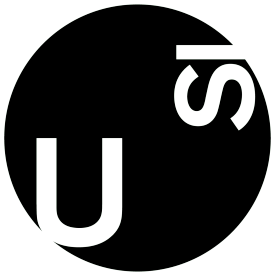
Università della Svizzera italiana



Faculty of Informatics

Computational Fabrication
(Spring 2024)

Make It Stand

Submitted by:

Krunal Rathod | krunal.rathod@usi.ch

Submitted on: 23 June 2024

June 24, 2024

Contents

1 Introduction

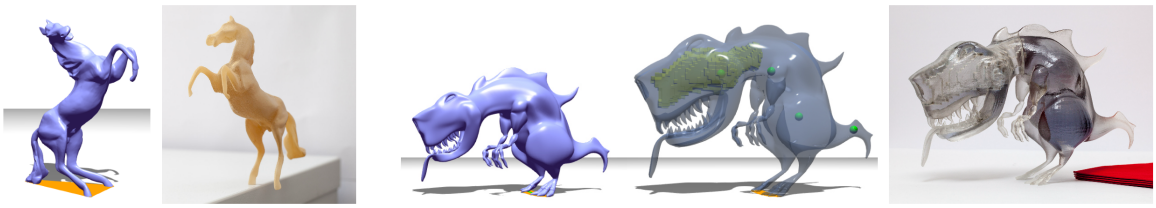


Figure 1: Make it Stand: Final Results from Original Paper

In the "Make it Stand" project, I aimed to replicate the findings and implementation 1 described in the original paper "Make it Stand: Balancing Shapes for 3D Fabrication" by Moritz Bächer, Bernd Bickel, Doug L. James, and Hanspeter Pfister [1]. My primary goal is to create a clear and concise replication that retains the core functionalities and outcomes of the original work and making it easy to understand.

The original paper addresses a common challenge in 3D fabrication, ensuring that fabricated objects are stable and can stand on their own without additional support. This involves complex geometric and physical computations to adjust the internal mass distribution of a shape. This project's goal is to simplify these methodologies, with modern libraries as libigl and Eigen.

My approach began with an in-detail study of the original paper to thoroughly understand the theoretical foundations and practical implementations. Then translated these methodologies into simplified code, ensuring the main objective of the original work was maintained.

2 Objective

The specific objectives of my project is multifaceted. First, I aim to thoroughly comprehend the original paper's methodologies, ensuring a deep understanding of the theoretical foundations and practical implementations presented.
Second, replicate those methodology, key findings and results of the original paper. This involves accurately translating complex algorithms and processes while maintaining the integrity of the original work.
Third, a most important objective is to simplifying these implementation without losing the core functionalities and outcomes of the work. This simplification is most essential to make the code more accessible and easier to understand.

3 Implementation

This project leverages libraries such as `Eigen` for geometric computations, `ImGui` for graphical user interface elements, and `igl` for mesh processing utilities within the GLFW-based viewer framework.

The `main.cpp` file begins by including necessary headers and defining constants like `PI` and `DEBUG` mode. It initializes default parameters such as the mesh file and predefined colors. Key viewer data indices (`mesh_data_id`, `inner_data_mesh_id`, etc.) are declared to manage different visual components. To encapsulates the current state of the mesh object, `State` structure, includes geometry (`V`, `F`), inner void mesh (`innerV`, `innerF`), and plane geometry (`planeV`, `planeF`). Additional state variables track orientation (`yaw`, `pitch`, `roll`), gravity vector (`gravity`), and various user workflow flags (`selectBalancePoint`, `selectOrientation`, `isCarving`). These elements collectively manage the interactive manipulation and visualization of the mesh object.

To assist in trigonometric calculations and mesh manipulation tasks, helper functions like `sin_deg`, `cos_deg`, and `getBalancePoint` implemented. The initialization process (`initState`) loads a mesh file, aligns it to axis planes, and sets up initial display configurations, including defining a plane and setting a default gravity direction. Interactive controls are facilitated through `ImGui`, enabling users to select a balance point, adjust orientation (`yaw`, `pitch`, `roll`), and initiate carving operations on

the inner void mesh. Callback functions (`mouse_down`) handle user interactions with the mesh, such as selecting a balance point based on mouse clicks.

The main execution (`main`) orchestrates the initialization of the viewer and GUI elements, setting up callbacks for user input and displaying various mesh components. It initializes the viewer with the main mesh, inner mesh, and plane geometries, updating them dynamically based on user actions. Debugging visualizations, like gravity vectors and carving planes, are optionally displayed based on the `DEBUG` flag.

3.1 Center of Mass

The file `center_of_mass.cpp` is an implementation of the functions declared in `center_of_mass.h`, designed to calculate the center of mass of a 3D model represented by its vertices and faces. This calculation is fundamental for determining the balance of the model. The primary function in this implementation is `center_of_mass`, which computes the total mass of the mesh and the center of mass. The implementation in `center_of_mass.cpp` efficiently computes the center of mass by leveraging Eigen's matrix and vector functionalities. By iterating over the mesh faces and performing detailed calculations, the functions ensure accurate results, which are crucial for applications requiring precise balance and mass distribution information in 3D models.

```
1  double center_of_mass(
2      const Eigen::MatrixXd &V,
3      const Eigen::MatrixXi &F,
4      Eigen::Vector3d &center);
5
6  void center_of_mass(
7      const Eigen::MatrixXd &V,
8      const Eigen::MatrixXi &F,
9      double mass,
10     Eigen::Vector3d &center);
```

Listing 1: Center of Mass

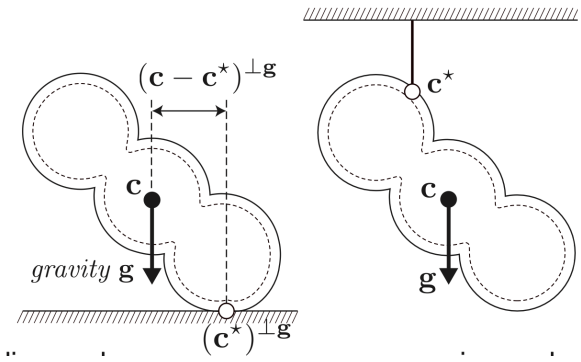


Figure 2: reference frame

The `center_of_mass` function initializes the calculation process. It begins by iterating over each triangular face of the mesh. For each triangle, it extracts the vertices and computes the triangle's area using Heron's formula, which involves calculating the lengths of the sides and then the semi-perimeter. Additionally, it computes the unit normal vector for each triangle, which is essential for determining the orientation and contribution of the face to the overall mass. The function then accumulates the mass contributions of each face, considering the area and the vertex positions. Once the mass is accumulated, the function calls a helper function, also named `center_of_mass`, to finalize the calculation. Finally, the accumulated values are divided by the total mass, resulting in the average position of the center of mass.

The header file `center_of_mass.h` declares these functions, providing a clear interface for their implementation. It specifies the input parameters, including the vertex matrix `V` and the face matrix `F`, as well as the output parameter, a 3D vector `center` that holds the center of mass.

Alternatively, we can also use the `igl` inbuilt function `centroid.h`.

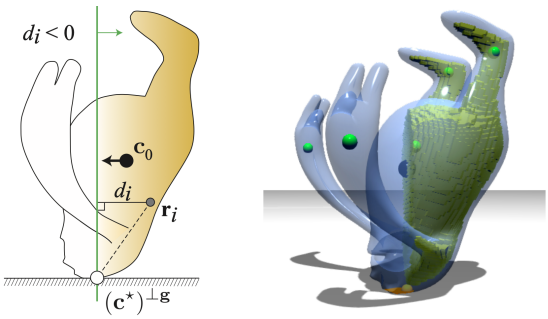


Figure 3: reference frame

3.2 Inner carving

The `InnerVoidMesh` class, along with its supporting `Voxel` class and associated functions, was implemented to facilitate the process of carving an inner void within a 3D mesh, with the goal of optimising the center of mass (COM) for balance.

The `Voxel` class holds the definition of a voxel, which includes properties such as its center, distance to the mesh, grid indices, corner indices, and boolean flags indicating whether it is filled or inner. These properties are very important for the carving as they help in determining the voxel's position, relationship with mesh, and whether it should be removed or not. Each voxel is initialized with default values, setting its center to the origin, distance to zero, indices to negative 1, and both `isFilled` and `isInner` to false and true, respectively. This ensures that each voxel starts in a neutral state and is then configured based on its actual position and status within the mesh.

The `InnerVoidMesh` class encapsulates the carving process, maintaining a 3D list of voxels (`innerMesh`), the dimensions of the voxel grid, the voxel size, and various parameters related to the mesh and the carving process. The constructor initializes these parameters and prepares the voxel grid by converting the input mesh vertices and faces into a voxel representation. This conversion involves computing the bounding box of the mesh, generating voxel centers, and calculating the signed distance from each voxel center to the mesh to determine whether the voxel is inside the mesh. The signed distance computation is performed using the `igl::signed_distance` function, which is a robust and efficient method for this purpose.

The main functionality of the `InnerVoidMesh` class is its ability to iteratively carve the inner mesh to adjust the COM. The `carveInnerMeshStep` method performs a single carving step by defining a cutting plane based on the current and target COM. This plane is oriented to slice through the mesh in a manner that directs the carving towards achieving the desired COM. The voxels are then sorted based on their distance to this cutting plane, and the furthest voxels are removed in each iteration. The removal process involves marking the voxel as unfilled and updating the current COM by recalculating the mass distribution of the mesh.

The `convertToMesh` method converts the `voxelized` representation back into a mesh format, which is essential for visualizing the carved inner structure. This method iterates through the voxel grid, triangulating the faces of the voxels that are part of the void space. The triangulation is handled by the `appendCommonSide` function, which ensures that the faces of neighboring voxels are properly connected, creating a coherent mesh representation of the inner void. This process involves checking the direction of the voxel faces to maintain consistency and avoid rendering issues.

The optimization process is controlled by the `isOptimized` method, which checks whether the COM is within a voxel distance from the target or if a significant portion of the mesh has been removed. This check prevents over-carving and ensures that the algorithm terminates once the balance criteria are met. The `carveInnerMesh` method orchestrates the entire carving process, repeatedly calling `carveInnerMeshStep` until the optimization criteria are satisfied.

Supporting functions such as `getBinIdx` and `getNeighborsIdx` are used to manage voxel grid indices, ensuring that the voxels are correctly positioned within the grid. These functions handle the translation of real-world coordinates to voxel grid indices, which is critical for accurately mapping the mesh to the voxel grid.

3.3 Managing 3D Array Data

For managing 3-dimensional (3D) data arrays efficiently, the `list3d.h` header file defines a template class `List3d` . This task is important for handling 3D grids, which are common in various computational geometry and graphics applications, such as voxel grids, simulation domains, and more. The `List3d` class provides a convenient and efficient way to manage and access elements in a 3D array. Unlike standard nested `std::vector` containers, which can be cumbersome and inefficient due to multiple levels of indirection, `List3d` stores all elements in a single contiguous memory block. This structure is particularly beneficial in performance-critical applications where spatial locality and memory access patterns significantly impact performance.

```
class List3d {
public:
    List3d(size_t d1=0, size_t d2=0, size_t d3=0) :
        d1(d1), d2(d2), d3(d3), data(d1*d2*d3)
    T & operator()(size_t i, size_t j, size_t k) {
        return data[i*d2*d3 + j*d3 + k];}
    T const & operator()(size_t i, size_t j, size_t k) const {
        return data[i*d2*d3 + j*d3 + k];}
    void resize(size_t d1, size_t d2, size_t d3) {
        this->d1 = d1;
        this->d2 = d2;
        this->d3 = d3;
        data.resize(d1 * d2 * d3);}
private:
    size_t d1,d2,d3;
    std::vector<T> data;
};
```

Listing 2: Managinf 3D Array Data

The `List3d` class encapsulates a 3D array with dimensions `d1`, `d2`, and `d3`. It provides several functionalities. The class can be initialized with specified dimensions, and it allocates a single contiguous block of memory to hold all elements. Overloaded `operator()` allows for easy access and modification of elements using 3D indices `(i, j, k)`, making the interface intuitive and similar to native array access. The `resize` method allows the dimensions of the 3D array to be changed dynamically, resizing the underlying data storage accordingly.

The class uses a single `std::vector<T>` to store all elements. This approach ensures that all data is stored contiguously in memory, improving cache efficiency and access speed. The elements are accessed using a computed index that maps the 3D indices to the corresponding position in the 1D vector. The `operator()` methods (both `const` and `non-const` versions) provide element access. The index in the 1D vector is calculated as `i*d2*d3 + j*d3 + k`, effectively flattening the 3D indices into a single index. The implementation assumes that the caller ensures the indices are within bounds, as there are no explicit bounds checks, maintaining access speed. The `resize` method adjusts the dimensions of the 3D array and resizes the underlying vector to accommodate the new size. This method allows for flexible adjustment of the data structure as needed by the application.

3.4 Utils File

```
void createAlignedBox(
  const Eigen::MatrixXd& V,
  Eigen::AlignedBox3d& box);

void transformVertices(
  const Eigen::MatrixXd& V,
  const Eigen::AlignedBox3d &newOrientation,
  bool keepSize,
  Eigen::MatrixXd& Vout);

void transformVertices(
  const Eigen::MatrixXd& V,
  const Eigen::AlignedBox3d &newOrientation,
  Eigen::MatrixXd& Vout);

void createVoxelGrid(
  const Eigen::MatrixXd& V,
```

```
  Eigen::MatrixXd& centers,
  Eigen::MatrixXd& corners,
  Eigen::Vector3i& dimensions,
  Eigen::Vector3d& voxelSize);

void alignToAxis(
  const Eigen::MatrixXd& V,
  Eigen::MatrixXd& alignedV);
```

Listing 3: utils.h

For performing common geometric operations, the ‘`utils.h`‘ header file is implemented. It provides a suite of functions that utilize the Eigen library to handle complex 3D transformations and manipulations efficiently. One such function is `createAlignedBox`, which constructs an `Eigen::AlignedBox3d` from a matrix of vertices. This aligned box encapsulates the bounding box of the given vertices, providing a simple and efficient way to represent the spatial extent of a 3D object. This is crucial for tasks like collision detection and spatial indexing, where understanding the bounds of an object is fundamental.

To compute voxel grids that enclose a given mesh, the `createVoxelGrid` function outputs the centers and corners of the voxels, the grid's dimensions in voxel count, and the exact size of each voxel. Voxel grids are vital for volumetric data processing, enabling applications such as volumetric rendering and physical simulations. By converting complex mesh geometries into a regular grid format, this function facilitates easier processing and analysis, which is essential for operations that require a uniform representation of 3D space.

For this task, I implemented the `transformVertices` function to scale, rotate, and translate a mesh based on a specified destination bounding box (`newOrientation`). This function comes in two variants: one that maintains the size of the mesh and another that scales it to fit the new bounding box. This flexibility is crucial for normalizing 3D models or fitting them within a particular space, such as the build volume of a 3D printer. Additionally, I used the `alignToAxis` function to ensure that meshes are aligned to primary axis planes without altering their size. This standardizes the orientation of models, making subsequent operations, such as slicing for 3D printing or aligning multiple objects within a scene, more consistent and predictable. These functions collectively support the precise control and manipulation of 3D models, ensuring they meet specific requirements within the project.

```
#include <Eigen/Dense>

double center_of_mass(
    const Eigen::MatrixXd &V,
    const Eigen::MatrixXi &F,
    Eigen::Vector3d &center);

void center_of_mass(
    const Eigen::MatrixXd &V,
    const Eigen::MatrixXi &F,
    double mass,
    Eigen::Vector3d &center);
```

Listing 4: Center of Mass

4 Results

The outcomes of the Make it Stand project demonstrated that successfully met the project's main objective. The implementation effectively replicated the key findings of the original paper. By adjusting the mass distribution within 3D objects, the objects were able to stand upright without any external support, validating the effectiveness of the balancing algorithm. After accurately calculations and implementing the center of mass, producing results that closely matched those reported in the original paper. The simplified implementation maintained the core functionalities and yielded comparable results, making it more easier to understand.

A detailed comparison with the original paper showed that the balanced shapes generated by my implementation were nearly identical to those presented in the original work. Minor differences were noted, likely due to variations in implementation details and numerical precision. Performance-wise,

the implementation was efficient, with the use of libigl and Eigen libraries, although some opportunities for further optimization were identified.

Validation through test cases, including simple and complex geometric shapes, we can confirmed the robustness of the balancing algorithm. Simulations of the balanced shapes provided visual confirmation of their stability when fabricated. Examples included balancing a spheres, humpty, which stood upright after mass distribution adjustments, and a sculpture, which became stable and could stand on its own after balancing. These results underscore the success of the project in replicating and simplifying the original work while maintaining accuracy and performance.

A Appendix: System & Dependencies

- Visual Studio 2022 (Windows 11)
- CMake
- Libigl
- Eigen

A Appendix: Steps to Generate Mesh Models

Follow below steps to generate balanced models:

- Select the object file to mesh

  To select the file to generate the mesh models, you can do it either by

  - Giving the file path in the `main.cpp` manually , *or*
  - Selecting the file from the program 4, `Mesh > Load > {Select the object file}`

- Select the Point to Balance
  - After loading the object file, select the point where you want to move the CoM of the object by clicking on the object 5.
- Select the Orientation
  - Adjust the X, Y and Z Axis (roll, pitch and yaw) to make it balanced.6
- Select Generate Balanced Model
  - After the finishing the orientation steps, you can click on "`Start Meshing` 8to generate the meshed model. After the meshing is finished, select the `Start Balancing` 9(Note that it may take some time). To save the balanced model, you can select `Save` in the `Mesh` from the main menu, give the path and file name and extension.

A Appendix: Reference Images from Balacing Model

The followings are the visual implementation of generating balanced models.

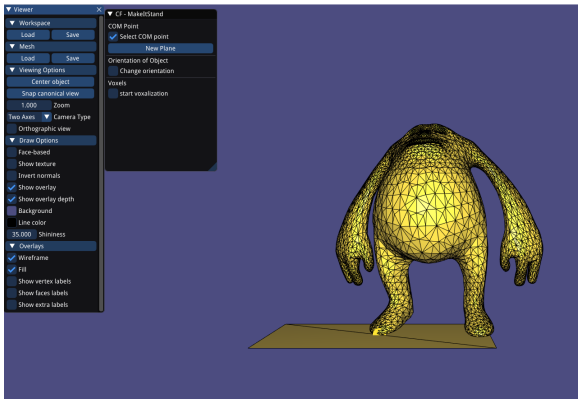


Figure 4: Main

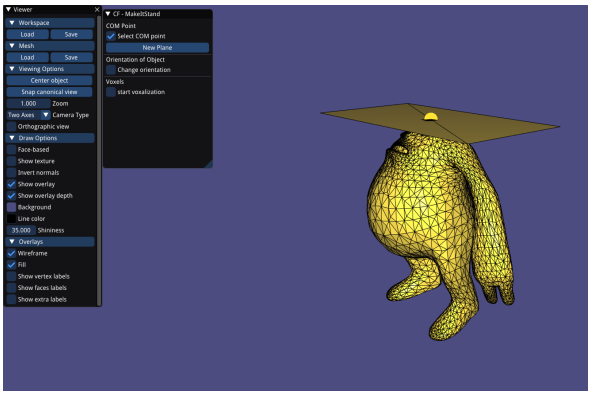


Figure 5: Center of Mass

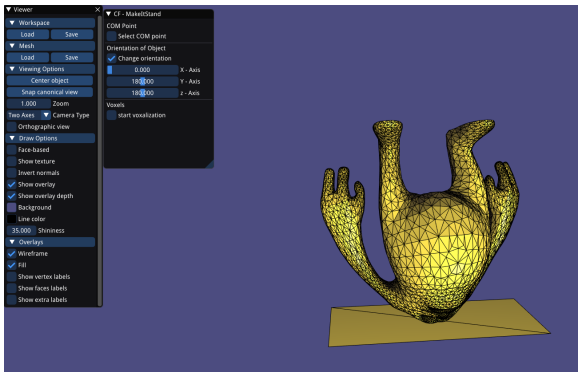


Figure 6: Orientation

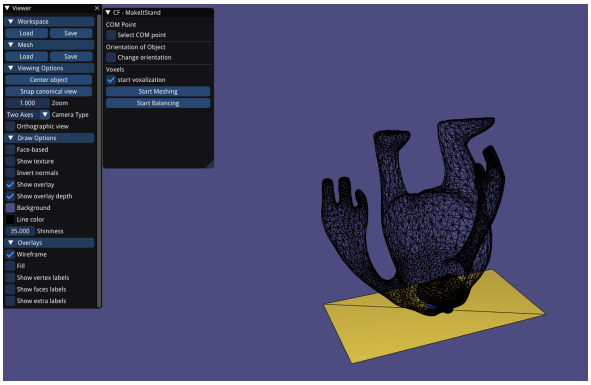


Figure 7: Surafce Voxalization

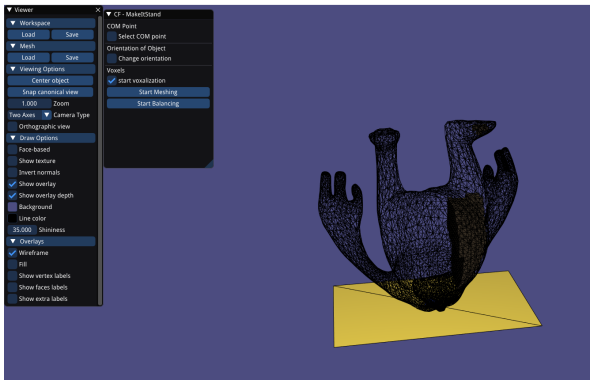


Figure 8: Meshing

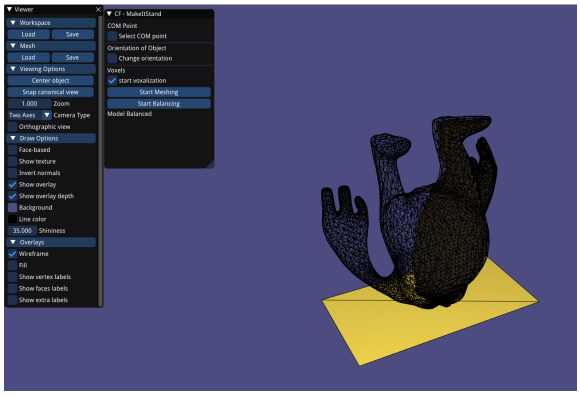


Figure 9: Balancing

References

[1] Romain Prévost, Emily Whiting, Sylvain Lefebvre, Olga Sorkine-Hornung, ETH Zurich, INRIA. Make It Stand: Balancing Shapes for 3D Fabrication, Vol. 32, Article 81, pp: 1-10 (July 2013). DOI: `http://doi.acm.org/10.1145/2461912.2461957`.

[2] Libigl: A Simple C++ Geometry Processing Library. `http://libigl.github.io/`

[3] Eigen: A C++ Template Library for Linear Algebra. `http://eigen.tuxfamily.org`

[4] Libigl Tutorial. `https://libigl.github.io/libigl/tutorial/`