Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Jakub Wida

Nr albumu: 1113470

Losowe upakowania układów złożonych z dysków z wykorzystaniem kart graficznych

Praca magisterska
na kierunku Informatyka Stosowana

Praca wykonana pod kierunkiem
dr. hab. Michała Cieśli
z Zakładu Fizyki Statystycznej

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....	.....
Kraków, dnia	Podpis kierującego pracą

Chapter 1

Introduction

TODO make this at the end, when all else is done

Contents

Chapter 2

Problem Overwiev

2.1 Random Sequential Adsorbtion

Random Sequential Adsorbtion is a stochastic process, that can be described as sequential insertion of given shapes onto an empty, limited euclidean space. The shapes are inserted with random position, and in such a way, that if the shape is colliding with an already inserted shape, it is then rejected in the process. The shapes and the space may be defined as having one or more dimensions, with any kind of inserted shape type. The shapes may be generated with random coordinates within given dimensions, as well as random rotation - although this may be optional.[1][2]

2.1.1 Shape Types

The inserted shape that were previously studied include rectangles, squares and other polygons, cubes and hypercubes. Also, the most basic shape usually investigated is a sphere - up to eight dimensions. Other studies involved spheroids, hyperdisks and, what is going to be further investigated in this thesis, disk polymers, or simply two dimensional groups of disks. In case of non spherical shapes, the shape angle can be randomly generated.[3]

2.1.2 Applications

The Random Sequential Adsorbtion can be applied to multiple problems. The process as well as the generated shapes can be used to model a variety of phenomena. These include the ion implantation in semiconductors, structure of the cement paste, particles in cell membranes, protein adsorption and settlement of animal territories.[4] In general, it can be used to model

[1] G Zhang. "Precise algorithm to generate random sequential adsorption of hard polygons at saturation". In: *Physical Review E* 97 (Mar. 2018). DOI: 10.1103/PhysRevE.97.043311, p. 1.

[2] Jens Feder. "Random sequential adsorption". In: *Journal of Theoretical Biology* 87.2 (1980), pp. 237 –254. ISSN: 0022-5193. DOI: https://doi.org/10.1016/0022-5193(80)90358-6. URL: http://www.sciencedirect.com/science/article/pii/0022519380903586, p. 1.

[3] Zhang, "Precise algorithm to generate random sequential adsorption of hard polygons at saturation", op. cit., p. 1.

[4] Ibid., p. 1.

tightly - but still randomly - packed particles.

A significant example is the Poisson Disk Sampling, which is a process of selection of points in subdomain, in such a way that within a given distance from the selected point, no other are taken. This distribution is used in computer graphics for rendering, texture generation, and more. In ray tracing, it is used to create soft shadow, motion blur and the depth of field. In physics, it can be used for mesh generation, interpolation and process modeling. In some cases, the generated meshes have improved quality and are generated more robustly.[5]

2.1.3 Challenges

While the principle behind RSA algorithms is simple, the execution of it's naive implementations will commonly lead to problems. As the figures are randomly generated and inserted to the area, the probability of rejection of said shape will grow with the saturation of the system. In simpler words, the more figures are already inserted, the more likely it is that a new one will overlap a pre-existing one, which leads to long execution times and uncertain end conditions. The goal of many approaches is to propose an algorithm that will be able to quickly generate shapes within given space, and have clear end conditions signifying a fully saturated state.

2.2 Thesis Goals

This thesis focuses on one particular subset of RSA, namely, the generation of polydisks in two dimensional space. It's goal was to create an algorithm that would take advantage of great degree of parallelization available through using the GPU. A pre-existing algorithm developed by Michał Cieśla enables quick sequential or thread-level parallel generation of polydisks in a 2D area. Such algorithm can be used to simulate the packings of particles on a surface; Which itself is usefull in physics and material research.[6]

[5] Mohamed S. Ebeida et al. "A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions". In: *Computer Graphics Forum* 31.2pt4 (), pp. 785–794. DOI: 10.1111/j.1467-8659.2012.03059.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.03059.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03059.x, p. 1.

[6] **ciesla**.

Chapter 3

Proposed Algorithm

3.1 Voxel-Based Algorithm

The basic algorithm used in multiple RSA applications involves the use of voxels. The voxels are defined as subspaces of the saturated area, which cover it entirely. After one or more shapes are inserted, the voxels may be removed if within them no more figures can be inserted. Developing an algorithm for "voxel rejection" is the main concern of those undertakings. Only within the non-removed voxels the figures can be generated. After inserting more figures, if the voxels are not rejected or the figures are rejected too commonly - the voxels are split into smaller voxels to be able to further reject them. If all voxels are removed, the system is saturated. An example of such algorithm, using the two dimensional circles as shapes is demonstrated in the figures 3.1, 3.2.

3.2 Sequential Algorithm

The algorithm developed by dr. hab. Cieśla is an expansion on the previusly described basic model. However, some changes have been made to accomodate the algorithm to a different shape, more efficient execution and a modification in the behavior of space.

3.2.1 Periodical Edge Conditions

The algorithm was supposed to operate using a space that, unlike previously shown model, would *loop* at the edges. This means, that if a shape is placed nearby the edge of given space, the shapes placed at the opposite edge of the space collide with it. They behave as if the opposite edge of the space would be a continuation of this edge. The periodical edge conditions serve to improve packing efficiency, and remove edge-related artifacts. In essence, the periodical edge conditions 'simulate' an infinite area, and thus may be helpful to model surfaces of far greater area than the provided space.[1]

[1] Michał Cieśla and Robert M Ziff. "Boundary conditions in random sequential adsorption". In: *Journal of Statistical Mechanics: Theory and Experiment* 2018.4 (2018), p. 043302. DOI: 10.1088/1742-5468/aab685. URL: https://doi.org/10.1088%2F1742-5468%2Faab685, p. 1.

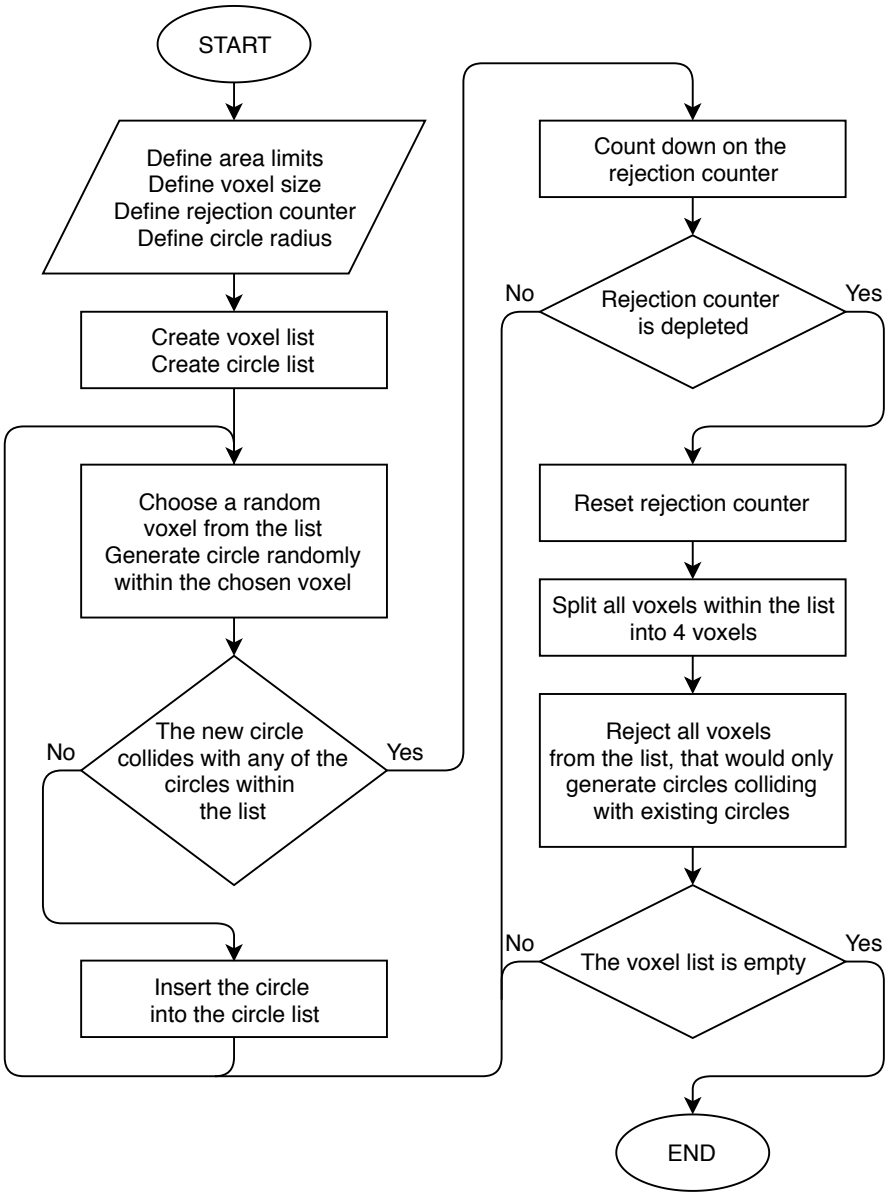
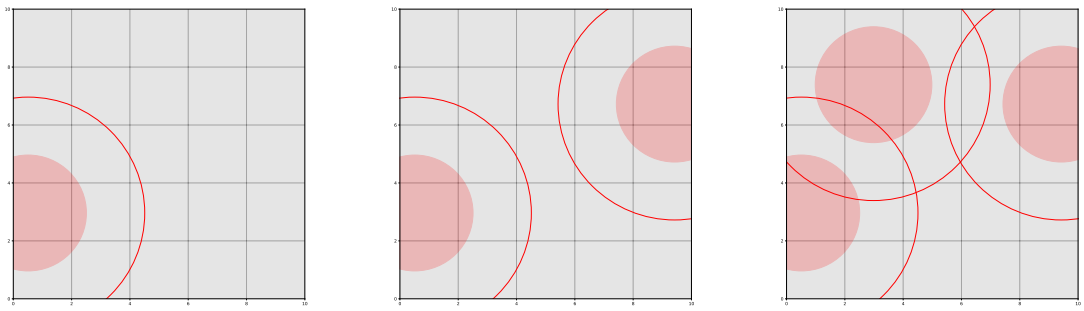


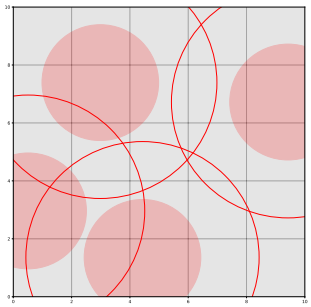
Figure 3.1: The flowchart of simple 2D RSA algorithm, using circles.
This algorithm can be expanded to remove voxels after every successful figure insertion, or to split voxels at a different condition.



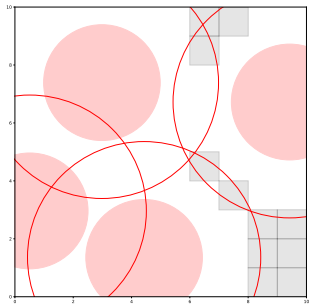
(a)

(b)

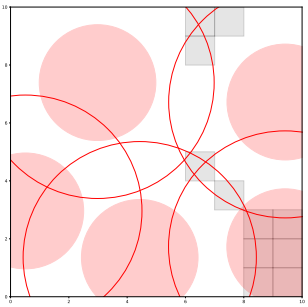
(c)



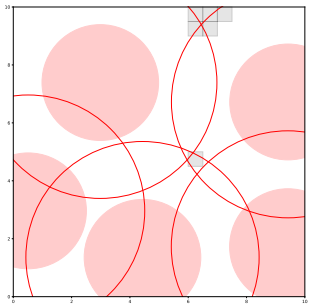
(d)



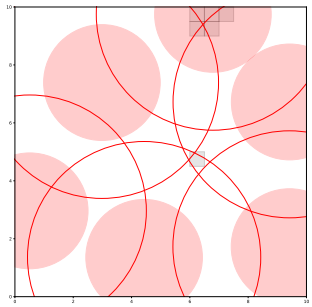
(e)



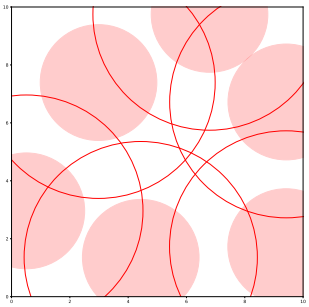
(f)



(g)



(h)



(i)

Figure 3.2: The figures illustrate a simple run of the described simple 2D RSA algorithm. The world size is 10.0 by 10.0, the inserted circles have the radius of 2.0. The initial voxel side has length of 2.0. After 10 failed attempts at inserting the circle, the voxels are subdivied and rejected.
The figures *a,b,c,d* show the insertion of the first four circles. The figure *d* illustrates the effect of the voxel subdivision and rejection. Following figures illustrate next iterations of circle insertions, voxel subdivisions and removals, and finally, at figure *i*, completely saturated system with no remaining voxels.
The voxels are rejected if they lie entirely within the double radius of any circle - in such case any circle generated within the voxel would collide with the evaluated circle. This is determined by examining if distance to all vertices of a voxel is less than the circle's doubled radius. This doubled radius is marked here as a solid red outline.

3.2.2 Adjacency Matrix

During the execution of the algorithm, there are two places where a function must be performed at two objects, many different times. Firs, when the shape is inserted, it is determined if it collides with any other existing shape. The other time, it is when the voxel is rejected - it must be checked against all shapes, until one that would cause it's rejection is found. It is possible to limit the executions of these function to a far lesser number; By dividing the world into a number of cells, with sides as long as the radius of the shape's minimum binding circle, it is only nescessary to check the collisions of the shapes that belong to the same, or neighboring cells (similarly with voxels). This limits the number of collision checks that need to be performed to a theoretical maximum, based on how many shapes can fit into a three-by-three cell neighborhood.
This solution can also be helpful with the periodical edge conditions problem. Having a quick access to the cell given it's coordinates enables the program to easily reference shapes from the opposite edge, by tapping into the target cell and translating the shapes stored by it.

3.2.3 Polydisk Collision

Since the algorithm uses a shape consisting of a list of disks with given radiuses and a relative position to an arbitrary center, it is relatively easy to check if two such shapes collide. It is enough to loop over both lists of shapes' circles absolute positions, checking if any two collide.

3.2.4 Voxel Rejection

The much more complex problem is the voxel rejection. The first problem is that since the shapes can have a different angle, this needs to be implemented in the voxels structure - the solution was to add a third dimension, representing the available angles in which the shapes may appear.

The algorithm proposed by Michał Cieśla covers the solution to this problem.
The voxel is rejected if no new figure can be inserted within it, at any position or angle, because there exists at least one shape near the voxel, that would overlap any new polydisk. It is enough that one circle of these pre existing polydisks would overlap any new circle from a virtually inserted shape. The voxel is represented by coordinates: (x, y, α)
Where x, y are it's carthesian position, while α represent's its angular position.
The voxel's spatial size is represented by δr and angular by $\delta\alpha$
The position of the virtually inserted disk is represented by:
$(x + f_x * \delta r, y + f_y * \delta r, \alpha + f_\alpha * \delta\alpha)$
where $f_x, f_{yf\alpha}$ are any numbers in range $(0, 1]$.
A disk, belonging to a pre-existing particle has radius: r and position: (x_0, y_0). The distance between the disk and the virtual disk is described as:
$d(f_x, f_\alpha) = d_x(f_x, f_\alpha) + d_y(f_y, f_\alpha)$
where:
$d_x(f_x, f_\alpha) = [x + f_x * \delta r + R_i * cos(\alpha_i + f_alpha * \delta\alpha) - x_0]^2$

$d_y(f_y, f_\alpha) = [x + f_y * \delta r + R_i * sin(\alpha_i + f_a lpha * \delta\alpha) - y_0$
where R_i,$_i$ are the length and angle of the vector pointing to i'th circle in a virtual molecule. If the maximal value of $d(f_x, f_\alpha)$ is smaller than $(R_i + r)^2$ the shapes will always collide.

TODO UNFINSHED - finish the equations

3.3 Parallel Application

Having the available algorithm for sequential execution, the main challenge consisted of modifying it to enable a highly parallel execution using the GPU. The modified algorithm executes some parts of the problem sequentially, as it was not possible to parallelize it entirely.

3.3.1 Block Diagram

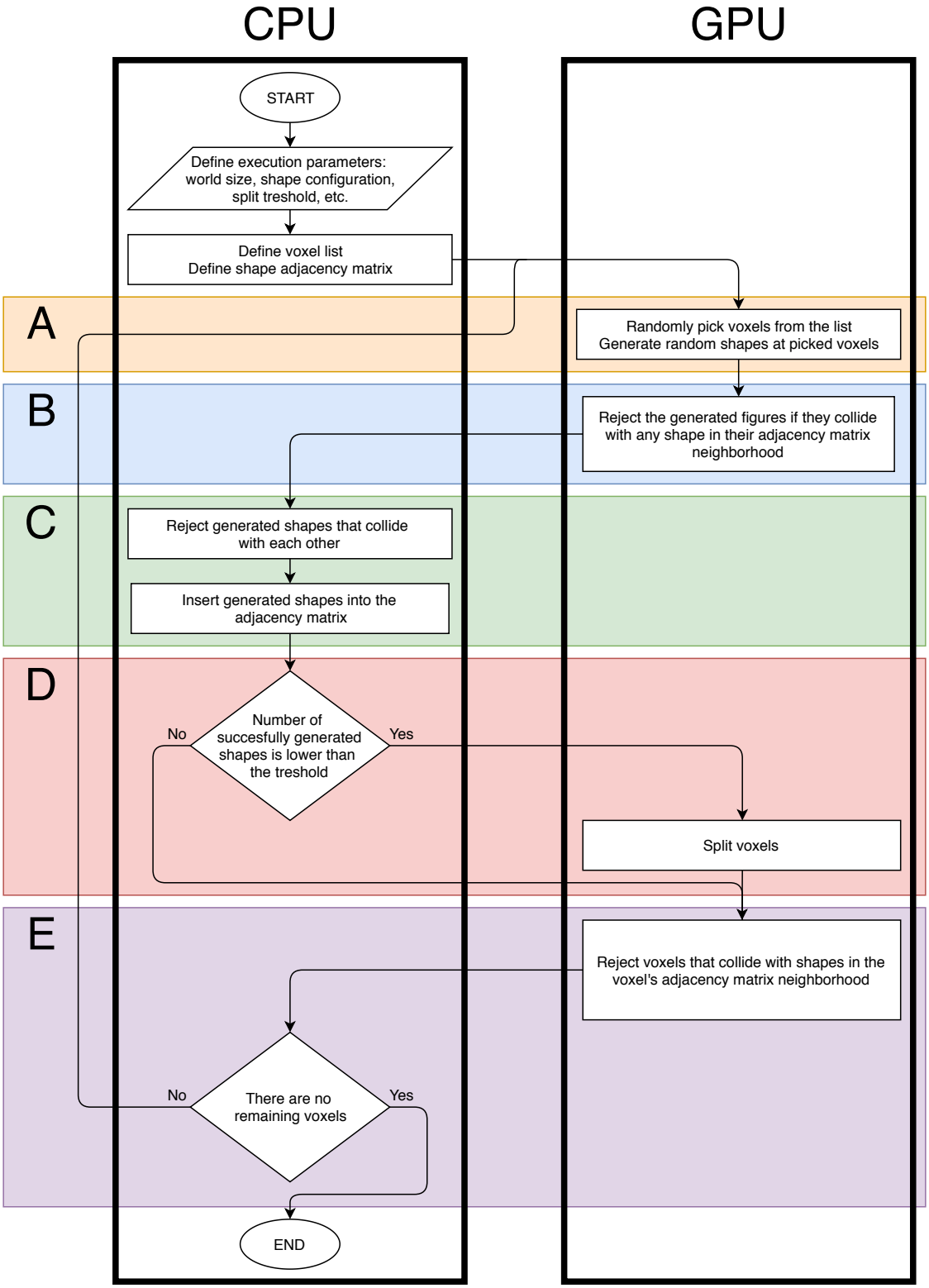


Figure 3.3: This flowchart represents the proposed parallel algorithm.
Blocks on the right, within the 'GPU' column are executed in parallel, while those on the left are executed sequentially

3.4 Implementation

The implementation of the algorithm involved several challenges. It necessitated the use of programming tools capable of utilising the GPU processing. Also, a number of smaller auxillary tasks needed to be implemented as well, such as visualisation, optimisation of the figures, etc.

3.4.1 PyCuda

The Python programming language was chosen to implement these tasks. The ease of writing code, as well as the vast choice of available libraries were deemed crucial in implementing the algorithm. While, as an interpreted language, Python may lack some execution speed, as compared to C or C++, it can partially bridge that gap using the dedicated mathematical libraries such as NumPy. Also, the main acceleration would come from using the parallel approach by utilising the GPU.[2]
However, the Python interpreter has no built-in implementation of GPU parallelism, and must rely on external libraries to perform these operations. The library chosen for this task was PyCuda. This library enables the use of Nvidia CUDA's capabilities, simultaneously wrapping parts of it's execution into Python functions. While, it only provides functionalities for initialising the CUDA execution and some basic array operations in Python, it still greately simplifies the code as compared to pure C/C++ implementation. Any more complex parallel code must be written in C. As such, the code for the algorithm was divided into two parts, where all parallel operations were written in C using CUDA library, while all sequential implementation used Python with PyCuda to initialise and communicate with the GPU-parallel part.[3]

3.4.2 Step-by-step examination

Initialisation

The starting parameters of the algorithm include:

- **The shape configuration** (circle positions and radiuses relative to the shape origin).
- **Number of added shapes** per iteration.
- Adjacency matrix **cell size** (must be greater than the shape size).
- **World size** (in number of adjacency matrix cells).
- **Voxel split treshold** - the proportion of added shapes that were rejected that would trigger the voxel split.

[2] Travis Oliphant. *Numpy website*. 2001–. URL: `https://www.numpy.org/` (visited on 08/10/2019), p. 1.

[3] Andreas Klöckner et al. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: `10.1016/j.parco.2011.09.001`, p. 1.

As these parameters are set, a number of additional functions are executed. The shape area is being calculated, and the shape itself is being optimised. The latter involves re-calculating the origin of the shape, so that it's bounding circle is the smallest. These functions are auxillary to the main algorithm.
During the initialisation, there are created the data structures used throught the further execution. These involve:

- **The shape array**: An array with shape: 3xN, where N is the number of shapes in the packing. Every row has the shape's coordinates (x,y) and it's angle.
- **The neighborhood matrix**: The 3D array with shape: CXxCYxSPN where CX and CY are the world dimensions in numbers of cells, while SPN is the theoretical maximum of figures per 3x3 cells (overestimated approximate). Such groupings of cells are here referred to as the "neighborhoods", and they contains the indexes of shapes that may collide with a shape in a given adjacency matrix cell. This whole data structure serves as the adjacency matrix.
- **The voxel array**: Similar to the shape array, it contains the spatial and angular positions of the existing voxels.

Additionally, there are set variables that are subject to change: the **current voxel spatial size** and the **current voxel angular size**. Finally, the CUDA parallel execution is initialised, and it's pseudorandom generator is seeded with the current time.

Part A: Shape generation

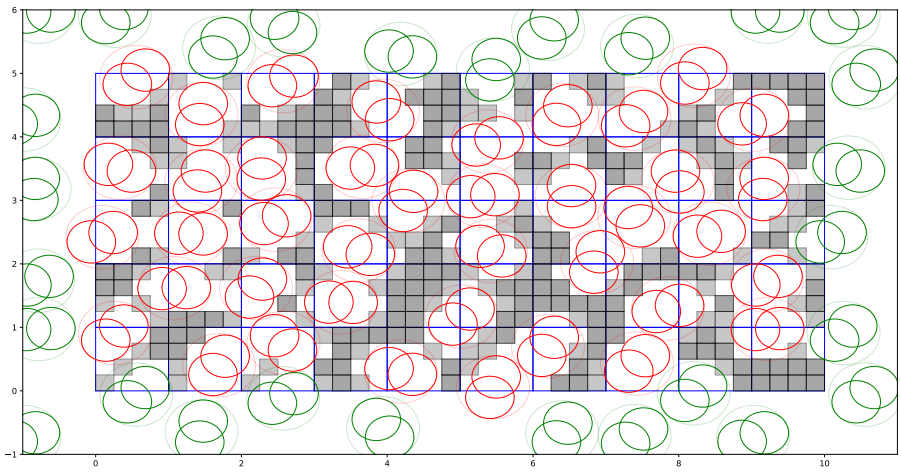


Figure 3.4: The packing after several iterations, before generating a new set of shapes.
The existing shapes are marked in red. The shapes appearing at the edges, as a part of periodical boundary conditons are green. The faint circles around the shapes are their bounding circles. The voxels are represented by gray squares, while the blue, empty squares are the adjacency matrix cells.

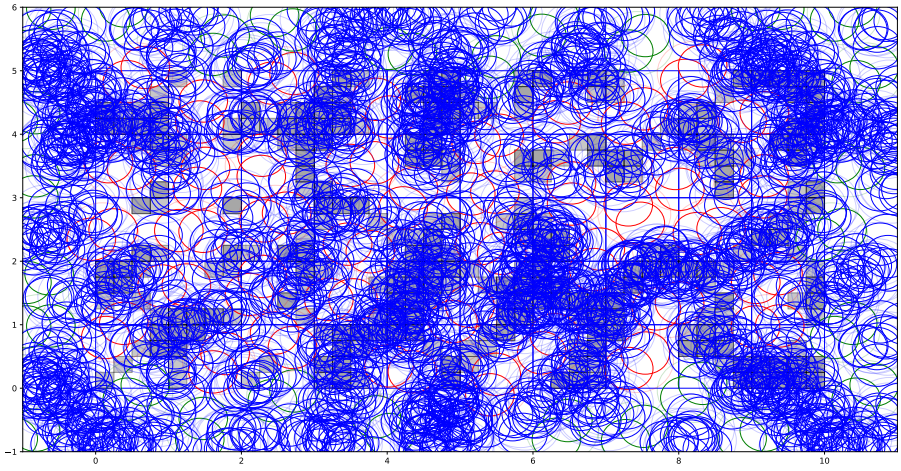


Figure 3.5: The packing after shape generation. The new shapes (blue) were generated randomly at existing voxels.

This part is executed entirely in parallel.
In this part, an additional array, **the added shape array** is created. It will be later merged with **the shape array**.
A number of CUDA threads is created, equal to **the number of added shapes**. Every thread randomly picks a voxel from the **the voxel array**. A shape is generated with it's poisition and angle within the voxel. The resulting shapes are put into the **the added shape array**.

Part B: Shape rejection against pre-existing shapes

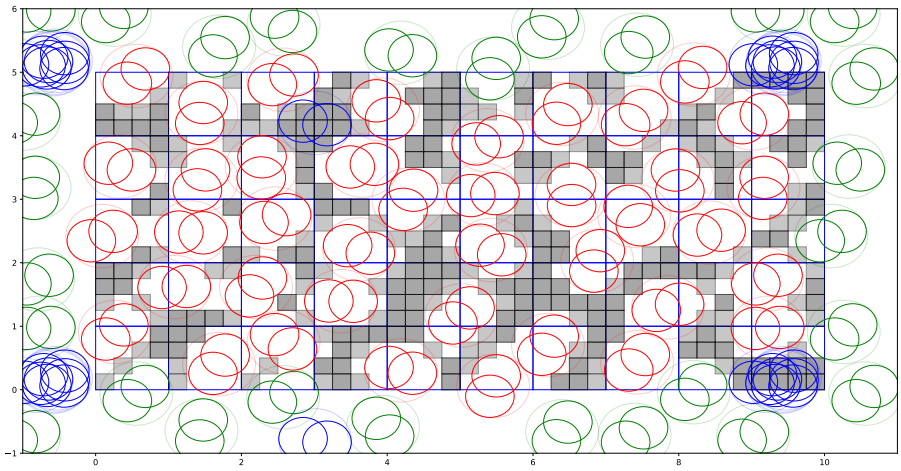


Figure 3.6: Packing after rejection of shapes against the pre-existing ones. Only small groupings of new shapes remain.

This part is executed entirely in parallel.
The same GPU threads are used as in the last part, each corresponding to a generated shape. The shapes' adjacency matrix cell positions are calculated. Basing on this, the collision is detected between the shape in the thread and all of the already existing shapes in it's potential neighborhood. If the shape collides with any of the pre-existing shapes, it is rejected, which is marked by setting it's coordinates in **the added shape array** to [-1.0,-1.0,-1.0]

Part C: Shape rejection against other new shapes

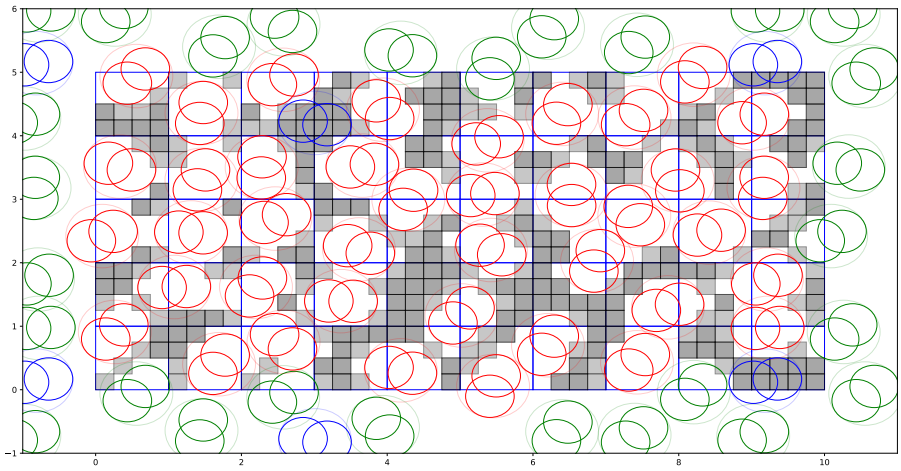


Figure 3.7: Packing after rejecting the remaining new shapes against each other.

This part is executed sequentially.
The added shape array is returned from the CUDA execution, and "squashed" to remove all rejected shapes. Since the shapes do not collide against any pre-existing ones, only the collisions within the list of new shapes need to be calculated. A temporary, empty neighborhood matrix is created. Shapes are added to this matrix sequentially, and rejected if they collide with one existing there. When all shapes are added, the shapes are merged into the actual **neighborhood matrix**, and **the added shape array** is concatentated to **the shape array**.

Part D: Voxel splitting

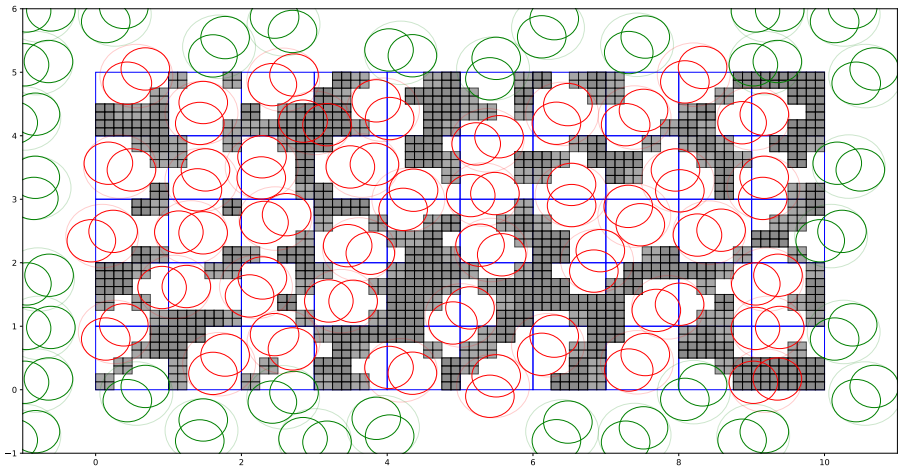


Figure 3.8: Packing after voxel splitting. Every old voxel has been split into eight new ones.

This part is executed entirely in parallel.
If the proportion of figures succesfully added to the packing in the previous part is lower than **the voxel split treshold**, this part is executed.
The number of CUDA threads is created, for every currently existing voxel. A temporary new array, **the new voxel array** is created, with eight times as many elements as **the voxel array**. In every thread, the voxel is being split into 8 new ones, each with halved dimensions, and given new positions, so that they cover the entirety of their "predecessor". These voxels are placed in **the new voxel array** which afterwards replaces **the voxel array**. The values of **the voxel spatial size** and **the voxel angular size** are halved.

Part E: Voxel rejection

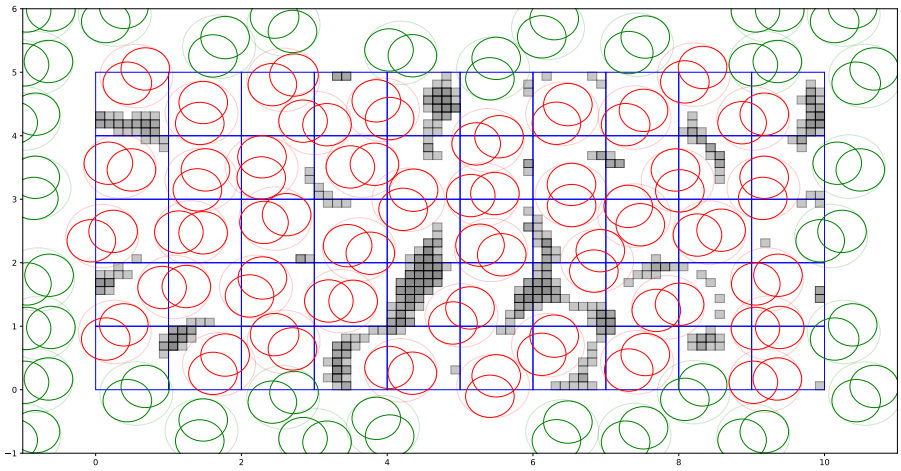


Figure 3.9: Packing after voxel rejection. Voxels in which no new figure could have been insterted, were removed.

This part is executed partially in parallel, and partially sequentially.
The number of CUDA threads is created, for every voxel in **the voxel array**. For every voxel, the Cieśla's voxel rejection algorithm is executed against every figure in the voxel's neighborhood. If any figure causes the voxel to be rejected, it is marked as such on **the voxel array**.
After the parallel part is executed, the voxel array is returned to the CPU memory. It is "squashed" to remove the rejected voxels, and if the number of remaining voxels is equal to 0 - the algorithm stops.

3.4.3 Shape optimisation

During the initialisation, the provided shape configuration is being optimised. This task is performed by a separate module. The shape optimisation entails translating the shape's circles' positions so that the origin lies in the center of the polydisk's minimal bounding circle. This can lead to smaller possible adjacency matrix cell sizes, smaller neighborhoods, and thus more efficient figure and voxel rejection.
The implemented algorithm will perform ideal optimisation only for some cases, while for others, it will create an oversetimated approximation of the bounding circle. The algorithm creates a bounding circle for the set of two circles, containing points laying furthest from each other. If there exists a circle with a point outside the initial bounding circle, the latter is expanded to cover the outlying circle. The coordinates of the circles are then translated, so that the center of the bounding cirlce is the new origin point.

3.4.4 Other

The shape area is calculated using the Shapely python geometry library.[4]
The images depicting the RSA iterations were created using the Matplotlib python plotting library.[5]

TODO: pictures for shape optimisation, references to shapely, matplotlib

3.4.5 Result management

In order to perform analisys of statistical data, some data concerning the algorithm run, besides the output list of figure positions, can be saved. They are saved to the "results.json" file, and include:

- Number of figures and voxels
- Proportion of the world's area covered by the figures
- Time taken to perform each part (A-D) of the algorithm

These data are saved per every iteration, as well as a summary, with the total execution time.

3.4.6 Summary

TODO unfinished, do in near future

[4]Sean Gillies et al. *Shapely: manipulation and analysis of geometric objects*. toblerity.org. 2007–. URL: `https://github.com/Toblerity/Shapely` (visited on 08/10/2019), p. 1.

[5]J. D. Hunter. *Matplotlib website*. 2019. URL: `https://matplotlib.org/index.html` (visited on 08/10/2019), p. 1.

Chapter 4

Results Examination

4.1 Performance Evaluation

4.1.1 Parameter Influence over Performance

4.1.2 Shape Influence over Performance

TODO unfinished, do later

Bibliography

Cieśla, Michał and Robert M Ziff. "Boundary conditions in random sequential adsorption". In: *Journal of Statistical Mechanics: Theory and Experiment* 2018.4 (2018), p. 043302. DOI: 10.1088/1742-5468/aab685. URL: https://doi.org/10.1088%2F1742-5468%2Faab685.

Ebeida, Mohamed S. et al. "A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions". In: *Computer Graphics Forum* 31.2pt4 (), pp. 785–794. DOI: 10.1111/j.1467-8659.2012.03059.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.03059.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03059.x.

Feder, Jens. "Random sequential adsorption". In: *Journal of Theoretical Biology* 87.2 (1980), pp. 237 –254. ISSN: 0022-5193. DOI: https://doi.org/10.1016/0022-5193(80)90358-6. URL: http://www.sciencedirect.com/science/article/pii/0022519380903586.

Gillies, Sean et al. *Shapely: manipulation and analysis of geometric objects*. toblerity.org. 2007–. URL: https://github.com/Toblerity/Shapely (visited on 08/10/2019).

Hunter, J. D. *Matplotlib website*. 2019. URL: https://matplotlib.org/index.html (visited on 08/10/2019).

Klöckner, Andreas et al. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

Oliphant, Travis. *Numpy website*. 2001–. URL: https://www.numpy.org/ (visited on 08/10/2019).

Zhang, G. "Precise algorithm to generate random sequential adsorption of hard polygons at saturation". In: *Physical Review E* 97 (Mar. 2018). DOI: 10.1103/PhysRevE.97.043311.