



**University of Arkansas – CSCE Department
Capstone II – Final Report – Fall 2021**

In-House Packing Engine for MARSHALLTOWN

Carey Lawrence, Akhila Parvathaneni, and Evelyn Smith

Abstract

In this paper we outline the in-house packing engine for MARSHALLTOWN to improve shipping costs. This packing engine will optimize packaging smaller items in the most appropriate larger box as well as the stacking of all items on a pallet. This packing engine will take into account individual box dimensions, weight, space limitations from the client and shipping company and the vast array of potential solutions. In order to achieve this, we are starting with a space optimization program and test databases provided by MARSHALLTOWN.

We started with the front-end development to get the user interface working correctly. The next step was to take each of the additional limitations of packing. Moving this software in-house provides many benefits. The first benefit of our project is creating a pallet optimization that will take into account weight instead of just box dimensions. Additionally, MARSHALLTOWN will now have the ability to adapt the program as needed and add new features as new limitations present themselves. Lastly, it helps pack the same amount of area more efficiently and therefore reduce overall shipping costs.

1.0 Problem

MARSHALLTOWN needs a packing engine that provides the warehouse workers the best options for packing items requested by the client. Before going any further, let us define what we mean by “packing engine”. For MARSHALLTOWN the packing engine does two things. The first problem is finding the best way to pack the smaller items into a larger, more portable box in order to save money on shipping costs, but not too large of a box that consumes more airspace than needed. The second half of the problem is taking the larger packed boxes and larger items in general to pack them onto a pallet of which has size limitations specified on the client’s end.

As of now, they have a packing engine already in place. The issue with this is that the service is outsourced from another company. Meaning that they are not really sure how the program is working. MARSHALLTOWN would like to move the packing engine in house to have a better idea of how the code is working, possibly make it more efficient and to add on other requirements as needed.

2.0 Objective

This project has multiple objectives. The first being to create a packing engine that takes smaller items and the different available box sizes to find the most efficient way to pack the smaller

items. In this case efficient takes on the meaning of containing the least amount of open space in a box. We want to fill the box as full as possible with the items given and choose the appropriate box to pack in. The next objective of this project is to efficiently palletize all of the requested products within the height and weight limitations of the client and also the delivery service. This is done to minimize additional fees from both the client and delivery service as well as take down the entire cost of delivery by taking up less space in general.

3.0 Background

3.1 Key Concepts

Some key technologies that are relevant to this project are C#, .NET Standard, Blazor, and SQL. C# is an object-oriented language developed by Microsoft that has similarities to Java. C# enables users to develop Web Applications. The .NET Standard framework is multi-platform which allows for other languages such as C# to use it. .NET can provide services specific to building the application such as accessing the time. There is another tool known as ASP.NET that allows the user to develop web programs. We will be using .NET as the backend for our project.

For the front-end, we will be using Blazor which is an open-source framework that allows the user to build interactive web applications using C#. The name itself combines “Browser” and “Razor” and Blazor supports the performance of the client-side views. More about how Blazor is used specifically for our project can be found in a later section of this report.

Another language that is relevant to our project is SQL. SQL or Structured-Query Language is language used in programming to establish communication with the database. The SQL statements are used to obtain specific pieces of data from the database using certain conditions. SQL is also used to update or modify the database depending on the situation. In general for this project we will be connecting to the SQL database using linq2db library from .NET to evaluate data.

4.0 Design

4.1 Requirements and/or Use Cases and/or Design Goals

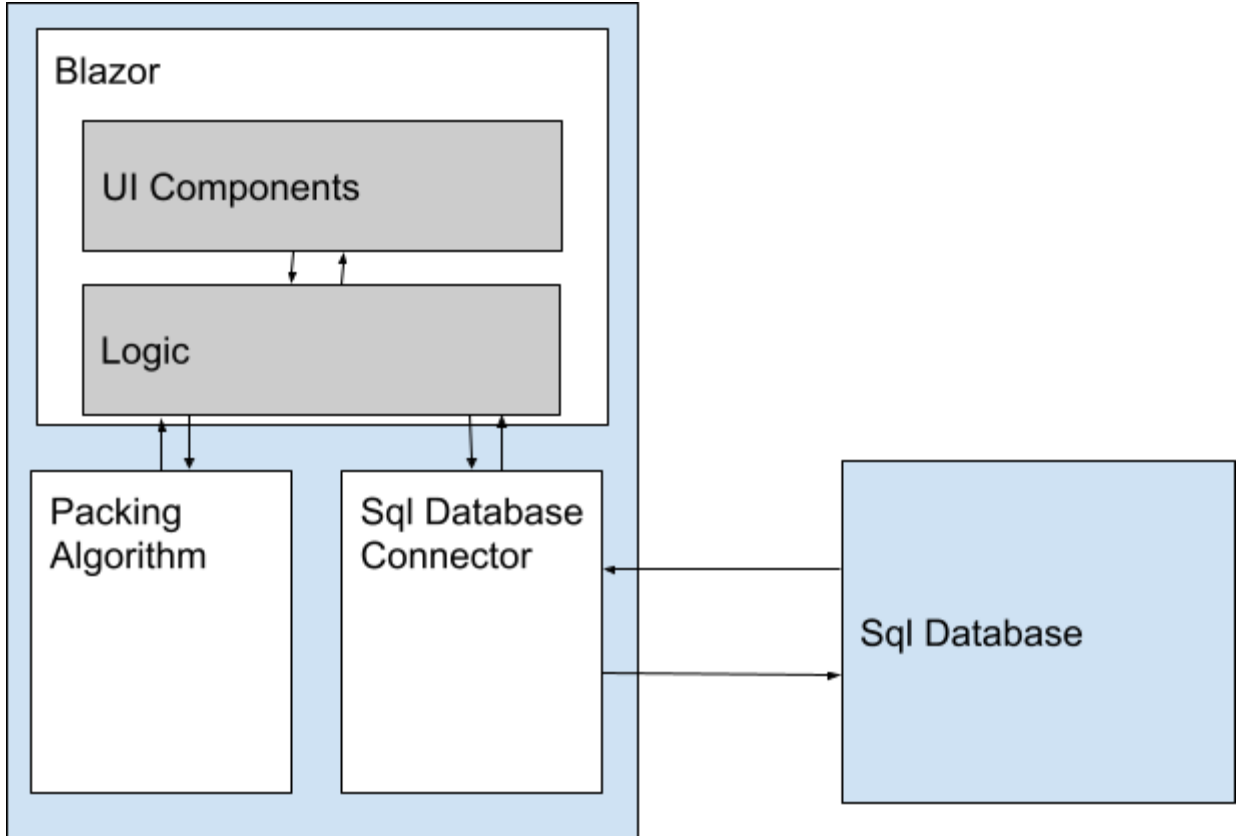
As mentioned, there are two different jobs that this application will serve. The first is selecting appropriately sized boxes to pack mixed cases and how said boxes will be packed. The second will be to determine the most efficient way to pack a pallet given its dimensions and what’s to be packed on said pallet. The front-end requirement for this project will be to create an interface that works intuitively as possible to provide the user with an easy way to enter any items, boxes, or dimensions that will restrict the order.

Because of the products that MARSHALLTOWN produces, many additional requirements will factor into the algorithm we will be designing. At MARSHALLTOWN, many products ship in case quantities or are mixed cases with many different items inside. A requirement of this algorithm is to prioritize heaviest cases at the bottom. With MARSHALLTOWN, the method of

weight stacking used is noted as the ABC method where and A cannot be stacked on a B or C class item, a B item can be stacked on an A but not a C, and finally, C class items can be stacked atop any of the classes. Additionally, many products must be shipped following a certain orientation, therefore the algorithm may not be allowed to alter the orientation of certain items, even at the cost of efficiency. MARSHALLTOWN products such as industrial rakes have abnormal, triangle-shaped boxes and are packed with an exposed handle with a length of around 60". A final requirement given to us by MARSHALLTOWN was to try to pack items with items of their kind as much as possible to reduce confusion with production workers and minimize the time spent searching for different items. These requirements cause great difficulty when packing and are what makes this project with MARSHALLTOWN so unique.

4.2 Detailed Architecture

For the front-end design of our webapp, we plan to use the Blazor framework. Blazor is an extension of the .NET developer platform that allows for the creation of Razor components. These components then allow for the integration of HTML and C# code to create web pages. Unlike MVC apps, Blazor does not follow a request/response model, but instead centers around client-side UI logic where the logic can then make further calls to the .NET Standard backend [1]. This UI is meant to be intuitive to enter data into, and offer clear explanations of how the results of a call to the packing algorithm should be interpreted. This front-end utilizes a JavaScript library called THREE.js to perform the modeling and display the items in the order they're packed for an easier visualization of the packing process. As suggested, THREE.js allows for us to produce 3-d objects and map them to a defined view as it interfaces with WebGL to provide an easier development process than simply utilizing WebGL.



This project will utilize a sql database of sample data used at MARSHALLTOWN. To access, manipulate, and create the stored procedures for the database, we will be using SQL Server Express as our software of choice.

Our connector to the SQL database will utilize the .NET Library linq2db to send and retrieve the data to and from our SQL database. The data acquired through this method will be used to validate items and box sizes, as well as to retrieve information regarding specific customers' needs for pallet packing.

The final large component of our project is the packing algorithm. While we have been given a starting C# solution for this project, we will find the majority of the changes in this project being made in this part of our work. The packing algorithm is what this project largely revolves around, with the end goal of this specific component being to accurately compute the most cost efficient way to pack boxes and pallets. This component will communicate with our Blazor project to take in the data regarding what is to be packed, and then to return a model of how those items should be packed.

4.3 Tasks

As this is still in the earliest phases of development, the tasks listed are likely more general than what will end up becoming of them as we progress.

1. Determine and create script for SQL Database creation
2. Learning new software and frameworks
3. Testing current project and determining any additional shortcomings
4. Adjusting database schema to fit requirements
5. Creating and inserting sample set(s)
6. Familiarization with algorithm
7. Adjusting the algorithm for heaviest at the bottom
8. Adjusting the algorithm to work with items that cannot be reoriented
9. Making packing algorithm work for both pallet packing, and box packing...
10. UI and visualization rework
11. Edge case testing application/Catchup
 - a. Debugging minor issue with items that can't be reoriented
12. Final deliverables (Paper, website, poster, etc.)

4.4 Schedule

| Tasks | Dates |
|--|-------------|
| 1. Determine and create script for SQL Database creation | 9/13-9/22 |
| 2. Learning new software and frameworks | 9/13-9/22 |
| 3. Testing current project and determining any additional shortcomings | 9/22-10/1 |
| 4. Adjusting database schema to fit requirements | 10/1-10/4 |
| 5. Creating and inserting sample set(s) | 10/4-10/8 |
| 6. Familiarization with algorithm | 10/8-10/15 |
| 7. Adjusting the algorithm for heaviest at the bottom | 10/15-10/22 |
| 8. Adjusting the algorithm to work with items that cannot be reoriented | 10/22-10/29 |
| 9. Making packing algorithm work for both pallet packing, and box packing... | 11/5-11/12 |
| 10. UI and visualization rework | 11/12-11/19 |
| 11. Edge case testing application/Catchup | 11/19-12/3 |
| 12. Final deliverables (Paper, website, poster, etc.) | 12/3-12/8 |

4.5 Deliverables

For this project, we will have two main coded deliverables:

1. C# Pallet Packing solution containing 4 projects...
 - a. Blazor front-end with .html layouts, .razor components, startup settings, and other classes used directly from the Blazor front-end
 - b. SQL Database connector using .NET's linq2db library with methods to get and map the data returned from our SQL calls
 - c. Packing Algorithm project with all necessary C# files
2. SQL Database scripts
 - a. Script for setting up the database we will be working with
 - b. Script for populating the database
 - c. Script(s) for creating any stored procedures we may use

4.5.1 Pallet Packing C# Solution

With the starter project we were given, some of the design decisions regarding the packing algorithm had already been set for us. We were given a basic web interface that utilizes the EB-AFIT algorithm developed by Erhan Baltacioglu of the US Airforce in 2001. This algorithm essentially takes in a list of items with dimension attributes, as well as a container, then outputs an ordered pack list of the most efficient method of packing said container. That list is then taken in by the THREE.js component of the front-end to provide an incrementable, 3-d representation of the packing result that can be used on the production floor. The final major part of the C# Solution we implemented was the SQL connector that populates the information from the database, including the possible container sizes and the order to be processed.

4.5.1.1 SQL Connector

With the Linq2Sql library we implemented in this project, we were easily able to achieve information retrieval for the project. To start, we had to create a .tt file with the connection information regarding our database. With Linq2Sql, all we then had to do was save the file and a C# model of our database was generated along with all its tables and attributes.

```
[Table(Schema="dbo", Name="ItemsToPalletize")]
4 references
public partial class ItemsToPalletize
{
    1 reference
    [Column, NotNull ] public string  CustOrderNo  { get; set; } // varchar(6)
    1 reference
    [Column, NotNull ] public string  Alias         { get; set; } // varchar(20)
    1 reference
    [Column, NotNull ] public int     QTY          { get; set; } // int
    0 references
    [Column, NotNull ] public string  BoxNumber     { get; set; } // varchar(21)
    0 references
    [Column, NotNull ] public string  BoxName       { get; set; } // varchar(20)
    1 reference
    [Column, NotNull ] public decimal BoxLength    { get; set; } // decimal(12, 2)
    1 reference
    [Column, NotNull ] public decimal BoxWidth     { get; set; } // decimal(12, 2)
    1 reference
    [Column, NotNull ] public decimal BoxHeight    { get; set; } // decimal(12, 2)
    0 references
    [Column, NotNull ] public decimal BoxWeight    { get; set; } // decimal(12, 2)
    0 references
    [Column, Nullable] public int?    PalletNumber  { get; set; } // int
    0 references
    [Column, Nullable] public string  PalletName    { get; set; } // varchar(50)
    [Column, Nullable] public string  OSCOStatus   { get; set; } // varchar(50)
    1 reference
    [PrimaryKey, Identity ] public int  ITPID      { get; set; } // int
    1 reference
    [Column, NotNull ] public bool   CanBeFlagpole { get; set; } // bit
    1 reference
    [Column, NotNull ] public char   CharWeight   { get; set; } // char(1)
    1 reference
    [Column, NotNull ] public bool   AbnormalShape { get; set; } // bit
}
```

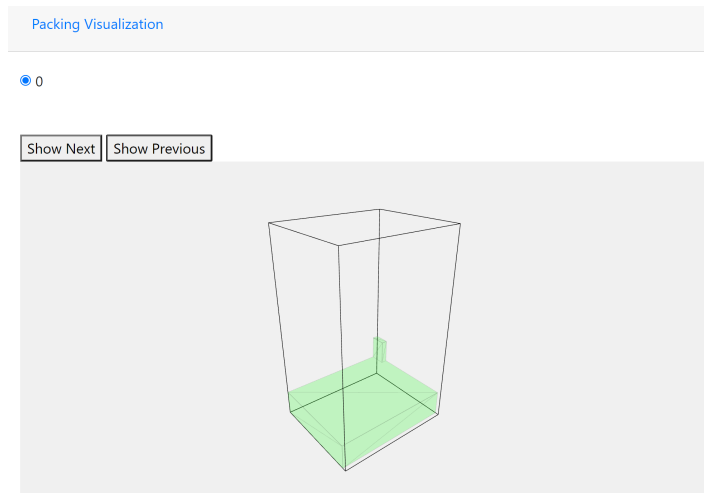
With this model generated, we then had to create a class that would interface with outside C# packages and actually populate the item and container models. This class included various methods, including GetContainers() which retrieves and maps the container information; GetItemsToPack() which returns the items in a given order, grouped by item name; and GroupItems() which groups the item by name and adjusts their quantity as needed per grouped item. With this implemented, we have all the information required from the database to start packing pallets.

4.5.1.2 Packing Algorithm

The Packing algorithm itself is where the bulk of the project work was performed. As mentioned, the EB-AFIT algorithm is very accurate at packing containers and maximizing the space. It doesn't, however, take into account any real-life conditions that a manufacturer might require when shipping items. The base algorithm essentially takes a layer-by-layer approach to packing. It tries to fill as flat and efficient a layer as possible, checking different variations of how the items may be placed to maximize the efficiency. Then, if a layer is not flat, it recurses through the gaps in an attempt to fill them and maximize those pockets as best as possible. It does this across 6 possible container variants and chooses the one deemed most efficient by the program at the end to pack. It then uses that best variant to call the Report method to run through the packing iterations again using only the most efficient container orientation to recalculate and do the actual packing of the layers, before returning the list of items in packed order with their appropriate locations set.

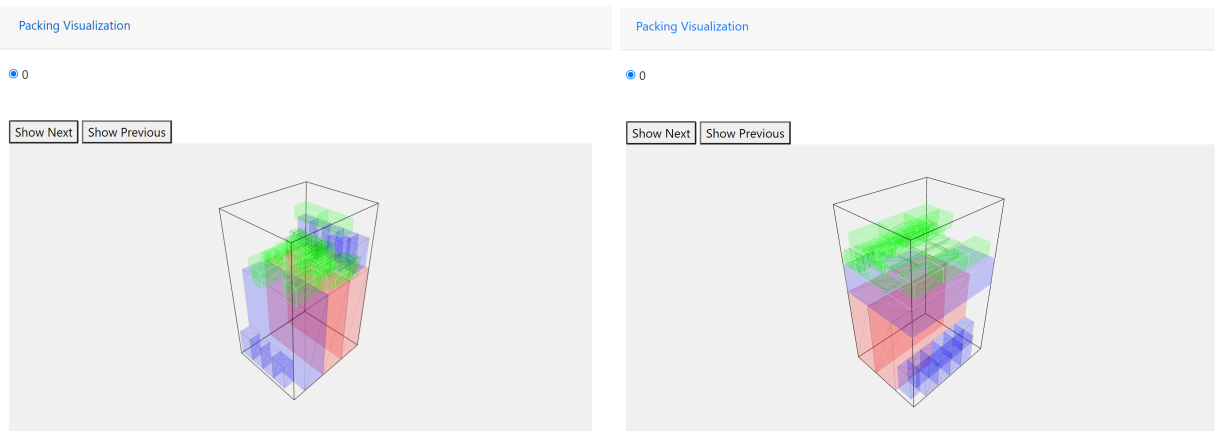
Grouping Items

To group our items into complete layers as best as possible, we didn't end up having to change the EB-AFIT algorithm itself, but rather how the algorithm was entered. To do so, we utilized a method named LayerItems that takes in one item type at a time. It then parses this item type into a list of that item that is the length of the total quantity of that item in the current order. At this point, that list is sent into our EB-AFIT algorithm to return our packing results for that specific item. We can then perform a little math on the packing result to find out if a full layer was packed and if there were any leftover items that didn't constitute a full layer. If we were able to create a perfect layer of those items, we group them as a layer of that item type and then return any leftover items. While initially this method works, as we proceeded with the project we found there were some shortcomings with the method as a layer of items with weight class C will always be required to be atop or next to items of class A or B, meaning there is a large possibility for that layer to be placed somewhere that the algorithm thinks one item of full size is possible but isn't when broken down. Pictured below is a layer of items that has been grouped together as described. The layer consists of the individual boxes shown atop the conglomerate layer.



Fixed Item Orientation

To tackle the issue of fixed item orientation, we had to start by altering the initial schema of the database to include a column of the type 'bit' specifying if the item can be packed flagpole. We then had to alter the attributes of the item class within the C# project to add a boolean that will be mapped to the corresponding bit of the database's table. At this point, it was a matter of breaking down the algorithm to find out where it is that we should determine if an item can be packed following a certain orientation or not. The solution to this ended up being in the FindBox method where we cycle through the array of items to find if and what orientation of the current item will fit within the given area by calling AnalyzeBox to find the best fitting orientation. Unintuitively enough, the coordinate plane system used during visualization regards what would truly be the z axis as the y axis, thus causing some confusion and issue during the implementation when trying to restrict an item's length from being aligned along what would otherwise be considered the z axis.



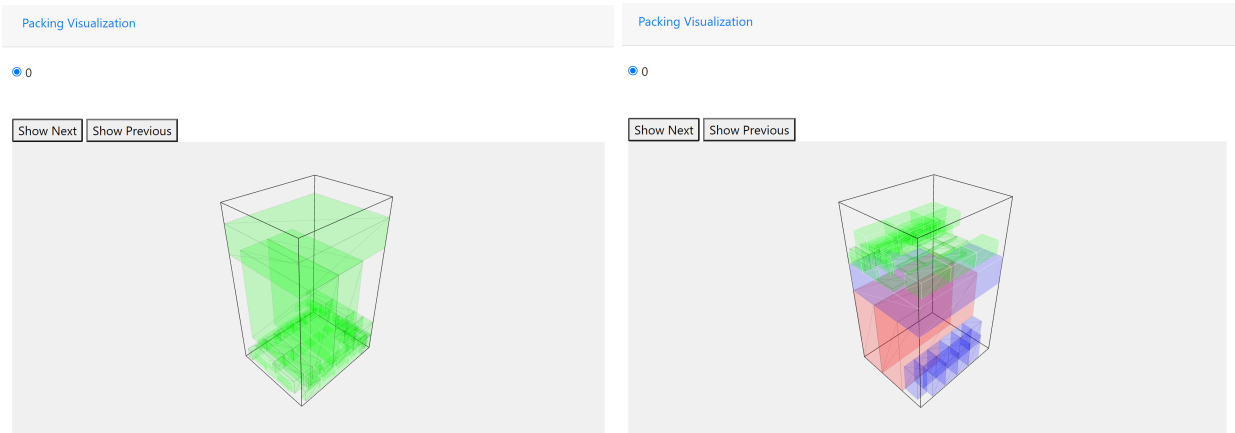
Note: Boxes with black outline can be packed vertically, boxes with white outlines cannot

Pictured above is an example of how restricting the orientation of one item (the large blue box) can totally restructure an entire packing solution. On the left, the box is allowed to be placed with its length vertical, however on the right we have altered the dataset so that option is no longer possible.

A B C Item Weighting

The ABC item weighting turned out to be a more in-depth change than those mentioned. With this method of item weighting, we had to start by assigning a weight of A, B, or C to every item within the database by altering the schema to include a column named "CharWeight". Following this, we had to update the item model to have a partner attribute for that value to be mapped to, which we called "WeightDef". From there, the method we pursued for implementation revolved around editing the contents of the EB-AFIT algorithm. To do this, on entering the initialize function within the algorithm, we split the `_itemsToPack` List that contains the full list of every individual item to pack into three arrays: `_itemsToPackA`, `_itemsToPackB`, and `_itemsToPackC`. These lists are, as named, all of the A, B, and C items in the order. We also retained a copy of the original `_itemsToPack` List as well as of the lengths of each of the Lists we created. We then set `_itemsToPack` equal to A, B, or C, with that priority order, depending on whether the

respective List has any members belonging to it. Because of the design of the EB_AFIT algorithm, all of these variables were of the private access modifier, meaning any of the methods within this algorithm could access them, thus making this process a bit more difficult than just splitting up that array and packing each one. In order to achieve a correct implementation with all of these private variables, we had to go through the entirety of the algorithm and add various checks and triggers so that if the `_itemsToPack` list was being looped through anywhere, it must reassign it the the appropriate A, B, or C list after finishing a number of iterations equal to the length of the A, B, or C list, as well as preserve the integrity of any changes made to the contents of the A, B, and C lists. In other places, such as `ExecuteIterations` and `Report`, the entirety of both lists are reset to start from scratch with dimension checking. In these areas, we essentially had to re-instantiate each list based on the copy of `_itemsToPack` we created early on. Given the size and complexity of the EB-AFIT algorithm, this process ended up being quite tedious and took a significant amount of time to complete as any incorrect alteration to the algorithm would cause items to be unpacked, infinite loops, or just wouldn't pack correctly.



Note: Red boxes are class A, blue boxes are class B, and green boxes are class C

As pictured above are instances of container packing where everything is treated with the same weight, versus the case where everything is weighted according to the ABC method. While the unweighted method is definitely more space saving, it should be noted that the algorithm created the base using only the smallest items, then stacked the three largest boxes atop them in a way that would likely damage MARSHALLTOWN goods. On the right, one might speculate that there is room for improvement by stacking C weighted (green) boxes along the edge atop the blue boxes, however this would violate the rules set in place with container packing where we are specifically required to never layer a B box above a C box.

Container Packing

The objective of the container packing portion of this project was to pack smaller items into a larger box for easier shipping purposes. This should work in much the same way that the palletizing portion of the algorithm did. However, the main difference was that we needed to find an optimal packing solution and pick the container thereafter instead of having predetermined packing dimensions. The container packing utilizes the EB-AFIT algorithm just in an opposite manner. The possible small item configuration is tested within the limitations of each of the possible larger container sizes that MARSHALLTOWN has at their disposal. The algorithm

should then choose which larger container has the least amount of air space within the box with all items packed in the most space-saving configuration. The container with the least amount of air space is then chosen and that box goes into the database to be properly palletized with the rest of the order.

It is important to note that this portion of the project is not fully complete at this time.

4.5.1.3 Blazor with THREE.js Front-end

The basic front-end is quite simple and consists firstly of a text box where the user can input the desired customer order number to retrieve the items on the order from the database via the “Get Items for Order” button. Accompanying that is the button that signals for the work to begin, entitled simply “Pack Pallets”.

The following few tiles are expandable and display various information regarding the order and the algorithm. When expanded, Items shows all information regarding the items on the current order, including Name, Length, Width, Height, Qty, Weight, and if the item can be packed vertically (also considered flagpole). Containers tile displays information about the various boxes MARSHALLTOWN uses to pack smaller items inside of and shows the boxes’ Name, Length, Width, Height, and Volume. These containers are used for the items to containers part of the packing problem posed to us and having their information on hand is valuable on the production floor. The next tile, Pack Results, is more useful to developers. It has information about the algorithm used (in our case, always the EB-AFIT algorithm), the pack time in ms, the percentage of the container used, and how many items we were and weren’t able to pack.

[Items](#)

| Name | L | W | H | Qty | Weight | Can be packed floppable? |
|-----------|-------|-------|-------|-----|--------|--------------------------|
| 28167x6 | 14.30 | 5.70 | 8.10 | 10 | B | Yes |
| 28168x6 | 14.60 | 6.50 | 8.20 | 8 | C | No |
| 28169x6 | 9.00 | 1.40 | 6.40 | 10 | C | Yes |
| 28169x576 | 48.00 | 14.10 | 40.00 | 2 | A | Yes |
| 28170x6 | 9.80 | 1.70 | 6.20 | 10 | C | No |
| 28170x504 | 48.00 | 13.00 | 40.00 | 1 | B | Yes |
| 28171x6 | 9.20 | 2.50 | 6.20 | 10 | C | Yes |
| 28172x6 | 9.00 | 1.20 | 5.10 | 10 | C | No |

[Containers](#)

| Name | L | W | H | Volume |
|--------|--------|--------|-------|-----------|
| WS1075 | 37 | 11 | 9.3 | 3785.1 |
| SN3300 | 21 | 14 | 16 | 4704 |
| SN3433 | 31 | 16 | 11.5 | 5704.0 |
| WS390 | 17 | 13.625 | 15.5 | 3590.1875 |
| WS391 | 21 | 11 | 12.25 | 2829.75 |
| SN3434 | 39 | 20 | 11.5 | 8970.0 |
| WS399F | 15.25 | 8.75 | 12 | 1601.2500 |
| WS432 | 14.125 | 8.25 | 5 | 582.65625 |

[Pack Results](#)

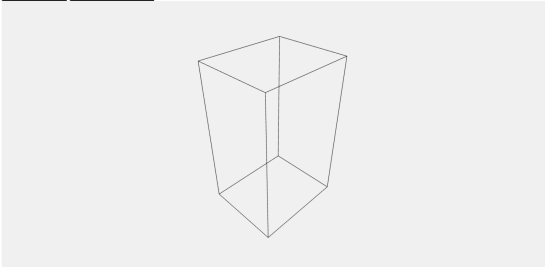
| Algorithm Name | Pack Time (ms) | % Cont. Used | # Items Packed | # Items Unpacked |
|----------------|----------------|--------------|----------------|------------------|
| EB-AFIT | 19 | 69.26 | 61 | 0 |

On packing the pallets, not only is information populated into the Pack Results tab, but a new view created using THREE.js appears with a visualization of the pallet's area with no items packed in it. At this point, the user may utilize the "Show Next" and "Show Previous" buttons to step through the packing of the pallet/container and visualize how the pallet should be packed. These buttons interact with the JavaScript via JS Interop to create and push THREE.js mesh cubes into the render and display them depending on their assigned coordinates as returned from the algorithm.

Packing Visualization

0

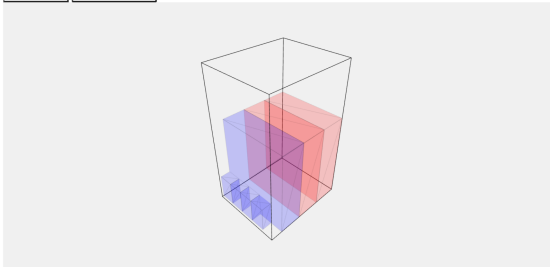
[Show Next](#) [Show Previous](#)



Packing Visualization

0

[Show Next](#) [Show Previous](#)

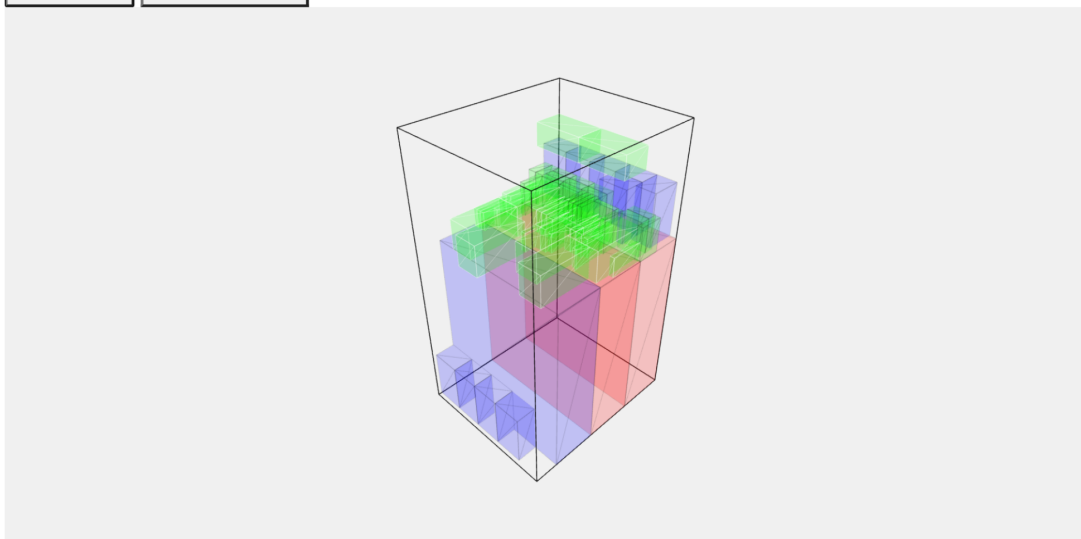


Packing Visualization

0

Show Next

Show Previous



This final image is a completely packed order. Something of note is the coloration of the meshes and why they differ. The items' meshes are assigned colors based on the item's attributes. A red mesh means the item is of weight class A. A blue mesh means the item is of weight class B. Finally, a green mesh means the item is of weight class C. Additionally, these meshes have been assigned wireframes that differ based on whether the item can be oriented vertically or not. For an item where that case is allowed, we set the wireframe to black, otherwise it's set to white, as pictured above.

4.5.2 Database Structure

The database scripting language we used for this project was SQL, as per standard within MARSHALLTOWN. To start our database, we were given a restore file which we used on a local database instance to ensure everything was working with what they gave us. We then used a rather large dataset given to us from MARSHALLTOWN that contained the entirety of one large order to populate our ItemsToPalletize table. While we were given seven total tables to work with, only three are required for our project to work, with only two being used in the current iteration of the project. These tables were ItemsToPalletize, which contained all of the information regarding the items on an order and BoxData, which contained all of the information regarding the sizes of containers we had available and could be used for packing items in. Pictured below are the schemas for said tables.

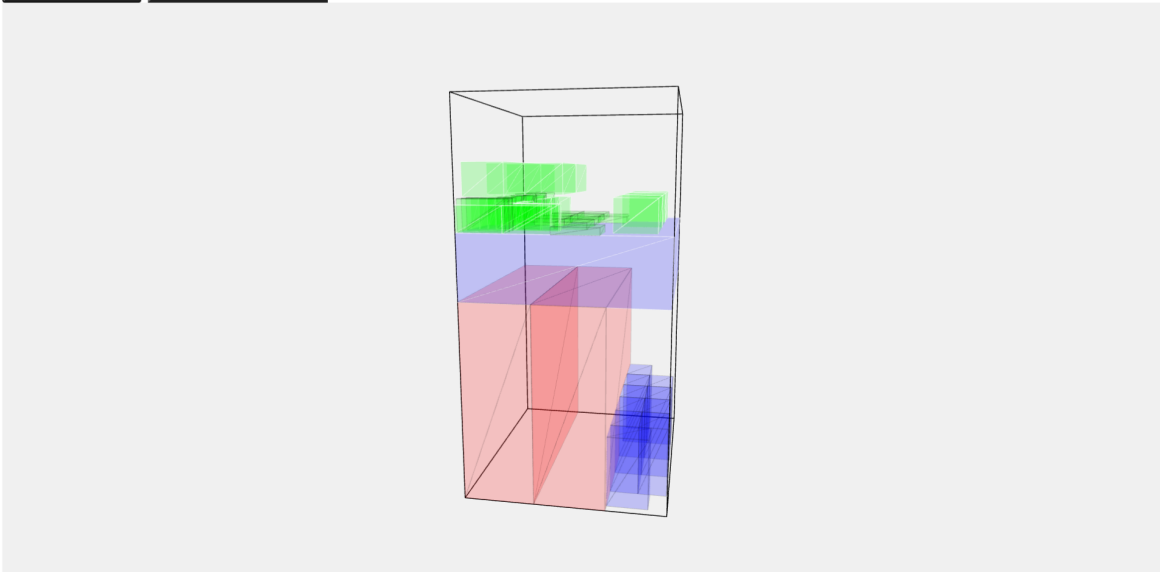
| Column Name | Data Type | Allow Nulls |
|---------------|----------------|-------------------------------------|
| CustOrderNo | varchar(6) | <input type="checkbox"/> |
| Alias | varchar(20) | <input type="checkbox"/> |
| QTY | int | <input type="checkbox"/> |
| BoxNumber | varchar(21) | <input type="checkbox"/> |
| BoxName | varchar(20) | <input type="checkbox"/> |
| BoxLength | decimal(12, 2) | <input type="checkbox"/> |
| BoxWidth | decimal(12, 2) | <input type="checkbox"/> |
| BoxHeight | decimal(12, 2) | <input type="checkbox"/> |
| BoxWeight | decimal(12, 2) | <input type="checkbox"/> |
| PalletNumber | int | <input checked="" type="checkbox"/> |
| PalletName | varchar(50) | <input checked="" type="checkbox"/> |
| OSCOSStatus | varchar(50) | <input checked="" type="checkbox"/> |
| ITPID | int | <input type="checkbox"/> |
| CanBeFlagpole | bit | <input type="checkbox"/> |
| CharWeight | char(1) | <input type="checkbox"/> |
| AbnormalShape | bit | <input type="checkbox"/> |

| Column Name | Data Type | Allow Nulls |
|-------------|-------------|-------------------------------------|
| Name | varchar(50) | <input type="checkbox"/> |
| Length | float | <input type="checkbox"/> |
| Width | float | <input type="checkbox"/> |
| Height | float | <input type="checkbox"/> |
| MaxWeight | float | <input type="checkbox"/> |
| Weight | float | <input checked="" type="checkbox"/> |
| COST_FACTOR | float | <input checked="" type="checkbox"/> |

After getting the database working locally, we then deployed using Microsoft Azure to provide the whole team access to the server and tables needed to make progress on the project.

5.0 Future Work

In the future, we would like to work on building onto the code for abnormal items. Items that are triangular or any other non-cuboid forms may need to be placed on the top. This way those that are already in a cuboid form may maximize efficiency in the bottom to fit in as many boxes as possible. Another thing we would like to work on is refining the packing process so that it is more efficient. There are some items that seem to be floating in air, which is not a realistic solution to this packing engine. We are going to explore ways to go directly from the packing containers to the packing pallets along with the containers. We would also like to make the UI more clear so that it would be easier to identify which boxes are being packed. Perhaps we could have the box selected to be highlighted so the user can easily make out where that item is located in the packing engine. We need to finish container packing as it is important to choose the container with less air space to be inserted into the database in order to be appropriately palletized with the other boxes. There are still some minor layering issues we need to modify and make sure efficiency is maximized.



Seemingly floating boxes, B item C item placement seems like it should pack up the edge of the pallet, however it doesn't because that would layer a B item above a C item.

6.0 Key Personnel

Evelyn Smith – Smith is a senior Computer Science major in the Computer Science and Computer Engineering Department with a minor in Mathematics at the University of Arkansas. She has taken Database Management, Programming Paradigms, Software Engineering, and is currently enrolled in Algorithms. Smith has been working with MARSHALLTOWN since June of 2021 and has had the opportunity to work with all of the frameworks and libraries used in this project in a professional setting. Responsible for SQL Database and Blazor UI.

Carey Lawrence – Lawrence is a senior Computer Science major in the Computer Science and Computer Engineering Department with a minor in Mathematics at the University of Arkansas. She has completed relevant courses including: Algorithms, Database Management, Programming Paradigms, and Software Engineering. Additionally, Lawrence has done research into machine learning as well as full-stack development for multiple web based applications through Credera Consulting. Responsible for developing our algorithm to work with pallet restrictions.

Akhila Parvathaneni – Parvathaneni is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. Parvathaneni has already obtained a Bachelor Degree in Biology and a minor in Mathematics. She has completed relevant courses including: Artificial Intelligence, Algorithms, Software Engineering, and Programming Paradigms. She is familiar and has experience with iOS/Android Development and Full-Stack development. Responsible for developing our algorithm to work with mixed case restrictions.

Craig Wall, Industry champion – Wall did not provide us with a short biography by the time that this was due.

Jeff Schnieder, Industry champion/IT Director – Schnieder has been developing software professionally for 20 years and previously owned his own company for 8 years before joining MARSHALLTOWN in 2010. Schnieder manages the organization's development and systems teams. We have developed many novel software products for our own use and in 2014 started selling some to other distributors/manufacturers. Since then we have productized a SDK that makes interaction between C# software and ERP systems, mainly Microsoft Dynamics AX based systems, much easier to write. Now we have a software company component in our department and we have many customers in the US and Europe.

1.0 Facilities and Equipment

The facilities and equipment used for our project as mentioned earlier involve the C# code with the packing algorithm, SQL Database connector, and Blazor for the front end.

7.0 References

[1] Architecture comparison of ASP.NET Web Forms and Blazor, <https://docs.microsoft.com/en-us/dotnet/architecture/blazor-for-web-forms-developers/architecture-comparison>

[2] linq2db - Introduction, <https://github.com/linq2db/linq2db/wiki/Introduction>

[3] Baltacioglu, Erhan, "The Distributer's Three-Dimensional Pallet-Packing Problem: A Human Intelligence-Based Heuristic Approach" (2001). Theses and Dissertations. 4563.
<https://scholar.afit.edu/etd/4563>

[4]William Knechtel, 3D Container Packing in C#, <https://github.com/wknechtel/3d-bin-pack/>