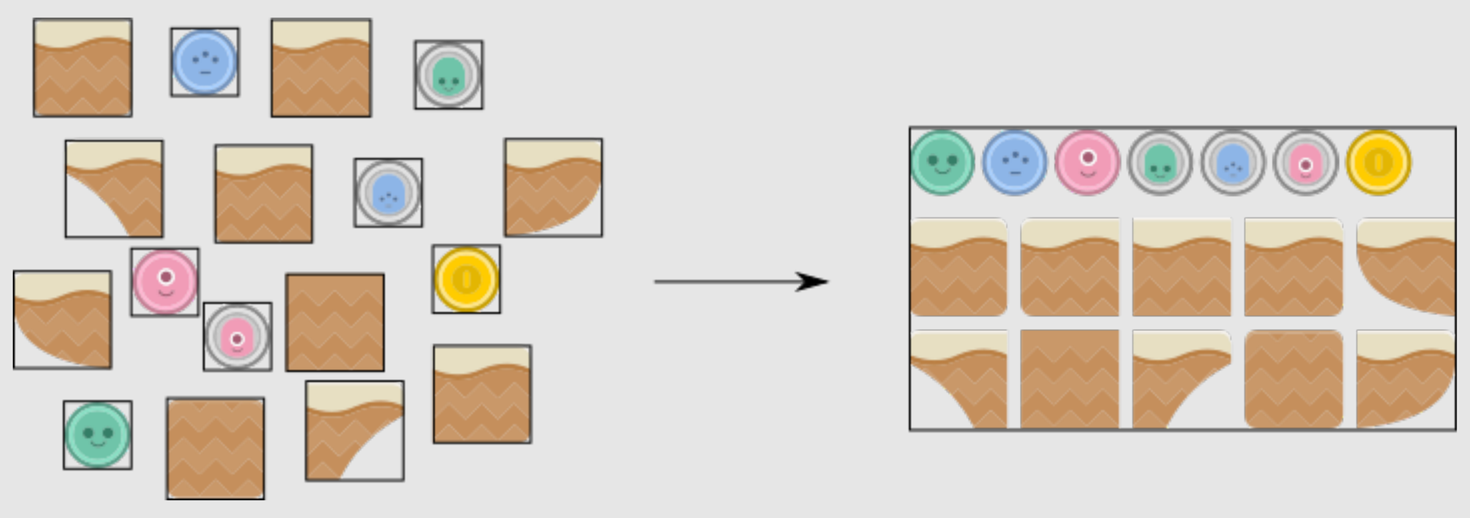
PaunPacker - Texture Atlas Generator

author: Patrik Dokoupil | supervisor: Mgr. Pavel Ježek, Ph.D. | bachelor thesis 2019

Introduction

Texture packing is a process of joining textures together into a single, larger texture called texture atlas. Texture atlases are used in 2D game development to improve rendering performance by reducing the number of draw calls. The process of creating a texture atlas is illustrated in the following picture:



Texture packing is one of many applications of optimization problems called packing problems which are proved to be NP-hard. However, the goal of this thesis is not to devise new algorithms but to create an extensible application with GUI that will allow users to generate texture atlases. This application will be called PaunPacker.

Goals

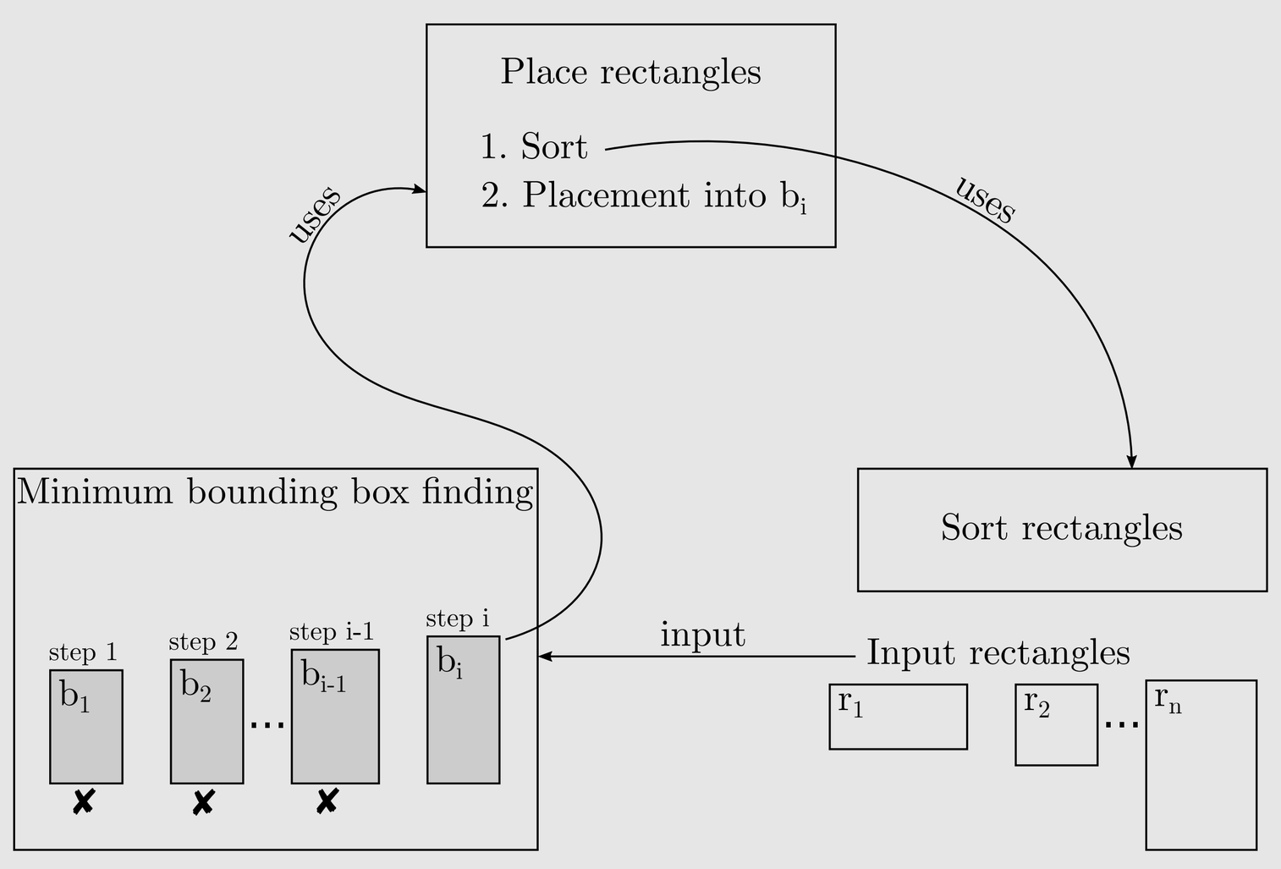
The PaunPacker should meet the following goals:

- Free to use
- Extensible (allowing to create and load plugins)
- Offer additional image processing features (padding, trimming, etc.)
- Have GUI
- Provide a basic toolset for plugin development
- Implement several heuristic algorithms:
  - Maximal rectangles algorithm
  - Genetic based algorithm
  - Bottom-Left algorithm
  - Guillotine algorithm
  - Skyline algorithm
- Implement alias creation

Solution Approach and Architecture

In the PaunPacker we have decided to use one of the standard approaches to rectangle packing which is to decompose the process of finding a "packing" of the input rectangles into three steps:

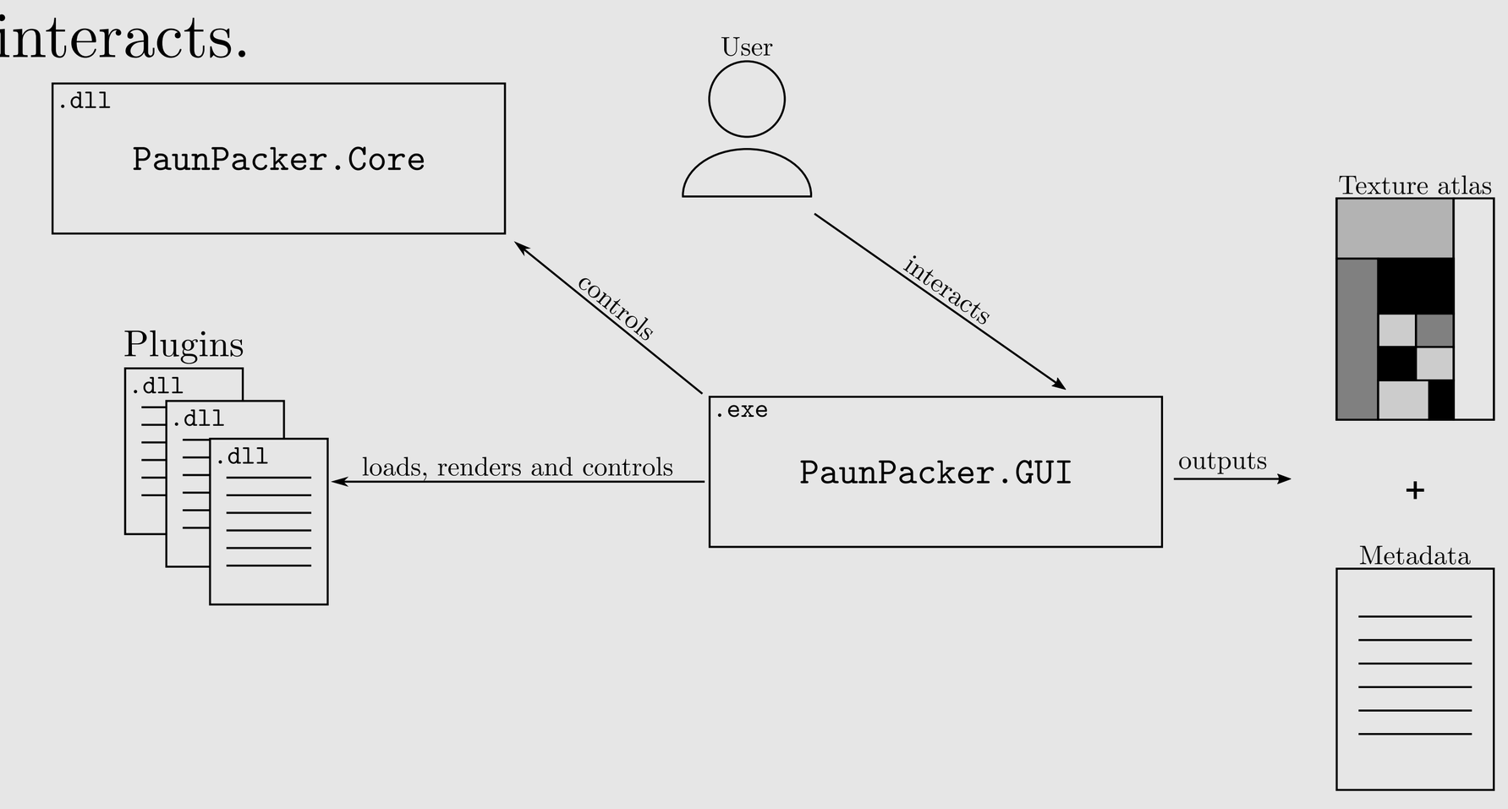
1) Minimum bounding box finding - Finds a minimum bounding box which contains the input rectangles r_1, ..., r_n. Generally, some sequence of bounding boxes b_1, ..., b_k could be tested.
2) Place rectangles - Placement of the input rectangles r_1, ..., r_n into a fixed size bounding box b_i.
3) Sort rectangles - Sorting of the rectangles according to some criterion.



The main feature of PaunPacker is the ability to parameterize the whole algorithm by the individual steps that could be loaded from a plugin. Therefore instead of only dealing with self-contained algorithms, the algorithms are allowed to consist of three independent parts that are dealt with separately.

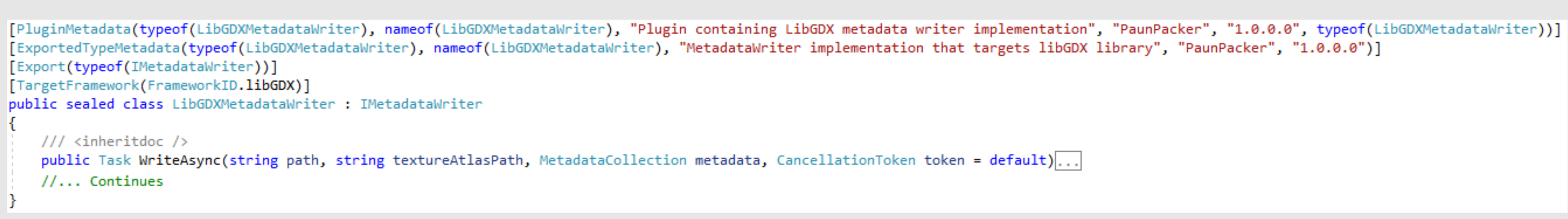
The main parts of the software solution are:

1) `PaunPacker.Core` library (.NET Standard 2)
   - Contains implementations of packing algorithms together with some packing related types.
   - Meant to serve as a basic toolset for future plugin development.
   - Allows to reuse the algorithms for general rectangle packing, i.e. not dependent on GUI of the application.
2) `PaunPacker.GUI` application (.NET Core 3)
   - The main application with which the user interacts.
   - Loads and renders the plugins.
   - Outputs texture atlas and metadata obtained from controlling the algorithms loaded from plugins and core library.
3) Plugins (.NET Standard 2/.NET Core 3)
   - Contain implementations of extensible components or exports components that are implemented in the core library.



Plugins and Extensibility

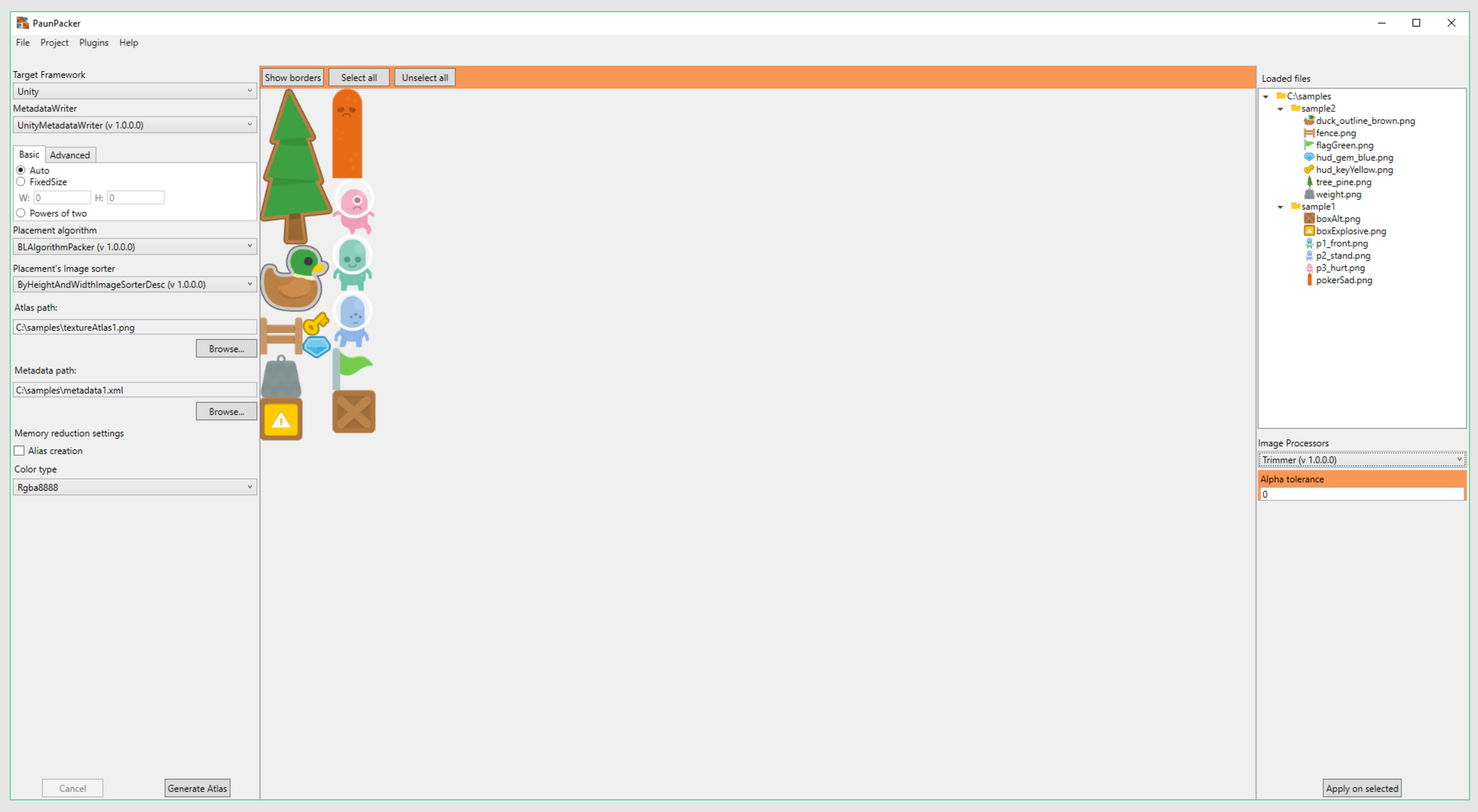
The PaunPacker was from the very beginning designed with a strong emphasis on extensibility. The result is that the application could be extended by plugins that contain implementations of packing algorithms (or any of their three steps), metadata exporters and image processing tools. Individual plugins could also have a view associated with them, that is later rendered in the main window at a location decided by the host application based on the type of the component being exported from a plugin. The easiest way to create plugins is using the MEF's `Export` attribute as illustrated in the picture below where exporting of a metadata exporter (`IMetadataWriter`) is shown. MEF is used mainly for its simplicity that allows developers to create simple (without a view) plugins easily.



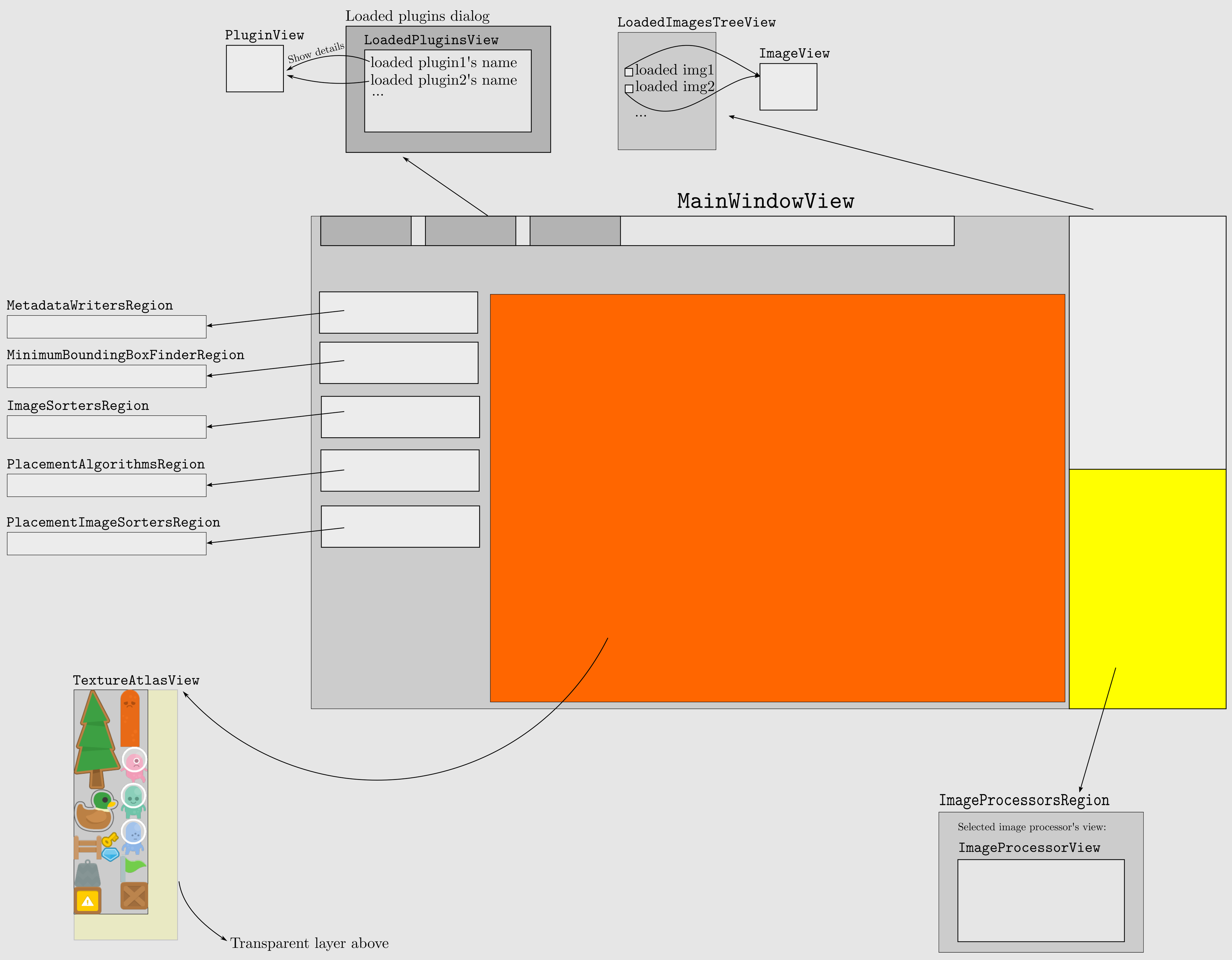
```
[PluginMetadata(typeof(LibGDXMetadataWriter), nameof(LibGDXMetadataWriter), "Plugin containing LibGDX metadata writer implementation", "PaunPacker", "1.0.0.0", typeof(LibGDXMetadataWriter))]
[ExportedTypeMetadata(typeof(LibGDXMetadataWriter), nameof(LibGDXMetadataWriter), "MetadataWriter implementation that targets libGDX library", "PaunPacker", "1.0.0.0")]
[Export(typeof(IMetadataWriter))]
[TargetFramework(FrameworkID.libGDX)]
public sealed class LibGDXMetadataWriter : IMetadataWriter
{
    /// <inheritdoc />
    public Task WriteAsync(string path, string textureAtlasPath, MetadataCollection metadata, CancellationToken token = default)[...]
    //... Continues
}
```

GUI Application

The GUI of the PaunPacker allows the user to adjust various settings, generate texture atlas, process individual images of the texture atlas, load images, manipulate loaded images, etc. and it is shown in the following picture:



Not only the packing algorithms are extensible, but also the GUI of the application is extensible because the types exported from plugins could have a view associated with them. Such a view is then rendered into a proper region inside the main window. To simplify working with regions and views they are showing, Prism Framework is used. Prism is also a second option (besides previously mentioned MEF) for loading of plugins but unlike MEF it also performs their initialization, loading and then showing their views inside a corresponding region. When compared to MEF, the Prism offers a more general and powerful approach for plugin development because it allows creating plugins with views. The picture below shows PaunPacker's main window view and its decomposition into views and regions. Regions determine the place where the view of a particular (selected) extensible component is rendered. Notice that the views could be composed of other views recursively.



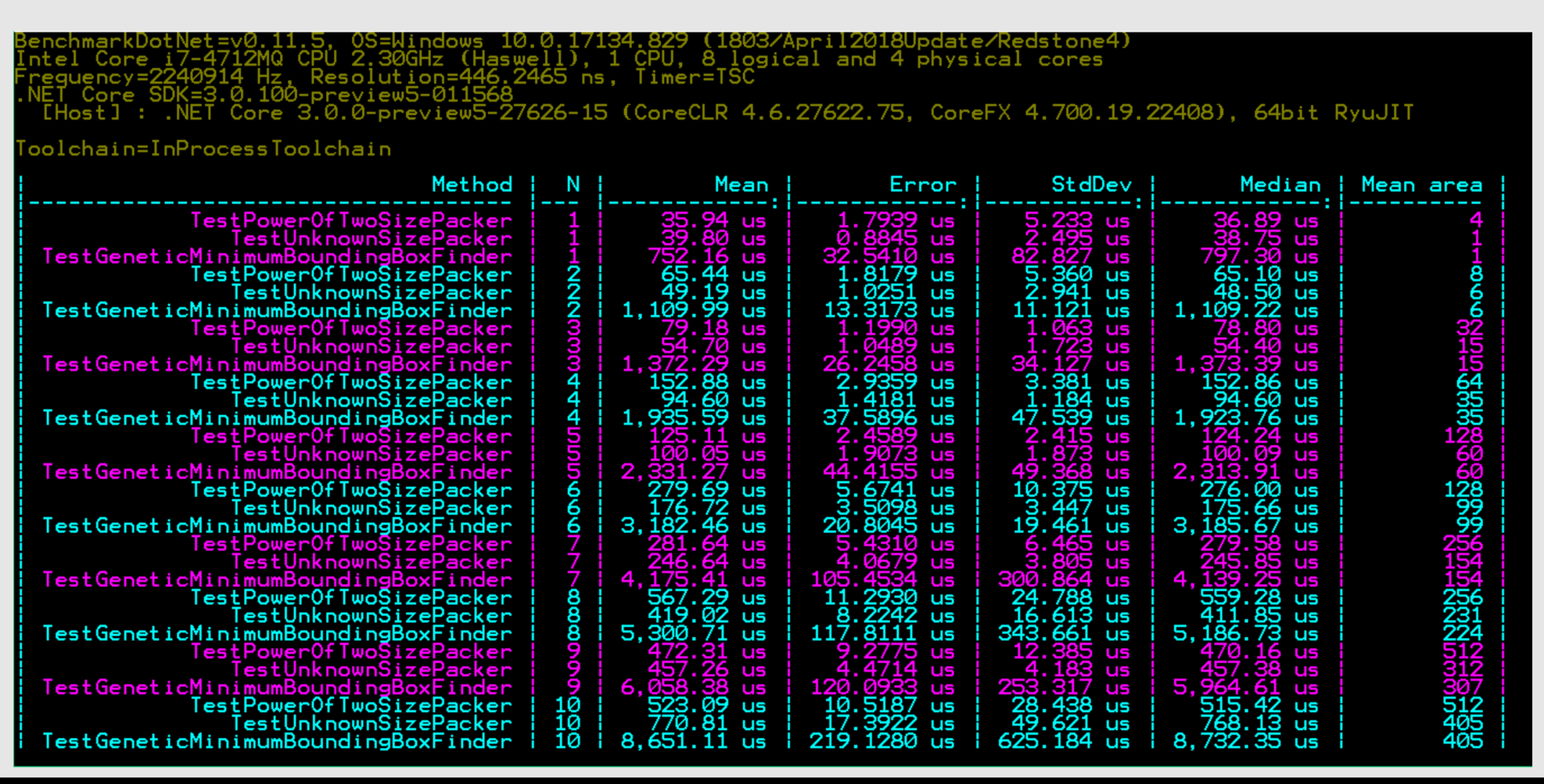
Benchmarks

In addition to the main application for texture atlas generation, a simple CLI application that allows running benchmarks of minimum bounding box finders and placement algorithms has been developed. The benchmarks measure time and average area of the packing result. The benchmarks are generated at runtime for all the types that were loaded from plugins and that could be easily instantiated (via IoC resolution). These benchmarks are later compiled using Roslyn and passed to `BenchmarkDotNet` for an execution.

Currently, there are two simple benchmark scenarios:

1) Squares of dimensions *1x1, ..., NxN*
2) *N* rectangles of random dimensions

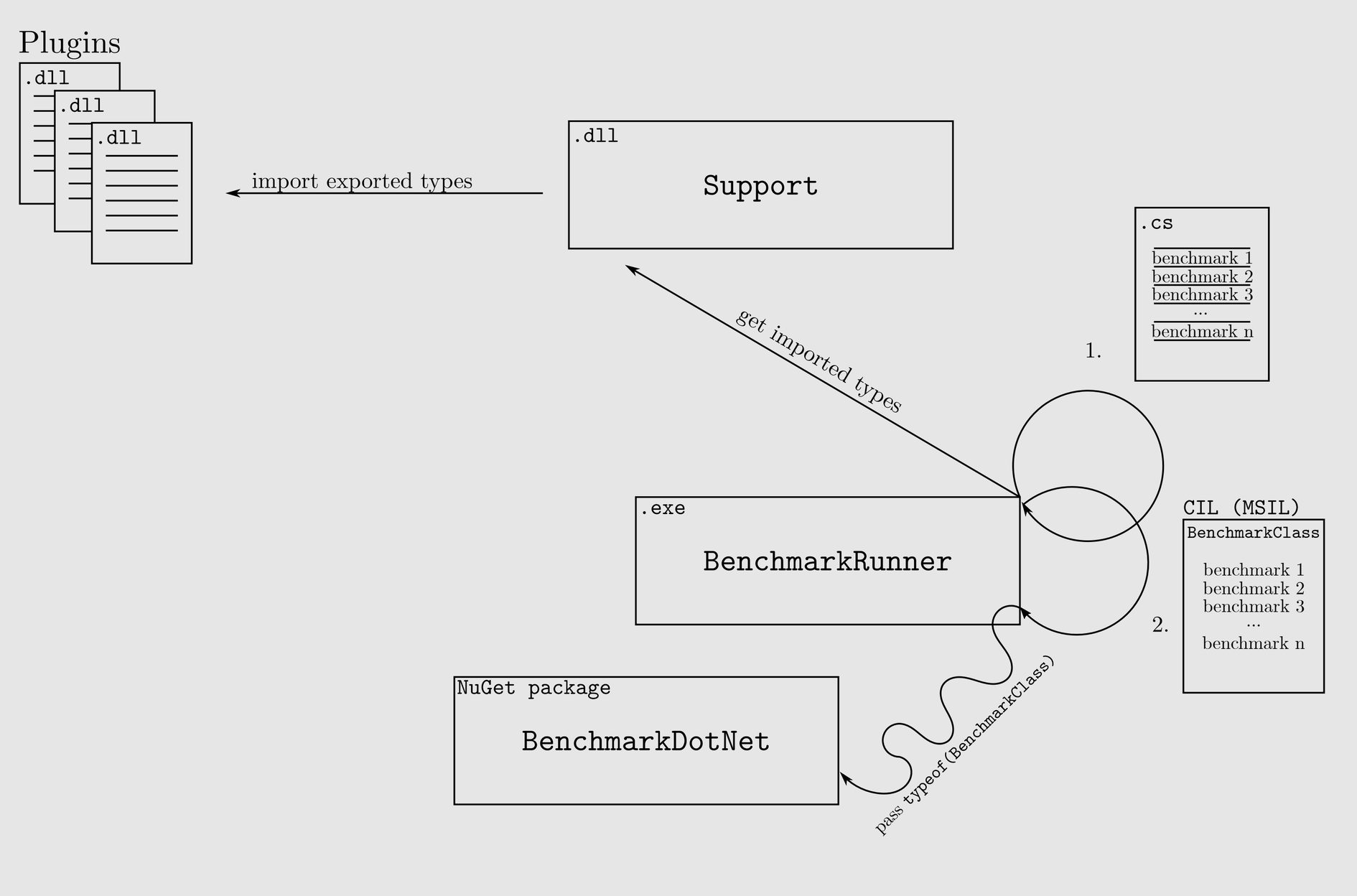
The benchmarks are only expected to be used for a comparison of the individual implementations and not for general comparison of packing algorithms.



BenchmarkDotNet=v0.11.5, OS=Windows 10.0.17134.829 (1803/April2018Update/Redstone4)
Intel Core i7-4712MQ CPU 2.30GHz (Haswell), 1 CPU, 8 logical and 4 physical cores
Frequency=2240914 Hz, Resolution=446.2465 ns, Timer=TSC
.NET Core SDK=3.0.100-preview5-011568
[Host] : .NET Core 3.0.0-preview5-27626-15 (CoreCLR 4.6.27622.75, CoreFX 4.700.19.22408), 64bit RyuJIT

Toolchain=InProcessToolchain

Method	N	Mean	Error	StdDev	Median	Mean area
TestPowerOfTwoSizePacker	1	35.94 us	1.7939 us	5.233 us	36.89 us	4
TestUnknownSizePacker	1	39.80 us	0.8845 us	2.495 us	38.75 us	1
TestGeneticMinimumBoundingBoxFinder	1	752.16 us	32.5410 us	82.827 us	797.30 us	1
TestPowerOfTwoSizePacker	2	65.44 us	1.8179 us	5.360 us	65.10 us	8
TestUnknownSizePacker	2	49.19 us	1.0251 us	2.941 us	48.50 us	6
TestGeneticMinimumBoundingBoxFinder	2	1,109.99 us	13.3173 us	11.121 us	1,109.22 us	6
TestPowerOfTwoSizePacker	3	79.18 us	1.1990 us	1.063 us	78.80 us	32
TestUnknownSizePacker	3	54.70 us	1.0489 us	1.723 us	54.40 us	15
TestGeneticMinimumBoundingBoxFinder	3	1,372.29 us	26.2458 us	34.127 us	1,373.39 us	15
TestPowerOfTwoSizePacker	4	152.88 us	2.9359 us	3.381 us	152.86 us	64
TestUnknownSizePacker	4	94.60 us	1.4181 us	1.184 us	94.60 us	35
TestGeneticMinimumBoundingBoxFinder	4	1,935.59 us	37.5896 us	47.539 us	1,923.76 us	35
TestPowerOfTwoSizePacker	5	125.11 us	2.4589 us	2.415 us	124.24 us	128
TestUnknownSizePacker	5	100.05 us	1.9073 us	1.873 us	100.09 us	60
TestGeneticMinimumBoundingBoxFinder	5	2,331.27 us	44.4155 us	49.368 us	2,313.91 us	60
TestPowerOfTwoSizePacker	6	279.69 us	5.6741 us	10.375 us	276.00 us	128
TestUnknownSizePacker	6	176.72 us	3.5098 us	3.447 us	175.66 us	99
TestGeneticMinimumBoundingBoxFinder	6	3,182.46 us	20.8045 us	19.461 us	3,185.67 us	99
TestPowerOfTwoSizePacker	7	281.64 us	5.4310 us	6.465 us	279.58 us	256
TestUnknownSizePacker	7	246.64 us	4.0679 us	3.805 us	245.85 us	154
TestGeneticMinimumBoundingBoxFinder	7	4,175.41 us	105.4534 us	300.864 us	4,139.25 us	154
TestPowerOfTwoSizePacker	8	567.29 us	11.2930 us	24.788 us	559.28 us	256
TestUnknownSizePacker	8	419.02 us	8.2242 us	16.613 us	411.85 us	231
TestGeneticMinimumBoundingBoxFinder	8	5,300.71 us	117.8111 us	343.661 us	5,186.73 us	224
TestPowerOfTwoSizePacker	9	472.31 us	9.2775 us	12.385 us	470.16 us	512
TestUnknownSizePacker	9	457.26 us	4.4714 us	4.183 us	457.38 us	312
TestGeneticMinimumBoundingBoxFinder	9	6,058.38 us	120.0933 us	253.317 us	5,964.61 us	307
TestPowerOfTwoSizePacker	10	523.09 us	10.5187 us	28.438 us	515.42 us	512
TestUnknownSizePacker	10	770.81 us	17.3922 us	49.621 us	768.13 us	405
TestGeneticMinimumBoundingBoxFinder	10	8,651.11 us	219.1280 us	625.184 us	8,732.35 us	405



Conclusion

The application that has been developed meets all the goals and it is also easily extensible by plugins containing new (parts of) algorithms for packing, tools for image processing and metadata exporters. Also, a simple application that allows performing benchmarks of the individual packing algorithms in order to compare them has been developed. Suggestions for future work include the implementation of new features for enhancing the user experience and optimizing the performance of the implemented packing algorithms.