acorn, can be dynamically positioned on a terrain using Python and the blueprints 3D modeling library. This process ensures the objects are placed accurately on a realistic terrain that accounts for elevation changes.

**Step 1: Generating the Terrain**

The terrain is created as a **heightmap**, a 2D/3D grid where each value represents the elevation at a specific (x, z) coordinate. Perlin noise is used to generate natural-looking height variations as stated in above tutorial. To make the terrain resemble an island, a gradient effect is applied, which reduces the height as we move farther from the center.

**Step 2: Loading 3D Objects**

Objects like an apple or an acorn are loaded using 3D model files (Chestnut.obj and Acorn.obj) with the blue.geoms.Mesh function as stated in above tutorial and code implementation is shown in Fig 18. These objects can be assigned colors for better visibility and then scaled to fit the terrain's proportions using the scaled method as shown in Fig 28.



**Fig 28 Scaling of object**

**Step 3: Calculating Position on the Terrain**

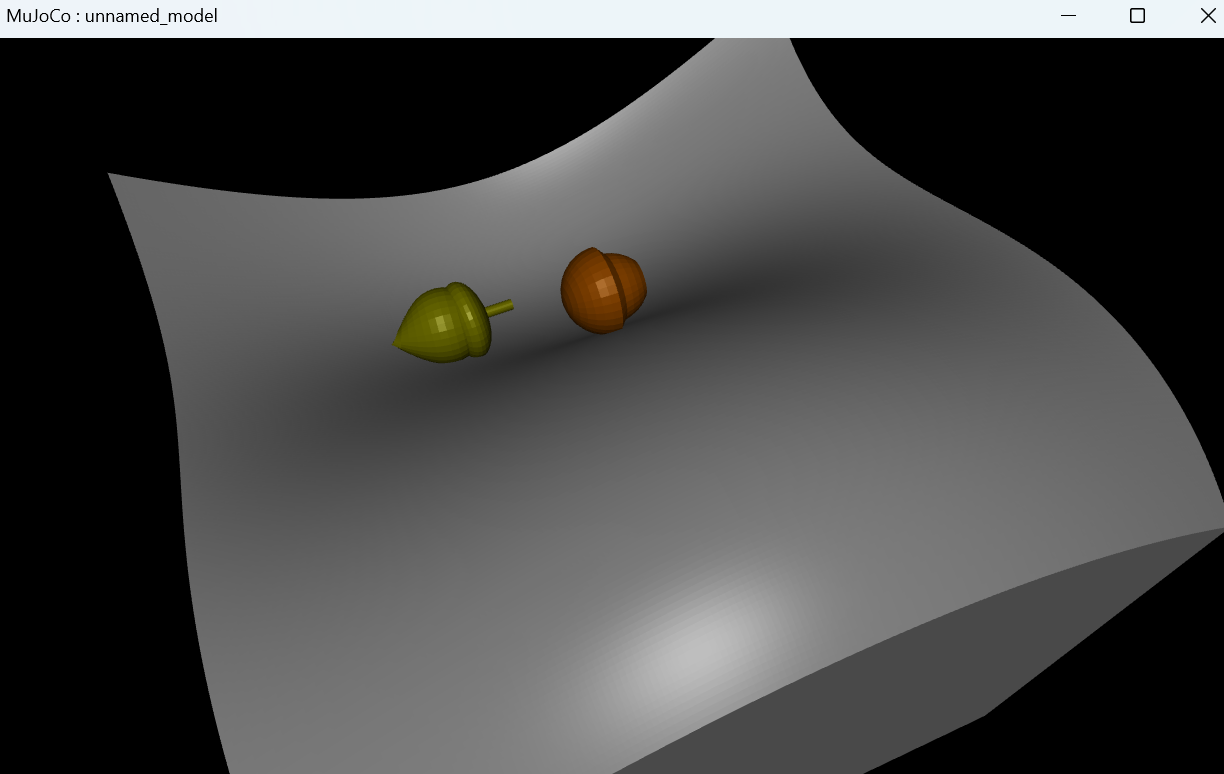
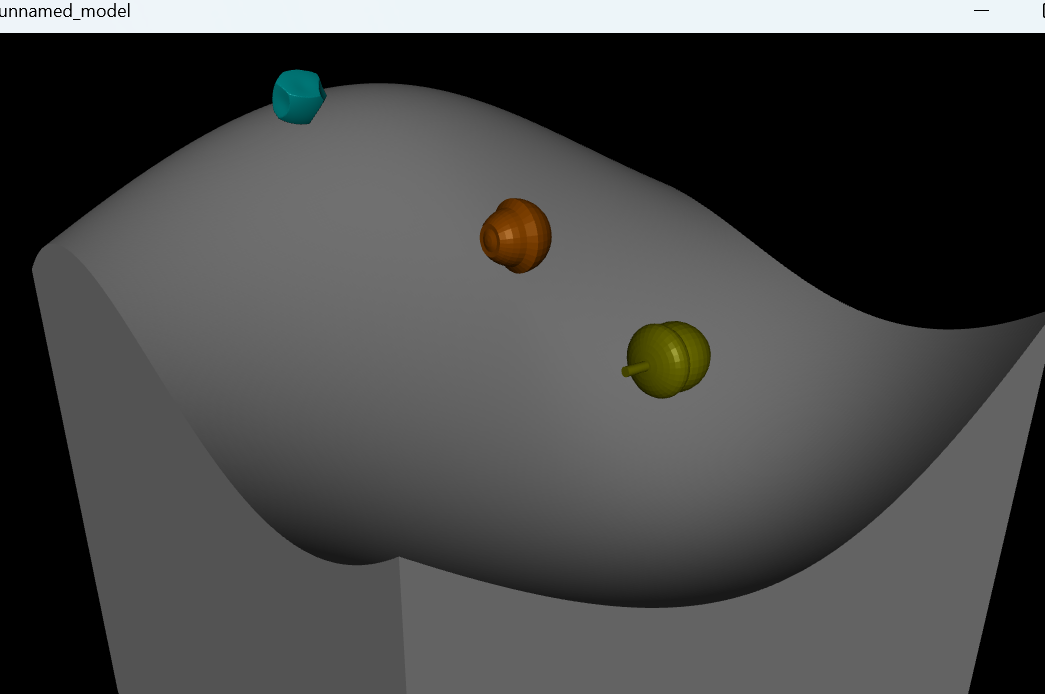
To position an object accurately on the terrain, the heightmap is queried at the desired (x, z) coordinate to retrieve the elevation. A vertical offset is then added to ensure the object is slightly above the terrain, preventing it from appearing embedded in the surface.

**Step 4: Attaching Objects to the World**

Once the objects have been correctly positioned, they are attached to the 3D world using world.attach as shown in Fig 24. This integrates the objects into the scene and ensures they appear when the environment is rendered.

**Step 5: Visualizing the Environment**

Finally, the 3D world, along with the terrain and positioned objects, is displayed using world.view as example is shown in Fig 26 and 27. This function renders the complete environment, allowing you to see the terrain and objects combined in a realistic setting which is shown in Fig 29.

**Fig 29 Integrated Environment**

**References:**

* <https://medium.com/@samuelsullins/blender-basics-part-1-the-very-basics-of-blender-2ad869403a4d>
* <https://www.blender.org/>
* <https://www.blenderguru.com/>
* <https://docs.blender.org/>
* <https://www.youtube.com/watch?v=hYUZl6uZr-0>
* <https://www.youtube.com/watch?v=CsskhkwtvAs>
* <https://www.youtube.com/watch?v=lNUpgO3SLMU>
* [https://medium.com/@travall/procedural-2d-island-generation-noise-functions- 13976bddeaf9](https://medium.com/@travall/procedural-2d-island-generation-noise-functions-%2013976bddeaf9)
* https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html