Irregular Packing for Mobile Game Texture Atlases

Jarkko Pöyry

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.8.2018

Supervisor

Assoc. Prof. Keijo Heljanko

Advisor

M.Sc. (Tech.) Teppo Soininen

Aalto University
School of Science

Copyright © 2018 Ministry of Games Oy

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Jarkko Pöyry

Title Irregular Packing for Mobile Game Texture Atlases

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI304

Supervisor Assoc. Prof. Keijo Heljanko

Advisor M.Sc. (Tech.) Teppo Soininen

Date 21.8.2018 **Number of pages** 65 **Language** English

Abstract

For the efficient realtime rendering on a mobile device and for efficient delivery of a mobile game application, the texture atlases used in a mobile game need to packed as tightly as possible, but at the same time due to various practical limitations, the time required for the packing should be minimized.

The packing of a texture atlas means positioning a set irregular 2D shapes in non-overlapping positions on a minimum number of atlases with limited item rotations and mirrorings being allowed. This task is well-known subclass of Cutting and Packing problems, and we investigate the various approaches for irregular packing in the literature and introduce two new placement algorithms based on the existing methods.

We evaluate 22 combinations of packing algorithm variations for both the achieved packing density and the required compute time when packing production content derived from a medium-sized mobile game. We are able to show that for low compute time budgets, a placement method we introduced is able to outperform the traditional methods.

Keywords irregular packing, texture atlas, realtime graphics, mobile device

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Jarkko Pöyry

Työn nimi Epäsäännöllisten kappaleiden asemointi mobiilipelien pintakarttakokoelmia varten

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI304

Työn valvoja Prof. Keijo Heljanko

Työn ohjaaja DI Teppo Soininen

Päivämäärä 21.8.2018 **Sivumäärä** 65 **Kieli** Englanti

Tiivistelmä

Mobiililaitteella tapahtuvaa suorituskykyistä reaaliaikaista grafiikkalaskentaa varten ja tehokasta mobiilisovellusten jakelua varten on tarpeellista pakata mobiilipelin käyttämät pintakarttakokoelmat mahdollisimman tiiviisti, mutta samalla on kuitenkin käytännön rajoituksista johtuen minimoitava pakkaamiseen käytetty aika.

Tämä pintakarttakokoelmien pakkaaminen tarkoittaa kaksiulotteisten epäsäännöllisten kappaleiden asemointia niin, etteivät ne leikkaa toisiaan mahdollisimman pienelle määrälle pintakarttakokoelmia samalla sallien kappaleiden rajallisen kääntämisen ja peilaamisen. Tämä ongelma kuuluu tunnettuun leikkaus- ja pakkausongelmien alaluokkaan ja tässä diplomityössä tutkitaan monia kirjallisuudessa tunnetuja menetelmiä epäsäännöllisten kappaleiden pakkaamiseen ja esitellään kaksi uutta asemointimenetelmää, jotka pohjautuvat tunnettuihin menetelmiin.

Työssä arvioidaan kokeellisesti 22 eri pakkausmenetelmäyhdistelmän saavuttamaa pakkaustiheuttä ja kuluttamaa laskenta-aikaa pakatessa keskikokoisen mobiilipelin tuotantomateriaalia. Kokeet osoittavat, että matalilla laskenta-aikabudjeteilla esittämämme asemointimenelmä on parempi kuin perinteiset menetelmät.

Avainsanat epäsäännöllisten kappaleiden asemointi, pintakarttakokoelma, reaaliaikainen grafiikka, mobiililaite

Contents

Abstract 3

Abstract (in Finnish) 4

Contents 5

1 Introduction 6

2 Constraints of a mobile game 8
2.1 Rendering constraints . . . 9
2.2 Filtering constraints . . . 11
2.3 Storage constraints . . . 13
2.4 Development constraints . . . 14

3 Cutting and Packing 17
3.1 Cutting and Packing problem types . . . 17
3.2 Geometry representation . . . 18
3.3 Placement algorithms . . . 21
3.3.1 Bottom-Left placement policy . . . 22
3.3.2 Strict Bottom-Left . . . 22
3.3.3 Bottom-Left procedure . . . 25
3.3.4 Two-Step Bottom-Left procedure . . . 28
3.3.5 Bottom-Left Resolving . . . 30
3.3.6 Clustering approaches . . . 30
3.3.7 Mixed-Integer Models . . . 31
3.3.8 Item orientation . . . 32
3.4 Scheduling algorithms . . . 33
3.5 Metaheuristics and local search . . . 36

4 Evaluation of packing methods 38
4.1 Evaluation methodology . . . 38
4.2 Enforcing constraints . . . 41
4.3 Implementation considerations . . . 43
4.4 Evaluation of packing density . . . 48
4.5 Evaluation of packing performance . . . 51

5 Conclusions 60

1 Introduction

A mobile game is a highly-interactive application running on mobile and hence usually a resource constrained device. As such, a successful mobile game should minimize the amount of resources used, and conversely maximize the provided value. One front on this optimization battle is the layouting of image assets of various shape and size into a minimal number of square shaped images. While this problem is a very traditional irregular Cutting and Packing problem, and similar problems have arisen in the metal and fabric-cutting industries [13, 8], we note that the context of mobile game poses new non-trivial constraints for the packing algorithms described later. In this thesis, we investigate and adapt various irregular packing algorithms and then evaluate them using shapes derived from a real medium-sized mobile game.

In the video game applications, it is often required to draw images of various objects, such as animals or buildings, on the device screen. These images are often represented as a rectangle or a polygon, on which an associated 2D image texture containing the image of the desired object is mapped. As the screen updating needs to be fast to enable high-interactivity and a pleasant user experience, and as the mobile devices are power and computationally constrained devices, these images, or so called sprites, need to be drawn with specialized graphics processing hardware called GPU (Graphics Processing Unit). These GPUs and their APIs however have various limitations and characteristics which in turn cause various limitations on the sprite image textures. Additionally, as mobile devices are also constrained on available storage, memory and data transfer over mobile network, new, non-trivial constraints arise for the sprite data representation.

To render a single sprite on the screen, in the widely-available mobile graphics APIs its texture need first to be *bound* and then a draw call for the sprite geometry to be issued. This texture binding conceptually puts the GPU and the API in into a state in which the sprite's texture containing the associated image can be accessed. [29] So, naively, when multiple sprites need to be rendered on the device, in this scheme each and every draw call is to be preceded by a state change. This however is wasteful due to both state change overhead [15] and draw call overhead such as for the necessary validation [29, 26]. The commonly used mitigation for these costs is to try to render multiple items with a single state change and a draw call in a so called *batch* to minimize the relative overhead. A batch needs to share the API state including the bound texture state, the bound texture image must contain the all the images of the rendered sprites in the batch. This bound texture which contains multiple subimage is called a texture atlas.

To create such an texture atlas, as many as possible sprite shapes need to be placed on non-overlapping positions on a larger image. This problem is a traditional Cutting and Packing problem, and since our test shapes irregular polygon meshes, this specific problem class is in the typology defined in [47] known as Irregular 2D Single Bin Size Bin Packing Problem. This problem class typically arises in manufacturing industries such as in garment and ship-building [13], where the high-density cutting layout will reduce material wastage and hence has a direct economic impact. However, the packing required for texture atlases is subject to very different set of constraints

than with these traditional industries and hence the need arises for the evaluation of Cutting and Packing algorithms fit for mobile game texture atlas generation done in this thesis.

The Cutting and Packing problems are generally in the NP-Hard class of problems [10, 44]. As any exhaustive search for any realistically sized problem is prohibitively expensive, various heuristical approaches have been introduced during the many decades the Cutting and Packing problems have been investigated. In this thesis, we both examine various existing methods known in the literature but also define two new packing algorithm variations. We then evaluate these methods subject to the constraints adhering to texture atlas generation and report both obtained packing results and the required compute time. In our evaluation we show that one of the introduced new packing variants is able to produce higher-density layouts with a lower computational cost than some traditional, well-known approaches.

This thesis is structured as follows: First in Section 2 we derive the practical limitations of real-time rendering on a mobile device and describe how they dictate the requirements for the Cutting and Packing problem. Following this in Section we define the variant of this packing problem using the accepted terminology, and then both describe existing approaches to Cutting and Packing and declare our novel packing variations bases on these known approaches. Next in Section 4, we move onto evaluating the selected 22 Cutting and Packing methods and variations and discuss the adaptations required to the methods in order to satisfy the inherent requirement of texture atlas generation we declared previously. We then evaluate these adapted Cutting and Packing methods with real game content and analyze both the obtained packing quality and the required computation time. Finally in Section 5 we lay out the conclusions.

2 Constraints of a mobile game

A mobile game is an interactive application for a mobile device, such as a phone or a tablet, that is often running in a resource constrained environment. In this section, we explain the various constraints imposed on a game running on multiple mobile platforms, and how they ultimately induce the Cutting and Packing problem variant described in this paper. All in all, in the typology introduced in [47] this can be interpreted as a more constrained case of Irregular 2D Single Bin Size Bin Packing Problem and has the following characteristics of the packing problem:

- Input shapes, i.e. the texture atlases into which items are packed into, are all identical squares.
- The number of texture atlases is to be minimized.
- The output shapes, i.e. the items to be packed, are irregular simple polygons with a large portion being convex. They have no holes, are mostly convex, and a large subset is almost rectangular in shape.
- Items may be may be mirrored both horizontally and vertically, and rotated in 90 degree increments. This also known as the symmetry group D_4 [38].
- Not only must the placed items not overlap, each placed item must have a controllable buffer region around it onto which no other item content is placed.
- The packing algorithm implementation should complete in the minimal possible wall-clock time.

In addition to these hard requirements, we also would packing to also optimize for and adhere to the following objectives:

- Items sharing a certain externally defined property should be packed onto the same input shape, i.e. texture atlas page, if possible. In this paper, we call this property the *affinity* of an item. This affinity is only used for converging and for example two items with different or no affinities may still be placed on the same page.
- The packing algorithm implementation results should be deterministic, and it would be desirable to have a method that allows for insertion and removal of small number of items while only causing a small amount of local changes.

This section is organized as follows. First, we describe the requirements for efficient hardware-accelerated rendering and how satisfying its requirements yields the requirements of fixed power-of-two sized squares for the input shapes and the benefits of the controllable item affinity. Then we justify the need for controllable margins, following which we discuss the various facets of storage requirements and how they induce the minimization and the consistency requirements. Finally, we define the remaining practical and project-specific characteristics stemming from the project requirements.

2.1 Rendering constraints

Like stated earlier, for efficiency all graphics on the application window must be rendered using the dedicated graphics acceleration hardware on the mobile device. In practice, rather than instructing the hardware directly with manufacturer and GPU model specific methods and semantics, this is done by using a hardware abstracting API. Multiple such APIs exist with varying feature sets and supported platforms. For example, on desktop computers commonly used API include Glide [1], OpenGL [31], Direct3D [36], Mantle [2] and its derivative Vulkan [20]. On mobile platforms the APIs available are varying versions of OpenGL ES [29, 30] and Vulkan on Android devices, varying versions of OpenGL ES and Metal [4] on iOS and Direct3D on Windows Phone platforms.

As we our target application only targets Android and iOS devices, we have chosen to use the OpenGL ES family of APIs. In practice, support on real mobile devices in the field running OpenGL ES can be split into two categories: devices supporting only OpenGL ES 2.0 and devices supporting OpenGL ES 3.0 or a later API compatible version. Even though the ES 2.0 is rather old, according to statistics gathered by Unity Technologies, 33.5% of mobile devices still support only OpenGL ES 2.0 [45], and according to Google, the number is 36.4% [41]. It is important to note that the values gathered by Unity Technologies, known for their Unity game engine, can be assumed to give a good representation of the actual games-oriented market rather than the raw numbers on active devices. Since OpenGL ES 2.0 is still significantly represented and since OpenGL ES 2.0 feature set is with a few exceptions a subset of the feature set of OpenGL ES 3.0, we have chosen a hybrid approach where we primarily target OpenGL ES 3.0 but seamlessly fall back to OpenGL ES 2.0 if support is not available. However, since rendering must work in either case with a reasonable performance, for the rest of the paper we will assume OpenGL ES 2.0 level hardware and the OpenGL ES 2.0 API limitations unless otherwise stated.

Rendering an image on the application screen using OpenGL ES 2.0 requires multiple steps. First, the application must upload the image to the graphics driver managed memory (known as the server in OpenGL terminology) creating a graphics hardware samplable image known as a texture. In the simplest case, the uploading is done once in the application startup. This texture next needs to be bound, which puts the conceptual OpenGL state machine into a state in which the selected texture is active, i.e. a shader program can sample from it. Sampling means fetching texture pixels, or texels, around the specified sample position and applying a filtering function to these values. Then finally a primitive covering the destination area needs to be rendered with a shader program that actually samples from the bound texture.

Like described earlier, since due to non-zero state change overhead [15] and draw call overhead such as for the mandatory validation [29, 26], it is not feasible to render even a medium number of images one-by-one. The commonly used mitigation for these costs is to try to render these image in larger batches, i.e. in groups of items that share the same API state, and hence pack multiple images into the same atlas texture. Unfortunately in many cases, images cannot be placed on a single atlas texture and instead multiple atlases need to be used. As our goal is to minimize

the rendering costs, it its clear that the number of state changes and draw calls need to be minimized. By packing the images into the smallest amount of atlases, the number of state changes is minimized in the average scene of multiple different sprites. Hence, our goal is to minimize the number of the input shapes, i.e. the number of atlases required.

However, this reasoning only holds in the average case where items on the scene are randomly selected. In the actual target city building game, certain images or sprites may have very high probability of appearing at the same time, and hence packing those images on the same atlas may yield significant advantages in the controlled scene as compared to unguided packing. For example, a dense foliage surrounding a city composed of various tree and grass sprites could be rendered with just a single state change and render call if all foliage images can be guided to be packed on the same image atlas. Since this is is very important optimization for selected important scenes, and as these relationships are heavily content specific, we require the packing algorithm to allow for these external item-specific constraints. We denote this artist controlled property the affinity of an item. If two items share an affinity, they should be placed on the same atlas. But since there are no downsides in having additional non-affinity-sharing items on a single atlas, two items not sharing an affinity are still allowed to placed on the same atlas.

An alternative approach to avoid API state changed between two sprites from two different atlases is to craft a shader program that can sample from multiple atlas textures within the same draw call invocation. Unfortunately, OpenGL ES 2.0 does not support dynamically selecting texture sampler unit at shader runtime [28], conditional texture sampling might still cost as much as unconditional sampling, and unconditionally sampling all atlases would be extremely expensive. However on OpenGL ES 3.0, all atlas pages could be placed into a structure called texture array which allows for efficient access across layers. We will not describe the feature further other than state that it requires levels, i.e. all atlases, to have the same dimensions. As this is a very important optimization for the majority of the devices on the market, we also require this for packing algorithm.

As the textures are created from our atlas images, it is clear that any limitations is directly mirrored to the atlas image construction. Most obviously the texture shape and dimensions are constrained. The OpenGL ES 2.0 specification does not require a conforming implementation to support arbitrary large textures and allows each implementation to specify the value for maximal supported image size dimension. While the minimum required maximum value is specified to be just 64 meaning that only textures no larger than 64×64 are guaranteed to not be unsupported due to their size [29], in practice via experimentation we have noticed that texture dimensions up to 2048 texels-per-side are generally available. Hence our packing input image size should be at maximum 2048 units high and wide.

In addition to the generic texture size API limitations, different texture formats also have different runtime performance. Texture formats are data encodings from the image data bytestream into the image pixel values. For example simple format such as RGBA8 describes a 4-channel image format where each component is 8 bits long and RGB565 a 3-channel format with R, G, and B components of length 5,

6, and 5 bits, but also more complex so called compressed texture formats exist. Compressed formats are lossy encodings from original texture data to an internal representation which when decoded usually perceptually resembles the original image. Different compression formats have different feature sets, encoding rates, encoding complexities, supported platforms, and different quality with different content. Note that encoding is usually done offline and hence encoding complexity does not pose any constraints on rendering. Like the name implies, compressed formats reduce the amount of texture data required to represent an image, and for example Ericsson Texture Compression versions 1 (ETC1) and 2 (ETC2) reach 4 bits per pixel encoding RGB8 [27, 30] and PowerVR Texture Compression (PVRTC) reaches 4 or 2 bits per pixel encoding RGBA8 depending on the desired quality [18]. Reducing the texture data size is essential as many mobile devices are bound by texture bandwidth [15] and it additionally reduces the amount of data needed to be copied during texture upload improving loading time.

With the improved performance of the compressed formats comes additional limitations. While we only concentrate on ETC1 and ETC2 on Android devices and PVRTC on iOS devices, we take the combination of their limitation to keep the asset pipeline platform agnostic. ETC1 and ETC2 are block based fixed-rate encodings that do no impose any additional limitations on texture shape [27, 30]. However since we pay by block instead of pixels, aligning the texture size to block size is beneficial. PVRTC instead has very strict limitations and requires texture dimensions to be powers-of-two [22]. However, in practice the limitation is more severe since by experimentation it can be observed that iOS devices additionally require the image to be square. As we only use PVRTC on iOS platforms, we consider this a limitation of the texture format and require it to be satisfied in the packing.

2.2 Filtering constraints

As stated above, the actual rendering of the sprite image to the application window happens by issuing a render call for a set of primitives, usually a triangles, that cover the sprite area completely. These primitives are rasterized and for the resulting application window pixels, the fragment shader of the current shader program is executed. This fragment shader then samples the sprite atlas texture by offsetting the default texture sampling coordinates by the sprite's location in the texture atlas. This results in sampling results and hence the final image are as if we had been sampling an independent texture containing only the sprite image. But since the sampling positions are fully controllable, not only can we offset but we could apply any transformation to the sampling coordinates. However, not all transformations are admissible and would result in the same final image. Since sampling results depend on the local 2×2 texel neighborhood of the sampling position [29], the sampling results can only be uphold in the general case by making sure all 2×2 neighborhoods can be represented exactly in the transformed texture. This leaves us with the combination of moving the sprite to an offset that is a multiple of the texel size and the symmetry group of the 2×2 texel block, i.e. 90 degree rotations and axis aligned mirroring.

While these transformations do maintain the neighborhoods within the sprite shape, we note an issue near the primitive edges. When sampling at a position near the primitive edges, the 2×2 sampling footprint actually exceeds the sprite item bounds. As the packing only guarantees that items and hence their contents do not overlap in the atlas, by sampling at an item edge, content from another sprite in the atlas may still fall into the filtering neighborhood, and hence would affect the sampling result. In this case, the rendering result would exhibit a color bleeding artifacts near the sprite edges. As this is not acceptable, it is necessary to avoid placing items such that a sample filter area overlaps multiple objects. As our sample positions are naturally always within the item itself, we only need to guarantee that no other item is overlapping the Minkowski sum [44] of the original item area and the sample filter area. We call this area extension the buffer of an item.

Unfortunately, while the buffer area created for 2×2 filter is sufficient for sampling the original image, we will still have the same color bleeding issue in when sampling with mipmapping. Mipmaps are precomputed downscaled versions of the original texture which are used in downscaling texture sampling [29]. They form a pyramid with the original at the top and each subsequent level being half the resolution in each dimension than the above, up to the peak of 1×1 miplevel. Downscaling in this case means that the sprite on the render target framebuffer has smaller resolution than the original sprite asset. As the filter region is fixed 2×2 texels, by making the rendered sprite smaller and smaller, the texel area a single rasterized pixel covers on the sprite texture eventually becomes larger than this filter area. As the filter area does not cover all the texels represented by the pixel, the sampling quality suffers. For example in the extreme case, whole sprite is minified onto a one single pixel on the window. Ideally, the resulting sample color should depend on all the texels of the sprite as they all are represented by that pixel and hence should be able to contribute to the result. But since the filter area is always the 2×2 texels, this is clearly not the case and the pixel color would be determined by a single 2×2 neighborhood somewhere in the sprite. By providing mipmaps we allow texture sampler to sample from an already downscaled lower resolution texture miplevel that has similar resolution to the output image. This way the sampled texels of 2×2 neighborhood may not necessarily be those of the original image but rather precomputed results of filtering with a larger radius. All in all, we imitate a larger filter by separating into an arbitrary precomputed filter and the 2×2 runtime sampling.

Since we essentially downscale an already downscaled image, we now have two sources sampling of error that can cause color bleeding. When creating the precomputed downscaled miplevels if we were to naively just downscale the primary image, the resulting texels could naturally cover over multiple packed items resulting in a mixture of colors from different assets, i.e. a color bleed. This can avoided by carefully controlling the filtering taps to avoid them falling on multiple items or by fixing a filter-wide color boundary on item edges, and hence we don't see color bleeding as an issue within the precomputed miplevels.

However even still the color bleeding is inevitable. Even if we did avoid color bleeding in within any single mipmap texel as explained above, since we do 2×2

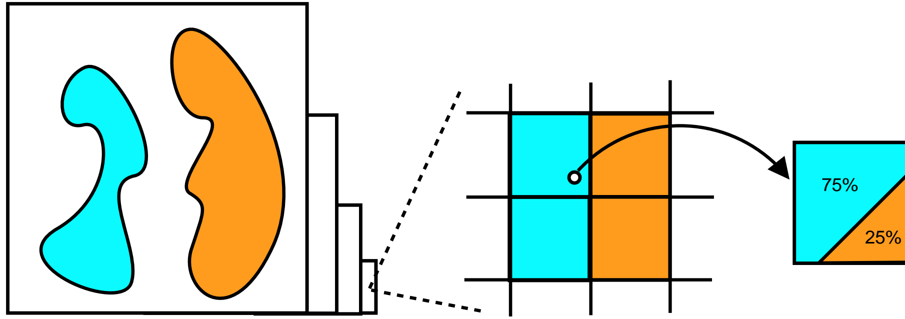


Figure 1: Color bleeding due to sampling from a mipmap.

sampling from the mipmap, neighboring miptexels might contain color from different atlas items and hence the their color bleed in to neighboring items as pictured in Figure 1. On the base miplevel, we worked around this with the filter sized buffer region around atlas items. To do this on any other miplevel, would need to do the same thing on the downscaled image, i.e. increase the filter size proportionally to the downscaling factor. Clearly this cannot be satisfied for all miplevels, as in the topmost mipmap size is 1×1 texels, and hence the effective filter is larger than the original texture. Instead we may select up to a certain miplevel to which color bleeding is eliminated, with each additional level doubling the buffer area width.

As the miplevel selection depends on asset image resolution and sprite resolution on the render target, some sprites are more likely to hit lower levels than others. Additionally, some sprites may only be visible for a moment while some sprites are shown for a long time increasing likelihood that the user is affected by the artifact. Hence, color bleeding prevention is less important for some assets than others, and optimize for this we desire to configure the buffer area width sprite by sprite basis.

2.3 Storage constraints

The most obvious limitation on the application deliverable is the application package size during the acquisition and after it has been installed on the device. As mobile devices are resource-constrained devices also with respect to the size of the persistent storage, it is important to minimize the negative impact to the user for having the application installed. Similarly, as mobile applications are usually acquired as downloads over a network connection, the application install package size directly defines the delay before the application is usable by the user. Clearly to minimize user discomfort, this time and hence the size of the installation package should be minimized as well.

While the exact contribution of these texture atlases to the total installed application size varies from game to game, in our target application the contribution is very high. As an example, in the current version of the target game for android the encoded texture atlas files account for around 67 % of the total application size. This underlines the dire need for efficient atlas packing.

As described in the previous section, our target application uses either ETC1, ETC2 or PVRTC texture compression for the atlas pages as a necessity. These

texture formats are block based fixed rate encodings [27, 30, 18] causing the encoded bitstream length in bytes be always directly proportional to the number of pixel-blocks in the input image. Since the pixel-block sizes are also fixed to 4×4 in all three formats with the quality level we have chosen, this simplifies our goal considerably. Rather than having to minimize total size of the bitstreams after a complex encoding operation, we can simply minimize the total area of the atlases.

The resulting atlas image bitstreams could be losslessy encoded further using various general purpose compression algorithms such as Zstandard [17], and in our internal tests we have seen some gains with such methods. However, in this paper we do not consider the effects of these transforms as they have proven to be very unpredictable and complex optimization targets.

In addition to these platform-agnostic storage constraints, various mobile ecosystems have their specific set of challenges. Mobile applications are usually delivered using a platform-specific delivery mechanisms, such as Google Play Store on Android and App Store on iOS. These delivery platforms are usually controlled the platform owner and they impose the application developers various constraints on the delivered application package. For example on Google Play Store, the application package can by default be only 100 MB in size after which a complex multi-part packaging and downloading scheme need to be utilized [42]. Since we want to avoid the need for such complexity and resulting user discomfort, minimizing the application package size is important.

However, in addition to the plain size, more complex platform-specific storage characteristics exist. When a mobile application is updated, the whole new version of the application deliverable is not downloaded on the device but instead just a smaller delta-encoded patch file is received and applied over the old deliverable version [39]. As we keep developing the game, we often need to add and modify already shipped image assets that are packed into atlases. Clearly if an atlas contents change between the versions, the update must contain the delta between the bitstreams of these atlases. Since we employ various complex bitstream compression methods, it can be assumed that any local change in the atlas contents will cause large differences in the bitstream, and hence the delta encoding does not benefit us much with a modified atlas. However, all unmodified atlases are free. In these cases, it would be beneficial to have the packing algorithm to allow for these incremental modifications that only cause changes within the atlases the changed items reside in, as this would minimize the update size.

2.4 Development constraints

In addition to the user discomfort, the developer discomfort needs to be addressed as well. In this section we first describe our asset pipeline and how it affects the cutting and packing problem at hand. We then continue with describing our development process and how it induces additional requirements and finally we talk about the characteristic of the compute environment we use and its implications.

In our target game, the sprite assets are generated by pre-rendering high-complexity, stylized 3D-models with a semi-realistic lighting model. The renderer

used is a commercial path-tracing software used widely in the industry. To make them suitable for realtime rendering, these generated sprite assets are then cut into an efficient triangle mesh representation that minimizes fully transparent section using a custom tool. As this custom tool uses a pseudo-randomized search, it is very chaotic and unstable by nature and even the smallest change in the source asset can change the resulting mesh significantly. But the path-tracing is very time consuming and especially at the very large per-pixel sample counts it becomes prohibitively expensive for our time budget. Instead, we render the assets with a moderate sample count which is slow but still tolerable. However, by lowering the sample count, we introduce low amplitude noise to the resulting images which while invisible to a human eye evokes the triangle mesh generating tool. As the tool is fed noisy input data, even very similar images, such as for adjacent keyframes in an animation, a different triangle mesh will be generated. Hence due to this interaction, our input content becomes heterogeneous: while the generated meshes are similar, each mesh to be packed is unique.

Whether these heterogeneous shapes classify as strongly or weakly heterogeneous shapes is unclear: on one hand all shapes are unique which is a good indicator of strong heterogeneity. On the other hand, relatively large groups of items share similar, but not exactly identical shape, which suggests weak heterogeneity. In any case, this classification has no real world implications and so we do not make any distinction between the cases.

Our development process is based on the practice of Continuous Integration for both the code and the art assets. All committed changes are integrated into our master branch and then both the application and the associated texture atlases are rebuilt for our all our target mobile platforms. The resulting texture atlases are then encoded with the target texture codec and the resulting code and asset bundle are then built into a mobile application package which is pushed into a third party cloud service. The developers can then pull the latest builds from this service to their mobile devices for validation. Additionally for more rapid and experimental changes, a local build path exists which with a similar build pipeline allows generating test builds for both mobile devices but also for Windows and Linux-based desktop operating systems.

For rapid iteration and to maintain the developer focus it is important that this modification-to-feedback process is fast. As the ability to validate and react to changes is essentially bound by the speed of this build process, it is imporant to minimize the wall clock time consumed for both texture atlas compilation and the texture encoding. We approach this problem from two fronts: first we aim to minimize the time used the irregular packing needed for the atlas generation, and hence declare it a parallel objective in addition to the minimization of the number of atlases. Second, we attempt to avoid the highly compute intensive texture encoding step when possible by memoizing the encoded results. At the beginning of the texture encoding step, we first hash the input contents of the texture atlas and then perform a lookup to a globally shared, cloud-backed result cache. If a result is not found in the cache, the texture is encoded and the result is pushed to the cache for potential future use. For this caching scheme to work effectively, it is critical for the packing

method to maintain stable results. Hence, we require the packing method to be deterministic.

Finally, as explained above we are interested on minimizing the consumed wall clock time mainly to provide a better development experience. However, there is also an alternative reason: as the packing process is computed on a third party cloud based pay-per-use compute service, each consumed compute second has a direct monetary cost to the company. Hence by minimizing the compute time, we also minimize our costs. In this rented compute environment, our packing process is exposed with 8 virtual CPU cores and so any opportunities on parallellizing the texture atlas packing task are very interesting to us, albeit not the focus of this paper. Furthermore, as there are no GPUs available in this compute service, we do not pursue any GPU based packing solutions.

3 Cutting and Packing

Cutting and Packing problems are very well-studied class of combinatorial problems, that traditionally arise in manufacturing industry [13]. Since Cutting and Packing problems are generally NP-Hard [10, 44], the general focus of the research has been in development and in evaluation of various heuristical approaches to approximating the solution [6]. As the problem class has been studied for long periods of time, a large number of variants and subclasses have been identified. In this section, we first identify the problem types most relevant for our use case. We then take a more practical step, and investigate various approaches to representing geometry in Cutting and Packing literature. Finally, as methods for multiple bin packing have been traditionally seen as a combination of three rather independent components, we discuss these approaches to each step one by one: first we investigate various methods for placing items within a single bin, then we examine various approaches of selecting which item goes into which bin, and finally we move onto describing the commonly used local search schemes.

3.1 Cutting and Packing problem types

As we described above the Cutting and Packing problems cover a vast set of different problem variants. While these problem classes generally share similar problem structure usually consisting of combination of combinatorial problems, the approaches and the inherent complexities can vary significantly. For example, while 1D bin packing problem is already a hard problem, the 2D variant is considerably harder. And in particular, the packing problems of irregular pieces are considered to comprise the most difficult class of packing problems [13]. In this section we discuss various problem type known in the literature that are relevant to our use case. Notably, we will not be using the traditional terminology emerging from various industries [13] but instead favor the modern, more explicit terminology.

As described earlier in the Section 2, our task is to position a set of irregular 2-dimensional shapes into a fixed equal sized squares while minimizing the number of the squares required with respect some additional soft and hard constraints. Using the typology proposed in [47], this problem can be classified either as Irregular 2D Single Stock Size Cutting Stock Problem (Irregular 2D SSSCSP) or as Irregular 2D Single Bin Size Bin Packing Problem (Irregular 2D SBSBPP) depending on whether the placed items are considered to be weakly or strongly heterogeneous. While this distinction was discussed previously in Section 2 and arguments were proposed for both stances, in practice as we later go on to the descriptions of various relevant algorithms, we will notice that this distinction has no practical relevance for our target workload. Hence, for the rest of the paper we will use problem classes SSSCSP and SBSBPP interchangeably.

The SBSBPP (and hence SSSCSP) is a combination of two separate combinatorial optimization problems: the problem of allocating (or scheduling) packed items into the bins and the placing problem of positioning items within a bin in non-overlapping legal positions [34]. The objective is commonly to minimize the number of bins

or stock sheets required while cutting a desired number of items. We discuss the approaches to both components in Sections 3.4 and 3.3, respectively.

In addition to the SSSCSP and SBSBPP problem types, we will also be discussing so called 2D Irregular Open Dimension Problems or more specifically various 2D Irregular Strip Packing algorithms. The Irregular Strip Packing Problem is an optimization problem in which a set of irregular shapes need to placed on a rectangle that has a fixed height (width) such that the required width (height) of the rectangle is minimized. This roughly corresponds to manufacturing pieces by cutting a coil of metal in metalworking industries. By minimizing the open dimension, we minimize the cutting pattern length and hence minimize the material costs.

While Strip Packing per se is orthogonal to our goals as we need multiple bin layouts and cannot utilize non-square packing results as discussed in Section 2, the Strip Packing algorithms are still interesting to our goal two reasons. First, as mentioned earlier the strip packing problems have emerged in many manufacturing industries and have been studied for decades. As such large quantities of insights, research, and test cases have been developed for the problem, and some of it can be ported to the branch relevant to our interests.

The second reason is that by its very nature, the strip packing is similar to the placing problem component of an SBSBPP problem. By controlling the set of items to be packed on a strip, we may select such a set that the length of the resulting strip is at maximum our desired bin size. Hence in various cases by choosing the strip width to be our bin size, we can reuse or adapt strip packing algorithms into solving SBSBPP problems by simply slapping on a suitable bin allocator strategy or adding more constraints to the placement process. We discuss this approach in more detail in Section 3.3.

3.2 Geometry representation

As the irregular two-dimensional cutting and packing problems are essentially optimization tasks on placing non-regular geometric shapes into other geometric shapes in non-overlapping positions, the representation of the shapes in the packing algorithms are often fundamental to the algorithm, and it has an impact on algorithm run time, packing quality and even feature set. In this section we describe and discuss various representations of the geometric shapes in cutting and packing literature and describe how common Cutting and Packing operations can be implemented using them.

The two common techniques to represent an arbitrary shape are to either represent it using various geometric primitives, such as polygons, or to rasterize it on a uniform grid and use the resulting image to represent the shape [44]. With the rasterization technique, the choice of the grid resolution is not insignificant as it represents trade-off between accuracy and the compute time [8]. A higher resolution can represent a shape more accurately but it will also increase the storage and compute requirements. However, in our peculiar case as we require pixel-fitting for the resulting shapes as discussed in Section 2, by choosing the grid resolution to match the texel resolution, all shapes are represented exactly.

For the raster technique, there exists multiple approaches to storing the bitmap

data, each with different complexities and feature sets. A straightforward approach is to store the bitmap as pixel matrix. While convenient and able to represent any shape, it is also very expensive in compute time in various common operations. For example the cost of an overlap test is linear to the surface area of the axis-aligned bounding rectangle of shape which in turn quadric to the grid resolution. An another straightforward approach is to store only the coordinates of the occupied pixels as a bag of fragments. This avoids having to pay for the unused pixels in the bounding rectangle but the number of fragments is still quadric to the grid resolution.

This quadric behavior is clearly not desirable and in many cases we can do better. In the case the shape is convex, we can inspect on occupied pixels on each row, and due to convexity, on each such scanline only a contiguous set of pixels may be occupied. By storing these occupied spans, we have a bag of spans representing the shape. This construct grows only linearly to the grid resolution and also has a linear time overlap detection. A similar approach can also be used even if the shape is not convex by first finding the convex components of the shape and then applying the strategy. While the number of the convex components is in the worst case quadric to the resolution, the actual number will naturally depend on the characteristics of the shapes. For the well behaving shapes, such as with our content, the number can be expected to be a lot less. In addition to these approaches, there are also other methods that are quadric in the worst case but may not be in practice. For example, by storing a signed distance field in the bitmap matrix instead of simple occupancy, the runtime cost of the overlap test can be significantly reduced. However, we consider this merely an implementation optimization and we do not pursue this topic further.

Similar to the raster techniques, there are many approaches to storing and processing geometric primitives. If the shapes are polygons or sets of polygons, a straightforward approach is store the edge line segments and use them for the geometric operations. We call this the direct geometry approach. This approach is very convenient and it allows for a precise representation of all polygonal shapes, both convex and concave, with holes and even shapes with disjoint components. Any curves or circular edges however can only be approximately represented by converting the edge to piecewise-linear edges covering the original shape with the desired precision [8]. The downside to this generality is the cost. An overlap detection for two sets of edges consists of an edge intersection test and a final test if one shape is completely inside the other. This edge intersection test for shapes of n and m edges takes $\mathcal{O}(nm)$ time [44].

Like with the raster approaches, we can do better in many more constrained cases. One such important constraint is again the convexity of the shapes. In case the shapes are convex, the edge intersection test can be done for shapes of n and m edges $\mathcal{O}(n + m)$ time [44]. And similarly to the raster case, this approach can be adapted to non-convex shape by simply finding the convex components and representing the shape as their set. Yet again this does not on a theoretical level solve the worst case complexity, but in practice with realistic shapes the performance might be sufficient.

In addition to the raw representation format of the shapes, the various constructs on these representations are also important. One such construct is the No-Fit Polygon

(NFP) originally devised in [5] with the name Envelope. The Envelope or NFP is essentially is Minkowski Sum of a shape and a complement of another shape [37] and it represents the set of positions a for shape can such that is overlaps another shape. Hence, the positions on the edge of the NFP represent positions in which the two positions touch and the positions outside the NFP positions in which the two shapes are separated. For an example, see Figure 2. We select the yellow point as the reference point of the shape B and it represents the location of the shape. Now in the No-Fit Polygon of A and B, we observe that all relative positions in which the shapes A and B would overlap, the reference point marked with yellow would also overlap the generated NFP.

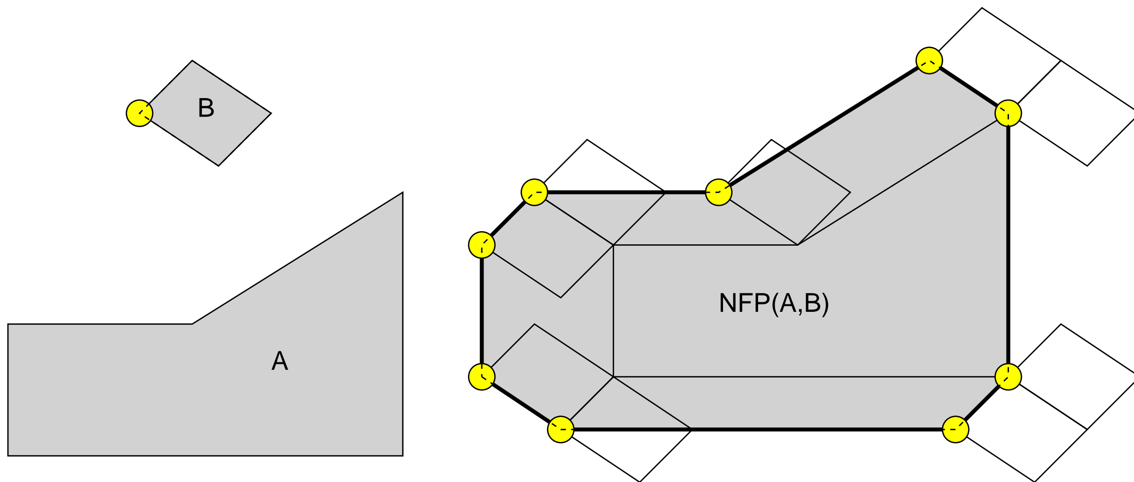


Figure 2: The No-Fit Polygon of the shapes A and B. Yellow point represents the chosen reference point of B.

The value of the NFP is that it transforms a complex polygon-polygon intersection test into a relatively simple point-in-polygon test, albeit with a more complex polygon [9]. While in practice with well-behaving shapes the resulting polygon in simple enough for the transform to be beneficial, with complex shapes the NFP can grow very large. In the case of noncovex polygons of n and m edges, the resulting NFP can be of $\mathcal{O}(n^2m^2)$ in size [44]. In addition to the size of NFP, also the computational and implementation complexity for generating a NFP becomes high for complex shapes. While NFP can be solved for convex polygons by simply tracing their edges while sliding one shape against another [44], for arbitrary shapes it is significantly more complex and error-prone as it can for example result in both holes with surface area and degenerate point and line-shaped holes. For a comprehensive study for the challenges and approaches for NFP generation, see [9].

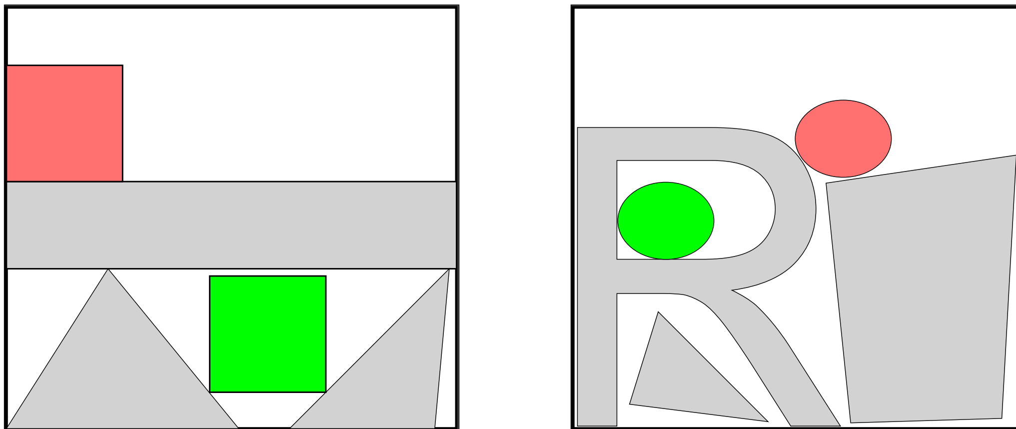
In addition to the NFP, another similar construct is the Inner Fit Polygon (IFP). In bin packing, the Inner Fit Polygon is essentially complement of the NFP of the bin boundaries and a shape and hence it will similarly represent the positions in which the shape does not overlap the bin edges. And similarly, the positions on the IFP represents positions in which the shape would merely touch the bin boundary. Since the bins are commonly rectangular, the valid space inside the bin is also rectangular

and hence IFP can be represented as a rectangle.

3.3 Placement algorithms

The problem of placing shapes in non-overlapping positions is the foundational problem in Cutting and Packing problems and many approaches to it have been attempted. While the problem scope is large, we in this paper will mainly concentrate on methods suitable for multiple bin packing. As such, our focus on the placement methods is on constructive and what we call incremental methods. This section is structured as follows: first we define our terminology to allow meaningful discussion on the presented methods and their abilities. Then we move on the BL-class of placement methods and describe its many variations. Next we investigate recent integer programming models developed for item placement and finally shortly declare the various approaches to handling the free and limited item rotation within the placement process.

In the search of a valid Cutting and Packing problem solution, the placement is commonly required to be solved for very similar shape sets. For example, in many scheduling approaches described later, items are placed one by one into the bins and the layout is validated after each step. So clearly, if the placement algorithm is able use a valid partial solution for a subset of input items computed earlier, a lot of computational effort can be saved. We call these algorithms constructive. We also define an even stronger requirement for so called incremental algorithms. Instead of merely being able to reuse a previous partial result in an unspecified way, an incremental algorithm must not alter the previously placed items and can only add new items in new, unused areas. Hence a solution for the input sequence A, B, and C must contain exactly the partial solution computed for A and B. With this definition, an incremental algorithm can be also be interpreted as an online algorithm with no state.



(a) Emergent hole caused by multiple shapes.　　(b) Explicit hole in a shape.

Figure 3: Hole-filling in placement. An example hole-filling placement is marked with green and example non-hole-filling position with red.

In addition to these architectural properties, more practical properties exist. The

placement algorithms are generally separated into two categories based on whether they still can or they no longer can perform so called hole-filling [14]. In this case the hole-filling means ability to select a position that is obscured by other placed items around the position such that the set of valid candidate positions is disjoint. This can be interpreted as that a no-hole-filling algorithm only maintains a single connected set of valid positions or a placing front such that after placing a few items, the space “behind” them can become inaccessible. As an example, see Figure 3(a). In the case the placement algorithm was to place the square to the bottommost position, a hole-filling-capable algorithm could choose the position marked with green. But if the algorithm cannot perform the hole-filling, it would choose the red position. In addition to filling these emergent, multi-item holes, the hole-filling can also mean the ability to the fill holes in the shapes themselves [8], illustrated in Figure 3(b). As our content does not contain holes, we do not pursue this further.

Finally, as noted previously in Section 3.1 the placement problem in some Cutting and Packing problem classes can be very similar. As such in this section we also discuss placement methods originally designed not only for the multiple bin packing problem but also the strip packing problem. But as these problems are not exactly identical since to strip packing placement will never fail due to ability to always allocate more strip, we adapt the methods as necessary. As these adaptations are rather trivial and non-intrusive, their descriptions are omitted.

3.3.1 Bottom-Left placement policy

A popular approach for solving the placement problem is the using so called bottom-left placement policy [14]. As defined in [6], the bottom-left placement is an incremental placement algorithm that selects items to be positioned from a list one by one and then places each item to the bottommost valid position. In case there are multiple valid bottommost positions, the left-most valid position is selected. While simple and shown a reasonable effective heuristic, no method is actually given for finding such a position in the originating paper. Presumably for this reason, multiple different algorithms sharing the name or parts of it have been proposed in the literature, including some that do not exactly satisfy the bottom-left policy but merely approximate it with varying methods and varying caveats. In this paper, we call this class of approaches the BL-class methods.

3.3.2 Strict Bottom-Left

As a baseline, we first define an algorithm class that exactly satisfies the bottom-left placement policy. In this paper, we call this class Strict Bottom-Left algorithms (Strict BL). While we below define the Strict BL only for models using rasterized representation and for two NFP variants, there is presumably no reason why they couldn’t be constructed on other geometry representations as well. However, these other representations have been left out as their implementation complexity can be significantly larger than the expected benefit gained from the result. In particular, finding the satisfying position on direct geometry edge list representation can essen-

tially be reduced into the task resembling the generation of an NFP using the orbital method with the insert shape and the union of the placed shapes.

```
input  : R_W,R_H the bitmap resolution
input  : placed the list of placed shapes
input  : inserted the shape to be inserted. A list of (x, y) fragment
         coordinates in the range of x >= 0 and y >= 0.
output : Valid position satisfying BL-placement policy for insert shape or
         error if item cannot be placed
// mark occupied cells in cells with 1
initialize cells [0..R_W, 0..R_H] to 0;
foreach shape s of placed do
    foreach fragment x, y of s do cells [x, y] ← 1;
end
// search for a satisfying position, bottom-up, left-to-right
for y_s ← 0 to R_H − 1 do
    for x_s ← 0 to R_W − 1 do
        valid ← 1;
        foreach fragment x, y of inserted do
            if (x + x_s, y + y_s) not in bin area or cells [x + x_s, y + y_s] = 1 then
             valid ← 0;
        end
        if valid = 1 then return (x_s, y_s);
    end
end
return error
```

Algorithm 1: Strict BL Placement using bag-of-fragments representation. See Figure 4 for an illustration.

Strict Bottom Left using rasterized bitmap representation is essentially a very simple search in bottom-up, left-to-right order over all possible positions on the grid and selecting the first fitting position. As it strictly implements the bottom-left placement policy, it is by definition an incremental placement algorithm. A naive example implementation is shown in Algorithm 1 and illustrated in Figure 4. In addition to being incremental for the resulting positions, the Strict Bottom Left on raster map can also be trivially implemented to be incremental also in the bin state by reusing the grid cell state of the placed shapes from an earlier run. In this example implementation we have chosen to represent the shapes as bags of fragments, but implementations using other raster representations discussed in Section 3.2 could also be possible with minimal work. While these other representations would likely exhibit better runtime performance they are slightly more verbose, and as their implementations are close to trivial we have chosen to omit their descriptions here.

The Strict Bottom Left using the NFP representation has a significantly more complex implementation but its theory is also very simple. For completeness, we describe the method for both global and per-item NFP which were defined in Sec-

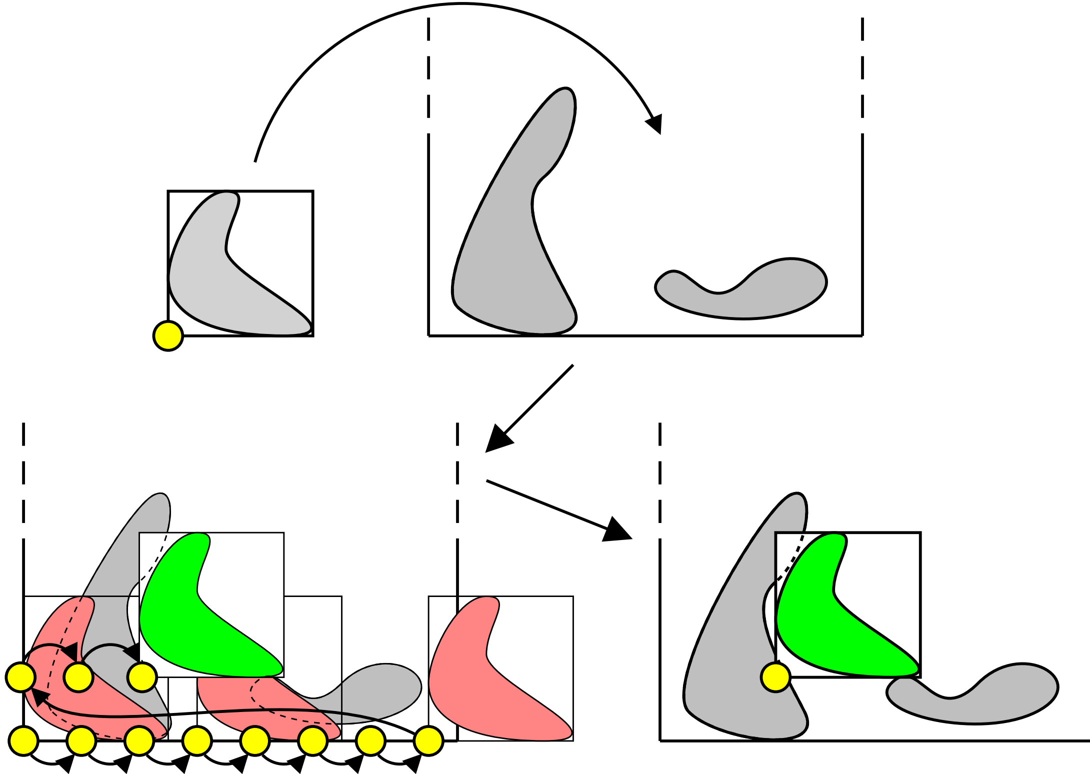


Figure 4: Finding the BL-position for an inserted shape (top left) in a partially filled bin (top right) with an exhaustive search, illustrating Algorithm 1. The potential positions are validated and rejected in BL order until a valid position, green, is found. The yellow points represent the reference point of the inserted shape and the tested potential positions.

tion 3.2. For the global NFP, the algorithm is very simple. The generated complete NFP of the placed shapes for the inserted shape represents exactly the positions in which the inserted shape would overlap a placed shape. Hence, the BL-satisfying position must reside on the NFP edge. For a proof, consider a case in which we have a BL-satisfying position outside NFP that is not on the NFP edge. Since it is not on an edge, we can move it downwards by ϵ while still staying outside the NFP. Since this new position is valid and at a lower position than the original position, the original position cannot be BL-satisfying contradicting our assumption. Since similar reasoning can be used for the IFP of the bin, it can be trivially shown that the BL-satisfying position must be a vertex of the union of the IFP and the inverse of the NFP. If this union is empty, the item cannot be placed. An alternative method is to check IFP corners for validity and select the bottom-leftmost between them and the intersection points of the NFP and the IFP. If no such point exists, the item cannot be placed.

Similar to the Strict Bottom Left using the global NFP, the Strict Bottom Left on per-item NFP is also very simple. Using the same reasoning as with the global NFP, the BL-satisfying position must reside either on the IFP or some NFP edge.

With further inspection, it can be shown that the BL point must be either on IFP or NFP vertex, or at some intersection point of IFP and some NFP or of two or multiple NFPs. Hence to find such a point, we simply search for all such intersection points and vertices and select the bottom-leftmost valid position. The validity check is required since not all vertices and intersection points are valid as they could reside inside an NFP of a third shape. Fortunately as with the global NFP case, these validity checks on NFPs are simple point-in-polygon test as discussed in Section 3.2. A detailed description of the of the per-item NFP Strict BL is presented in [14].

As with other Strict BL methods, the Strict Bottom Left on both global and per-item NFP are clearly incremental with respect to the placed positions, but unfortunately this does not benefit us much as much as with the raster model. The reason for this is that the NFP is always dependent on the shape of the inserted item and the placed shapes, which in the general case prevents us from reusing the computed NFPs for the items. As discussed earlier, the computation of the NFPs is computationally complex and with these methods the NFP computation time can be expected to dominate the whole runtime. With the global NFP this is means that NFP needs to be recomputed for the bin for each insert.

With the per-item NFP, the invalidation problem is not as severe as the NFPs can be generated lazily on-demand as the intersection point search advances from bottom to the top. This way, computation of NFPs for the top items can possibly be avoided if a fitting position is already found in a lower position. Additionally, the computed NFPs for the tested shape pairs could be cached and later reused. However, we do not expect this to be useful for us as our target content as discussed earlier is strongly heterogeneous. By assuming each item has different shape, if a single item's placement to a bin succeeds, it is placed immediately and the cached NFPs are useless to following items. On the other hand, if the placement to a bin fails, the generated NFPs exist only for those shapes in the bin the inserted shape does not fit to. Hence the cache would be useless to any other upcoming bin. However, this only assumes caching over a single bin insertion sequence. With more complex scheduling algorithms which are described later and with various local search methods also described later, it is possible that NFPs could be reused over multiple local search or scheduling iterations and hence caching might yield some performance benefits.

3.3.3 Bottom-Left procedure

In addition to the Strict BL search algorithms, various approximations have also been proposed. One such method is the BL-algorithm defined in [23] that we for clarity call in this paper the Bottom-Left Procedure. The core idea is like in Strict BL to select items to be placed one-by-one from a predefined list but then instead of finding the true BL placement policy satisfying position, only a locally bottom-leftmost position is used. For finding such an approximation position, two variations are proposed.

The first BL-procedure variation is extremely simple: The item to be placed is initially placed to the top-rightmost valid position of the bin, and then it is alternately shifted downwards and leftwards to the last valid position on that axis. When the

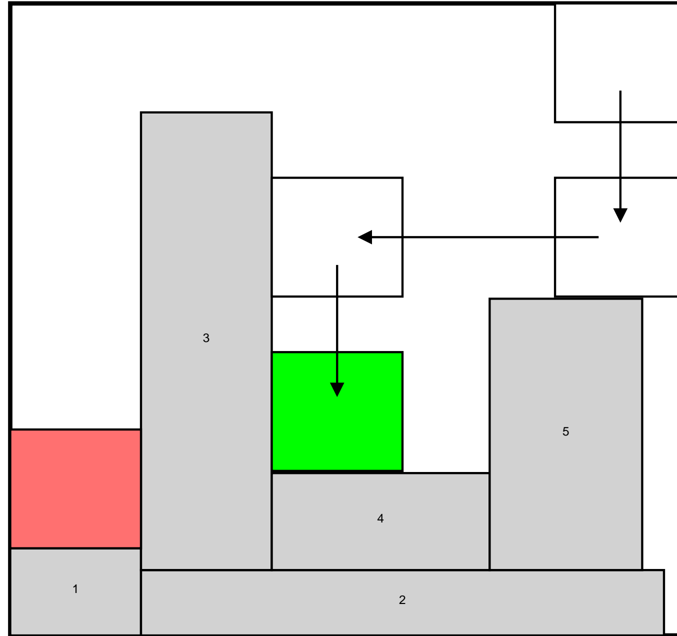


Figure 5: BL-procedure finds locally BL-placement policy satisfying position (green) but misses the globally satisfying position (red).

position can no longer be improved, the search ends, as show in Figure 5. If the original top-rightmost position is not valid, the item cannot be placed. Like Strict BL, this variation is clearly incremental, but unlike Strict BL due to the iterative traversal through only the valid solution space, it cannot perform hole-filling. For an example see Figure 5. The procedure settles to the green position from which is not the bottom-leftmost valid position in bin (marked as red). Even with changed shifting order such that item is first moved left and then down, the result does not change since the inserted item cannot fit over shape 3, which limits the packing front on its right side.

Although only producing an approximation of the BL position, the BL Procedure has very favorable performance behavior. As the search begins at the top right corner and advances towards the better positions, the amount of shifting decreases as the number of items in the bin increases. This behavior continues until the bin is full and the initial search position is no longer valid and the search is immediately terminated. Hence the BL Procedure is able reject early infeasible placements to full bins, potentially saving a significant amount of compute time that would be wasted in the futile attempt to find a valid position. Additionally, as the shifting is not a complex operation, the whole algorithm is also generally fast.

In the original paper, this BL-procedure variation was only defined for regular axis-aligned rectangles. However, the procedure can be easily applied to irregular geometry as well by simply implementing the item shifting on the used geometry representation. As with Strict BL, we define the variations for raster based representations, per-item-NFP, and for global-NFP but additionally also for direct geometry.

For the raster based geometry, the shifting is very simple: the item is merely moved

grid cell by grid cell on the desired axis and the new position is checked for overlap with the placed items. This is then repeated until the new position would overlap or the bin edge is reached. While this could be implemented more efficiently, for example by only checking the edges of the rasterized shape or by keeping the distance to the next occupied cell per row and column in a separate array, we consider them the same algorithm and do not discuss the various implementation-level optimizations here.

For the per-item-NFP and global-NFP, the shifting is also very simple. Since each shifting moves the inserted item either as right or as down as possible, the inserted item on each new position must touch either the bin edge or a placed item. Clearly if this were not the case, we could move it even further in the desired direction. Hence the reference point of the inserted item must reside on an edge of the IFP or an NFP. In order to find such a position, we simply find the first IFP and NFP edge directly below (or to the right) of the current position. For both per-item-NFP and global-NFP this is can be implemented with just axis-aligned ray to line-segment intersection tests which is very efficient. Similarly to the Strict BL using NFP, the computation time for BL-procedure can be expected to be dominated by the NFP generation. But similarly with the per-item-NFP, this can be mitigated with NFP caching and computing NFPs only on demand.

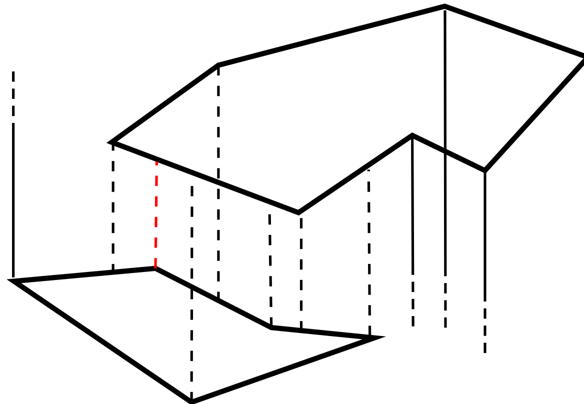


Figure 6: Vertical distance of two shapes (red).

For the direct geometry, the strategy is similar to the algorithm with NFP. To shift an item downwards to the local bottommost position, we compute the vertical distance to all the shapes that have edges directly below the inserted shape and the bin boundary, and then select the minimum. As with the NFP, the vertical distance between two shapes is the vertical distance the first shape can be moved before it will touch the second. Since shapes consist of a list of edges, the vertical distance is naturally the minimum distance each edge of the first shape can be moved vertically until it touches the second shape. For an edge to touch a shape, it must either have either a vertex on the edge of the shape, or the shape must have a vertex on the edge. To compute the distance of an edge vertex to the placed shape, we simply cast a vertical ray down from the vertex, and compute its distance. Similarly, to compute the vertical distance of the shape to this edge, we simply cast vertical ray up from

each vertex of the shape and select the minimum distance. As shown in the Figure 6, the minimum ray cast distance shown red is the vertical distance of the shapes.

3.3.4 Two-Step Bottom-Left procedure

Like stated earlier, the first BL-procedure variation was originally only proposed for rectangles. This was because computation using polygonal shapes was deemed too computationally expensive, especially when item scheduling was determined by a genetic algorithm search. To mitigate the associated costs of dealing with polygonal shapes, a second variation of the BL-algorithm was proposed in the paper [23]. In this variation all polygons are initially placed only coarsely using their axis aligned bounding box using the BL-procedure, and then after all the items are initially placed, a final compacting pass would occur. The compacting step would then select each item in the same placement order and apply BL-procedure on the polygonal level.

The algorithm as-is is only suitable for strip packing where the final compaction can reduce strip length. But for bin packing the final compaction after all items are allocated in the bin only moves shapes around without increasing the density. Hence, we define a following variant suitable for bin packing. As shown in Algorithm 2, the bin packing state consists of the lists of fine and coarse shape placements, both initially empty. Placement works as with original BL-procedure for rectangles until coarse placement can no longer be found. In this case if there are coarsely placed items in the state, we refine their positions and try again. If there are no coarsely placed items, the placement fails.

To refine coarsely placed shapes, a shape is placed on the bin its coarse position and then the standard BL procedure is applied to the shape against all the other refined shapes. Notably this BL Procedure does not consider other coarsely positioned shapes. Hence a care need to be taken to avoid refining one shape position such that is overlaps the coarse position of yet-to-be-refined shape. If such an overlap were to happen, a coarse position could become overlapping with a refined shape, violating the BL Procedure precondition. But since the BL Procedure only moves items down and left and since the coarse positions are initially non-overlapping, a valid item refinement order can easily be obtained with a topological sort with a condition that items to the left and to the down precede items to the right and to the top.

This variation is essentially a coarse approximation of the BL Procedure to avoid then-high computational cost involved and with present-day computational power available this optimization is less relevant. As we will see later in the evaluation results, a modern computer can easily position even a large number of irregular shapes using the traditional BL procedure in a reasonable time, and thus this method could be disregarded as useless for our content. However, as we inspect the Algorithm 2, we note that the coarse level placement (*TryFindCoarsePlacement*) is completely independent component in the algorithm. Hence we propose a following hybrid variation in which the coarse positioning is chosen by Strict BL instead.

We presume this modification significantly alters the characteristics of the packing algorithm to a degree in which this variation can considered a separate method. In

```
Function Insert
    data   : bin state as CoarsePlacements and FinePlacements
    input  : inserted shape as Shape
    output : success if inserted into the bin
    placement ← TryFindPlacement
    if error then return error
    CoarsePlacements ← append(CoarsePlacements, placement)
    return success
end
Function TryFindPlacement
    data   : bin state as CoarsePlacements and FinePlacements
    input  : inserted shape as Shape
    output : a valid placement or error
    coarsePlacement ← TryFindCoarsePlacement
    if no error then return coarsePlacement
    if any CoarsePlacements then
        RefineCoarsePlacements
        coarsePlacement ← TryFindCoarsePlacement
        if no error then return coarsePlacement
    end
    return error
end
Function TryFindCoarsePlacement
    input  : bin state as CoarsePlacements and FinePlacements
             inserted shape as Shape
    output : a valid placement or error
    // find valid position for the bounding rectangle of Shape
        among CoarsePlacements and FinePlacements using BL Procedure
end
Function RefineCoarsePlacements
    data   : bin state as CoarsePlacements and FinePlacements
    // compress CoarsePlacements, append them to FinePlacements, and
        clear CoarsePlacements
end
```

Algorithm 2: Placing a shape using the Two-Step BL Procedure.

particular, by using the Strict BL for the coarse positioning we mitigate the low quality positions of the standard BL Procedure caused by the limited view to the solution space. For example in cases such as the insurmountable packing front shown in Figure 5, Two-Step BL Procedure is able to explore set of potential positions and is able to find the true BL position. In the rest of this paper, we refer to exactly this variation with Two-Step Bottom-Left procedure.

3.3.5 Bottom-Left Resolving

While BL-procedure algorithms traversed through the valid solution space until we could no longer improve, an alternative is naturally to traverse through the invalid solution space until a valid position is found. In this paper we call these BL-Resolving algorithms.

As proposed in the paper [8] as an approach for strip packing, this resolving algorithm operates by first inserting a shape in the bottom left corner of the bin and testing whether it overlaps any already placed shape. If the placement is valid, the algorithm terminates. If not, the algorithm resolves the conflict by moving the candidate position upwards a distance just enough to make item-to-be-placed no longer overlap with some originally overlapping placed shape. Additionally, if the resolved candidate position ends up over the bin upper boundary, the candidate position is reset to the bottom of the bin and moved right a small fixed step. This process is then repeated until a valid position is found.

While the main focus and contribution of the paper lay in the packing of geometric shapes with curves and other non-polygonal features, we note this method is also suitable for packing more traditionally constructed shapes. But with this constraint, observe that by simply switching the horizontal and vertical axes and by using a small enough fixed step, this algorithm is essentially an alternative implementation of the Strict BL placement and as such is not very interesting.

However, this method inspires us to propose a following variant. Instead of tracing a single candidate position, we maintain a priority queue of of candidates. At the beginning, the bottom left position is inserted to the queue and the algorithm advances by popping the bottommost left position in the queue. If the position is valid, the position is taken and the algorithm terminates. If not, we resolve both the minimum horizontal and the minimum vertical offset required to resolve the overlap with some overlapping shape. Then both of these resolved points are inserted into the work queue and the process is repeated. If a resolved position causes the shape to lie outside the bin area, the point is ignored and removed. If the work queue is exhausted without finding any legal position, the algorithm terminates with an error.

We essentially just replace the fixed step with an adaptive step and form a tree-like scan pattern which skips over large placed items. As we need to validate a lot smaller set of positions, we expect this method to find legal positions faster than the standard Strict BL. But since the found positions are not necessarily globally bottommost left valid position or even locally bottommost left valid positions, we expect the packing density to be lower. In particular, we expect the hole-filling ability to suffer for complex shapes: as each overlap resolution always moves the item completely on the top and the right side of the placed item, many potentially interesting positions between are omitted. In the rest of this paper, we refer to exactly this variation with Bottom-Left Resolving placement.

3.3.6 Clustering approaches

The placement methods presented above all attempted to compute the absolute position of the inserted shapes. Naturally, an alternative approach is to instead

compute relative positions of the placed items and only finally commit these relative positionings into the bin. We call this relative positioning of a set of items a cluster, and in this section we present various positioning algorithms based on the clustering approach.

In [40], the authors propose a method called TOPOS in which a single item cluster is built by attaching unplaced items to it one-by-one, incrementally. The next item to be attached it selected by testing all possible options and choosing the one that results in the lowest error metric value. For error metric, multiple approaches are shown and evaluated in the paper. The algorithm terminates when all items have been placed.

In [21], the authors present a clustering approach for irregular shape bin packing with guillotine constraints. In their approach, the shapes are attached pairwise together using a dynamically weighted error function that attempts to balance between tight packing and rectangular cluster shape. These clustered shapes are then attached to other shapes and other clusters with the same error metric, creating a small set of potentially large clusters. Finally, these clusters are placed on the bin.

Finally in [16] the authors propose a partial clustering strategy for strip packing. In their approach, all non-covex items are grouped together in pairs and these pairs are then packed together. The selection and the relative positioning of the grouped pairs is based on feature matching that attempts minimize the wasted area on the combined convex hull.

3.3.7 Mixed-Integer Models

In addition to the various heuristics shown so far, there has been recently interest in generalizing the placement problem into a Mixed-Integer Linear Programming (MIP) problem [43]. A MIP problem is an optimization problem in which a set of linear integer and real constraints and objective function is given for a set of variables, and the task is to find the variable assignment maximizing the objective function. For example, by encoding the non-overlapping criteria into a MIP model constraints and feeding this problem into a general purpose MIP solver, a valid layot can be found, if such is feasible. The benefit this approach is that by constraining the desired error metric, the resulting layouts can be solved up any desired error value, provided that such constraint can be satisfied. This in turn can be used to find and prove the optimal layout.

In practice however for finding the optimal layout and encoding the placement constraints for all shapes produces quickly very complex MIP models as the number and complexity of placed shapes grows. Solving these models is prohibitively expensive computationally for any larger number of items, and for example the currently known largest optimally solved layout problem are around 30 items. As such, even with many complexity mitigations and simplifications, these methods are not useful to our content but we list out various efforts for completeness.

In [46], authors propose a raster model named Dotted-Board Model in which the bin is represented as a grid of dots. Each dot is a set of boolean decision variables, each corresponding to the type of the placed piece. If such a boolean variable is set,

then the reference point of a shape instance lies on that dot. The reference point of a shape is any chosen point from which No-Fit-Raster is expanded upon, similarly to the reference point of the No-Fit-Polygon. Now, for each type and each dot, it is constrained such that if the type on the dot is set, then for every type there may not be other reference points set within the No-Fit-Raster of these two types. This, and additional constraints for items being inside the bin, enforce a valid resulting layout. While being straightforward in design, it is unfortunately computationally very expensive as the number of shapes or the resolution grows. Additionally, due to the inherent rasterization in the core of the model, the solved layouts are not guaranteed optimal.

In [11], authors provide MIP problem encodings for the strip and single bin packing problem with variants for using either direct geometry representation or an NFP construction generated on each of the shape's convex components. In their tests the NFP construct had significantly better performance. Even still the authors speculate that problem instances larger than 30 pieces would already become computationally infeasible.

In [3], the authors encode a polygonal strip packing problem into a MIP using a sliced NFP construction. As with other MIP models, the approach is computationally very expensive and only small problem sets can solved within a reasonable time. Then recently, this approach is in [34] adapted for multiple bin packing using an clustering, constructive approach. As with the underlying method, this approach has initially severe limitations with the performance when solving the layout for a single bin for any larger number of items. Hence, the authors mitigate this issue by switching at to a higher-performance approximation if a chosen time limit is exceeded during the layout. This approximation is composed by fixing the relative positions of the placed items in the previous step of the construction, and only letting the inserted item move freely. But even with this mitigation, this approach is prohibitively expensive for our use case.

3.3.8 Item orientation

Finally for placing an item into a bin, it is clearly not sufficient to just decide upon the positiong of an item, but the item orientation also needs to be chosen. This selection is not trivial and poses yet another variable to our combinatorial problem. In this section, we discuss various methods for choosing an orientation found in the literature.

For an incremental placement algorithm, a commonly known strategy for handling a small, fixed number of orientations is to simply test a placement for each of them consecutively, and then select the orientation that provided the best positioning. This greedy approach of solving the orientation during the placement is used many works [23, 16, 40], and it has many attractive properties for practical implementations: first, it is decoupled from the placement logic enabling more flexibility in implementation. Additionally if a a valid position is found early in the sequence, the resulting positioning can in some cases used as an upper bound for the remaining orientation tests potentially saving a significant amount of compute effort.

However, it is clear that this approach does not support free rotation and the naive approach is to retort to quantization to the potential angles. But this is not feasible in practice, as it is also clear that the method does not scale well as the number of different orientations to test grows. Hence, in [35] an approach is introduced for selecting only a small subset of promising orientations to test. In their method, a set of candidate orientation is generated by matching the edge directions of the inserted shape to the directions of the edges of the placed items and of the bin boundaries. Then the final candidates are selected based on how prominently each angle is represented in the candidate set and based on the length of the matching edges. By controlling the cutoff for the final candidates, the balance between quality and performance can be controlled. This approach is used in [34].

Finally, an alternative approach is to abstract away the whole concept of item orientation selection from the item placement component, and move it to the overlying local search implementation. This approach is taken in [23] where the item orientation is encoded along the item insert sequence and this then updated with the insert order within a genetic algorithm search. The benefit of this approach is that by solving the orientations externally, the packing algorithm can use a placement policy that does not support item rotation, such as the Dotted-Board Model.

Additionally, as the local search has global vision to the complete packing process, it can be expected to be able to eventually find a better layouts than with the greedy orientation decisions that only optimize the for the local fitness. However, at the same time due to the lack of local vision the greedy orientation selection has, and due to the global packing fitness being very indirect indicator of fitness of any single orientation decision, the search can be expected to be extremely difficult. Hence we do not expect the local search to be able to match the quality of the local selection without needing an excessive number of search iterations.

3.4 Scheduling algorithms

Like described earlier, in addition to the placement algorithm, a solution for an SBSBPP also requires a decision for each item into which bin it is allocated and then subsequently positioned. In this paper we call this process of allocating items into the bins the scheduling of the items. In this section, we describe and discuss various scheduling methods developed in the literature.

Simple construction heuristic (SCH) is a commonly used and simple model for the bin assignment. [34] In SCH, we in the beginning create a single bin, and then keep adding new items into one-by-one in an externally specified order until the placement algorithm can no longer find a satisfying position for all the items. Now we remove that last overflowing item from the bin, making the placement satisfiable, and then consider that bin complete. Next, we create a new bin, add the overflowing item into it and continue from the where we left off adding items into it one-by-one. When we run out of items, the algorithm terminates and each item is now assigned into a bin and each bin has a satisfiable placement.

This repeated packing naturally leads to a large number of very similar a bin packing operations, which can lead to high computational cost. Luckily we in many

cases can avoid it: if the placement algorithm is incremental, we can improve the SCH implementation run time by keeping the placement state of the currently open bin, and then incrementally adding the next item. If placement succeeds, we repeat the process. But if not, the kept placement contains the maximal bin configuration, and we can simply create a new bin and continue with the process there. But even if the underlying placement algorithm is not strictly incremental but merely constructive, we can potentially partially reuse previous results. For example in the case of [34], the authors develop a hybrid strategy where for each bin composition a search for an optimal layout is attempted. If however a certain time threshold is exceeded in this search, a faster approximation method is used. In this approximation, the previous item layout is kept unchanged, i.e. reused, and the next inserted item is placed adjacent to the existing layout.

Clearly, this method is very sensitive to the order of items to be placed and the packing quality can collapse quickly if an unfavorable series of items is encountered. In a pathological example, if three items of size 9 and three items of size 2 are to be placed into bins of size 6, in the optimal order only 4 bins are required. If they however are placed in the order of 9, 2, 9, 2, 9, 2, each item will be placed into separate bin resulting in the total of 6 bins. Naturally this is not a problem in practice as the impact of poorly behaving item order will be mitigated by the external, higher-level local search solution. However, we expect this high sensitivity to the insert order to make the local search harder.

In our previous example of pathological item order of SCH, we exploited the fact that after allocating a new bin, SCH no longer tries to insert into the old bins, even if were mostly empty. While this can be mitigated on a higher level, we can also try to solve it by defining an aggressive variant of SCH (aSCH) that is not susceptible to this collapse. Our aggressive variant differs from the vanilla by simply never considering any bin complete and always trying to add the current item into any of the created bins in the creation order before falling back to creating a new bin. Essentially, we try to pack all the items into the first bin, and the remaining overflowing items into the second bin and so on until all items are placed. Similar to the SCH we can optimize for incremental placement algorithms but instead of keeping just the current placement we naturally need to keep the placement state of all the created bins all the time.

While this aggressive variant solves our pathological-to-SCH case, it's clearly not without downsides. Since the bins are never deemed finished the first bins will quickly end up very full, but will still be considered as potential bins for the new items. This can be presumed to lead to large amounts of futile packing attempts into practically full bins when the number of items and the number of bins is large. Unless the placement algorithm can reject early these very likely overflowing items, this can potentially lead to large computational costs.

Additionally, since an item is placed into the first bin it fits, multiple different orderings of the input items can lead to the same bin allocation and item layout within the bins. This means that multiple item order permutations, especially if modifications are not in the early parts of the sequence, will result in the same packing. This behavior can cause suboptimal performance on the higher level searches that

have no visibility to the resulting bin allocation, for example with tabu searches and genetic algorithms operating on the item order that are discussed later in Section 3.5.

The scheduling algorithms shown above have all operated greedily in the externally provided item insertion order and then making the bin assignments online. However, this is not required by the problem description as we have information on all the items to be packed and this can be assumed to be just an unnecessary restriction. In the following parts of this section we concentrate on offline algorithms that try to solve the item bin assignment having available the information of all of the items.

In [34], the authors introduce and evaluate 4 novel scheduling algorithms. The first one, Bin Packing with Greedy Decisions (BPGD) is based on the idea of collapsing the polygon and the bins to merely their surface areas, and approximating the problem as a 1-dimensional bin packing problem (1DBPP). By solving this 1DBPP, the shapes are allocated into a minimum number of bins such that the total surface area of a items in a certain bin do not exceed the bin surface area. This optimal allocation naturally only holds in the 1-dimensional case, and for the 2D case it merely implies that the allocation is not trivially impossible. Now with the initial, approximate bin allocation, the 2D layout is attempted for the bin with the highest surface area utilization. Within this bin, the allocated items are inserted in a desired order until all items are placed or a placement cannot be made. If items are exhausted, the bin is complete and we move on to the next open bin with the highest surface area utilization. But if an item cannot be placed, we switch into a aSCH-like mode for this bin where all remaining items, even in other bins, are attempted to placed sequentially in a desired order. Since this operation might succeed in placing items initially allocated to other bins, the 1DBPP approximation is recomputed and the process is continued to the next open bin with the highest surface area utilization.

The second one, First Fit Algorithm (FF) is very similar to the previous BPGD algorithm but instead of optimally solving the 1DBPP, the initial allocation is approximated with First Fit Decreasing (FFD) [24] algorithm. This FFD approach is essentially aSCH for one dimensional packing: items are input in a specified order, and each item is placed to the first bin with capacity for it. If no such bin exists, the item is inserted into a new bin. Similarly to BPGD, the approximate allocation is then attempted to be placed and when a placement error occurs, the process is repeated for the remaining items.

The third scheduling algorithm, Two Phases Strategy (TPS), is also a variation of the BPGD approach. However, instead of simply solving the 1DBPP to minimize the number of bins, it also tries to distribute the items within the bins such that the actual packing will result in more efficient layouts. As the authors note, it is widely known in the literature that naive greedy solutions often lead to poor layouts as large, more-difficult-to-place items are more likely rejected and easy items accepted in the early parts of the process. Hence, these hard-to-place items and only hard-to-place items will be left near the end of the process causing sparse, non-optimal layouts. To combat this behavior, the authors propose a model that prioritises placing large objects in the beginning of the process. This is completed in two phases: first the 1DBPP of the BPGD is solved and the minimum number of bins is recorded. Then a second 1D Integer Programming (IP) optimization problem is solved that assigns

the item surface areas to the given number of bins while also rewarding the larger items in the early bins. With this tweaked initial allocation, the algorithm proceeds as the BPGD.

Finally, the fourth proposed scheduling algorithm, Partial Bin Packing (PBP), takes a different approach than the previous three: instead of trying to solve the initial approximate allocation for all of the bins and then refining allocations as the placements fail, the PBP simply generates an approximate allocation for the currently open bin. Similarly to the TPS, the PBP tries to prioritise the placement of large, hard-to-place items first to avoid ending up with only hard items left. This is performed by assigning each item a both a cost and a value, the cost being its surface area and the value a surface area squared, and then selecting a set of items such that the total value is maximized and the total surface area does not exceed the bin capacity. After this Knapsack Assignment problem is solved, the selected items are then attempted to be placed into the bin one-by-one as with the BPGD. When a placement error occurs, the Knapsack Assignment is solved for the remaining items and for the remaining bin capacity and the placement process is repeated. If the Knapsack Assignment cannot select any item and there are still remaining unplaced items, a new bin is opened and the process is repeated.

3.5 Metaheuristics and local search

A placement policy combined with a scheduling algorithm is sufficient to produce some valid packing layout for a certain input. But since the goal is to minimize the number of bins, the problem becomes NP-Hard and exhaustive methods become quickly infeasible. As such, various heuristical methods are commonly used in search of an improved layout. In this section, we briefly list various approaches in the literature.

Many local search approaches operate on the item insert order. In [23], the authors use a genetic algorithm to search for neighboring packing layouts and in their construction, the item insert order is encoded directly as the genome. The genome fitness is based on the amount of unused surface area. In [25], the authors similarly encode the item insert order into the genome but fitness function is the packing strip length. And in [8], the authors use a combination of tabu search and hill climbing in attempt to find an item insert order that minimized the packing strip length.

Alternative approaches exists too, and in some approaches the search operates on the order constraints and other less direct influences. In [19], the authors use a genetic algorithm to search for neighboring packing layouts, but in their construction, they encode in the genome the item pair neighboring relationships such as coarse relative positioning. The genome fitness is similarly based on the amount of unused surface area. In [12], authors propose the Jostle method, in which all placed items items are repeatedly inserted to the bottom-leftmost and to the bottom-rightmost position. The order in which the items are moved is based on the horizontal location they had on the last iteration, and the items in the "shake" direction are solved first. This essentially selects the item insert order from the resulting positioning.

In addition to item insert ordering, the local search can be implemented on the

scheduling level. In [34], the authors use a hill climbing approach for improving layout quality in multiple bin packing task. In their approaches, items from less utilized bins are moved into either one or multiple more utilized bins, and in case the all items in the source bin can be moved to other bins, the empty bin is removed. And in [7], the authors adapt a beam search to the item clustering selection used by the TOPOS algorithm discussed earlier in Section 3.3.6.

Finally, the local search can also be computed directly on the placement level. In [16], the authors use a cuckoo search implementation to find new positions for the items. As the search is suspect to getting trapped in local minima, the authors combine the cuckoo search with a guided local search approach. And in [32], the authors propose a local search based on swapping the positions of two randomly selected items and then correcting the layout. To avoid local the local minima trap, the authors use a tabu list to blacklist swapping a recently swapped items again.

4 Evaluation of packing methods

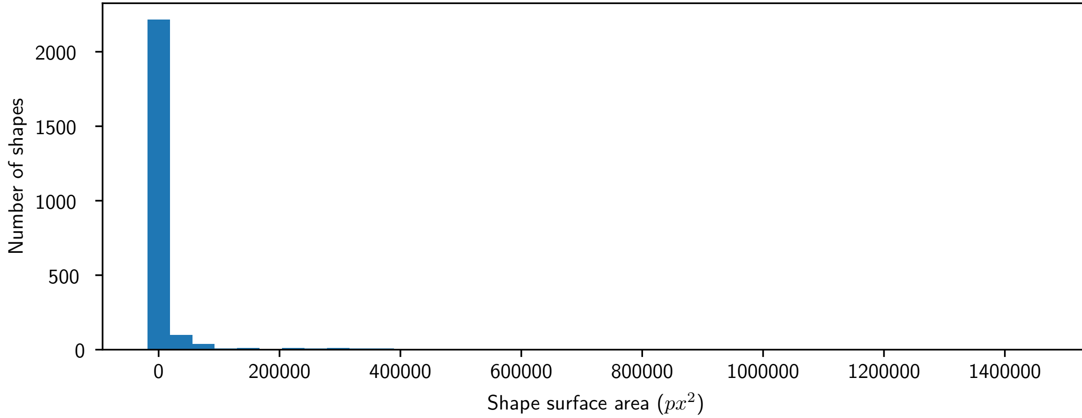
We evaluate packing methods suitable for medium sized irregular bin backing tasks needed for mobile game texture atlas generation. As described in Section 2, our goals are both minimization of required wall clock time and the number of atlases, i.e. bins, required while satisfying the additional graphics hardware originating constraints. In this section, we first define our general evaluation methodology and then the necessary steps taken to adhere to these declared constraints. We then disclose and discuss the issues and details emerging from the practical implementations of the evaluated algorithms. Then we perform the initial evaluation of the layout density obtained with the methods, and discuss the findings. Finally we evaluate the packing algorithms on basis of the compute time required, both as is and by relating it to the achieved packing density, and analyze the results.

4.1 Evaluation methodology

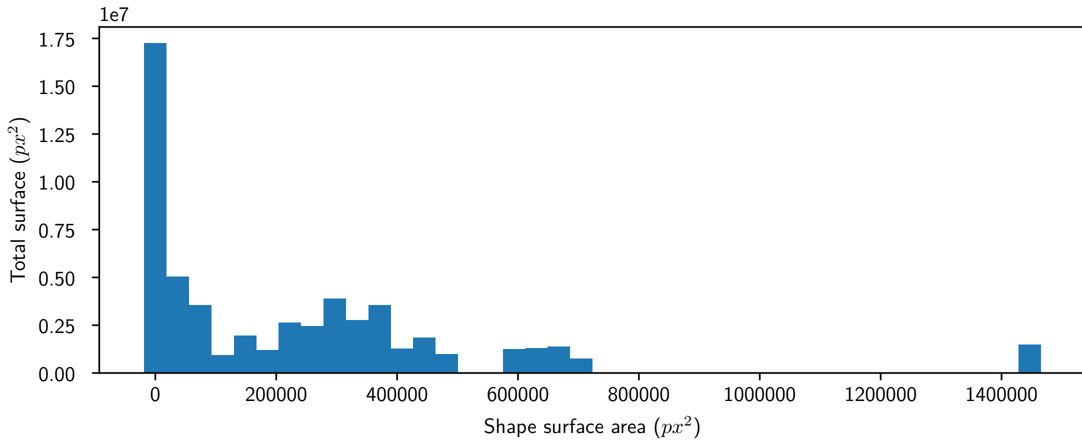
For a deterministic algorithm, the result is the sum of the algorithm and the input. In Cutting and Packing as we have earlier described, the algorithm is commonly the sum of its packing policy, scheduling policy, and a local search. As per our proposed requirements, the eligible algorithms need to be deterministic and since our test input is fixed, it is sufficient to describe each component independently. First we describe our test content, and then continue with selected packing policies, scheduling policies and our approach to local search. Finally, we state our goal metrics and define how they are computed.

Our test content is derived from the sprites of a development version of the target game by choosing a single revision in the version history and then generating bounding polygons for all sprites required in that particular build. The test content consists of 2443 polygonal shapes each with 3 – 10 vertices, and the test shapes are mostly convex. What is notable is that the size distribution of the items is very uneven and uncommon. As seen in Figure 7(a), the size distribution is very skewed and dominated by small items. Since in some approaches, such as rasterization based methods, the computational time depends directly from surface area of the item to be placed, we also present the total area contributed by each area size bin in Figure 7(b). In this graph we also observe that the total surface area is also heavily biased towards small items and hence large portions of the resulting bins will be consumed just for small items. Additionally the test input contains one affinity group with 23 items with varying sizes, and with a total size of about one quarter of the bin size. It is hence unlikely that the group affinity would be honored without an explicit enforcement mechanism. We discuss the implementation of this mechanism later in the Section 4.2.

For placement policies, we test in the evaluation the Strict BL, the BL Procedure, the Two-Step BL Procedure, and the BL Resolving methods. We select both Strict BL and BL Procedure as both are widely used and evaluated in the literature and they are often confused with each other. Additionally we select the Two-Step BL Procedure and the BL Resolving as the variants we have introduced are novel and



(a) Number of shapes by shape surface area.



(b) Total surface area by shape surface area.

Figure 7: Input size distribution.

hence interesting. Notably we do not test any MIP or clustering model. As all tested policies are of BL class, the analysis of the approaches can be made more in-depth and the comparisons can be assumed to become more meaningful.

To handle item rotations, we have chosen to solve the orientations greedily during the item placement. As the item transform set is of the symmetry group D_4, for each item placement we test perform a test placement in each of the 8 transforms and then select the best. This best placement is then committed into the bin. Notably we do not solve the rotations as a part of the local search. As the local search lacks direct vision to the bin layouts, it can expected that for the first search iterations the chosen orientations would be sub-optimal. In this evaluation, we want to avoid excessive local search as we will discuss later. Hence tying the orientation to the search would likely yield uncharacteristic, unrealistic results.

In this evaluation, we test three scheduling policies: SCH, Aggressive SCH and PBP. We have chosen SCH since it is fast, simple and very commonly used in Cutting and Packing literature. While we expect it to produce relatively low-density layouts since is closes a bin on first placement failure, we expect it provide us good and

comparable baseline results for various packing policies in terms of CPU time. At the opposite end of the spectrum is the Aggressive SCH which we expect to produce high-density results but due to its brute-force approach with a high CPU cost. We expect this to provide us good, comparable results for the packing policies in terms of packing density. Finally, we evaluate packing methods using PBP scheduling which has been shown to produce higher-quality results than SCH within a comparable CPU time [34].

To mitigate the combinatorial explosion from including various local search approaches, no conventional local search is done. Instead we emulate the local search by sampling the packing methods with various fixed random inputs and reporting the best packing quality found. This process can be interpreted as a hill climbing with an infinite neighborhood radius. While this approach is expected to result in lower quality results than with a proper local search, we expect the loss of quality to manifest relatively consistently in all test samples which should keep the packing quality results comparable.

Naturally this assumption might not hold if an irregular packing method has very sporadic quality variations and our selected input patterns just happen to hit the pathological cases. Were such the case, we interpret the method to be inherently flaky that was only masked by a well-functioning local search. As this kind of flakiness is not tolerable in our production pipeline, the worsened results for such methods are acceptable to us.

For SCH and aggressive SCH, we implement this by randomizing the input shape order with various fixed random seeds. Additionally we test with shape sequences ordered by area and circumference in both ascending and descending order. For PBP, we similarly test randomized item weights, and additionally with weights corresponding to shape area and circumference. It should be noted that some of these tested patterns could be pathological for the some packing algorithms and hence results could vary a lot. Similarly to our approach to local search, we don't consider this as a problem and rather see sporadic performance during the evaluation than in production.

As for compute time, we track the average time over iterations using a single logical CPU core of Intel Core i7-3610QM CPU with a base clockrate of 2.30GHz and ability to scale up to 3.30GHz if the thermal budged allows for. As the computer was idle except for the test process, we believe the actual clockrate to be very close to this upper limit of 3.30GHz. Even though our goals explicitly stated that we are interested in wall clock time using multiple cores as the fine-grained parallellization is vital for any forward-looking real-world implementation, our assumption is that with a reasonably efficient packing iteration the parallellization can trivially be added to the local search level. With this assumption, all the evaluated methods have fairly very similar parallellization characteristics and hence the single threaded performance can be seen as a scaled approximator of the expected performance. While the results are not indicative of the real world performance, they can be expected to be mutually comparable.

Finally, as the number of bins expected is low (less than 20) using the bin count directly as the error metric is not very informative. While minimizing the number

of bins is stated as our main objective, we aim to minimize the number of bins not only with our test content but with all production content of similar characteristics. To this end, we try to evaluate the packing densities achieved on the texture atlas subset the packing method has “consumed”, i.e. is either in use by a placed shape or is free space that likely cannot be used for any further placements.

Multiple approaches exist to approximate this density solely on the consumed area. In [21], the authors describe method which produces a continuous metric for bin usage. In this method bins are first divided into full bins and a single partial bin, where the single partial bin is the one with least utilization. As visualized in Figure 8, the partial bin is then cut into two parts, one with placed items and one consisting completely of unused space. The cut is made with a horizontal or vertical cutting line such that the portion of the unused area is maximized. The metric is then obtained by summing the number of full bins and the relative size of the used half of the partial bin. In our example figure, this continuous bin count would be around 2.6.

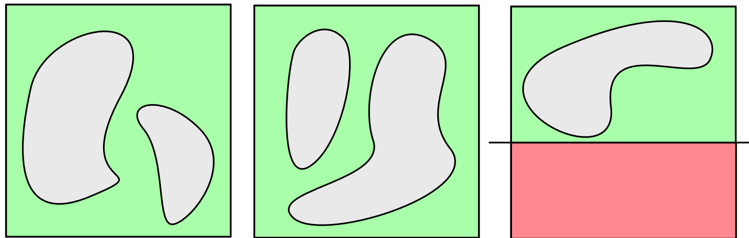


Figure 8: Fractional bin count. Consumed area in green, reusable area in red.

In [33], an alternative density f metric is introduced based on the relative area consumption of the bins:

$$f = \frac{\sum_{N}^{i=1} U_i^2}{N} \tag{1}$$

where N is the number of the bins and U_i is the relative size of the consumed area in bin i, i.e. sum of the shape areas in the bin divided by the bin area. This metric is is designed to reward packing layouts in which most bins are full or almost full instead of distributing items equally between the bins.

In this paper, we report the number of bins required and the density metric f described above. We believe the f value is good metric for assessing the density of incomplete packing processes. Notably we do not report the partial bin count as defined in [21], as we believe this approach assumes linear, axis-aligned packing front. This method might be appropriate for physical material reuse or for visualization of the consumed space, but in our use case where we incrementally insert more shapes into any valid location, this metric would report larger values than is desired for a method able to perform hole-filling.

4.2 Enforcing constraints

As we discussed earlier in Section 2, packing atlases suitable for mobile game rendering imposes additional constraints on packing algorithms. While some requirements

such as the need for bins to have power-of-two size are trivial to satisfy, fulfilling some requirements need significant changes to the original methods. In this chapter we describe the necessary changes and other non-trivial steps taken to satisfy the constraints.

To mitigate color bleeding caused by texture filtering, we defined that each shape to be packed should have a buffer region approximating the filter range around it on which no other shape is placed. We implement this buffer by generating an NFP of the original shape and a filter shape scaled down to a half, and then packing those instead. With this scheme, each shape is separated by at least two half-filter sized regions, satisfying the requirement. If the filter shape approximation is symmetric is D_4, this can be done as a simple preprocessing step. However in anticipation of adapting to texture compression formats with non-isotropic resolution along the axes, we do not assume any symmetry for the filter shape, and hence we recompute this shape-filter NFP when the original shape is rotated. To avoid adding useless boundary between bin edges and shapes, we also expand the bin boundaries such that added buffers do not affect the effective positions of the original shapes.

We also require that texture filtering samples should be exactly identical between the original and the packed sprite asset. To guarantee this, we necessarily need one-to-one mapping from each original asset texel to the atlas texels. For raster models, we satisfy this by naturally selecting the raster grid resolution to match the texture atlas resolution. Hence each raster cell matches a texel in the resulting atlas texture excluding the cells falling under the extended bin boundaries, and hence one-to-one mapping is trivially preserved. For polygonal modes, the solution is also conceptually simple. We simply select the scale of the original shape to original sprite texture size, bin size to match texture atlas size, and then make sure shape offsets, i.e. results from the chosen packing policy, are multiples of the texels size. Since we chose scales to match assets sizes, the texel size becomes 1 unit by 1 unit and hence its sufficient to guarantee each position is a whole number.

To guarantee integral positions for the packing policies, we may not just round a non-integral packing position and use the resulting position as a substitute, since rounding a position would by definition move the shape from a legal, non-overlapping position to a position that has no such guarantees. One option would be to augment filtering boundaries further by NFP with 2×2 square at origin representing rounding radius. But since this would be wasteful, we instead modify placement policies as follows. For Strict BL we simply consider and validate the rounded-up-to-texel coordinates for each BL position candidate. For BL procedure, we fix the initial position to the texel-grid and round down each shift length to integral length. For Two-Step BL Procedure, we additionally align the coarse positions to the grid, and finally for resolving models, we similarly fix the initial position to the grid and round up each resolution length.

To make SCH and Aggressive SCH bin scheduling methods to adhere to the item affinity requirements, we implement a dynamic reordering and rollback mechanism. When an item within an affinity group is to be placed, we take a snapshot of the current bin state, and then try to place all the items of the group into the bin effectively reordering all affinity group members into a contiguous span in the shape

list. If any of the group members cannot be placed into the current bin, we roll back to the original bin state and move all group member shapes to a separate next-bin list. When a new bin is opened, this next-bin list is then prepended back to the shape list.

To implement affinity in PBP scheduling, we modify both initial and final placement phases. In the first phase while solving the1D knapsack problem, we replace each item within an affinity group with a single item representing the whole group summing the weights of its parts. Then during the final placement into a bin, we employ the similar dynamic reorder and rollback mechanism as with SCH. However unlike with SCH variants, on failure we do not force the group to be packed in the next bin. Our assumption is that since the group was initially allocated into this bin, the group must have relatively high a combined weight and hence the group would likely to be still packed in the next bin.

Finally, to support for incremental adding of shapes, we build on the defined affinity mechanism and the inherent determinism of the methods. To add an an shape to an existing layout, we set each existing item into bin-specific affinity groups and append the new shape with no affinity requirement. By fixing any local search parameters, due to deterministic scheduling and placment the resulting bins with no changes will stay unmodified. Due to triviality, we do not pursue this further in this paper.

4.3 Implementation considerations

While we are mainly interested in the packing density achieved with the presented algorithms, a method to be feasible for industry use must find the solution in a reasonable time frame. To assess this aspect of the algorithms, we track the average CPU time during our test runs. However this actual run time depends on many implementation details that do not affect method correctness or be in any way visible in the algorithm descriptions provided earlier. In this section, we disclose and discuss such potential and taken optimizations in the evaluated implementations.

The PBP scheduling requires solving an Optimal Knapsack Assignment problem to select items that are attempted to be placed into the bin. In our tests we noted that this Knapsack Assignment becomes quickly extremely time consuming as the number of placed items grows. This behavior is is not completely unexpected as the Knapsack Assignment is NP-Hard and the number of items to be packed in our production content is significantly larger than in any known results for utilizing PBP scheduling. More specifically, we assume the catastrophic performance is due to the large number of small items the test set contains as described earlier. With this content, a bin will have very high number of potential configurations with each having similar total value and hence a traditional Branch-and-Bound search will not be effective enough to solve the problem in a reasonable time. However, to prevent PBP solutions from becoming uselessly slow for our internal needs, we instead only approximate the Knapsack Assignment by applying the greedy algorithm on randomized initial configurations. We don't expect this optimization to affect results significantly as assigning items solely by the area is already very coarse an approximation.

In the Two-Step BL Procedure variant we defined, the initial coarse positioning is computed for the axis aligned bounding boxes using the Strict BL rule. As the number of items per bin is expected to be relatively low, this BL search is implemented in a rather naive way. For each placed axis aligned bounding box, we store a so called X and Y anchor lines in a separate sorted sets. The X anchor line is a vertical (i.e. fixed X coordinate) half line upwards from the bottom right corner and the Y anchor is horizontal half line rightward from the top left corner of the bounding box. Finally we store anchors for the bin bottom left corner. As the BL position if it exists must reside at an intersection of these anchors, we simply iterate the intersection points in BL order and at each intersection point validate the position. The validation is a simple bounding box intersection test against all other bounding boxes. Since both anchor sets grow at $\mathcal{O}(n)$ and our naive validation test is also $\mathcal{O}(n)$, it is clear that this $\mathcal{O}(n^3)$ approach is wasteful and will not scale for a large number of items. In our tests and with our test content however the validation was observed to be reasonably efficient proving this approach sufficient.

Like discussed in Section 3.2, various geometry representations can have vastly different implementations with different feature sets and performance implications. For raster models, we have chosen to represent the shapes as bitmaps as this approach has many benefits. First, bitmaps can trivially represent any shape be it convex, concave, with holes, or even disjoint. While this level of flexibility is not strictly necessary for the problems presented in this paper, it allows us trivially extend and continue our work if even if atlas packing requirements for the game or the game content characteristics significantly change. For example, a change in game theme, art direction or even in some game design elements would significantly alter the packing content characteristics. Secondly, bitmaps are extremely simple and various algorithms can easily be implemented with high confidence of their correctness. Finally, bitmap algorithms usually enjoy a consistent performance that does not depend on the highly unpredictable quirks of the input content.

Unfortunately, this consistent performance is with medium to large bitmaps usually also a very low performance. For example, finding the Strict BL position for a shape represented by a $A \times B$ bitmap in a bin represented by $C \times D$ bitmap in a straightforward implementation will take at worst $ABCD$ cell comparisons. Similarly, the shifts required for BL Procedure and BL Resolving placement methods will also have a low performance when implemented in a naive manner. As such, we have implemented various heuristic and algorithm-level improvements to mitigate the issue outlined below.

For shifts required for BL Process and Two-Step BL Process, the goal is to move a bitmap either horizontally or vertically in one direction such that it touches but not overlaps either another shape or the bin boundary. We implement this by computing the distances from each bitmap edge cell to the first occupied bin cell in the move direction and selecting the minimum. While this significantly faster than the naivest implementation, the repeated computation of what is essentially the directed distance field on the bin is still wasteful and could be improved.

We also employ these edge cell distances in the shifts required for Strict BL search and BL Resolving process. In these cases instead of scanning with the complete shape

edge as a whole, we first scan in the desired direction with few selected probe locations that are contained by the shape. As it is necessary for these probe locations to be satisfied in a position that satisfies the whole shape, by choosing well representing set of probe points reject early many invalid positions. Essentially this approach is a hybrid of bitmap and bag-of-points raster representations.

Additionally, since some scheduling algorithms will result in a very high number of packing attempts that will fail (such as Aggressive SCH), the computation time will be dominated by how fast a placement policy can reject the placement. Since Strict BL policy does not have early rejection mechanisms combinations like this exhibited extremely low performance with runtime counted in multiple hours which prevented any meaningful evaluation. To keep these methods viable, we add a following early rejection mechanism: before the placement, we generate a span list approximation of the inserted shape and measure maximal free horizontal spans in the bin. As long as the span list approximation is a subset of the of the original shape, the span list to fitting in the bin is a necessary precondition for the actual shape to fit into the bin. However rather than properly fitting the spanlist into the bin, we simply validate a few preconditions necessary for such fitting to be possible. While this test could be tightened, we observed that even this minimal rejection logic mitigated the catastrophic performance and was sufficient for our needs.

Unfortunately, even with these optimization we believe our bitmap algorithm implementations to be of low quality. In general, while the bitmap approaches do provide the high generality and a consistent, predictable behavior, we believe the associated cost is prohibitively expensive in any real world, medium-to-large scale industry setting. We expect any industry use to utilize a raster representation better suitable for their particular use case. Even still, we have chosen to use bitmaps in our evaluation for the reasons we listed above and to avoid discriminating any single method or class of algorithms.

Similarly to the bitmap geometries, various implementation details concern the polygonal geometry methods as well. For direct geometry, the most crucial operations are overlap detection and computation of a horizontal and vertical distances and resolves. As the naive implementations of these methods significantly slower than our version, we describe our methods below.

For polygon overlap detection, we take following four steps: first we reject overlap if the precomputed axis-aligned bounding boxes do not overlap. Then we check either polygon contains a point of the other polygon. If so, the polygons overlap. Next, we validate if the contours of the polygons intersect at any non-vertex position, and finally if an evidence of an overlap or non-overlap has not yet been found we validate if the polygons are exactly equal. With the exception of the first early-exit, the last three checks have proven to be necessary and sufficient in our production use. As the early bounding box rejection and the any-point-in-poly check are nearly always sufficient for a result, we have implemented the polygon contour overlap check with a naive test that simply compares each edge of the first polygon to the each edge of the second polygon. While there are few incorrectly assessed configurations, such as a case where all vertices of the second polygon reside on the edges of the first polygon, we consider these cases very rare and have chosen to ignore their presence

here and to handle the any potential artifacts later in the asset pipeline.

To validate if a point is inside a polygon defined as a closed contour with directed edge segments, one may simply cast a ray in a desired direction and analyze intersections with the edge segments. Whether the point is inside can then be seen from the direction of the nearest intersecting edge. But for a large sets of points, this is wasteful. In order to do this efficiently, for each shape we precompute a list of "left" and "right" edges such that left edges have raising Y-coordinate and right edges have falling Y-coordinate, and sort this by the highest edge Y-coordinate. Now, each point inside the polygon must have left edge on its left side and right edge on its right side. To validate if a set of points is inside the polygon, we consider the cases of convex and non-covex polygons separately. For convex polygons, we sort the test point set by Y-coordinate and iterate them while walking sorted point set and the sorted left and right edge lists in ascending Y order as show in Figure 9. Each point needs is only validated against the maximum of two edges on the horizontal band, the ones following the vertices last visited, improving the performance significantly. For non-covex polygons, there might be multiple left and right edges on the band and we fall back to validating all edges, early-rejecting with the Y-coordinate where possible.

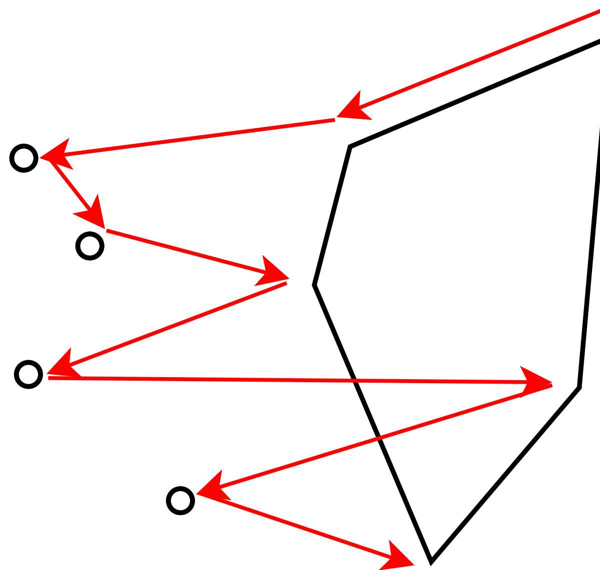


Figure 9: Edge and point walk order in convex points-in-poly check.

For polygon distance computation, we use similar strategy as with points intersection test. For horizontal distance, we separate edges into left and right edges, and for vertical distance we separate the top and bottom edges to separate sorted lists. Next for convex polygons, we walk a desired edge list and it's opposing edge list in order, testing distance at each vertex to the edge following the previous opposing vertex as shown in Figure 10. And as with the point overlap test, with non-convex polygons we fall back to validating all vertex distances with appropriate early rejections.

The polygon overlap resolve is essentially a dual of the distance computation. Instead of asking far much a polygon can be moved on certain axis before it will

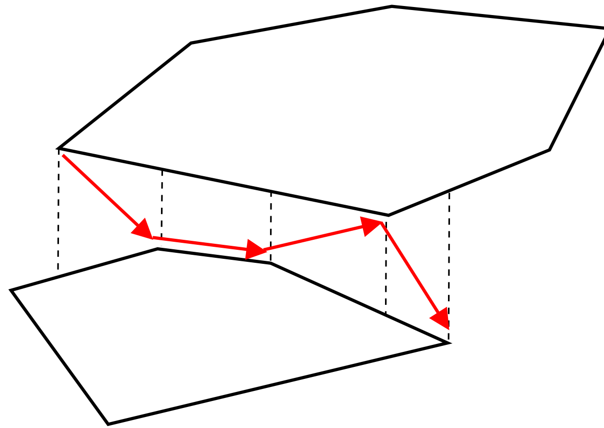


Figure 10: Polygon distance test vertex and edge walk order.

overlap the second polygon, we ask how far a polygon needs to be moved on certain axis so that it will no longer overlap the second polygon. Naturally, we solve it with a similar approach as the distance but we simply select opposing edge lists and negate the resulting distance.

With NFP representation, the most relevant operations are the NFP generation but also the polygon overlap detection and computation of distances and resolved. For NFP generation for shapes A on B, we choose the reference point of A as the origin in the original sprite asset. Since this point represents a corner in the original sprite image, it is commonly outside the bounding polygon generated for the image. Hence our reference point of A is commonly outside the shape A itself. We believe this choice of a non-vertex reference point is uncommon, but this approach streamlines implementation significantly. As the polygon overlap detection is technically a point-in-polygon test, and distance and resolve computations are a single raycast in against a polygon, we reuse the methods described above. For point overlap test, we simply run the points-in-poly test with a single test point, and for distance and resolve computation we simply switch one of the walked lists with a list of one vertex. Instead of walking the opposing lists, we could for convex shapes binary search the clipping edge but we decided to not implement this as our polygons are relative simple and only have a small number of edges.

Finally, none of the evaluated implementations cache and share the processed shapes over to the following test iterations, potentially resulting in a large amount of duplicated work. In our initial implementations we applied a very aggressive caching strategy in order to emphasize on packing time and to ignore the somewhat constant preprocessing time. For geometric methods, we cached the original input shapes fattened with the NFP of the texture filtering area, for bitmap methods we cached the conservatively rasterized bitmaps for each orientation encountered, and for NFP approaches we cached the generated NFPs they were needed. For small validation input samples this approach worked seemingly fine, but for with real production content this approach was observed to cause excessive memory usage and became unusable in our production pipeline. In particular with the NFP caching where each NFP is unique for each pair of shapes and for each combination of the orientations,

the memory pressure grew so high the test process got consistently terminated on the test computer with 16GB of RAM. As selectively disabling caching for some methods would not result in meaningful packing time comparison, and since our input shapes are all essentially unique, we have disabled geometry caching in all test variants. While we expect this to cause increased packing time, we expect the increase to be unbiased and compute times to remain mutually comparable.

4.4 Evaluation of packing density

We evaluate the packing density achieved with various strategies presented earlier in the paper. This section is structured as follows: first we describe and motivate the criteria under which we have chosen the strategies to be evaluated. Then we list out the strategies and the packing results obtained and finally we analyze and discuss the results.

As we have discussed earlier in Section 2, the goal of the game atlas packer is to minimize the number of atlases, i.e. bins, required. As we have also discussed in Section 3.2, the choice of the representation of the geometry while being theoretically and technically interesting does not significantly affect the packing results, provided that the chosen geometry representations are sufficiently accurate. As have previously noted, by choosing a raster density to match the texel density, the raster shapes will represent exactly the texels we are to pack.

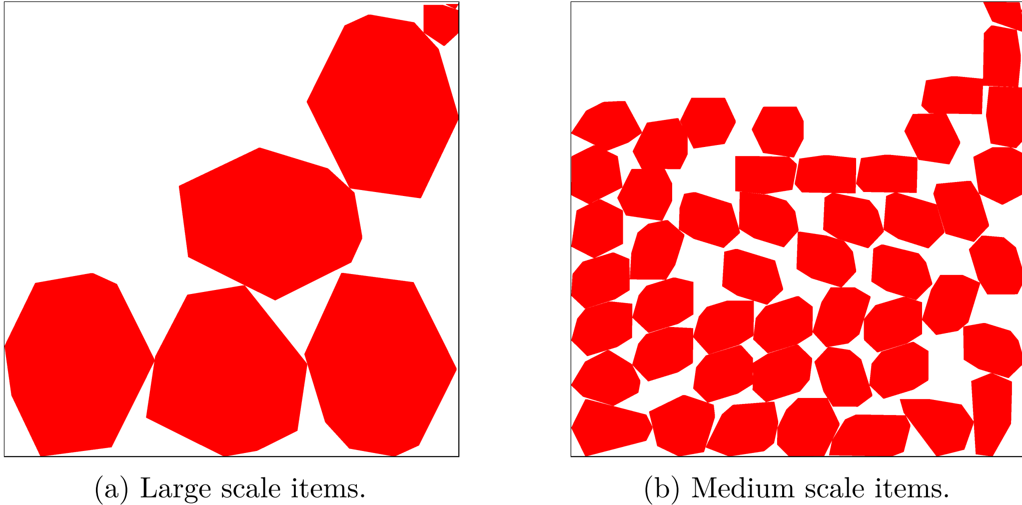
To this background, it is sufficient to evaluate packing solely using in the raster representation. For the packing policies and the scheduling algorithms, we use methods chosen and justified in Section 4.1. In summary, for packing policies, we select Strict BL, defined in Section 3.3.2, BL Procedure, defined in Section 3.3.3, Two-Step BL Procedure, defined in Section 3.3.4, and BL Resolving, defined in Section 3.3.5. For scheduling we use SCH, Aggressive SCH (aSCH), and PBP which we defined in Section 3.4. This combination of 4 packing policies and 3 scheduling strategies produces in total of 12 different configurations we test. The packing results for these configurations are shown in Table 1 in which we report both the number of bins required and the f density metric we discussed in Section 4.1.

	Number of bins			f		
	SCH	aSCH	PBP	SCH	aSCH	PBP
Strict BL	19	16	16	0.390643	0.561051	0.615372
BL Procedure	25	22	23	0.237074	0.300647	0.289098
Two-Step BL Procedure	25	19	19	0.242248	0.380350	0.414519
BL Resolving	19	16	16	0.380070	0.560101	0.611696

Table 1: Packing density with packing methods.

These results are not surprising and multiple clear patterns can be seen. Firstly, the we note that the BL Procedure produces consistently low quality results. As expected the BL Procedure due to its limited visibility to the solution space commonly chooses a very low quality local optimums resulting in low quality total packing. As

we inspect the results, we observe a “tower-building” behavior clearly emerging in the bins. As show in Figures 11(a) and 11(b), the BL Procedure commonly gets stuck in the slopes of irregular shapes and a very high position is chosen. When this is repeated, a tower emerges in the top right corner eventually blocking the top right position, preventing any further items to be placed. This naturally results in low packing density.



(a) Large scale items.

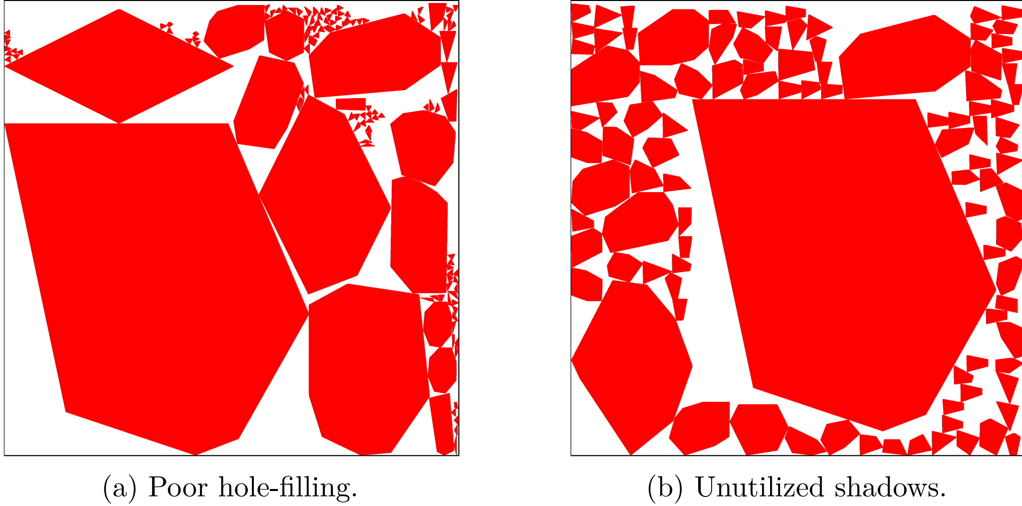
(b) Medium scale items.

Figure 11: Tower-building in BL Procedure.

Also as shown in the figures, the tower-building behavior seems consistent with multiple relative shape sizes. Additionally we have observed a two-towered variant, in which a secondary tower is built on the left edge of the bin. Finally, we note that similar tower-like structures are sometimes generated not only on bin edges but also in the central areas of the bin. All in all it is clear that BL procedure is very finicky and very likely to produce low density packing configurations. While we believe it is possible to select a shape order on which these artifacts would not manifest, we also belive that finding such an order to be extremely hard and that it’s unlikely that any reasonable local search scheme could make BL Procedure a practical placement method for tasks that require high packing density.

A second observation is the relatively poor packing density with Two-Step BL Procedure. While the variation we defined performs better than the traditional BL Procedure and is able to perform coarse-level hole-filling, this limited hole-filling is not in many cases sufficient. As in Figure 12(a), we note that the coarse bounding-box level hole-filling leaves many true packing opportunities unutilized. Additionally as we see in Figure 12(b), even while coarse hole-filling performs adequately, the position refining is bound to the limitations of the traditional BL Procedure. This we observe causes visible “shadows” of unused space on the left sides of the large shapes as the refinement procedure does not consider potential position improvements on the right side of the current location. These traits in combination result in the low packing density.

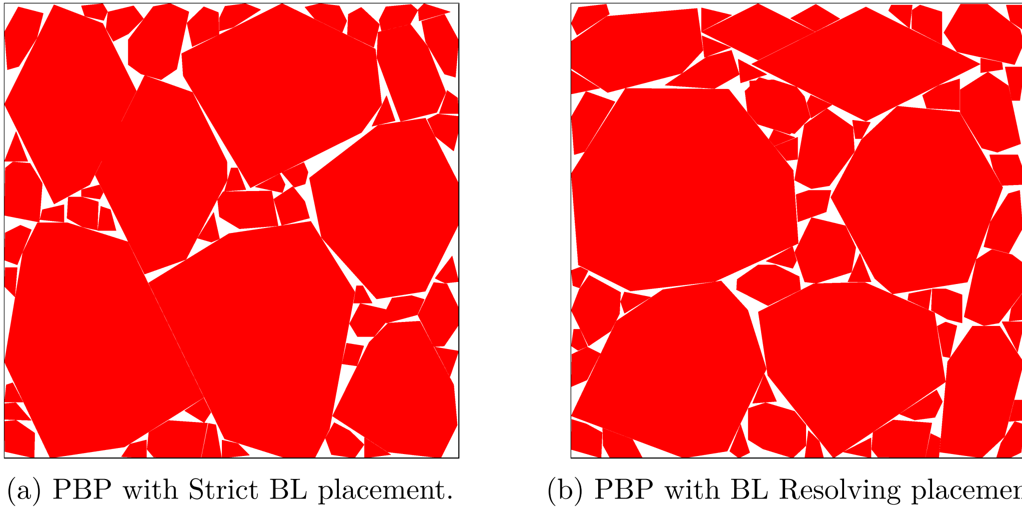
For the other placement policies, the Strict BL and the BL Resolving perform well. As we see in Figures 13(a) and 13(b), both the Strict BL and the BL Resolving



(a) Poor hole-filling. (b) Unutilized shadows.

Figure 12: Low-quality hole-filling with Two-Step BL Procedure.

respectively are able to fill bins with very little free space left. By looking the densities shown in Table 1, we note that the Strict BL produces consistently the highest density results while BL Resolving is very close second producing only slightly worse placements. In particular with aSCH scheduling, the differences in density are too small to draw any conclusions. This similarity in results is somewhat surprising as it implies that the potentially weak hole-filling characteristic of BL Resolving do not manifest in practice, or at least with not with our test content.



(a) PBP with Strict BL placement. (b) PBP with BL Resolving placement.

Figure 13: High-quality layout with Strict BL and BL Resolving placement policies.

In the results we also note that SCH scheduling is consistently producing weaker results than the alternatives. This is expected since as discussed earlier the SCH only keeps a single bin open at a time and closes the bin on the first placement failure. This makes the SCH very sensitive to the packing order as even a single non-optimally ordered shape can cause completion of a bin even if the bin still had

large unutilized areas available. As with BL Procedure, we believe that finding a well-behaving shape order for SCH is possible but very hard. While a local search solution can be presumed to be able to find a reasonable shape insert order, we expect such a local search to require a very large number iterations making any real world use impractical.

While SCH is significantly faster as will be discussed later in Section 4.5 and hence multiple search iterations could be completed by the time an alternative implementation terminates, one could argue that for an unbiased comparison the time saved by SCH should be used for refining the results more local search. This way the comparison would be of different strategies within a similar time budget. For an another use case this would make sense, but however as we stated in Section 4.1, we consider this overreliance on local search as an indicator of flakiness. As we discussed earlier, masking this flakiness would not be meaningful for us.

Finally we see that in general PBP produces highest packing quality. While aSCH reaches to the equal amount of bins, the PBP has provides consistently significantly higher density metric than the aSCH. However, there is an anomaly with the BL Procedure in which case the situation is reversed. We assume this anomaly is due to an interaction with the tower-building in BL Procedure and the general preference of packing large objects first in PBP. As we saw in Figure 11(a), a fatal towel can be formed by just a few large objects. With this observation we speculate that by prioritizing the large object in the first bin, PBP guides the shape selection towards the pathological configurations BL Procedure will handle poorly. We conclude that this combination of BL Procedure packing and PBP scheduling should be avoided.

4.5 Evaluation of packing performance

In this section we evaluate the performance of the various packing methods discussed previously. We first define and justify the selection criteria for packing methods tested, then we lay out the obtained test results. We then analyze the results on the three separate axes: first on the choice of the scheduling algorithm, then on the choice of the geometry representation and finally on the choice of the placement policy. After this analysis, we conclude the section by relating the obtained compute time results with the packing densities achieved in the previous section and discuss its implications.

As we stated earlier in Section 2, our main goal is to find a set of suitable irregular packing methods that produce both high-density results and can be executed in a reasonable time. To this end it is clear that evaluating the CPU time consumed by a method that is known to produce unacceptably low density results would not be meaningful. As we have realistic evaluation results for the achievable packing densities available shown in Section , we have decided to exclude some configurations from the performance evaluation that were deemed unfit for production use.

First as we noted, the BL Procedure produces consistently very low quality bins regardless of the scheduling used. Likewise the SCH scheduling exhibits similar issues resulting in very underutilized bins. Hence it is not meaningful for us to evaluate their performance any further with different geometry representations as the results have

already shown to be unworkable for our use case. We evaluated the computation times for Strict BL, Two-Step BL Procedure, and BL Resolving placement policies using Raster, NFP and Direct Geometry representations with aSCH and PBP scheduling for which the results are shown in Table 2. Additionally we also report the compute times obtained during the packing density tests in Section 4.5.

		SCH	aSCH	PBP
Strict BL	NFP	-	839	699
	Raster	156	1060	1691
BL Procedure	Raster	550	502	609
Two-Step BL Procedure	Direct Geometry	-	61	121
	NFP	-	58	122
	Raster	42	116	164
BL Resolving	Direct Geometry	-	3005	2605
	NFP	-	2634	2428
	Raster	567	6501	5241

Table 2: Time required (seconds) for packing with different scheduling algorithms, packing policies and geometry representations.

We observe multiple interesting phenomena in these result. First, the SCH scheduling is able to complete the test packing tasks significantly faster than other scheduling methods, which was expected, except for the BL Procedure with the Raster representation in which the aggressive SCH variant is surprisingly faster and, as discussed earlier, produces better results. The general high performance confirms our assumption that by closing a bin on first error, the packing algorithm can avoid a lot of futile placement attempts to almost full bins, resulting in faster execution.

For the anomaly with the BL Procedure with the Raster representations, we attribute it to two characteristics of the test case: First, the BL Procedure is able to early reject items to full or almost full bins. If the item does not fit to the initial position that is top right corner, the placement is rejected immediately. Since this rejection test is very efficient, the cost of the futile packing attempts is mitigated. Second, as SCH is more likely to open a new bin than other schedulers, the bins in which items are placed into are less full. Since the BL Procedure functions by moving shapes down and to the left, less full bins will result in more and longer shifts. In raster representation, a longer shift will require longer test probes and larger areas to validate resulting in larger time consumption. This is issue is additionally exaggerated due to, as we have described, our raster geometry operation implementations not being highly-optimized.

With these assumptions, we are able to explain the timing difference in BL Procedure between SCH and aSCH, and aSCH and PBP, as in both cases the faster is the one producing more dense packing. However, this does not apply between SCH and PBP, in which SCH in is faster but produces less dense bins. We attribute this exception to the exception to the cost of computing the approximate Knapsack Assignment required in PBP and to the characteristics of BL Procedure. As the BL Procedure produces low density placements in the bin, at each step the PBP

is executed the remaining bin capacity will be very large and not representative of the actual usable space. This causes our approximate Knapsack Assignment algorithm to consume more time as the search cannot be bound and pruned as early as with more realistic capacity estimation. This becomes catastrophic in the case of "tower-building" where this assignment task essentially becomes the hot path of the test code: after a tower has been build, the remaining capacity of the bin is still large but the true capacity of a bin is very close to zero. Hence the PBP will select a large set of items to be placed with the Knapsack Assignment algorithm, but when the first is inserted, it will very likely fail. Thus the first item of the set is removed, and the process is repeated until all items have attempted and failed to be placed. As a result, a large amount useless computation is performed.

While the SCH scheduling was always the fastest in our test set, similar clear cut rule does not exist for the aSCH and PBP. For the both BL Procedure and the Two Step BL Procedure we observe PBP being significantly slower and in the case of Two Step BL Procedure, the PBP requires at worst over two times compute time. But interestingly with the BL Resolving, the opposite can be observed and PBP is observably faster. Finally the Strict BL reports conflicting numbers where NFP implementation greatly benefits from the PBP but the Raster implementation is significantly faster on aSCH. While the results are seemingly contradictory, we believe these numbers consistent and can be attributed to the various characteristics of the each tested configuration.

With the BL Procedure and the Two-Step BL Procedure, we observed that PBP consumes more time than the aSCH variant. The PBP algorithm tries to maximize the bin value by selecting the most valuable but potentially fitting items and as such can be interpreted as an early rejection mechanism to the placement step. The aSCH however does not have any rejection mechanism and it instead tries to place each item unconditionally whether it can fit into the bin or not. Hence it is clear that within a single bin, the placement attempts emitted by the PBP must be a subset the ones emitted by the aSCH and hence the aSCH variant will be performing more placement attempts. Since the PBP does less placement work than aSCH, but is still slower, it suggests the observed difference in the consumed time must stem from some other component of the tested configuration.

We attribute this behavior to two characteristics of the tested system: first, the both BL Procedure and Two-Step BL Procedure are able to perform very efficient early-rejection of items and thus cost of the futile placement attempts by aSCH are completely mitigated. Second, the PBP scheduling requires solving maximal Knapsack Assignment problems. We only approximate the solution to prevent catastrophic performance as we discussed previously in Section 4.3, but even the approximation is not free, unlike the aSCH scheduling. While the PBP produces a different packing layout than aSCH, it could be possible that the intermediate layouts with PBP are somehow more expensive to the placement policy or the geometry representation or otherwise indirectly negatively affect other components of the system, we do not believe this is the case. As we observe the results with different geometry representations on the Two-Step BL Procedure, we note that the time increase from aSCH to PBP scheduling is rather constant. In our interpretation, this

rules out any possibility of an indirect impact on certain geometry representations and other interference. All in all, we believe the performance drop as observed with PBP scheduling is mainly caused by the PBP computation and that this cost is fundamental to the algorithm.

For the BL Resolving class of packing algorithms, we observe PBP being faster than aSCH. This is not unexpected as the BL Resolving is unable early-reject items but instead each inserted item must traverse trough the bin in BL order for it to be conclusive rejected. As the PBP is able to provide this early-rejection of items that surely cannot be fit, it is clear that some compute time can be saved. But interestingly, and unlike with Two-Step Procedure, the absolute and the relative performance delta varies between different geometry representations, which suggests an additional interaction between the components. We suspect that this interaction manifests through the different resulting packing layouts with the PBP and the aSCH. As we saw in Table 1, the PBP produces results in significantly higher f density with the BL Resolving class of algorithms than the aSCH. Since the f density is higher, the amount of free, unused space between the placed items is smaller, which implies a tighter packing layout. And since the layout is tighter, we suspect that each resolution step in the BL Resolving is more likely to result in shorter resolution distances. As the resolution distances on average become shorter, it will naturally take larger number of resolution steps to either find a valid position or reject the inserted item. With these assumptions the seemingly inconsistent performance gain observed with PBP can be explained.

Finally for the Strict BL class of algorithms, we observe contradictory results: when using the NFP geometry representation, the packing process is observably faster with PBP than with the aSCH scheduling. But when the packing is performed in Raster bitmap representation, the PBP is suddenly consuming about 60% more time than the aSCH. We expect this is caused by a similar interaction as with the BL Resolving: The Strict BL cannot early-reject placed items, so the rejection mechanism provided by PBP increases the packing performance. At the same time however, the PBP with its item selection logic is able to produce significantly tigher layouts as we saw in the previous section. While with the NFP this increase in density does not observably seem to cause any issues, it is catastrophic on the raster model. We suspect, similarly as with the BL Resolving, that by increasing the density our bitmap, the early-rejection test probes we employ as described in Section 4.3 will on average become shorter. As we are able to rule out fewer positions, more bitmap intersection tests need to be employed explaining the low performance in the raster variant.

For the choice of geometry, we see also interesting patterns: The time required for a raster approach is usually around the double of the geometric methods. Even though our raster geometry operation implementations were not highly-optimized and lower-than-optimal performance was expected, these results were a surprise. We suspect that the relative impact of the unoptimized raster methods is amplified by our rather high rasterization resolution, the quite low polygonal complexity of the shapes and our comparatively highly-optimized polygonal geometry implementation. As the polygonal shapes had commonly $10 - 20$ vertices after the texture filter boundary

extrusion, a lot fewer operations need to be taken than the ones needed to validate the hundreds of cells of the rasterized shapes. Even though the raster operations are a lot simpler and faster, it seems that the lower number of higher-complexity operations is beneficial for our test content. Additionally, as most of the shapes are convex, we also often take the optimized fast paths in the polygonal code, resulting in even larger difference. We expect that we could reach the performance parity by optimizing the raster implementation, but we also expect to be able to improve the performance of the polygonal methods on many fronts. From these results it is clear to us that the raster representation is not very good a fit for our high-resolution, low-complexity test content.

For using NFP or Direct Geometric approach the choice is not as clear. In Two-Step BL Procedure we see very little difference between the approaches as with aSCH the NFP was only slightly faster while with PBP the NFP was slightly slower. In both cases the difference is less than 5% which we do no consider significant. This non-conclusive result is reasonable since in the Two-Step BL Procedure most of the items will be rejected by the coarse positioning phase even before any NFP or Direct Geometry tests are executed. Hence only a small number of BL Process shifting operations are done, and in both cases, those shifting operations are relatively highly-optimized. While one would expect the cost of generating the NFPs necessary the operations to outweigh the relatively simple polygon directed distance computation in Direct Geometry, this interestingly cannot be observed in the results. We assume this cost of NFP generation is not observable due to the relative simplicity of our shapes, and suspect that with more complex geometry the generation could become significant factor in the total time usage.

However with BL Resolving an anomaly emerges: with both aSCH and PBP scheduling, the NFP approach is significantly faster that the Direct Geometry variant. This is seemingly unexpected since the computationally intensive generation of the NFP and then the simple resolve computation on that NFP should not be faster than the only-moderately-expensive Direct Geometry resolve computation. As the resolve computation is fundamentally the dual of the directed distance computation, both the BL Resolving policy and the Two-Step BL Procedure policy use essentially identical algorithms with similar expected performance. Hence, one would expect the similar behavior to the Two-Step BL Procedure regarding the compute times.

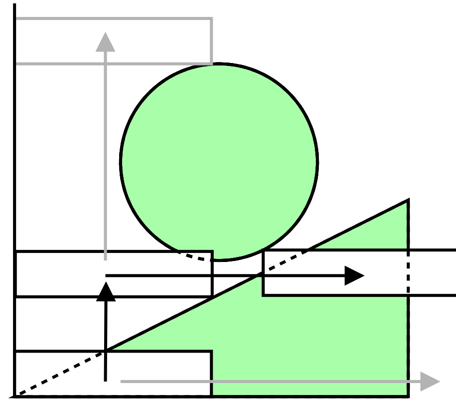


Figure 14: Resolve step re-introducing already resolved overlap.

However, while NFPs are not cached over multiple test instances, they still can be reused within a single placement attempt. While a single resolve step resolves a single shape pair overlap by moving the inserted shape either upwards or to the right the minimum required distance to solve the overlap, any following resolve step can by moving the shape more to the right or

upwards respectively make the original shape pair overlapping again, requiring a new resolve for the original pair, as depicted in Figure 14. For this new resolve, the NFP generated for the initial resolve can then be reused. As such, we attribute this unexpected performance improvement to the NFP reuse and predict that with more aggressive NFP caching strategy, the total compute time could be significantly reduced further.

For the placement policies employed, the results are very consistent and conclusive. Regardless of the geometry representation or the scheduling algorithm, the Two-Step BL Procedure is always fastest with a significant margin. This is not surprising as the coarse placement is very fast and the inner BL Procedure is not slow either. The policy is also able to efficiently early-reject items mitigating the cost of futile placement attempts. Interestingly with the raster geometry, the Two Step BL Procedure was is even faster than the traditional BL Procedure, but we attributed this anomaly to unoptimized quality raster implementation.

Our second fastest placement policy was the BL Procedure. While the policy was not comprehensively tested due to producing consistently low-density packing layouts, with the Raster representation the method was relatively fast. While there was an anomaly with the SCH scheduling which we attributed to our implementation, the compute times in general were low. Due to its simplicity and the ability to early reject placed items, we expect that Direct Geometry and NFP implementation to reach or surpass the compute performance of the Two-Step BL Procedure.

Finally our slowest placement strategies are the Strict BL and the BL Resolving placement policy. As neither method can early-reject input shapes as in the BL Procedure and the Two-Step BL Procedure, it is no surprise that compute times end up being higher due to the futile placement attempts. The impact of these attempts can be roughly estimated from the difference of the SCH and aSCH scheduling methods, and for both BL Resolving and Strict BL this difference is very large. While both methods are similar, and have analogous loop to walk from the bottom left towards the top right in search of a valid position, interestingly the BL Resolving is consistently significantly slower than the Strict BL. This behavior was unexpected and we aim to explain it.

For the raster representations, both the Strict BL and the BL Resolving are very similar on the surface: both walk though the bin in BL order and to select the next horizontal test position, both employ similar pixel probing to rule out certainly invalid positions. For the vertical advancement they differ a little: Strict BL simply selects the next row, while BL Resolving pops a next from its candidate queue which is filled with positions obtained with vertical pixel probes. However, these methods have a fundamental difference in the scope of these steps: the Strict BL is always probing and testing against the total bin layout, while BL Resolving is always by design only testing against a single overlapping shape. This lack of global awareness in BL Resolving explains the significantly higher compute times. As per-item resolutions will always be shorter on average than resolution over multiple shapes, the BL Resolving will need to validate a larger set of potential positions than the simple Strict BL. Additionally, as the Strict BL operates on the global bin layout, we were able to implement the spanlist-based rejection mechanism described

in Section 4.3 which improved the performance significantly. But the same rejection mechanism cannot be trivially ported to BL Resolving as its vertical advancement depends on the vertical resolutions of the failed placement attempts. If all failing placement attempts on high enough a span were avoided, no new candidate positions would get placed beyond this gap halting the search process.

While Strict BL and the BL Resolving policies when operating on a NFP geometry representation do not use similar placement probes as the Raster models, the same lack of global awareness on the part of BL Resolving plagues the method. As each candidate position after the resolve step is only guaranteed to not intersect a single already-placed existing shape, it is clear that each placement process will incur a large number of illegal candidate positions. Even though the Strict BL needs higher-complexity computation to find all potential reference positions on the overlapping NFPs (vertices, NFP intersection points, IFP intersection points), this process results in higher quality, less likely to be invalid positions. Additionally, after all reference points have been tested and proved invalid on a certain shape, it is conclusively proven that the inserted shape cannot share a boundary with the shape. This allows us to reject any further positions siding that shape, saving us slight amount of compute. The same rule does not apply to the BL Resolving where subsequent resolve steps may reintroduce already resolved overlap pairs as we saw in Figure 14. All in all, this higher complexity, but better quality approach in Strict BL prevails.

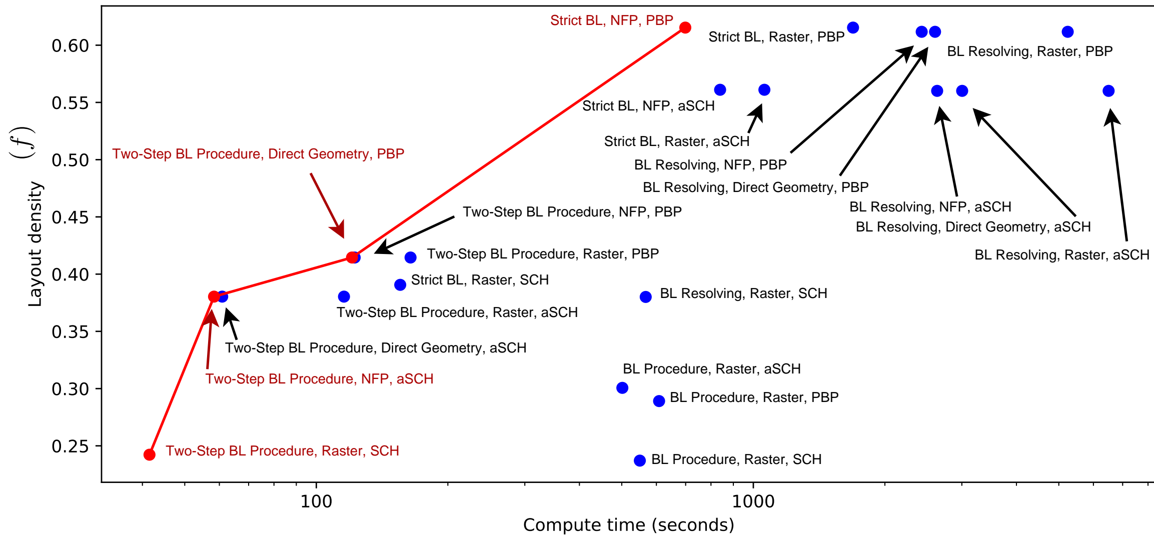


Figure 15: Compute time and layout density with different packing algorithms. Pareto optimal frontier in red.

Ultimately, while performance implications of each packing algorithm component and the absolute total performance of each packing algorithm are interesting, the performance only becomes meaningful when evaluated against the packing quality they are able to generate. In Figure 15 we show the obtained packing quality measured with the f density as a function of the average compute time for all 22

packing algorithms evaluated in this paper. We mark with red the algorithms for which no single faster and denser algorithm exists, which forms the pareto bound for our objective of fast yet dense packing method. As the compute times vary significantly in their magnitude, have chosen to use logarithmic scale for the time axis. While this distorts the time differences by exaggerating the differences between the fast methods and diminishing the differences between the slow methods, using this logarithmic scale did provide an overall better readability.

In this graph we immediately see multiple interesting patterns. First, the BL Resolving variants almost entirely occupy the top right corner of the graph, far away from the red line. As we have discussed before, the BL Resolving placement policy is based on the idea of chaining pairwise overlap resolutions until a valid position is found. With our results, it is apparent that this approach does not scale well, at least for our test content. While providing top-class results, it comes with an unreasonably high computational cost: as we see in the graph, for any packing quality level, a faster method exists. And conversely since Strict BL produced slightly denser layouts, for any compute time budget, a more dense algorithm exists. In conclusion the BL Resolving occupies a space where it is not best in any class, and hence we cannot recommend it for any polygonal packing task.

Then, we observe a cluster of BL Procedure variants near the bottom center of the graph. While we earlier excluded its variants from any further investigations, it was done on the basis of the low packing density inherent to the method. But by looking at the graph, it is clear that while producing low quality results, the algorithms itself is also very slow. While some of the performance issues were be attributed to our unoptimized implementation, we don't expect it to become competitive with our Two-Step BL Procedure variant with any level of optimizations. Like the BL Resolving, the BL Procedure occupies a space in which faster and higher-quality methods exists. As such, we don't see any meaningful use for this placement policy.

Next by looking along the red optimality front, we observe the Two-Step BL Procedure variants on the high-performance end and the Strict BL variant on the high-quality end. In general, the Two-Step BL Procedure appears very successful strategy and able to secure the highest performance position on many quality levels. An especially interesting observation is the relation to the scheduling algorithms used: At the highest performance variants, SCH is used. At higher quality level, aSCH is used, and finally for the highest possible quality level, PBP is chosen. Due to this wide spectrum, at low to medium computational budgets the Two-Step BL Procedure has unparalleled quality, and only at the higher computational budgets it is eventually superceded by the Strict BL. Notably, these datapoints are separated by a wide performance and packing quality gap. From a developer perspective, this gap is problematic as it limits the controllable tradeoff of the speed and the quality. For even a slight quality increase, the compute budget must be increased immensely.

Additionally in the graph we note The Two-Step BL Procedure seems very flexible on the geometry representation used as in both cases the Direct Geometry and the NFP variants have very similar performance. This robustness can be very beneficial when selecting a packing algorithm but the content characteristics are not yet completely known, as can be the case in game development. As computing the

NFPs can become very complicated and error-prone process with certain shape pairs, committing and investing into a packing method is less risky if the uncertain representation can be switched with only a little effort and with only a small performance cost. For example with the Strict BL this is not the case and the performance is very dependent on the representation used. While the significantly higher packing quality might in certain scenarios be worth it, the high dependence on the NFP representation casts doubts on the future-proofness and reliability of the method.

5 Conclusions

In this paper we have defined the need for hardware-accelerated rendering for mobile games and the limitations and their common workarounds of graphics hardware on the mobile devices presently on the market. We have also discussed the external requirements caused by various development and distribution related aspects inherent to mobile game industry, and we have then induced these needs and requirements into a well defined Irregular Multiple Bin Packing problem with the objectives of both minimizing the compute time and the number of bins required. We have reviewed and adapted various approaches in Cutting and Packing literature and introduced two new packing variants of existing algorithms to perform Irregular Multiple Bin Packing. Then we introduced the additional modifications necessary for the resulting packing layouts to conform to our graphics-hardware based requirements, and finally we evaluated 22 different packing algorithms for both the achieved packing density and the average compute time using real mobile game assets as the test set. These results varied significantly on both quality and performance axes.

The first introduced new cutting and packing variant, the Two-Step BL Procedure using Strict BL on axis-aligned bounding boxes for high-level coarse positioning and the classical BL Procedure for position refining, performed generally very well. In almost all cases, it was faster than any other packing method, and especially with low computational budgets it provided very competitive layout densities. This result holds promise for all hierarchical placement algorithms. Additionally, the method was shown to be very tolerant to the underlying geometry implementation, making this method both flexible and reliable, increasing the confidence in its future-proofness.

The second introduced variant, the BL Resolving with both horizontal and vertical resolution probes, was not as successful. While producing high quality results, the computational costs were significantly larger than with any other evaluated method. While the method is approachable, can be reasoned about, and is fairly robust with the geometry implementation, the prohibitive high cost makes it unusable for any production use.

Surprisingly in our evaluation the classic Cutting and Packing tools SCH scheduling and the BL Procedure placement produced both very low quality layouts, albeit with a high performance. We assume this result is specific to our test scenario: in this paper, all the evaluations were performed on real mobile game assets which have a very peculiar size distribution. As the portion of very small items is very large, it might violate the implicit assumptions of various packing methods potentially resulting in lower performance or layout density than with a more traditional, "well-behaving" test set. However, there were no observable anomalies with the very traditional Strict BL placement policy.

Even still, our evaluation differs on many aspects from the evaluations in the literature. Our modifications to comply with realtime rendering constraints are very intrusive and may cause significant quality and performance degradation. While our texel-fitting process can be assumed to have been relatively harmless, the rollback and reorder implemented on the scheduling level to enforce the affinity group item placement to same bin complicates the schedulers significantly and

may cause unforeseen interactions. Additionally due to our general mistrust of the consistency of randomized algorithms, in our evaluations the amount of local search was limited by the number of iterations and not by total time. This naturally benefits slower, higher-quality algorithms and punishes fast, more fickle methods, as was our intention. In environments where the consistency is not as critical as in our use case, more laxed local search approach could alter the results greatly.

On the performance front, we were able to identify the early rejection in the bin packing process as the key factor for the high performance. In general, the methods utilizing some form of rejection mechanism, be it inherent to the placement policy or be it provided by the scheduler, were significantly faster than those without. Especially in the case of PBP, much was gained even though the cost necessary for solving the PBP problems was shown to be non-trivial. This is impressive since we suspect the PBP to have performed less-than-optimally due to the large number of very small items. In this case we expect the remaining bin area capacity to become vastly larger than the actual usable area, essentially preventing any meaningful the item rejections.

As our highest-quality placement methods in the evaluation do not possess the ability to early reject items, this ability of rejecting items during the scheduling becomes very interesting: the results obtained with PBP shows that rejection decisions based even on poor metrics are are better than none. As such, we believe the investigation for advanced item scheduling and rejection methods warrants future research. An interesting question is the sensibility of surface area based capacity metrics in general. As the surface area is very lossy approximation, it would seem reasonable to assume higher-complexity, geometric properties would provide significantly higher-quality rejection scheme. An alternative but as interesting path would be to ignore ahead-of-time the shape approximations altogether but instead rely on placement success or failure feedback to train a probabilistic filter at runtime.

An additional avenue for future research is due to the relative good general performance and quality attained with the hierarchical Two-Step BL Procedure. By improving the compressing positioning step for to avoid the “shadows” or by partitioning the large pieces into sets of bounding boxes to work around the poor hole filling, it is likely that the quality could be significantly improved with a minor, controllable performance impact. This approach by improving the layout density could fill the wide performance and quality gap observed between the Two-Step BL Procedure and the Strict BL, providing developers with more options for time and quality bound packing tasks.

References

[1] 3dfx Interactive. Develop 3dfx. https://web.archive.org/web/19990831064222/http://www.3dfx.com:80/view.asp?PAGE=nusdeveloper, 1999. Accessed: 2018-08-12.

[2] Advanced Micro Devices, Inc. Mantle White Paper. Technical report, 2014.

[3] R. Alvarez-Valdes, A. Martinez, and J.M. Tamarit. A branch & bound algorithm for cutting and packing irregularly shaped pieces. *International Journal of Production Economics*, 145(2):463–477, 2013.

[4] Apple Inc. Metal framework. https://developer.apple.com/documentation/metal. Accessed: 2018-08-12.

[5] Richard Carl Art Jr. *An approach to the two dimensional irregular cutting stock problem.* PhD thesis, Massachusetts Institute of Technology, 1966.

[6] Brenda S. Baker, Edward G. Coffman Jr., and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855, 1980.

[7] Julia A. Bennell and Xiang Song. A beam search implementation for the irregular shape packing problem. *J. Heuristics*, 16(2):167–188, 2010.

[8] Edmund K. Burke, Robert S. R. Hellier, Graham Kendall, and Glenn Whitwell. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research*, 54(3):587–601, 2006.

[9] Edmund K. Burke, Robert S. R. Hellier, Graham Kendall, and Glenn Whitwell. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research*, 179(1):27–49, 2007.

[10] Alberto Caprara and Ulrich Pferschy. Worst-case analysis of the subset sum algorithm for bin packing. *Oper. Res. Lett.*, 32(2):159–166, 2004.

[11] Luiz H. Cherri, Leandro Resende Mundim, Marina Andretta, Franklina Maria Bragion Toledo, José Fernando Oliveira, and Maria Antónia Carravilla. Robust mixed-integer linear programming models for the irregular strip packing problem. *European Journal of Operational Research*, 253(3):570–583, 2016.

[12] KA Dowsland, WB Dowsland, and JA Bennell. Jostling for position: Local improvement for irregular cutting patterns. *Journal of the Operational Research Society*, 49(6):647–658, 1998.

[13] Kathryn A Dowsland and William B Dowsland. Solution approaches to irregular nesting problems. *European Journal of Operational Research*, 84(3):506–521, 1995.

[14] Kathryn A. Dowsland, Subodh Vaid, and William B. Dowsland. An algorithm for polygon placement using a bottom-left strategy. *European Journal of Operational Research*, 141(2):371–381, 2002.

[15] drawElements. drawElements Market Analysis - Free version, 2013. Version 1.1.

[16] Ahmed Elkeran. A new approach for sheet nesting problem using guided cuckoo search and pairwise clustering. *European Journal of Operational Research*, 231(3):757–769, 2013.

[17] Facebook, Inc. Zstandard software. https://facebook.github.io/zstd/. Accessed: 2018-08-12.

[18] Simon Fenney. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 84–91. Eurographics Association, 2003.

[19] Kikuo Fujita, Shinsuke Akagi, and Noriyasu Hirokawa. Hybrid approach for optimal nesting using a genetic algorithm and a local minimization algorithm. In *Proceedings of the 19th annual ASME design automation conference*, volume 1, pages 477–484. ASME, 1993.

[20] The Khronos Vulkan Working Group. *Vulkan - A Specification*. The Khronos Group Inc., 7 2018. Version 1.1.82, 2018-07-30 10:29:28Z.

[21] Wei Han, Julia A. Bennell, Xiaozhou Zhao, and Xiang Song. Construction heuristics for two-dimensional irregular shape bin packing with guillotine constraints. *European Journal of Operational Research*, 230(3):495–504, 2013.

[22] Imagination Technologies Limited. *IMG_texture_compression_pvrtc*. Version 1.3, 20 September 2012.

[23] Stefan Jakobs. On genetic algorithms for the packing of polygons. *European journal of operational research*, 88(1):165–181, 1996.

[24] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, M. R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[25] Bonfim A Junior, Plácido R Pinheiro, and Rommel D Saraiva. A hybrid methodology for tackling the irregular strip packing problem. *IFAC Proceedings Volumes*, 46(7):396–401, 2013.

[26] The Khronos Group Inc. *KHR_no_error*. Version 6, February 25, 2015.

[27] The Khronos Group Inc. *OES_compressed_ETC1_RGB8_texture*. April 24, 2008.

[28] The Khronos Group Inc. *The OpenGL ES Shading Language*, 5 2009. Language Version: 1.00, Document Revision: 17.

[29] The Khronos Group Inc. *OpenGL ES Common Profile Specification*, 11 2011. Version 2.0.25.

[30] The Khronos Group Inc. *OpenGL ES*, 11 2016. Version 3.0.5.

[31] The Khronos Group Inc. *OpenGL Core Profile Specification*, 4 2018. Version 4.6.

[32] Stephen C. H. Leung, Yangbin Lin, and Defu Zhang. Extended local search algorithm based on nonlinear programming for two-dimensional irregular strip packing problem. *Computers & OR*, 39(3):678–686, 2012.

[33] Eunice López-Camacho, Gabriela Ochoa, Hugo Terashima-Marín, and Edmund K. Burke. An effective heuristic for the two-dimensional irregular bin packing problem. *Annals OR*, 206(1):241–264, 2013.

[34] A Martinez-Sykora, Ramón Alvarez-Valdés, Julia A Bennell, R Ruiz, and José Manuel Tamarit. Matheuristics for the irregular bin packing problem with free rotations. *European Journal of Operational Research*, 258(2):440–455, 2017.

[35] Antonio Martinez-Sykora, Ramon Alvarez-Valdes, Julia Bennell, and Jose Manuel Tamarit. Constructive procedures to solve 2-dimensional bin packing problems with irregular pieces and guillotine cuts. *Omega*, 52:15–32, 2015.

[36] Microsoft Corporation. What is Direct3D 12? https://docs.microsoft.com/en-us/windows/desktop/direct3d12/what-is-directx-12-, 2018. Accessed: 2018-08-12.

[37] Victor Milenkovic, Karen Daniels, and Zhenyu Li. *Placement and compaction of nonconvex polygons for clothing manufacture*. Citeseer, 1992.

[38] Willard Miller. *Symmetry groups and their applications*, volume 50. Academic Press, 1973.

[39] Anthony Morris and Andrew Hayden. Improvements for smaller app downloads on Google Play . https://android-developers.googleblog.com/2016/07/improvements-for-smaller-app-downloads.html, 2016. Accessed: 2018-04-07.

[40] José F Oliveira, A Miguel Gomes, and J Soeiro Ferreira. TOPOS – A new constructive algorithm for nesting problems. *OR-Spektrum*, 22(2):263–284, 2000.

[41] Android Open Source Project. Dashboards - OpenGL version . https://developer.android.com/about/dashboards/index.html, 2018. Accessed: 2017-03-31.

[42] Android Open Source Project. Mobile Hardware Stats. https://developer.android.com/google/play/expansion-files.html, 2018. Accessed: 2018-04-07.

[43] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[44] Steven Skiena. *The algorithm design manual*. Springer, 1997.

[45] Unity Technologies. Mobile Hardware Stats. https://web.archive.org/web/20170729230854/http://hwstats.unity3d.com:80/mobile/gpu.html, 2017. Accessed: 2017-03-25.

[46] Franklina MB Toledo, Maria Antónia Carravilla, Cristina Ribeiro, José F Oliveira, and A Miguel Gomes. The dotted-board model: a new mip model for nesting irregular shapes. *International Journal of Production Economics*, 145(2):478–487, 2013.

[47] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.