TnT | Block Recipes & Mass

Requirements:

- Block recipes
    - Predefined & Shared per block class
    - Free choice of items, order and amounts for designers
    - Functional level
    - Automatically scales with block size
- Static Block mass
    - Based on Block recipe
    - Stable mass across build stages
        - Variable based on missing fractures (?)
- Dynamic Block mass
    - Based on inventory contents
    - Consider resource (oxygen, hydrogen) contents too (?)
- Aggregation
    - Block mass is propagated and applied to grid accurately
    - Fast for frequently updated cases
    - Optimized memory usage
    - Correctly replicated between Client & Server

Static Mass

Computation

Static mass for a block is calculated using 2 values:

- the block mass curve modifier (floating-point factor inputted by designers),
- the block size - represented by the count of occupied grid cells.

Giving us the following formula:

$$Block\ mass\ =\ Curve\ Modifier\ *\ sqrt(Occupied\ Cells)\ *\ log10(Occupied\ Cells)\ +\ Minimum\ Block\ Mass$$

Given that static mass is constant across build stages, it can be computed at *[Init]* time by *CubeBlockDefinition* then stored as floating-point field *CubeBlockDefinition.Mass*.

```
public partial class CubeBlockDefinition : Definition, IValidation
{
   [...]

   public CubeBlockDensityDefinition Density { get; private set; }

   [NoSerialize]
   public float Mass { get; private set; }

   [...]
}
```

CubeBlockDensityDefinition

Having a separate definition for density instead of just a field allows editing and use of density presets independently from the block recipe itself, as well as reusing existing presets for new blocks.

```
/// <summary>
/// Settings used for computing the block's mass relative to its size in
/// occupied cells.
/// </summary>
public partial class CubeBlockDensityDefinition : Definition
{
   /// <summary>
   /// The value computed from the block volume (number of occupied cells)
   /// will be multiplied by this value to compute the final mass.
   /// </summary>
   public float MassCurveModifier { get; private set; }
}
```

Recipes

Computation

Amounts

The recipe for a block is calculated using 4 values:

- the block's recipe percentage for each item,
- the mass of each item,
- the block efficiency (see CubeBlockEfficiencyDefinition),
- the block's static mass (as specified above).

Giving us the following formula:

$$Item\ quantity\ = \frac{Block\ mass * Item\ percentage}{Item\ mass} * \ Block\ efficiency$$

where $Item\ percentage\ = \frac{Item\ count}{Total\ recipe\ item\ count}$

Just like the static mass, each item quantity is calculated at *[Init]* time by *CubeBlockDefinition*. This value is then rounded. This data is stored as lists of *RecipeItem* values (see CubeBlockRecipeDefinition).

```
public partial class CubeBlockDefinition : Definition, IValidation
{
    [...]

    public CubeBlockEfficiencyDefinition Efficiency { get; private set; }

    [HideInInspector]
    [NoSerialize]
    public ImmutableArray<RecipeItem> Items { get; private set; }

    public CubeBlockRecipeDefinition Recipe { get; private set; }

    [...]
}
```

Functional level

After the final recipe amounts have been computed, at *[Init]* time, *FunctionalBlockDefinition* computes its minimum functional build progress based on the computed recipe amounts and the following formula:

$$Min\ functional\ level = \frac{Sum\ of\ critical\ recipe\ \ amounts}{Total\ sum\ of\ recipe\ amounts}$$

Critical items (see CubeBlockRecipeDefinition) are mandatory for a *FunctionalBlockComponent* to start working and they need to be added to the build before non-critical items. As such, the build progress necessary is the ratio of critical items in the recipe.

```
public partial class FunctionalBlockDefinition : CubeBlockDefinition
{
   /// <summary>
   /// Minimum build progress before the block starts working
   /// </summary>
   [NoSerialize]
   public float MinFunctionalBuildProgress { get; private set; }
}
```

CubeBlockEfficiencyDefinition

When each item's amount is computed, *CubeBlockEfficiencyDefinition.Efficiency* is used as a coefficient applied to every item amount - in other words, the lower it is, the less items will be required and the more "efficient" it is. Similarly to CubeBlockDensityDefinition, a separate definition allows for the use of preset efficiency profiles.

```
/// <summary>
/// Defines the efficiency applied to the CubeBlockRecipeDefinition for the
/// CubeBlockDefinition.
/// </summary>
public partial class CubeBlockEfficiencyDefinition : Definition
{
   /// <summary>
```

```
    /// Coefficient of the amounts of all items required by the associated
    /// CubeBlockRecipeDefinition, must be positive.
    /// </summary>
    public float Efficiency { get; private set; }
}
```

CubeBlockRecipeDefinition

The *CubeBlockRecipeDefinition* contains the immutable data defining the necessary items proportions and their criticality for building a *CubeBlockComponent*. For each item, this data is represented by a *RecipeItem*, stacks are sorted.

There needs to be at least one critical item in the recipe. This is enforced through validation by *CubeBlockRecipeDefinition*.

```
/// <summary>
/// Defines the proportions and criticality of items needed to build a given
/// CubeBlockComponent.
/// </summary>
public partial class CubeBlockRecipeDefinition : Definition, IValidation
{
    /// <summary>
    /// List of critical items to build the CubeBlockComponent
    /// </summary>
    public MergeableList<RecipeItem> CriticalItems { get; private set; }

    /// <summary>
    /// List of optional items to build the CubeBlockComponent
    /// </summary>
    public MergeableList<RecipeItem> OptionalItems { get; private set; }

    Task IValidation.ValidateAsync(ValidationContext context, MemberReference
member
    {
      /// Ensuring there is at least one critical item here...
    }
}

/// <summary>
/// Defines the amount of an item in BlockRecipeDefinition.
/// </summary>
public struct RecipeItem
{
    public ItemDefinition Item;
```

```
    public FixedPoint Amount;
}
```

Dynamic Mass

Computation

Server-side only

Inventory components and data are only present on the server and are necessary to determine a block's dynamic mass, thus dynamic mass needs to be computed only ***server-side*** as well. This happens when an *InventoryComponent.InventoryData* is attached to a *CubeBlockComponent*'s Entity, or it is changed or removed.

Aggregation

IDynamicMassProvider & use case of inventories

The block's dynamic mass can be provided by different components on its Entity. Such components providing and computing their dynamic mass must implement interface *IDynamicMassProvider*. The *IDynamicMassProvider.DynamicMassChanged* signal is handled by *CubeBlockComponent*.

```
/// <summary>
/// Interface for components providing a dynamic mass to <see
cref="CubeBlockComponent"/>.
/// </summary>
public interface IDynamicMassProvider
{
    /// <summary>
    /// Returns total dynamic mass caused by the component.
    /// </summary>
    public float DynamicMass { get; }

    /// <summary>
    /// Reports a dynamic mass change on a component of the Entity.
    /// </summary>
    public partial class DynamicMassChanged : SignalAttribute {}
```

```
}
```

Each *IDynamicMassProvider* is responsible for returning its current dynamic mass and triggering *IDynamicMassProvider.DynamicMassChanged*. Ideally, this value should be precomputed when accessed to compute the total block dynamic mass.

In the case of *InventoryComponent*, the precomputation happens when *InventoryData* is changed by adding or removing the weight of items to the mass of the inventory depending on the operation - using *ItemDefinition.MassPerUnit*.

The pre-computed value is stored in *InventoryData*, which aggregates data from all *InventoryComponents* in one struct. It is stored as a field on *InventoryComponent.IndexData*, the per-inventory data struct stored in *InventoryData*. It is then simply returned by *InventoryComponent* as needed when aggregating the total block dynamic mass.

```
/// <summary>
/// Data representing basic inventory information
/// </summary>
public struct IndexData
{
   public int ItemCount;
   public int StartingIndex;

   public FixedPoint VolumeCurrent;
   public FixedPoint VolumeMax;

   public float MassCurrent;
}

public float DynamicMass
{
   get
   {
       return SharedData.PerInventoryData[InventoryIndex].MassCurrent;
   }
}


[CustomSignal(Signal = typeof(IDynamicMassProvider.DynamicMassChanged))]
public partial void DynamicMassChanged();

[OnInventoryChanged]
[OnAdded(typeof(InventoryData))]
[OnRemoved(typeof(InventoryData))]
```

```
[OnChanged(typeof(InventoryData))]
public static void OnInventoryDataChanged(Entity entity, in InventoryData
data)
{
   // getting the first component to access signal trigger
   var inventory = entity.First<InventoryComponent>();
     // only needs to be triggered on one component because InventoryData
aggregates all inventories
   // i.e: they share their static mass
   inventory.DynamicMassChanged();
}
```

CubeBlockComponent mass aggregation

The handler *CubeBlockComponent.OnDynamicMassChanged* notifies the grid physics that its mass changed through *GridPhysicsComponent.BlockMassChanged(CubeBlockComponent)*.

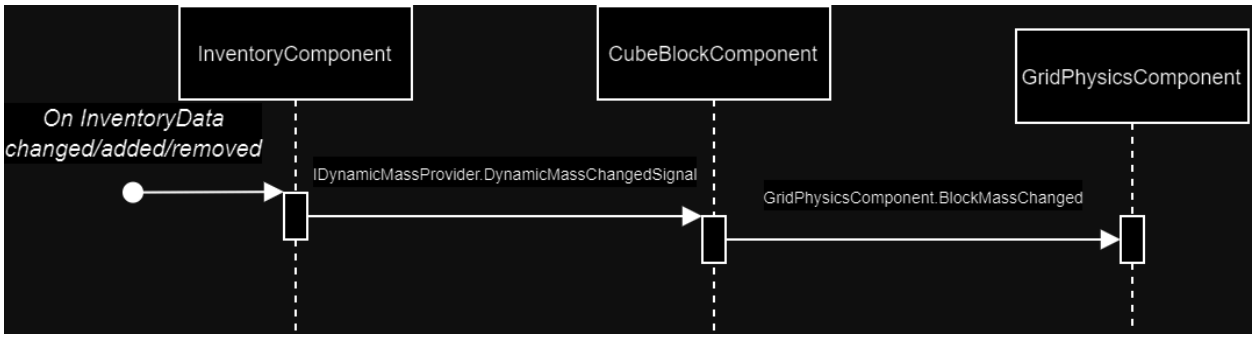
The aggregated dynamic mass value of a block is not stored on any component. It is recomputed ***server-side*** every time it's needed by *CubeBlockComponent* which adds up the masses stored as *IDynamicMassProvider.DynamicMass*. The grid physics triggers the recomputation of the block's total dynamic mass when notified that a *CubeBlockComponent*'s mass has changed.

The current total dynamic mass of a block can be computed through *CubeBlockComponent.ComputeTotalDynamicMass()*.

```
public partial class CubeBlockComponent : GameComponent, IInSceneListener
{
   [...]

   [IDynamicMassProvider.DynamicMassChanged]
   private void OnDynamicMassChanged(Entity entity)
   {
      Grid.Entity.Get<GridPhysicsComponent>().BlockMassChanged(this);
   }

   public float ComputeTotalDynamicMass()
   {
      float totalDynamicMass = 0;
      foreach (var provider in Entity.All<IDynamicMassProvider>())
      {
```

```
            totalDynamicMass += provider.DynamicMass;
        }

        return totalDynamicMass;
    }

    [...]
}
```

Grid Aggregation (use case of inventories)



The aggregation of the mass of each block up to their root grid is through 2 calls:

IDynamicMassProvider.DynamicMassChangedSignal

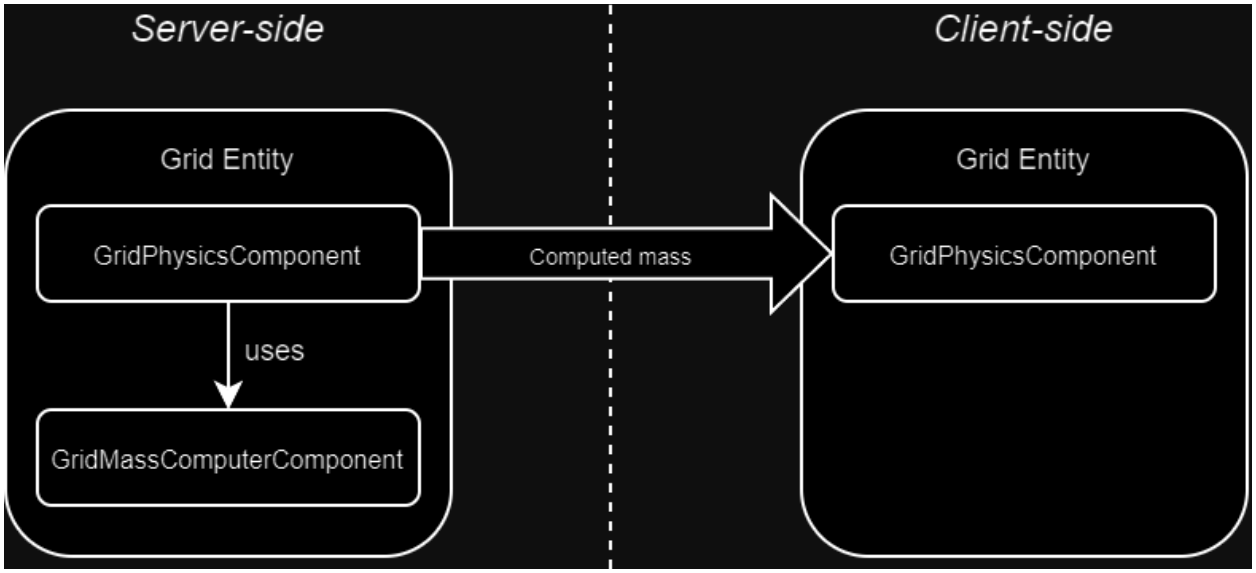
Signal triggered by a static job on *InventoryComponent* when an attached *InventoryData* is updated in any way - see IDynamicMassProvider. *CubeBlockComponent* reports the change up to its parent grid's *GridPhysicsComponent*.

GridPhysicsComponent.BlockMassChanged(CubeBlockComponent)

Method called by *CubeBlockComponent* on *GridPhysicsComponent* when its dynamic mass changes. The *GridPhysicsComponent* then computes the new dynamic mass *only* for the block that changed and aggregates its static and new dynamic mass to get its total mass.

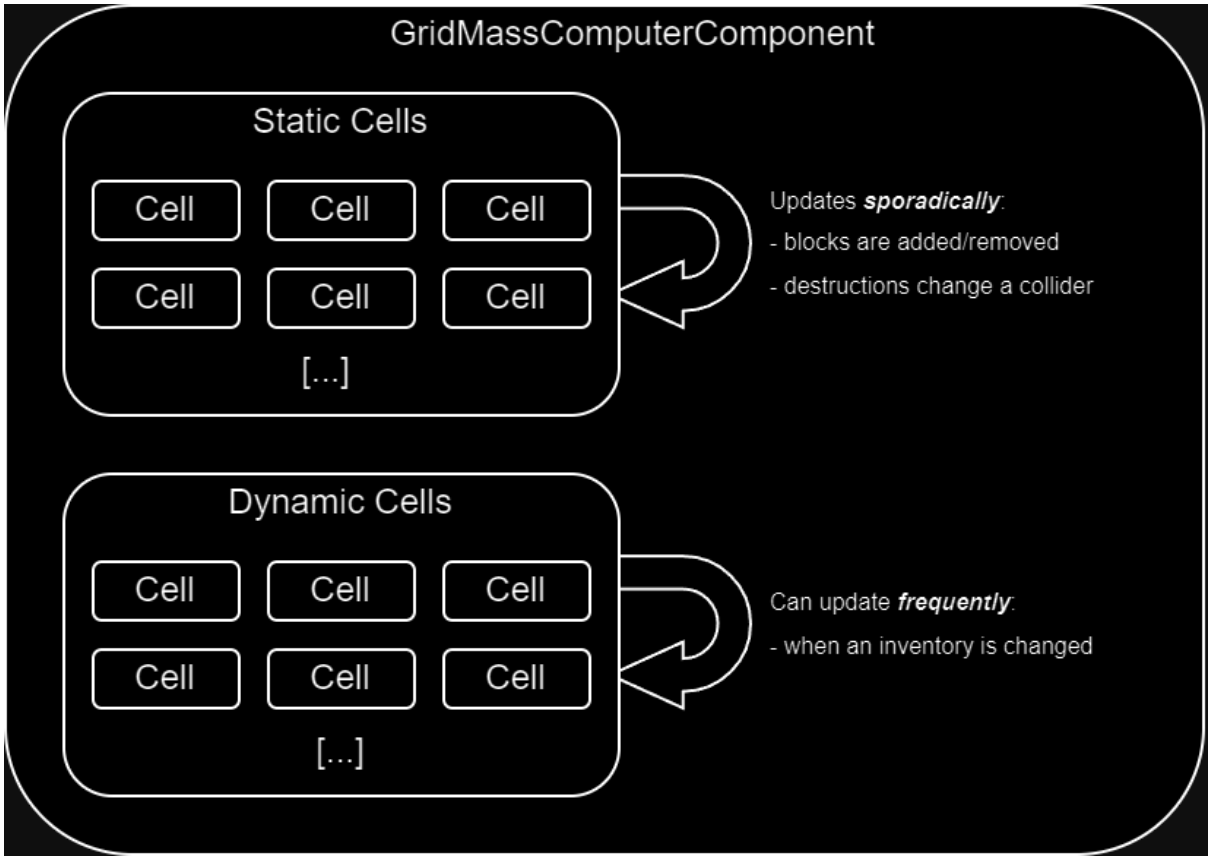
It then updates the block's total mass on ***server-only*** *GridMassComputerComponent*, which is then used to trigger the recomputation of the mass properties of the whole grid as a result

of the mass change.



When processing block mass changes, the Server accesses this Component in order to trigger the recomputation. *GridMassComputerComponent* aggregates all the blocks of the updated grid within spatialized cell objects, when a block's total mass is changed, only its cell will get recomputed and thus all the neighboring blocks contained in the same cell as well.

For performance reasons in the case of frequent updates, *GridMassComputerComponent* aggregates dynamic and static cells in separate trees. Static cells only contain blocks which have no dynamic mass, and vice versa for dynamic cells. This ensures that we don't recompute the mass of blocks which have no dynamic mass when dynamic mass has changed.



The mass properties of all cells then get aggregated to give the mass properties of the grid. The value of which is then stored and applied to the *server* grid collider.

Clients will get the computed mass afterwards through *SyncProperty* (see Client synchronization) and apply the mass to *their* grid colliders.

Grid physics computation aside from the mass, such as the addition and removal of block colliders to the grid, is still performed on ***both*** client and server by *GridPhysicsComponent*.

Client synchronization

The rigidbody mass properties are stored on *GridPhysicsComponent* as *SyncProperty*, which is finally synced from server to client so it can be applied to the grid collider on the client's side.

```
public partial class GridPhysicsComponent : RigidBodyComponent,
```

```
IPhysicsColliderProvider, IArmorProcessor, IPhysicsMotionProvider
{
   [...]

   [SyncProperty]
   private RigidBodyMassProperties _gridMassProperties;

   [GridMassPropertiesProperty]
   [Server, Clients]
   private void OnGridMassChanged(RigidBodyMassProperties mp)
   {
      // Must be propagated to the client in time for physics to see it.
      Assert.True(Entity.Scene.CurrentStageIsBefore<Physics>());
      _sparseCollider.SetMassProperties(mp);
   }

   [...]
}
```

Performance considerations

For the following section, it is assumed that blocks that have dynamic mass will be a minority on the average grid. The measured test cases use grids with ~10% of blocks having dynamic mass.

Memory allocations

The most significant memory footprint of the presented components comes from the *GridMassComputerComponents*.

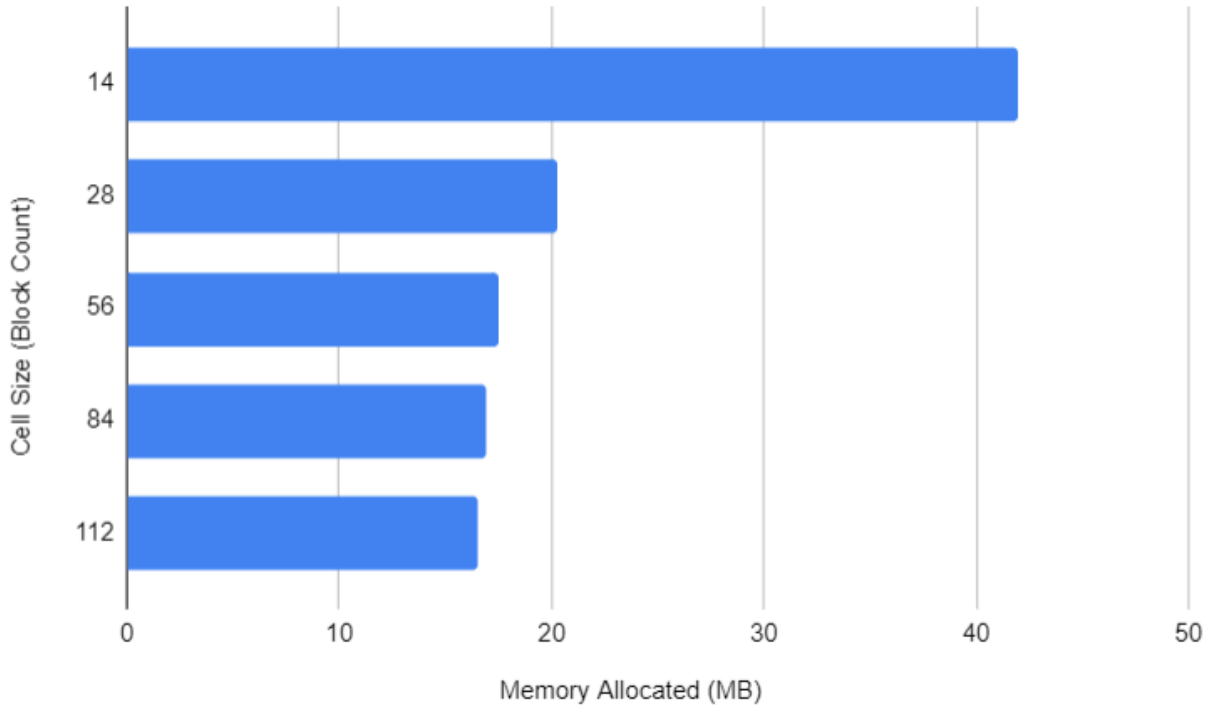
The size of this footprint is dependent on *GridMassComputerComponent.STATIC_CELL_SIZE*, constant value which determines how many blocks can be aggregated within a given static cell. The bigger this size is, the more blocks will be in every given static cell. The bigger the cells are, the less cells will need to be created and the less storage they will occupy in memory in *GridMassComputerComponent*.

The *GridMassComputerComputer.DYNAMIC_CELL_SIZE* functions in much the same way for dynamic cells. Since we assume that there will be less blocks that have dynamic mass than static mass blocks, we can afford to aggregate more in a given cell. As a consequence, the

dynamic cell size is bigger - 2 times bigger than the static cell size in the case of the measurements. This keeps the total cell count lower, reducing the allocated memory.

It's important however to limit the cell size as much as possible, to avoid impacting the performance of mass recomputation, which will only reprocess the changed cell's blocks. As a consequence, a bigger cell size will increase the amount of blocks being reprocessed every time a single cell's mass is recomputed and can impact performance as a result.

Charted below is the memory footprint of the *GridMassComputerComponents* of a ***Blue Ship***, ***Red Ship*** **and** ***Porcupine*** in a scene for increasingly higher cell sizes.



The cells have a **size of 56**, which gives us a ~**17.5 MB** memory footprint in this test case. Higher cell sizes give increasingly lower memory gains, which is why this value is used.

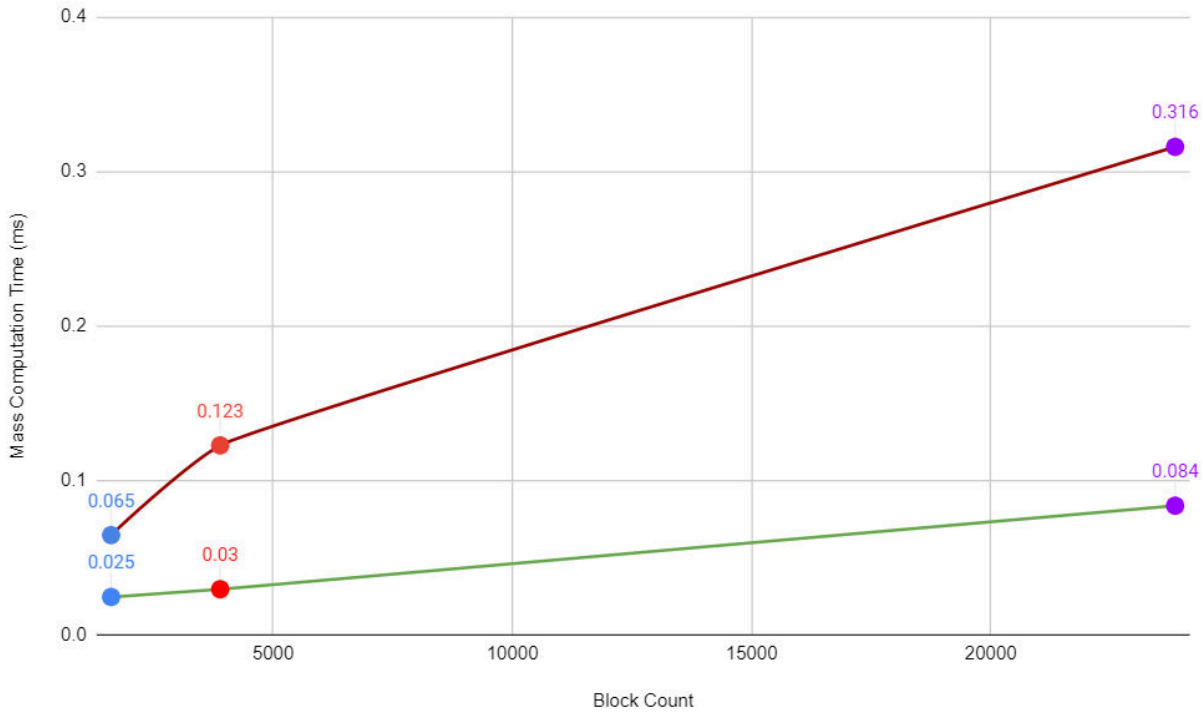
Frequent updating

The separation of static and dynamic mass cell trees in *GridMassComputerComponent* ensures that even if *every dynamic block's mass on the grid changes*, we'll only have to

recompute, at most, the mass of all dynamic blocks. As such, the static block count won't impact performance of dynamic mass recomputation. Recomputation of all dynamic blocks is ***O(n) with n being the amount of dynamic blocks***.

Below are plotted computation times for the mass of three ship grids with evenly distributed dynamic mass blocks processed on the main thread in Dev mode. The green and red curves respectively represent the best and worst case scenarios.

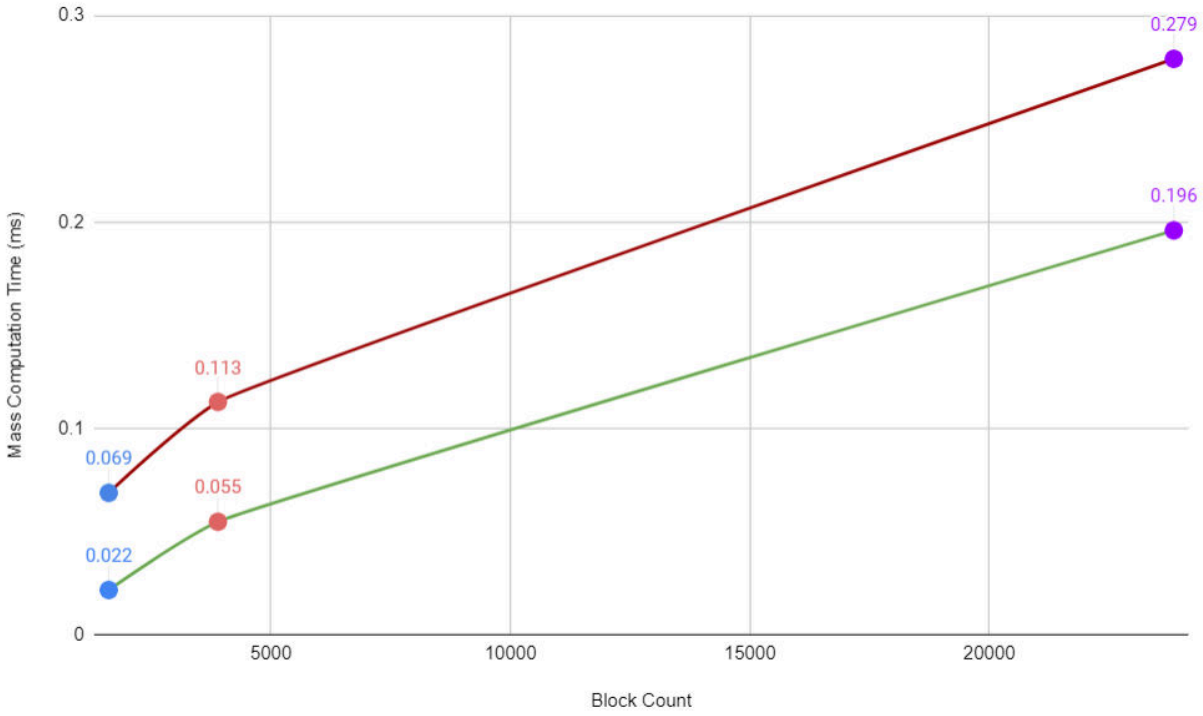
The worst case scenario is every dynamic cell's mass being recomputed and the best case scenario is only one dynamic cell being recomputed.



Computation times with dynamic mass blocks evenly distributed on the grid.

If we compare it to a test case where the blocks with dynamic mass are tightly packed on the grid, we can see that the best case scenario performs significantly worse for bigger grids. This is because the dynamic cells effectively aggregate more blocks when they are tightly packed, so recomputing one single cell gets more expensive.

The worst case scenario however is roughly comparable. This is because whether the 10% of dynamic blocks is tightly packed or evenly distributed, we are still processing the same total amount of blocks in both cases when *all* dynamic cells need to be recomputed.



Computation times with dynamic mass blocks tightly packed on the grid.

In order to reduce the computation time for cases with several potentially big grids needing total dynamic mass recomputation over the same tick, the mass computation runs in a parallelized job:

```
public partial class GridMassComputerComponent : Component
{

   [...]

   ///  Method called by GridPhysicsComponent to trigger computation.
   public void RequestUpdatedMass()
   {
      if (_rootCell.IsEmpty)
      {
          return;
      }
```

```
      if (_dirtyCells.Count == 0)
      {
          Assert.Fail("Updating MassProperties even though nothing was
dirty");
      }


Entity.Scene.FinishDuring<GridPhysicsComponent.MassPropertiesComputation>(Comp
uteUpdatedMass());
   }

   ///  Parallelized mass computation, outputs results to attached
ComputedMassProperties data struct.
   private async Task ComputeUpdatedMass()
   {
      await Entity.Scene.MoveToParallel();

      foreach (var dirtyCellId in _dirtyCells)
      {
          if (!TryGetCell(dirtyCellId, out var cell))
              continue;

          var totalMassForCell =
VRagePhysics.Colliders.ComputeTotalMass(cell!.Items.Span);

          _rootCell.TryGetIndex(dirtyCellId, out var cellIndex);
          _allCellMassProperties[cellIndex] = (totalMassForCell,
RelativeTransform.Identity);
      }

      UnmarkDirtyAll();

      var newGridMass =
VRagePhysics.Colliders.ComputeTotalMass(_allCellMassProperties.Span);

      await Entity.Scene.MoveToDCS();

      Data.Set(new ComputedMassProperties()
      {
          SparseCollidedMassProperties = newGridMass
      });
   }
```