



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Patrik Dokoupil

PaunPacker - Texture Atlas Generator

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

signature of the author

I would like to thank my supervisor Mgr. Pavel Ježek, Ph.D. for his time, patience and the helpful advice he gave me. Furthermore, I would like to thank my family, friends, and everyone who has supported me during my studies.

Title: PaunPacker - Texture Atlas Generator

Author: Patrik Dokoupil

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The goal of this thesis was to create an extensible application for packing textures into texture atlases, that could then be used in 2D game development. The extensibility lies in the possibility to create and import plugins, containing algorithms for packing, image processing, and metadata exporting. The ability to extend the application by means of plugins makes our application also suitable for testing of newly invented algorithms or for testing of custom variations of the existing ones.

The software solution includes application with user interface that allows the user to create texture atlases and perform additional processing of the textures. Apart from that, we have also included several default implementations of some of the extensible components, namely: placement algorithms, image processing tools and metadata exporters. The concrete algorithms that are implemented in our solution are (among others): Bottom-left algorithm, Skyline algorithm, Guillotine algorithm and also a genetic-based algorithm. All of that can be used as a starting point when developing new plugins.

In addition to generating texture atlases, our application can also generate metadata, that can then be imported by supported game frameworks or libraries. The process of metadata serialization is also customizable, and so users can supply the application with serializers that produce metadata that matches their needs. Two metadata serializers are included in the application itself.

We have also performed several benchmarks regarding performance measured in both time and area of the resulting texture atlas. The results of benchmarks are included in the text.

Keywords: Texture atlas Rectangle packing WPF Plugin

Contents

1	Introduction	3
1.1	Texture atlas	4
1.1.1	Metadata	6
1.1.2	Using texture atlas	9
1.2	Packing tools	14
1.3	Goals	23
2	Theoretical Background	25
2.1	Packing problems	25
2.1.1	Dyckhoff's typology	25
2.1.2	Problem definition	26
2.1.3	Two-dimensional rectangle bin packing	28
2.1.4	Two-dimensional strip packing problem	28
2.1.5	Rectangle packing problem	29
2.2	Solution approaches	31
2.2.1	Heuristic algorithms	31
2.2.2	Meta-heuristics	37
3	Implementation Analysis	39
3.1	Goals revised	39
3.2	Architecture overview	41
3.2.1	High-level components overview	42
3.2.2	PaunPacker's workflow	43
3.3	Choice of platform and development technologies	44
3.4	PaunPacker.GUI	45
3.5	Plugins	47
3.6	PaunPacker.Core	54
3.7	PaunPacker.GUI revisited	58
4	Developer Documentation	61
4.1	NuGet package dependencies	62
4.2	PaunPacker's workflow revisited	64
4.3	PaunPacker.Core	65
4.3.1	Representation of a rectangle	66
4.3.2	Metadata export	66
4.3.3	Packing process representation	66
4.3.4	Representation of image processors	68
4.4	PaunPacker.GUI	69
4.4.1	Application's entry point	69
4.4.2	Views	73
4.4.3	Data binding, INotifyPropertyChanged and ObservableCollection	74
4.4.4	Commands	74
4.4.5	Behaviors	75
4.4.6	Converters	75

4.4.7	Events	75
4.4.8	Services	76
4.4.9	Dialogs	77
4.4.10	Workarounds	77
4.4.11	Resources	78
4.4.12	ViewModels	78
4.4.13	Exported types instantiation	78
4.4.14	Export of built-in extensible components	79
4.5	PaunPacker.GUI.WPF.Common	82
4.6	Tests	85
4.6.1	Benchmarks	86
4.7	Plugins	87
5	User Documentation	89
5.1	Installation and running the PaunPacker	89
5.2	Plugin installation	89
5.3	Limitations	89
5.4	GUI overview	90
5.5	PaunPacker menu	91
5.6	Packing settings and texture atlas generation	93
5.7	Interacting with the texture atlas	96
5.8	Managing the loaded images	97
5.9	Working with Image Processors	98
5.10	Benchmarks	101
6	Creating Plugins Tutorial	103
6.1	Prerequisites and common guideline	103
6.2	Minimal plugin template	106
6.3	Creating plugins without GUI	108
6.4	Creating plugins with GUI	108
6.5	Managing external dependencies	110
6.6	Creating extensible plugins	112
6.7	Best practices	115
	Conclusion	117
	Bibliography	121
	A Attachment	125

1. Introduction

Texture atlas¹ is a set of textures that are packed together as tightly as possible. The packing of textures means taking the input textures and producing a resulting texture (texture atlas) by joining these textures together. Illustration of packing is shown in Figure 1.1.

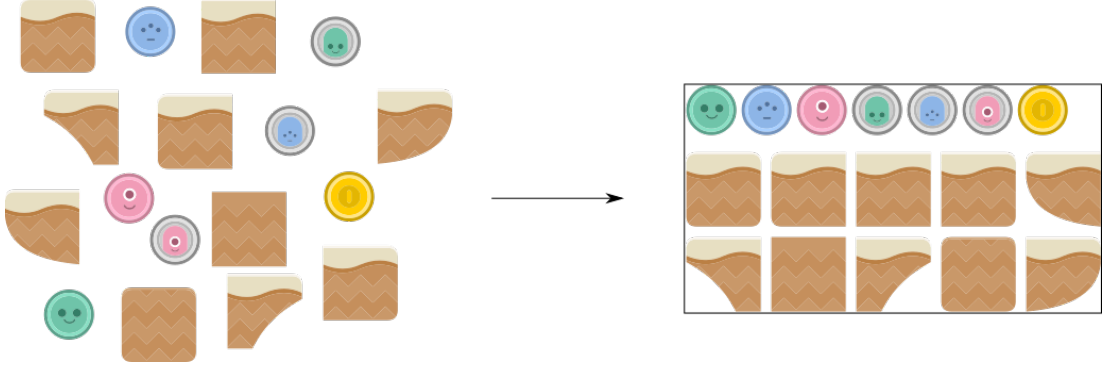


Figure 1.1: Illustration of packing a set of textures into a texture atlas. The textures used in the images are taken from: OpenGameArt [1] where they are published under Public Domain (CC0) license.

On the left side of the Figure 1.1 there are individual textures (input textures) that are being packed together, producing the texture atlas shown on the right. The resulting texture atlas contains all of the input textures, each of them exactly once.

The texture atlases have several use cases in various contexts, but the most common context where they are used is 2D game development and that is also the context regarded in this thesis. In this context, it is useful for a game developer if the tool for texture atlas generation also offers some additional features, for example, it is often useful to remove redundant transparent pixels from the border of the texture in order to make the resulting texture atlas smaller. Other additional features are described later in the Section 1.2.

There exists a couple of tools for texture atlas generation, both paid and free. The problem is, that while the paid tools offer a wide range of additional features, the free tools usually offer only the packing of textures, with either limited set of additional features or no additional features at all. The absence of a free tool for texture atlas generation with sufficient additional functionality has led us to a decision to create such an application on our own. It is worth mentioning, that our intent is not to devise new algorithms for packing, but to come up with an application that will allow users to pack textures, perform some processing on the textures and export metadata about the texture atlas. The metadata contains various kinds of information about the texture atlas and its sub-textures, for example, the positions of the sub-textures. Other information stored in metadata will be described in Section 1.1.1. The target use case is using the texture atlases and metadata generated by our application in the previously mentioned context

¹Texture atlases are also known under names sprite sheet, image sprite, texture map or sprite map.

of 2D game development, together with some game framework that will provide additional support to load the texture atlas and manipulate its sub-textures. It's important to mention, that the game framework's additional support relies on using the metadata when loading the texture atlas and allowing access or manipulation with texture atlas' sub-textures. Before explaining the details about the implementation of the application, the general problem of packing including its related concepts should be introduced. We have decided to accompany the explanation of texture packing and its related concepts with code examples to better illustrate the typical workflow of using texture atlases in the context of 2D game development and to relieve the reader of understanding the explained concepts. To provide code examples, an appropriate game framework has to be chosen and cross-platform Java game development framework called libGDX² seems as a good candidate for this case, because it allows to write self-contained, simple to understand examples and offers a decent support for texture atlases. It is important to mention, that although a concrete game framework was chosen, the concepts depicted in the examples are generic and therefore they will work similarly in other game frameworks.

1.1 Texture atlas

This section is devoted to description and explanation of basic concepts about texture atlases together with illustrations of its use cases. The explanation is accompanied by several examples which should make the explained concepts easier to grasp. Although the examples are based on libGDX and its `.atlas` metadata format, the underlying concepts, and ideas are generic and apply to other game frameworks as well. In particular, most of the information described by `.atlas` is described by other metadata formats too and the support for texture atlas manipulation that libGDX provides is typically also provided by other game frameworks in a similar extent.

Each of the examples mentioned in this section will show the following two methods: `create()`, `render()`. The method `create` is called when the game is first created and it will always contain instantiation and initialization of game's members. The method `render` is called when the game should be rendered and it will always contain code for rendering the game screen. It is important to mention that in this section, all the lengths, distances and dimensions are measured in pixels.

The rendering of textures in `render` method is done by using `SpriteBatch` class. The word `Sprite` in the name of `SpriteBatch` relates to a `Sprite` class that contains geometry (position), color and texture information (texture or more generally texture region, rotation, size, etc.). The `SpriteBatch` class as its name says, batches the sprites (instances of `Sprite` class) with the same texture together into a single draw call. The `SpriteBatch` starts the batch with a call to its `begin` method which means, that when a subsequent call to `Sprite`'s `draw` method comes, the `SpriteBatch` collects the `Sprite`'s geometry instead of sending it immediately to the GPU. The collected geometry is sent once the `end` method is called on `SpriteBatch` or when the following scenario happens: sup-

²More information about libGDX framework can be found on its website [2].

pose a sequence of n sprite draw requests³ S_1, \dots, S_n ordered by the time when they have occurred, from oldest to newest. When new sprite draw request S_{n+1} comes and the texture corresponding to this draw request is different from the texture that corresponds to the draw request S_n , the batch ends. The `SpriteBatch` is rather heavy (as stated in libGDX’s developer documentation [3]), so typically only a single instance is created. An example of rendering textures is shown in the Listing 1.

```
1  @Override
2  public void create () {
3      batch = new SpriteBatch();
4      sampleTexture = new Texture("someTexture.png");
5      sampleTextureRegion = new TextureRegion(sampleTexture, 0, 0, 50, 50);
6      sampleSprite = new Sprite(sampleTexture);
7      sampleSprite.setPosition(400,400);
8  }
9  @Override
10 public void render () {
11     ...
12     batch.begin();
13     batch.draw(sampleTexture, 0, 0);
14     batch.draw(sampleTextureRegion, 100, 100);
15     sampleSprite.draw(batch);
16     batch.end();
17 }
```

Listing 1: An example of using `SpriteBatch` to render `Texture`, `TextureRegion` and a `Sprite`.

In the Listing 1 there is a `create` method, where `SpriteBatch`, `Texture`, `TextureRegion` and `Sprite` are created. The `Texture` represents a texture and it can be created from a texture file located on a disk, in this case, from the texture file called "someTexture.png". The `TextureRegion` represents a rectangular area of a texture and it is created by specifying the source texture, coordinates of region’s bottom-left corner and the size of the region. In this example, the created texture region represents a rectangular area of `sampleTexture` with the bottom-left corner at coordinates 0,0 and the size of 50x50 pixels. At the end of the `create` method, the `sampleSprite` is created from the `sampleTexture` and its position is set to 400,400. The `render` method then renders `sampleTexture` at position 0,0, `sampleTextureRegion` at position 100,100 and finally `sampleSprite` at the position it has set—400,400.

³Here the draw request means a call to `Sprite`’s `draw` method.

1.1.1 Metadata

Texture atlases are typically accompanied by some metadata in one of the various kinds of formats—for example, libGDX uses the format called `.atlas`. The formats of metadata are typically related to the target game framework that will be used to load the texture atlas, but this relation is not one-to-one, because there are metadata formats that are understood by multiple game frameworks, but on the other hand, there exist metadata formats that are not understood by any game framework, because developers could create their own (custom) metadata formats and work with the formats on their own, without additional support of any game framework. The metadata contains additional information about the sub-textures within the texture atlas, for example, sub-texture’s size, its location within texture atlas, etc.

The `.atlas` format

The files in `.atlas` format are simple text files containing information about texture atlases and their sub-textures. The file is divided into pages that are further divided into regions (as stated in comments inside libGDX’s GitHub repository [4] or at Spine’s⁴ website [6]). Each page represent one texture atlas and the regions inside the page represents its sub-textures. Pages are separated by an empty line and each page starts with a name (actually it’s a path) of the corresponding texture atlas file followed by the following attributes:

1. `size`
2. `format`
3. `filter`
4. `repeat`

After these attributes, there comes a list of regions, where each region starts with a region name followed by some (not all of them are mandatory) of the following attributes:

1. `rotate`
2. `xy`
3. `size`
4. `split`
5. `pad`
6. `orig`
7. `offset`
8. `index`

The structure of `.atlas` file is illustrated in the Figure 1.2.

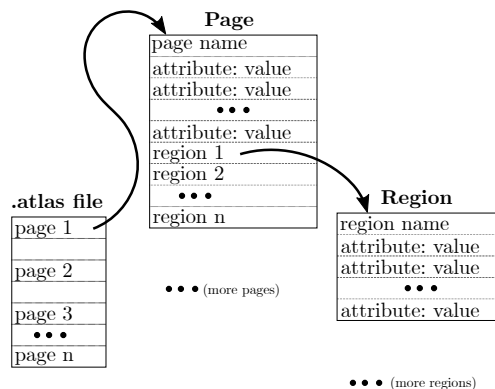


Figure 1.2: The `.atlas` file and its division into pages and regions.

⁴Spine is an animation tool developed by Esoteric Software LLC. More about Spine can be found at its website [5].

The left side of Figure 1.2 shows a `.atlas` file that contains `n` pages, called from `page 1` to `page n`. These pages are listed one by one in the `.atlas` file, separated by an empty line. Each page contains the page name and multiple attributes followed by the regions as can be seen in the middle of the Figure 1.2. Each of the page's regions contains a region name followed by its own attributes. Instead of showing the structure of `.atlas` file by listing its contents, the figure uses arrows (the arrows means "gets replaced by") to represent the `.atlas` file's structure.

An example illustrating the syntax of `.atlas` format on a concrete `.atlas` metadata file is shown in Figure 1.3.

```
textureAtlas.png
size: 92, 161
format: RGBA8888
filter: Linear,Linear
repeat: none
hud_coins
  rotate: false
  xy: 44, 114
  size: 47, 47
  orig: 47, 47
  offset: 0, 0
  index: -1
```

Figure 1.3: The example of `.atlas` file with a single page and a single region.

The Figure 1.3 shows a `.atlas` file containing a single page called `textureAtlas.png` with a size of 92x161 pixels. The `format` corresponds to pixel format that will be used when storing the texture atlas in memory and it is `RGBA8888` in this case. The `filter` configures the minification⁵ and magnification of the texture atlas, in this case, they are both set to `Linear` (linear interpolation). The `repeat` attribute specifies the axes in which the texture should repeat⁶ and in this case, it is set to `none`, so the sub-texture will not repeat in any of the axes. The page contains a single region called `hud_coins` with the following attributes: `rotate`, `xy`, `size`, `orig`, `offset` and `index`. The attribute `rotate` indicates whether the region is stored rotated by 90 degrees in the texture atlas. The `xy` are coordinates of the top-left corner of the region with respect to the top-left corner of the texture atlas and `size` is the size of the region. The `orig` attribute is the original size of the region, in this case, it is equal to the size of the region which means that the size did not change during the packing. The `offset` is the number of whitespace pixels removed from left and bottom of the original sub-texture. In this case, the `offset` is 0,0, meaning that no whitespace pixels were removed. The `index` allows several regions (even on the same page) to have the same name, as long as they have a different index. The index determines the order of regions with the same name. The index is useful when dealing with animations because then the index represents the animation's frame in which the

⁵More about minification and magnification can be found at LearnOpenGL website [7]

⁶This attribute is related to texture wrapping. More about texture wrapping can be found at LearnOpenGL website [7].

region (its corresponding sub-texture) should be shown. In this example, the region has `index` equal to `-1` which means that the `index` is not used. For the `index`, the texture packing tools typically use the number between after last `'_'` in the file name (the name without file extension is considered) of the corresponding sub-texture or `-1` when such a number is not present.

The last example of this section, showing metadata in `.atlas` format is shown in the Figure 1.4 on the left, together with its corresponding texture atlas on the right.

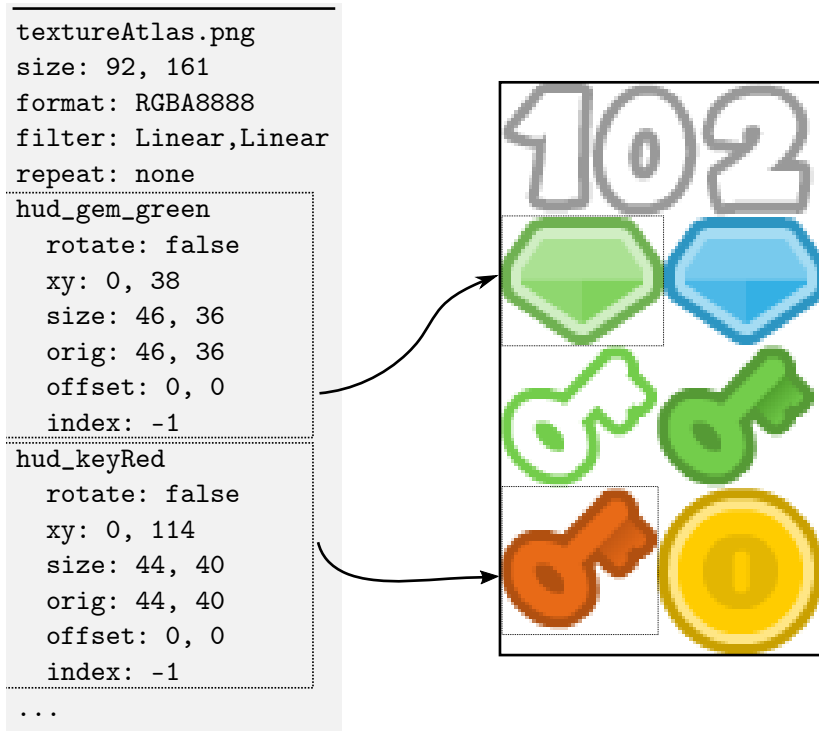


Figure 1.4: Example of metadata in `.atlas` and its corresponding texture atlas. The textures used in the images are taken from: OpenGameArt [1] where it is published under Public Domain (CC0) license.

The Figure 1.4 shows a texture atlas on the right together with a part of the texture atlas' corresponding metadata on the left. The listed metadata contains a single page called `textureAtlas.png` that contains two regions called `hud_gem_green` and `hud_keyRed`. The region called `hud_gem_green` corresponds to the green diamond sub-texture and `hud_keyRed` corresponds to the red key sub-texture. The regions corresponding to the remaining sub-textures (green key, blue diamond, etc.) were omitted for simplicity.

1.1.2 Using texture atlas

As already mentioned at the beginning of Section 1, the target use case is using texture atlases for 2D game development, together with some game framework. To illustrate this use case, imagine the following situation: a game developer that is working on a 2D game using the libGDX framework. Suppose that the game consists of several game objects (player, characters, enemies, etc.) each of them having its corresponding (not necessarily unique) texture and position and that these game objects have to be rendered onto the screen. A simple way to render the objects would be to store every texture in a separate texture file, create `Texture` for each texture file, then create `Sprite` for each game object (by calling `Sprite`'s constructor with corresponding `Texture` as a parameter) and finally calling the `draw` method on every `Sprite`. This approach is illustrated in the Listing 2.

```
1  @Override
2  public void create () {
3      ...
4      gameObjectSprites = new Sprite[10];
5      spriteTextures = new Texture[10];
6      for (int i = 0; i < 10; i++) {
7          //Create texture from a texture file for a given sprite
8          spriteTextures[i] = new Texture("sprite_" + i + ".png");
9          gameObjectSprites[i] = new Sprite(spriteTextures[i]);
10         ... //Set positions of the Sprites
11     }
12 }
13 @Override
14 public void render () {
15     ...
16     batch.begin();
17     for (int i = 0; i < 10; i++) {
18         gameObjectSprites[i].draw(batch);
19     }
20     batch.end();
21 }
```

Listing 2: An simple approach of drawing 10 `Sprite`s with 10 corresponding `Textures`.

In the `create` method shown in the Listing 2, there are 10 `Sprite`s created together with their 10 corresponding `Textures`. These sprites are then rendered one by one in the `render` method. Because every `Sprite` uses a different texture, each call to `Sprite`'s `draw` method causes the collected geometry of a single `Sprite` to be sent to the GPU immediately which results in 10 draw calls.

The simple approach that is shown in Listing 2 works, but it is inefficient because—as previously mentioned in Section 1.1—when comes a request to draw a new `Sprite` S_{n+1} with a texture different from the texture of a previous `Sprite`

S_n , the **batch** is finished, the collected geometry is sent to the GPU and the **batch** then begins again, by collecting the geometry of the **Sprite** S_{n+1} . With an increasing amount of textures used by the **Sprites** (game objects) this inefficiency increases even further, until it will eventually slow down the whole game in such a way, that the player's play experience could be ruined. The remedy to this problem is to use texture atlases and instead of storing each texture in a separate texture file, store all of the textures together in the texture atlas and create the **Sprites** using the **TextureRegions** of this texture atlas. A different version of code from Listing 2, improved by using texture atlas, is shown in Listing 3.

```

1  @Override
2  public void create () {
3      ...
4      textureAtlas = new TextureAtlas("metadata.atlas");
5      gameObjectSprites = textureAtlas.createSprites();
6      ... //Set positions of Sprites
7  }
8  @Override
9  public void render () {
10     ...
11     batch.begin();
12     for (int i = 0; i < 10; i++) {
13         gameObjectSprites[i].draw(batch);
14     }
15     batch.end();
16 }

```

Listing 3: An simple approach of drawing 10 **Sprites** with 10 corresponding **Textures**.

In the **create** method shown in the Listing 3 an instance of **TextureAtlas** class is created from a texture atlas' metadata file called "metadata.atlas". The **TextureAtlas** contains 10 **Sprites** whose instances are obtained by calling the **createSprites** method on the **TextureAtlas**. These sprites are then rendered one by one in the **render** method. But unlike the code example from Listing 2, this time the texture atlas is used. By using the texture atlas, all of the sprites will refer to the same texture, and therefore more **Sprites** could be drawn in a single draw call to the GPU, because the **batch** will not finish until the moment when the **end** method is called. It should be mentioned, that it is not always necessary to store all of the textures in a single texture atlas, because sometimes the amount of textures is so large, that the texture atlas would become extremely large and possibly exceed certain limits on a texture size that are given by the GPU. If that is the case, the textures could be divided into (arbitrary) groups that can be stored separately in their own texture atlases. An important note is that both the performance issue caused by storing the textures in separate files

and the solution to this issue are not specific to libGDX⁷ but to the rendering on modern GPUs in general.

Texture atlases could also help to reduce memory consumption in some cases. For example, some mobile GPUs constraint the dimensions of textures to be in powers of two. When this is the case, then storing multiple textures in one texture atlas with dimensions that are powers of two can result in lower memory consumption (per single sub-texture) compared to storing each of the sub-texture in an individual texture, because individual sub-textures will typically have to be padded with transparent pixels in order to have dimensions that are powers of two. An illustration of the aforementioned case is shown in Figure 1.5.

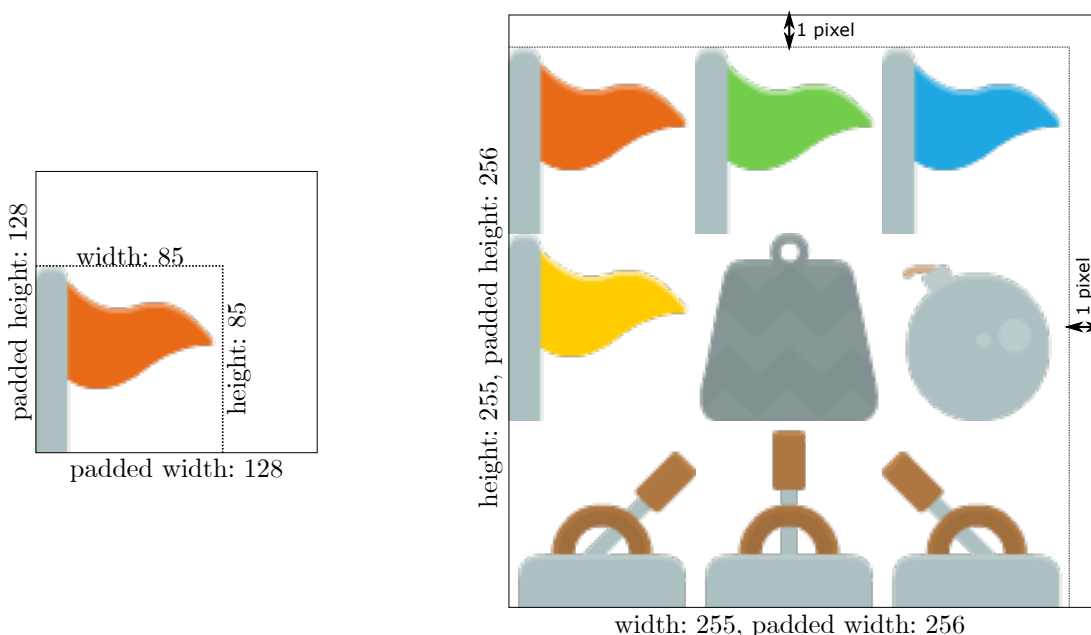


Figure 1.5: An example of memory wasting caused by padding of textures with dimensions that are not in the powers of two, together with an illustration of how texture atlases mitigate this issue.

On the left of the Figure 1.5 there is a single texture of size 85x85 pixels that is packed into a single texture atlas with a size of 128x128 pixels, resulting in a waste of 9159 pixels. On the right of this figure, there are 9 textures with same size—85x85 pixels—that are packed together into a single texture atlas with a size of 256x256 pixels, resulting in a total waste of 511 pixels and 57 pixels wasted per single sub-texture.

2D Animations

Texture atlases are frequently used to store frames of 2D animations, for example, an animation of a movement of some game character. The 2D animations consist of so-called animation frames—views of animated objects, or more generally, views of an animated scene—which are rendered one by one in a given order at

⁷Actually, in libGDX this issue is slightly mitigated thanks to use of the `TextureBatch` that allows at least a sequence of `Sprites` with the same texture to be rendered in a single draw call.

set intervals, yielding the animation. Consider an animation of game character with 2 animation frames shown in the Figure 1.6.

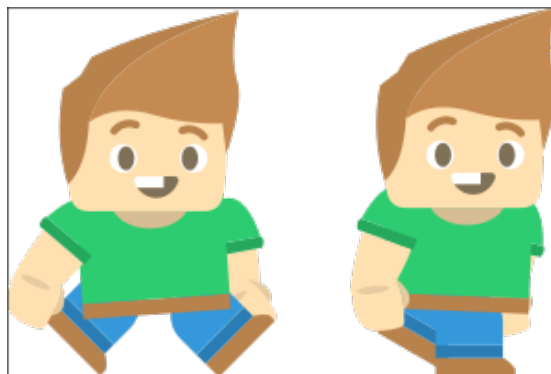


Figure 1.6: A part of texture atlas containing animation frames for character movement animation. The textures used in the images are taken from: OpenGameArt [8] where it is published under Public Domain (CC0) license.

The texture atlas in Figure 1.6 contains 2 sub-textures that are representing individual key frames of the character movement animation. When both sub-textures are alternately rendered, the game character will appear like it is moving to the right.

Sometimes, the game frameworks provide special support for working with animations, for example, they may provide a convenient way to load and play animations. Figure 4 shows an example of loading an animation from a texture atlas that is shown in 1.6.

```
1  @Override
2  public void create () {
3      ...
4      anim = new Animation<TextureRegion>(1.0f / atlas.getRegions().length,
5                                          atlas.findRegions("walk"),
6                                          Animation.PlayMode.LOOP);
7  }
8  @Override
9  public void render () {
10     ...
11     animTime += Gdx.graphics.getDeltaTime(); //sum elapsed animation time
12     //obtain animation frame corresponding to elapsed animation time
13     currentFrame = anim.getKeyFrame(animTime);
14     batch.begin();
15     batch.draw(currFrame, 0, 0); //draw the current frame at position 0,0
16     batch.end();
17 }
```

Listing 4: Code example for animation rendering, illustrating the libGDX's special support for animations.

In the `create` method shown in the Listing 4, an instance of `Animation` class is

created specifying the frame duration, regions representing the animation frames and the looping play mode. In this case, the frame duration is set to 0.5 seconds (because there are 2 regions for a walk) and animation mode is LOOP, meaning that the animation will be played over and over again. Notice that the regions (animation frames) are obtained by calling `atlas.findRegions("walk")`, this works because of the `index` attribute in `.atlas` format that was described in Section 1.1.1. In the `render` method an animation time is calculated then the `TextureRegion` corresponding to this time is obtained and rendered at position 0,0.

1.2 Packing tools

Currently, there are several texture packing tools available, each of them possessing a different set of additional features. But as already mentioned at the beginning of this chapter, the problem with the current state is that packing tools are usually either paid—with the only limited free version or with no free version at all—or they offer only the ability to pack the textures together, without any additional features. One of the best-known and smartest tools available are TexturePacker⁸ and Zwoptex⁹. Both TexturePacker and Zwoptex are great tools with a lot of smart features, but that comes with a price—Zwoptex costs about \$10 per user license and TexturePacker is even more expensive and costs about \$39.99 for a year license or \$99.99 for a lifetime license. This leads to **requirement R1: *Our application should be free to use.***

There are also several packing tools that are free to use, or better yet, open source. For example, there is an open source tool called SpriteMapper¹⁰, which unfortunately supports only the packing of texture together with a very limited set of additional features. This leads to **requirement R2: *Our application should provide several additional features for image processing.*** Now, some selected packing tools are going to be introduced, their pros & cons described and finally they will be compared based on the following characteristics:

1. Platforms where the tool is available
2. Additional tools for image processing (image processors)
3. Additional features that the tool offers
4. Supported metadata formats and whether metadata in custom format could be exported
5. Available packing algorithms
6. Extensibility of packing algorithms
7. Extensibility of image processors
8. GUI¹¹
9. CLI¹²
10. Price

Before giving the introduction of the selected packing tools, some of the aforementioned characteristics will be described, beginning with an explanation of the concept of an image processor. The image processor is an abstraction that can be thought of as a function that takes a texture as the input and returns a modified copy of the texture as the output. The image processing tools that are frequently present in the packing tools and that will be used to compare the packing tools later in this section, are:

- Trim
- Crop
- Extrude

⁸More information about TexturePacker can be found at its website [9]

⁹More information about Zwoptex can be found at Zwoptex's website [10]

¹⁰More information about SpriteMapper can be found at its website [11]

¹¹Graphical User Interface

¹²Command Line Interface

- Padding
- Color type change
- Heuristic mask
- Common divisor

The description of each image processing tool will be accompanied by a figure illustrating the effect of the tool. Each figure will show the input texture on its left and the output texture on its right. All of the textures in the remainder of this section are taken from: OpenGameArt [1] where they are published under Public Domain (CC0) license.

Trim removes transparent pixels from the border of a texture, which reduces the size of the texture. With metadata that allows storing information about trimming and with additional support from the target game framework, the size of the original texture may be preserved and the framework will read the metadata and restore the transparent pixels appropriately when the texture will be rendered.

Crop permanently removes transparent pixels from the border of texture, meaning that unlike in the case of trimming, the removed pixels will not be restored when the texture will be rendered. The Figure 1.7 shows the process of trimming/cropping.

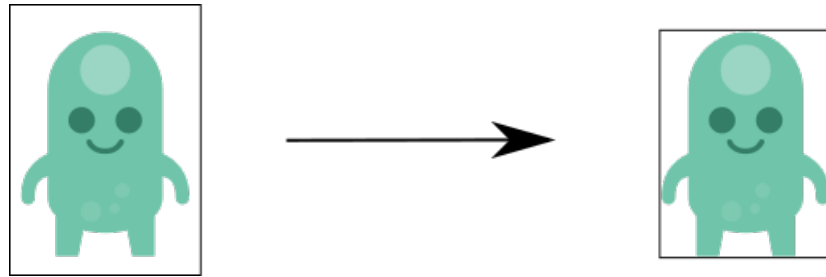


Figure 1.7: Crop / Trim texture.

Extrude replicates the pixels from the border of texture n times, in order to reduce flickering in cases where textures are rendered close to each other. The process of extruding is depicted in the Figure 1.8.

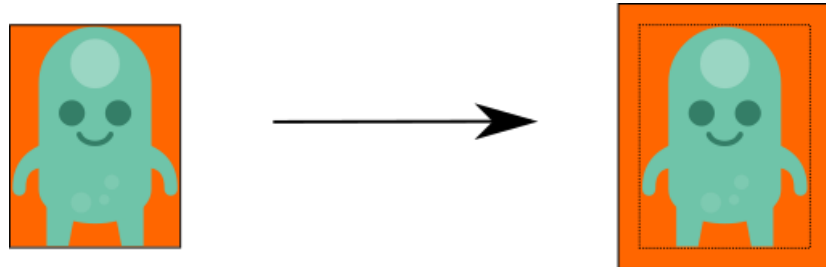


Figure 1.8: Illustration of extrude on a texture.

Padding exists in several variants, but in this thesis, only the following variants will be considered: *Border Padding* is a space between the bounding box of the textures in the texture atlas and texture atlas's border. *Shape Padding* is a space between the textures in the texture atlas. *Inner Padding* is a space that

is added around each texture in the texture atlas. The example of adding inner padding to a texture is given in Figure 1.9.

Similarly to trimming, with metadata that supports padding and width additional support from the target game framework, the textures will be read from the texture atlas without the padding.

It should be mentioned that from these three variants, only the Inner Padding can be considered as an image processing tool because for the other two, some additional support from the packing tool is needed.

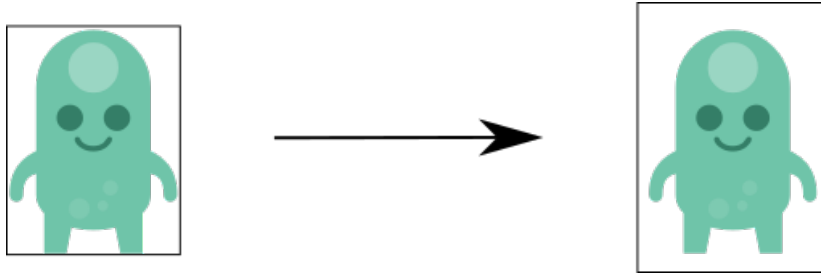


Figure 1.9: Adding padding to a texture.

Color type change changes the format of pixels in order to reduce memory consumption. Different packing tools offer different sets of supported pixel formats, for example: RGBA8888, RGBA4444, Gray8, etc. The example of changing color type to 8-bit gray-scale is shown in Figure 1.10.

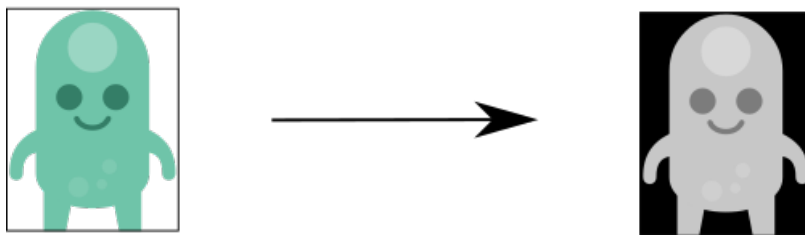


Figure 1.10: Change texture's color type to Gray8.

Heuristic mask allows to remove background from the texture, i.e. to make the background of the texture transparent. However, this tool is typically restricted to single color backgrounds only. The use of a heuristic mask to remove the background from the texture is shown in Figure 1.11.

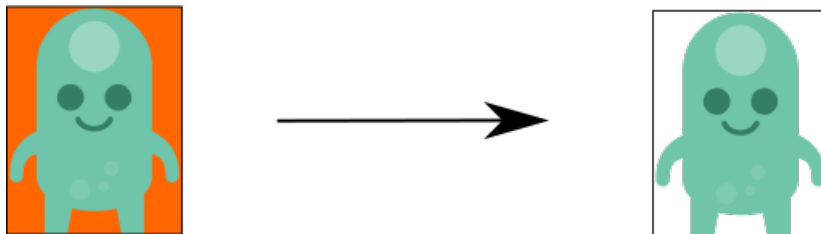


Figure 1.11: Remove texture's background using heuristic mask.

Common divisor extends the size of texture by adding transparent pixels, such that the size of the texture becomes divisible by a given number. This might be used to force the identical size of the textures.

The four tools that were selected to be described in detail and then compared, include three previously mentioned packing tools: TexturePacker, Zwop-tex, SpriteMapper and another tool called GameDevUtils.com¹³. With an intent to provide a more diverse comparison of the selected packing tools, the selection was done in such a way that two tools are commercial and the other two are open source.

Characteristics 6 (Extensibility of packing algorithms) & 7 (Extensibility of image processors) means whether the tool provides a convenient way to be extended, for example, by using plugins. Without this definition, every open source tool would satisfy these two characteristics, because everyone could obtain the source code of the open source tool and modify it in any way.

TexturePacker

TexturePacker can be considered as the most advanced, feature-rich packing tool that is currently available. Unfortunately, this tool is commercial and although there exists a free version, it has only a limited set of features when compared to the PRO version. For example, the PRO version has the following features: *MaxRects packing algorithm, Trimming, command line interface, packing of multiple sprites at once, ability to export metadata in a custom format, etc.*, but none of them is available in the free version. TexturePacker offers both GUI and CLI and whereas the GUI is available in both free and PRO version, the CLI is available only in the PRO version. The GUI of TexturePacker is shown in Figure 1.12.

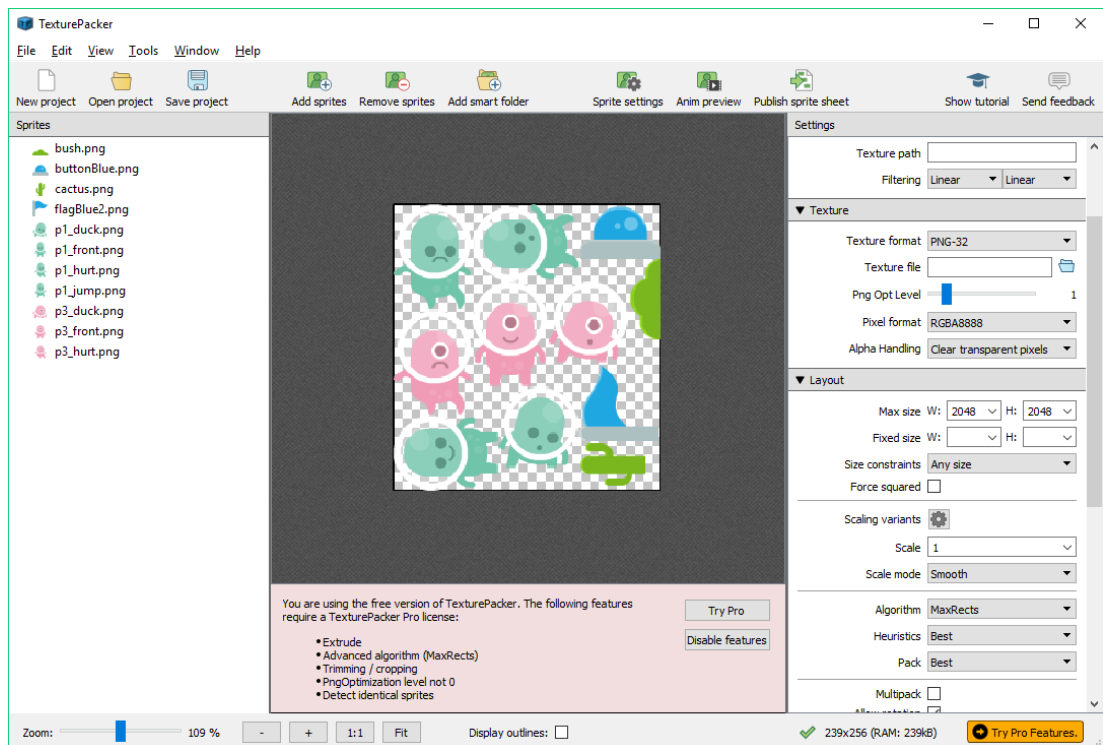


Figure 1.12: GUI of TexturePacker. The screenshot is taken from free version of TexturePacker(v 4.6.3).

¹³GameDevUtils texture packing tool can be found at its website [12]

TexturePacker has the following aspects:

Positive aspects

- Tools for memory consumption reduction
- Great documentation and tutorials
- Maintained and often updated
- Features enhancing productivity
- A lot of frameworks supported
- Multi-OS support
- Animation preview
- Ability to export metadata in a custom format

Negative aspects

- Only two packing algorithms
- Paid, free version is rather limited
- Not extensible by new image processors or packing algorithms

The non-extensibility of the TexturePacker can be seen as quite a large disadvantage, because the game developer may wish to repeatedly process all of the textures in some way that is not allowed by TexturePacker hereby forcing the developer to use separate tools for that. This issue would be eliminated when the developer was allowed to extend the TexturePacker by the given tool he needs. This problem leads to requirement R3: ***Our application should be extensible by allowing the users of the application to create their own:***

- *Packing algorithms*
- *Metadata exportes*
- *Image processors*

Zwoptex

Zwoptex is another commercial packing tool that has features similar to TexturePacker, however, its features are usually not so customizable and there are not so many of them. For example, Zwoptex allows to export metadata in a custom format, but the number of formats that are included in the application out-of-the-box is much lower than in TexturePacker. As with TexturePacker, Zwoptex offers both GUI and CLI. The GUI of Zwoptex is shown in Figure 1.13.

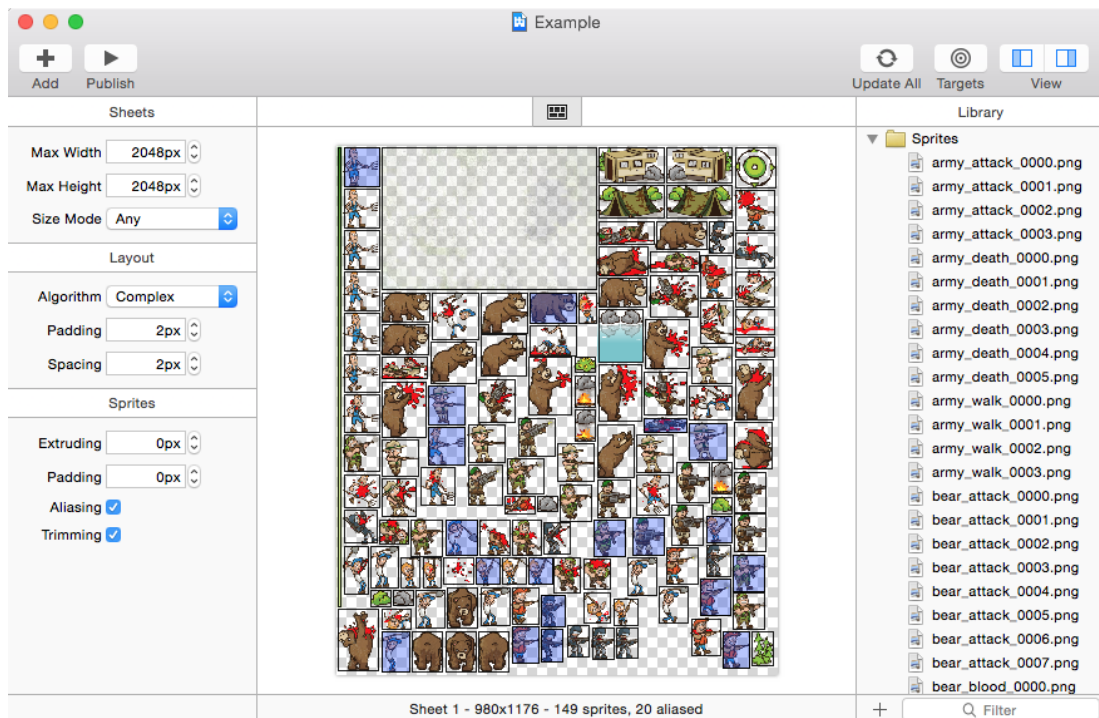


Figure 1.13: GUI of Zwoptex. The screenshot is taken from Zwoptex’s website [10].

Zwoptex has the following aspects:

Positive aspects

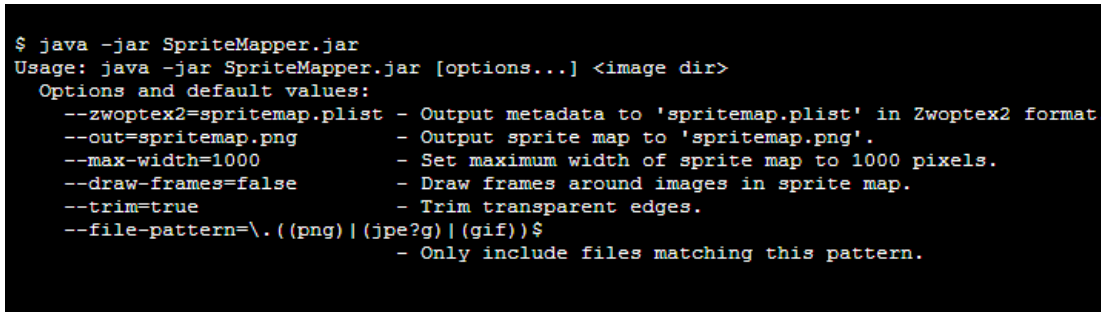
- Cheaper than TexturePacker
- Ability to export metadata in custom format

Negative aspects

- Supports only macOS
- Only two packing algorithms
- Not extensible by new image processors or packing algorithms
- Paid, free version adds a watermark
- Rarely updated

SpriteMapper

SpriteMapper is an open source tool written in Java, that can be controlled either via command line interface or using Ant tasks. The CLI of SpriteMapper is shown in Figure 1.14.

A screenshot of a terminal window showing the command-line interface for SpriteMapper. The text is as follows:

```
$ java -jar SpriteMapper.jar
Usage: java -jar SpriteMapper.jar [options...] <image dir>
Options and default values:
  --zwoptex2=spritemap.plist - Output metadata to 'spritemap.plist' in Zwoptex2 format
  --out=spritemap.png        - Output sprite map to 'spritemap.png'.
  --max-width=1000           - Set maximum width of sprite map to 1000 pixels.
  --draw-frames=false        - Draw frames around images in sprite map.
  --trim=true                - Trim transparent edges.
  --file-pattern=\.(png)|(jpe?g)|(gif))$ - Only include files matching this pattern.
```

Figure 1.14: Using SpriteMapper via its CLI. The screenshot is taken from SpriteMapper’s website [11].

SpriteMapper has the following aspects:

Positive aspects

- Several packing algorithms
- Exports metadata in Zwoptex2 format
- Provides Ant task
- Open source
- Multi-platform

Negative aspects

- No GUI
- Can export metadata in Zwoptex2 format only
- Not extensible by new image processors or packing algorithms.
- Besides trimming, there are no image processors
- Not updated since 2013

GameDevUtils.com

The website GameDevUtils.com offers (besides other tools) texture packing tool, that is written in vanilla HTML5 & JavaScript and runs in the browser. The GameDevUtils.com packing tool is shown in Figure 1.15. The GameDevUtils.com

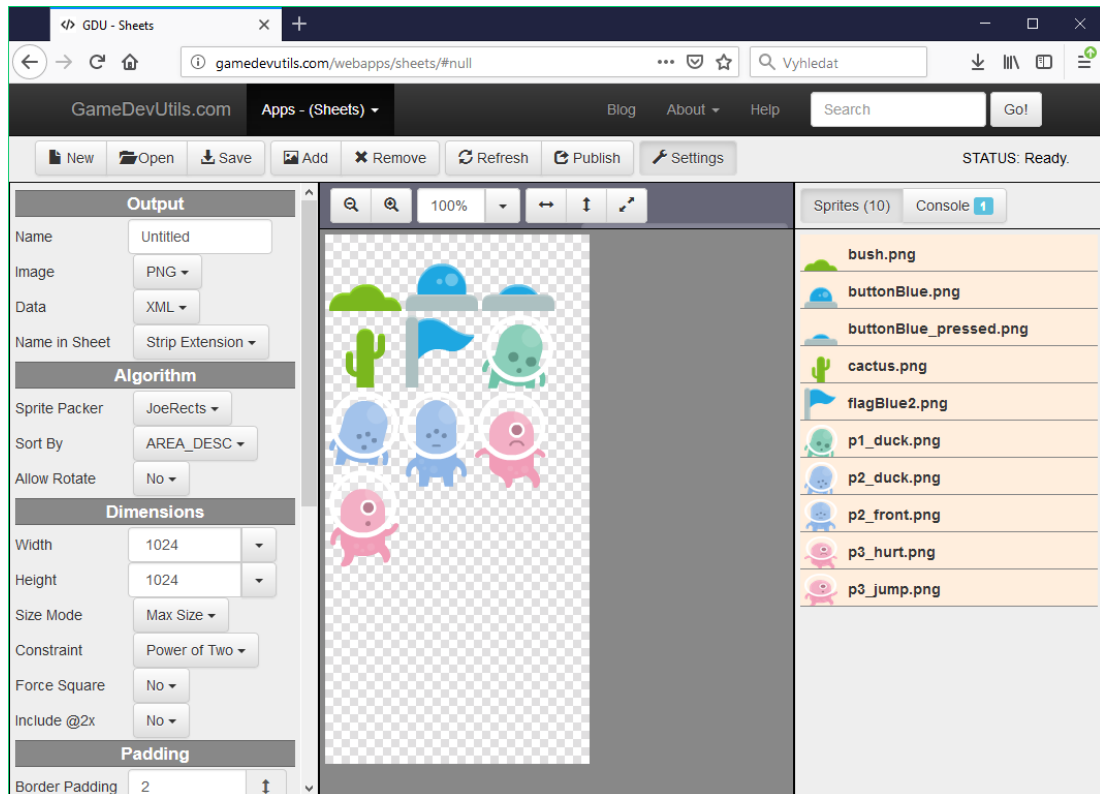


Figure 1.15: GUI of GameDevUtils.com

packer has the following aspects:

Positive aspects

- Runs in the browser, so it can be used from multiple platforms
- Open source
- Several image processors

Negative aspects

- Does not support any framework out-of-box, exported metadata is in their custom XML/JSON/CSS format.
- No additional support for extensibility
- No documentation

To conclude this section, a detailed comparison of the four selected packing tools is shown in Table 1.1.

	TexturePacker	Zwoptex	SpriteMapper	GameDevU- tils.com
Available platforms	Windows, macOS, Linux	macOS	everywhere — written in Java	everywhere — runs in web browser
Image processors	Trim, Crop, Extrude, Padding, Color type change, Heuristic mask, Common divisor	Trim, Extrude, Padding	Trim	Trim, Padding, Heuristic mask
Additional features	Alias creation, Scaling	Alias creation	-	Alias creation
Supported metadata formats	Allows custom format, out-of-box support for metadata of a lot of frameworks	Allows custom format, out-of-box support for metadata of several frameworks	same as Zwoptex, open source license allows for extensibility	XML, JSON, CSS, open source license allows for extensibility
Packing algorithms	MaxRects, Basic (unknown)	Complex (unknown), Basic (that does not try to avoid waste)	MaxRects, Guillotine, Shelf	MaxRects, Shelf
Extensible packing algorithms	✗	✗	no special support, but is open source	no special support, but is open source
Extensible image processors	✗	✗	no special support, but is open source	no special support, but is open source
GUI	✓	✓	✗	✓
CLI	✓	✓	✓	✗
Price	\$39.99 per year, or \$99.99 for a lifetime license	\$10 for a lifetime license	free (open source)	free (open source)
Offers Free Version	✓	✓	✓	✓

Table 1.1: Feature comparison of selected packing tools

As can be seen in Table 1.1 none of the tools can be extended by new packing algorithms or image processors. Notice that of all the compared tools, the SpriteMapper offers the most packing algorithms, namely: MaxRects, Guillotine, Shelf and that most of the tools have MaxRects algorithm included. All of the mentioned algorithms will be described in Chapter 2.

The compared tools also differ in the number of metadata formats that they are supporting out-of-the-box. This criterion is dominated by TexturePacker that supports more than 50 metadata formats for various game frameworks. On the other way, GameDevUtils.com does not support metadata frameworks of any game framework, but instead, it only supports its own metadata formats in XML, JSON, and CSS that the game developer has to parse manually.

Notice, that the most common image processor is Trim as it is available in all four packing tools. Alias creation feature is also quite common because except for SpriteMapper, all the compared tools have it.

The conclusion is that both commercial tools: TexturePacker and Zwoptex, offer more additional features, image processors and generally have more functionality than the two open source tools: SpriteMapper and GameDevUtil.com. The only exception to this observation is the number of packing algorithms that the tools offer, here wins the SpriteMapper with 3 algorithms, instead of 2 that the other tools are offering. The essential observation is, that none of the compared tools allows for extensibility of packing algorithms or image processors.

1.3 Goals

After exploring the available packing tools and researching the situation in the area, we have come up with the following list of goals, that we should satisfy with the implementation of our own packing tool:

1. Create a packing tool with included GUI
2. Implement Trim, Crop, Extrude, Color type change, Heuristic mask
3. Implement Alias creation
4. Provide two sample metadata exporters, first of them targeting libGDX and the second targeting Unity.
5. Provide basic toolset (in a form of the dynamic library) that could be reused for future plugin development
6. Satisfy all of the requirements (R1 to R3) previously mentioned in this section

2. Theoretical Background

This chapter will provide a more theoretical look at the texture packing and its relation to a set of optimization problems that are called packing problems. Although this section is going to be a little bit more theoretical, it is not an intent to present neither rigorous definitions nor mathematical proofs, because it would only make the text more difficult to read. Readers that are already familiar with concepts of 2D packing problems may skip this chapter (except for the problem definition in Section 2.1.2 that is important and partially specific to this thesis) and continue with Chapter 3.

2.1 Packing problems

The terminology in this area is very rich and sometimes may be confusing because many identical packing problems are called by different names, and conversely, many different packing problems are called by the same name. In order to avoid misinterpretation caused by ambiguities in the terminology, a concrete terminology that will be used in this thesis should be introduced. For this reason, the Section 2.1.1 first introduces the typology introduced by Dyckhoff [13]—because even though this typology is not enough in general (as mentioned by G. Wäscher, H. Haußner and H. Schumann in [14]) it is enough for this thesis—and then presents a terminology used in this thesis. In order to give the unambiguous meaning of the terms, every term introduced in the Section 2.1.1 that will correspond to any packing problem will be categorized into a corresponding category from the Dyckhoff’s typology. It is very important to mention, that it can be shown, that bin packing and its related problems are NP-hard. This can be proved by reduction from a vertex cover problem to bin packing (as stated in *Combinatorial Optimization notes* [15]).

2.1.1 Dyckhoff’s typology

H. Dyckhoff published his paper called *A typology of cutting and packing problems* in 1990, with an effort to further develop a “consistent and systematic approach for a comprehensive typology integrating the various kinds of problems” [13]. Purpose of the paper was to unify different notions used in the literature, however, this attempt has not succeeded because his typology was not always accepted as widely as was desirable (the fact that he had introduced abbreviations of German words in his coding scheme probably also contributed to this). Dyckhoff categorizes cutting and packing problems into categories, using the following four criteria [13]:

1. Dimensionality
 - (1) One-dimensional
 - (2) Two-dimensional
 - (3) Three-dimensional
 - (N) N -dimensional, where $N > 3$

2. Kind of assignment

- (B) All objects and selection of items
- (V) A selection of objects and all items

3. Assortment of large objects¹

- (O) One object
- (I) Identical figures
- (D) Different figures

4. Assortment of small items²

- (F) Few items of different figures
- (M) Many items of many different figures
- (R) Many items of relatively few different figures
- (C) Congruent figures

The next Section 2.1.2 defines the packing problem considered in this thesis and the following Sections 2.1.3, 2.1.4, 2.1.5 describe packing problems that are relevant to this thesis and which will be referred to. It should be noted that these packing problems come in several variants which differ in the following factors:

1. Whether the rectangles could be rotated
2. Whether the packing has to be orthogonal
3. Whether the packing has to be guillotinable³
4. Whether the rectangles are known in advance or not

There are more factors than these that just were mentioned, but they are not important for this thesis and therefore they will be omitted. More information about these characteristics could be found in *Algorithms for Two-Dimensional Bin Packing and Assignment Problems* [17].

2.1.2 Problem definition

First, basic term called *geometric condition* should be defined. Geometric condition mandates the following two statements to hold:

1. All small items lie within large objects
2. Small items do not overlap with each other

¹Large objects are containers/empty space.

²Small items are items that are assigned to containers—i.e., the items that occupy empty space.

³Definition of guillotinable packing can be found in *A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing* [16].

This thesis has to deal with a packing of textures and as will be explained in Section 2.1.5, the texture packing is closely related to rectangle packing and to other packing problems that will be mentioned in Sections 2.1.3 and 2.1.4. This close relation and the fact that algorithms for solving these packing problems will eventually be used to pack the textures makes it is reasonable to only deal with variants of the packing problems that are having certain properties. These properties were selected in such a way, that the solution of the packing problem satisfying given properties could be reconstructed (as described in Section 2.3) into a texture atlas. For example, without mandating the property that none of the rectangles could overlap, it would cause that textures in the texture atlas could possibly overlap, which is naturally undesirable. Some of the properties allow producing better packing (packings with the smaller area) without losing the ability to reconstruct the texture atlas. For example, by allowing rotations of the rectangles, the texture could be potentially stored as rotated, but it does not make any problem when reconstructing a texture atlas or when working with the reconstructed texture atlas. The following properties are assumed:

1. Dimensionality $d = 2$
2. All items are assigned into a selection of objects
3. All items are known in advance (offline algorithm)
4. Many items of many different figures (rectangles of different sizes)
5. Orthogonal packing (i.e., only rotations by 90 degrees are allowed)
6. Geometric condition must hold

Objective: There are several possible objectives, but in this thesis either the wasted space will be minimized or the area of the packing will be minimized.

2.1.3 Two-dimensional rectangle bin packing

In this problem, a sequence of n rectangles is given, together with a sequence of m identical bins with sizes $W \times H$, and the goal is to pack the rectangles into a minimum number of bins k so that the geometric condition and other statements from 2.1.2 hold. A more detailed description of this problem can be found in Lodi [17]. From now on, two-dimensional rectangle bin packing will be referred to as a “bin packing”. The illustration of bin packing is shown in Figure 2.1.

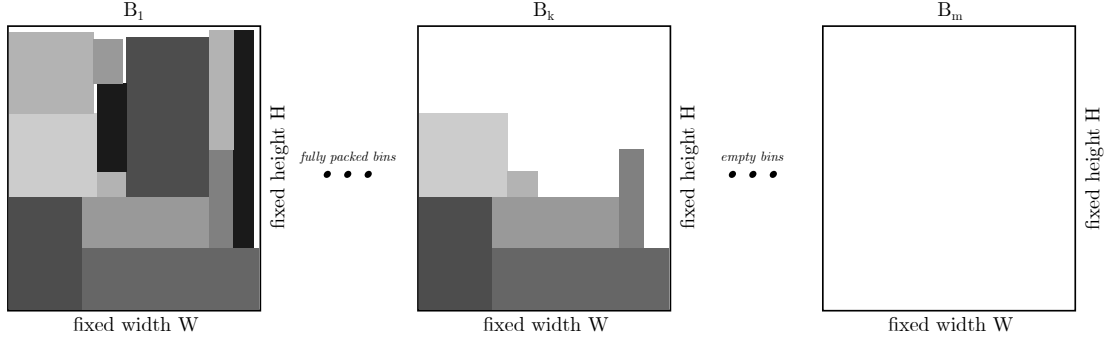


Figure 2.1: Illustration of bin packing

Figure 2.1 shows a solution to bin packing with m bins: B_1, \dots, B_m where first $k - 1$ bins B_1, \dots, B_{k-1} are fully packed and k -th bin B_k is half-packed. The remaining $m - k$ bins: B_{k+1}, \dots, B_m are empty. The variation of this problem that will be considered for this thesis belongs to type $2/V/I/M$ of Dyckhoff’s typology.

2.1.4 Two-dimensional strip packing problem

In this problem, a sequence of n rectangles is given and the goal is to pack the rectangles into a strip of fixed width W and unlimited height, such that the height H is minimized and the packing satisfies the geometric condition and other statements from 2.1.2. A more detailed description of this problem can be found in *A Survey On Heuristics For The Two-Dimensional Rectangular Strip Packing Problem* [18]. From now on, two-dimensional strip packing will be referred to as a “strip packing”. The illustration of strip packing with a minimal height of H is shown in Figure 2.2.

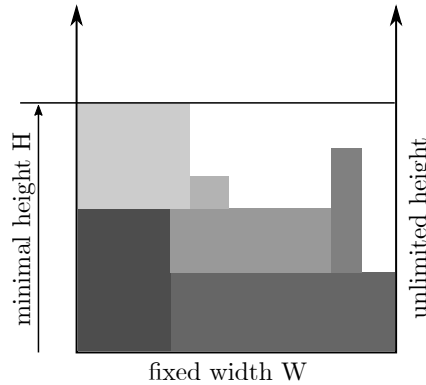


Figure 2.2: Illustration of strip packing

The variation of this problem that will be considered in this thesis belongs to type $2/V/O/M$ of Dyckhoff’s typology. For more details about this problem, see Dyckhoff [13] and Lodi [17].

2.1.5 Rectangle packing problem

In this problem, a sequence of n rectangles is given and the goal is to pack the rectangles into a single larger rectangle of unknown size, such that the area $W \times H$ of an enclosing rectangle (with size $W \times H$) is minimized and that the geometric condition together with other statements from 2.1.2 hold. Stated in other words: the goal is to find an enclosing rectangle of a minimum area that will contain all of the n rectangles and that will satisfy the statements from 2.1.2. The variation of this problem that will be used in this thesis allows (but does not mandate) the rectangles in the input sequence of n rectangle to contain rectangles of different sizes. The aforementioned variation of this problem belongs to type $2/V/I/M$ of Dyckhoff’s typology—notice that this is exactly the same type of Dyckhoff’s typology as for bin packing problem and although the two problems are not the same, in the Chapter 3 will be an explanation that rectangle packing can be reduced to the bin packing (and also to strip packing). More information about (optimal) rectangle packing can be found in *Optimal rectangle packing: Initial results* [19]. The illustration of rectangle packing is shown in Figure 2.3, where 6 rectangles are packed into a single rectangle of size $W \times H$.

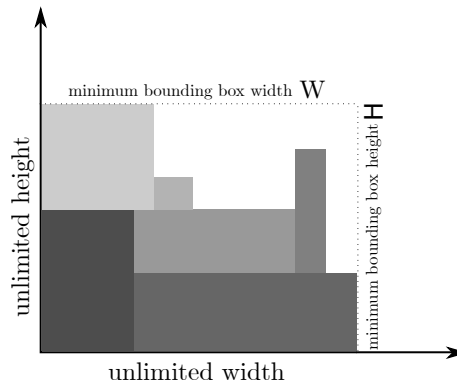


Figure 2.3: Illustration of rectangle packing

Relation to Texture Packing

Now, when the rectangle packing has been defined, it should be mentioned how the texture packing relates to general bin packing problems. A problem of packing the textures together could be solved by replacing each texture with its corresponding rectangle—i.e, with the rectangle that has the same dimensions as the texture—and solving the rectangle packing problem on these rectangles. A solution to the rectangle packing problem then has to be reconstructed to a texture atlas. In order to be able to perform such a reconstruction, each rectangle should remember to which texture it corresponds and the reconstruction will work in such a way, that each rectangle inside the solution to rectangle packing problem

will be replaced back by the texture it corresponds to (possibly rotating the texture, if the rectangle is rotated in the rectangle packing problem solution). The illustration of this process is shown in Figure 2.4.

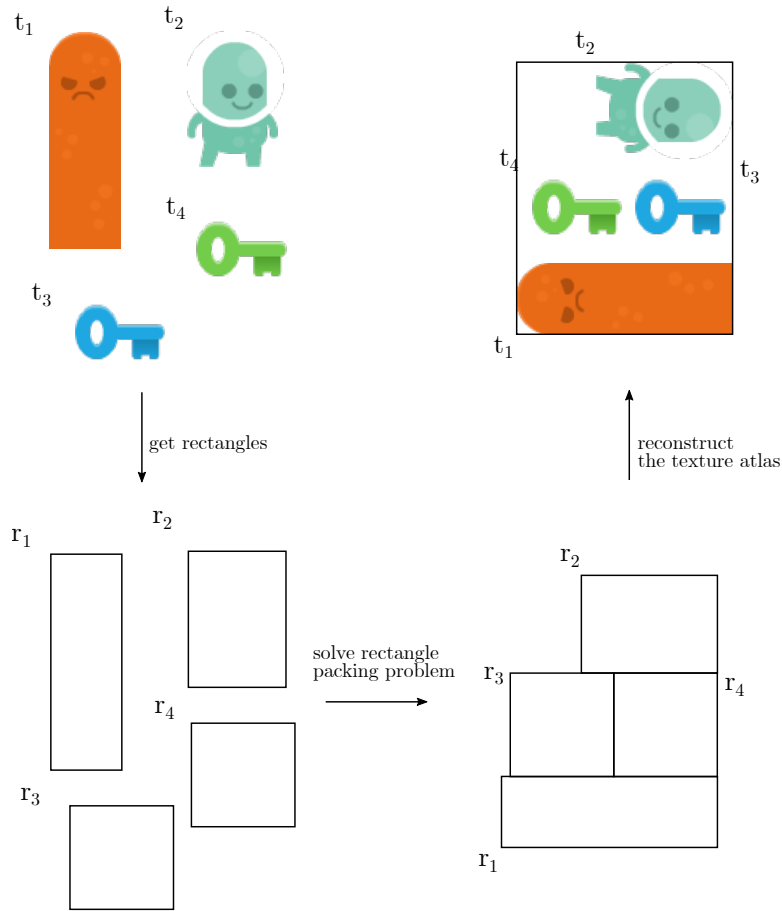


Figure 2.4: Illustration of converting a texture packing problem to rectangle packing problem

On the left of the Figure 2.4 there are 4 textures t_1, t_2, t_3, t_4 that are first replaced by their corresponding rectangles r_1, r_2, r_3, r_4 respectively, as can be seen on the bottom of the Figure 2.4. These corresponding rectangles are then used as the input for rectangle packing problem whose result can be seen on the bottom-right of this figure. Finally, the result of the rectangle packing problem is reconstructed into the texture atlas that can be seen on the right of Figure 2.4.

2.2 Solution approaches

As already stated in the previous section, packing problems—at least those that were mentioned above, in Sections 2.1.3, 2.1.4 and 2.3—are NP-hard [15], therefore with an increasing number of rectangles, finding an optimal solution becomes impractical. Rather than trying to find an optimal solution, heuristics and meta-heuristics are used to find a (possibly) non-optimal solutions. The solutions found by heuristic and meta-heuristic algorithms do not have to be optimal (and usually are not), but the results are usually good enough in practice and can be found in a reasonable amount of time.

2.2.1 Heuristic algorithms

Heuristic algorithms are algorithms which—unlike exact methods that find optimal solutions—are used to find approximate solutions for a given problem, therefore they are dependent on a given problem (as stated in *Metaheuristics From Design To Implementation* [20]). Another property of heuristic algorithms is that they are generally not able to detect whether the found solution is optimal, or how far (measured by some metric function) from the optimal solution it is. The heuristic algorithms that are commonly used for the packing problems are the following:

- Bottom-left algorithm (BL)
- Bottom left-fill algorithm (BLF)
- Improved bottom-left algorithm (improved BL)
- Shelf algorithm
- Guillotine algorithm
- Maximal rectangles algorithm
- Skyline algorithm

There exist several other heuristic algorithms for finding solutions to packing problems that will not be described in this section. Instead of explaining all the algorithms, only the algorithms that are commonly used (for example, the Maximal rectangles algorithm is very frequently used in texture packing tools, as was shown in Section 1.2) will be described.

Bottom-left algorithm

The essential idea of this algorithm is to place each rectangle at a lowest and leftmost position in the larger (bounding) rectangle. Consider an input sequence of n rectangles r_1, \dots, r_n . At i -th step the algorithm takes a rectangle r_i , sets its starting position to the top-right corner of the bounding rectangle and then tries to place r_i at the lowest and leftmost position inside the bounding rectangle. The placing is done by alternately moving the r_i down and to the left. During the placement of r_i , each movement is extended for as long as possible, i.e., when the rectangle is moved to the left, it is moved to the left as long as possible (until it touches either the edge of the bounding rectangle or any other rectangle that was already placed). When the placement succeeds (i.e., when there is a space to place r_i into the bounding rectangle) next rectangle r_{i+1} is taken and this process repeats by trying to place the rectangle r_{i+1} . If the placement of r_i has

failed, then this sequence of rectangles cannot be packed into a given bounding rectangle (this can only happen when the bounding rectangle has fixed size). This algorithm was proposed by Baker et al. [21] in 1980 and was proven to have a time complexity $O(n^2)$ [21].

Bottom-left-fill algorithm

The bottom-left-fill algorithm is similar to BL algorithm in a sense that it also tries to place the currently selected rectangle at the lowest and leftmost position in the bounding rectangle. But unlike BL algorithm, that does not allow to place into so-called "holes" (spaces that are surrounded by the rectangles from all its sides as illustrated in Figure 2.5), this algorithm allows to place rectangles into these holes. The ability to place rectangles into holes allows to create packing results with a smaller area, but it comes with a price of worse time complexity. This algorithm was proposed by Chazelle in 1983 [22] and it is proven to have a time complexity of $O(n^3)$.

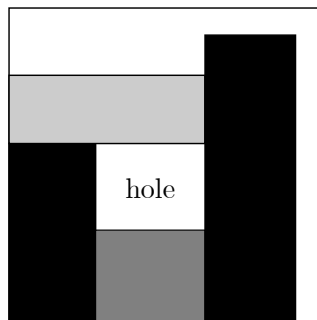


Figure 2.5: Illustration of a "hole" in BL-fill algorithm.

Improved bottom-left algorithm

The improved BL algorithm is same as BL algorithm except that it prefers to move the rectangles downwards during their placement and when the currently placed rectangle cannot be moved further to the bottom, it is moved to the left, but only as long as needed to start moving it downwards again. This algorithm was proposed by Liu and Tang in 1999 [23] and it is proven to have a time complexity of $O(n^2)$.

Shelf algorithms

The algorithms in this group of algorithms work in a way, that the rectangles are placed onto so-called "levels" inside the bounding rectangle. Levels are simply sub-rectangles of the bounding rectangle that have the same width W as the bounding rectangle, but a different (except for the case when there is only a single level) height. Suppose a sequence of n rectangles r_1, \dots, r_n that should be packed and that for any rectangle r_i its width and height is denoted by $w(r_i)$, $h(r_i)$ respectively. Algorithm starts by creating a first level L_1 with height equal to $\min(w(r_i), h(r_i))$ at the bottom of the bounding rectangle. At each step of the algorithm the topmost level L_j is said to be "open", while all the levels below L_j are said to be "closed" and the distinction between these two is that the height of

the open level can be increased (because there is only an empty space above the topmost level) but the height of closed levels cannot be changed. When packing a rectangle r_i at the i -th step of the algorithm, there are two cases: First, if r_i fits into a L_j then it is simply placed there (the orientation of the r_i is selected so that the height of L_j is utilized as much as possible), rectangle r_{i+1} is selected and this process repeats by trying to place r_{i+1} . Second, if r_i does not fit into L_j , new (open) level L_{j+1} with a height equal to the $\min(w(r_i), h(r_i))$ is created above the level L_j . The level L_j is closed, r_i is placed there (possibly rotated by 90 degrees, if $h(r_i) > w(r_i)$) then r_{i+1} is selected and the process repeats by trying to place r_{i+1} . The i -th step of this algorithm is illustrated in Figure 2.6.

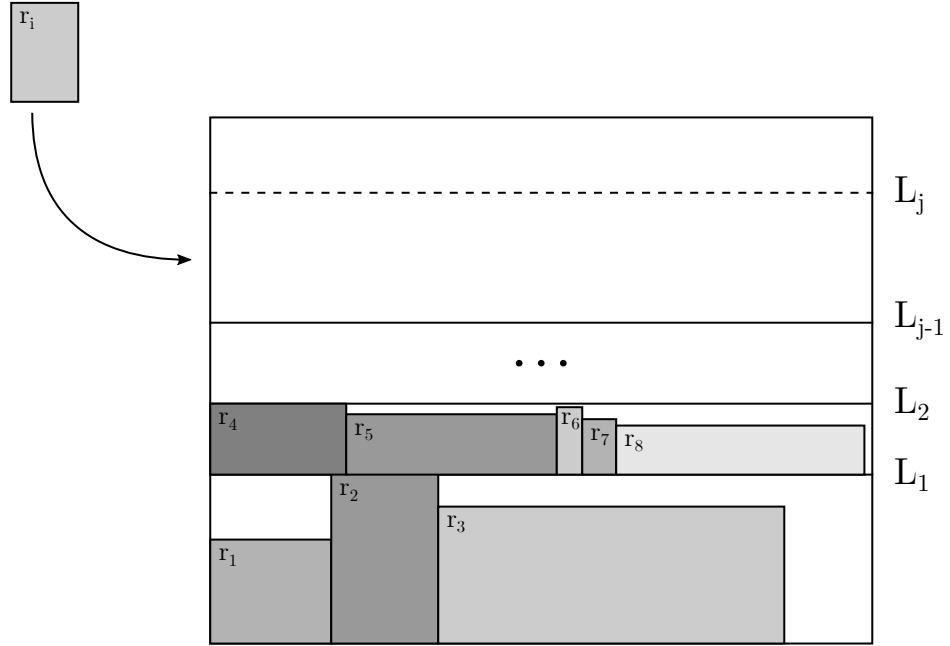


Figure 2.6: Illustration of placing the rectangle r_i .

The figure 2.6 illustrates the i -th step of a Shelf algorithm, that is, the step when rectangle r_i is being placed. Notice that all the levels below the current (open) level L_j are closed. This figure illustrates the variation of a Shelf algorithm that does not even allow to place rectangles on closed levels and where the rectangles are being placed only at the open level L_j (from the left to right).

These algorithms are one of the simplest packing algorithms and they produce worse packing results than all the previously mentioned algorithms. However, they are not useless, because they could be used to address specific practical problems that require the packing to be guillotinable. The algorithms in this group differ by the rules that are used when selecting the best level where a currently placed rectangle should be moved. Another difference between algorithms inside this group is the way they use the ability to increase the height of the open level.

Guillotine algorithms

Guillotine algorithms work in a slightly different way than Shelf algorithms or all the BL algorithm variants that were mentioned earlier. Instead of tracking the

space that is filled by rectangles, these algorithms track the free space. Suppose a sequence of n rectangles r_1, \dots, r_n . At i -th step, a rectangle r_i is going to be placed and it is done by selecting a free rectangle f_j from a set of free rectangles $F = \{f_1, \dots, f_k\}$. The f_j has to be feasible (i.e., the r_i has to fit into f_j) otherwise a different free rectangle has to be chosen. When there is no feasible free rectangle, the input sequence of rectangles cannot be packed into the given bounding rectangle. Initially, there is only a single free rectangle and that is the whole bounding rectangle—this suggests that the size of the bounding rectangle must be fixed and known in advance, but it actually works even for bounding rectangle with unlimited size, because such a rectangle can be thought of as a rectangle with dimensions $\infty \times \infty$. After successfully placing a rectangle r_i into the free rectangle f_j , the f_j has to be adjusted—because a portion of it is now occupied by r_i . The adjustment is done by splitting the $f_j \setminus r_i$ into two free parts—that will be called f_{j_1}, f_{j_2} —by cutting it with either x-axis or y-axis. After the adjustment has been done, f_j is removed from F and f_{j_1} together with f_{j_2} are added into F . Then next rectangle r_{i+1} is selected and the process is repeated by trying to place r_{i+1} . Figure 2.7 shows a possible splits of free rectangle f_j , after placing a rectangle r_i inside it. The algorithms in this group differ by the rules that are used when selecting the split axis, free rectangle and the position where the r_i will be placed inside f_j .

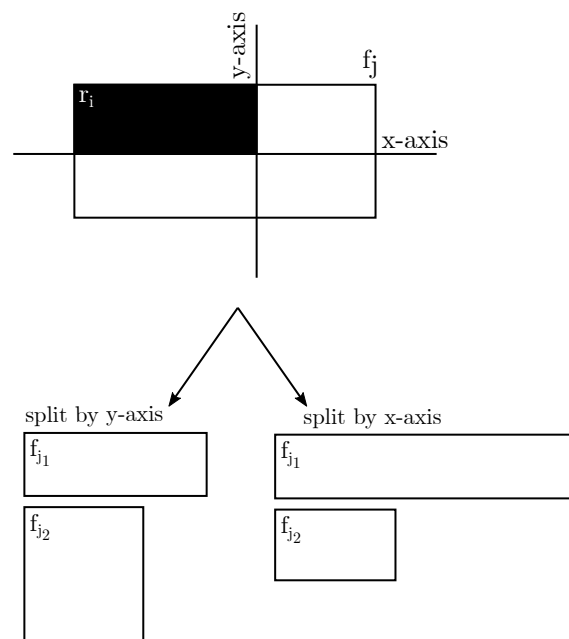


Figure 2.7: Illustration of splitting the free rectangle f_j with a x-axis and y-axis separately.

The left side of Figure 2.7 shows a result of splitting the free rectangle f_j by cutting it with x-axis and the right side of Figure 2.7 shows the result of splitting the same free rectangle, but now cutting it by y-axis instead of the x-axis.

Maximal rectangles algorithms

Maximal rectangles algorithms work in a similar way to Guillotine algorithms, but instead of splitting only by a single axis, the split is done by both axes as

can be seen in Figure 2.8. Performing both splits at once gives two (assuming that the r_i is placed into a corner of a free rectangle, otherwise, it would be four free rectangles) free rectangles f_{j_1} , f_{j_2} that apparently does not have an empty intersection. Because the free rectangles inside F are not disjoint anymore, they have to be adjusted every time some rectangle r_i is placed into a free rectangle f_j by updating all the free rectangles $f_k \in F, f_k \neq f_j$. The update of the rectangle f_k is done by splitting the f_k into at most four rectangles—one to the left of r_i , one to the right of r_i , one above r_i and finally, one below r_i . Because this process may be more difficult to understand, it is illustrated in Figure 2.9. After splitting r_k into four rectangles (call them a_1, a_2, a_3, a_4) the set of free rectangles F is updated to $F = (F \setminus r_k) \cup \{a_1, a_2, a_3, a_4\}$. The final step of this whole update process is to remove all free rectangles f_i for which there exist a free rectangle $f_j, f_i \neq f_j$ such that f_j contains f_i (stated in other words, we remove all the free rectangles that are fully contained within another free rectangle).

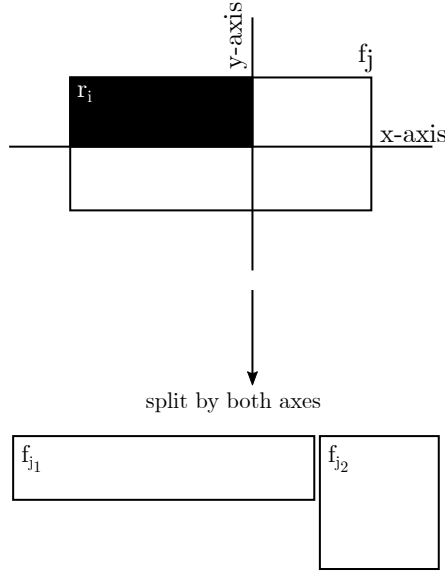


Figure 2.8: Illustration of splitting the free rectangle f_j with both x-axis and y-axis at once.

The Figure 2.8 shows a splitting of free rectangle f_j into two free rectangles f_{j_1} and f_{j_2} by performing the split by both axes at the same time.

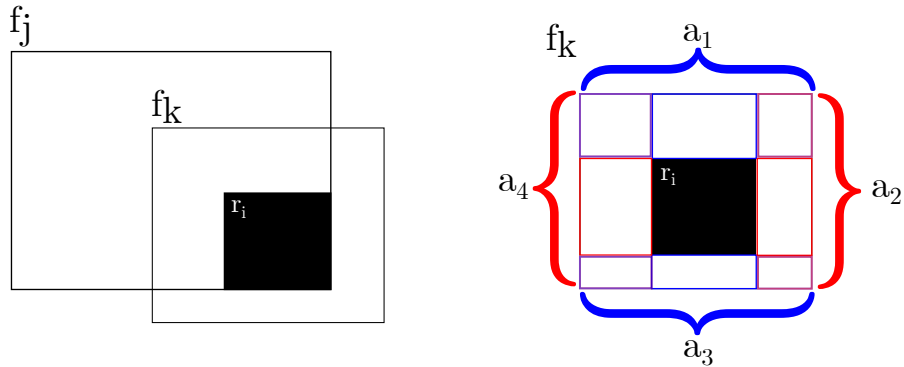


Figure 2.9: Illustration of updating the free rectangle f_k .

Figure 2.9 illustrates the splitting of the free rectangle f_k —rectangle that

has a non-empty intersection with a free rectangle f_j that was selected as a free rectangle where r_i will be placed. This update results in four rectangles: a_1, a_2, a_3, a_4 that will then replace the f_j inside a set of free rectangles F .

The algorithms in this group differ in a similar way the Guillotine algorithms differ among themselves.

Skyline algorithms

Instead of maintaining a list of all free rectangles F like Maximal rectangles algorithms do, Skyline algorithms maintain only a so-called "skyline" (or sometimes called "envelope", which is probably more intuitive, so it will be used here). The envelope is a sequence of lines (these lines are paraxial line segments, i.e. they are either vertical or horizontal), that separates already placed rectangles from the (usable) free space, that is, there is only a single "chunk" of free space. The free space inside this chunk satisfies a condition that it is either above some already placed rectangle r_j or to the right of r_j . The illustration of this idea is depicted in Figure 2.10. When placing a rectangle r_i , it can be placed only on the envelope and after it is placed, the envelope has to be updated. The question is, where the rectangle r_i should be placed on the envelope. Different algorithms from this group of algorithms answer this question differently, for example, some algorithms use a notion of so-called "feasible position" that are the positions where the slope of the envelope changes from vertical to horizontal. This approach is used, for example, by L. Wei et al. [24].

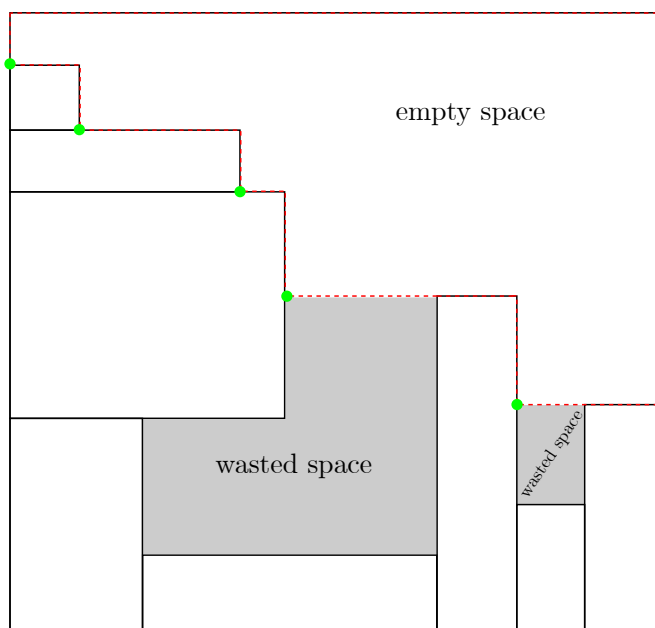


Figure 2.10: Illustration of the notion of envelope together with feasible positions⁴

Figure 2.10 shows an envelope (red dashed line) and 5 feasible positions (green dots). The free area is the white area above the envelope. Notice that this algorithm tends to create a wasted area, that is, the area below the envelope, because when a rectangle r_i is placed, only the points on skyline are considered (in other words, every rectangle r_i can be placed only at such a position that all

⁴Feasible positions in the sense they are used by L. Wei et al. [24].

the rectangles that were already placed are either to the left or below the placed rectangle r_i). Considering only points on the envelope dramatically simplify the check whether a rectangle can be placed on a given position because it simply suffices to check whether the placed rectangle oversteps the border of the bounding rectangle or not. Skyline algorithms generally produce worse packing results (because of the reason that the rectangles can only be placed on the envelope) than Maximal rectangles algorithms, but they are faster.

Remark: Notice, that all of the previously mentioned algorithms depend on the order of input rectangles. The dependence on the order of input rectangles is best seen on BL algorithm, as illustrated in Figure 2.11.

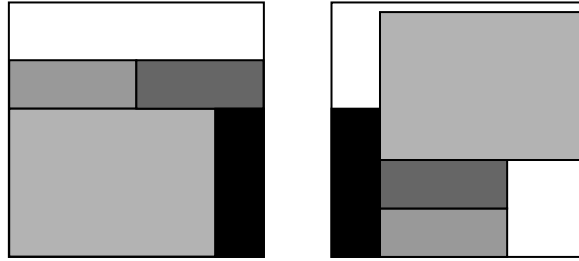


Figure 2.11: Dependence of packing results from BL algorithm on the order of input rectangles

On the left of Figure 2.11 there is a result of packing a sequence of n rectangles, sorted in descending order by their height. The right side of Figure 2.11 shows a result of packing the same sequence but this time, sorted in ascending order by their width. Notice that the packing on the left is much better than the one on the right.

2.2.2 Meta-heuristics

Meta-heuristics are general—i.e., unlike heuristics, they are not specific for a particular problem—methods for searching the space of solutions (as stated in [20]). Meta-heuristics that are commonly used for solving packing problems are the following:

- Genetic algorithms
- Simulated annealing
- Tabu Search

The advantage of the aforementioned meta-heuristics is that when compared to local search algorithms such as hill climbing, these meta-heuristics can escape from local optima. The prevention of getting stuck in the local optima is usually done by temporarily accepting a worse solution, for example, at each step, Tabu Search allows to step into a worse state when no improving step is available. The

description of these meta-heuristics is out of the scope of this thesis, moreover, the meta-heuristics are general methods that are not strictly linked to rectangle packing problems, therefore it seems reasonable to omit their explanation here. However, it should be mentioned, that when these meta-heuristics are used to solve packing problems, the algorithms from Section 2.2.1 are frequently used as a subprocedure when building the space of (temporary) solutions that the meta-heuristic searches through. Description and additional information about these meta-heuristics can be found for example in *Metaheuristics From Design To Implementation* [20].

3. Implementation Analysis

This chapter analyses a possible decisions that have to be made when implementing application for texture packing satisfying all the goals mentioned in Section 1.3 together with potential problems that could arise during the development of the application. This chapter presents an overview of the application's architecture along with a description of how all the goals from Section 1.3 could be addressed, as well as some possible problems that might arise during the development of such an application, and provides a discussion about the possible ways how to solve these problems. The application that we are implementing was named PaunPacker¹ so it will be referred to by using this name.

3.1 Goals revised

As was already mentioned in Chapter 2, the packing problems are proven to be NP-hard so trying to find optimal solutions is not a good idea in the case of texture packing application, because the PaunPacker should be reasonably fast, otherwise the user would not be very happy if the packing of textures took 10 minutes or so to complete. The most time-consuming operation performed by PaunPacker will be the packing of textures, therefore the packing of the textures should be reasonably fast so it seems appropriate to use heuristic/meta-heuristic algorithms for the packing. This decision is backed by the fact that it is not absolutely necessary to produce optimal packing, instead, a certain loss in the quality of the packing should be allowed in exchange for improved performance. The decision to use heuristic and meta-heuristic algorithms, brings the following question: *Which algorithms from Chapter 2 should be implemented?* To answer this question, all of the algorithms from the list of heuristic algorithms that was given in Section 2.2.1 should be inspected and matched against the goals that are imposed on the PaunPacker. As a reminder, the list contains three variants of BL algorithm, Shelf algorithm, Guillotine algorithm, Maximal rectangles algorithm, and Skyline algorithm.

BL algorithms could be considered as the most fundamental packing algorithms because a lot of advanced algorithms use some of the BL algorithm variants as a sub-routine, so this algorithm should definitely be included in the PaunPacker. The concrete variant of BL algorithm that should be implemented is not so important, because each of them has certain benefits so there is not a totally bad choice. Classic BL algorithm has a good time complexity and is quite easy to implement. Improved BL offers the same time complexity and produces packing results that have either better quality or the same quality. The BLF produces better packing results but has worse time complexity. It was decided to implement improved BL algorithm because it is used more frequently by other algorithms (it should be noted that some authors even call this algorithm simply BL algorithm) and it provides a great trade-off between performance and time complexity.

¹This name does not have any special meaning, it is only the author's belief that giving the application unique name is better than simply naming it "Texture Packing Application".

Shelf algorithms generally are not much useful for texture packing, mainly because of the two following reasons:

1. They produce worse packing results than all the BL algorithms and even though they have better time complexity, the time complexity of BL algorithms is considered to be good enough.
2. They do not suit the case where the bounding rectangle has dimensions $\infty \times \infty$ very well.

The second reason should now be described in greater detail—the problem is, that when the packing textures, the size of the bounding rectangle is unknown (because the goal is to find a bounding rectangle with a minimal area) and the Shelf algorithm assumes the bounding rectangle to have a fixed size (or at least a fixed width). This assumption could be achieved by saying that the bounding rectangle has dimensions $\infty \times \infty$, but the problem is, that the first level L_1 will have infinite width, therefore this shelf will (possibly) never get closed and all the rectangles will be packed in this single level. Because of the aforementioned reasons, none of the shelf algorithms will be implemented inside PaunPacker.

Maximal rectangles algorithm seems like a must-have, simply because almost all well-known packing tools have an implementation of this algorithm, so omitting this algorithm inside the PaunPacker would look like a huge disadvantage when comparing the PaunPacker to other packing tools. Therefore this algorithm will be included in the PaunPacker.

Although Guillotine algorithm is inferior to Maximal rectangles algorithm (in terms of packing result quality), it is useful to implement it, because Maximal rectangles algorithm could be considered as a slightly modified version of Guillotine algorithm. Therefore by a nifty design that will allow Guillotine algorithm to be parameterized in a certain way (the details about this parameterization will be given later in this chapter). The Maximal rectangles algorithm then could be obtained as a Guillotine algorithm with certain parameters. This seems like a great approach to be taken because it also allows for code reuse so Guillotine algorithm will also be included in PaunPacker.

To provide a more diverse set of algorithms and also to offer something that other well-known packing tools do not offer, another two algorithms will be implemented in PaunPacker. These algorithms are Skyline algorithm and genetic algorithm. Because there is not only one Skyline algorithm but a whole group of them, it should be mentioned that the variant of Skyline algorithm that will be included in PaunPacker is based on the algorithm proposed by L. Wei et al [24]. The genetic algorithm should be implemented because it seems advantageous to include any of meta-heuristic algorithms. The advantage comes from the fact that if the meta-heuristic was implemented in a modular way, then the portion of meta-heuristic that searches the spaces of (temporary) solutions will be reusable in conjunction with other heuristic algorithms as a subroutine (for example, with BL algorithm plugged in). The details about achieving this kind of modularity will be addressed later in Section 4.3. Another advantage of including a meta-heuristic algorithm is the fact that they are iterative, that is, when the process of packing stops in the half, some solution is given on the output (this does not hold for any of the heuristic algorithms that were mentioned in this thesis because

they would simply return no solution at all). A reason for choosing a genetic algorithm and not any of the other two meta-heuristics are the following:

- It could be parameterized easily in several ways (cross over, mutation, population size, etc.)
- It could be easily parallelized (for example, multiple individuals could be evaluated in parallel)

The genetic algorithm that was decided to be implemented is based on one of the first algorithms utilizing the genetic approach (Jakobs 1996 [25]).

The thoughts presented in this section result in a new goal that should be added to the goals listed in the Section 1.3. This new goal is to implement Improved BL algorithm, Guillotine algorithm, Maximal rectangles algorithm, Sky-line algorithm, and genetic algorithm. The implementations of these algorithms should adhere to the requirements that are imposed on PaunPacker, specifically to the requirement that algorithms should be extensible. Therefore the algorithms should be implemented in a modular way that will allow parameterizing individual parts of each of the algorithms. This degree of modularity also supports the goal to provide a toolset for creating new algorithms because the individual parts of the algorithms could be reused for algorithm development.

3.2 Architecture overview

Before starting the whole development process, it is essential to create a detailed design of the future application because it will save a lot of problems that would otherwise pop-up during the development of the PaunPacker.

Satisfying the requirement **R3** (requiring the PaunPacker to be extensible by allowing the users to create new packing algorithms, metadata exporters and image processors) that was introduced in Section 1.2 can basically be done in two ways—either by allowing to create and load plugins that will represent the extensible parts (that is packing algorithms, metadata exporters and image processors) or by creating some template language that will allow to extend these parts. The former approach seems better because it gives more flexibility to extend the extensible parts. The slight disadvantage of this approach is that the user wanting to create own parts will have to deal with source code, so there is a little bit more effort required, however, because the target users are primarily game developers, this disadvantage becomes negligible. The advantage of higher flexibility is due to the fact, that the plugin developer is not limited in any way what the plugin could do. Of course, the plugin developer will have to develop the plugin in such a way that it will respect some prescribed interface but as long as the compliance of this interface is maintained, the developer can do literally anything with the plugin. This does not hold for template language where the developer of the plugin can only use the features that are included in the template language. Because of these reasons, the PaunPacker should address the extensibility by allowing to create and load plugins.

3.2.1 High-level components overview

In order to reduce the amount of duplicated code and to relieve the plugin developers from having to "reinvent the wheel" a basic toolset should be made available for plugin developers. This idea adheres to the goal of having a "basic toolset (in a form of the dynamic library) that could be reused for future plugin development". Therefore this basic toolset should be included in a separate dynamic library and from now on, it will be referred to as **PaunPacker.Core**. Another component should contain the GUI related stuff (having a GUI is also one of the goals). Although the GUI and the basic toolset could potentially be glued together, it seems like a bad practice to do so, because the plugin developer should not be forced to load executable containing parts that are not needed for plugin development (the GUI parts). Moreover, the plugin should not be able to modify GUI in any way, so it should be separated from the GUI. This separation also suggests that all the general functionality should be inside **PaunPacker.Core** and not inside the executable with containing the GUI, so that the plugin developers could use this functionality. The idea to separate the GUI from all the packing related functionality seems very natural and therefore this approach will be taken. The assembly containing **PaunPacker's** GUI will be referred to as **PaunPacker.GUI**. The high-level overview of **PaunPacker** is depicted in Figure 3.1.

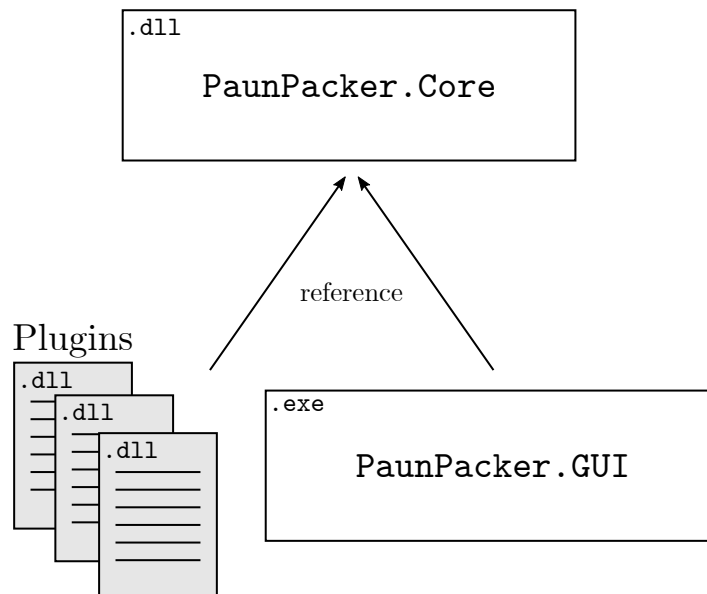


Figure 3.1: High-level overview of **PaunPacker's** architecture

Figure 3.1 shows an overview of **PaunPacker's** architecture. At this high level, the architecture consists of three kinds of components: **PaunPacker.Core**, **PaunPacker.GUI** and the plugins. **PaunPacker.Core** provides a shared functionality that is related to packing (i.e., the parts that are not specific for GUI) and it is in the form of dynamic library (**.dll**). The **PaunPacker.GUI** contains the client application and all the GUI related stuff, this component should be in the form of executable (**.exe**) and it references the **PaunPacker.Core** because the client application needs to use the packing functionality. Finally, there are all the plugins that will get loaded into **PaunPacker**, they are also referencing **PaunPacker.Core** to reuse the provided functionality and also to obtain the re-

quired interface. There are several more questions and problems that should be addressed, namely:

- What is the interface that the plugins should adhere to, to be considered as valid plugins?
- Where in the GUI should the plugins be displayed?
- Which components (apart from image processors, packing algorithms and metadata exporters) should be modular?
- How to differ between all the kinds of modular components?
- How to tell the GUI what parameters (taken from the user) are needed by the plugin in order to display corresponding input elements in the GUI?

All of these questions are going to be addressed in later subsections in this section.

3.2.2 PaunPacker's workflow

Generally speaking, the expected workflow of any packing tool is that the user specifies textures to be packed, select metadata format and sets some additional settings that will affect the process of texture atlas generation (e.g. output format, output path, etc.) and then tell the tool to generate the texture atlas. This simple workflow is depicted in Figure 3.2.

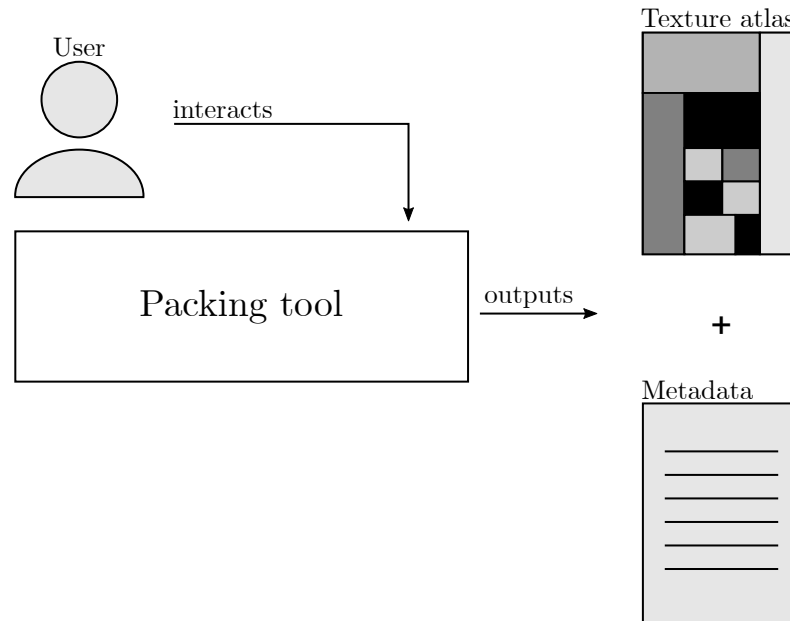


Figure 3.2: Illustration of a general workflow of a packing tool.

PaunPacker will incorporate a similar workflow to the one that was shown in Figure 3.2 and because PaunPacker will include GUI, it is natural to state that all the interaction between PaunPacker and the user should be done via the GUI. The `PaunPacker.GUI` should then process this interaction by utilizing either the `PaunPacker.Core` or the plugins. Obviously, the plugins have to be loaded into client application (`PaunPacker.GUI`). The very important requirement that

should be imposed on the application is to allow the user to interact with plugins via GUI, this requirement comes from the fact that plugins very often allows to set some parameters and therefore the user should be able to use this option. From the requirement that the user should be able to interact with plugins and the fact that all the user's interaction with PaunPacker should be done via the GUI, it implies that plugins should be displayed in GUI somehow—the exact way how they should be displayed are going to be discussed in Section 3.5. Figure 3.3 shows the workflow from Figure 3.2 adjusted to PaunPacker in accordance with the discussion above.

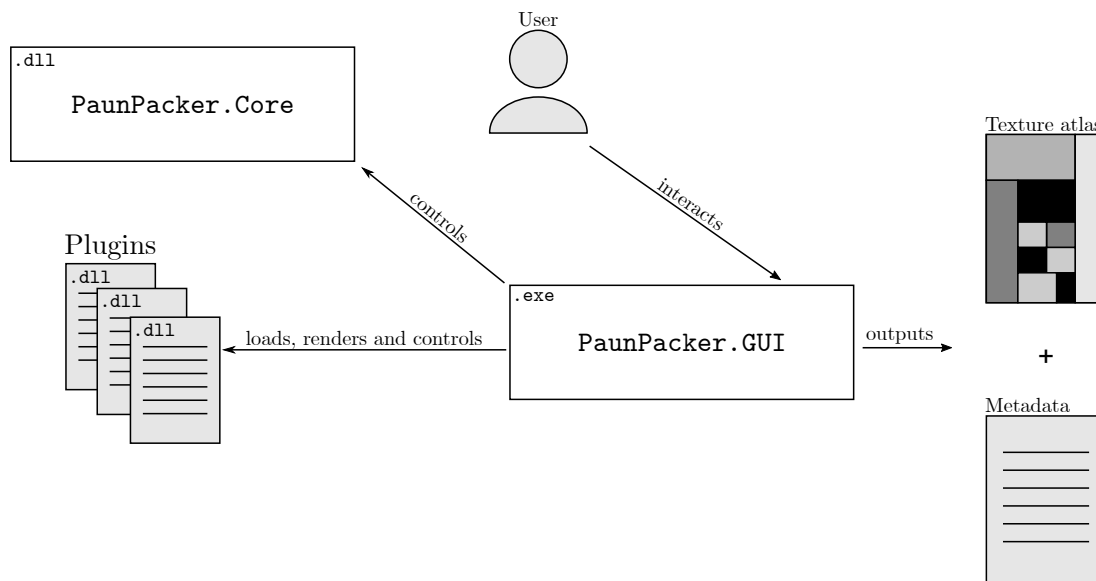


Figure 3.3: Illustration of PaunPacker's workflow.

In the top-center of the Figure 3.3 there is a user that manipulates with `PaunPacker.GUI` which handles the user's interactions and acts appropriately. When `PaunPacker.GUI` is handling the user's interaction, it is using the functionality either from `PaunPacker.Core` or from the loaded plugins. Once the interaction is handled, the result is presented via GUI back to the user. When the user interaction is a request to pack the input textures, the result is a texture atlas together with its corresponding metadata.

3.3 Choice of platform and development technologies

After acquiring an initial architecture of PaunPacker, it is time to choose the right platform and development technologies. When choosing a right programming language, C# over .NET platform seems like a good candidate because it has a very extensive standard library and there exists a lot of other libraries that can be installed via NuGet in a very convenient way. One could argue, that other languages, for example, C++ and Java also provide quite extensive standard library and that is true but a problem with C++ is that it does not offer reflection which will be very useful when dealing with plugins. Although Java could be considered equivalent to C# (at least for a task to develop an application such as

PaunPacker), the C# was preferred over Java because the author of this thesis has more experience with it. There exist three .NET implementations: .NET Framework, .NET Core and .NET Standard. The differences between these three implementations can be found in MSDN Magazine [26], but what should be mentioned here is that it is preferred to use .NET Standard or at least .NET Core whenever possible, because these implementations offer several advantages compared to .NET Framework—for example, they are open source, cross-platform and allow to create high-performance scalable systems. This recommendation applies especially in the case of `PaunPacker.Core`, where it is very appropriate and desired to use .NET Standard because it will make the `PaunPacker.Core` reusable on multiple platforms. However, it is not always possible to use one of these two implementations, for example, when referencing a dependency (e.g. NuGet package) that is available only for .NET Framework and in such a case, nothing else is left then using the .NET Framework (or choosing a different NuGet package).

In addition to the programming language, a GUI library has to be chosen and because the chosen language is C#, the two major, commonly used candidates are WinForms and WPF because they are both developed by Microsoft and work very well with C#. It was decided to use WPF because of the following reasons:

- It offers better separation of code and layout design leading to a cleaner design.
- It is more flexible than WinForms (for example, control composition).
- It is more modern than WinForms.
- It is resolution independent.

Last technology that was decided to be used in the very beginning of the whole development process was a graphical library called SkiaSharp. This library was chosen because it is cross-platform and it provides a convenient way to work with textures (bitmaps) which is a feature that will be needed for an application such as PaunPacker.

3.4 PaunPacker.GUI

In the Section 3.2.1 it was decided that the PaunPacker’s GUI functionality should be separated into a separate assembly called `PaunPacker.GUI`. This section will describe `PaunPacker.GUI` in greater detail.

The `PaunPacker.GUI` should contain a presentation logic of a whole PaunPacker. This assembly was initially intended to target the .NET Framework and to be platform-specific because the use of WPF brings dependence on Windows but during the development of PaunPacker, revolutionary changes happened. These changes are the release of a preview version of .NET Core 3 that includes support for WPF. The use of WPF together with .NET Core 3 still does not allow to create cross-platform WPF application, but at least it brings the other .NET Core benefits. Therefore it was decided (after some time) to let the `PaunPacker.GUI` target the .NET Core 3.

As it was illustrated in Figure 3.3, the logic inside `PaunPacker.GUI` assembly should be responsible for the processing of interactions that the user issues through the GUI. This processing will be done by delegating the work that has

to be done either to `PaunPacker.Core` or to any of the plugins. When the component to which the work was delegated finishes, the `PaunPacker.GUI` should present the results of the delegated work to the user by displaying these results in the GUI.

Whenever dealing with larger applications, it is very recommended to separate the domain logic from the rest of the user interface and there are several architectural patterns that solve this issue, three most known are: MVC, MVP, MVVM. It was decided that `PaunPacker` should use MVVM and the reason for this decision is the fact that this pattern is very natural for WPF because of its very powerful and excellent concept called `DataBinding`. The architecture of `PaunPacker.GUI` is depicted in Figure 3.4.

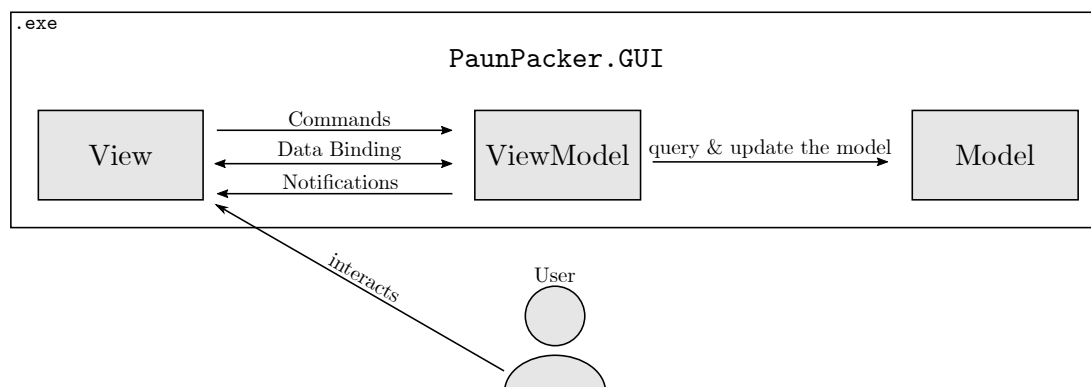


Figure 3.4: Architecture of `PaunPacker.GUI`.

Figure 3.4 shows an architecture of `PaunPacker.GUI`. The `ViewModel` serves as a way to "connect" the `View` with the `Model` and it exposes the data that are needed by the `View`. The `ViewModel` is also responsible for event handling (this is typically done via `Commands`). The `model` holds the data that is queried and updated by the `ViewModel`. The `Model` should not be aware of its `ViewModel`. The `View` contains the controls that should be displayed to the user and in order for `View` to populate these controls with proper data, it fetches all the needed data from `ViewModel` (this is done automatically via `DataBinding`). Notice that the `View` is completely isolated from a `Model`. Although this figure shows only a single view, in reality, there will be several views each of them having its own view model and model. This situation is depicted in figure 3.5.

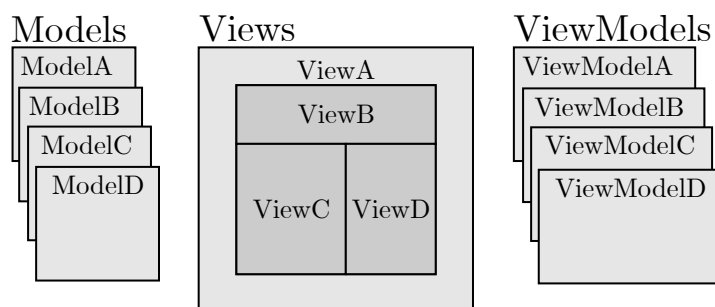


Figure 3.5: MVVM in `PaunPacker.GUI`

Figure 3.5 illustrates a typical scenario with several views available, specifically, there are four views called `ViewA`, `ViewB`, `ViewC`, `ViewD` with their cor-

responding models **ViewA**, **ViewB**, **ViewC**, **ViewD** respectively. Each view with a model also has their corresponding view model, for example, **ViewA** with **ModelA** have their corresponding view model called **ViewModelA**, **ViewB** with **ModelB** have their corresponding view model called **ViewModelB** and so on. The figure also illustrates that views could be composed of other views in the case of this figure, **ViewA** contains all the other views (**ViewB**, **ViewC**, **ViewD**). The ability to compose views together is inherent feature of WPF and it is something very powerful which will be used extensively.

Using WPF with MVVM pattern also makes it possible to completely eliminate code in code-behind, for example, code-behind typically contains event handlers, but when the MVVM pattern is followed thoroughly, the events could be replaced by commands whose implementation is fully contained within a view model. The nice thing about eliminating the code in code-behind is that it makes the view to contain no logic at all. Although there is nothing fundamentally wrong about having a code in code-behind, leaving the code-behind empty is considered as a good practice and it improves maintainability and testability.

3.5 Plugins

When creating modular application—i.e., an application that could be extended by means of plugins that are loaded into the application—there are several questions and that could arise and that should be addressed. These questions are the following:

1. How the plugins should be stored and loaded?
2. How to recognize that an assembly is a plugin?
3. What are the required capabilities that a plugin should have?
4. Which components should be represented by a plugin?
5. How to allow user's interaction with plugins?

This section will discuss possible answers to these questions in the context of PaunPacker.

Representation of a plugin

Plugins are actually dynamic libraries (**.dll**) stored in a separate **.dll** file that eventually gets loaded by an application. The problem is, that in general, the **.dll** file could contain almost "anything", therefore some mechanism to detect what should be considered as a plugin is needed. A commonly taken approach is to create some interface—call it **IPlugin**—that all the plugins should implement. This approach assumes that a plugin could be represented by a single class. Another (declarative) approach would be to use some custom attribute—call it **PluginAttribute**—and use this attribute to mark every class that should represent a valid plugin.

It is often very useful to report some metadata about a plugin to the host² application, these metadata include, for example, plugin version, author, description, etc. The advantage of representing the plugins using the `PluginAttribute` is the fact that metadata is generally known at compile-time and because attributes are compile-time constructs it seems cleaner to declare this information at compile-time rather than delay the evaluation of metadata to runtime. On the other hand, because attributes are compile-time constructs, the approach using the `PluginAttribute` does not allow to obtain information from plugin at runtime and this is a major shortcoming because it is typically required to either allow the host application to query state of the plugin or to call some initialization method on the plugin passing in some parameters from the host application. The initialization method that allows passing parameters for the plugin is very useful because the host application could pass for example an IoC container that the plugin could use to resolve or register dependencies.

In the case of PaunPacker, it is crucial to allow plugins to obtain parameters from the host application because it allows a certain way of communication between the host application and the plugin which brings greater flexibility for plugin developers.

It follows from the foregoing considerations that in the case of PaunPacker, the `IPlugin` should be used. However, in fact, nothing prevents the use of both approaches and this "hybrid" approach is actually the approach that is taken by PaunPacker, where all plugins should implement `IPlugin` interface and (optionally) export its metadata using `PluginMetadataAttribute` this approach brings both the best of the two previously mentioned approach at the cost of slightly longer code.

Storing and loading the plugins

Now, when the way how the plugins should be represented has been defined, it is time to design a mechanism for loading the plugins and to come up with an answer to the question where the plugins should be stored.

The simplest way to load plugins into the host application is to store all the plugin `.dlls` inside a specified directory and let the host application to load all the plugins from the directory (using reflection).

Another approach could be to create some web service where the plugins would be uploaded and then equip the PaunPacker with GUI that would allow installing of arbitrary plugins from this web service. This approach would make the distribution of plugins easy and convenient because anyone would be able to create plugins and upload them to the web service and then anyone would be able to download and use these plugins.

However, the second approach (although generally better) seems like overkill for PaunPacker, simply because it is not expected that there will be thousands of people creating thousands of plugins and also because the web service responsible for managing the storage of plugins would bring some extra maintenance overhead (for an extra cost). So even though the first approach is rather simple, it suits the need of PaunPacker very well and it is surely the right way to go.

²The host application is the application that hosts plugins, that is the applications where the plugins are loaded.

When all the plugin `.dll`s are stored inside a specified folder the host application should load all the plugins either at startup or by using some kind of lazy loading. The lazy loading could be used in such a way, that the plugin `.dll` would get loaded the first time it is needed (or possibly unloaded when it is not used for a certain amount of time). Although the lazy loading is a nice feature, it is again an overkill because it is not expected that PaunPacker would load very large amounts of very complex plugins. The expected scenario is to load tens of plugins and this expectation should not restrict PaunPacker in any meaningful way. Another nice feature is to allow to load plugins during the runtime of the host application, but because this feature becomes useful only for applications whose plugins are either very often updated, installed or whose start-up takes several minutes, PaunPacker does not need this feature either.

The loading of the plugins itself simply enumerates all the `.dll` files inside the specified folder loads these files and finds (inside these files) all the classes that implement an `IPlugin` interface. These classes are then loaded and their instances are created by the host application. The loading of `.dll`s, searching of types and their inspection are all possible thanks to a powerful feature called reflection. This approach is depicted in Figure 3.6.

PaunPacker process

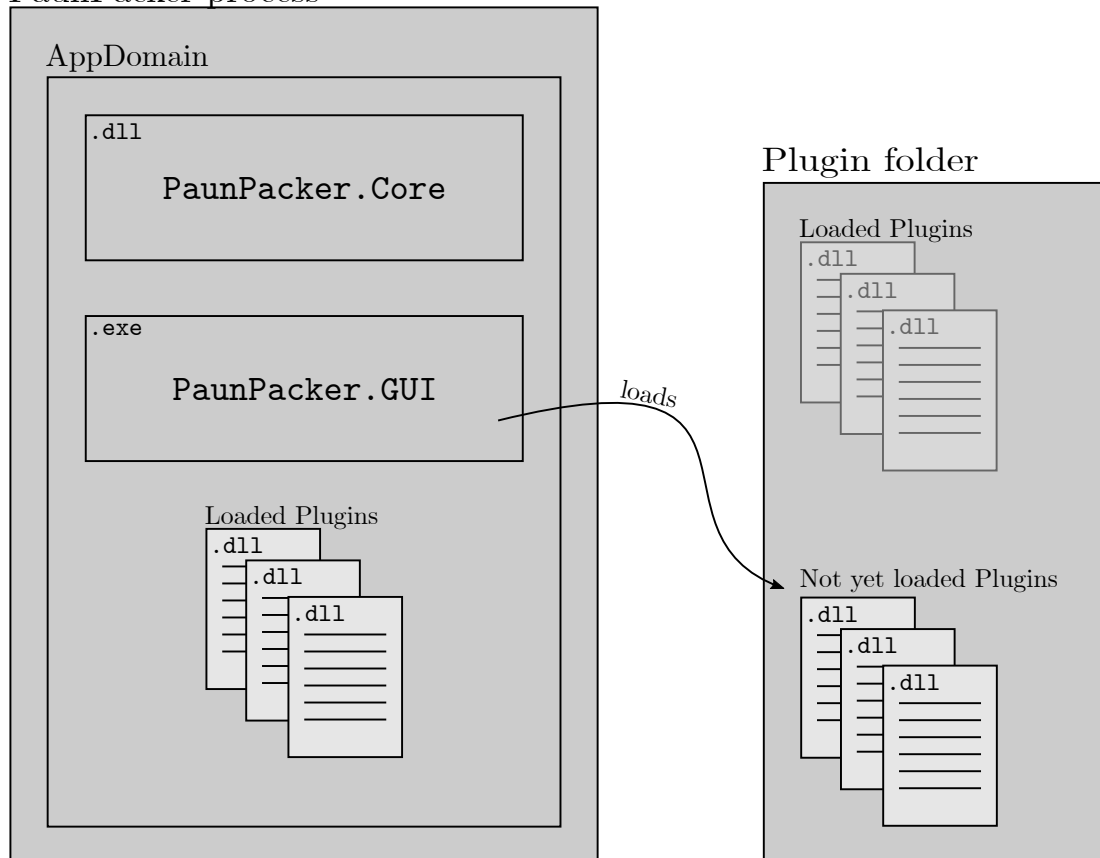


Figure 3.6: Depiction of loading plugins from `.dll`s inside a specified folder

Figure 3.6 shows a simple approach illustrating the `PaunPacker.GUI` that is loading the plugins from a plugin folder. Notice that all the plugins are loaded inside a single AppDomain that is the same as the AppDomain where the PaunPacker itself is loaded.

This approach has some security issues, specifically, the plugins may contain malicious code that will eventually get executed. But this issue is something that has to be accepted because although there exist several techniques to mitigate this issue, none of them works on 100% in 100% of cases. For example this issue is often addressed by creating a separate AppDomain and loading the plugins inside this AppDomain and although this approach brings some benefits—for example, when plugin crashes, the host application does not crash but continues running—it still allows the plugin to execute a malicious code that could do something wrong to the user’s computer. Therefore it was decided to do not address this issue inside PaunPacker.

The previously mentioned issue (although it is serious) is not the biggest problem to be tackled by PaunPacker. To present the issue of greater importance consider a scenario, where the plugin that has to be loaded does not contain parameter-less constructor but only a constructor that requires some parameters (call them dependencies). How should the plugin loader deal with this situation? There is probably only a single way to tackle this issue and that is to create an uninitialized instance of a given plugin, that is, to use reflection to create an instance of a class, without calling any of its constructors, but this is really problematic and it could cause serious problems, because the developer of the plugin may design the plugin in such a way that it will simply do not work without a proper initialization as illustrated in Figure 3.7.

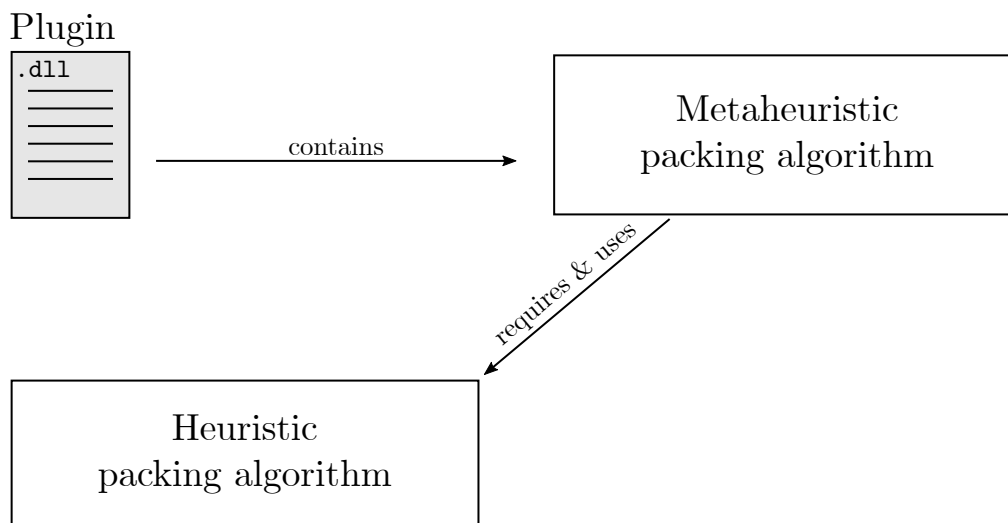


Figure 3.7: Plugin dependence on another class.

Figure 3.7 illustrates a scenario where a plugin contains a metaheuristic packing algorithm that requires (as a constructor parameter) and uses some heuristic packing algorithm. Without the ability to provide an instance of any heuristic packing algorithm to the metaheuristic algorithm, the metaheuristic algorithm could not be initialized properly.

The issue with the absence of parameter-less constructor in plugins could be solved by using an IoC container, but because manual usage of dependency container for this kind of issue would require a lot of code to get everything working, it is better to start directly using a framework called MEF (Managed extensibility framework). MEF is an extension framework that also allows to discover plugins, load plugins and also to satisfy dependencies. So using MEF

will solve the issue with the absence of parameter-less constructor and it will also automatize the process of the loading plugins from a specified directory, therefore the plugins will do not have to be loaded manually anymore. MEF has a capability to automatically discover plugins inside the specified folder, automatically load them and instantiate them. The plugins could specify dependencies that the MEF should give them. However, MEF does not (magically) know what to give to the plugin as these dependencies, so inside the host application, the dependencies first have to be registered.

When it comes to plugin representation, instead of relying on `IPlugin` interface MEF takes a declarative approach and uses `Export` attribute when looking for plugins inside `.dlls`. So all the types that are representing a plugin should be decorated with `Export` attribute. When MEF scans classes inside `.dll` it looks for classes that have this `Export` attribute and try to load them. In addition to `Export` attribute, there also exist `Import` attribute that could be used on class members. It is out of the scope of this thesis to explain how MEF proceeds when it finds any of these attributes, but roughly speaking, when `Import` decorates a member (field, property or constructor parameter) of type A, and the MEF encounters this attribute, it tries to assign an instance of class B that is decorated with `Export(typeof(A))` attribute to the member decorated with the `Import` attribute. More details about these attributes could be found in [27].

User's interaction with plugins

The use of MEF that was proposed in Section 3.5 allows PaunPacker to export dependencies from the host application to the plugins, but this level of plugin parameterization is still insufficient because the plugins often need to obtain parameters that are originated from the user and not from the host application. To illustrate a situation where this need arises, suppose an arbitrary genetic algorithm for packing. It does not matter how the genetic algorithm is implemented, but inherent property of all genetic algorithms, in general, is that they are working with a so-called population³. The population is a perfect candidate for parameterization, instead of hardcoding a fixed size of the population inside the algorithm, it would be very convenient to allow the user to specify this population using GUI of the host application. To allow the user to do so, the plugin has to be rendered in some way. There are basically two ways to do it, the first way is to create a set of custom attributes and use these attributes to indicate that certain parameters should be injected. The host application would then look at a decorated member and based either on the type of the member or on the attribute, would render appropriate controls inside GUI. When the user would fill in these controls, the host app would notify the plugin about values the user has entered and the plugin would be able to use them in an arbitrary way. The problem with this approach is that it is limited to certain types of controls and therefore to certain types of parameters. Concretely speaking, it would be quite easy to render control accepting string/int, but harder to render something that would allow the user to specify Image or something more difficult. Another disadvantage is the increase of host application's logic because a logic for deciding

³Explanation of genetic algorithms is out of the scope of this text therefore it will be omitted and the reader could find an appropriate algorithm in [20].

which control to render and logic for the rendering itself would have to be added. These reasons have to lead to the refusal of this approach.

A more general approach is to let the plugin developer design GUI of the plugin and load the plugin together with its GUI inside the application. This approach also has some problems, notably that it adds the dependency on WPF into the plugin but this should be only considered as a minor (esthetical) problem because the plugins are designed to be used with PaunPacker and PaunPacker will not likely be ported to another graphical library and if yes, only a `PaunPacker.GUI` would have to be ported. Because `PaunPacker.GUI` is also responsible for plugin loading, the ported version of `PaunPacker.GUI` would simply load the same plugins but without their GUI. To clarify this idea, the plugin developer should be allowed to create a view (in a sense of MVVM) for the plugin. The host application will then load the plugin together with the view, instantiate the view and add it to its GUI and the user will then work directly with the GUI of the plugin. The idea of loading the plugin together with its view is illustrated in Figure 3.8.

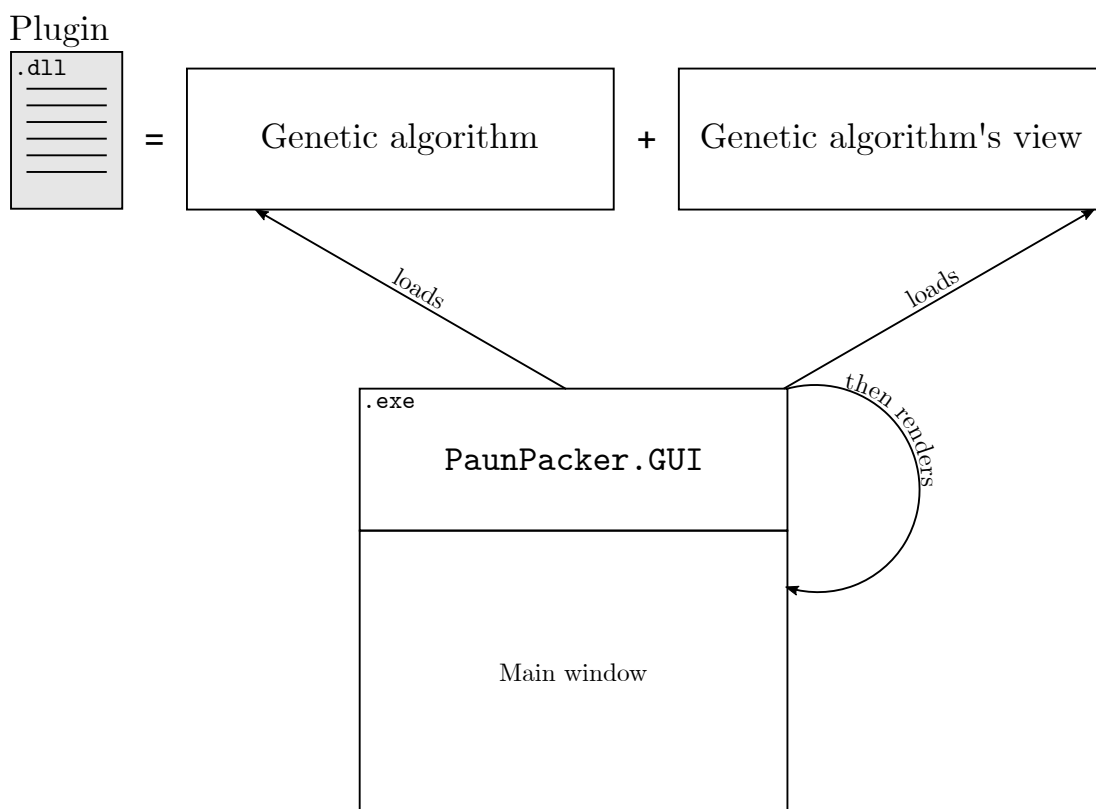


Figure 3.8: Illustration of the idea of loading the plugin's view.

But there is another problem: where inside the GUI of the host application should the view of the plugin be placed? Because PaunPacker should be able to load various kinds of plugins (metadata writers, packing algorithms, etc.) it should display each kind of plugin in a separate part (region) of the GUI. In order to do this, the GUI of the host application should contain a region where the view of a plugin could be rendered and because there are different kinds of plugins there should also be different regions—displaying different kinds of plugins at different places in the application seems like a reasonable thing to do. Although the creation of regions and loading of the plugin's views into these regions could

be done manually, it would be quite cumbersome, therefore, it is better to use some library that is capable of doing it automatically. One such a library is Prism Library and even though there certainly exists other libraries, this library seems like a perfect fit for PaunPacker because it offers great support for WPF, it is very actively updated and developed, offers nice documentation and it is open source version of Prism guidance that was originally proposed by Microsoft. And actually, the Prism is a framework for creating modular (loosely coupled) applications as PaunPacker certainly is. Moreover, Prism could be efficiently used together with MVVM pattern that is also used within the PaunPacker. The idea of loading plugin's view and registering it inside a certain region is illustrated in Figure 3.9.

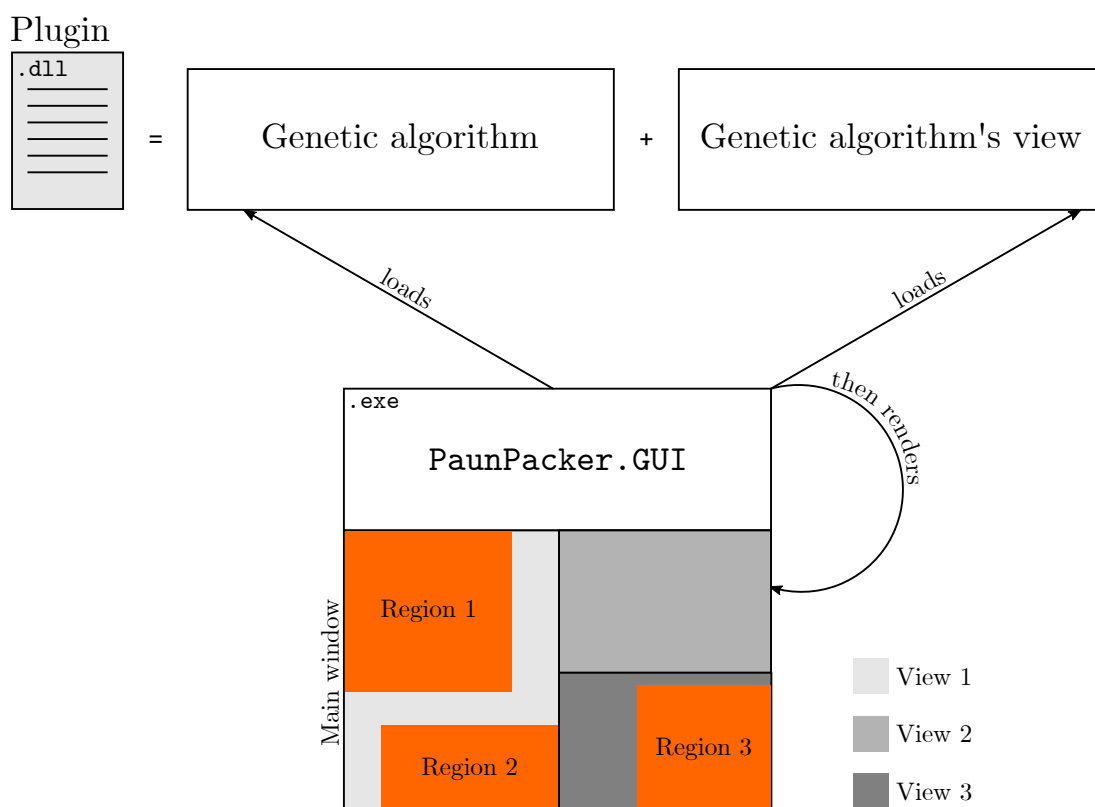


Figure 3.9: Illustration of the idea of loading the plugin's view and registering into a region.

Figure 3.9 illustrates a scenario where the **PaunPacker.GUI** loads a plugin together with the view of the plugin and then renders this view into one of the regions within the main window. Notice that the main window consists of several views and each view contains a different number of regions (or no region at all). In this case, the view is rendered into the third region.

It is important to mention that Prism also offers **IModule** interface which more or less corresponds to the "fictive" **IPlugin** interface introduced in this chapter, therefore, their names will be used interchangeably, but in the actual code of the application the Prism's **IModule** will be used.

To summarize this Section: the plugin developer is able to provide the plugin together with a view and could let the **PaunPacker.GUI** to render the view in order to obtain certain parameters from the user (via the user's interaction

with `PaunPacker.GUI`). The plugin developer could use these parameters inside the plugin's view model (because it could correctly handle the user's interaction with a view). But there is still yet another problem: the view is typically present in order to parameterize the plugin (and not for some internal use) and if this is the case, then when `PaunPacker.GUI` loads the plugin it should obtain the instance that is correctly parameterized and not a non-parameterized instance. This is quite a problem because MEF's `Export` attribute creates a new instance so the parameterization would get lost. There is a way to deal with it (although it might be seen as a MEF misuse) and that is to use a method called `ComposeExportedValue` on a MEF's container. This method allows to export an existing instance (e.g. with parameters) but the problem is that MEF will import such an instance only once, so if the parameterization changes in the future (during the runtime) these changes will not be reflected on this instance. After all these problems it was concluded that MEF is not probably strong enough for this kind of task and it was decided to use some pure and more powerful IoC container. Because `PaunPacker` is using Prism, it was decided that the used IoC container will be Unity (because Prism works very well with Unity, even better than with MEF as comes from the fact that Prism does not support MEF in its latest release version). Unity allows to register a factory that will create an instance of a type that should be exported from the plugin (with proper, up-to-date parameters) and this factory will be called every time the resolution is requested (the resolution could be invoked by some user interaction). The technical details of how this could be implemented will be given in Chapter 4.

3.6 PaunPacker.Core

This is the main assembly that should contain the toolset for creating plugins. This toolset includes: packing algorithms and their interfaces, interfaces for image processors and metadata writers and other useful types that will be described in developer's documentation (Chapter 4).

The main purpose of this assembly should be to equip plugin developers with a toolset that they could use when creating new plugins. Even though all the functionality from this assembly could be stored inside `PaunPacker.GUI`, the first approach seems better because the second approach (storing the plugin development toolset inside `PaunPacker.GUI`) has the following disadvantages:

- It forces the developer to reference an assembly that contains a lot of functionality that is not needed for plugin development (for example, the GUI related stuff). This was already mentioned in Section 3.2.1.
- Because WPF is still somewhat Windows-specific and it is linked to .NET Framework (actually as the time of writing this thesis there is a brand new .NET Core 3 preview that allows WPF development targeting .NET Core) therefore it would not be possible to target .NET Standard and whole `PaunPacker` would require the .NET Framework or the .NET Core 3.

The compliance with the .NET Standard makes it easy to port `PaunPacker` to other platforms or to create a new user interface, for example, it may be useful to create cross-platform CLI for `PaunPacker` in the future, and with the

separation of `PaunPacker.Core` and `PaunPacker.GUI`, doing so would be quite easy because the new (CLI) application would be able to reference and use the functionality inside `PaunPacker.Core`. It would even be possible to create a web interface (using ASP.NET) that would be able to use the functionality exposed by `PaunPacker.Core` assembly.

Adapting general packing algorithms to texture packing

The initial attempt was to create an interface that would represent a packing algorithm (call it `IPackingAlgorithm`). This interface was supposed to have a single method (call it `Pack`) that would take the textures on the input and return the texture atlas on the output. However, it turned out that this approach was a pretty bad idea for several reasons.

The first reason is that it is not a good idea to restrict the packing algorithm only for textures because the texture packing is actually based on rectangle packing, it seems more appropriate to adjust `IPackingAlgorithm`'s `Pack` method to accept rectangles instead of textures. This adjustment brings an advantage for unit testing or benchmarking because it is usually faster and more convenient to generate a huge amount of random rectangles than generate/load a huge number of textures. Another benefit is that this interface allows the packing algorithm to be used for different kinds of data (image arbitrary data that uses a rectangle as a key) thereby allowing `PaunPacker.Core` to be used as a standalone library for different purposes than is texture packing.

The second reason is related to code reusability and code duplicity. For example, almost all packing algorithms sort the input rectangles in some way, therefore when some there is some algorithm (call it *A*) that utilizes another algorithm as its subroutine (call it *B*) there is always a chance that both *A* and *B* sort the input (because *A* does not always know whether *B* sorts or not if *B* seems to *A* as a black-box. This situation has two bad consequences, the first is, that sorting the sequence twice wastes performance and the second is, that sometimes either *A* or *B* rely heavily on a certain order of input rectangle and if that is the case, if the other algorithm breaks this order by sorting it according to a different rules, the algorithm could break.

The third reason is the fact that some algorithms work without specifying the dimension of the bounding rectangle (i.e., they are used to find or better say approximate the minimum bounding rectangle) but some do not. Moreover, the algorithms that require to know dimensions of the bounding rectangle in advance sometimes simply do not work well when $\infty \times \infty$ hack is used.

These three reasons had led to the decision to find a different representation of packing algorithms. The nice approach was proposed by Korf [19] and it will be described now. Korf's approach is based on the idea to split the whole packing problem into two sub-problems: *containment problem* and *minimum bounding box⁴ problem*, where the former sub-problem tries to place the input rectangles into a given bounding rectangle while the later sub-problem finds a minimum bounding rectangle. The algorithm for solving the minimum bounding box problem traverses (in order of increasing area) through a set of possible bounding

⁴Korf's *minimum bounding box* has the same meaning as the term *minimum bounding rectangle* that is used in this thesis, so they will be used interchangeably.

boxes and for a given bounding box solves the containment problem on the input rectangles. Once a bounding box that could contain all the input rectangles is found, the minimum bounding box algorithm stops and yields the current bounding box as a result. The containment problem tries to place all the input rectangle into a given bounding box. In the same paper, Korf also proposed a method to decrease the number of bounding boxes that have to be traversed when minimum bounding box is searched. There is a quadratic amount of possible bounding boxes, but Korf's proposed method examines the only linear amount of bounding boxes. This method starts with a bounding box having certain dimensions and uses a containment algorithm if the containment succeeds the bounding box is remembered as the best so far and then its dimensions are reduced, otherwise, if it fails, the bounding box is expended. The actual details of this algorithm can be found in Korf's paper [19].

It was decided to use a similar approach to the one proposed by Korf, but with some small adjustments. These adjustments are an introduction of a third step—sorting step—and also the modification of the containment problem. The reason for adding a sorting step is the fact the order of rectangles as was illustrated in Figure 2.11 at the end of Section 2.2.1 and also the "third reason" above, because when the sorting step is separated, it can be reused. The original containment problem simply checks whether the bounding rectangle with given dimensions could contain the given input rectangles and returns either `true` or `false`. But notice that the containment problem actually try to pack the input rectangles inside the bounding rectangle with given dimensions so for a problem which is considered in this thesis, it is better to return either the packing result or an empty (`null`) result instead of `true/false`. This change in behavior of the containment problem removes the necessity to call the packing algorithm (doing the same thing as the containment algorithm but yielding a packing result instead of boolean) right after the containment problem itself. This change also suggests changing the minimum bounding box algorithm to return the whole packing result instead of simply returning the dimensions of the minimum bounding box.

So finally, it was decided to decompose the packing problem into three parts:

1. *Search for a minimum bounding rectangles containing the input rectangles.*
2. *Placement of the rectangles* (the aforementioned modification of containment problem).
3. *Sorting of the input rectangles.*

The first part is parameterized by the second part which is then parameterized by the third part. The relation between these three steps is depicted in Figure 3.10.

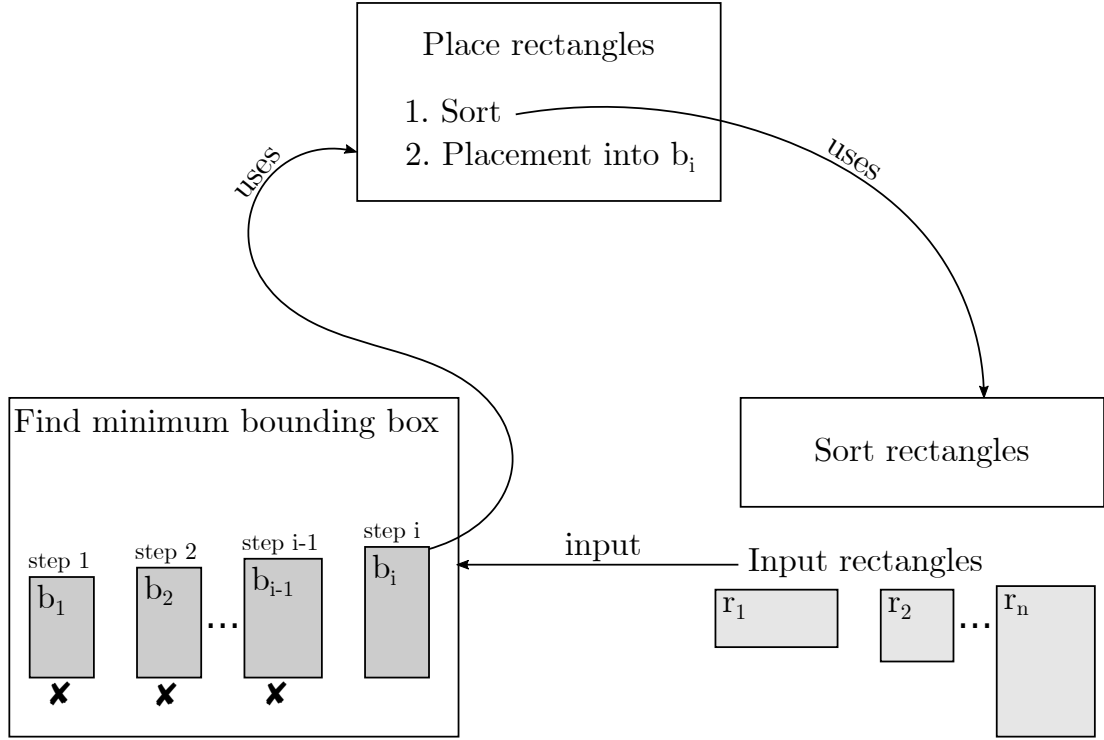


Figure 3.10: Decomposition of rectangle packing into three parts.

Figure 3.10 shows a decomposition of the packing process into three parts. The first part is to search for a minimum bounding box for the rectangles that are given on the input. This search works traversing through bounding boxes b_1, \dots, b_i . At i -th step, the input rectangles are tried to be placed inside the bounding box b_i by using placement algorithm as a sub-procedure. If the placement fails the bounding box b_i is discarded and the search continues with bounding box b_{i+1} , otherwise, if it succeeds, b_i is considered as a minimum bounding box and the result (b_i containing all the input rectangles) is returned. In this Figure, none of the bounding boxes b_1, \dots, b_{i-1} could contain the input rectangles. Notice that the placement part works in two steps, first, it sorts the input rectangles using and then does the placement itself.

Modular components

The modular⁵ components are the components that could be loaded from plugins, i.e., the components that should be extensible. In order to satisfy the requirement **R3** (that the PaunPacker could be extended by new packing algorithms, image processors and metadata writers) it seems natural to say that all these components will be modular. However, in the case of packing algorithms, some changes are needed. Because it was decided to decompose the whole packing process into three parts, it is proficient to make all these parts modular. That is, to allow users to also create their own sorters, placement algorithms, and minimum bounding box finders. To summarize it, the modular components are the following:

- Image processors

⁵It should be mentioned that the modular components are sometimes also called the extensible components.

- Metadata writers
- Minimum bounding box finders
- Placement algorithms
- Sorters

Consider a plugin developer that develops a minimum bounding box finder (*MBBF*). Because *MBBF* is parameterized by a placement algorithm (as was already discussed above, in Section 3.6) it makes sense to allow the user to select a placement algorithm for this *MBBF*. However this is different from the situation where the user has to provide some parameter (integer, for example) that was also discussed above in Section 3.5 because the placement algorithm is not something that the user could input (via keyboard/mouse) to the `PaunPacker`, it is something that is either contained in `PaunPacker.Core` (for the case of algorithms that are included in the basic toolset for plugin development) or something that is loaded from another plugin and the user only specifies which should be used (for example, via selecting the packing algorithm in some combo-box control). This brings a new challenge—to allow a user to parameterize *MBBF* from `PluginA` by a placement algorithm from different `PluginB` (similarly for the parameterization of placement algorithm by the sorter).

This challenge could be solved by using an IoC container. The idea is the following: users select a placement algorithm that should be used, then this algorithm is registered into IoC container as a placement algorithm and when a *MBBF* is created, the selected placement algorithm could be resolved from the IoC container. All this could happen inside `PaunPacker.GUI` without any intervention from a plugin so it seems like a right approach. The `PaunPacker.GUI` could simply store the IoC container, register the types in accordance with the algorithms that the user has selected and then do the resolution using the IoC container.

3.7 PaunPacker.GUI revisited

Now after the meaning and representation of plugins was described, it is time to design GUI architecture in greater detail. To summarize the previous subsections, the current state is the following: there are plugins that could be rendered into regions inside `PaunPacker.GUI`. The question to be answered is the following: Who should decide, into which region inside the GUI of the host applications should the plugins be rendered?

As it was already mentioned in Section 3.5 the plugins should be rendered into the region, but because `PaunPacker` contains several types of plugins, it is advisory to have more such regions. These regions are simply `ContentControls` inside the view, therefore, it is possible to create a single region in every view, or even multiple regions inside a single view.

The idea is to create a single region for each of the modular parts, that is for the following parts: image processors, image sorters, minimum bounding box finders, placement algorithms where plugins of an appropriate type should be

registered. Prism has a concept of so-called **RegionManager** that allows registering views inside a specific region (keyed by region name i.e. by simple string). Therefore there are basically three options who could decide where the view of a given plugin should be rendered, these options are the following:

1. The host application should decide where to place a view of a given plugin, i.e., the plugin developer should not be able to say to which region the view of the plugin should be rendered. But instead, the views should be assigned to the regions based on the types of the plugin.
2. The plugin developer should decide by decorating the plugin class with a custom attribute—call it **RegionAttribute**—that would take a string with a name of the region where the plugin’s view should be placed.
3. The plugin developer should decide by using the instance of the region manager that is passed by to the plugin by the host application. The region manager allows the plugin developer to register the view inside the arbitrary region.

The approach that chosen in the initial phase of development of PaunPacker was the option **3**)—to give the plugin developer an instance of plugin manager and allow the plugin’s view to be registered anywhere within the host application’s GUI. At that time, there was a certain thought process that led to the choice of this option. This flow of thoughts is described in the next paragraph.

Initial reasoning behind choosing the option 3 The first option certainly gives the plugin developer least flexibility of all three options and it also adds more logic to PaunPacker itself, because the type of each loaded plugin has to be checked and assigned to region appropriately. Therefore it does not suit the needs of PaunPacker very well. The second option is slightly better because it allows the plugin developer to specify an arbitrary region name, but it still adds more logic to PaunPacker. The last option offers the same flexibility (or even more, because the developer could export whatever and wherever e.g. multiple instances of views inside different regions), moreover, it does not add any additional logic to PaunPacker. the only possible concern is security, is it safe to let the plugin developer do possibly anything with the region manager? And the answer is yes because of the two following reasons:

- The person who installs a plugin does it on own risk because there is always some possibility (unavoidable) that a plugin contains malicious code (this was already discussed above in this section).
- The actual ownership of reference to **RegionManager** could only mean that the plugin developer could (possibly) accidentally mess up the whole GUI but that is something that the user would have inevitably recognized and could solve by simply removing the broken plugin.

However, with the passage of time, these ideas turned out to be wrong and the selected option actually turned out to be completely wrong, the updated reasoning follows. Although the third option provides most of the flexibility and it does not add any logic to PaunPacker, this optioned turned out to be inferior to

some of the other options. One of the reasons why it is worse is the fact, that the plugins should be unaware of the GUI. The plugin's unawareness is a very strong benefit because it allows updating (e.g. change regions, or delete some regions, etc.) the GUI of the host application or even to port the host application to a different platform (with different GUI framework) without breaking the existing plugins. It was already mentioned in Section 3.5 that in the case of porting `PaunPacker.GUI` to another platform, the Plugin's view would simply not get loaded by `PaunPacker.GUI` but that actually does not work, when the third option is chosen. Because the third option mandates the `PaunPacker.GUI` to give an instance of region manager to the plugin, the plugin has control over the GUI and if it does not get an instance of region manager (which is the case when `PaunPacker.GUI` would be ported) the plugin does not get loaded to host application at all. The second option has the same disadvantage. Therefore it seems best to use the first option and this decision is also supported by the fact that it is better to add just a little bit of functionality once to the `PaunPacker` than to force all the plugin developers to add extra code for region registration and thus making the plugin development more difficult and cumbersome. It was therefore decided to abandon the first decision to choose the third option and instead choose the **option 1**.

4. Developer Documentation

The whole implementation part of this thesis is contained within a single Visual Studio 2019 solution called **PaunPacker**. This solution consists of a total of 20 projects, all of them containing C# code and targeting either the .NET Core 3 or .NET Standard 2. The structure of **PaunPacker** solution is shown in Figure 4.1.

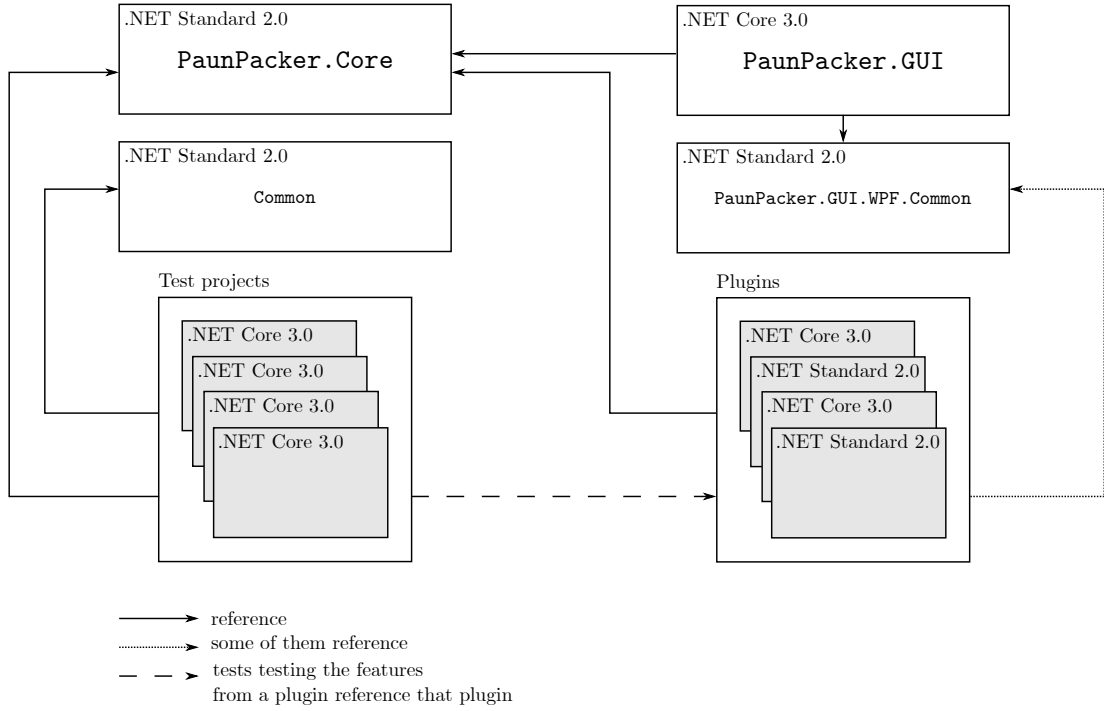


Figure 4.1: Structure of **PaunPacker** solution.

Figure 4.1 shows a structure of **PaunPacker** solution together with references (arrows) between these projects. The entire solution could be "virtually" divided into four parts—**PaunPacker.Core**, **PaunPacker.GUI**, *Tests* and *Plugins*. Actually, there are two more parts (or better to say projects) which are not related to the **PaunPacker** itself, therefore, they will only be briefly described now and then omitted in other sections of this chapter. These projects are:

- **Setup** – is responsible for generating a Windows installer for the **PaunPacker** and it requires WiX Toolset [28].
- **Documentation** – is responsible for generating a documentation, this project requires Sandcastle Help File Builder [29].

PaunPacker.Core contains tool-set for plugin development, notably: interfaces and several implementations of packing related algorithms. **PaunPacker.GUI** contains the WPF packing application. *Tests* part consists of several other projects that are testing certain algorithms or other parts from **PaunPacker.Core** and they are placed inside *Tests* folder and within **PaunPacker.Tests** namespace. *Plugins* part consists of plugins that export functionality from **PaunPacker.Core**. All this functionality could be exported directly from **PaunPacker.Core** but it was

decided not doing so, and rather, separate the export into plugins. This separation brings two benefits: first, plugins could have a WPF view associated with them without breaking `PaunPacker.Core`'s compatibility with .NET Standard, and second, plugin loading logic could be reused and thus either decreasing the amount of additional logic needed inside either `PaunPacker.GUI` or eliminating the use of MEF attributes inside `PaunPacker.Core`. The detailed description of these parts will be given later in this chapter (sections: 4.3, 4.4, 4.6, 4.7).

All the projects are targeting .NET Standard 2 whenever possible, but sometimes it is not possible—for example, when the assembly depends on WPF—and in that case the target framework is .NET Core 3. The prerequisites for this solution are Visual Studio 2019 and .NET Core 3 preview 5 SDK.

4.1 NuGet package dependencies

The projects in PaunPacker solution depend on several NuGet packages. These packages are described in Table 4.1.

Actually, there are more NuGet package dependencies but the remaining ones are related to tests (for example, `Microsoft.NET.Test.Sdk.dll`) and therefore they will not be listed or described here.

Package Description
<p>SkiaSharp</p> <p>SkiaSharp.dll NuGet package contains the SkiaSharp library that was already mentioned in Section 3.3.</p>
<p>Unity</p> <p>Unity is the IoC container that is used in the PaunPacker (aparat from the unity container, this package also installs Unity.Abstractions.dll that contains some handy extension methods, particularly: RegisterFactory that is used extensively in the PaunPacker).</p>
<p>Microsoft.CodeAnalysis.FxCopAnalyzers</p> <p>This NuGet package contains Roslyn analyzers that perform static analysis of the code and report potential issues regarding performance, security, etc.</p>
<p>Prism.Core & Prism.Wpf</p> <p>Prism.Core.dll and Prism.Wpf.dll NuGet packages provide access to Prism's features (Prism.Wpf.dll is now finally .NET Core 3 compliant and it provides, for example, the RegionManager). There also exists a NuGet package Prism.Unity.dll which offers functionality for using Prism together with Unity container but this package is not used (although initially it was used) because it internally uses an old version of Unity that does not have RegisterFactory method. However, the decision to stop using the Prism.Unity.dll has led to certain workarounds that were needed to be done. These workarounds are described later, in Section 4.4.10. At the time of writing this thesis, the pre-release version of Prism.Wpf.dll NuGet package had to be used because the release version was not .NET Core compliant.</p>
<p>System.Composition</p> <p>System.Composition.dll contains a .NET Standard compliant version of the original MEF implementation (that was contained within System.ComponentModel.Composition.dll NuGet package).</p>
<p>MoreLinq</p> <p>MoreLinq.dll package provides some additional extension methods for LINQ. This package is used in packing algorithms mostly because of its MinBy method that allows to select an object from IEnumerable that has a minimum key where the key is selected by a delegate.</p>

Table 4.1: NuGet packages

4.2 PaunPacker's workflow revisited

A high-level overview of PaunPacker's workflow was already given in sub-section 3.2.2 (together with an illustration in Figure 3.3) and now it is time to describe the workflow in greater detail. The more detailed overview of PaunPacker's workflow is depicted in Figure 4.2.

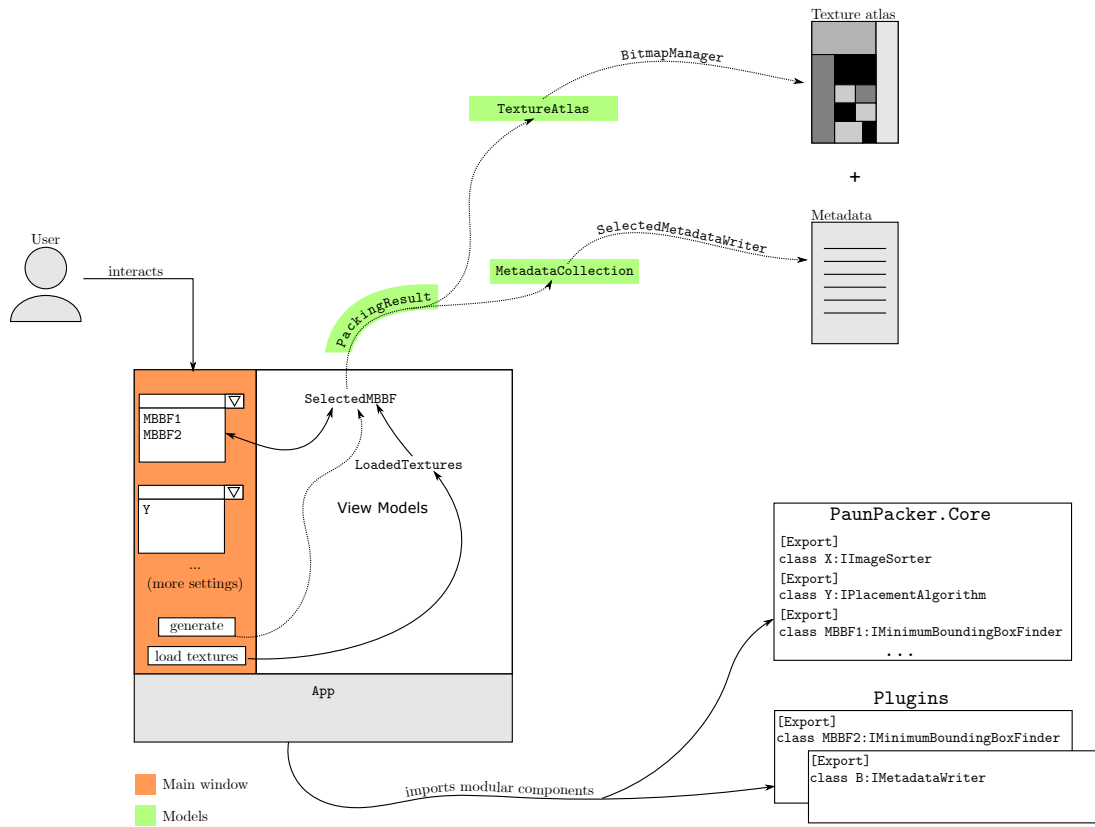


Figure 4.2: An overview of PaunPacker's workflow

As can be seen from Figure 4.2, there is a class called **App** that is responsible for importing all the modular components that are exported by the plugins and also the basic packing algorithm implementations from **PaunPacker.Core** assembly. These imported types are then rendered in the main window so that the user is able to select which algorithms to use.

When the user interacts with the PaunPacker's GUI (namely with the main window) these interactions are passed from the main window to the main window's view model (**MainWindowVM**) which then handles these interactions appropriately. The user's interaction typically starts by loading the input textures and the **MainWindowVM** handles this request by using **BitmapManager**.

When the user wants to generate texture atlas and clicks on the generate button, the **MainWindowVM** uses the selected minimum bounding box finder to pack the loaded textures which results in the instance of **PackingResult** class. The **PackingResult** class represents the result of the packing. It is a lightweight immutable class that only remembers rectangles inside the packing (including their positions) and the dimensions of the bounding box. The **PackingResult** is also used to create an instance of **TextureAtlas** class which represents a texture atlas. **TextureAtlas** has certain properties for retrieving information about tex-

ture atlas (width, height, rectangles contained within the texture atlas and the bitmap representing the texture atlas) and again, this class is immutable.

A relationship between `PackingResult`, `TextureAtlas`, `ImageMetadata` and `MetadataCollection` is depicted in Figure 4.3.

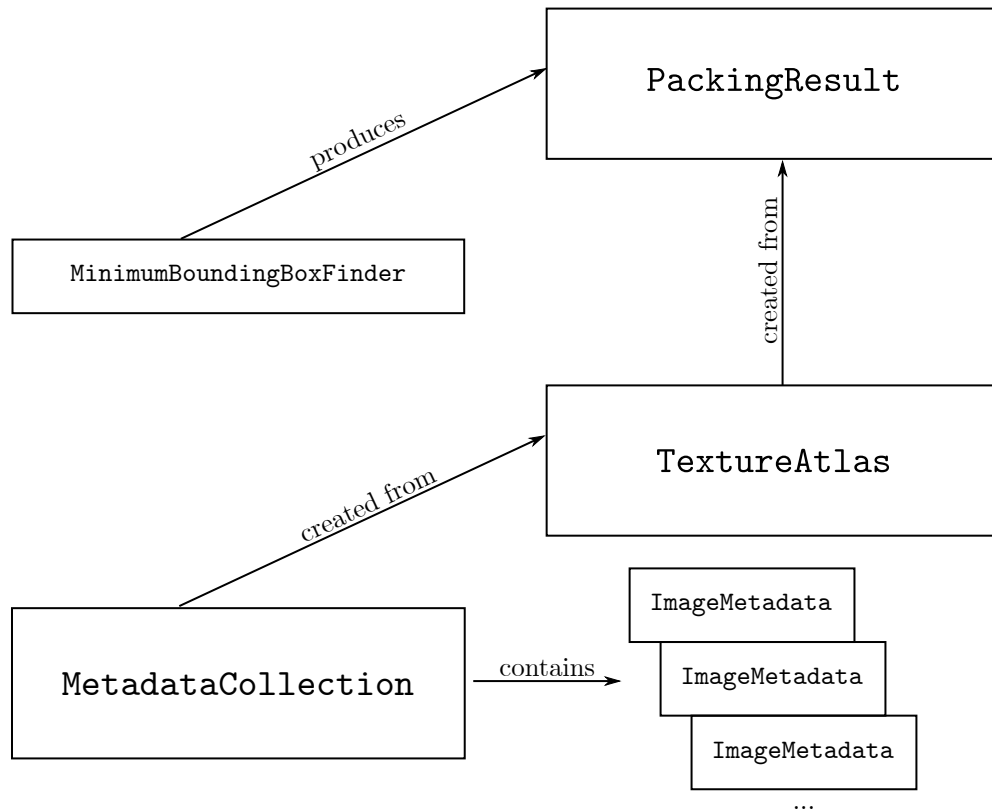


Figure 4.3: The relationship between `PackingResult`, `TextureAtlas`, `MetadataCollection` and `ImageMetadata`.

Once the `PackingResult` is created, a `MetadataCollection` is created from it and then the `MetadataCollection` is serialized to a selected file using the selected metadata writer. The `MetadataCollection` class represents the metadata of the texture atlas and it further consists of individual `ImageMetadata` (attribute, value pairs). The `MetadataCollection` can only be constructed by calling its constructor that takes `TextureAtlas` as its only parameter. The passed `TextureAtlas` is traversed and for each rectangle within the `TextureAtlas` instance of `ImageMetadata` is created and appended to the `MetadataCollection`.

The individual parts of this workflow are going to be described thoroughly in the rest of this chapter. However, it does not seem appropriate to describe every implementation detail, so instead, it was decided to describe the main concepts and core ideas that are used in the `PaunPacker`.

4.3 PaunPacker.Core

The `PaunPacker.Core` project contains interfaces that are representing individual, packing-related parts:

- Image sorters (`IImageSorter`)

- Placement algorithms (`IPlacementAlgorithm`)
- Minimum bounding box finders (`IMinimumBoundingBoxFinder`)
- Image processors (`IImageProcessor`)
- Metadata writers (`IMetadataWriter`)

together with additional packing-related types (most importantly `TextureAtlas` and `MetadataCollection`). Building this project yields a .NET Standard 2.0 dynamic library (.dll)

The `PaunPacker.Core` also contains `IProgressReporter` interface which is inherited by all the previously mentioned interfaces except for `IImageSorter` and `IImageProcessor`.

4.3.1 Representation of a rectangle

In Chapter 3 it was stated that it is better to perform the packing with rectangles instead of images and that the rectangle should somehow remember the image it corresponds to. For this reason, `PaunPacker.Core` also contains the `PPRect`¹ class that represents a rectangle and also holds a reference to the corresponding image. This reference could be `null` and in that case, it is simply a rectangle (this is useful when generating test cases). `PPRect` works as a wrapper around `SKRect` and the image reference is of type `PPImage`—that is another wrapper, around `SKBitmap`. `PaunPacker` internally uses types from `SkiaSharp` heavily, but there was an effort to eliminate the presence of these types in public API.

4.3.2 Metadata export

Metadata could be exported using any of the classes implementing `IMetadataWriter` interface. `IMetadataWriter` has single method called `WriteAsync` that takes path where the metadata will be written to, path of the corresponding texture atlas, `MetadataCollection` containing the metadata, and the last, optional parameter is a `CancellationToken`. The particular implementation of this class then outputs metadata in an appropriate format. The `WriteAsync` method returns `Task`.

4.3.3 Packing process representation

As it was already mentioned, the whole packing is decomposed into three parts operating on the input rectangles: sort, placement, minimum bounding box finding and these are respectively represented by the following:

- `IImageSorter`
- `IPlacementAlgorithm`
- `IMinimumBoundingBoxFinder`

The `IImageSorter` interface has a single method `SortImages` that takes input rectangles as `IEnumerable<PPRect>` and returns these rectangles in a particular order. The implementations of `IImageSorter` that are included within `PaunPacker.Core` are:

¹The prefix PP is an abbreviation for `PaunPacker`.

- `ByHeightAndWidthImageSorter`
- `ByHeightAndWidthImageSorterDesc`
- `PreserveOrderImageSorter`

The way how these implementations work should be pretty straightforward from their name, except for the `PreserveOrderImageSorter` that simply does not sort the input rectangles at all, i.e., it simply returns the input rectangles.

The `PreserveOrderImageSorter` is useful, for example, when developing a minimum bounding box finder and using some packing algorithm that requires `IImageSorter`, when the developer of the minimum bounding box finder wants to provide a certain order of input rectangles and give it to the placement algorithm, the developer could parameterize the placement algorithm by `PreserveOrderImageSorter` and sort the sequence using some different algorithm (potentially without dealing with `IImageSorter` at all) and then pass the sorted sequence to the placement algorithm.

The `IPlacementAlgorithm` interface has a single method accepting four arguments, where the first two arguments are width and height of the bounding box to which the placement will be done the third parameter are the input rectangles given as `IEnumerable<PPRect>` and the last, optional parameter is the `CancellationToken`. `PaunPacker.Core` contains several implementations of placement algorithms, namely:

- `BLAlgorithm`
- `SkylineAlgorithm`
- `GuillotineAlgorithm`
- `MaximalRectanglesAlgorithm`

In addition to `IImageSorter`, the `GuillotineAlgorithm` is parameterizable by the following interfaces:

- `IFreeRectangleExtractor`
- `IFreeRectangleSplitter`
- `IFreeRectangleMerger`
- `IRectOrientationSelector`
- `IFreeRectanglePostProcessor`

that are allowing to parameterize the individual steps of the algorithm (some of the parameterizable steps were already mentioned in Section 2.2.1). Due to the possibility of this parameterization, the `MaximalRectangles` algorithm is just `GuillotineAlgorithm` with a particular parameterization of the steps, specifically:

- `MaxRectsFreeRectangleSplitter`
- `MaxRectsFreeRectangleSortedMerger`
- `MaxRectsFreeRectanglePostProcessor`

The `IMinimumBoundingBoxFinder` interface has also a single method that accepts `IEnumerable<PPRect>` of input rectangles and a `CancellationToken`. The reason for accept `CancellationToken` is that it is expected for a minimum bounding box finder algorithms to run longer and therefore it could be useful to be able to interrupt them. The `PaunPacker.Core` contains three implementations of this interface, namely:

- **FixedSizePacker** – packs the rectangles into a bounding box with fixed dimensions.
- **PowerOfTwoSizePacker** – packs the rectangles into a bounding box with dimensions which are minimal and in powers of two.
- **UnknownSizePacker** – packs the rectangles into a bounding box with dimensions which are minimal.

All these implementations are parameterizable by the **IPlacementAlgorithm**.

Finally, there is a **GeneticMinimumBoundingBoxFinder** that implements the genetic packing algorithm and is also parameterizable by **IPlacementAlgorithm**.

It is advised that whenever implementing **IPlacementAlgorithm**, the implementing class should accept **IImageSorter** as one of its constructor parameters and that whenever implementing **IMinimumBoundingBoxFinder** the implementing class should accept **IPlacementAlgorithm** as one of its constructor parameters. This kind of parameterization is highly recommended (in order to provide maximum flexibility and code reusability) and therefore all the algorithms that are contained within **PaunPacker.Core** adhere to this recommendation. On the other side, this parameterization is not forced so it is allowed to create non-parameterizable minimum bounding box finders and placement algorithms (and sometimes it could make a good sense).

4.3.4 Representation of image processors

Image processors are represented by the **IImageProcessor** interface that has a single method accepting **SKBitmap** with an optional **CancellationToken** and returning new (modified) **SKBitmap**. **PaunPacker.Core** also includes three abstract base classes:

- **BackgroundRemoverBase**
- **ColorTypeChangerBase**
- **TrimmerBase**

that could be used when developing new implementations of these common image processors. The **PaunPacker.Core** itself includes an implementation of each of these base classes plus the following implementations:

- **Extruder**
- **CroppingTrimmer**
- **PaddingAdder**

These implementations does not have any corresponding base classes because they are either reusing existing implementations of previously mentioned base classes or it did not seem to make sense having base classes for them. For example, there are not many ways how the **Extruder** could replicate the border pixels. This is very different from the **BackgroundRemover** where there are several ways of removing background, ranging from very simple to very complex algorithms.

4.4 PaunPacker.GUI

Building this project yields a .NET Core 3 executable (.exe). This project contains the views, view models and models (models are the types from PaunPacker.Core e.g. TextureAtlas) and logic for loading the plugins.

4.4.1 Application's entry point

The entry point main method is contained within a class called `App` inside `App.cs`. The `main` method starts by showing a splash screen and then creating an instance of `App` and calling the `App`'s `Run` method. The `App` represents a whole application and it is a subclass of Prism's `PrismApplication`. The working steps of `App` are the following:

1. `CreateModuleCatalog`
2. `RegisterTypes`
3. `CreateShell`
4. `InitializeModules`

The `CreateModuleCatalog` traverses all the .dll files inside the specified folder (call it *plugins* folder) and attempts to load each one of them, then it enumerates all the types within the loaded assembly that are implementing `IModule` interface and that are not abstract. These types are then added to the so-called `ModuleCatalog`.

Next, because it was decided to also allow use of MEF (because it provides a very simple and concise way to export plugins) this method also creates a new instance of `ContainerConfiguration` adds the found assemblies to it and then creates the MEF container (`CompositionHost`) from this configuration. This allows importing all the plugins that are decorated by MEF's `Export` attribute.

The `RegisterTypes` serves for additional registration of types that will be later used by the application. The `App` class used this method to register an instance of `RegionManager`—which is responsible for region manipulation—and several dialogs.

The `CreateShell` should create the so called "shell". In the case of PaunPacker, the "shell" is simply an object of a type inheriting from the `Window`, specifically, an instance of `MainWindow`. The `CreateShell` method therefore creates `MainWindow` and most importantly, sets the `DataContext` property of this window to a new instance of `MainWindowViewModel` to wire the view (`MainWindow`) to its view model (`MainWindowViewModel`).

The `InitializeModules` is responsible for module initialization. This method starts with a call to base class' implementation of `InitializeModules` that instantiates the plugins—non-abstract classes implementing `IModule` interface—and then calls the two methods—`RegisterTypes`, `OnInitialized`—given by this interface on these plugins. The meaning of `RegisterTypes`, `OnInitialized` will be described later, in Section 6.1. After calling them—when the plugins have been initialized—the types exported from plugins are extracted and passed to the `MainWindowViewModel` so that the `MainWindow` could show them. An extraction of plugins is shown below, in Listing 5. These exported types comes from two sources, first, each plugin gets an instance of `IUnityContainer` that can

be used for registrations, and second, the MEF automatically imports the types decorated by the `Export` attribute.

A third potential source of types is the application itself, meaning that some types that are part of `PaunPacker.Core` and initially they were exported directly from the `PaunPacker.Core` using the MEF's `Export` attribute, but this turned out to be a wrong approach as it polluted the `PaunPacker.Core` with the `Export` attributes and added a dependency on the MEF. The second approach that was taken was to instantiate some types from `PaunPacker.Core` inside the `PaunPacker.Core` and add them to the rest of the loaded types. However, this approach also turned out to not to be the best. The last approach that was taken as a final solution was to create a plugin which would export the types, in other words, this approach is same as the previous one, but instead from exporting from `PaunPacker.GUI`, the export happens inside the standalone plugin. The last approach seems superior to the other two approaches because it does not pollute the `PaunPacker.Core` and it also allows to reuse the plugin loading mechanism. More details about exporting the extensible components will be given in Section 4.4.14.

```
1  //...
2  //All the placement algorithm instances
3  var placementAlgorithms = mefContainer.GetExports<IPlacementAlgorithm>()
4      .Concat(corePlacementAlgorithms);
5  //Get All exported types (from both mefContainer & unityContainer)
6  var placementAlgorithmTypes = placementAlgorithms
7      .Select(x => x.GetType())
8      .Concat(UnityContainer.Registrations.Select(x => x.RegisteredType)
9          .Where(x => typeof(IPlacementAlgorithm).IsAssignableFrom(x) &&
10              !x.IsAbstract &&
11              !x.IsInterface));
12  //Set the types to the view model of main window
13  mainWindowVM.PlacementAlgorithmTypes = placementAlgorithmTypes;
```

Listing 5: An example of extracting plugin's exported types.

The Listing 5 shows a type extraction of all the classes implementing the `IImageSorter` from the loaded plugins. First the instances of `IImageSorter` implementations from `PaunPacker.Core` are created, then more instances of `IImageSorter` are exported from MEF container. From these instances, their types are extracted and concatenated with types registered in Unity container. It is intentional that types and not instances are exported from the Unity container, because it was decided to allow register type factories in order to provide higher flexibility (so instance may vary at runtime). The similar registration routine is done for all the modular parts, that is, `IImageSorter`, `IPlacementAlgorithm`, `IMinimumBoundingBoxFinder`, `IMetadataWriter`, `IImageProcessor`.

After all the types from loaded plugins have been exported, the views for all exported `IPlacementAlgorithm` and `IMinimumBoundingBox` types are extracted, registered to the appropriate region (via `RegionManager`) and set to hidden (they will be turned to visible based on the user selection in GUI). The export of views and region registration is shown in Listing 6.

```
1  //...
2  //Get views for PlacementAlgorithms, register them and hide all of them
3  collection = regionManager.Regions[RegionNames.PlacementAlgorithmsRegion];
4  foreach (var placementAlgorithm in mainWindowVM.PlacementAlgorithmTypes)
5  {
6      var view = UnityContainer.Resolve<UserControl>(PluginViewWiring
7          .GetViewName(placementAlgorithm)
8      );
9      if (view != null)
10     {
11         collection.Add(view,
12             PluginViewWiring.GetViewName(placementAlgorithm));
13
14         collection.Deactivate(view);
15     }
16 }
```

Listing 6: An example of extracting plugin’s exported types.

The code in Listing 6 attempts to extract a view for each of the exported placement algorithm types, register this view into `PlacementAlgorithmsRegion` and set it to invisible.

At the end of the `InitializeModules` method, an event signaling the finished plugins loading and initialization is published using a Prism’s `IEventAggregator`. Finally, the `Initialize` method on the main window’s view model is called. The publishing of `ModulesLoadedEvent` is shown in Listing 7.

Listing 7 shows an example of using `IEventAggregator` to publish an event, in this case, the event is `ModulesLoadedEvent` and its payload is `ModulesLoaded-Payload` that contains all the types that were exported from loaded plugins. All the subscribers of this event will get notified that the plugins were loaded successfully, allowing, for example, to subscribe to this event inside some plugin and react appropriately (the reasons why this might be useful will be demonstrated later in Chapter 6).

```
1 //...
2 var eventAggregator = UnityContainer.Resolve<IEventAggregator>();
3 eventAggregator.GetEvent<ModulesLoadedEvent>()
4     .Publish(new ModulesLoadedPayload(
5         imageSorterTypes
6             .Concat(placementAlgorithmTypes)
7             .Concat(minimumBoundingBoxFinderTypes)
8             .Concat(metadataWriterTypes)
9             .Concat(imageProcessorTypes)
10    ));
11 mainWindowVM.Initialize();
```

Listing 7: An example of publishing an event.

4.4.2 Views

The views are located in `PaunPacker.GUI.Views` namespace and they are either `UserControls`, `Windows` or `Regions` (regions then contain `UserControl`). The most important view is `MainWindowView` that represents the whole main window of the application and it consists of several other views that are recursively composed of other views. The view whole structure of view composition inside `PaunPacker.GUI` is depicted in Figure 4.4.

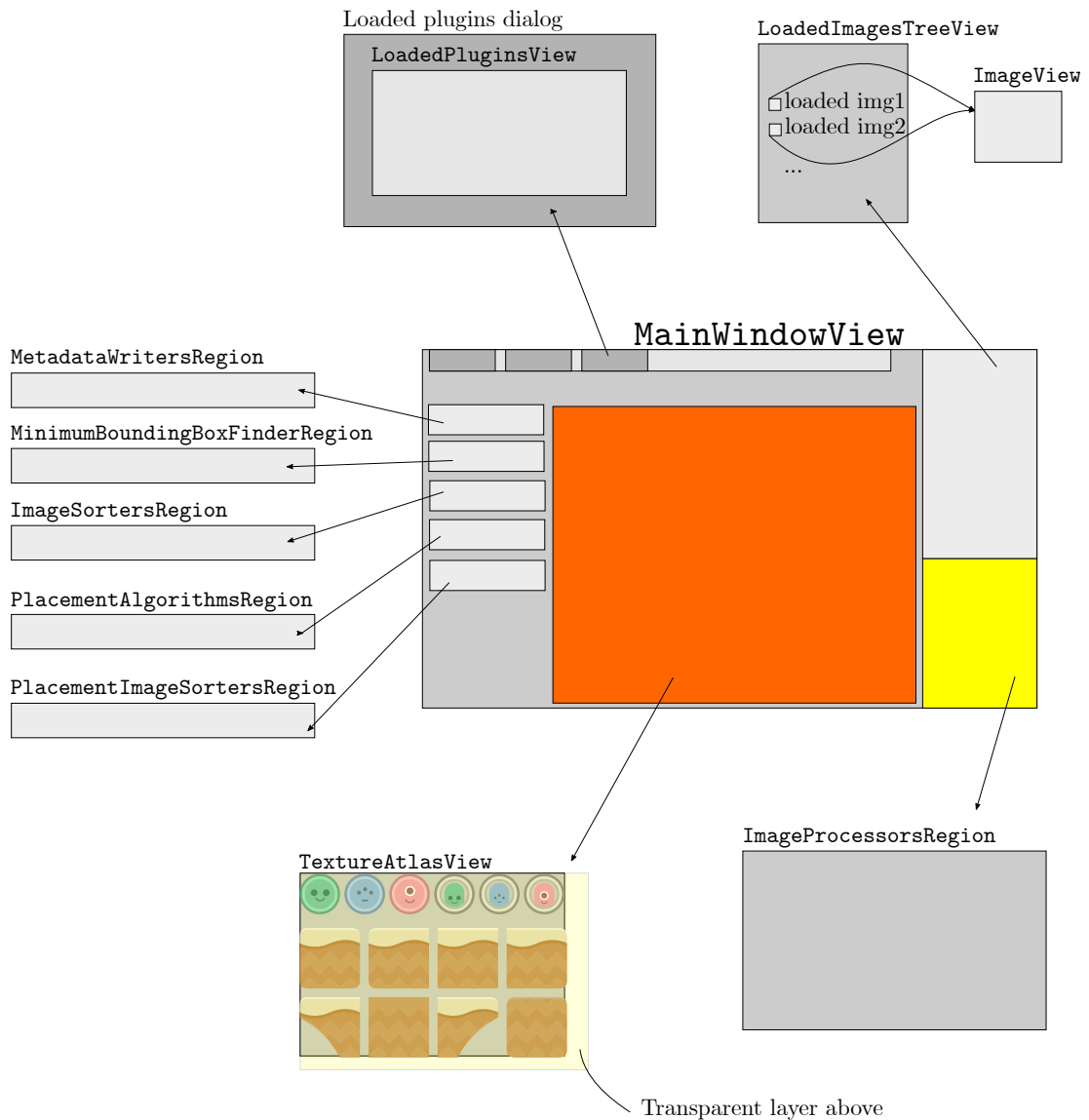


Figure 4.4: Illustration of view composition in `PaunPacker.GUI`

Figure 4.4 shows how `MainWindowView` is composed from several other views.

The orange rectangle contains a `TextureAtlasView` plus a thin, transparent layer above the `TextureAtlasView`. This layer is used to highlight certain parts (more specifically the rectangles that the user have clicked on) of the texture atlas.

The yellow rectangle contains a `ImageProcessorsRegion`, that is the place where all the loaded image processors that have a view associated with them are rendered.

The left part of this Figure contains five more regions, for the remaining extensible components. These regions are the following:

- `MetadataWritersRegion`
- `MinimumBoundingBoxFinderRegion`
- `ImageSortersRegion`
- `PlacementAlgorithmsRegion`
- `PlacementImageSortersRegion`

The regions from the list above are used for rendering the corresponding extensible components (again, only those that have a view associated with them).

On top of the main window there is a menu and this menu allows to show a dialog with loaded plugins. This dialog shows the plugins that were loaded by the `PaunPacker` together with some information about these plugins.

Lastly, on the top-right of the main window, there is a `LoadedImagesTreeView` that displays all the images that are loaded inside the `PaunPacker` (their thumbnail and file name are shown). This view is further composed of `ImageViews` that shows the thumbnail of the loaded image.

4.4.3 Data binding, `INotifyPropertyChanged` and `ObservableCollection`

In `PaunPacker.GUI`, the data binding is used to bind the properties of the view model to the attributes (in XAML) of the corresponding view. To get the data binding work properly, the view model should send a notification whenever the bound property has changed. This is achieved by implementing the `INotifyPropertyChanged` interface and raising the `NotifyPropertyChanged` event. This applies generally to all the view models so because it is so frequent, it was decided to create an abstract base class called `ViewModelBase` that inherits from the `INotifyPropertyChanged`. The `ViewModelBase` class is located in `PaunPacker.GUI.WPF.Common` project and it will be described in greater detail later, in Section 4.5. All the view models should implement this base class.

Sometimes the view binds an attribute to some collection instead of a single view model's property, for example, some views bind and `ItemSource` attribute. The problem is that when the bound collection changes in the sense that some item was added/removed from the collection, the view does not get notified about this change. To solve this issue and avoid calling `NotifyPropertyChanged` event at every place where some item is added or removed from the collection, the `ObservableCollection` is used. The `ObservableCollection` sends a notification whenever the collection itself was modified (item was added or removed). The `ObservableCollection` is used in, for example, `LoadedImagesTreeViewModel`.

4.4.4 Commands

In order to avoid the code in code-behind, `PaunPacker.GUI` uses commands instead of click events whenever possible. For example, instead of defining `OnClick` event on a `Button` control, a command could be created and then bound (thanks to the WPF's data binding capability) to the button. The command is represented by the `ICommand` interface and whenever the button is clicked, the com-

mand gets executed. Commands are created inside the view model thus eliminating the need to pollute view's code-behind. `PaunPacker.GUI` contains an implementation of `ICommand` interface called `RelayCommand` (more information about commands and `RelayCommand` can be found at MSDN website [30]).

4.4.5 Behaviors

In some situations, the event could not be handled by using commands, for example, the mouse wheel event cannot be handled with them. However, this situation could be solved by using a concept called `Behavior`. The `PaunPacker.GUI` contains two behaviors:

- `MouseClickBehavior`
- `MouseWheelBehavior`

The `MouseClickBehavior` is defined for a `UserControl` and it is used on the `TextureAtlasView`. It simply handles the mouse click event that occurs on the `TextureAtlasView` and processes the event by highlighting the rectangle (within the texture atlas) that was clicked on (if any). The `MouseWheelBehavior` is defined on the `MainWindowView` and it handles the mouse wheel event and processes it by zooming in/out the texture atlas.

Using the behaviors in addition to the commands has led to the consequence that the views have roughly zero code in their code-behinds (only the mandatory `InitializeComponent()` and the partial class declaration are present).

4.4.6 Converters

Converters are used to convert from one type (referenced from XAML) to another type that possibly has better visual interpretation.

For example, `PaunPacker.GUI` often uses the `NullToVisibilityConverter` that obtains a reference to an object and whenever this reference is `null` it returns the `Visibility.Collapsed` value. Otherwise, it returns `Visibility.Visible`. This converter is useful when some control references (by data binding) a member within its view model so that whenever this member is `null`, nothing is shown and when it is not `null`, the member should be shown (with a possible replacement by its view if an appropriate data template was defined). Converting the `null` to `Visibility.Collapsed` also has an effect that the hidden control will not take a space in the layout.

There are several other converters (`BooleanToVisibilityConverter`, `TypeToPluginInfoConverter`, etc.), but the underlying principle is the same so they are not going to be described.

4.4.7 Events

When developing with MVVM pattern, there often comes a need to perform communication between the individual view models and this was also the case for `PaunPacker.GUI`. It was decided to use events for the inter-ViewModel communication because the Prism offers an `EventAggregator` that allows doing this easily. There are two events defined in `PaunPacker.GUI`:

1. RectanglesSelectedEvent
2. UnloadImageEvent

The first event is published by the `TextureAtlasViewModel` when the user selects any of the rectangles inside the texture atlas. And this event is later processed by the `MainWindowViewModel` that processes the selected rectangles (images) and replaced the view model representing all the loaded images it holds with a new view model that contains the processed rectangles (images). The `UnloadImageEvent` is published by the `LoadedImagesTreeViewModel` (the user sees the images that are loaded and can unload them) and later processed by the `AllRectanglesViewModel`.

4.4.8 Services

The view models sometimes need to allow the user to select some files, folders that should be opened or select a path where the file should be saved. In order to do this and at the same time, to avoid polluting the view models with a platform-specific code (dialogs, windows) the services are used. The idea is to move this platform-specific code to the services. As an example, consider a situation where the user clicks on the "Load file(s)" button. This click would execute the associated command in the view model and the view model should handle it. What is important, is the fact that at the end of the user interaction, the view model should obtain the file path(s) of the files that the user has selected. And what happens in-between does not matter for the view model. So instead of showing the dialog from the view model, the view model simply uses the `OpenFileService` and obtains these file paths regardless of how the `OpenFileService` is implemented. This situation is illustrated in the Listing 8.

```
1 loadFileCommand = new RelayCommand((_) =>
2 {
3     var service = new Services.OpenFileService(
4         Common.FileFilters.IMAGE_EXTENSION_FILTER);
5     var files = service.GetFiles();
6     if (files != null)
7     {
8         // Do something with the files
9     }
10 }, (_) => true);
```

Listing 8: Illustration of `OpenFileService`.

The Listing 8 shows the handling of the event which is caused by the user clicking on the "Load file(s)" button. The event is handled from the `loadFileCommand` by using the `OpenFileService` that only allows to return file paths with a `IMAGE_EXTENSION_FILTER` extension.

Note that the current implementation of the `OpenFileService` simply shows the `OpenFileDialog` with an appropriate file extension filter and then return the paths of the files the user has selected.

4.4.9 Dialogs

Sometimes it is useful to show a simple dialog displaying some message—possibly a notification or an error—to the user. As it was already mentioned in the previous Section 4.4.8 it is undesired to pollute the view models with platform-specific dialogs or to use a view related parts (instantiating windows) directly from the view models.

Fortunately, the Prism recently come up with a `IDialogService` [31] that allows creating dialogs that could be designed in an arbitrary way. `PaunPacker.GUI` contains several dialogs, for example, the `MessageDialog` that allows showing a dialog with a given message. An example of using dialogs inside the `PaunPacker` is shown in Listing 9.

```
1 var parameters = new DialogParameters()
2 {
3     { MessageDialogParameterNames.Title, title},
4     { MessageDialogParameterNames.Message, message }
5 };
6 dialogService.ShowDialog(DialogNames.MessageDialog, parameters, (x) => { });
```

Listing 9: Illustration of `OpenFileService`.

The code in Listing 9 shows a notification dialog with a given message and title.

4.4.10 Workarounds

The `PaunPacker.GUI` contains two workarounds that were needed to be done in order to maintain the .NET Core compliance of the project. These workarounds are the `SkiaSharpExtensions` and `PrismApplication` that are both located in the `PaunPacker.GUI.Workarounds` namespace.

The `SkiaSharpExtensions` is a static class that contains several methods for converting the `SKBitmap` and `SKImage` to `WriteableBitmap` that could further be processed and displayed in the WPF controls. These extensions are WPF specific and they are included inside `SkiaSharp.Views` NuGet package together with other platform-specific classes. The problem with this NuGet package is that it is not .NET Core compliant. Because only these conversion methods were needed, it was decided to pick them from the `SkiaSharp`'s GitHub repository [32] instead of using the NuGet package and violating the .NET Core 3 compliance.

The `PrismApplication` was partially taken from Prism's GitHub repository [33] and normally it is acquired by installing the `Prism.Unity` NuGet package but this approach did not suit the `PaunPacker.GUI`'s needs. The reason for this workaround is the fact that the version of unity container used by Prism (`Prism.Unity`) is older than the one in `Unity` NuGet package and it does not contain `Unity.Abstraction` which contains useful extension methods, namely, `RegisterFactory`. Because of the lack of an ability to register a factory, it was decided to create a custom `PrismApplication` that will internally use the unity container taken from `Unity` NuGet package.

4.4.11 Resources

The `PaunPacker.GUI` project contains a directory called `Resources` that contains all the WPF's `.xaml` resources (resource dictionaries). The resources typically set `x:key` attribute for the converters and also creates a data template that ensures that whenever a `ViewModel` should be rendered, its view will get rendered instead. These two scenarios are shown in Listing 10.

```
<vc:NullToVisibilityConverter x:Key="NullToVisibilityConverter"/>

<DataTemplate DataType="{x:Type vm:TextureAtlasViewModel}">
    <vw:TextureAtlasView/>
</DataTemplate>
```

Listing 10: Converters and data templates inside resource dictionary

The Listing 10 shows a part of a resource dictionary. At the beginning of this resource dictionary, an `x:key` attribute is assigned to the `NullToVisibilityConverter` making it accessible by only specifying `NullToVisibilityConverter` within XAML (without the namespace). Then the `DataTemplate` is created that specifies that whenever a `TextureAtlasViewModel` should be rendered—for example, when it is set as a content of some control—the `TextureAtlasView` should be rendered instead.

There are few more resource directories, but they usually contain a similar kind of code. The only exception is the `MainWindowResources.xaml` that contains something more, namely some `Styles` and `ObjectDataProviders`. The `Styles` handle different states of the Progress bar that is displayed in the Main-Window when the texture atlas is being generated. When the texture atlas is being generated, the color of the progress bar is green and when the user cancels the generation, its color turns red, this and similar effects are handled by the `Styles`. The `ObjectDataProviders` are used to obtain all the possible values from certain `enums` (one of them is `SKColorType` for example). The extracted values are then displayed inside the `MainWindow`. It does not seem appropriate to describe these concepts in greater detail and this kind of brief overview should be sufficient for the developers.

4.4.12 ViewModels

Each view has its corresponding view model that is responsible for event handling, giving the data that should be displayed to the view and another logic behind the view. The data are transferred to the view using the data binding and the events are handled by using the commands and behaviors as described above.

4.4.13 Exported types instantiation

The user of `PaunPacker` is able to select the algorithms for packing, image processors, metadata writers and other extensible components that should be used for the texture atlas creation. Once the extensible components are loaded from

the plugins, they are passed to the `MainWindowViewModel` to properties that are bound to the `MainWindowView`. Therefore the user is able to select any of these extensible components (via the interaction with the GUI).

However, the controls in the `MainWindowView` does not contain the whole instances of these extensible components, but only their types²—imagine a combo box whose `ItemSource` is bound to the collection of `Types`. It is a responsibility of `MainWindowViewModel` to create an instance of the types selected by the user and then use them. Because these types may have dependencies (as it was already described in Section 4.3.3) it is not (at least not easily) possible to create these instances directly using the reflection. So instead, the Unity container is used to resolve the type that the user has selected as illustrated in Listing 11.

```
1 //In MainWindowViewModel's constructor
2 IoC.RegisterFactory<IMinimumBoundingBoxFinder>((_) =>
3 {
4     return childContainer.Resolve(MinimumBoundingBoxFinderVM.ExportedType)
5         as IMinimumBoundingBoxFinder;
6 });
7 //In MainWindowViewModel before generating the atlas
8 MinimumBoundingBoxFinder = IoC.Resolve<IMinimumBoundingBoxFinder>();
9 //Somewhere in a plugin
10 IoC.RegisterFactory<MinimumBoundingBoxFinderImpl>((_) =>
11 {
12     return new MinimumBoundingBoxFinderImpl(param1, ...);
13 });
```

Listing 11: Resolving the user selected types

The Listing 11 shows a resolution of a `MinimumBoundingBoxFinder` that will eventually be used to create the texture atlas.

4.4.14 Export of built-in extensible components

All the implementations of extensible components that are provided by `PaunPacker` are implemented in the `PaunPacker.Core` and they were initially exported directly from there by using the MEF's `Export` attribute. The downside of that approach was adding a dependency on MEF to the `PaunPacker.Core` which was undesired because it seems like the wrong design.

The design of exporting built-in extensible components was later changed and currently, these implementations are exported from the following plugins:

- `DefaultImplemtationsProviderPlugin` exports almost all the types that are implemented in the `PaunPacker.Core` except for skyline algorithm, guillotine algorithm and fixed size minimum bounding box finder. This plugin

²These types are then converted by using `TypeToPluginInfoConverter` to a more readable and meaningful format

also contains views for image processors that need it, for example, the extruder has a view which allows the user to specify the number of pixels that should be added around the borders of the texture.

- **SkylineAlgorithmProviderPlugin** exports the skyline algorithm together with some other types used by the algorithm. This plugin also contains a view which allows the user to parameterize the skyline algorithm by selecting one of the available implementations of **ISkylineRectAndPointPicker** interface. Note that the implementations of **ISkylineRectAndPointPicker** itself could be exported from other plugins.
- **GuillotineAlgorithmProviderPlugin** works similarly to the above mentioned **SkylineAlgorithmProviderPlugin**. It exports the skyline algorithm and several types used to parameterize the guillotine algorithm. The plugin also exports the view which allows the user to select the concrete implementations that should be used to parameterize the guillotine algorithm.

The implementations itself are still located in the **PaunPacker.Core**, but these plugins were created in order to load them and then export them. This has a nice consequence that (almost) all the algorithms are exported from plugins and therefore there is no need for some exceptions and special handling of certain types. The following Listing 12 illustrates by the example of skyline algorithm, how the types implemented inside the **PaunPacker.Core** are exported from a plugin.

```
1 public void OnInitialized(IContainerProvider containerProvider)
2 {
3     unityContainer = containerProvider.Resolve<IUnityContainer>();
4
5     //Register known RectAndPointPickers
6     unityContainer.RegisterType<LightweightRectAndPointPicker>();
7     unityContainer.RegisterType<MinimalAreaWasteRectAndPointPicker>();
8
9     //Register the Skyline algorithm
10    unityContainer.RegisterType<SkylineAlgorithm>();
11
12    //Register the OnModulesLoaded event
13    var eventAggregator = unityContainer
14        .Resolve<IEventAggregator>();
15    eventAggregator.GetEvent<ModulesLoadedEvent>()
16        .Subscribe(OnModulesLoaded);
17    //...
18 }
```

Listing 12: Exporting the types from SkylineAlgorithmProviderPlugin

The Listing 12 shows a part of plugin entry-point's `OnInitialized` method that exports the skyline algorithm and its dependencies. The registered event handler `OnModulesLoaded` first exports the view and then loads all the `ISkylineRectAndPointPickers` that were loaded from other plugins. This can be seen in Listing 13.

```

1 private void OnModulesLoaded (ModulesLoadedPayload payload)
2 {
3     var view = new SkylineAlgorithmView()
4     {
5         DataContext = new SkylineAlgorithmViewModel(payload
6             .GetLoadedTypes<ISkylineRectAndPointPicker>()
7             .Concat(unityContainer.Registrations
8                 .Select(x => x.RegisteredType)
9                 .Where(x =>
10                     typeof(ISkylineRectAndPointPicker).IsAssignableFrom(x) &&
11                     !x.IsAbstract &&
12                     !x.IsInterface))
13             .Distinct(), unityContainer));
14
15     unityContainer.RegisterInstance<UserControl>(
16         PluginViewWiring.GetViewName(typeof(SkylineAlgorithm)), view);
17 }

```

Listing 13: Exporting the view for Skyline algorithm

The code in Listing 13 creates a view and its view model which gets all the available types that are implementing the `ISkylineRectAndPointPicker`.

Lastly, the view model registers the implementation of the `ISkylineRectAndPointPicker` that is currently selected by the user (via the view) into the container as the `ISkylineRectAndPointPicker` and that implementation will be used when resolving the skyline algorithm from the container. For completeness, the part of code responsible for the aforementioned export is shown in the Listing 14.

The meaning of `OnInitialize` method and other constructs used in the Listings 12 and 13 will be clarified later, in Chapter 6. Similar code is also present at other places, for example, in the `GuillotineAlgorithmProviderPlugin`, but they will not be presented in this thesis. The reason for presenting this piece of code was to provide an example—that could be useful for plugin developers—of how it is done in the `PaunPacker`.

It should be mentioned, that all the previously mentioned types could be exported from the `DefaultImplementationsProviderPlugin` and the reason why it was split, was an effort to show how flexible the plugin loading is and to provide useful examples of dealing with plugins for future plugin developers.

The reason why the fixed minimum bounding box finder is exported from `PaunPacker.GUI` is its dependence on text boxes from the main window, that are describing the dimensions (the fixed size) of the bounding box. But again, it

```

1 public SkylineAlgorithmViewModel(
2     IEnumerable<Type> loadedISkylineRectAndPointPickerTypes,
3     IUnityContainer unityContainer
4 )
5 {
6     SkylineRectAndPointPickerVMs = loadedISkylineRectAndPointPickerTypes
7         .Select(x => new RectAndPointPickerViewModel(x));
8
9     unityContainer.RegisterFactory<ISkylineRectAndPointPicker>((_) =>
10     {
11         try
12         {
13             var picker = unityContainer
14                 .Resolve(SelectedRectAndPointPickerVM.RectAndPointPickerType)
15                 as ISkylineRectAndPointPicker;
16             picker ??= new MinimalAreaWasteRectAndPointPicker();
17             return picker;
18         }
19         catch (ResolutionFailedException)
20         {
21             //Return default implementation
22             return new MinimalAreaWasteRectAndPointPicker();
23         }
24     });
25 }

```

Listing 14: Registers the selected implementation of the `ISkylineRectAndPointPicker`

could have been implemented in a separate plugin with an additional view.

4.5 PaunPacker.GUI.WPF.Common

This project contains functionality that can be useful when creating WPF views. Because the plugin developers may also want to create WPF views, it was decided to take this functionality outside of the `PaunPacker.GUI`. It could be tempting to say that the functionality could be moved to the `PaunPacker.Core` but that would only pollute the `PaunPacker.GUI` with platform-specific code and more importantly, it would break its .NET Standard compliance.

This functionality includes the following types:

- `ModulesLoadedEvent`
- `ViewModelBase`
- `PluginViewWiring`

The `ModulesLoadedEvent` is published when all the modules have been loaded by the `App` and their initialization methods were called. This event could be

subscribed by the plugins for the following reason: consider a plugin developer who develops a plugin with some placement algorithm `PlacementAlgorithm` then sometimes it might be useful to parameterize this `PlacementAlgorithm` by some parameter type and allow other plugin developers to create the implementations of this parameter.

To be even more specific, consider the `SkylineAlgorithm`. It was already mentioned that this class is parametrized by the `ISkylineRectAndPointPicker`. What could be done to achieve this is to create a plugin that will contain the GUI with a combo box containing the names of all the `ISkylineRectAndPointPicker`. The plugin will register the `ModulesLoaded` event and when all the modules are loaded, the handler will get called. The handler then could traverse the Unity container and find all the types implementing `ISkylineRectAndPointPicker` that other plugins have registered (using the same unity container) and assign them to the collection that is bound to the combo box. Without the `ModulesLoadedEvent` this could not be done because when the initialization of the module is called, some of the plugins exporting the types implementing `ISkylineRectAndPointPicker` could have not been loaded.

The `ViewModelBase` serves as a base class for all the view modes, it simply inherits the `INotifyPropertyChanged` that the view modules should implement.

The `PluginViewWiring` is used to uniquely identify the view corresponding to a given type that represents one of the extensible components. Each such a class could have one view associated with it and it is given that the view should be registered under a fixed name to the unity container so that given an extensible type, the view could be resolved from the unity container. This allows to register several views as a `UserControl` but each under the different key. Later, when the plugins are loaded and the extensible types are imported from these plugins, for each of the imported types its view is tried to be resolved.

Attributes

The `PaunPacker.GUI.WPF.Common` project also contains attributes that could be used to leverage (in some way) how the main window treats the exported types. These attributes are the following:

- `TargetFramework`
- `ExportTypeMetadata`
- `PluginMetadata`
- `SelfContainedAttribute`
- `PartiallyContained`

An explanation of the meanings of the attributes from the listing above is going to be given in the following paragraphs.

Metadata exporters should be decorated by the `TargetFramework` attribute which indicated which game framework the decorated metadata exporter targets.

`AvailableTo` attribute is used to say that an exported type marked with this attribute is intended for the specified target framework. The target framework is given by the `FrameworkID` enum. The main window inside `PaunPacker.GUI` then allows the user to select only the types that target the game framework selected by the user and hides the rest.

`PluginMetadata` attribute allows the plugin developer to export metadata about the plugin. This metadata includes: plugin name, version, author, description and exported types—that is, all the types that are exported from the plugin. `ExportedTypesMetadata` is similar to `PluginMetadata` attribute, but it is intended to be used to export metadata (name, author, version, description) about individual types that are exported from a single plugin. As it will be justified later, in sub-section `subject:exportingMetadata`, these two attributes are strictly separated.

`PaunPacker` emphasizes the parameterization of the packing process, but sometimes this parameterization is undesirable, for example, when plugin developer develops a plugin containing an implementation of one of the extensible components such that, on the one hand, allows parameterization, but on the other hand, the parameters for this implementation are expected to be provided from within the plugin (for example using the Plugin's view). If that is the case, the user should not be able to provide other parameters using the Main window. For a concrete example, consider the implementation of minimum bounding box finder called `MbbfA` that has a single public constructor accepting `IPlacementAlgorithm`. With the previous approach to parameterization, the user would be allowed to select any of the available placement algorithms and the `MbbfA` would get constructed with this selected placement algorithm. But sometimes the plugin developer does not want to allow the user to select any placement algorithm and if that is the case, the plugin developer should mark the `MbbfA` with `SelfContained` attribute thus disallowing the user to specify a placement algorithm (the combo box for selecting the placement algorithm will be hidden).

Even more complicated scenario is the situation, when the exported type has a constructor with several parameters where some of the parameters (but not all) are, again, expected to be provided from within the plugin, but some other parameters are expected to be provided by the user from outside the plugin. As a specific example, consider a plugin developer developing a plugin called `GeneticMbbf` that takes 3 parameters: `IPlacementAlgorithm` population size and a number of generations and that also contains a view which allows the user to specify the last two parameters. If the user wishes to allow the user to specify the remaining (the `IPlacementAlgorithm`) parameter, then the `GeneticMbbf` should be decorated by the `PartiallyContained` attribute thus saying that some of the type's constructor parameters should be provided from outside of the plugin (for example, from the main window). More detailed explanation of `SelfContained` attribute and `PartiallyContained` attribute, including few examples, will be given in Chapter 6.

These attributes were initially present in `PaunPacker.Core` because they do not contain any platform-specific code, but later, it was decided to move them inside the `PaunPacker.GUI.WPF.Common` because their meaning is related to the GUI (although not specific for WPF) and therefore it should not be available inside `PaunPacker.Core` that should contain exclusive functionality related to the packing.

4.6 Tests

The whole PaunPacker solution is accompanied by several test projects. These test project tries to test basic correctness—for example, test situations in which there is not feasible packing result, verify certain properties of the produced packing result, etc.—of the placement algorithms and minimum bounding box finders. Some tests are testing individual parts of the algorithms, for example, there are tests for individual, parameterizable components of the guillotine algorithm. The test projects that are included in PaunPacker solution are depicted in Figure 4.5.

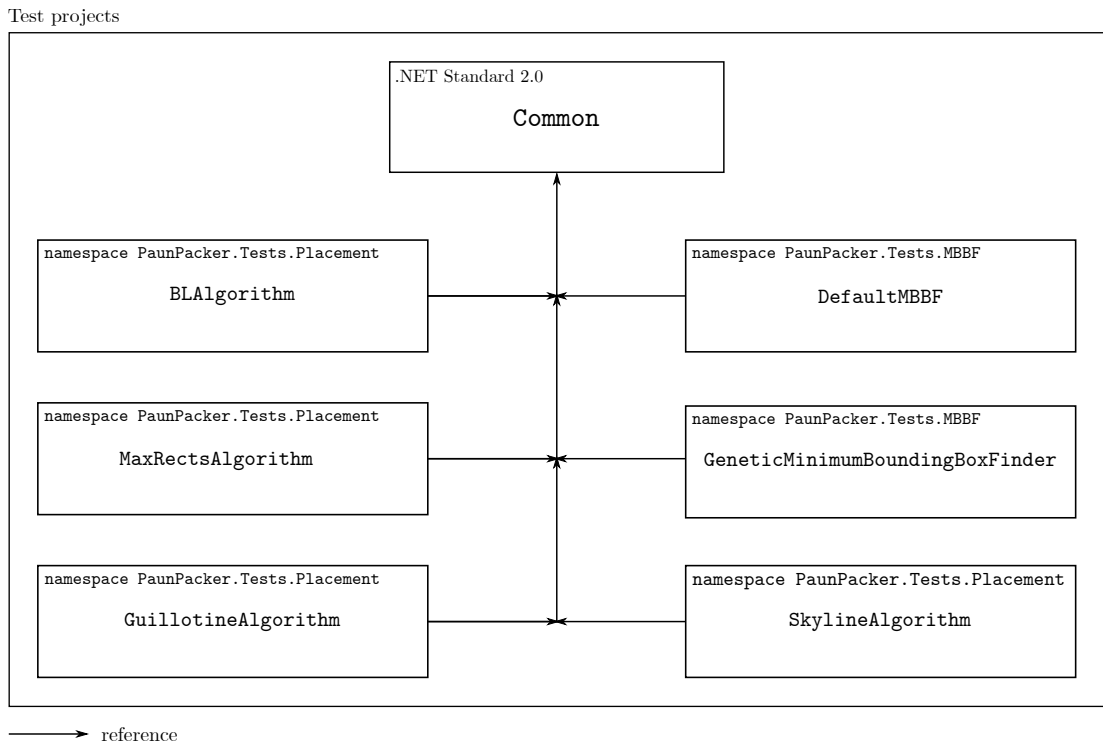


Figure 4.5: Tests projects in PaunPacker

The **Common** project contains functionality that is common to all the tests testing the packing related algorithms. This functionality is exposed by the only class inside this project, that is, the static class **TestUtil**. **TestUtil** contains a method for generating squares with dimensions of $1 \times 1, \dots, n \times n$ where the n is given as a parameter. Another method inside this class is the **shuffle** method that—as its name suggests—returns the shuffled (randomly permuted) version of the passed **IEnumerable**. Lastly, there is a **Succeed** method that takes four parameters: **actualWidth**, **actualHeight**, **expectedWidth**, **expectedHeight** and returns whether the dimensions are the same (including the case where the dimensions might just be swapped, this the whole rectangle might only be rotated but it also should be considered as an equivalent because the rotated rectangle has the same area (e.g. it does not matter if the result is 7×11 pixels or 11×7 pixels)).

4.6.1 Benchmarks

It might be useful to compare the performance of the individual implementations of the placement algorithm and minimum bounding box finders. For this reason, there are projects called **BenchmarkRunner** and **Support** which are located in the *Tests/Benchmarks/* directory.

The **BenchmarkRunner** contains a simple CLI application that allows the user to benchmark placement algorithms and minimum bounding box finders imported from the plugins. Note that not all the types exported from the plugins are available for benchmarks, this is simply because some types could not be instantiated easily without user interaction—mostly the types that are accompanied by a view which allows their parameterization—therefore only a simple types having a constructor that takes either no parameters at all or any combination of the **IImageSorter** and/or **IPlacementAlgorithm**.

The **BenchmarkRunner** uses a **BenchmarkDotNet.dll** NuGet package for the benchmarks but doing so required to tackle several problems. The first problem was that the benchmarked types are loaded dynamically from plugins. The **BenchmarkDotNet** allows to run benchmarks from a C# script given as a string but this feature is only available on the .NET Framework and not on the .NET Core, therefore, this feature cannot be used. Fortunately, another possibility is to run the benchmarks by specifying a **Type** of a class containing the benchmark and this feature is available on .NET Core.

The way in which the benchmark class could be obtained is to generate the C# script containing the class with the benchmarks at runtime, then compile it (using the Roslyn compiler), emit the generated CIL code into a memory stream and obtain the **Type** from it. The obtained **Type** is then passed to **BenchmarkDotNet** for an execution.

Another problem was that even though **BenchmarkDotNet** allows to add custom columns to the benchmark results it does not allow to track the values returned from the benchmark methods. This problem was solved quite suboptimally by saving the results of the methods into **ConcurrentDictionary** and later, when all the benchmarks are done, saving these results into the file. Once the execution of benchmarks is finished and the execution is yielded back, the file contents are read and added to a custom column. The saving to file is done once the benchmarks are done so it does not affect the measured time. On the other side, the addition of a single item into the **ConcurrentDictionary** is done within the benchmark method, but because all the benchmark method contains this addition, they should always be impacted in the more or less the same way. Therefore the time of all the benchmark method should only be shifted by a certain constant (which arises from adding to the dictionary).

To sum-it-up, the benchmark runner first shows the user some CLI which allows the user to configure the tests and then generates the tests, compiles them and let the **BenchmarkDotNet** run them. Once the **BenchmarkDotNet** has finished, the **BenchmarkRunner** reads the results of tests from the file and adds it to the custom column which is later displayed in the results report produced by the **BenchmarkDotNet**.

The **Support** contains class **PluginTypeLoader** which is responsible for loading types of placement algorithms and minimum bounding box finders exported from the plugins. The code in the **PluginTypeLoader** is very similar to the code

in `App` implementation the `PrismApplication` but slightly different.

A description of where the benchmarks are located and how they could be executed—together with an output example—are shown later, in Section 5.10 of the Chapter 5.

4.7 Plugins

Plugins are classes implementing the `IModule` interface which ensures that the Prism loads the plugins and call the initialization methods on them. The two initialization methods that are exposed by the `IModule` interface and that all the classes implementing this interface have to implement, are (in the order they will get called): `OnInitialized` and `RegisterTypes`.

The plugins are expected to implement the `IModule` interface and then export the types from the plugin by using either MEF or Unity container. The details about implementing the plugins and plugin's dependencies on NuGet packages and possible are described in greater detail in Chapter 6.

PaunPacker solution itself contains the following projects with plugins:

- `DefaultImplementationsPluginProvider` is used to export some of the default implementations from the `PaunPacker.Core` and the details about how this export is done was already described above, in Section 4.4.14.
- `UnityMetadataWriter` project contains a class with the same name which implements the `IMetadataWriter` interface. The `UnityMetadataWriter` class produces an output in an XML format compatible with the Unity game engine.
- `LibGDXMetadataWriter` project contains a class called `LibGDXMetadata` that is yet another implementation of `IMetadataWriter`. This class exports metadata in the `.atlas` format that were already described in the Section 1.1.1.
- `GeneticMinimumBoundingBoxFinder` project contains a genetic minimum bounding box finder called `GeneticMinimumBoundingBoxFinder` and its view that allows the user to specify the size of the population and the number of generations.
- `SkylineAlgorithmPluginProvider` is used to export the Skyline algorithm with types related to this algorithm and it was already described above in the Section 4.4.14. The Skyline algorithm itself is implemented as a class called `SkylineAlgorithm` inside the `PaunPacker.Core` project.
- `GuillotineAlgorithmPluginProvider` is used to export the Guillotine algorithm with its related types as was already described above, in Section 4.4.14. The algorithm itself is implemented in the `PaunPacker.Core` inside a class called `GuillotinePlacementAlgorithm`. This class is fully parameterizable so that it could be used to obtain different variations of the Guillotine algorithm. One such a variation is implemented inside a class called `GuillotineBestAreaFitAlgorithm` and another variation is inside the `MaximalRectanglesAlgorithm` class.

It should be mentioned here, at least briefly, how the plugins with views work. Suppose a class implementing the `IModule` interface. When the `OnInitialized` method is called, the plugin should create a view together with its view model, set the views data context to the view model and export the view via the `UnityContainer` as a `UserControl` under the key obtained by using the `PluginViewWiring` class. When all the plugins get loaded, the `App` traverses all the types that are exported from the plugins, and for the types that implement extensible components that are allowed to have GUI, the `App` tries to resolve the `UserControl` with the same key (the one obtained via `PluginViewWiring`), if it succeeds, the view is added to the appropriate region and is set to hidden (that is the default state). Later, when the user selects an extensible component that should be used, it is checked whether the region manager contains a view for the selected type and if it does, the view is shown (after hiding the currently shown view, if any). A more detailed explanation of creating plugins with views will be given in Chapter 4.

5. User Documentation

This chapter contains the user documentation for PaunPacker and it will be presented in a form of short tutorials illustrating and guiding the expected usage of the PaunPacker. The textures used in the illustrations throughout this section are taken from: OpenGameArt [1] where they are published under Public Domain (CC0) license.

5.1 Installation and running the PaunPacker

Before describing the installation of PaunPacker, it should be mentioned that the PaunPacker works well with Windows 10 version 1803 and above. Older versions of Windows were not tested but it is not impossible that PaunPacker would work on them too.

The PaunPacker solution contains a folder called *Setup* which contains a `.msi` installer for the PaunPacker called *Setup.msi*. In order to install the PaunPacker, the user should execute the *Setup.msi* installer and follow the installation steps.

Successful installation of the PaunPacker adds the PaunPacker into Program menu folder (under the Start Panel) and also places a shortcut onto the desktop. The PaunPacker could then be started using that shortcut. PaunPacker could be uninstalled from the control panel in a standard way.

Apart from the *Setup* directory, the PaunPacker also contains a directory called *Portable* which contains an executable allowing the run the PaunPacker without installing it.

Both Portable and Setup versions do not require any additional dependencies because they were created as self-contained executable. The user who has installed .NET Core preview 5 and above, wishing to save some space on the disk may want to install the PaunPacker from a *SetupMinimal.msi* located within the *SetupMinimal* directory. The minimal installation does not include `.dll` files that are part of a .NET Core and therefore requires much less of the disk space.

5.2 Plugin installation

Plugins could be installed by simply copying the `.dll` file containing the plugin inside the *plugins* folder. The location of the plugins folder is the same as the location of PaunPacker's executable (`PaunPacker.exe`).

5.3 Limitations

There are certain limitations related to the number of textures that could be used when using PaunPacker. Because the implemented algorithms work with 32-bit integers, the total area of the texture atlas should not exceed 2^{31} or rather 2^{30} pixels². Because the application is expected to be used in the context of 2D game development—as it was already mentioned in Chapter 1, this limitation should not be too constraining for the user because texture atlases are not usually larger

than 4096×4096 pixels due to the possible limitations of certain GPUs (this was also described in Chapter 1).

Even if the algorithm did not experience an arithmetic overflow, the algorithms are not optimized for very large outputs. Nevertheless, if the user must pack something large, it is best to use genetic minimum bounding box finder with a very low population and iteration count. Using the genetic algorithm with *population* = 2 and *iterations* = 1 it was tested to pack a 1000 square textures with dimensions 100×100 pixels. This sample data is available in the attachment inside the *samples* folder.

5.4 GUI overview

The PaunPacker consists of a single window—the Main Window—which the user is interacting with. The Main Window is shown in the Figure 5.1.

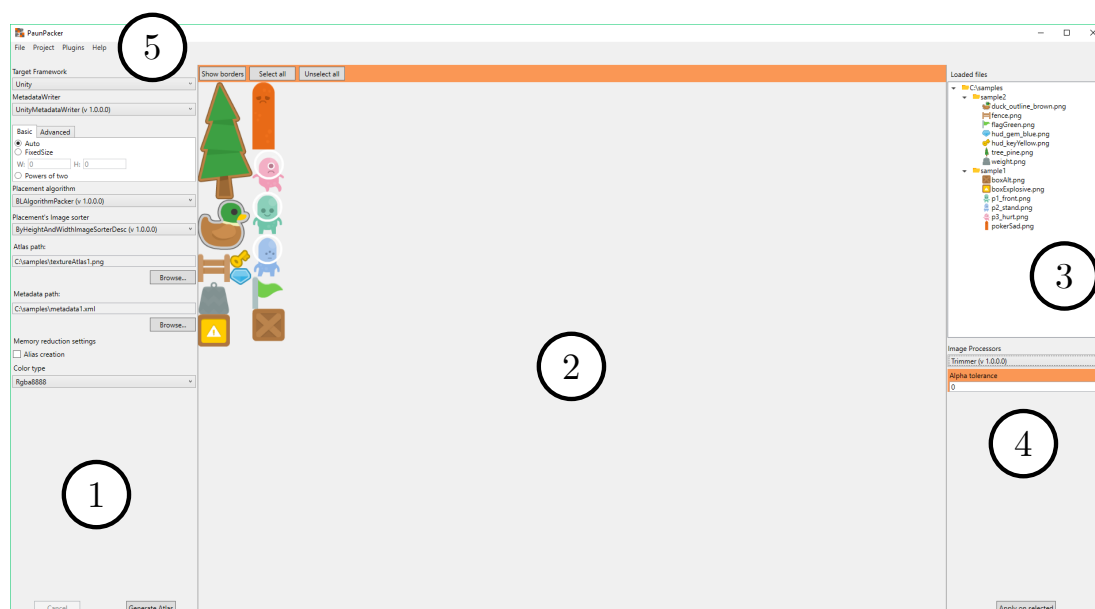


Figure 5.1: The PaunPacker’s main window.

The main window shown in the Figure 5.1 could be logically divided into five regions:

1. Shows the panel with settings related to the packing. These settings include a selection of target framework, metadata writer, placement algorithm, etc. The meaning of individual parts of this setting will be described later, in Section 5.6.
2. Is responsible for displaying the generated texture atlas and allows the user to select individual images of the texture atlas.
3. Lists all the images that are loaded by the PaunPacker.
4. Contains a list with the available image processors and displays a view of the selected one.

5. A strip menu which allows the user to load images, open projects, display loaded plugins, etc.

All of the previously mentioned parts of the Main Window will be described in the following sections.

5.5 PaunPacker menu

The PaunPacker's menu allows to perform the following tasks:

1. Load (image) file or folder
2. Open project
3. Create a new project
4. Close project
5. Save project
6. Save project As
7. Exit the application
8. Show installed plugins

1. In order to load an image, the user should select *File* → *Load file(s)* and then select the files that should be loaded. The same effect could be achieved using a shortcut key **Ctrl+F**.

2. Loading a folder means loading all the images that are contained (recursively) within the folder. In order to do that, the user should select *File* → *Load folder* (or using **Shift+F** shortcut) and then select the folder that should be loaded.

3. To create a new project, the user should either select *File* → *Create a new project* or use the **Ctrl+N** shortcut and then fill in the *path* and *project name* into the dialog that is just shown and click on *Create Project*. Once a project is created or opened two new options (5., 6.), namely: *Save project* and *Save project as* are shown in the *File* menu and the images tracked by the project file are loaded.

4. The currently opened project could be closed by selecting *File* → *Close project*. Closing the project does not unload the images but it only stops writing the changes (newly loaded images) into the project file.

5. Saving the project could be initiated using the **Ctrl+S** shortcut or via *File* → *Save project*. The project file of the currently opened project is overwritten by the new (up-to-date) version.

6. The *Save project as* command allows saving a currently opened project to a different project file. The *Save project as* could be done by using **Ctrl+Shift+S** shortcut or via *File* → *project as*.

7. The application could be closed either using the *File* → *Exit*, by using a shortcut **Alt+F4** or using any other standard Windows mechanism for closing an application.

8. When the user goes to *Plugins* → *Show installed plugins* a dialog showing all the installed plugins is shown. This dialog is depicted in Figure 5.2. The user could obtain more information about a particular plugin by selecting it and clicking on the *Show Details* button which shows another dialog—depicted in Figure 5.3—that contains details about the selected plugin. These details include:

- A name of the plugin
- A description of the plugin
- An author of the plugin
- A version of the plugin
- Name, description, author, and version about all the types exported from the plugin

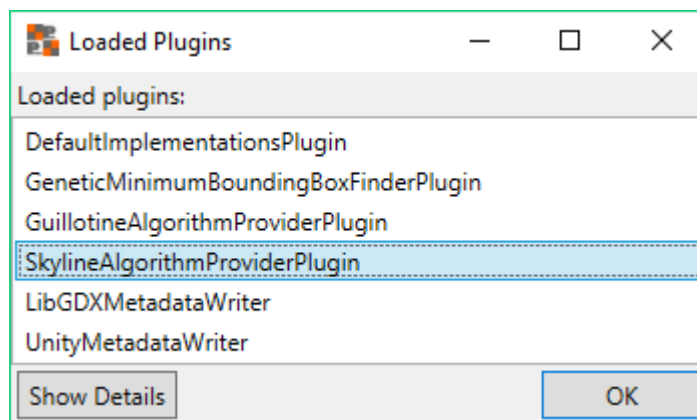


Figure 5.2: The dialog showing all the loaded plugins.

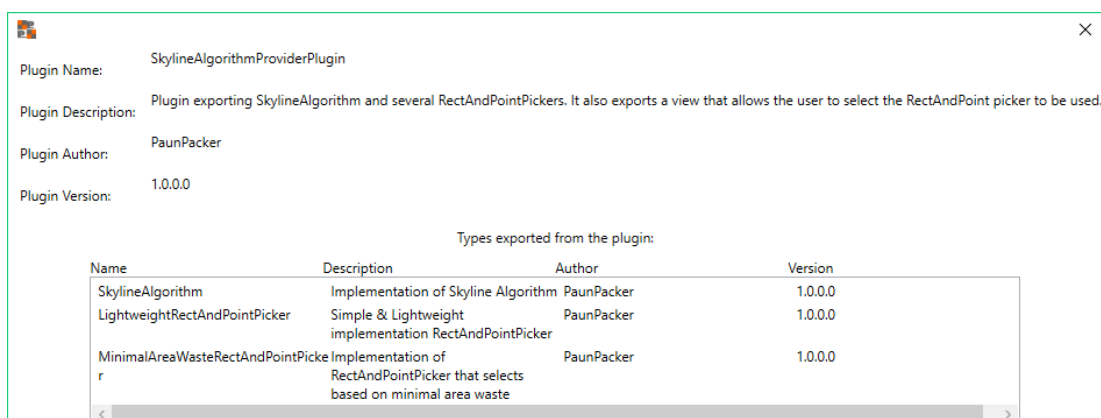


Figure 5.3: The dialog showing a details of the *SkylineAlgorithmProviderPlugin*.

The Figure 5.3 shows the details about the plugin which exports the Skyline algorithm. Notice that the plugin also exports two implementations of the *ISkylineRectAndPointPicker*. It should be mentioned that the exported view is not mentioned in the dialog because it does not seem to give the user any useful information.

5.6 Packing settings and texture atlas generation

All the possible settings are shown on the left of the Figure 5.1 and this section will contain their description, going from top to bottom.

At the very beginning, the user should select the target framework for which the generated texture atlas (and especially the metadata) will be used. Choosing the target framework has also impact on the features that will be available because different frameworks support different features that could be encoded in the metadata format they support.

Once the user has selected the target framework, the metadata writer should be selected. The choice of target framework typically leaves only a single option for this choice so it should be a fairly simple decision.

The next setting is called *Mode* of the packing and it could be either *Basic* or *Advanced*. The *Basic* setting contains only three minimum bounding box finders (the so-called default implementations):

1. UnknownSizePacker
2. FixedSizePacker
3. PoTSizePacker

that allow to pack into a bounding box with:

1. Minimal but initially unknown size (option *Auto*)
2. Fixed size (option *FixedSize*)
3. Minimal but initially unknown size that is in powers of two (option *Powers of two*)

Regardless of the choice that the user has made, the placement algorithm could be selected, because all these default implementations could be parameterized by it. Based on the selected placement algorithm, the user could be able to also select the image sorter that will be used by the selected placement algorithm. An example selection of these three parts is shown in Figure 5.4. The selection of minimum bounding box finder, placement algorithm and (possible) the image sorter will be used later for texture atlas generation.

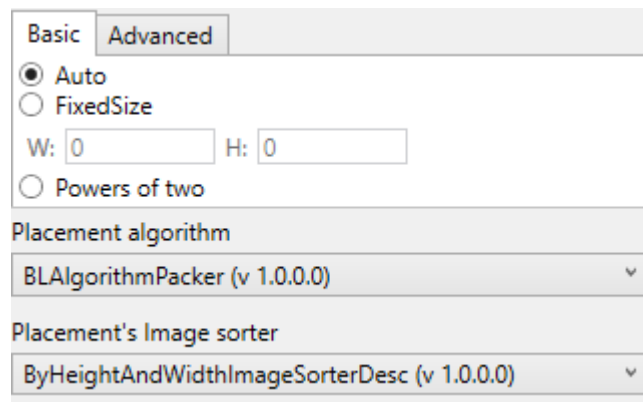


Figure 5.4: Example of packing algorithms settings.

The Figure 5.4 shows the Basic mode with a selection of `UnknownSizePacker` (Auto) minimum bounding box finder, `BLAlgorithm` placement algorithm and an

image sorter which sorts the images in the descending order first by their height and then by their width.

The *Advanced* mode differs from the *Basic* mode in that the user is able to select any of the loaded minimum bounding box finders and not only any of the three default minimum bounding box finders mentioned above. However, the plugin developers might want to disable parameterization of their minimum bounding box finder, so in the advanced mode, it does no longer hold that the user is always select placement algorithm used by the minimum bounding box finder as it was in the basic setting which shows only default implementations that are fully parameterizable.

The rest of the settings for texture atlas generation which is shown in Figure 5.5 is shared by both *Basic* and *Advanced* modes.

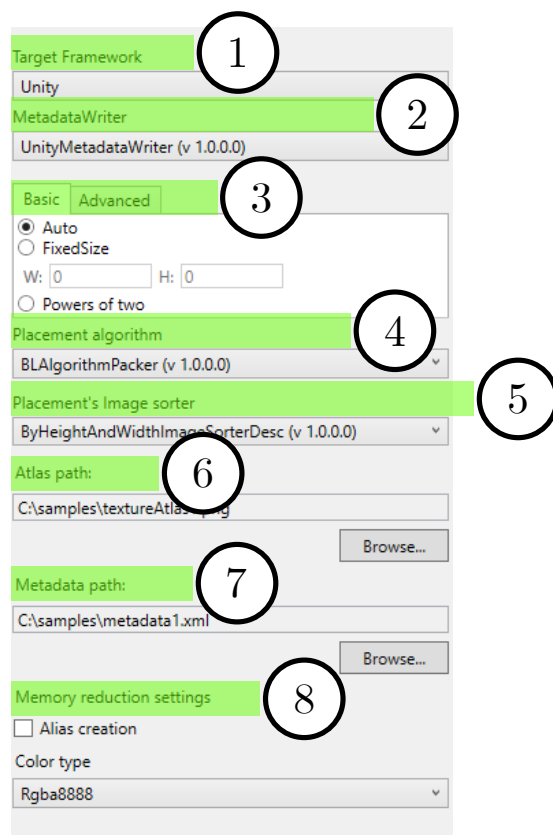


Figure 5.5: Settings of the texture atlas generation.

The Figure shows the following, packing related settings:

1. Selection of target framework which further restricts which features (settings) will be made available. In this example, the selected target framework is Unity.
2. Selection of metadata writer for a selected target framework. In this example, the selected metadata writer creates metadata for the Unity game engine.
3. Selection of packing mode which was already described above in this Section.

4. Placement algorithm used by the selected minimum bounding box finder. In this example, the selected minimum bounding box finder is determined by the selected packing mode **Auto** and it will be using the BL algorithm as its subroutine.
5. Selection of image sorter that will be used by the selected placement algorithm.
6. A path where the texture atlas bitmap (.png) will be saved.
7. A path where the texture atlas metadata (in this case .xml) will be saved.
8. Memory reduction settings which allow enabling alias creation which was already described in Section 1.2 and color type of the generated texture atlas bitmap.

When the user clicks on the *Generate* button, the texture atlas and its metadata are generated and stored at the specified paths. When no path is specified, the texture atlas would only be generated but not stored.

Using the settings from Figure 5.5 with the images shown in Figure 5.6 produces an output that was shown in Figure 5.1.

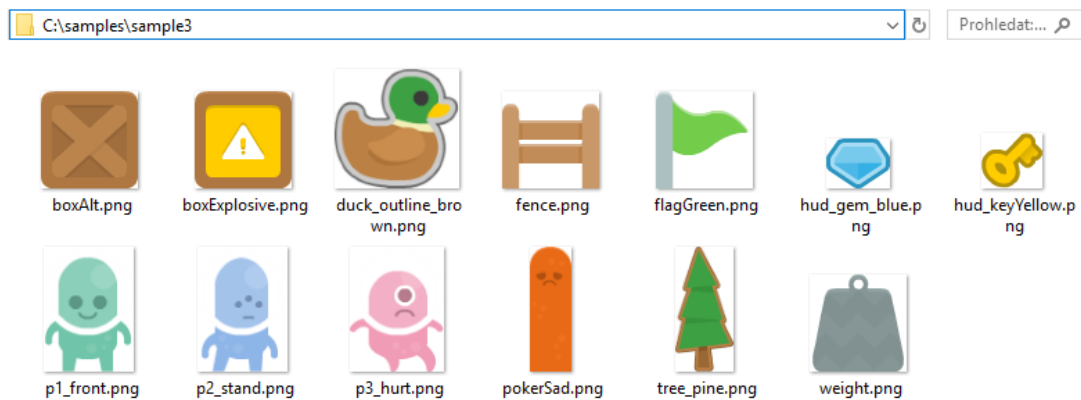


Figure 5.6: Images used to generate the texture atlas.

During the texture atlas generation, a progress bar is shown. The progress bar either displays a percentage representing the amount of generation that is done or it only indicates (without the percentage) that the algorithm is working. The form of progress bar depends on the minimum bounding box that is used. The user is able to cancel the generation by clicking on the *Cancel* button. It is important to mention that it is not ensured that the selected minimum bounding box accepts the cancellation request, although plugin developers are advised to accept that cancellation request. When the texture atlas is being generated, other packing related settings is disabled until the generation finishes. A progress bar showing the progress of **GeneticMinimumBoundingBoxFinder** is shown in Figure 5.7.

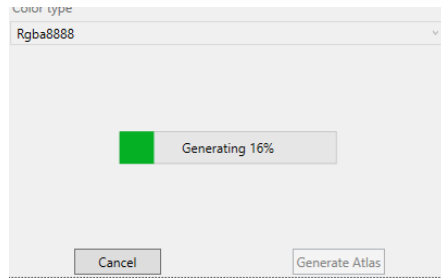


Figure 5.7: Progress bar during texture atlas generation using GeneticMinimumBoundingBoxFinder.

5.7 Interacting with the texture atlas

Once the texture atlas is generated, the user is able to select individual images within the texture atlas by clicking on them. If an image within the texture atlas is clicked, two situations can occur: the image is not selected yet, therefore it will be selected now, or the image is already selected and therefore it will be unselected. This behavior allows the user to select multiple images. The selected images are highlighted as can be seen in Figure 5.8



Figure 5.8: Selection of images inside the texture atlas.

The Figure 5.8 shows a texture atlas consisting of 7 images where on the left of this figure there are no image selected, while on the right, three images have been selected.

Apart from selecting the individual images, the user is also allowed to select all the images at once using the *Select all* button. An inverse operation, that is, to unselect all the rectangles could be done by using the *Unselect all*. The user can also show borders around the individual images by using the *Show borders* button.

The user could also zoom the texture atlas using the mouse wheel when the mouse cursor is placed above the texture atlas. When the texture atlas is zoomed so much that it does not fit into the window, scroll bars are shown.

5.8 Managing the loaded images

The third part of the Figure 5.1 contains a tree view that shows all the currently loaded images. Right-clicking on any of the loaded images (or any of the folders) shows a context menu that allows the user to unload the selected image (or whole folder which unloads all the images recursively) by clicking on *Unload* option in the context menu. The context menu also contains an option called *Open in File Explorer* which opens the system's file explorer at a location where the selected file or folder is located. The tree view with loaded images and its context menu is illustrated in Figure 5.9.

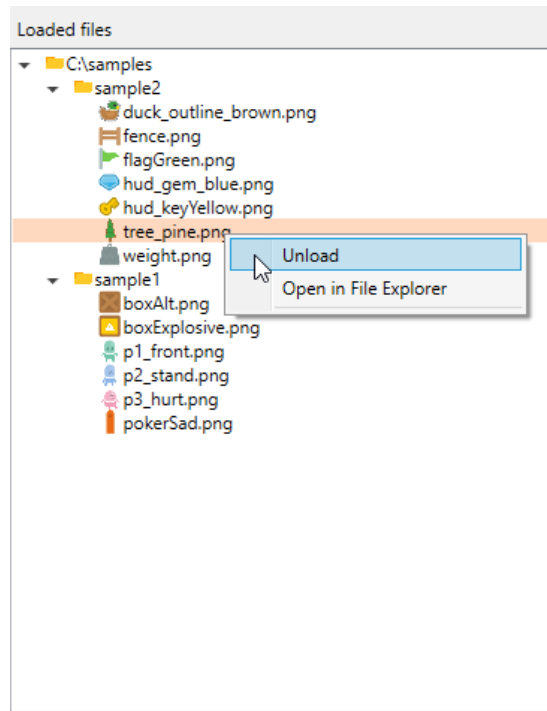


Figure 5.9: Tree view displaying files loaded in PaunPacker.

5.9 Working with Image Processors

The region containing the image processors is shown in the bottom-right of the main window as can be seen from Figure 5.1. This region contains a combo box which lists all the loaded image processor and if the selected image processor has its own GUI, that GUI is shown directly below that combo box as illustrated in Figure 5.10.

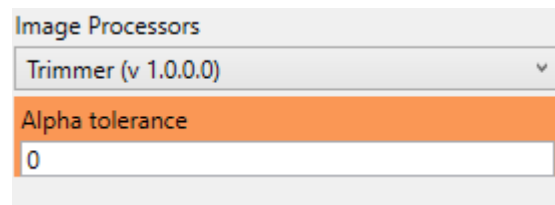


Figure 5.10: A GUI of **Trimmer** image processor.

The Figure 5.10 shows the **Trimmer** image processor which has GUI that allows to specify alpha tolerance that serves as an upper bound when determining which pixels should be considered as transparent and which should not.

To invoke the image processor and process the images, the user should select the images for processing and then click on the *Process images* button. This will process the selected images and cause the texture atlas (if already generated) to be invalidated. Application of image processors on the individual images is not recorded in the project file so whenever a project is opened, the images are in its original state. The original images are neither modified nor overwritten by image processors.

The way how the implemented image processors works go in hand with what was already described in Section 1.2. However, it should be mentioned that the **PaddingAdder** performs the inner padding as it was named and described in the Section 1.2. Another note is that the alias creation feature only affects the output texture atlas file and metadata file but not the texture atlas shown in the PaunPacker's window.

Image processing example

The following figures illustrate the usage of **Trimmer** image processor to perform trimming of selected textures. In these examples, there are three textures being loaded by the PaunPacker which are shown in the Figure 5.11. The first (taken from the left) has a background filled with full opacity black color, the second has a blue background with 50% opacity and the last texture has a fully transparent background.

1. Start by running the PaunPacking, loading the textures, generating the texture atlas and then showing the borders of the loaded textures. This step should produce the result which is depicted in Figure 5.11.

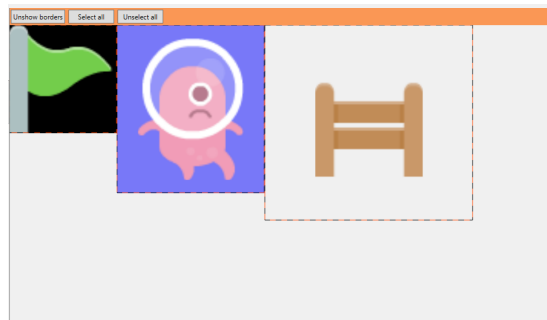


Figure 5.11: Loaded images with shown borders

2. Continue by selecting the **Trimmer** image processor as shown in Figure 5.12

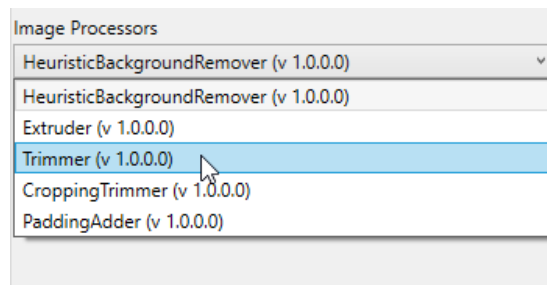


Figure 5.12: Selecting the **Trimmer** image processor

3. When the **Trimmer** is selected, select all the textures using the *Select all* button as shown in the Figure 5.13

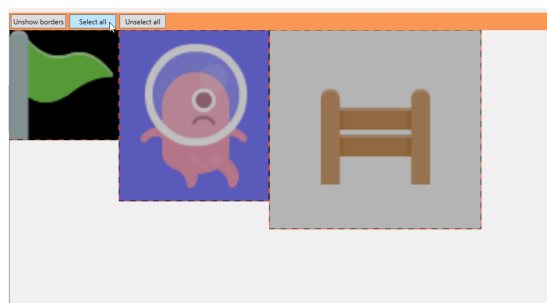


Figure 5.13: Selecting all the loaded textures.

4. After selecting the textures, set the **Trimmer**'s *alpha tolerance* parameter via the **Trimmer**'s GUI and process the selected images using the *Apply on selected* button as illustrated in Figure 5.14.

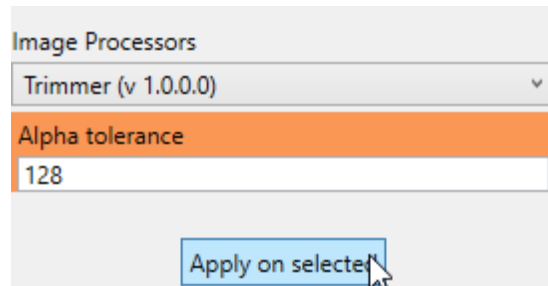


Figure 5.14: Applying the **Trimmer** on selected textures.

5. When the textures are processed, the texture atlas should automatically get re-generated into from trimmed textures and should be the same as the texture atlas depicted in Figure 5.15.

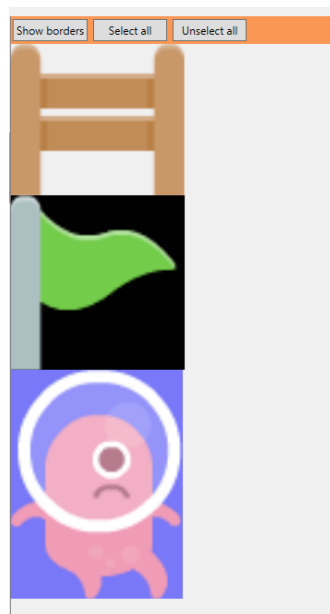


Figure 5.15: Resulting texture atlas after the textures were trimmed.

5.10 Benchmarks

The PaunPacker also contains benchmarks which the user could run by executing the *BenchmarkRunner.exe* located in the *Setup/Benchmarks* folder. The *BenchmarkRunner.exe* allows the user to perform benchmarks with placement algorithms and bounding box finders that were loaded from plugins. Benchmarks are presented in the console window and the whole benchmark runner contains a very simple (but self-describing) CLI where the user is able to select which types should be tested (either placement algorithms or minimum bounding box finders) and then which type of test should be executed. The CLI of *BenchmarkRunner.exe* is shown in Figure 5.16.



```
PaunPacker Benchmarks
=====
Commands that could be entered (without quotes) and confirmed by pressing ENTER:
Type 'm' to run MinimumBoundingBoxFinder benchmarks
Type 'p' to run PlacementAlgorithm benchmarks
Type 'q' to quit the program
Type 'h' to display help
Type 'c' to clear the output
=====

m
Select image sorter to be used:
Enter 'a' to select ByHeightAndWidthImageSorter
Enter 'b' to select ByHeightAndWidthImageSorterDesc
Enter 'c' to select PreserveOrderImageSorter
b
Select placement algorithm to be used:
Enter 'a' to select BLAlgorithmPacker
Enter 'b' to select SkylineAlgorithm
Enter 'c' to select MaximalRectanglesAlgorithm
c
```

Figure 5.16: Command line interface of benchmarks.

Currently there are types of tests:

- Pack/place squares of sizes $1 \times 1, \dots, k \times k$ where the $k \in \{1 \times 1, \dots, n \times n\}$ is a parameter specified by the user.
- Pack/place n rectangles with random dimensions where the n is a parameter specified by the user. For this kind of test, the user is also able to select random seed in order to (try to) make the benchmark results reproducible.

It should be mentioned, that only types having simple constructor dependencies (only *IImageProcessor/IPlacementAlgorithm*) are included in the tests.

The output of the tests lists (to the console), for each n and each tested type, the average area of the packing result produced by the tested type. An example of benchmark output is shown in the Figure 5.17.

The Figure 5.17 shows the results of a benchmark of three minimum bounding box finders:

- *PowerOfTwoSizePacker*
- *UnknownSizePacker*
- *GeneticMinimumBoundingBoxFinder*

In the benchmark, these three minimum bounding box finders were tested on the sequences of input squares $1 \times 1, \dots, k \times k$ where the $k \in \{1 \times 1, \dots, n \times n\}$ and the n was given as a parameter. The benchmark tracks execution times and also arithmetic mean of packing result areas. Notice that the best algorithm (in

```

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	792.80 us	0.8845 us	2.439 us	38.70 us	1
TestGeneticMinimumBoundingBoxFinder	2	85.44 us	1.8119 us	8.2427 us	79.30 us	1
TestPowerOfTwoSizePacker	2	49.19 us	1.0251 us	3.360 us	45.10 us	8
TestUnknownSizePacker	2	1,109.99 us	13.3173 us	11.121 us	1,109.22 us	6
TestGeneticMinimumBoundingBoxFinder	2	79.18 us	1.1990 us	1.063 us	78.80 us	1
TestPowerOfTwoSizePacker	2	54.70 us	1.2489 us	1.723 us	54.30 us	1
TestUnknownSizePacker	2	1,372.20 us	26.2453 us	34.1723 us	1,372.20 us	1
TestGeneticMinimumBoundingBoxFinder	4	152.88 us	2.9359 us	3.381 us	152.86 us	1
TestPowerOfTwoSizePacker	4	94.60 us	1.4181 us	1.184 us	94.60 us	1
TestUnknownSizePacker	4	1,935.59 us	37.5896 us	47.539 us	1,923.76 us	1
TestGeneticMinimumBoundingBoxFinder	4	125.11 us	2.4592 us	2.415 us	124.24 us	1
TestPowerOfTwoSizePacker	4	100.06 us	1.2072 us	1.873 us	100.09 us	1
TestUnknownSizePacker	4	2,331.27 us	44.4155 us	49.368 us	2,313.91 us	1
TestGeneticMinimumBoundingBoxFinder	6	279.69 us	5.6741 us	10.375 us	276.00 us	1
TestPowerOfTwoSizePacker	6	176.72 us	3.5098 us	3.447 us	175.66 us	1
TestUnknownSizePacker	6	3,182.46 us	20.8045 us	19.461 us	3,185.67 us	1
TestGeneticMinimumBoundingBoxFinder	6	281.64 us	5.2410 us	3.030 us	279.33 us	1
TestPowerOfTwoSizePacker	6	246.64 us	4.0640 us	8.805 us	245.80 us	1
TestUnknownSizePacker	6	4,175.41 us	105.4534 us	300.864 us	4,139.20 us	1
TestGeneticMinimumBoundingBoxFinder	7	567.29 us	11.2930 us	24.788 us	559.28 us	1
TestPowerOfTwoSizePacker	7	419.02 us	8.2242 us	36.613 us	411.85 us	1
TestUnknownSizePacker	7	5,380.71 us	117.8111 us	346.613 us	5,186.73 us	1
TestGeneticMinimumBoundingBoxFinder	7	472.31 us	9.2775 us	12.385 us	470.16 us	1
TestPowerOfTwoSizePacker	7	457.26 us	4.4714 us	4.183 us	457.38 us	1
TestUnknownSizePacker	7	6,058.38 us	120.0933 us	253.317 us	5,964.61 us	1
TestGeneticMinimumBoundingBoxFinder	9	523.09 us	10.5187 us	28.438 us	515.42 us	1
TestPowerOfTwoSizePacker	9	770.81 us	17.3922 us	30.621 us	768.16 us	1
TestUnknownSizePacker	9	8,651.11 us	219.1280 us	625.184 us	8,732.36 us	1
TestGeneticMinimumBoundingBoxFinder	10					

Figure 5.17: Output of the benchmarks.

terms of minimum packing area) was the `GeneticMinimumBoundingBoxFinder` while the worst was the `PowersOfTwoSizePacker`. This observation goes in hand with the intuition that the `PowersOfTwoSizePacker` creates a wasted transparent space in order to satisfy the power of two requirement.

The benchmark also shows that the `UnknownSizePacker` was out-performed (for $N = 8$ and $N = 9$) by the `GeneticMinimumBoundingBoxFinder`. This is because the genetic algorithm performs a lot more iterations and also contains random mutation which could result in better results.

However, when it comes to speed, the `GeneticMinimumBoundingBoxFinder` was by far the slowest of the tested minimum bounding box finders. Observe that the `PowerOfTwoSizePacker` was faster than the `UnknownSize` packer, the reason for this result is the number of bounding boxes enumerated and attempted to get packed. The `PowersOfTwoSizePacker` tests much less bounding boxes (only powers of two) then the `UnknownSizePacker`.

Please beware that the benchmark runner performs a lot of warm-up iterations before doing the actual workload and that the benchmarks are generated and compiled dynamically, therefore the execution of benchmarks might take slightly longer time to complete. For example, the benchmark whose results were shown in Figure 5.17 on the laptop setup shown at the top of the Figure 5.17 took about 23 minutes to complete.

6. Creating Plugins Tutorial

This chapter provides a tutorial for developers wishing to create new plugins. The tutorial is divided into three parts that cover the following scenarios (ordered by their complexity):

1. Creating plugins without GUI
2. Creating plugins with GUI
3. Creating plugins requiring other (external) dependencies

Before delving into details of tackling the particular scenario, the prerequisites common to all these three scenarios should be mentioned. These prerequisites are the topic of the next Section 6.1.

6.1 Prerequisites and common guideline

To create a plugin `.dll` one should first create a Visual Studio Class Library project targeting either the `.NET Standard` or the `.NET Core` (in the case that the plugin needs GUI or depends on something that is not `.NET Standard` compliant, e.g. on some NuGet package), add the implementations of any extensible components to this project and then ensure that these implementations will get properly exported in order to be subsequently imported by the `PaunPacker`. The sections in the rest of this chapter will assume that the developer already has an empty project created.

Compiling the project with the plugin yields a `.dll` file. This file should be copied to the *plugins* folder. The implementations of the extensible components that will be exported properly will get loaded by the `PaunPacker` upon `PaunPacker`'s startup.

As it was already mentioned in Chapter 3, `PaunPacker` combines two approaches when searching for types within the plugin that should be imported. First, the types that are marked with `Export` attribute and second, the types that implement `IModule` interface. The `IModule` interface has two methods that have to be implemented: `OnInitialized` and `RegisterTypes` that are called automatically (by the Prism) when the plugin is being loaded.

The expected use case of first (`Export`) approach is for exporting simple classes that have parameterless constructor, by simply decorating the class with the `Export` attribute. Note that the `Export` attribute is defined in MEF assembly, so the developer should add appropriate `.dll` references, and the most convenient way to do it is to download a NuGet package called `System.Composition`.

The second approach is expected to be used in more complicated situations, for example, when the exported types do not have parameterless constructor or they need GUI. The difference between the second and the first approach is that in the second approach, instead of letting all the types that should be exported to implement the `IModule` interface, it is better to create only a single class that will implement the `IModule` (call this class `PluginEntry`). `PluginEntry` could be thought of like a plugin's entry point and all the types that should be

exported could be exported from `PluginEntry` by using either the Unity IoC container that can be obtained from the `IContainerProvider` that is passed to the `OnInitialized` method or the `IContainerRegistry` that is passed to `RegisterTypes` method. The details about how these types could be exported via the IoC container are described in the following Sections 6.4 and 6.6. Using the `IModule` interface requires the project to reference `Prism.Wpf.dll` NuGet package.

Exporting metadata about plugin

The developer may want to export metadata either about the whole plugin—that is about the class that implements the `IModule` interface—or about the individual types that are exported from the plugin. These two scenarios are different and their distinction comes from a fact that a single plugin class could serve as a plugin entry point that will export several types.

In order to export metadata about the whole plugin, the plugin should be decorated by the `PluginMetadataAttribute` that accepts the type of the plugin, name of the plugin, description of the plugin, author, version of the plugin and the types that are exported from this plugin.

To export metadata about the individual types that are exported from the plugin there exists the `ExportedTypeMetadataAttribute` which could either decorate the exported type (if possible) or the whole plugin. This attribute also accepts the type, name, description, author and the version of the exported type. The name property of the `ExportedTypeMetadataAttribute` will be displayed in the `PaunPacker`'s main window. The reason why this attribute could also decorate the plugin itself is that some plugins only serve as a providers of types implemented somewhere else (for example in the `PaunPacker.Core`) and in such a case, it is desired to give the plugin developer an ability to export metadata about the types being exported so that the user could see the exported types in the `PaunPacker`.

Reporting progress

Because the process of creating a texture atlas may take some time, it seems reasonable to inform the user about the progress of the packing. For this reason, `PaunPacker` contains a progress bar that is shown to the user when then texture atlas is being created. The idea of the progress was extended so that currently the progress bar is able to display the progress of not only the packing but also of the metadata writing, file loading, etc. The important thing for the plugin developers is, that there exists an interface called `IProgressReporter` which is automatically inherited by the following interfaces:

- `IPlacementAlgorithm`
- `IMinimumBoundingBoxFinder`
- `IMetadataWriter`

This means that all extensible components—except for the image processors and image sorters—could report information about their progress by implementing this interface. Note that the final output of the packing process is determined by

the minimum bounding box finder, not by the placement algorithm, but the minimum bounding box finder could use the progress of the used placement algorithm in order to obtain a better estimate of the progress.

The `IReportsProgress` interface has an integer property called `Progress` which should return the current progress of the algorithm (an integer between 0 and 100 that represents the percentage of the algorithm's progress) and an event called `ProgressChanged` which should be raised whenever the progress of the algorithm has changed (increased). This interface also contains a property called `ReportsProgress` which could be useful in situations where some type is only a wrapper around another type and therefore its ability to report progress is based on the ability (to report progress) of the wrapped type being used. This approach is used, for example, in the `FixedSizePacker`.

Targeting specific game frameworks

It was previously mentioned, that the `PaunPacker.GUI.WPF.Common` project contains an attribute called `AvailableTo` which could decorate any of the exported types. The reason for the presence of this attribute is the fact that certain features cannot be described by a metadata format used by some game framework "A" but could be described by a metadata format used by another game framework "B". One approach to this issue would be to silently exclude this information from the resulting metadata, but that could confuse the user if the user has adjusted the packing settings in some way that eventually is not reflected by the packing result. Because of this reason, it is better to explicitly decorate the exported types with the target framework and hide the features that are not available to the framework that the user has selected (this was already briefly described in Section 4.5). It is a responsibility of the plugin developer to mark the exported types with an `AvailableTo` attribute.

The `AvailableTo` attribute has two public constructors, the first accepts `FrameworkIDs` of the frameworks to which the type decorated by the `AvailableTo` should be made available to. And the second constructor that does not accept any parameters. If the `AvailableTo` is constructed using the parameterless constructor then it means that it is available to all the frameworks.

Metadata exporters then should be marked with an `TargetFramework` attribute which declares the game framework to which the given `MetadataExporter` exports the metadata. `TargetFramework` attribute has two constructors, one that accepts a `FrameworkID` of the target framework and second that has no parameters. When the `TargetFramework` attribute is constructed using the parameterless constructor, it means that the metadata exporter marked with such attribute targets all (arbitrary) game frameworks. This is useful, for example when the target framework is not present in the `FrameworkID` enum.

6.2 Minimal plugin template

This section presents two examples which are representing a template for a minimal, fully working plugin. These templates could be used by the plugin developers as a starting point when developing a plugin. The templates differ in the approach that they take when exporting the plugin. The first template uses the MEF's `Export` attribute while the second template implements. The first template is given in Listing 15.

```
1      [PluginMetadata(typeof(SomeExportedType),
2          nameof(SomeExportedType),
3          "<Description>",
4          "<Author>",
5          "<Version>",
6          typeof(SomeExportedType))]
7      [ExportedTypeMetadata(typeof(SomeExportedType), nameof(SomeExportedType),
8          "<Description>", "<Author>", "<Version>")]
9      [Export(typeof(SomeExportedType))]
10     public sealed class SomeExportedType :
11         <IImageProcessor |
12         IMetadataWriter |
13         IPlacementAlgorithm |
14         IImageSorter |
15         IMinimumBoundingBoxFinder>
16     {
17         //... Implementation
18     }
```

Listing 15: The first (MEF based) template for (almost) minimal fully working plugin

The Listing 15 shows a plugin template that exports a type that is called `SomeExportedType` by using the MEF's `Export` attribute. Every text enclosed in the "<>" should be appropriately replaced by a meaningful information. The `SomeExportedType` class should implement one of the following interfaces:

- `IImageProcessor`
- `IMetadataWriter`
- `IPlacementAlgorithm`
- `IImageSorter`
- `IMinimumBoundingBoxFinder`

The second template is shown in the Listing 16.

```

1  [PluginMetadata(typeof(PluginEntryPoint),
2      nameof(PluginEntryPoint),
3      "<Description>",
4      "<Author>",
5      "<Version>",
6      "<typeof(ExportedType1),...,typeof(ExportedTypeN)>"
7      [ExportedTypeMetadata(typeof(ExportedType1), nameof(ExportedType1),
8          "<Description>", "<Author>", "<Version>")]
9      //...
10     [ExportedTypeMetadata(typeof(ExportedTypeN), nameof(ExportedTypeN),
11         "<Description>", "<Author>", "<Version>")]
12 public class PluginEntryPoint : IModule
13 {
14     //...
15     public void OnInitialized(IContainerProvider containerProvider)
16     {
17         unityContainer = containerProvider.Resolve<IUnityContainer>();
18         //Use the unityContainer to export types
19     }
20
21     public void RegisterTypes(IContainerRegistry containerRegistry)
22     {
23
24     }
25 }

```

Listing 16: The second (IModule based) template for (almost) minimal plugin

The Listing shows a template which uses the unity container for exporting the types. Similarly to the Listing 15, the values enclosed in <> should be filled-in by the plugin developer appropriately.

6.3 Creating plugins without GUI

The simplest scenario is to create plugins that have neither the GUI nor any external dependencies. This is the scenario that will be considered throughout this section.

First of all, the developer should create implementations of the extensible components (possibly of more than one extensible component). For example, the developer may decide to create an implementation of a metadata writer, so a new class (call it `NewMetadataWriter`) should be created and that class should implement the `IMetadataWriter` interface. Once this interface is implemented, the developer needs to ensure that the `NewMetadataWriter` will get imported by the `PaunPacker` properly. This could be done in two ways as it was already mentioned in the Section 6.1 and the developer could use one of the templates from Section 6.2 as a starting point.

The first way is to decorate `NewMetadataWriter` class with the `Export` attribute. Once the class is decorated with the `Export` attribute, the project is built and the resulting `.dll` gets copied into the plugin folder, the `NewMetadataWriter` should appear in `PaunPacker`'s option (if the `PaunPacker` was running prior to the `.dll` getting copied into the plugin folder, it has to be restarted)

The second way is to let the class to also implement (apart from implementing the `IMetadataWriter`) the `IModule` interface. This approach requires a slightly more code but it offers additional flexibility because `Prism` will call the methods prescribed by this interface and so the plugin developer could use them for initialization or other tasks. However, for very simple plugins containing only a single implementation of the extensible component, the `Export` attribute approach seems like a preferable and more appropriate way to go.

6.4 Creating plugins with GUI

When creating plugins with GUI, the developer has to create a class that implements the `IModule` interface. This class does not have to be—and preferably it is not—the same class as the class providing an implementation of a certain modular component, but instead, it should be a separate class that will serve as a plugin's entry point. Whenever a plugin developer develops a plugin with GUI, it is expected that the whole plugin project adheres to the MVVM pattern, therefore the developer should also create a view model in addition to the view. The plugin developer should simply create an instance of the view and view model and then set view's `DataContext` to its view model.

The view instance could be exported from the `OnInitialized` method by using the IoC container and registering the instance of the view as a `UserControl` under a key that should be obtained by using the `PluginViewWiring` class which was already described in Section 4.5. The Listing 17 demonstrates this process on a part of code taken from the plugin with default implementations.

Because the MVVM pattern is preserved, the view is managed by its view model and the view model then could export the particular extensible component with parameters that were obtained via the view. An example of such a parameter is the size of the population inside the genetic minimum bounding box finder. When some parameters are changed, it is sufficient to create a new instance of the

```

1 public void OnInitialized(IContainerProvider containerProvider)
2 {
3     IUnityContainer container = containerProvider.Resolve<IUnityContainer>();
4
5     var extruderView = new ExtruderView()
6     {
7         DataContext = new ExtruderViewModel(container)
8     };
9
10    container.RegisterInstance<UserControl>(PluginViewWiring
11        .GetViewName(typeof(Extruder)), extruderView);
12 }

```

Listing 17: Exporting view from a plugin

extensible component having these new parameters. But instead of creating new instance registration via the Unity IoC container, the type should be registered using the Unity's `RegisterFactory` method which allows to register a factory method for the type so that whenever that type is being resolved, the registered factory method is called and its return value is returned as the resolved instance. The factory method allows the plugin developer to register the instance with correct parameters. An example of using the `RegisterFactory` method to export an instance of `Extruder` with up-to-date parameter is shown in Listing 18.

```

1 public ExtruderViewModel(IUnityContainer unityContainer)
2 {
3     unityContainer.RegisterFactory<Extruder>((_) =>
4     {
5         return new Extruder(Amount);
6     });
7 }

```

Listing 18: Exporting instance of `GeneticMinimumBoundingBoxFinder`

When the `App` loads this plugin and imports all the types that the plugin exports, it also traverses through the imported types and for each one of them, it checks whether that type has a view associated with it and if it does, then the view is registered into appropriate region (the selection of the region is made by the `PaunPacker.GUI` as already explained and justified in Section 3.5). The registered view is shown once the user selects the extensible component which is associated with the view. For example, when the plugin exports the minimum bounding box finder called `MbbfA` which has a view called `ViewA` associated with it, the view gets registered into the region for minimum bounding box finders and when the user selects `MbbfA` as an algorithm for creating the texture atlas, the region manager shows the `ViewA` allowing the user to interact with it.

6.5 Managing external dependencies

As it was already discussed in Sections 4.5 and 3.6 the PaunPacker takes emphasis on the parameterization of the whole packing pipeline so that for a selected minimum bounding box finder (MBBF) the user is allowed to select a placement algorithm that will be used. Moreover, the placement algorithm could be further parameterized by the image sorter. The main window supports this parameterization by showing combo boxes to the user allowing to select individual parts of the packing pipeline. But sometimes, the plugin developer may want to prohibit the user from this parameterization. This section will discuss all the possible scenarios that might arise and suggests solutions to these scenarios. The combo boxes inside the main window that allow the user to parameterize the packing pipeline are illustrated in Figure 6.1.

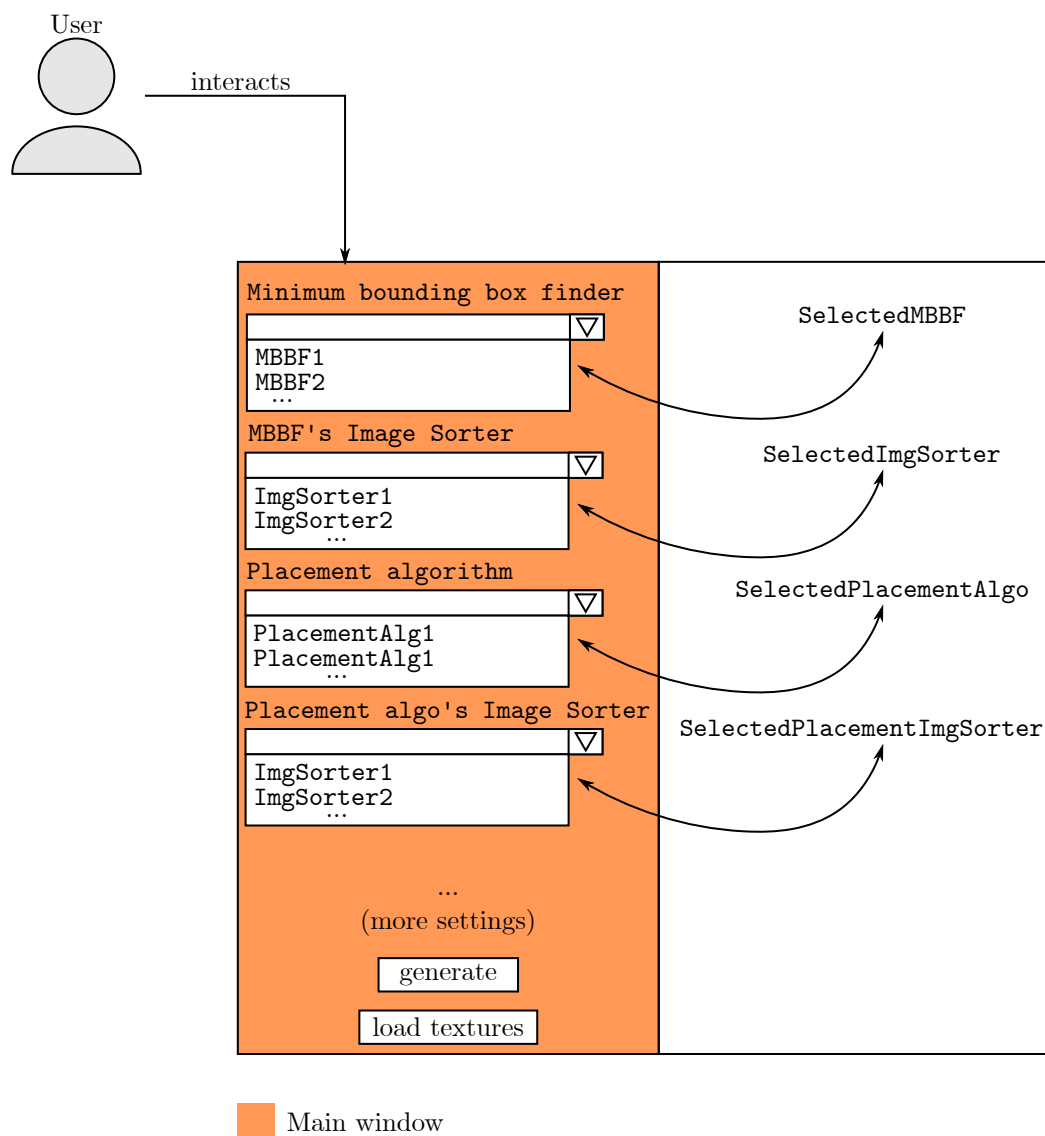


Figure 6.1: Parameterization of the packing pipeline

The Figure 6.1 shows a part of PaunPacker's main window that contains combo boxes that allow the user to select MBBF, placement algorithm used by the selected MBBF, image sorter used by the selected placement algorithm and

also image sorter used by the MBBF. Notice, that image sorter of the MBBF is separated from the image sorter of the placement algorithm and even though it is possible to select different image sorter for placement algorithm and for the MBBF, it is discouraged—unless there is a justified reason for it, for example, when the MBBF wants to do some preprocessing on sorted input before running the placement algorithm—doing so because of the reasons given in Section 3.6.

An MBBF could have external dependencies that have to be provided from outside of the plugin exporting the MBBF in order to properly instantiate the MBBF. In this case, the considered dependencies are the parameters inside the constructor of the MBBF. There are three scenarios that could happen (regarding the constructor dependencies):

1. No external dependencies (self-contained)
2. Some of the parameters are external dependencies (partially contained)
3. All the parameters are external dependencies

SelfContained

When the MBBF does not have any external dependencies it should be decorated by the **SelfContained** attribute. This will have the effect that the user is not allowed to select the placement algorithm used by the MBBF because the combo box for selecting placement algorithm will be hidden when the user selects MBBF that is decorated by **SelfContained** attribute. **SelfContained** could also decorate classes implementing **IPlacementAlgorithm** disallowing the user to select image sorter used by the placement algorithm.

PartiallyContained

If the MBBF has several dependencies and some of them are external while the other (**IPlacementAlgorithm** and/or **IImageSorter**) are not then this extensible component should be decorated by the **PartiallyContained** attribute. When the MBBF decorated by this attribute is loaded by the PaunPacker, its public constructors are scanned and if none of them accepts the **IPlacementAlgorithm**, then the combo box for selecting the placement algorithm is hidden. Similarly, if none of the constructors accepts **IImageSorter**, the combo box for choosing MBBF's image sorter is hidden.

Similarly, when a class implementing **IPlacementAlgorithm** is decorated by the **PartiallyContained** attribute, the constructors are scanned in the similar way as it was the case with MBBF, looking for the existence of a constructor accepting **IImageSorter**. If none of the public constructors accepts **IImageSorter** then the combo box for selecting the placement algorithm's image sorter will be hidden.

Default behavior

When the exported type is decorated neither by the **SelfContained** attribute nor by the **PartiallyContained** attribute then the default behavior—meaning that the type is fully parameterizable—is observed.

6.6 Creating extensible plugins

This section presents a more general approach to the resolution of dependencies by allowing the plugin developer to create a plugin which would be able to obtain dependencies registered by other plugins.

The approach from a previous section applies to minimum bounding box finders and placement algorithms whose dependencies are selected by the user using the PaunPacker's GUI. Other extensible components could also be parameterized, but their parameterization is not built inside PaunPacker's GUI, so these aforementioned scenarios do not apply to them and instead, they should resolve the dependencies on their own using a more general approach which is to create a plugin with a view, that will load all (including those exported from other plugins) the dependencies of a certain type and displays those types in its view. The user is then able to select any of the listed dependencies and the view model will export an instance parameterized by this selection.

As an example consider the Skyline algorithm which could be parameterized by the `IRectAndPointPicker`. The desired behavior is to allow other plugin developers to develop implementations of `IRectAndPointPicker` that the user could later select to create a parameterized instance of the Skyline algorithm. The way how this behavior could be reached is depicted in Listing 19.


```

1 public void OnInitialized(IContainerProvider containerProvider)
2 {
3     //...
4     //Register known RectAndPointPickers
5     unityContainer.RegisterType<LightweightRectAndPointPicker>();
6     unityContainer.RegisterType<MinimalAreaWasteRectAndPointPicker>();
7     //Register the Skyline algorithm
8     unityContainer.RegisterType<SkylineAlgorithm>();
9     var eventAggregator = unityContainer.Resolve<IEventAggregator>();
10    eventAggregator.GetEvent<ModulesLoadedEvent>().Subscribe(OnModulesLoaded);
11 }
12 private void OnModulesLoaded (ModulesLoadedPayload payload)
13 {
14     var view = new SkylineAlgorithmView()
15     {
16         DataContext = new SkylineAlgorithmViewModel(payload
17             .GetLoadedTypes<ISkylineRectAndPointPicker>()
18             .Concat(unityContainer.Registrations
19                 .Select(x => x.RegisteredType)
20                 .Where(x => typeof(ISkylineRectAndPointPicker)
21                     .IsAssignableFrom(x) && !x.IsAbstract && !x.IsInterface))
22             .Distinct(), unityContainer)
23     };
24     unityContainer.RegisterInstance<UserControl>(PluginViewWiring
25         .GetViewName(typeof(SkylineAlgorithm)), view);
26 }
27 public SkylineAlgorithmViewModel( //Constructor of SkylineAlgorithmViewModel
28     IEnumerable<Type> loadedISkylineRectAndPointPickerTypes,
29     IUnityContainer unityContainer)
30 {
31     SkylineRectAndPointPickerVMs = loadedISkylineRectAndPointPickerTypes
32         .Select(x => new RectAndPointPickerViewModel(x));
33     unityContainer.RegisterFactory<ISkylineRectAndPointPicker>((_) =>
34     {
35         try
36         {
37             var picker = (ISkylineRectAndPointPicker)unityContainer
38                 .Resolve(SelectedRectAndPointPickerVM.RectAndPointPickerType);
39             return picker ?? new MinimalAreaWasteRectAndPointPicker();
40         }
41         catch (ResolutionFailedException) //Return default implementation
42         {
43             return new MinimalAreaWasteRectAndPointPicker();
44         }
45     });
46 }

```

Listing 19: Creating and exporting an extensible plugin.

The Listing 19 shows a general approach to export types that require resolution of external dependencies illustrated on the Skyline algorithm. As can be seen from this Figure, the implementation of the `OnInitialized` method should obtain a Unity IoC container and resolve `EventAggregator` from it. The Unity IoC container should be used to register a type that should be exported (the `SkylineAlgorithm` in this case). Then a handler for a `ModulesLoadedEvent` should be subscribed using the `EventAggregator` and it should get all the dependencies (the implementations of `ISkylineRectAndPointPicker` in this case) and pass them to the view model. The view model then registers these dependencies so that whenever the `PaunPacker` tries to resolve the exported type, the container resolves the registered dependencies and instantiates the exported type using them. So in the case of Skyline algorithm, when the `PaunPacker` tries to instantiate the `SkylineAlgorithm`, the implementation of `IRectAndPointPicker` registered in the view model (shown in the Figure above) will be used.

An alternative approach would be to remove the type registration from the `OnInitialized` method and register factory in the view model. This alternative is partially illustrated in Listing 20.

```

1  public SkylineAlgorithmViewModel(
2      IEnumerable<Type> loadedISkylineRectAndPointPickerTypes,
3      IUnityContainer unityContainer)
4  {
5      SkylineRectAndPointPickerVMs = loadedISkylineRectAndPointPickerTypes
6          .Select(x => new RectAndPointPickerViewModel(x));
7      unityContainer.RegisterFactory<SkylineAlgorithm>((_) =>
8      {
9          //Sorter should be safe to resolve
10         var sorter = unityContainer.Resolve<IImageSorter>();
11         try
12         {
13             var picker = (ISkylineRectAndPointPicker)unityContainer
14                 .Resolve();
15             picker ??= new MinimalAreaWasteRectAndPointPicker();
16             return new SkylineAlgorithm(sorter, picker);
17         }
18         catch (ResolutionFailedException)
19         {
20             //Return default implementation
21             var picker = new MinimalAreaWasteRectAndPointPicker();
22             return new SkylineAlgorithm(picker);
23         }
24     });
25 }

```

Listing 20: An alternative approach for exporting extensible plugins by using `RegisterFactory` method.

It is important to ensure that the exported type could always be exported, therefore it is important to provide some fallback mechanism (default implementations) for the case that none of the dependencies was registered.

To summarize this section:

- Any extensible component could have arbitrary external dependencies
- Minimum bounding box finders and placement algorithms should also use `SelfContained` and `PartiallyContained` attribute whenever appropriate.

6.7 Best practices

There are a lot of ways to achieve a similar effect when it comes to developing plugins for PaunPacker. Because the plugin developer might be overwhelmed by the number of different ways of doing the same thing, it was decided to introduce this section which presents some of the best practices when developing the plugins. These best practices are the following:

- Always use `PluginMetadata` and `ExportedTypeMetadata` attributes to export metadata about plugins.
- Prefer export by using MEF's `Export` attribute over implementing `IModule` for simple types.
- Always try to equip the exported type with a parameterless constructor.
- When developing plugins that introduce plugin-specific dependency (for example, the Skyline algorithm and its `ISkylineRectAndPointPicker`, always provide a default implementation of the dependency and export it.
- Use `TargetFramework` and `AvailableTo` attributes whenever needed.
- Mark the exported types with `SelfContained` or `PartiallyContained` attribute whenever appropriate.
- Always try to report the progress from the extensible components that support it.
- When developing extensible components, always check the cancellation token for cancellation and cancel gracefully whenever the cancellation is requested (within a reasonable amount of time).

Conclusion

To conclude this thesis, the goals mentioned in Sections 1.3, 3.1 now should be reviewed and their fulfillment evaluated:

1. *Create a packing tool with included GUI*—It is safe to say that this goal was met because the PaunPacker solutions contain the `PaunPacker.GUI` written using WPF. Moreover, the final product looks more or less the same as the initial draft that the author of this thesis had in mind.
2. *Implement Trim, Crop, Extrude, Color type change, Heuristic mask*—All of these features are implemented, although some of them are not implemented using the most efficient or most generic algorithms solving the particular problem. For example, the heuristic mask (i.e. the background remover) is implemented in quite a simple way that definitely does not remove the background correctly in every case. Nevertheless, the background remover still works slightly better than was initially expected, because the initial idea was to implement it in a way so that it only allows to remove backgrounds from textures with a single color background.
3. *Implement Alias creation*—Successfully implemented.
4. *Provide two sample metadata exporters, first of them targeting libGDX and the second targeting Unity*—Both metadata writers (exporters) were implemented.
5. *Provide basic toolset (in a form of the dynamic library) that could be reused for future plugin development*—The reusable dynamic library is output from building the `PaunPacker.Core` project. What did better than expected is that the dynamic library is fully compliant with .NET Standard.
6. *Satisfy all of the requirements (R1 to R3) mentioned in Chapter 1*—The requirement **R1** (that the application should be free to use) is satisfied because the PaunPacker is free to use. The requirement **R2** (that the application should provide several additional features for image processing) is also satisfied because the image processors mentioned in 1. item in this list are implemented. Lastly, the requirement **R3** (that the application should be extensible) is also satisfied and fulfilled even more than it was expected because the extensible components could have their own views (GUI) and could depend on functionality exported from other plugins which is something that was not expected in the initial phase of development.
7. *Implement bottom-left algorithm (improved BL)*—The BL algorithm was implemented, although in a slightly different version than from the improved BL. The implemented BL algorithm variant tries to perform some additional steps in order to optimize the resulting packing but at the price of worse time complexity. It would be nice to either provide the original improved BL implementation or Chazelle’s efficient implementation [22] of the original BL algorithm in the future. However, the lack of a standard implementation of BL algorithm is compensated by the fact that the BL algorithm could

be implemented using proper parameterization of the Guillotine algorithm which is perfectly possible in the current implementation of the PaunPacker.

8. *Implement Guillotine algorithm*—The guillotine algorithm is implemented in a way that it is parameterizable more than it was expected. The guillotine algorithm is also accompanied by the GUI so that the user is able to parameterize it also directly from the GUI of the application.
9. *Implement Maximal Rectangles (MaxRects) algorithm*—Successfully implemented
10. *Implement Skyline algorithm*—Successfully implemented, in a parameterizable way.
11. *Implement one metaheuristic algorithm*—As it was already mentioned in Section 3.1, the genetic packing algorithm was selected as the metaheuristic algorithm that should be implemented. The genetic packing algorithm was successfully implemented and is also accompanied by the GUI that allows the user to parameterize the algorithm by the size of the population and number of iterations.

Future work

Although it could be said that the goals that were set have been accomplished, there is still a lot of space for improvement and additional work that could be done in order to improve the user experience and the whole application itself. Some of the nice-to-have features and possible improvements for the PaunPacker are mentioned in the following list:

- *Drag & Drop*—The drag & drop of input textures into the PaunPacker could dramatically improve the user experience of working with the application.
- *Undo & Redo*—This is yet another feature that users are used to using in a modern application and it would definitely improve the user's experience. The implementation of *Undo & Redo* would introduce a concept of a history which would track actions taken by the user. There are several possibilities what could this history track, for example, application of image processors.
- *Extend project files*—It seems useful to extend the amount of information that is stored within the project files by some additional project settings. In addition to project settings, it would be nice to increase the amount of state that is captured by the project, for example, the applied image processors so that they are reapplied when the project is opened. This supports the idea of implementing the *Undo & Redo* mentioned above.
- *Add per-folder settings*—Some applications allow to create settings files within individual folders, that are also applied recursively in inner folders (unless overridden). Folders with the settings files could then be tracked by the PaunPacker and images in them could automatically be loaded. The settings file then could specify, for example, sub-folders that should be excluded from the loading. A similar approach is taken in the paid version of TexturePacker.

- *Improved plugin management*—Currently, the user is only allowed to install plugins by copying them into the *plugins* folder and then show information about the plugins that were loaded and types exported from the individual plugins. It could be useful to allow the user to manage the plugins from the application, for example, allow the user to select which plugins should be installed (and then copy the appropriate files) or to uninstall the files from the application, etc.
- *Texture atlas post-processors*—The current implementation of PaunPacker only supports image processors that are processing individual images. But with the passage of time, it turned out that it would be useful to have an ability to post-process a whole generated texture atlas. Such post-processing could be made abstract by introducing the so-called *Texture atlas post-processors* that would take the generated texture atlas at the input, process it and then returned the processed version at the output. These post-processors would, therefore, be allowed to adjust the metadata corresponding to the texture atlas. The main reason is that certain operations need to look at the texture atlas as a whole and not only at the individual textures.
- Optimize the current implementations—Although the current implementations are working and are mostly implemented with a reference to a paper describing the given algorithm, there is always a place for an optimization of the algorithms, for example, by introducing parallelism.
- Add more metadata writers—Even though the PaunPacker contains quite a lot of algorithms and other functionality out-of-the-box, the number of metadata writers and therefore the frameworks that are supported—to be used as the target frameworks—is rather limited (only two implementations at the moment). It would be nice to add more metadata writers and because the PaunPacker is extensible, it should not take much effort.
- *Transfer some functionality from the main window's view model to other view models*—There are already several other view models being responsible for quite a lot of an application's logic, but the main window's view model is still enormous (in terms of the amount of code) when compared to the other view models.
- *Add support for multiple texture atlases generation*—It would be useful to provide the user an ability to automatically created multiple smaller texture atlases when the input textures could not fit into a fixed size texture atlas with dimensions specified by the user.

Bibliography

- [1] Kenney. Platformer Art Complete Pack. <https://opengameart.org/content/platformer-art-complete-pack-often-updated>, 2014. [Online; accessed 13-April-2019].
- [2] libgdx. <https://libgdx.badlogicgames.com/>. [Online; accessed 28-April-2019].
- [3] Batch (libgdx API). <https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com.badlogic.gdx.graphics.g2d.Batch.html>. [Online; accessed 12-May-2019].
- [4] libgdx/TexturePacker.java at master · libgdx/libgdx · GitHub. <https://github.com/libgdx/libgdx/blob/master/extensions/gdx-tools/src/com.badlogic.gdx/tools/texturepacker/TexturePacker.java#L349>. [Online; accessed 30-April-2019].
- [5] Esoteric Software LLC. Spine: In Depth. <http://esotericsoftware.com/spine-in-depth>. [Online; accessed 20-May-2019].
- [6] Atlas export format. <http://esotericsoftware.com/spine-atlas-format>. [Online; accessed 30-April-2019].
- [7] LearnOpenGL - Textures. <https://learnopengl.com/Getting-started/Textures>. [Online; accessed 20-May-2019].
- [8] Kenney. Platformer Characters 1 (5 characters). <https://opengameart.org/content/platformer-characters-1-5-characters>, 2017. [Online; accessed 11-May-2019].
- [9] CodeAndWeb GmbH. TexturePacker - Create Sprite Sheets for your game! <https://www.codeandweb.com/texturepacker>, 2019. [Online; accessed 14-May-2019].
- [10] Zwopple Limited. Zwoptex — Zwopple — Creative Software. <https://zwopple.com/zwoptex/>, 2017. [Online; accessed 14-May-2019].
- [11] CEGO ApS. Open Source Software by CEGO ApS - spritemapper. <http://opensource.cego.dk/spritemapper/>, 2013. [Online; accessed 14-May-2019].
- [12] Joe Hall. GDU - Sheets. <http://gamedevutils.com/webapps/sheets/>, 2016. [Online; accessed 14-May-2019].
- [13] Herald Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.
- [14] Gerhard Wäscher nad Heike Haußner and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 83:109–1130, 2007.

- [15] Frank de Zeeuw. Combinatorial Optimization. https://dcg.epfl.ch/wp-content/uploads/2018/10/CombinatorialOptimization_20160218.pdf, 2016. [Online lecture notes; accessed 13-April-2019].
- [16] Jukka Jylänki. A thousand ways to pack the bin – a practical approach to two-dimensional rectangle bin packing, 2010.
- [17] Andrea Lodi. Algorithms for Two-Dimensional Bin Packing and Assignment Problems. http://www.dei.unipd.it/~fisch/ricop/tesi/tesi_dottorato_Lodi_1999.pdf, 1999. [Online; accessed 22-May-2019].
- [18] José Fernando Oliveira, Alvaro Neuenfeldt Júnior, Elsa Silva, and Maria Antónia Carravilla. A Survey On Heuristics For The Two-Dimensional Rectangular Strip Packing Problem. *Pesquisa Operacional*, 36:197–226, 2016. [Online; accessed 22-May-2019].
- [19] Richard E. Korf. Optimal rectangle packing: Initial results. In *In Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-2003)*, pages 287–295, 2003.
- [20] El-Ghazali Talbi. *Metaheuristics From Design To Implementation*. Wiley, 2009.
- [21] Brenda S. Baker, Ed Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9:846–855, 11 1980.
- [22] Bernard Chazelle. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32(8):697–707, 8 1983.
- [23] Dequan Liu and Hongfei Teng. An improved bl-algorithm for genetic algorithm of the orthogonal packing of rectangles. *European Journal of Operational Research*, 112:413–420, 01 1999.
- [24] Wei Lijun, Defu Zhang, and Qingshan Chen. A least wasted first heuristic algorithm for the rectangular packing problem. *Computers & Operations Research*, 36:1608–1614, 05 2009.
- [25] Stefan Jakobs. On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88(1):165 – 181, 1996.
- [26] Immo Landwerth. Demystifying .NET Core and .NET Standard. *MSDN Magazine*, 32(9), 09 2017.
- [27] Attributed programming model overview (mef). <https://docs.microsoft.com/en-us/dotnet/framework/mef/attributed-programming-model-overview-mef>, 2017.
- [28] Wix toolset. <https://wixtoolset.org/>.
- [29] Github - ewsoftware/shfb: Sandcastle help file builder (shfb).a standalone gui, visual studio integration package, and msbuild tasks providing full configuration and extensibility for building help files with the sandcastle tools. <https://github.com/EWSSoftware/SHFB>.

- [30] RelayCommands MVVM Commands and EventToCommand. Demystifying .NET Core and .NET Standard. *MSDN Magazine*, 28(5), 05 2013.
- [31] A new idialogservice for wpf · issue #1666 · prismlibrary/prism · github. <https://github.com/PrismLibrary/Prism/issues/1666>.
- [32] Skiasharp/wpextensions.cs at master · mono/skiasharp · github. <https://github.com/mono/SkiaSharp/blob/master/source/SkiaSharp.Views/SkiaSharp.Views.WPF/WPFExtensions.cs>.
- [33] Prism/prismapplication.cs at master · prismlibrary/prism · github. <https://github.com/PrismLibrary/Prism/blob/master/Source/Wpf/Prism.Unity.Wpf/PrismApplication.cs>.

A. Attachment

The contents of the attachment:

- `/src` – folder containing the whole PaunPacker solution.
- `/samples` – folder containing some of the sample input textures.
- `/documentation` – contains the documentation for the projects from the PaunPacker solution.
- `/setup` – contains an installer and a portable version of the PaunPacker.
- `/thesis.pdf` – file containing this thesis.
- `/README.txt` – file describing the contents of the attachment.

